Model(s) for predicting average house pricing through time series forecasting using regression algorithms

---

Author: Umar Sultan
Student ID: 33615754

Supervisor: Nikolay Nikolaev

Goldsmiths University of London
BSc Computer Science
May 5th, 2023

Final Report

(Hand in 4)

Contents

1. Abstract
2. Introduction
3. Background Research (Literature Review)
4. Methods
   a. Development Methods
   b. Requirements
   c. System Design/Prototyping
   d. Entire System
   e. Development Process
   f. Ethical Considerations
   g. Methodology *(Start of directly discussing project)*
5. Data Breakdown
   a. Source
   b. Breakdown
6. Data Preprocessing
   a. Extraction
   b. Treatment
   c. Scaling
   d. Transformation/Data-split
7. Linear Regression
8. Polynomial Regression
9. MLP
10. Results
11. Discussions
12. Limitations and Future Works
13. Conclusions
14. Bibliography

You can find the implementation of my project in the form of an executed Jupyter notebook on my GitHub here, https://github.com/LiptonTeaa/Final-Project.git

Abstract

In this paper I forecast house prices based. I am utilising previous listing house prices of houses in specific boroughs in London to forecast future and interim prices of houses. This is something that has not seen much interest on the consumer level with most of the research on this topic being held and sae guarded by larger corporations. This has left a gap in the consumer market that has yet to really be filled. I hope my work can provide an alternative to relying on appraisers and real estate agents. To this end I am forecasting house prices by creating time series using previous pricing over the years and using regression algorithms to forecast not only future prices but missing values in registries from the past. At the end of this paper, we will find out how successful this approach has been to taking this problem in contrast to different methods of prediction.

Introduction

We will be considering the housing market in the UK specifically for this project, however I am fairly confident that a lot of these observations can translate just as well to other markets. Before we dissect the market, I would like to briefly touch on the inspiration that led me to this specific project.

I was in the not so distant past from when I am typing this, roped into helping family buy a house in London. It shed light on the many hurdles and complications buyers have to face, many of which I was oblivious too. The delta of prices between houses situated in the same areas, as well as adjacent areas was staggering. Keep in mind that most of these houses were seemingly similarly built and sized. It was a truly frustrating experience between over ambitious agents and limited information availability, but those are irrelevant details, save for context. It is worth noting that despite my first hand experience being mostly constraint to a certain parts of London, this experience translates the same for most of everyone from whomever I've spoken with.

With this, we can neatly transition into the state of the industry I experienced. For the many houses came and went, for each I was stuck checking land registries for their financial history, when they were available. Registries were updated once a month and there were many cases where the registries just hadn't been updated due to slow processing (relators and employees not filing paperwork). I was stuck relying on websites like 'Zoopla' and 'Rightmove' [1] which kept easier to access but often much more inconsistent registries of dates and prices house's were sold at. There just wasn't a hassle free solution to accurately gauge prices.

House pricing is vital information that every buyer should have a right to, be it past or present. While there is nothing I can do about completing the incomplete registries for properties, I can offer a alternate solution. This in theory should be able to fill in these pesky gaps that exist in registries, provided a sufficient amount of data has been previously recorded in the form of prices from which to draw a conclusion. With my project I should be able to fill this gap by predicting the older missing prices using previous pricing data. Not only that, but I should be able to predict current pricing values for houses using previous listing prices as well.

This is not a novel idea in and of itself. It is already an extensively researched topic. Large conglomerates like banks and corporations have for many years now invested heavily in personal, techniques and research into how to best predict prices. Not only for property but for many different avenues such as stocks and goods. They use this information to predict trends or crashes. They utilise this information to profit on their own terms. How exactly they go about doing this is subject to ethical considerations, something we will touch on later in the report. Most of what I've outlined for corporations is not at all true on the consumer side. There are no such widespread organised efforts working for consumers to utilise or benefit from. There are government agencies that collect this kind of information to small non profits that also work to produce similar information, but their purpose for doing is more towards the interest of uncovering fraud or exploitation.What I am proposing is something that can be used on a consumer level, for or by the consumers themselves.

The aim of this project is to build a model that will be able to predict housing prices by using pervious house pricing for that area. I will be

making a software implementation in the form of a model(s) that will be able to predict the price of a house if it is given a specific set of data that is relevant to the house. With all preconditions met, it should produce an accurate output within a small margin of error. My hope is that what is produced is something that would allow potential house buyers to have a tool that would allow them to better estimate the price of houses they are looking to purchase beyond just what the house is currently listed at.

Background Research (Literature Review)

There is a need for a solution for potential buyers such as I was at the time. Something that would at the very least provide a buyer with a baseline of what the property should really be worth.

As I alluded to before, there exist many commercial solutions to this problem that cater primarily to larger corporations that own a multitude of property that they can't be bothered to or simply don't possess the resources to survey and ascertain correct valuations of. They rely heavily on software solutions to fill that gap for them. An example of these property management software providers are 'Realpage' and 'AppFolio'. The tools they provide for their determinations are blocked behind paywalls and are not open to public scrutiny. This also causes ethics concerns, as we have seen with Realpage who is currently entangled in a lawsuit [2] for artificially raising rent prices in particular areas.

A solution for the public to use that is transparent in its methods would prove to be a powerful tool for consumers to make use of that would not only smoothen the acquisition process of a new home but could serve to be a deterrent against commercial property management companies and abuse their obscure systems. There do exist online resources that will provide you with price estimates, they will do so using information from a combination of sources that are known only to them. Meaning the information could be months or even years old, rendering it unable to make it an accurate prediction. One such service based in America for the American market is called Zillow [3]. They provide a service called zestimate or a zestimate market analysis which requires the use of one of their employees. As mentioned, they boast of their accuracy but their methods for this accuracy are shrouded in mystery.

In an ideal situation, your real estate agent should have done their own research to procure all the previous listing prices of a property on hand, but it was rarely so in my personal experience. My built solution would require you to have this information on hand, cumbersome as it is, its still free compared to hiring an appraiser.

It is worth pointing out that the solution I put forward can only be trained on data from pre-defined areas (boroughs in my case). This is a problem that plagues all solutions I have seen. Data from a particular markets tailors the results of what you achieve to that market the data belongs to. This is unavoidable as every market, even from a state to state level shows a great deal of variation in what factors or features most effect the price of those houses. This makes most solutions localised solutions and difficult to use as a blanket solution for all markets. Trying to feed a model with such broad data from different areas yields inconsistent accuracy compared to a local approach.

Kaggle [4], a competition platform where companies post looking for specific modelling solutions to problems they might have, has been the largest repository for similar projects to mine that I have come across. These companies post the data required to model the solution in their competition pages. There also exists a large repository of solutions made by other people utilising a myriad of techniques to predict pricing. I uncovered many housing dataset here made for public use or to be used in education purposes. To name two popular ones are the California and Boston housing market sets. These solutions are however, also localised solutions. Most by the nature of such modelling, can make no attempt branch out for a more generalised solution, nor can they take into account any factors other than what was provided in the data set.

With Kaggle, it is up to the competitors discretion to make their solutions to a competition public knowledge. It is a similar case for many research articles/papers that I have looked into. There is not a great deal of documentation available for my specific topic as it is a relatively new solution to the housing problem (deep learning simply wasn't advanced enough to sufficiently tackle these problems even 20 years ago) and what may exist in the corporate world, are close guarded company secrets. Despite this, a handful based on the exact same topic that we are talking about here with solutions rooted in deep learning do exist. Two such articles focus on making a prediction using

machine learning and neural networks [5], and, machine learning with python [6]. These two in specific discuss solutions using tools that I intended to utilise. They talk in detail about the methods they utilised to build their models, like regression techniques (linear regression, forest regression, etc) and the factors that have a bearing on the data, and methods used. They also provide numerical evidence as to the competency of their developed models. However, they fall short of providing the code for their developed models as open source software to further develop or better understand, neither do they make proper mention of the specific tools they made use of to develop these models they so endearingly discuss.

The regression techniques they make use of utilise house features like number of rooms and property size, however. While this was a possible approach I could take, it was not the approach I chose to follow. As we know, prices are extremely volatile and difficult to predict. As per my supervisors instruction, I began to look into forecasting which is more commonly used for price prediction. Not so much property but it is heavily used in predicting stocks and bonds. In addition to this, it was also what my supervisor was familiar with. My search did return many papers in regards to price prediction using forecasting [7]. The scale and complexity was far greater than what I was capable of. He also graciously provided me with slides from the neural networks course he taught to better understand the material online as well as containing simpler forecasting models for me understand and implement that I will be using in conjunction with regression algorithms. All in all, this topic will allow me to demonstrate the technical knowledge I have amassed in my degree.

Methods

Development Methods

I adopted an action research approach for a solution rooted in machine learning. My solution consists of different models of regression algorithms. Regression algorithms are most commonly used for a predictive analysis as they describe a relation between one or more independent variables and a target variable. ‘The goal of regression is to predict the value of one or more continuous target variables t given the value of a D-dimensional vector x of input variables’ [8, p. 137]. While there exist a number of regression algorithms, my primary focus

was on linear models, in particular an autoregression as well neural networks in the form of multi layer perceptrons (MLPs). To better understand these nuances of this approach, I used the machine learning book by Bishop as recommended by my supervisor[8].

From what I researched, let us take a look at linear regression first. ‘The goal of linear regression is to predict the value of t (target variable) for a new value of x (input variable)’ [8, p. 137], this is the basis of our very model. It is also worth remembering that most of everything else is built upon this basic concept. Linear regression serves mainly to provide a continuous output which we will utilise in forecasting. Such sequential learning methods are appropriate for real time applications in which data is arriving in a continuous stream.

‘Evaluation of the evidence function’ [8, p. 166-168], is a sub section in the book that discusses the developing and testing multiple linear and non linear models with a set data set and comparing their outcomes. For my linear model, I did plan to feed it two different, yet interconnected sets of data to compare and evaluate their outcomes.

The non linear neural networks I am primarily concerned with are ‘multilayer perceptrons’ [8] that comprise of multiple layers of regression models. These networks are primarily feed forward networks but make use of back propagation an (error function) as a method of increasing the accuracy of the function. For network training we can summarise, ‘there is a natural choice of both output unit activation function and matching error function, according to the type of problem being solved. For regression we use linear outputs and a sum of squares error’ [8, p. 236]. This eludes to the different approaches and functions we can make use of to approach our model and the loss functions that will provide us with a measurable quantity of success our network is managing to achieve. The metric I chose as advised was mean square error (MSE), to use across all my models as an equal basis of comparison.

My basic job was to feed my models, pricing data from a specific area, that it will train on. Our linear model will find regression coefficients (weights) and biases that it will apply to the time series to forecast (predict). The MLP will process this data by applying weights and biases in the hidden layers to provide us with an output that should in theory be the predicted price of the house. To implement all this, I worked through multiple iterations on which I consulted my supervisor to trouble shoot problems. Basically trial and error until I arrived at fully

functioning models. My courses on ML (machine learning) greatly bolstered my skills and allowed me to learn python. Python being the most versatile language for ML applications, as well as all the tools and resources I can utilise to construct the project. I did plan to log my progress using GitHub but I chose to forego it simply because of the number of iterations I ended up up going through, it became too difficult to log every alteration or the number of alterations I made.

Requirements

Here we will discuss in further detail exactly what the software I intend to develop is going to do or demonstrate.

