IB Diploma Programme

Extended Essay

The mathematics behind Stochastic Gradient descent in linear regression machine learning models.

Investigating Stochastic Gradient Descent:

How are linear regression machine learning models effectively constructed using mathematics to generate accurate predictions for multidimensional input data?

IB Mathematics Analysis and Approaches HL

Word count: 3941

Table of contents

Introduction

As machine learning becomes more significant in our everyday lives, from sorting email algorithms to profound language models, the underlying mathematics of the models is cleverly implemented to allow this. Machine learning(ML) focuses on the idea of machines being able to learn specific tasks which they can replicate to an adequate level of accuracy; a measure of how well the model performs on unknown data(Ronit Kumar).

Such models need to be trained. This focuses on feeding the algorithm data which it can use to draw conclusions from the patterns that it finds to make adequate predictions (Ronit Kumar). It is important to realize the patterns may seem abstract and may not be possible for humans to find. Machine learning model training is often categorized into supervised, unsupervised and reinforcement learning (Ronit Kumar). The key concept of supervised learning models is human labeled data, which algorithms use to train on. This may be oddly like the way we learn at a very fundamental level: After seeing a few examples from the same class of an object we are able to accurately classify a differently looking object of the same class.

Though, machine learning models require many more such examples. The underlying mathematics of the models is cleverly designed to make this possible. The significance of these models only grows and the processes behind them may almost appear incomprehensible; however, they can be comprehensibly understood through the constituting mathematics.

The Problem

Researchers employ machine learning models to accurately extrapolate predictions beyond the input domain, an example of which can be housing prices. Machine learning models contribute the advantage of taking in any number of variables and discovering important patterns within the data, which is crucial for increasing accuracy of predictions as most outputs depend on a variety of different factors. Linear regression ML models are fundamental to understanding these processes, thus through linear algebra, multivariable calculus and probability, aided with, Python, MatPlotLib, and TensorFlow, the Mathematics behind the processes will be explored.

Understanding the Mathematics

Linear regression

Linear regression is the first step to understanding gradient descent. Linear regression is the linear approach of modelling the relationship between explanatory variables (Dependent and independent variables. In the data $D = \{(y_i,\ x_i):\ i\ =\ 1, \ldots ,\ k\}$, where y_i is the i'th response measured on a continuous scale. $x_i = (x_{i1},\ \ldots\ ,\ x_{in})^t \in \mathbb{R}^n$ is the associated predictor vector; and $k(\gg n)$ is the sample size. The linear model is specified as: (Su et al.)

$$y = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_n x_{in} + \varepsilon_i \text{ —}\textit{Equation1}$$

for $i = 1,\ \ldots\ ,\ n$. In matrix form,

$$y = X\beta + \varepsilon \text{ —}\textit{Equation2}$$

Where $y = [y_i]k \times 1$ is the n-dimensional response vector; $X = \left(x_{ij}\right)_{\{k\times(n+1)\}}$with $x_{i0} = 1$ is often called the design matrix; and $\varepsilon = [\varepsilon_i]k \times 1$.

This general form(Equation1) can be viewed as the general form for a linear function $y = mx + c$ that has n number of input variables and thus n number of different coefficients.

There are four major statistical assumptions involved in the specification of model (1) or (2), and they are:

1. (Linearity) $\mu \equiv [E(y_i|x_i)]k \times 1 = X\beta$;

2. (Independence) ε_i's are independent of each "other;

3. (Homoscedasticity) ε_i's have equal variance σ 2;

4. (Normality) ε_i's are normally distributed(Su et al.).

The general *equation1* is denoted in a compact form using the sigma notation for summation:

$y_j = \sum_{j=0}^{n} \beta_j \, x_j^i$ where $x_0^0 = 1$ —*Equation3*

Where j is the j'th explanatory variable, and i is the i'th training example.

This form holds true for any j number of explanatory variables.

For equations, *1*, and *3*: The subscripts and superscripts only indicate the index of the variable, not the exponent.

The essence of ML algorithms is the proper adjustment of the parameters to minimize the error. For that reason, the three aforementioned stages are implemented:

1. Initializing the *hypothesis function*
2. Calculating the mean *square error*
3. Updating the parameters applying *gradient descent* algorithm

Hypothesis function

The hypothesis function is very similar in form to the common straight-line function $y = mx + c$, however it is defined for more than one input variable to allow for more accurate models,

$$h_{\beta_0,\ldots,\beta_j}\left(x_0^i, \ldots, x_j^i\right) = \sum_{j=0}^{n} \beta_j\, x_j^i, \text{ where } x_0^0 = 1$$

$$\equiv h(x) = \sum_{j=0}^{n} \beta_j\, x_j^i$$

Where:

$\beta =$ "Parameters"/ Weights

$m =$ "Number of training examples, e.g. rows in (See appendix 1)

$x =$ "Inputs/ Features / Explanatory variables/ Independent variables"

$y =$ "Output/ Target variable"

$(x,\, y) =$ "Training example"

$n =$ "Number of features/ Number of input variables.

$i =$ "Index of the current training example.

$j =$ "Index of current feature or explanatory variable.

The hypothesis function is a function of the explanatory variables x and the parameters β; therefore, the output is dependent on both. The coefficients are referred to as weights and the free term (β_0) is regarded to as the bias term.

Before any training process begins the parameters of the hypothesis function are initialized to random values. Consequently, the hypothesis function predicts a respective output value. The error function calculates an error.

Mean square error function

The algorithm requires a sort of metric to measure how well it is performing for the current weights. Alongside absolute mean square error, the mean square error is the standard metric used in model evaluation (Hodson), as the derivation is very natural through a set of probabilistic assumptions later explained.

The objective of this function is to calculate how much the prediction was off, with respect to the actual value for the respective values of the explanatory variables; this is done by adding the square differences between the predicted values by *function3* for the current set of weights and bias', and the actual corresponding dependent variables and calculating the mean of the sum of the result for each training example i. *Equation4* shows this exact function.

$$J(\beta_0, \dots, \beta_j) = \frac{1}{2m}\sum_{i=0}^{m}\left(h_{\beta_0,\dots,\beta_j}(x_0^i, \dots, x_j^i) - y^i\right)^2 \text{—}\textit{Equation4}$$

The error must be minimized to maximize the prediction accuracy of the model(Andrew Ng). Figure1.0, shows a general plot of $J(\beta_0, \dots, \beta_j)$ for one input variable, i.e. j=1.

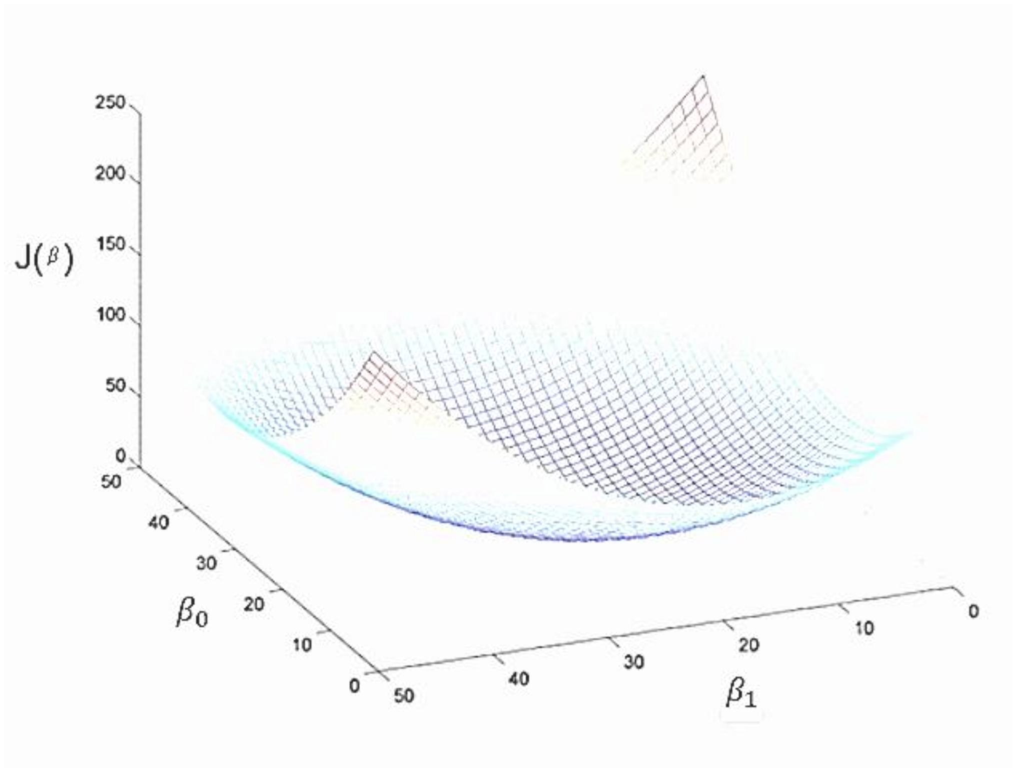


Figure 1.0 Mean squared error function plot for two function parameters. (Andrew Ng)

The function has a general convex shape and has a global minimum where $J(\beta_0, \ \dots, \ \beta_j)$ is minimized, this corresponds to the two optimal values for the parameters (Andrew Ng) that must be optimized. This is achieved by implementing the gradient descent algorithm.

Stochastic Gradient Descent

Gradient descent is a function implemented to minimize the mean square error function, and thus converge at a global optimum, in terms of the optimal values for the weights and biases so the predictions output by the hypothesis function are as close as possible to the actual labels for the input variables.

Gradient descent is an algorithm that is applied to each weight and bias individually keeping everything else the same, and then all parameters are updated at the same time, to avoid using already updated parameters in consequent predictions. Figure2.0 visualizes gradient descent of an example error function for two parameters. The starting position is defined by already pre-defined values of the parameters. The gradient descent algorithm can be understood by considering the starting position of a ball on the slope of the function and rolling down to the closest and lowest point of elevation(Andrew Ng). Thus $J(\beta_0, \ \ldots, \ \beta_j)$ is minimized. This algorithm is iterated until convergence or epoch (Specified number of iterations). One property of gradient descent is that it depends on the starting position, the initialized parameters, which means, that it can converge to different local optima.

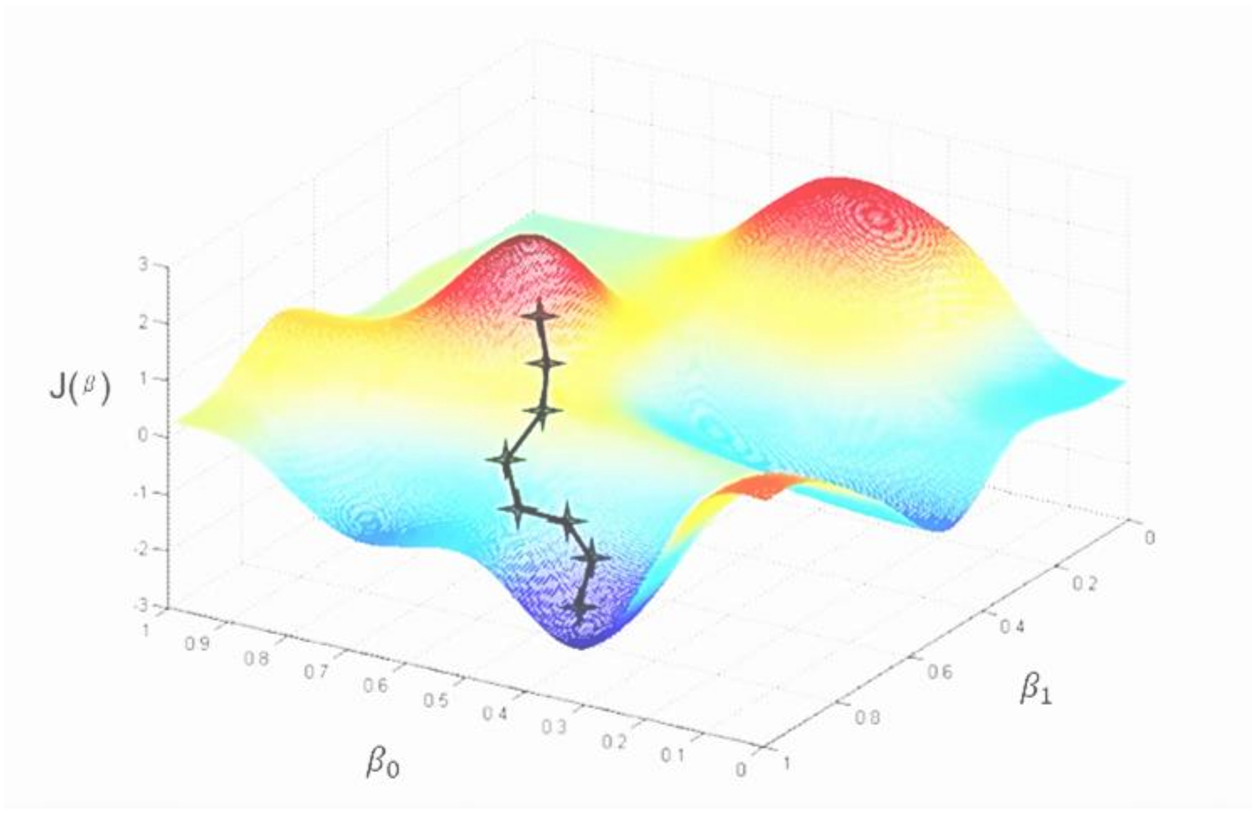


Figure 2.0, Gradient descent algorithm travel path while converging to a local optimum of a sample error function. (Andrew Ng)

The parameters are updated upon each iteration of the algorithm by subtracting the scalar product of the derivative of the error function from the current value of the parameter. Since the value of the derivative will get smaller and approach zero as the algorithm approaches the optimum (The slope of the function become shallower), near the optimum, the change in the value of the parameter will be virtually zero and determined as optimal depending on the number of iterations that the algorithm is specified to perform.

One step of gradient descent is implemented as:

$$\beta_j \coloneqq \beta_j - \alpha \frac{\partial}{\partial \beta_j} J\left(\beta_0,\ \ldots,\ \beta_j\right)$$

The β_j parameter is assigned a new value equal to itself, subtracted α , which is the learning rate multiplied by the partial derivative $\frac{\partial}{\partial \beta_j}$of the cost function $J(\beta_0,\ \ldots,\ \beta_n)$ with respect to the parameter β_j.

The derivative of a function defines its gradient at a particular point. Therefore, derivative with respect to one of the dimensions alters the parameter accurately within its dimension. The partial derivative is:

$$\frac{\partial}{\partial \beta_j} J(\beta_0,\ \ldots,\ \beta_n) = \frac{\partial}{\partial \beta_j} \frac{1}{2m} \sum_{i=0}^{m} \left(h_{\beta_0,\ldots,\beta n}\left(x_0^i, \ldots,\ x_n^i\right) - y^i\right)^2$$

By the chain rule, this simplifies to:

$$\frac{\partial}{\partial \beta_j} J\left(\beta_0,\ \ldots,\ \beta_j\right) = 2 \cdot \frac{1}{2m} \sum_{i=0}^{m} \left(h_{\beta_0,\ldots,\beta_n}\left(x_0^i, \ldots,\ x_n^i\right) - y^i\right) \cdot \frac{\partial}{\partial \beta_j} \sum_{i=0}^{m} \left(\beta_0 x_0^i + \beta_1 x_1^i + \cdots + \beta_n x_n^i - y^i\right)$$

$$= \frac{1}{m} \sum_{i=0}^{m} \left(h_{\beta_0,\ldots,\beta_n}\left(x_0^i, \ldots,\ x_n^i\right) - y^i\right) \cdot \frac{\partial}{\partial \beta_j} \sum_{i=0}^{m} \left(\beta_0 x_0^i + \beta_1 x_1^i + \cdots + \beta_n x_n^i - y^i\right)$$

The partial derivate of all terms is going to be zero, except for the term corresponding to j, which would be just x_j^i, therefore:

$$\frac{\partial}{\partial \beta_j} J\left(\beta_0,\ \ldots,\ \beta_j\right) = \frac{1}{m} \sum_{i=0}^{m} \left(h_{\beta_0,\ldots,\beta_n}\left(x_0^i, \ldots,\ x_n^i\right) - y^i\right) \cdot x_j^i$$

Therefore:

$$\beta_j \coloneqq \beta_j - \alpha \frac{1}{m} \sum_{i=0}^{m} \left(h_{\beta_0,\dots,\,\beta_n}\left(x_0^i, \dots,\, x_n^i\right) - y^i \right) \cdot x_j^i$$

This applied to all the features of the function and repeated until convergence, until finding optimal values for the parameters.