User requirement, ‘Specifies what the user expects the software to do’ [9].The end user will be limited to someone who understands the inherent technologies that facilitated the development of this software as it will is run directly on those technologies. A laymen would have no knowledge of how to navigate something like an integrated development environment (IDE), much less how to execute what is in front of them.

As it stands however, a user should be able to download the file and launch it in the correct environment, upload their own data file or use the one provided with the application, execute the file in the opened environment and receive an output.

Requirement specification, ’Is a description of a software system to be developed’ [10]. The software must exist in a downloadable format, one such that it can be downloaded by anyone who is using a machine that is capable of running the necessary tools. This file should be able to execute (contain executable code) in the correct environment on the right data set and return a tangible output in numerical terms or a as a visual representation. Any data set should work provided it contains the correct attributes and is correctly formatted before use.

This file, should contain the project and a data set for the project to operate on incase the user does not, or cannot provide their own data to operate on.

Functional specification, ’Specifies the functions that a system or a component must perform. The documentation typically describes what is needed by the system user as well as requested properties of inputs and outputs’ [11]. Anyone using a machine that is running a 64 bit

operating system should be able to execute it with an appropriate IDE (Jupyter in my case). I do not expect the tools I used for development to cause any interference on other platforms as the tools I used operate largely similar across platforms. The IDE would need to be able to support ".ipynb" files since it is the format I used for my project. The software will comprise of individual cells of executable code.

It largely comprises of regression algorithms and an MLP. The users system should be able to handle the load created particularly by the MLP. It is very taking to run on a cpu alone. A graphics card (GPU) could accelerate the data processing. The user must also have the correct packages I used for development in python installed, NumPy and Pandas to name a few. The imports in the beginning of my file state everything I used. An online environment like Google Colab provides all the packages and drivers pre-installed in addition to GPU acceleration. The correct data set for the file to operate on. The software must be executable on the provided data set or any appropriately formatted data and return a result in a numerical form or as series of outputs on a graph.

System Design/Prototyping

Lets talk about the prototyping I did before I actually set about building my project.The project I developed makes use of a myriad of techniques to arrive at an appropriate range of results. What techniques I will be utilising in my software and how those techniques function individually as well as briefly touch on how they will tie into the system.

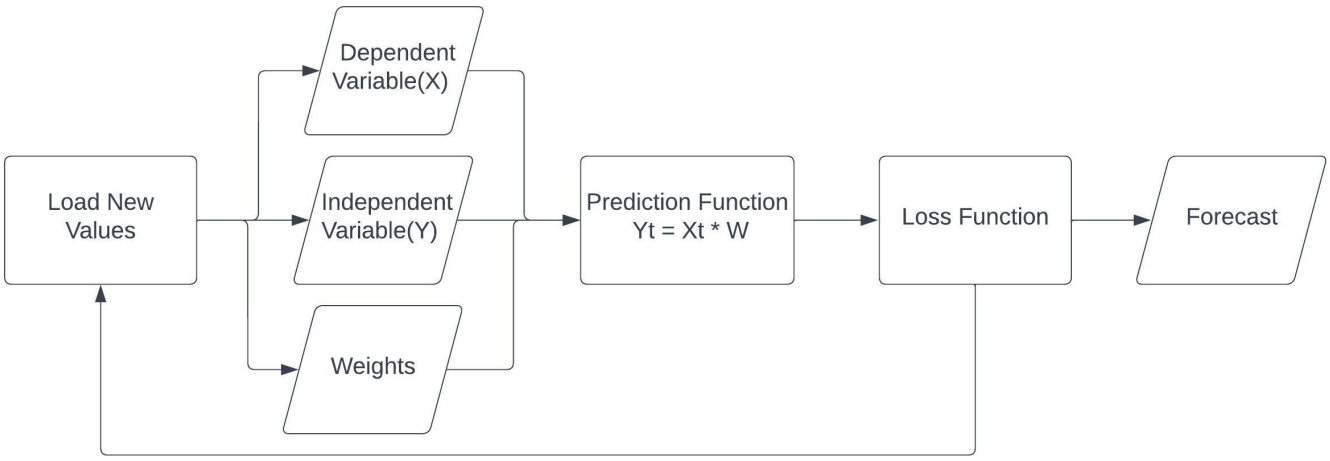
I am going to be representing these as informal prototypes in the form data flow diagrams and pseudocode to help us better understand, and serve as component prototypes for the software.

To start we will scale our and transform our data in preprocessing before we can put it through any forecast function. We will treat our data for any missing values, transform any non-numerical values to the correct numerical form or drop them. To scale our data, we will normalise and standardise our data. Normalisation will rescale the data to be between 1 and 0 and standardisation will scale every observation to have a mean of of 0 and standard deviation (SD) of 1. After which we can transform or data into a time series of lagged inputs. We will specify the lag and intervals at our discretion. It is worth noting that the

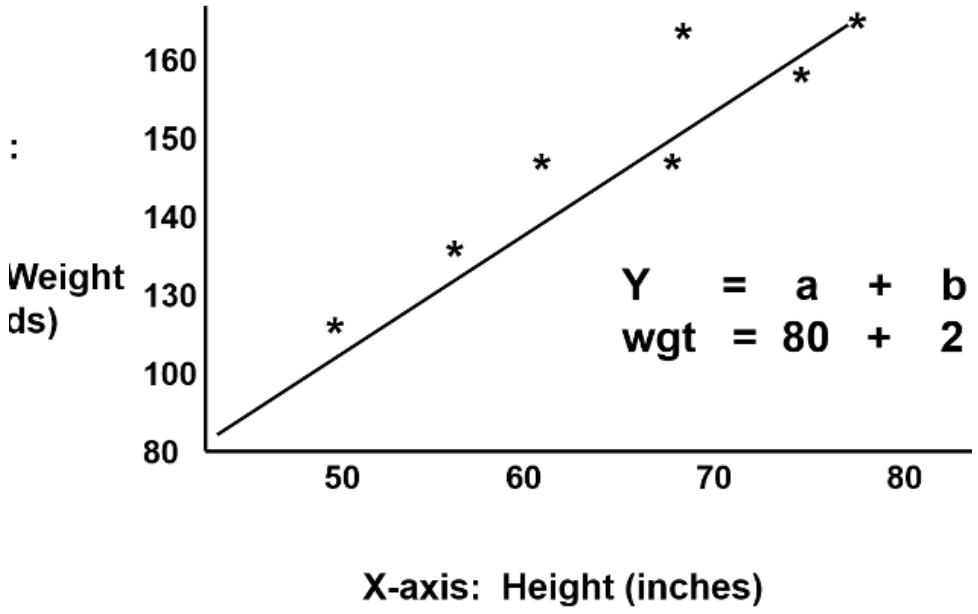
data has to be chronologically ordered for a time series to successfully forecast, this order must be maintained through all transformations.

Taking a very basic look at regression. This will also serve as a great example of how our software works as even our MLP will follow the some of the same principles with added complexity in each step. We transform our data into a time series and splice our data into training and testing data. We find a number of regression coefficients according to the time series and perform our prediction functions on each of those collections of observations (values). Once the prediction has been made, it will compare the predicted value against the real value for the observation it is trying to forecast. This job belongs to the loss function who finds the difference between the actual value and the forecasted value. It can use this information to tweak the weights and biases that are attached to the input observations to manipulate them. Let's now look at the models I used.

Linear regression,
Looking at a simple interpretation of linear regression. It correlates one independent variable to a dependent variable by applying weights and biases. The version of linear regression we're using is auto regression (AR). It is specifically built to be a linear superposition of past, lagged inputs, which we will have courtesy of our datas transformation to a time series. The formula it uses produces a best fit line between the forecasted and actual values when gathered and plotted. It is worth noting that while the graphical representation below produces a perfectly straight line of best fit for demonstration purposes, the real implementation will follow the curves of the actual values.



Linear Regression



Linear reg plot demonstration [12]

Pseudocode

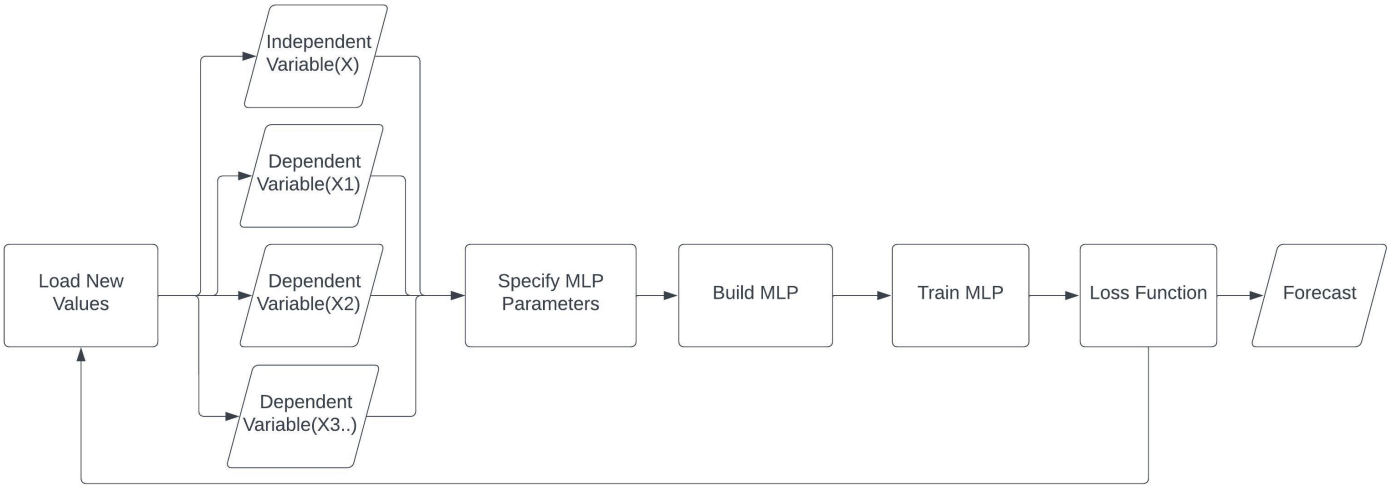
Require: Training as time series D, design matrix X from time series, time series vectors x, weights W, lambda (small value) λ, identity matrix I, train vectors Y, forecasts y

1: Initialise weights using OLS (ordinary least squares)
2: xb ← (X.transpose).X
3: xb ← xb + (λ.I)
4: xb ← inverse(xb)
5: xb ← xb.(X.transpose)
6: W ← xb.Y
7: Forecast using AR
8: For each vector x in D do
9: $y_t \leftarrow \sum_{n=i}^{P} W_i x_i + E_t$
10: End for
11: Return $y_1, y_2, \ldots, y_t$