The cost function for two parameters in figure1.0 can be visualized in terms of its contours. The direction of steepest descent is always orthogonal to the contours of the function, however gradient descent does not take the most optimal path, as the weights and biases are trained, illustrated in figure3.0.

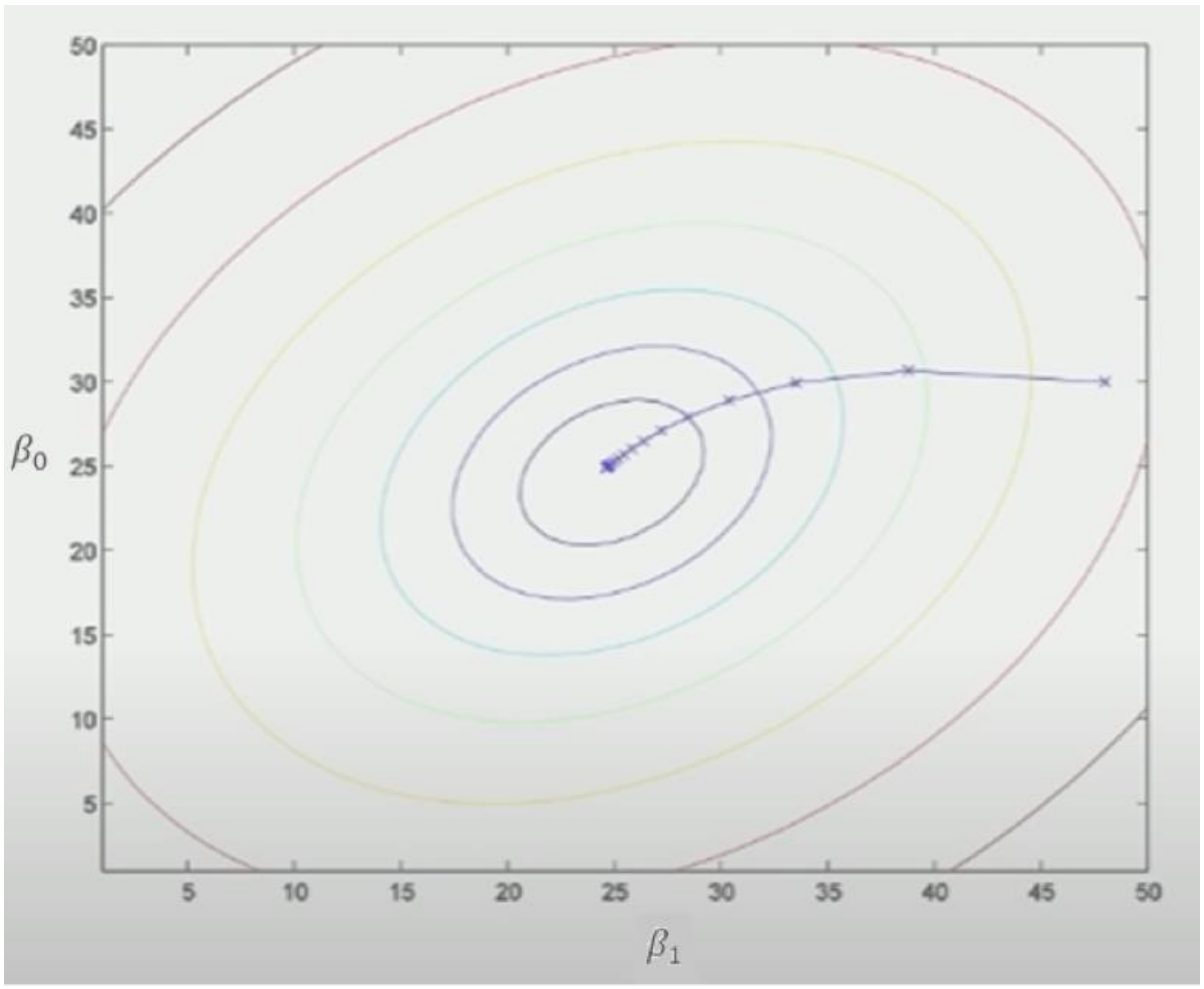


Figure 3.0, The path of the gradient descent algorithm across the contour of the mean square error function for two parameters, the concentric circles are decreasing in value closer to the center(Andrew Ng)

The learning rate α , must be set so that the algorithm finishes efficiently. As the parameters get closer to optimal, the value subtracted from the parameters gets smaller respectively as the value of the derivative decreases numerically. When the hyperparameter is initialized to a larger value, there is a possibility that it will overshoot, the steps will be too large and it will run past the global minimum and on the contrary if it is too small, the algorithm will be slow as the step size will be very small. In practice this parameter is initialized to different values to pick the most efficient value. A sample implementation of the gradient descent algorithm is shown using pseudo code:

```
Repeat until convergence {
    For j = 0 to m{
        For all parameters separately:
            $\beta_j := \beta_j - \alpha\left(h_{\beta_0,\dots,\beta_n}(x_0^i, \; \dots, \; x_n^i)\right) \cdot x_j^i$
        }
  }
```

The full name of the algorithm is batch gradient descent as it is the mean value calculated from all the training examples within each iteration. The main disadvantage of this is that in the era of big data, where the number of training examples can reach even 100 of millions.

Companies and firms often work with big data sets of several terabytes in size; therefore, efficiency and speed are vital aspect to consider. More efficiently the series of three steps should be implemented using linear algebra. This way data storage and data manipulation becomes much easier and efficient, as all the input variables can be processed simultaneously within one mathematical operation, instead of iteratively applying the set of steps using individual inputs.

Effective approach to machine learning algorithms

In this approach, the iterative implementation is completely omitted, and instead allows for convergence in just one step. This is called the normal equation, and it only works for linear regression. This is derived in the following steps.

The partial derivative (∇) of $J(\beta)$ with respect to β , where β is now an n-dimensional vector or $nx1$ matrix, where $\beta \in \mathbb{R}^n$(Andrew Ng), shown below.

$$\nabla_\beta J(\beta) = \begin{bmatrix} \frac{\partial J}{\partial \beta_0} \\ \frac{\partial J}{\partial \beta_1} \\ \vdots \\ \vdots \\ \frac{\partial J}{\partial \beta_n} \end{bmatrix}$$

The gradient of the function for any parameter must be zero, in other words the derivative of the function with respect to any of the parameters must be minimized, as it is the lowest point of the function where all the parameters have optimal values, therefore:

$$\nabla_\beta J(\beta) \overset{\text{set}}{=} \vec{0}$$

Taking the derivative with respect to β , β is a vector, and setting it equal to zero, thence solving for the value of the parameter will result in the derivative equaling zero.

So $J(\beta)$ maps from a vector to a real number, therefore taking the derivative with respect to each parameter in the vector allows you to derive a formula which allows for arriving at the local minimum of the cost function in just one step.

A new matrix X, called the design matrix is defined as following with all the training examples stacked in rows. The training examples were column vectors, so they are transposed(rows are interchanged with columns), making them into $1xm$ matrices.

$$X = \begin{bmatrix} --(x^0)^T-- \\ --(x^1)^T-- \\ \vdots \\ \vdots \\ --(x^m)^T-- \end{bmatrix}$$

It is subsequently multiplied by the parameter vector, to obtain a new vector containing the predictions for all training examples, therefore by matrix-vector multiplication. It is essentially the same procedure as obtaining the hypothesis function; however the advantage of this method is that it results in obtaining all the predictions for each set of examples in one calculation, resulting in prediction vector.

$$X \cdot \beta = \begin{bmatrix} --(x^0)^T-- \\ --(x^1)^T-- \\ \vdots \\ \vdots \\ --(x^m)^T-- \end{bmatrix} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \vdots \\ \beta_n \end{bmatrix}$$

$$= \begin{bmatrix} --(x^0)^T\beta-- \\ --(x^0)^T\beta-- \\ \vdots \\ \vdots \\ --(x^m)^T\beta-- \end{bmatrix} = \begin{bmatrix} h_\beta(x^1) \\ h_\beta(x^2) \\ \vdots \\ \vdots \\ h_\beta(x^m) \end{bmatrix}$$

This is because:

$$x_0^0\beta_0 + x_1^0\beta_1 + \cdots + x_n^0\beta_n = h_\beta(x^0)$$

The prediction is just the linear function of all the features for a particular training example. Subsequently the vector with the corresponding outputs is defined:

$$\vec{y} = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ \vdots \\ y^m \end{bmatrix}$$

Consequently, the output vector is subtracted from the prediction vector, like the mean squares error function, however the sum of the squares is implemented in the subsequent step:

$$X\beta - \vec{y} = \begin{bmatrix} h_\beta(x^1) - y^1 \\ h_\beta(x^2) - y^2 \\ \vdots \\ \vdots \\ h_\beta(x^m) - y^m \end{bmatrix}$$

To understand the sum of squares, first let's consider a vector $\mathcal{Z}$ transposed multiplied by itself (Not transposed), yields in the sum of squares of its components(Abhishek Bhatia):

$$z^T z = \sum_i z^2$$

This is because:

$$Z^T Z = [z_1, \quad z_2, \dots, \quad z_n] \cdot \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ \vdots \\ z_n \end{bmatrix} = [(z_1)^2, (z_2)^2, \dots, (z_n)^2]$$

While it is important to remember that the order does matter as the opposite order results in a nxn matrix: (Ng and Ma)

$$ZZ^T = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ \vdots \\ z_n \end{bmatrix} \cdot [z_1, \quad z_2, \dots, \quad z_n] = \begin{bmatrix} (z_1)^2 & z_1 z_2 & \cdots & z_1 z_n \\ z_2 z_1 & (z_2)^2 & \cdots & z_2 z_n \\ \vdots & \vdots & \ddots & \\ z_n z_1 & z_n z_2 & \cdots & (z_n)^2 \end{bmatrix}$$

Therefore, the cost function is expressed as the latter:

$$J(\beta) = \frac{1}{2m} (X\beta - y)^T (X\beta - y)$$

$$\because (X\beta - y)^T (X\beta - y) = \left[h_\beta(x^1) - y^1,\ h_\beta(x^2) - y^2, \dots, h_\beta(x^m) - y^m\right] \cdot \begin{bmatrix} h_\beta(x^1) - y^1 \\ h_\beta(x^2) - y^2 \\ \vdots \\ \vdots \\ h_\beta(x^m) - y^m \end{bmatrix}$$

$$= \left[(h_\beta(x^1) - y^1)^2,\ (h_\beta(x^2) - y^2)^2\ , \ldots , (h_\beta(x^m) - y^m)^2\ \right]$$

This is, because it yields in the sum of squares of the matrix's components, so, the derivative of the cost function is:

$$\nabla_\beta J(\beta) = \nabla_\beta \frac{1}{2m}\left[(h_\beta(x^1) - y^1)^2,\ (h_\beta(x^2) - y^2)^2\ , \ldots , (h_\beta(x^m) - y^m)^2\ \right] =$$

$$\nabla_\beta J(\beta) = \nabla_\beta \frac{1}{2m}(X\beta - y)^T(X\beta - y)$$

$$= \frac{1}{2m}\nabla_\beta (X^T\beta^T - y^T)(X\beta - y)$$

Further expanding it:

$$= \frac{1}{2m}\nabla_\beta [X^T\beta^T X\beta - \beta^T X^T y - y^T X\beta + y^T y]$$

This is the partial derivative with respect to all the parameters in the matrix, for each training example, so the final step is to take the derivative of each of the four components in the expansion with respect to the parameter. This requires more knowledge on matrix derivatives.

Matrix Derivatives

For a function $f\colon \mathbb{R}^{mxn} \to \mathbb{R}$ mapping from m by n matrix to real number, the partial derivative with respect to A is defined as:

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix}$$

Therefore, the gradient itself $\nabla_A f(A)$ is a matrix of the shape mxn, where its (i,j) element is $\partial f / \partial f_{ij}$(Ng and Ma). For instance, if $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ is a $2x2$ matrix and the function $f\colon \mathbb{R}^{2x2} \to \mathbb{R}$ is given by:

$$f(A) = \frac{3}{2}A_{11} + 5(A_{12})^2 + A_{21}A_{22}$$

Then,

$$\nabla_A f(A) = \begin{bmatrix} \frac{3}{2} & 10A_{12} \\ A_{22} & A_{21} \end{bmatrix}$$

It is important to introduce the trace operator defined as the sum of the diagonal entries of a square matrix nxn:

$$trA = \sum_{i=1}^{n} A_{ii}$$

An important property of the trace operator is that $trAB = trBA$, where A and B are matrices (Duchi).

Furthermore:

$$trABC = trCAB = trBCA,$$

$$trABCD = trDABC = trCDAB = trBCDA$$

For the following properties it is assumed that A and B are square matrices:

$$trA = trA^T$$

$$tr(A + B) = trA + trB$$

$$traA = atrA, where\ a\ is\ a\ scalar.$$

Following there are important facts about matrix derivatives (Duchi):

1. $\nabla_A \mathrm{tr}\ AB = B^T$
2. $\nabla_{A^T} f(A) = (\nabla_A f(A))^T$
3. $\nabla_A \mathrm{tr}\ ABA^T C = CAB + C^T AB^T$
4. $\nabla_A |A| = |A|(A^{-1})^T$

For the first fact, a fixed matrix is defined as $B \in \mathbb{R}^{nxm}$. A function $f(A) = trAB$ is defined for $f: \mathbb{R}^{mxn} \rightarrow \mathbb{R}$. This definition is coherent, since $A \in \mathbb{R}^{mxn}$, then AB is a square matrix and the trace operator can be applied(Ng and Ma). Applying the definition of matrix derivatives discussed in the beginning of this section $\nabla_A f(A)$, which itself is an mxn matrix. Thus, equation1 states that the (i, j)-entry of this matrix will be given by the (i, j)-entry of B^T, or equivalently B_{ji}. The following equations 1-3 are thus relatively simple to derive.

Coming back to the derivative of $J(\beta)$, recall,

$$\nabla_\beta J(\beta) \ = \frac{1}{2m} \nabla_\beta [X^T \beta^T X\beta - \beta^T X^T y - y^T X\beta + y^T y]$$

To minimize it, combining equations 1 and 3:

$$\nabla_{A^T} \mathrm{tr}\ ABA^T C = B^T A^T C^T + BA^T C \text{—}(5)$$

Therefore,

$$= \frac{1}{2m} \nabla_\beta \mathrm{tr}\ (\beta^T X^T X\beta - \beta^T X^T \vec{y} - \vec{y}^T X\beta + \vec{y}^T \vec{y})$$

$$= \frac{1}{2m} \nabla_\beta (\mathrm{tr}\ \beta^T X^T X\beta - 2\mathrm{tr}\ \vec{y}^T X\beta)$$

$$= \frac{1}{2m} (X^T X\beta + X^T X\beta - 2X^T \vec{y})$$

Finally, the derivate is set equal to zero to minimize the cost function, and optimize the values of the parameters: (Ng and Ma)

$$\frac{1}{m}[X^T X\beta - X^T y] \overset{\text{set}}{=} \vec{0}$$

Therefore, the Normal equation is derived as:

$$X^T X\beta = X^T y$$

So, the optimal value of the parameter is:

$$\beta = (X^T X)^{-1} X^T y$$

Therefore, the vector with optimal parameters of the function can be obtained in one step, making the process much more efficient in comparison to the iterative approach.

A question which arises is, what happens when X is not invertible, that usually implies that there are redundant features; the features are linearly dependent; however using pseudo inverse, the correct answer can still be obtained, as the new matrix is a “generalization of the matrix inverse when the matrix may not be invertible”(Heckert), however if the features are linearly dependent, most likely means that one features was repeated twice, and this can be fixed by localizing the repeated features, to remove the duplicates, to correct the problem.

This approach allows for a much more compact code and for faster and less computationally heavy training of the model. Resulting in a computationally and spatially more efficient process.

This implementation of linear regression is a common technique used in machine learning that makes sense to be implemented in these set of steps. This can be rationally explained using a probabilistic approach.