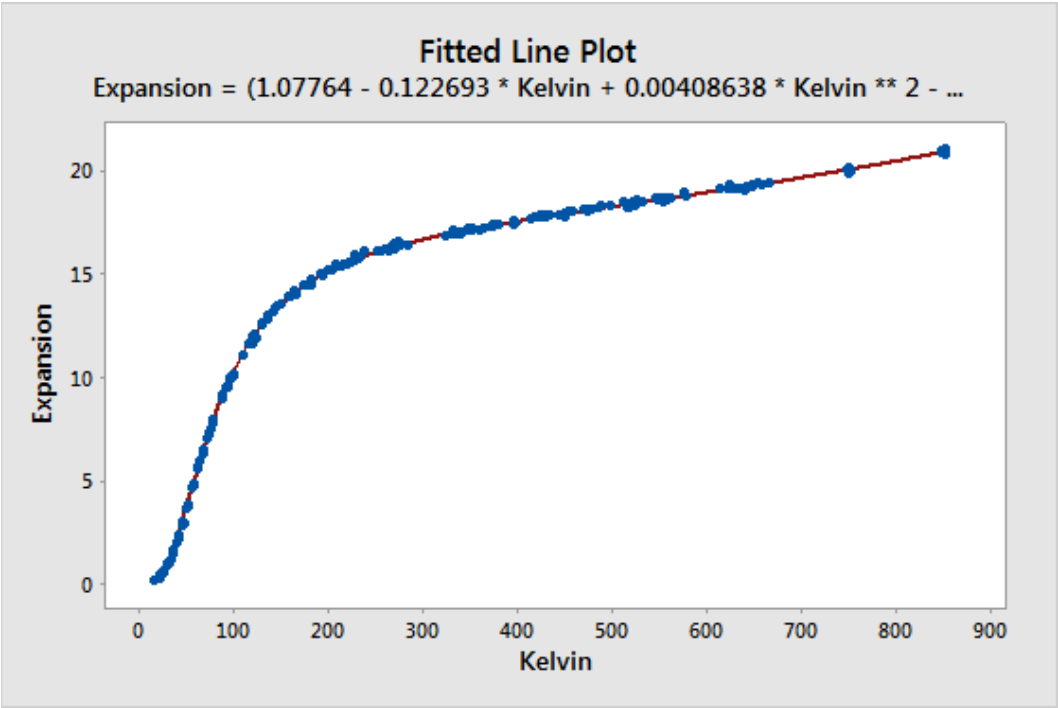
I also have another linear model. For this model however, I feed different data. I take our initial time series and use it to construct a polynomial version of that very data. By doing so I can further check the efficacy of our model.

Multi Layer Perceptron,
This can be taken to build on our previous model. It will also serve as a non linear model to contrast AR. While non linear regression and non linear neural networks may seem to be identical at a glance, there are subtle differences that add far more nuance to a neural network that a regression model cannot hope to achieve. It cannot just be referred to as just a parametric nonlinear regression model. They allow fitting models of very high complexity. They can cater to massive data sets that you cannot fit to a simple linear model. For some tasks, they're impressively accurate, much better than many other statistical learning models [12]. This makes it a far more versatile approach to prediction. MLP's, what we're specifically looking at are a less complex than Deep Neural Networks in the sense that they have far less layers.

Much like our linear models, it attaches weights and biases to input data to forecast. In this case however, it has multiple nodes in a layer. Each node has a weight attached to it and a single bias for the whole layer on which it computes. How much each node contributes to the end result is dictated by an activation function. Expanding on this, our MLP takes as input, multiple independent variables (our input vector from the design matrix), each of whom is assigned a weight. Each input is multiplied by its weight, added to each other and lastly the sum is multiplied by the bias. This true for every node in the network and happens in every layer individually (((X2 x W2) + (X3 x W3)) x B). Weights determine the influence each node has on the network and biases feed into the activation function, helping it decide which neuron's need to be fired next. All this to to predict the dependent variable. A loss function give us an honest estimate of your MLPs performance. This loss function finds the difference between forecast and actual values, then uses this information to tweak the weights and biases that are attached to each node to improve loss. It will continue doing this for every observation, tweaking the parameters (weights and bias) until the loss function settles to within an acceptable error range. Or when the parameters reach a point asymptotically and are no longer being altered. The curve here being the gradient for our loss function. It also employs a number of techniques to make sure we aren't stuck in a local or maximum minima, thus preventing us from achieving the lowest possible loss. The flowchart below demonstrates this.



MLP Flow



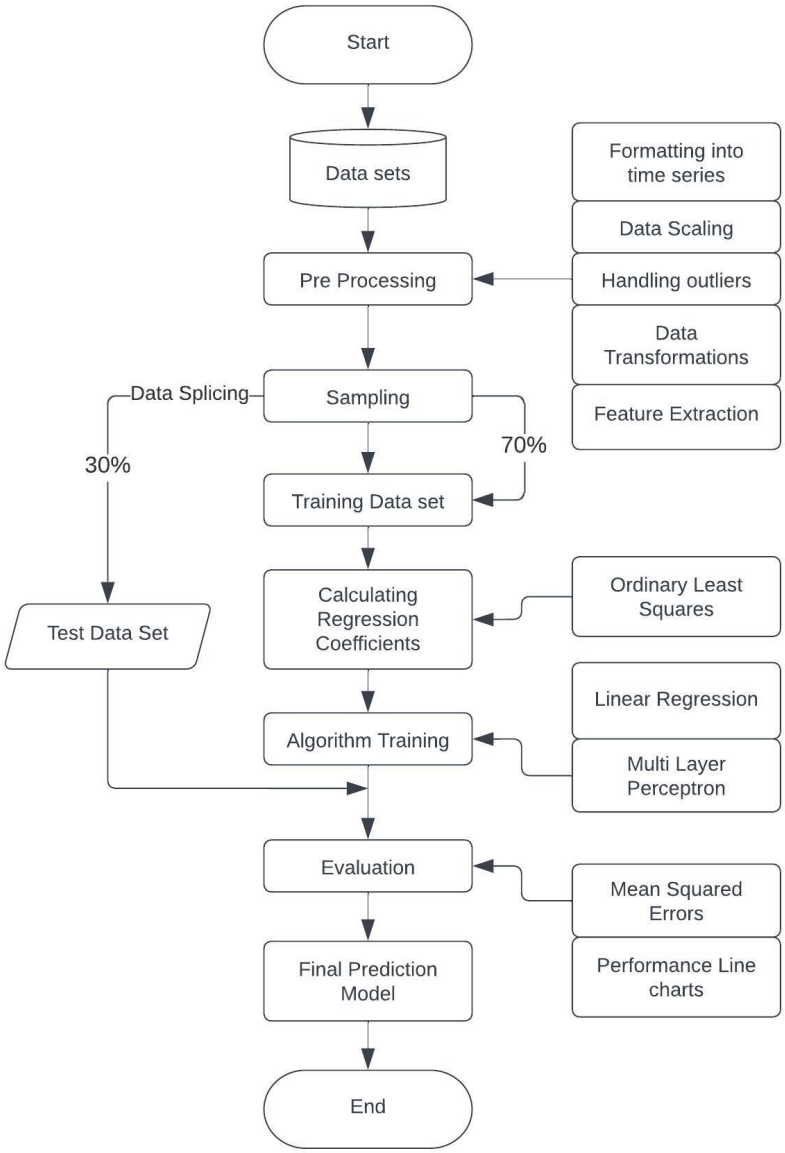
Demonstration of non linear output

The graph above demonstrates what a non-linear output from our MLP could looks like. It is worth mentioning that MLPs are not ideal for every scenario. A small dataset or a very simple feature can restrain what it is otherwise capable of.

Pseudocode [14]
1: chose initial weight vector -w

2: Initialise minimisation approach
3: While error does not converge do
4: For all (-x, -d) ∈ D do
5: Apply -x to network and calculate the network output
6: Calculate δe(-x)
7: End for
8: Calculate δE(D)
9: For all weights summing over all training patterns
10: Perform one update step of the minimisation approach
11: End while

Entire System



Entire system Flow

We have established what models we will be making use of in the previous sections. We have taken steps to ensure we end up with the most accurate forecasts possible. We treat and scale our data. We transform it into a time series and splice it into two chunks, one for training purposes and set aside a small chunk of our data to test our trained algorithm on. The testing data, which is unseen data for our models, when run on our trained models will provide us with an honest estimate of the accuracy of our models.

Our loss function will quantify this estimate for us. It will in addition serve as an optimiser function for our MLP to achieve a closer fit by tweaking weights and biases for every node during each iteration of the algorithm. It will continue training in this fashion until the loss within an acceptable range or it has run through the number of pre determined iterations. You do always run the risk of parameters not converging with a small number of iterations.
The flow chart depicts the flow of the entirety of our system.

Development Process

None of the software, development tools or environments I used for development are locked behind paywalls. They're all freeware or provided at no cost for educational purposes. I wrote my project in python as it is the most versatile language for machine learning applications. It is also heavily used in industry for this application. The development environment, I used was Jupyter notebooks to work, store and display my work as an executed .ipynb file on Github. Jupyter is also heavily used in industry for ML applications. Though my main reason for choosing it was because I had previous experience using it.

Ethical Considerations

My project is only aiming to predict short term housing prices. It will only be able to predict pricing for areas the data belongs to and from the time frame the data belongs to. It may be able to predict the trend the housing prices followed if the data covers enough time but the data I have does not cover that amount of detail.

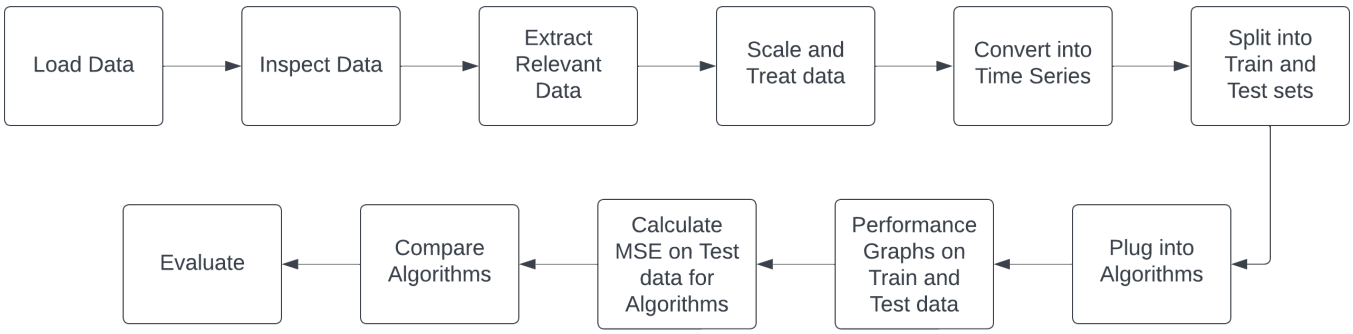
My work involves no human participants or human interaction other than my own to allow it to run. To make sure my work is ethical and GDPR compliant, I am closely following the guidance set out by the UK Data Service guidance and the EU. I have also downloaded the current

issue [15] for risk and compliance magazine and am using it as reference for good practices.

From this point forward, we will directly discuss the project in its current form.

Methodology

The methodology for the implementation of the project I have used is relatively simple. And all our work is done using python in Jupyter. We start by defining the data we are making use of. I define the data by dissecting its structure and visually representing it. We take a look at its indices and metrics. We locate and isolate the relevant section of data we need. We put this extracted data through preprocessing, treating cleaning all the observations in it and scaling it to improve the informational content of the data or for the regression algorithm we're going to be plugging it into. Once we have our data treated, we construct our time series with the defined step using that data. We create a design matrix and a single vector to create the two subsets our training and testing sets are going to consist of. The design matrix will serve as the time series we're using to predict the values contained in the single vector. Both testing and training sets will contain these. Once we have our data split into testing and training sets with their subsequent subsets, we can start plugging them into our regression. We're going to plot a graph for performance on both, our test set and our train set , as well as calculate the MSE for each of our algorithms performance on the training set. This should leave us with pairs of graphs for each of our three algorithms depicting performance on training and testing data. The graphs on testing data will allow us visualise each algorithms performance with respect to each other. In addition, MSE's for each that will provide us with a hard performance metric that will allow us to compare against each other to evaluate performance for each of our algorithms. The diagram consists of our simple methodology,



Methodology Flow

Data Breakdown

Before we directly move on to the methods used to preprocess the data, lets briefly examine the data first.