Probabilistic Interpretation

The solution to the previously explained regression problem, relies on the implementation of linear regression, cost function and the gradient descent algorithm, this is a reasonable choice explained in this section through a set of probabilistic assumptions. It is derived through a set of natural steps. It is assumed that the target variables and the inputs are related via this equation (Ng and Ma):

$$y^i = \beta^T x^i + \epsilon^i$$

Assuming that $XX^T$is an invertible matrix(usually checked before the calculation), if the number of linearly independent examples is less than the number of features or the features are not linearly independent then $X^T X$ is not invertible, however this can be fixed with several methods such as the pseudo matrix discussed previously.

ϵ^i, is an error term that captures either unmodeled effects, for example, features that are left out of the regression despite being very pertinent to the prediction, or any random noise(Hamza).

A consequent assumption of ϵ^i is that it is distributed IID (independently and identically distributed) according to a Gaussian distribution (also called a normal distribution) with mean zero and some variance σ^2. In statistics, Gaussian Distribution is a continuous probability distribution for a real-values random variable. It is represented as (Hamza):

$$f(\mathrm{x}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\mathrm{x}-\mu)^2}{2\sigma^2}\right)$$

Where μ is the mean or expectation of the distribution, also the median and the mode; there are many types of means utilized (“Arithmetic Mean | Britannica”):

Arithmetic mean: Sum of all number divided by the number of all numbers, expressed as:

$$x(line\ on\ top) = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Weighted Arithmetic mean, which combines the average values from different sized samples of the same population(Gish, Ng, and Ma):

$$x = \frac{\sum_{i=1}^{n} w_i X(Top - line)_i}{\sum_{i=1}^{n} w_i}$$

Where $\mathrm{X}(\mathrm{top} - \mathrm{line})_i$, and w_i are mean and size of sample I respectively, in other applications, measure for reliability of the influence upon the mean by the respective values (Ng and Ma).

The weighted arithmetic mean is usually chosen. The expected value (EV) is a generalization of the weighted average also called the first moment, or the expecting. Simply put, EV represents the average of many individual outcomes from a random variable that are chosen independently. For a random variable with a finite number of independent outcomes, its expected value (EV) is calculated by weighing each possible outcome. If there's a continuum of potential outcomes, the EV is determined through integration (Ng and Ma).

The σ term is the standard deviation (SD), which is the measure of the quantity of variation/Dispersion of a set of values, for instance, a low SD implies that the values are close to the mean (Also called the EV of a set.), and similarly a high SD implies that the values are spread over a wider range.

The σ squared, consequently is the variance of the distribution. The squared deviation from the mean of a random variable or square of the standard deviation. The variance is the measure of dispersion, which is the measure of how far a set of number is spread out from their average value. It is the second central moment of distribution.

A random variable with Gaussian distribution is said to be normally distributed, Notation:

$$N(N, \sigma^2)$$

Therefore, the assumption can be expressed as $\epsilon^i \sim N(0, \sigma^2)$, thus the density of ϵ^i, can be expressed as:

$$p\left(\epsilon^{(i)}\right) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(\epsilon^{(i)}\right)^2}{2\sigma^2}\right)$$

Consequently, this implies that:

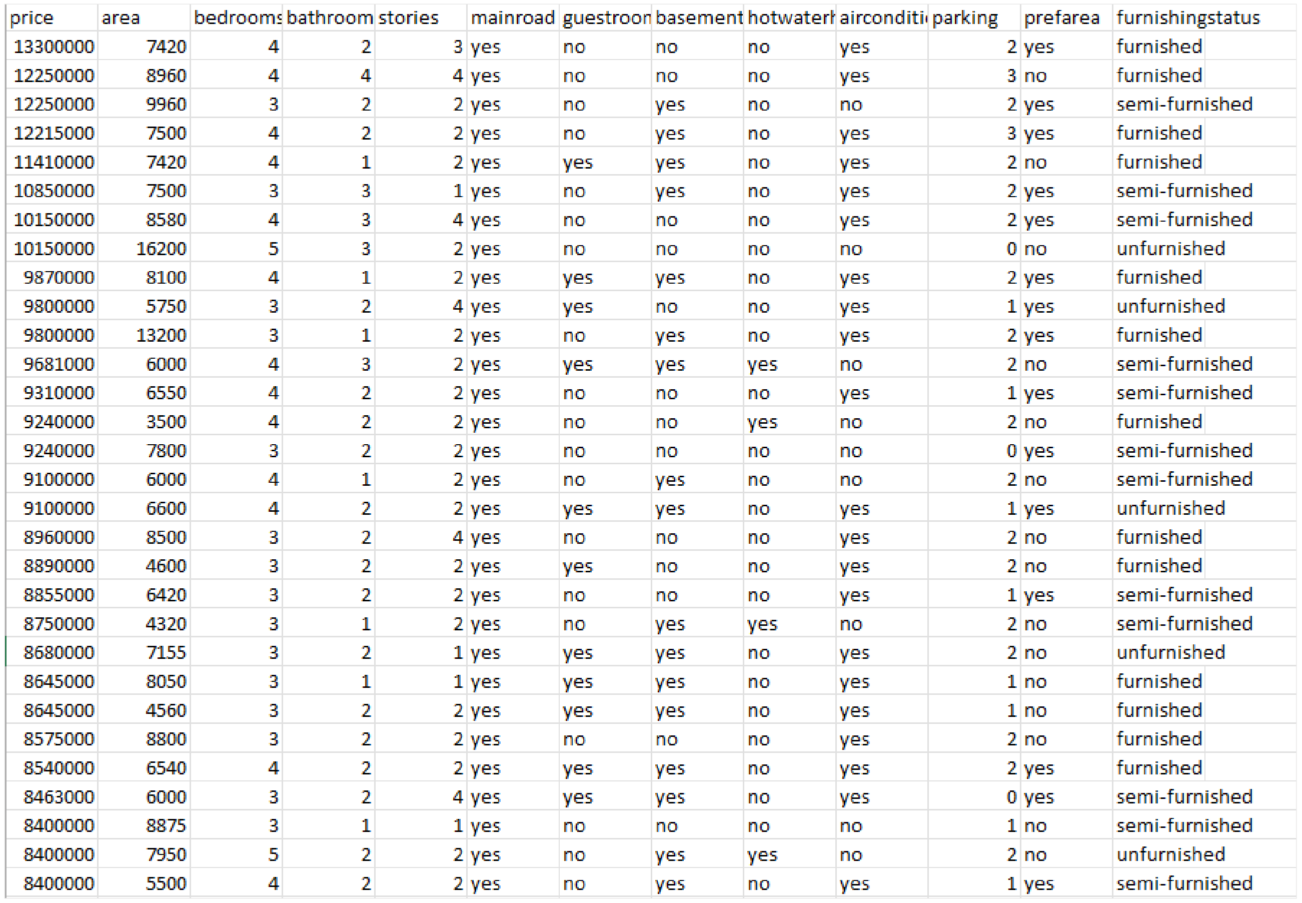
$$p\left(y^{(i)} \mid x^{(i)}; \theta\right) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right)$$

By using this notation $p\left(y^i \middle| x^i; \beta\right)$, x^i is parameterized by β, and it is a distribution of y^i. It should not condition on β $(p\left(y^i \middle| x^i; \beta\right))$, since β is not a random variable(Ng and Ma).

This way the mean square error function is derived using a very natural approach.

Analysis

To comprehensively examine the significance of machine learning models, the comparative advantage of being able to train on any number of input variables, and the effectiveness of the mathematics constituting the machine learning algorithms, a linear regression algorithm was implemented on a housing price data set, sample of which is shown in table 1.



price	area	bedrooms	bathroom	stories	mainroad	guestroon	basement	hotwaterl	airconditi	parking	prefarea	furnishingstatus
13300000	7420	4	2	3	yes	no	no	no	yes	2	yes	furnished
12250000	8960	4	4	4	yes	no	no	no	yes	3	no	furnished
12250000	9960	3	2	2	yes	no	yes	no	no	2	yes	semi-furnished
12215000	7500	4	2	2	yes	no	yes	no	yes	3	yes	furnished
11410000	7420	4	1	2	yes	yes	yes	no	yes	2	no	furnished
10850000	7500	3	3	1	yes	no	yes	no	yes	2	yes	semi-furnished
10150000	8580	4	3	4	yes	no	no	no	yes	2	yes	semi-furnished
10150000	16200	5	3	2	yes	no	no	no	no	0	no	unfurnished
9870000	8100	4	1	2	yes	yes	yes	no	yes	2	yes	furnished
9800000	5750	3	2	4	yes	yes	no	no	yes	1	yes	unfurnished
9800000	13200	3	1	2	yes	no	yes	no	yes	2	yes	furnished
9681000	6000	4	3	2	yes	yes	yes	yes	no	2	no	semi-furnished
9310000	6550	4	2	2	yes	no	no	no	yes	1	yes	semi-furnished
9240000	3500	4	2	2	yes	no	no	yes	no	2	no	furnished
9240000	7800	3	2	2	yes	no	no	no	no	0	yes	semi-furnished
9100000	6000	4	1	2	yes	no	yes	no	no	2	no	semi-furnished
9100000	6600	4	2	2	yes	yes	yes	no	yes	1	yes	unfurnished
8960000	8500	3	2	4	yes	no	no	no	yes	2	no	furnished
8890000	4600	3	2	2	yes	yes	no	no	yes	2	no	furnished
8855000	6420	3	2	2	yes	no	no	no	yes	1	yes	semi-furnished
8750000	4320	3	1	2	yes	no	yes	yes	no	2	no	semi-furnished
8680000	7155	3	2	1	yes	yes	yes	no	yes	2	no	unfurnished
8645000	8050	3	1	1	yes	yes	yes	no	yes	1	no	furnished
8645000	4560	3	2	2	yes	yes	yes	no	yes	1	no	furnished
8575000	8800	3	2	2	yes	no	no	no	yes	2	no	furnished
8540000	6540	4	2	2	yes	yes	yes	no	yes	2	yes	furnished
8463000	6000	3	2	4	yes	yes	yes	no	yes	0	yes	semi-furnished
8400000	8875	3	1	1	yes	no	no	no	no	1	no	semi-furnished
8400000	7950	5	2	2	yes	no	yes	yes	no	2	no	unfurnished
8400000	5500	4	2	2	yes	no	yes	no	yes	1	yes	semi-furnished

Table 1, Sample of the dataset used (Yasser H)

There are twelve explanatory variables. Inputs such as "yes" or "no" were converted into numerical values of 1 and 0 respectively, and inputs such as furnished, unfurnished and semi-furnished were converted into values of 2, 0, and 1 respectively.

The data was then divided into two matrices, price, and explanatory variable matrix. Consequently a linear regression machine learning models were constructed explicitly using mathematics and using TensorFlow (The models is presented in Appendix1).

The model was trained for 1000 iterations for three increasing sets of input explanatory variables and the final error of the model was recorded. The results were recorded in table 2.

Input variables	Error	r^2
Area	2488861458432.00	0.29
Area, Bedrooms, Bathrooms, Stories	1621756739584.00	0.54
Area, bedrooms, bathrooms, stories mainroad, guestroom, basement, hotwaterheating, airconditioning, parking, prefarea, furnishingstatus	1117238460416.00	0.68

Table 2, Accuracy metric results for the linear regression model. Evidently the accuracy increases as the number of input explanatory variables increases.

Evidently, the error clearly decreases as more variables are included. That being said, it is not clear whether certain input variables will reduce the error as there may not be any linear relationship between the price and them, therefore individual input variables can be plotted against the price and analyzed whether there is a correlation, this is called bivariate analysis (Shah), an example of which is illustrated in Figure4.

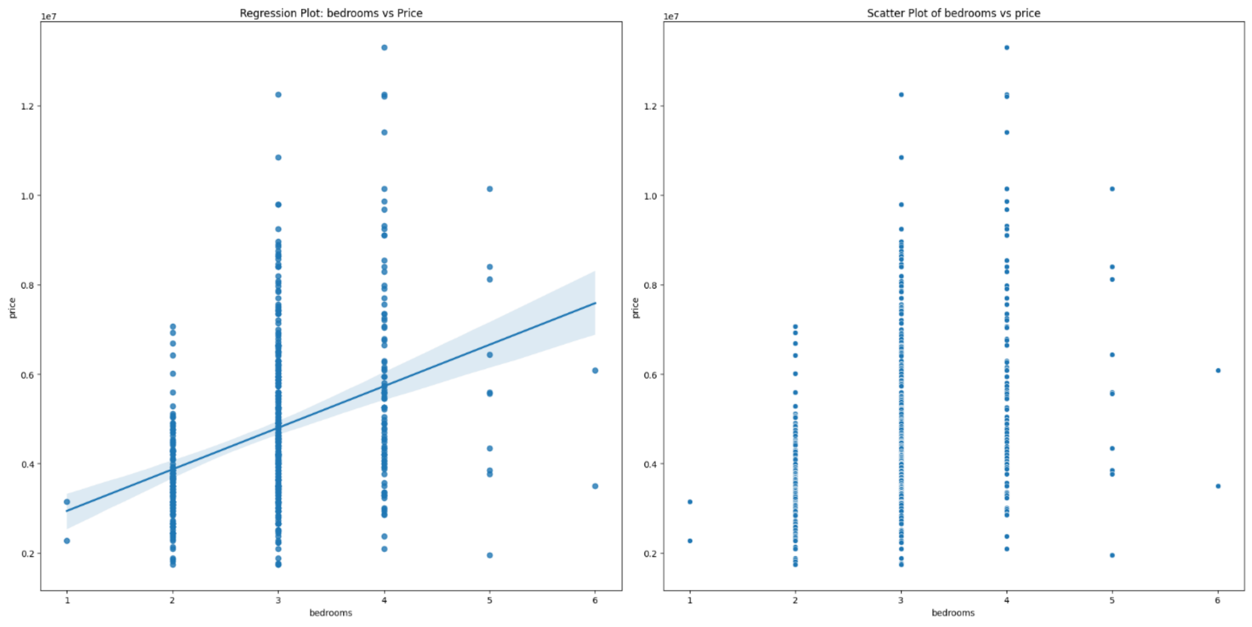


Figure 4, Sample of bivariate analysis for number of bedrooms vs price from the housing price dataset. Plotted using Seaborn.

Thus, training the model on the specific variables can reduce the error even more to 1008890236989.60, and increase the r^2 value to 0.72.

For illustrative purposes the training for one explanatory variable is presented as with an incre asing number of explanatory variables, visualization becomes increasingly more difficult.
Figure5 presents the optimal plot of the optimized function through 1000 iterations of gradient descent algorithm.