Source

The data I have sourced to use for my project was taken from the government run website [16] under data.london.gov.uk. It is run under the mayor of London and the London assembly, it only contains that pertains to London. They constantly collect and store data for everything in London from employment rates to transport statistics. They have metrics for many other things but, to our convenience, they also store housing data. This data is updated monthly and maintained. It is up to date for this year as well. This data is published under the Open Government Licence and is available for public use with the disclaimer [17]. The source of the data is also through government channels, HM Land Registry for England and Wales, Registers of Scotland and His Majesty's Revenue and Customs Stamp Duty Land Tax data for the Northern Ireland House Price Index. The data source is also assessed. With all of this, we can rest assured as to the integrity and availability of the data we're using.

Breakdown

The data was originally used to show the HPI (house price index) for the UK. This is why the original data contains a great deal of information such as sales volume, breakdowns of average price and HPI by types of houses like terraced or semi-detached. It also includes a page that is dedicated to house prices. These prices cover different parts of the country. But luckily for us, the most extensive data is for the London area. Here the prices are divided by borough. Each borough has its dedicated column of house prices. It is worth noting that each column contains *average house* prices, uses sales data and is not adjusted for inflation. They have calculated these averages themselves by applying a 3 month moving average to them. For example the estimate for March is a simple average of the calculated estimates for January, February and March. This helps them remove some of the volatility in the series at this level. You can read more about the data itself on their website [18].

The data spans a length of 23 years. It begins in January of 1995 and ends in January of 2023. This is true for all boroughs in the data, as well as other collective parts of the country. Each row (or observation) in our series denotes a month. In this fashion, our data covers all 12 months for each of the 23 years. Each observation records the average house price in the borough for that particular month. The months and year are the first column of data themselves in our data's raw form. That is the basic breakdown of the data we are using. Also worth noting, the original form the data is available in is a excel sheet that I converted into a .csv file using excels import function. It was this .csv file that I loaded into my project as the data frame called 'housing'.

Data Preprocessing

Now we move on to the single largest section of our code, the preprocessing for our data. Note that prior to this, we have already loaded our data into the Juypter notebook. Here we extract, treat, scale and transform the relevant data, in that order to better suit our purposes. Lets go about them discussing in that order,

Extraction

As we previously discussed in our data breakdown, the data is split by column where each column only contains data for a specific borough in the city. For my purposes, I selected the first column in our data at random. This is the data that we will be putting through our algorithms to train and test them on. The borough in particular being "City of London". Something important to note here is for time series, we need to have the associated date for each of our observations set as their index. Without this, we cannot run our data as a time series. To remedy this we extract not only the column of prices we need but the column for dates as well. I made a new data frame called 'remake', into this data frame I copied over the two columns we were going to need. This data frame is going to be what we make use of going forward.

```
def extract(z,name): # z=Dataframe to extract from, n=name of area to extract as string
    X = pd.DataFrame(housing)

    for column_headers in X.columns:
        if column_headers == name:
            cNum = int(X.columns.get_indexer([name]))
            #print(cNum)

    cols = [0,cNum]
    remake = X[X.columns[cols]]
    remake.set_index('Month-Year', drop = True, inplace = True)
    remake.replace(',','', regex=True, inplace=True) # Removing commas and converting to float.

    remake[name] = remake[name].astype(float)
    return remake
```

Extract Function

Treatment

We start our treatment by addressing the previously mentioned date index problem we had on our hands. I set the date column as the index for our prices column and drop the date column from out remake data frame (df). Normally at this point you would treat missing values in our data by dropping the rows with missing values or replace them with either the mean or median of the overall value distribution. Luckily for us, the top notch integrity of the data meant that, upon inspection before our extraction, there were no missing values to treat in our original housing df. Every column had a fixed number of 337 observations. Each column did start with its associated area code, but I dropped that row before we began our extraction. We would normally also need to encode non numerical data into a numerical from though a method such as one-hot encoding, but we have no need of that here since all our relevant data is already purely numerical. Back to our remake df, the data type of our data is currently of object type. This needs to be converted for it to be able to be operated on by our math. We accomplish this by first removing any special characters in our data such as comas in our case (separating units). We can then convert our data to float. It is important that our end data type is a float and not an integer because scaling will not properly apply on integer values.

The code snippet below depicts the extract function that I wrote. This will serve to be a useful tool for us. We simply hand it and our raw data and the name of the borough whose data we want. It will automatically extract and treat the data for that borough of us. This should make testing multiple areas much simpler.

Scaling

This is a very important step in machine learning. As the saying goes, “garbage in, garbage out”. Our data in its current form is garbage, to circumvent this, we scale it. Scaling data returns us with more consistent and accurate results from our regression algorithms. It makes sure very large or small observations (outliers) don’t dominate the data and skew our results too hard in one direction or the other. This is especially important for our MLP (NN) as it is particularly sensitive to outliers. In a nutshell, scaling data allows it to be read the same across all observations. Lets take a closer look at the scaling methods we used,

- Normalisation

Normalisation scales all our observation to be between the values of 0 and 1 or very close in terms of our testing set. The reason it is very close for our testing set is because the parameters we use to scale the data are derived from the training set but applied to both. The formula we’re using to normalise is, Normalisation = (X - min) / (max - min). Where X is an observation, min and max are the minimum and maximum values that exist in our training set respectively. Below is the function I wrote to implement this formula and normalise the treated data I handed it.

```
def normalData(z):

    low = 10000000000
    hi = 0

    temparr = np.empty(0, dtype=object)
    for i in range(0,math.floor(len(z)*(2/3)-3),1): # Converting df to usable vals
        temparr = np.append(temparr,remake.iloc[i:i+1,:])

    for i in range(0,len(temparr),1): # Find lowest val in training data
        if temparr[i] < low:
            low = temparr[i]

    for i in range(0,len(temparr),1): # Find highest val in training data
        if temparr[i] > hi:
            hi = temparr[i]

    for i in range(0,math.floor(len(z)*(2/3)-3),1):
        z.iloc[i:i+1,:] = ((z.iloc[i:i+1,:]) - low) / (hi - low) # Applies normalization to training data

    for i in range(math.floor(len(z)*(2/3)-3),len(z),1):
        #print(((z.iloc[i:i+1,:]) - low) / (hi - low)) # Works
        z.iloc[i:i+1,:] = ((z.iloc[i:i+1,:]) - low) / (hi - low) # Applies normalization to testing data
```

Function used to normalise data

- Standardisation

Now that we have normalised our data, we can standardise it. Standardisation scales our data to have a mean of 0 and a standard deviation of 1. This brings down all the data in the series to a common scale without distorting the differences in the range of the values. It is also true that standardisation assumes our data has a gaussian distribution (bell curve), but it does not mean that it can not help us despite our data not squarely falling in that bracket. The formula I implemented to apply standardisation is Standardisation = (X - mean) / standard deviation. Where X is a data point, mean and standard deviation is taken from our training set. The parameters are derived from the training set but applied to both. Below is the implementation in code I wrote.

```
def standData(z):

    temparr = np.empty(0, dtype=object) # can't operate on Df here
    for i in range(0,math.floor(len(z)*(2/3)-3),1):
        temparr = np.append(temparr,z.iloc[i:i+1,:])

    summ = 0
    count = 0
    summDist = 0
    countDist = 0

    for i in range(0,len(temparr),1):
        summ = summ + temparr[i] # operating on df, not np
        count = count + 1

    #print("sum:",summ,"count:",count)
    mean = summ / count
    print("mean:", mean)

    for i in range(0,len(temparr),1):
        #print("Values:", temparr[i])
        dist = temparr[i] - mean  # dist of x from mean
        #print("dist from mean:", dist)
        temp = dist * dist # square of dist
        summDist = summDist + dist # sum of sq dist
        countDist = countDist + 1

    #print("sum dist", summDist,"count:", countDist)
    std = math.sqrt(summDist/countDist) # calc Standard deviation

    for i in range(0,math.floor(len(z)*(2/3)-3),1):
        #print(((z.iloc[i:i+1,:]) - mean) / std) # Finally Works
        z.iloc[i:i+1,:] = ((z.iloc[i:i+1,:]) - mean) / std # Apply standardization to training data

    for i in range(math.floor(len(z)*(2/3)-3),len(z),1):
        #print(((z.iloc[i:i+1,:]) - mean) / std)
        z.iloc[i:i+1,:] = ((z.iloc[i:i+1,:]) - mean) / std # Apply standardization to testing data
```

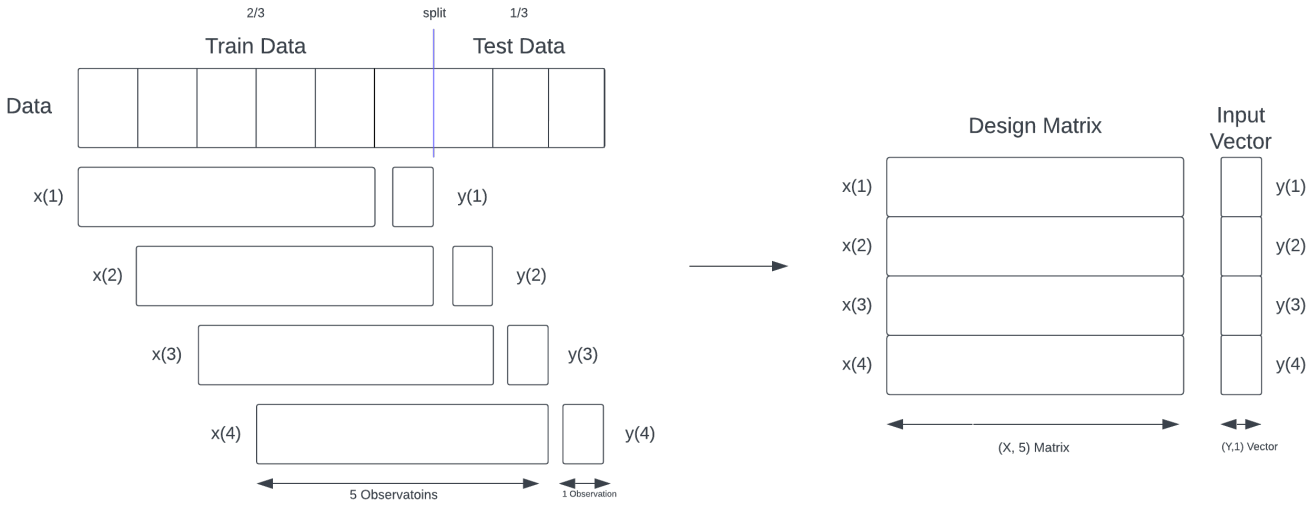
Function to standardise data

- Max-Scaling

We divide by our entire series by the biggest value in our series to scale our data further in case normalisation isn't enough by itself to appropriately scale our data. Unlike the previous two, this applies uniformly to all our data rather than separately on both sets. It is not appropriate for every instance of normalisation, and nor was it here. I wrote a function to implement it which is still present in the Scaling section of the Jupyter notebook but I will not include a snippet of here as I did not end up using it in the final result. When I did use it, all of our models performed significantly worse, which is why I chose to forego it.

Transformation/Data-split

Now that our data has been scaled, we can move onto transforming our data into a time series and splitting it into a train and test set. You can of course scale your data after splitting it but it was simpler for me to do it before hand as it makes calculations easier when scaling. To create our time series, we must first understand the structure of the time series we're creating. The diagram below will help us greatly visualise the structure we're using structure.



Time series structure

We split our data into a matrix X and a vector Y. We achieve this by progressively splitting the data into 5 observation long vectors with one step between each recorded vector and compiling them all at the end. This leaves us with a design matrix that is X rows tall and 5 columns wide. To build our Y vector, we grab the first observation after the end of each spilt we do making our making our design matrix X. The diagram above depicts this. These splits we're making with a step size of one is called lag inputs. These shifts in our design matrix denote the lag (there is small demonstration of this early on in the notebook). Our input vector Y, basically only contains the corresponding target values we're going to be aiming for using our design matrix. This is what our time series is.

The first function below, I wrote to transform our scaled data into the appropriate time series structure we just described. It will do this for the entire series so we have one design matrix and input vector. The second function is what will be splitting our structured data into two pairs consisting of a design matrix X and an input vector Y for our training and testing data. The training data is going contain 2/3rds of our series while the testing data is going be made of the remaining 1/3rd. This means 2/3rd of the design matrix and input vector is allocated to training X (design matrix) and training Y (input vector) and the remained of both becoming test X and test Y. With this, our data is at last fully prepared to be used in our models.

```
# where z(obj) = input data to use to create input vectors,
# sp(int) = the split for the data and st(int) = is the steps for the split.
def inVec(z,sp,st):
    # create a simple array with numpy empty()
    X = np.empty(0, dtype=object)
    Y = np.empty(0, dtype=object)
    for i in range(0,len(z.index)-sp,st):
        X = np.append(X,z.iloc[i:i+sp,:])
        # should grab Y
        Y = np.append(Y,z.iloc[i+sp:i+sp+1,:])
    temp = int(len(X)/sp)
    X = np.reshape(X,(temp,sp))
    Y = np.reshape(Y,(len(Y),1))
    return X, Y
```

Function to build our design matrix X and input vector Y

```
def split(z,c): #z(designMatX),c(inputVecY)
    trainX = z[0:math.floor(math.floor(len(z)*(2/3))):1]
    trainX = trainX.astype(float)
    trainY = c[0:math.floor(math.floor(len(c)*(2/3))):1]
    trainY = trainY.astype(float)
    testX = z[math.floor(math.floor(len(z)*(2/3))):len(z):1]
    testX = testX.astype(float)
    testY = c[math.floor(math.floor(len(c)*(2/3))):len(c):1]
    testY = testY.astype(float)
    return trainX, trainY, testX, testY
```

Function to split our time series into training and testing pairs

Linear Regression

We can now look at our first model. The general structure of operation we use for each model, how we calculate the weights using our data, using those weights to calculate predictions, how we apply these to both training and testing data to calculate performance on both and quantify that performance in the form of MSE.
This is a form on linear regression that we're going to use to make our forecasts (predictions). Being a linear regression, it follows the standard format of being a function of the summation of the weights (regression coefficients) times the observations, but lets take a closer look at what. I did.

```
# Implementing (linear) OLS fitting using our input matrix and vector to calculate the best weights for our model.
X = trainX

# W = (Xt * X + λI)-1 * Xt * Y
X_b = X.T.dot(X) # Xt * X
X_b = np.add(X_b,0.0001*np.eye(5)) # X_b * λI (replaces np.ones to stabilize inversion)
X_b = np.linalg.inv(X_b) # X -1
X_b = X_b.dot(X.T) # inverse * Xt   (NOT * X_B, BIG DIFFERENCE)
theta_best = X_b.dot(trainY) # X_b * Y

# should find us the best regression coefficients (weights) to reduce mse.
```

Implementation of OLS (ordinary least squares) fitting to find regression coefficients

Above is the implementation of OLS I wrote. This is going to find the the weights we're going to use in our autoregression (AR) later on to make our forecasts. We can think of our weights as being a measure of how much influence the observations they are attached to will have on the end prediction, higher value weights denote more influence and vice versa. The formula for OLS I was using was W = (Xt * X + λI)-1 * Xt * Y, Where Xt is our transposed design matrix, X is our design matrix, λ is a small value, I is an identity matrix and Y is our target input vector. Here, λI is simply serving to stabilise our matrix inversion. Our ideal weights will give us weights will make the squared distance between the output and the given target as small as possible over all points. This returns us 5 regression coefficients in accordance with the number of columns in our data.

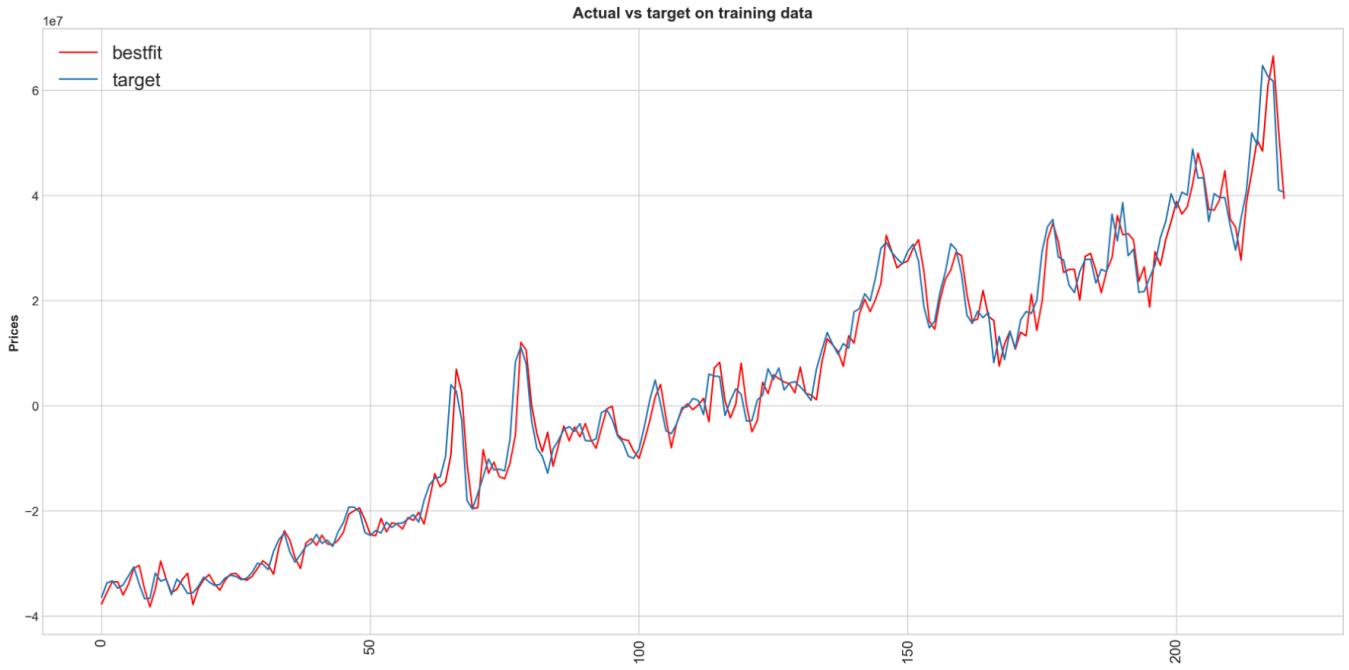
Now that we have our weights (regression coefficients), we can plug them into our autoregression depicted in the code above. The formal formula for autoregression is Yt = Xt * W + Et. Where Yt is our predictions, Xt is our design matrix, W the weights and Et the MSE calculated on our training data. Our output will be our predictions in accordance with the design matrix we put in and we can compare these with their corresponding targets in our input vector.

```
# Implementing Auto regressive model using our regression coefficients found using OLS

X_new_b = testX
y_predict2 = X_new_b.dot(theta_best) # should be Yt = Xt * W + Et (though we're not using Et)
print(len(y_predict))
y_predict2[90:96,:]
```

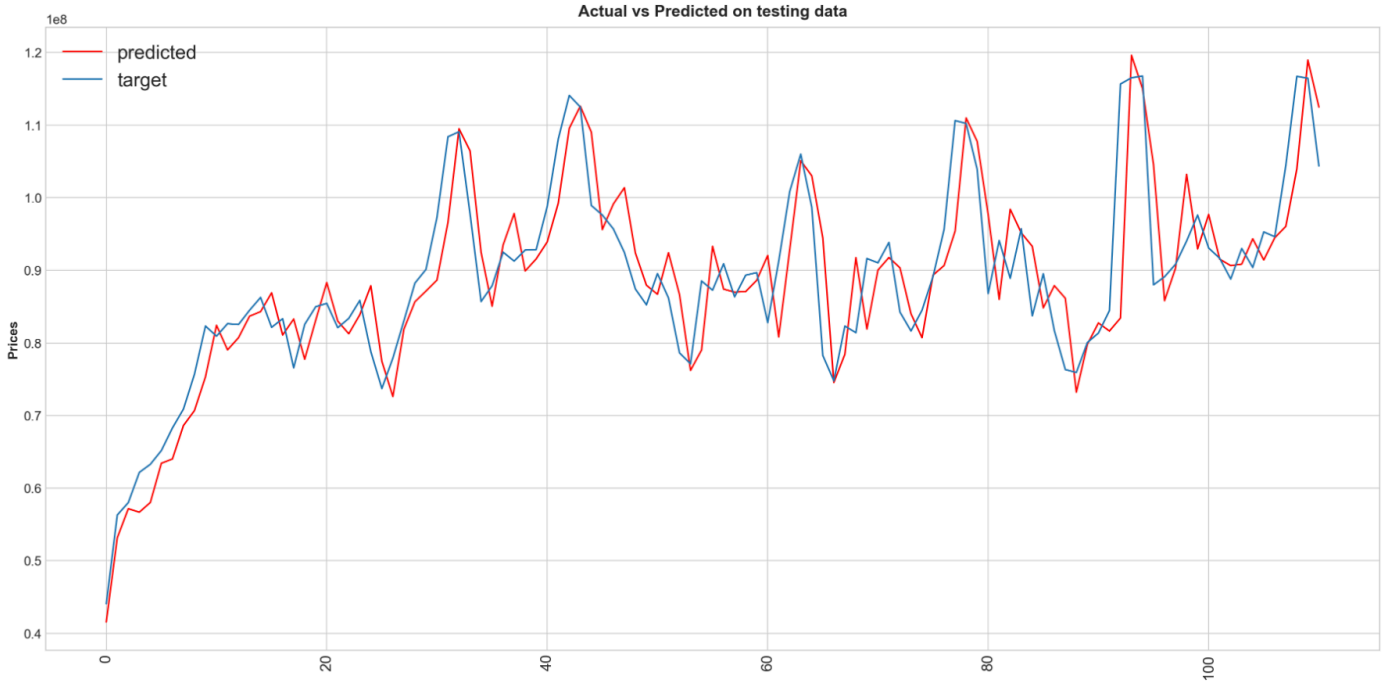
Implementation for Autoregression (AR) to make our forecasts (predictions)

First we run OLS on our training data 'trainX' and its corresponding input vector 'trainY' to calculate the weights we're going to be using in our AR. With these 5 calculated weights run AR on the training data alone to receive our predictions for the training data. We then proceed to plot these predictions against their target values in 'trainY'. We end up with the graph below and a calculated MSE of 1.31524549e+13.