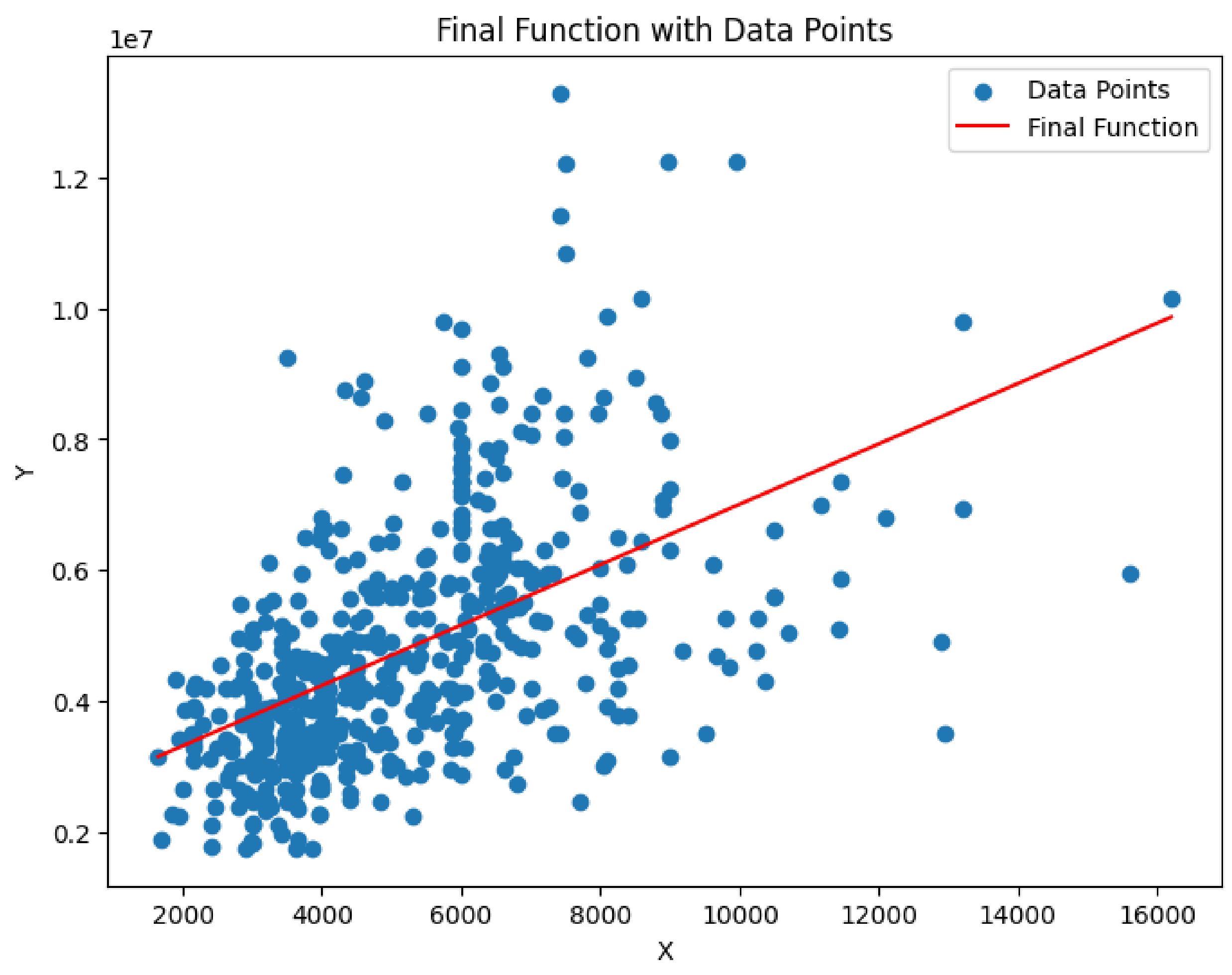


Figure 5, Function of the line of best fit plotted on housing prices vs house area plot, after 1000 iterations of the gradient descent algorithm to optimize the values of the function. Plotted using MatPlotLib.

Consequently, for the purposes of visualization the mean squared error function J(B), was plotted in a hyperplane, to visualize the error with respect to the different combinations of the two parameters. This is shown in Figure6.

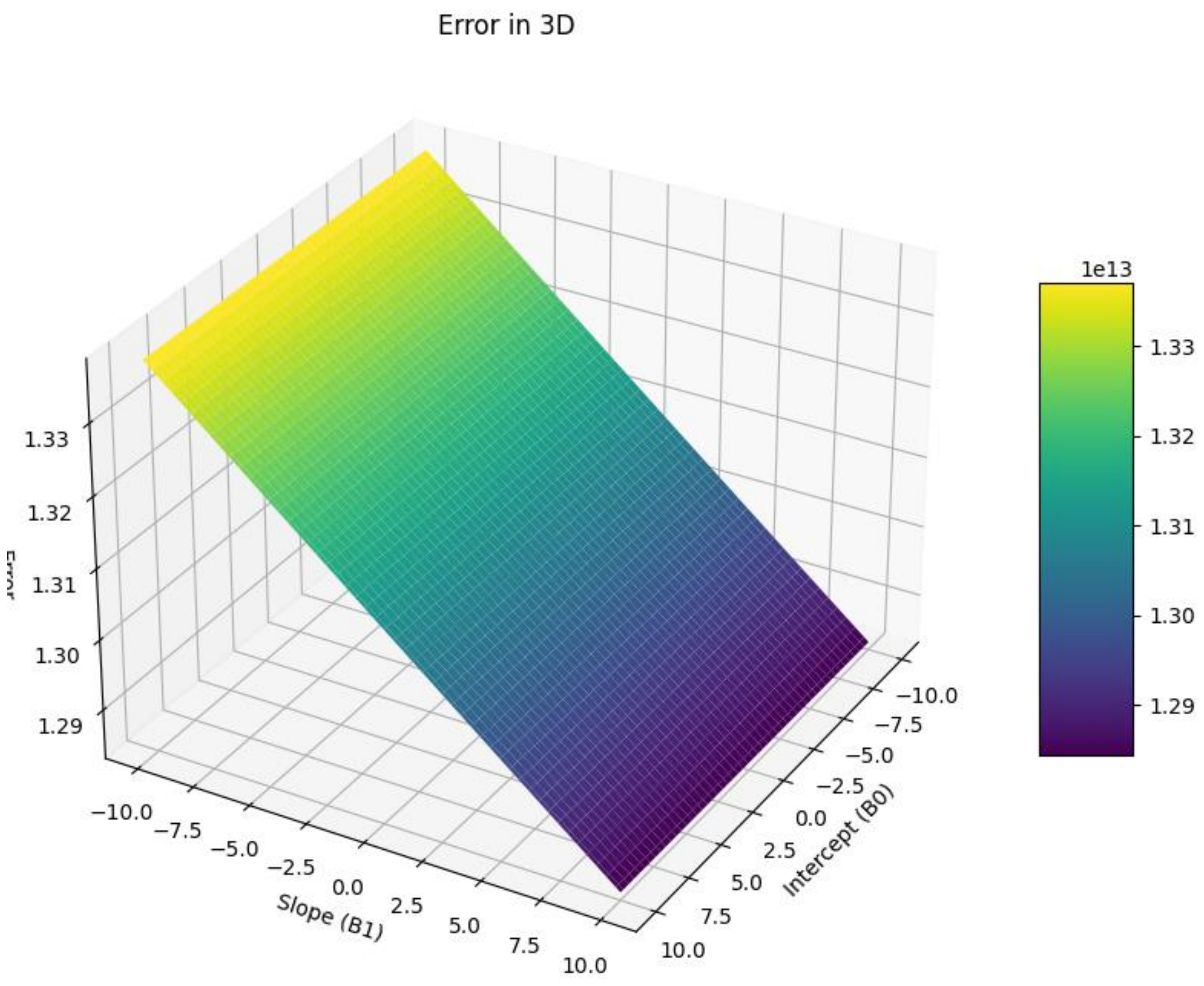


Figure 6, The square error function plotted for the two parameters of the function and the respective error. The general shape of the function is convex due to the nature of the mean square error function. Plotted using MatPlotLib.

The results correctly support the theory by effectively optimizing the parameters of the function to yield in a function of best fit that follows the trend in the data, and effectively generates a mean square error function, for which the parameters are optimized at the global minimum. It is evident from the plot that the error can be minimized, moreover by inclusion of more input variables, the error function only exists in the realm of numbers, while it cannot be explicitly visualized.

Conclusion

The mathematics behind machine learning linear regression models is effectively explored and evaluated through the simple iterative approach, through three core steps (Hypothesis function, mean square error function, and Gradient descent) to the algorithm and proven why it is a natural approach using the probabilistic assumptions to derive the mean square error function. Furthermore, the efficient approach to training the model is presented with the same depth of analysis, using linear algebra to simplify the calculations and make the process significantly more efficient, as seen both computationally and spatially Moreover the analysis extends to analyzing a dataset of varying number of input explanatory variables to optimize the parameters of the function. Moreover, it proves that generally as the number of input variables increases the model can train better, minimize the error, and yield in more accurate predictions.

Thus is it is shown, linear regression machine learning models can be used to effectively model relationships between variables that follow a linear trend across all the explanatory variables; however, many data sets utilized in real world situations, such as fuel prices, natural language processing(nlp), or image recognition, don’t follow a linear trend, therefore other models are employed to suit the type of dataset in question. That being said, one of the main limitations of the analysis is the employment of solely linear regression, as many other models such as artificial neural networks combined can be employed, especially when considering the condition of the house; entire reports can be analysed using nlp, which in connection with other parameters, potentially images of the house using Convolutional neural networks(CNNs), parameters previously discussed, and other parameters in connection with the broader picture, like considering the prices based on current world events, prices of materials, or laws and policies, can potentially yield in considerably more accurate results even generalizing for potential outliers within the dataset.

One unifying aspect of the latter discussion is data, more precisely human labelled data, which may be a problematic in many cases, as such data may not be available, data might be missing, or data will need to be labelled specifically for the current task.