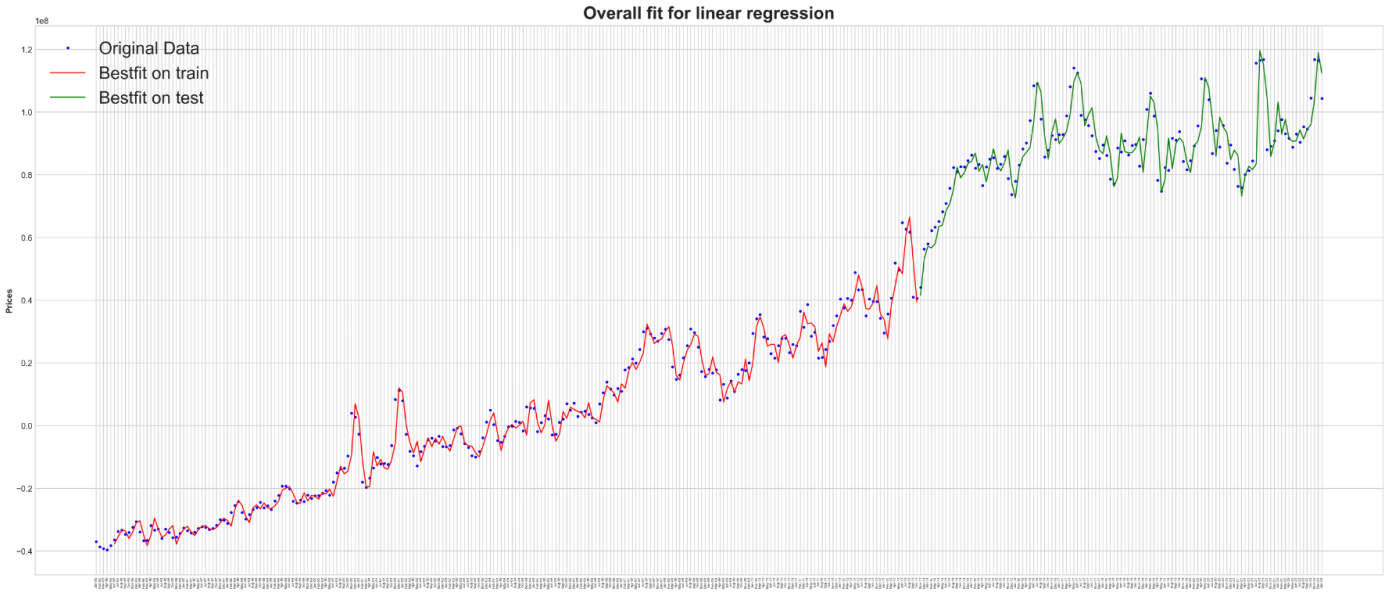
Performance on training data

Now we re run this same procedure on our test data using ‘testX’ and ‘trainY’. We update our weights using OLS for the testing data and run them through AR to make our predictions using the test data. Below we can see the graph I plotted to visually see the performance as well as having calculated the MSE on our test data to be 4.41450729e+13.



Performance on testing data

Now we can plot a graph to visualise the overall performance of this model and compare the MSE's we got for training and testing.



Overall performance on Linear

Our goal is to minimise the squared errors, MSE. With an MSE of 1.31524549e+13 on training and 4.41450729e+13 on testing, I believe the model is performing well. The performance metric itself is not high and the fact that we got a higher score testing data means that the model is not overfitting and responding well to unseen data, after all it would be natural to expect it to perform worse on unseen data. If our MSE was even lower for testing, we would assume it is overfitting as it has been too tailored to the training data and will not respond well to unseen data in a slightly different formats or trends.

It may be worth mentioning that I am calculating my own MSE using a function I Wrote which is depicted in the code below. This is what I'm using to calculate the use for every model as the output is more readable that what a built in library would return. I am also using another function I wrote which makes plotting our overall performance graphs possible by adding the date indices back to our predicted values.

```
# Manually finding MSE
# mse = (1/len(z)) * sigma(len(z)-1)((z-x)^2)

def findMSE(z,x): # Where z = target vals, x = predicted vals
    summ = 0
    for i in range(0,len(z),1):
        diff = z[i] - x[i]
        #print(z[i],x[i]) # For if you wanna compare the pairs
        sq = diff * diff
        summ = summ + sq
    mse = summ / len(z)
    return mse
```

Function to manually find MSE

Polynomial Regression

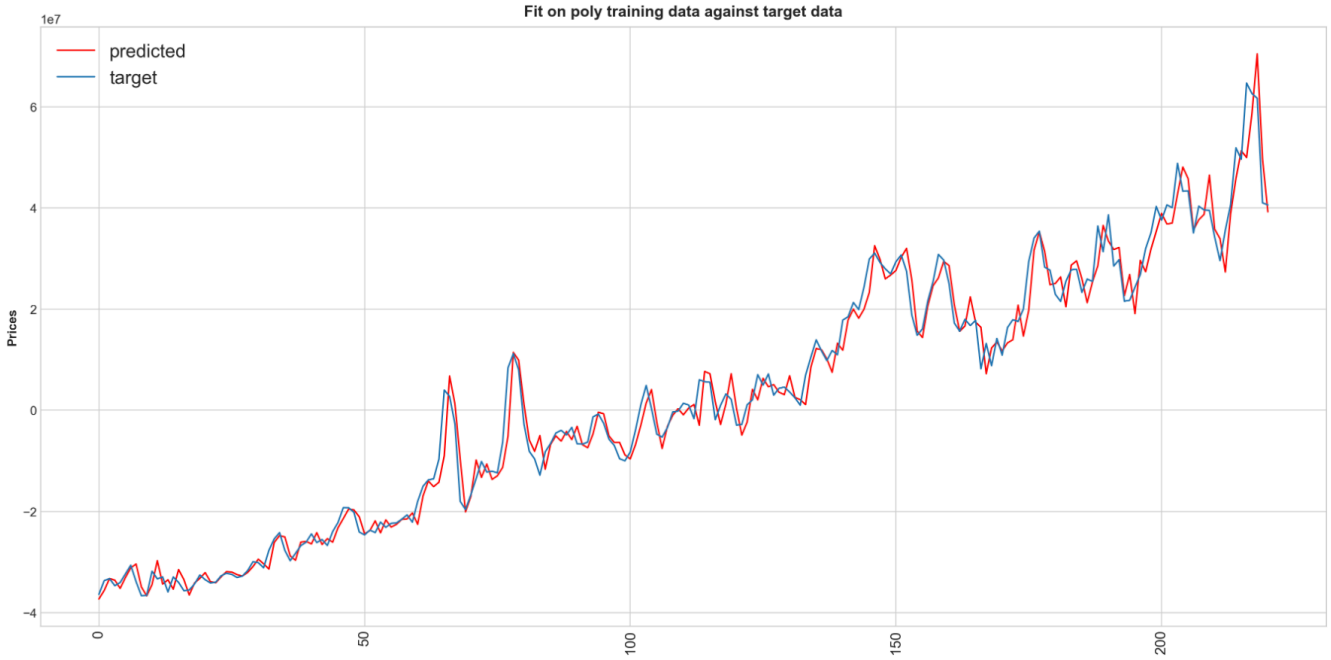
We will follow the general structure I defined at the beginning of linear regression. Calculate weights using OLS, plug weights into AR to make our predictions, plot graphs and calculate MSE to visualise and compare performance. I will however highlight the differences here. For starters, we're going to convert convert our data into it's polynomial form. To achieve this, I wrote a function that will do this for us,

```
def polyBuild(z): # two nested loops for accessing index i and j of the 2d array
    newCol = z
    for i in range(0,z.shape[1],1):
        tempCol = np.empty(0, dtype=object)
        for j in range(0,z.shape[0],1): # or you could transpose
            temp = z[j:j+1,i:i+1]
            sqa = temp*temp
            tempCol = np.append(tempCol,sqa)
            #print(tempCol)
        #print(newCol)
        newCol = np.c_[newCol, tempCol]
    return newCol
```

Function to convert normal Data into polynomial data

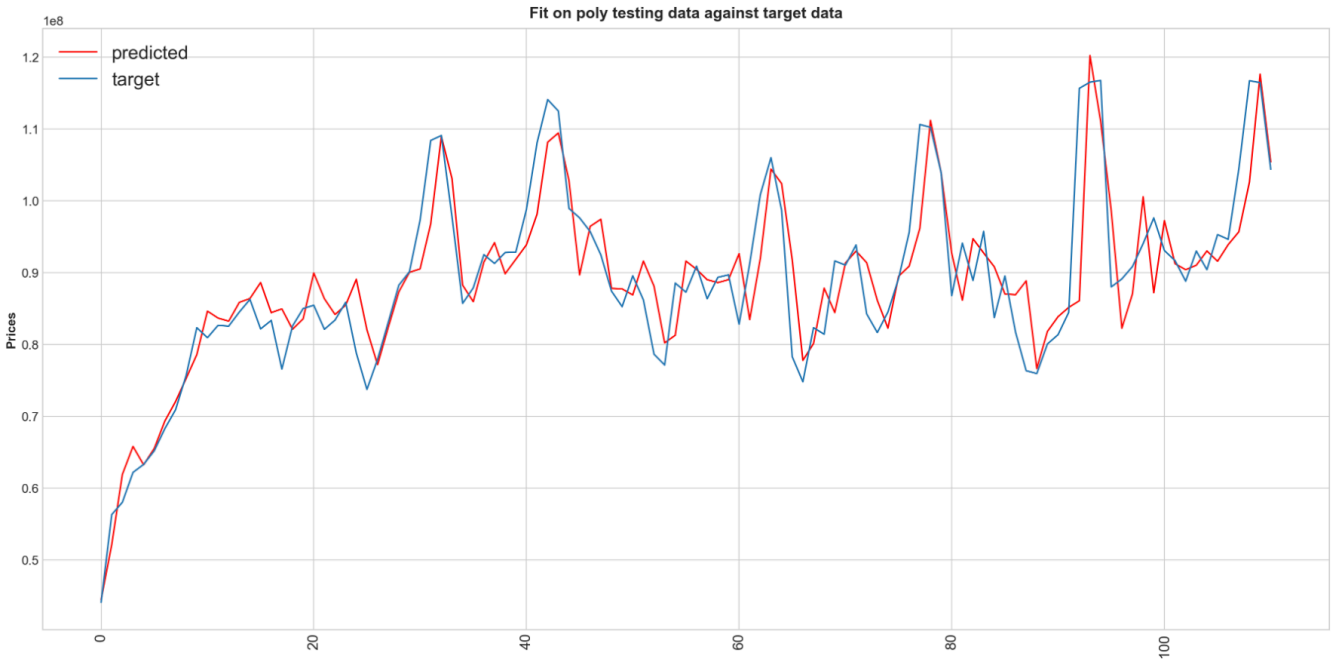
This function will extract each column of data, square every observation in it and concatenate this column of squared values to the end of our existing data. This way our 5 column data will have 5 squared columns added to it. X1 > X1^2 and so on for every column.

Now we plug this data into our OLS, with the only exception being that in the line X_b = np.add(X_b,0.0001*np.eye(10)), our identity matrix has become 10 by 10 to match our new data. Our output will also be 10 regression coefficients to reflect this change.



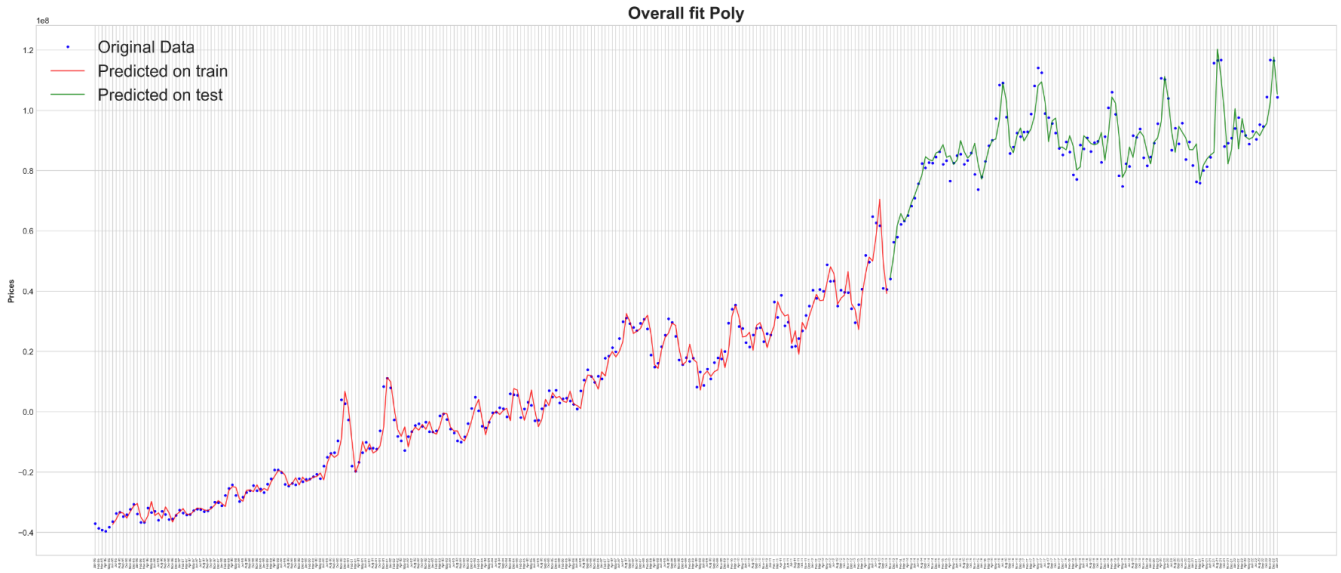
Performance on training data

Our calculated MSE on training data is 1.263439e+13.



Performance on testing data

Our calculated use on the testing data was 3.56085064e+13. We can draw largely the similar conclusion as did for linear regression. The thing of note here is that both training and testing MSE was lower than on linear meaning that our polynomial model performed better overall.



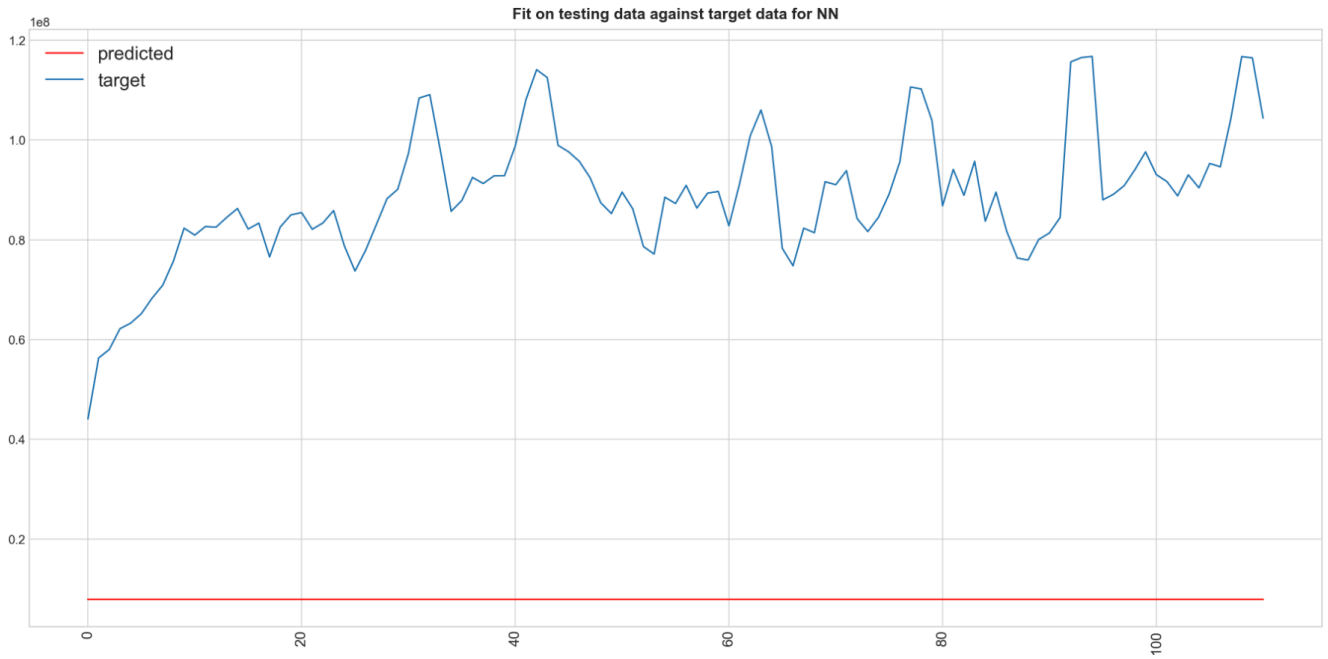
Overall performance on Polynomial

MLP

Now we move onto the last model, MLP. This is a NN (neural network) from the sklearn library. I would liked to have made this myself as well, but due to time constraints am using a library. Moving on, we can set many different parameters for our MLP.

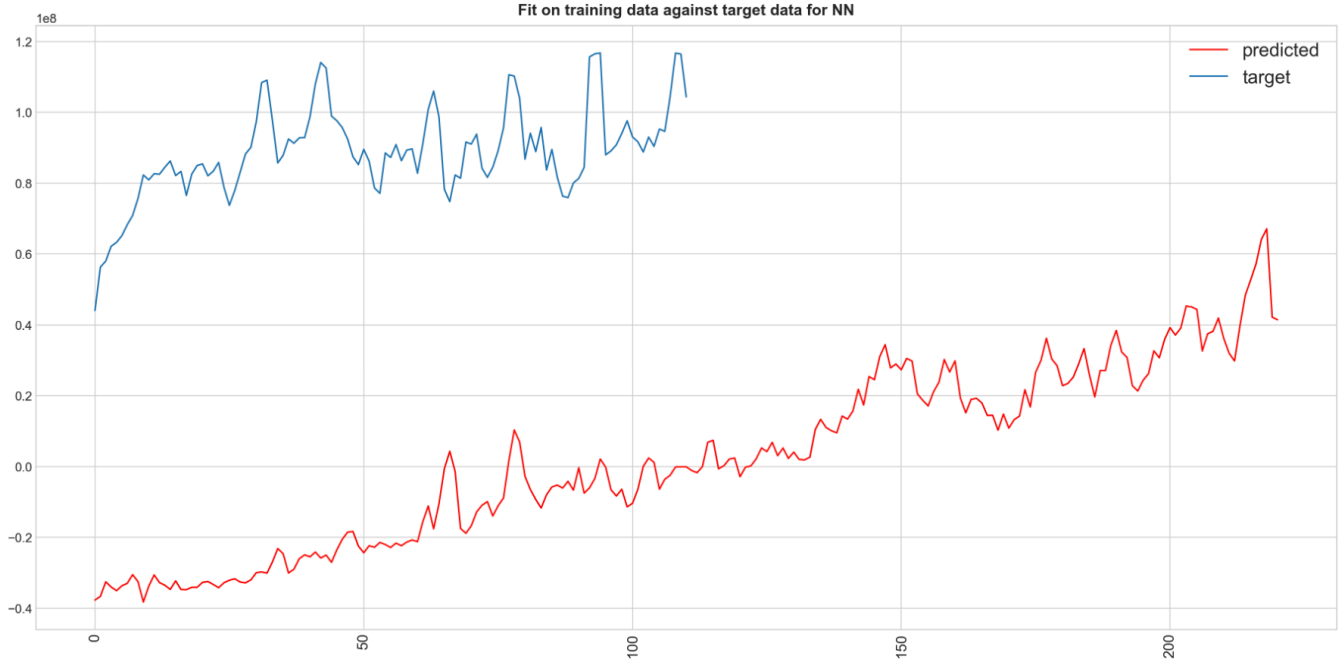
I have done a few different models for MLP with varying parameters, each with varying degrees of success. Lets look at a few models performance and parameters.

The graph for our first model with parameters: hidden_layer_sizes=1, activation='tanh', solver='sgd', alpha=0.0002, learning_rate='constant', learning_rate_init=0.001, max_iter=50, early_stopping=True, is depicted below. As we can see, the result is absolutely abysmal. So let us cut down on a few parameters and try again



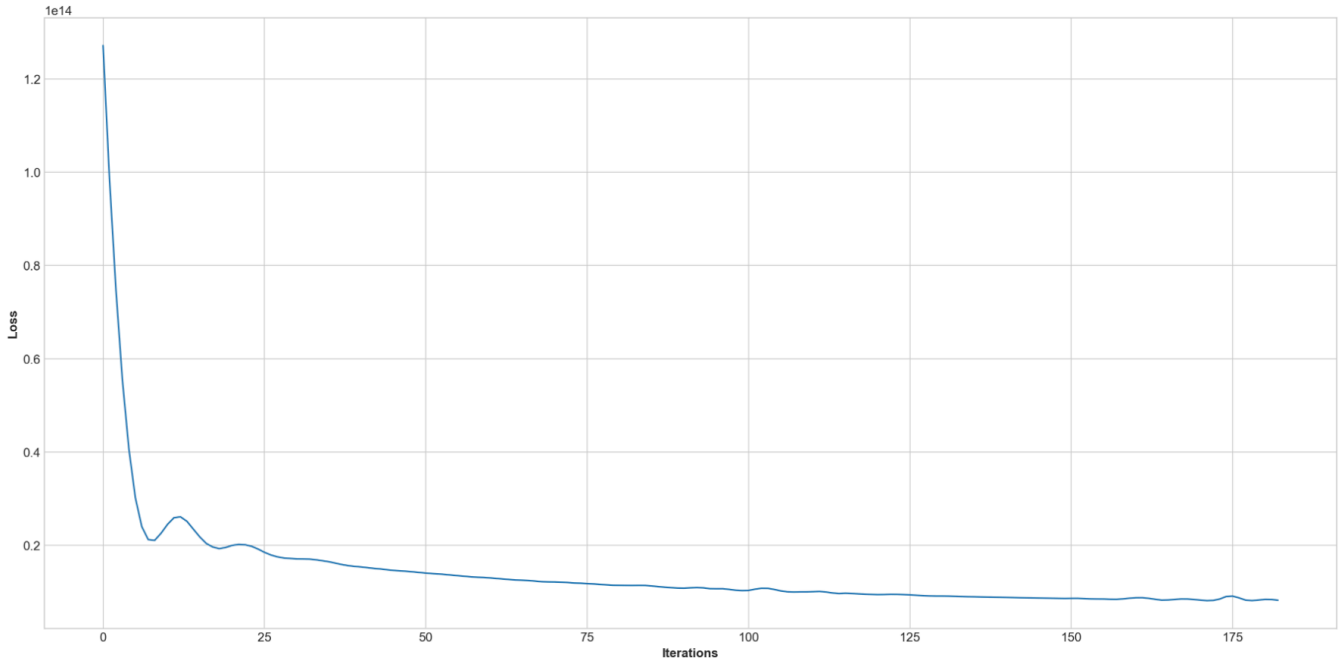
Model 1

Our second model below with parameters: random_state=1, max_iter=900, solver="lbfgs", sees a moderate improvement but far from what we'd like.



Model 2

The third and last model with parameters: random_state=1, max_iter=800, solver=“adam", below performed the best so far. It being solver Adam also lets us plot the loss



Loss over iterations

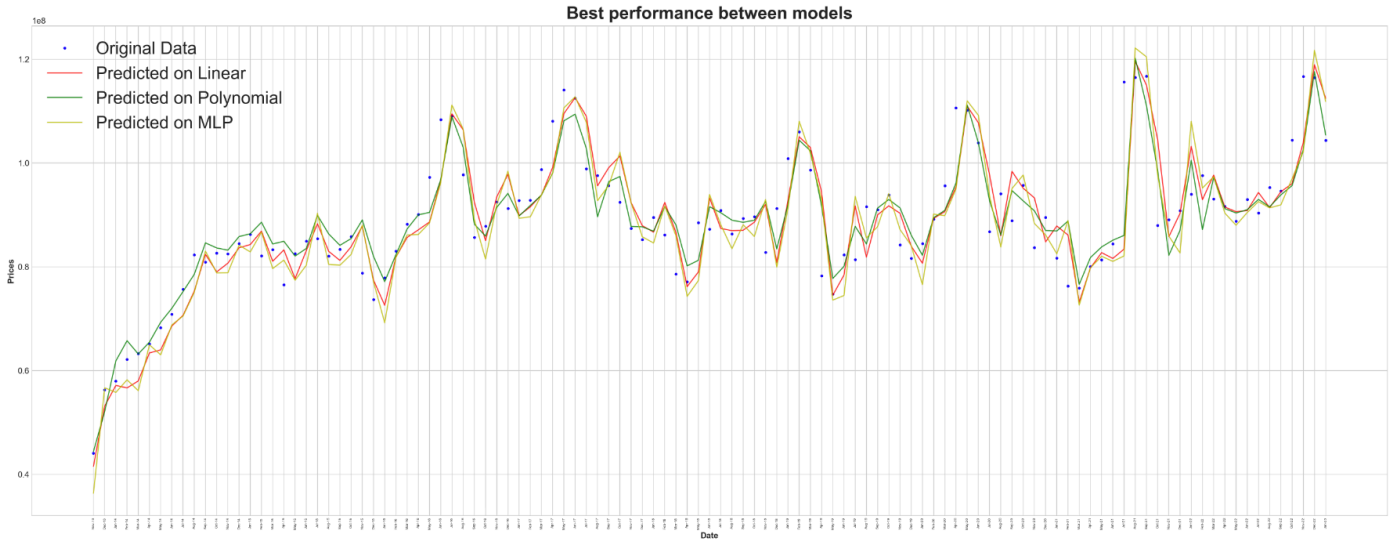
Model 3

This will take us forever, if we keep trying to manually plot models with different parameters to determine the best combination. To circumvent this, we will be making use of GridSearchCV which will hand a list of parameters to try. It will make and test models for with combinations of parameter in the list we handed it. It will then return the combination that performed the best. Lets look at these parameters and its performance.

The best combination GridSearchCV found was: 'activation': 'relu', 'alpha': 0.4, 'early_stopping': True, 'hidden_layer_sizes': 3, 'learning_rate': 'constant', 'learning_rate_init': 0.01, 'max_iter': 500, 'random_state': 0, 'solver': ‘lbfgs’. The MSE we calculated for this was 4.77002905e+13. Lets look at its graph on the testing data below.

Performance on best model determined by GridSearchCV

Results



Model Comparison

Now that we have made and tested on all our models, lets look at how they compared in comparison to each other, Lets start by plotting all their performance against each other, We have plotted all our models performance on the test data below,

Listing all their MSE's on the test data,

```
Mean square error on Linear [4.41450729e+13]
Mean square error on Polynomial [3.56085064e+13]
Mean square error on MLP: [4.77002905e+13]
```

Looking at just the MSE's and graphs, it is True that Polynomial performed the best. This is because of the way our data is structured and its quantity. The amount of data we we're using was not very large. At most, the number of observations our models had to shift through were 3320. This made our simple polynomial regression fit our structured and small amount of data quiet well.

If we were to have far more complex data, MPL would out perform all of our other models by a good margin. More comprehensive data, with more features of different kinds to process, would make our linear and polynomial regression struggle to the point where it would probably end up dropping certain features influence in the predictions it make, simply because it can't process them well. MLP would tackle this with far more ease. MLP's versatility to process and handle this kind of complex data is what makes it far more robust than any of our other models. Besides, despite the unfavourable conditions, it performed very close to linear regression.

Note,
I also tried a version of OLS where I concatenated a column of ones at the end of our design matrix to stabilise the inversion. This left us with 6 regression coefficients. I abandoned this approach in favour of the original one because the MSE was better on the original approach. I have however left it in my project still if you want to take a closer look at it.

Discussions

At last, we have not only built all our models but procured data for them, fit that data in a manner that would best suit them or allow them to perform their best. We fed this data through each one of them in different forms and received their results in turn. With these results we have evaluated their performance and compared it to each other. We know where we stand in terms of our best model for this specific task and have demonstrated the flexibility of others first hand. But how does this fit in the broader picture?

The route to predict I chose to follow from the two I had initially uncovered was forecasting. It was true that feature regression (where you predict based on a houses individual features) was a more popular

approach and forecasting house prices is a little more uncommon, but both end in the same place. And this was no different for me. My models performed just as I had intended them to, as had many others before mine. With all the perquisites met (and not so well met from testing), my models did indeed predict pricing within a very acceptable range of error. Though I have not pushed the envelope any further in context of the field than when I began, I do believe I've furthered the field a minuscule amount with what I've learnt. Perhaps this knowledge will one day allow me to truly push the envelope further when I'm professionally working. Anywho, the work fits well with other forecasting models I studied before hand in terms of receiving an accurate result, simple as my implementation was. I have firmly come to believe though, that neither of the two approaches on their own are through enough to handle such a volatile variable. If possible, I would like to see more in terms of either combining both approaches or to use both simultaneously to achieve greater accuracy and to be more impervious to external factors, or perhaps better predict them.

I would also like to briefly touch on the changes my project went through in its many iterations. When I initially started, I was not very well versed with working with machine learning and my initial research led me to follow the individual feature route. However, at my supervisors insistence, I looked into the forecasting side of things. With enough research, it did seem like the route to take. I was also initially planning to only use Neural networks for all my models, but once I actually sat down to write the code, I was hit with the sheer complexity of the task. Keeping that in mind, I switched to simpler models for the NN as well as to use non NN models. For all the models I built, my supervisor provided me with guidance and material to help me better understand what I was building. Without those materials I certainly would not have been able to build so much from scratch and would have had to rely far more on libraries.

Limitations and Future works

There was much more than I wanted to do but was unable to because of time constraints and complexity. To name the biggest limitation of my project, was the MLP I utilised. I had wanted to build the neural network from scratch as I have for my other models but couldn't find the time between my other work. My work is in all honesty simpler than I would have liked, but the results are irrefutable of what I have managed.

For future work, there is quiet a lot I would like to improve on. The main thing I would like to improve on is for one, my NN's obviously but also to be able to modify my models to be more robust. To make them more sensitive to picking up on external factors like inflation or market trends and utilise these to improve the accuracy of my results. Secondly, as I had stated in previous hand ins. I built this in mind as a consumer tool, but a layman can't use something like this in its current form. I would very much like to build an interface/UI that would allow anyone to use this with more ease. Also perhaps to be more flexible in terms of the data sets it can process.

Conclusion

We have achieved what we set out to do. We wanted to forecast house prices in specific areas using previous listing prices for the houses in those areas. These areas being borough in this specific case. More than that, we wanted to see how viable this approach to prediction would be in comparison to other methods.

We used a time series on regression algorithms to forecast house prices in the interim and future prices. I would say we succeeded in this endeavour. The results we received were positive and within a small margin of error. We can not conclude from this that forecasting is the best method of prediction for such a volatile thing. But we have definitely proved that this has the potential to be a very robust and reliable method of price prediction. Forecasting can be safely used outside of the standard things like stocks and bonds it is used for.

Bibliography

[1]. https://www.zoopla.co.uk/house-prices/ & https://www.rightmove.co.uk/house-prices.html

[2]. https://www.propublica.org/article/realpage-accused-of-collusion-in-new-lawsuit

[3]. https://www.zillow.com/z/zestimate/

[4]. https://www.kaggle.com/competitions

[5]. A. Varma, A. Sarma, S. Doshi and R. Nair, "House Price Prediction Using Machine Learning and Neural Networks," *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, 2018, pp. 1936-1939, doi: 10.1109/ICICCT.2018.8473231.

[6]. M. Jain, H. Rajput, N. Garg and P. Chawla, "Prediction of House Pricing using Machine Learning with Python," 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC), 2020, pp. 570-574, doi: 10.1109/ICESC48915.2020.9155839.

[7]. Lasse Bork, Stig V. Møller, “Forecasting house prices in the 50 states using Dynamic Model Averaging and Dynamic Model Selection, International Journal of Forecasting”, Volume 31, Issue 1, 2015, Pages 63-78, ISSN 0169-2070,

[8]. Bishop, Christopher M., Pattern Recognition and Machine Learning (New York: Springer, 2006)

[9]. https://en.wikipedia.org/wiki/User_requirements_document

[10]. https://en.wikipedia.org/wiki/ Software_requirements_specification

[11]. https://en.wikipedia.org/wiki/Functional_specification

[12]. https://stats.stackexchange.com/questions/344658/what-is-the-essential-difference-between-a-neural-network-and-nonlinear-regressi

[13]. https://support.minitab.com/en-us/minitab/20/help-and-how-to/statistical-modeling/regression/how-to/nonlinear-regression/before-you-start/example/

[14]. Akintola, Abimbola & Balogun, Abdullateef & Lafenwa-Balogun, Fatimah & Mojeed, Hammed. (2018). Comparative Analysis of Selected Heterogeneous Classifiers for Software Defects Prediction Using Filter-Based Feature Selection Methods. FUOYE Journal of Engineering and Technology. 3. 10.46792/fuoyejet.v3i1.178.

[15]. https://riskandcompliancemagazine.com/latest-issue

[16]. **https://data.london.gov.uk/dataset/uk-house-price-index**

[17]. *Contains HM Land Registry data © Crown copyright and database right 2020. This data is licensed under the Open Government Licence v3.0.*

[18] .**https://www.gov.uk/government/publications/about-the-uk-house-price-index/about-the-uk-house-price-index**

[19]. **https://github.com/LiptonTeaa/Final-Project.git** (My project)

Note,
I am using the IEEE reference system.