Nevertheless, the true beauty of this concept is the very fact that machine learning models are often trained on an immense number of explanatory variables, extending past three dimensions and becoming impossible to visualize, yet making absolute sense mathematically.

Appendix 1

Linear Regression Machine learning model implementation using the mathematical approach.

```
import numpy as np
import math
import pandas as pd
from numpy import random
from sympy import symbols, diff
import numpy as np
from matplotlib import pyplot as plt

from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

mapping = {
    'no': 0,
    'yes': 1,
    'furnished': 2,
    'semi-furnished': 1,
    'unfurnished': 0}

mapping2 = {
    "no": 1,
    "yes": 0
    }

df =
pd.read_csv(r"/content/drive/MyDrive/MachineLearning/Data_Sets/Housing777.csv")
print(df)

df = df.replace(mapping)
df['mainroad'] = df['mainroad'].replace(mapping2)

print(df)

Input = np.array(df)

Y, X0, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11  = np.hsplit(Input, [1, 2, 3,
4, 5, 6, 7, 8, 9,10,11,12])

print(f"X0 shape: {X0.shape}")
print(f"X1 shape: {X1.shape}")
print(f"X2 shape: {X2.shape}")
print(f"X3 shape: {X3.shape}")
print(f"X4 shape: {X4.shape}")
print(f"X5 shape: {X5.shape}")
print(f"X6 shape: {X6.shape}")
print(f"X7 shape: {X7.shape}")
print(f"X8 shape: {X8.shape}")
print(f"X9 shape: {X9.shape}")
print(f"X10 shape: {X10.shape}")
print(f"X11 shape: {X11.shape}")
print(f"Y shape: {Y.shape}")

X,Y = X0,Y

X=X.T

Y=Y.T

X.shape

x_train = np.array(X).T #One dimensional data set  array size:
y=Y
m = y.size

w0 = random.randint(100)
w1 = random.randint(100)

alpha = 0.00001
alpha = 0.00000001

print(x_train.shape, Y.shape)

th1 = []
th1.extend(range(-10, 10, 1))
th2 = []
th2.extend(range(-10, 10, 1))

def hypothesis(x_train,y,w0,w1):
  y_predict = (w0 + w1*x_train)
  #print(y_predict.shape)
  return y_predict

print(np.mean((hypothesis(x_train,y,w0,w1) - y.T)**2))

def error_function(y_predict, y):
  j = (np.mean( (y_predict - y.T)**2))/2
  return j

error_function(hypothesis(x_train, y, w0, w1),y)

plt.title("Size vs Price")
plt.xlabel("Size(SqFt")
plt.ylabel("Price")
plt.scatter(x_train,y.T)
plt.show()

def gradient_descent(w0, w1):
  for i in range(1000000):
    #print(w0, w1)
    #tw0 = w0 - alpha*(diff(error_function(hypothesis(x_train,y,w0,w1), y), w0))
    #tw1 = w1 - alpha*(diff(error_function(hypothesis(x_train,y,w0,w1), y), w1))
    tw0 = w0 - alpha*(np.mean( (hypothesis(x_train,y,w0,w1) - y.T) ))
    tw1 = w1 - alpha*(np.mean( (hypothesis(x_train,y,w0,w1) - y.T) * (x_train) ))

    if tw0 == w0 and tw1 == w1:
      break
    else:
      #Update
      w0 = tw0
      w1 = tw1

    if i%100==0:
      print(error_function(hypothesis(x_train,y,w0,w1), y))
  return w0, w1

#174973.37846968079 353604938.9206552
w0, w1 = gradient_descent(w0, w1)
print(w0, w1)

o =  w1 * x_train + w0
plt.xlabel("Size(SqFt")
plt.ylabel("Price")
plt.scatter(x_train,y.T)
plt.plot(x_train, o)
plt.show()

from mpl_toolkits.mplot3d import Axes3D

def plot_error_3d(x_train, y):
    w0_vals = np.linspace(-10, 10, 100)
```

```python
    w1_vals = np.linspace(-10, 10, 100)
    w0_vals, w1_vals = np.meshgrid(w0_vals, w1_vals)

    J_vals = np.zeros((len(w0_vals), len(w1_vals)))

    for i in range(len(w0_vals)):
        for j in range(len(w1_vals)):
            predictions = w0_vals[i, j] + w1_vals[i, j] * x_train
            J_vals[i, j] = np.mean((predictions - y.T) ** 2) / 2

    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(w0_vals, w1_vals, J_vals, cmap='viridis')

    ax.set_xlabel('Intercept (w0)')
    ax.set_ylabel('Slope (w1)')
    ax.set_zlabel('Error')
    ax.set_title('Error in 3D')

    fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.show()

plot_error_3d(x_train, y)

from mpl_toolkits.mplot3d import Axes3D

def plot_error_3d(x_train, y):
    w0_vals = np.linspace(-10, 10, 100)
    w1_vals = np.linspace(-10, 10, 100)
    w0_vals, w1_vals = np.meshgrid(w0_vals, w1_vals)

    J_vals = np.zeros((len(w0_vals), len(w1_vals)))

    for i in range(len(w0_vals)):
        for j in range(len(w1_vals)):
            predictions = w0_vals[i, j] + w1_vals[i, j] * x_train
            J_vals[i, j] = np.mean((predictions - y.T) ** 2) / 2

    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(w0_vals, w1_vals, J_vals, cmap='viridis')

    ax.set_xlabel('Intercept (B0)')
    ax.set_ylabel('Slope (B1)')
    ax.set_zlabel('Error')
    ax.set_title('Error in 3D')

    ax.view_init(elev=30, azim=30)

    fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.show()

plot_error_3d(x_train, y)

def plot_error_contour(x_train, y):
    w0_vals = np.linspace(-100, 100, 100)
    w1_vals = np.linspace(-100, 100, 100)
    w0_vals, w1_vals = np.meshgrid(w0_vals, w1_vals)

    J_vals = np.zeros((len(w0_vals), len(w1_vals)))

    for i in range(len(w0_vals)):
        for j in range(len(w1_vals)):
            predictions = w0_vals[i, j] + w1_vals[i, j] * x_train
            J_vals[i, j] = np.mean((predictions - y.T) ** 2) / 2

    plt.figure(figsize=(10, 8))
    plt.contourf(w0_vals, w1_vals, J_vals, levels=50, cmap='viridis')
    plt.colorbar(label='Error')
    plt.xlabel('Intercept (w0)')
    plt.ylabel('Slope (w1)')
    plt.title('Contour Plot of Error')
    plt.show()

plot_error_contour(x_train, y)

def calculate_r_squared(x_train, y, w0, w1):
    y_mean = np.mean(y)
    y_pred = w0 + w1 * x_train
    ss_total = np.sum((y - y_mean) ** 2)
    ss_residual = np.sum((y - y_pred) ** 2)

    r_squared = 1 - (ss_residual / ss_total)
    return r_squared
r_squared = calculate_r_squared(x_train, y.T, w0, w1)
print(f"R^2 Value: {r_squared}")
```

Linear Regression Machine learning model implementation, using Python and TensorFlow.

```
import numpy as np
import math
import pandas as pd
from numpy import random
from sympy import symbols, diff
import numpy as np
from matplotlib import pyplot as plt

from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/drive')

mapping = {
    'no': 0,
    'yes': 1,
    'furnished': 2,
    'semi-furnished': 1,
    'unfurnished': 0}

mapping2 = {
    "no": 1,
    "yes": 0
    }

df = df.replace(mapping)
df['mainroad'] = df['mainroad'].replace(mapping2)

print(df)

df =
pd.read_csv(r"/content/drive/MyDrive/MachineLearning/Data_Sets/Housing777.csv")
print(df)

Input = np.array(df)

Y, X0, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11  = np.hsplit(Input, [1, 2, 3,
4, 5, 6, 7, 8, 9,10,11,12])

print(f"X0 shape: {X0.shape}")
print(f"X1 shape: {X1.shape}")
print(f"X2 shape: {X2.shape}")
print(f"X3 shape: {X3.shape}")
print(f"X4 shape: {X4.shape}")
print(f"X5 shape: {X5.shape}")
print(f"X6 shape: {X6.shape}")
print(f"X7 shape: {X7.shape}")
print(f"X8 shape: {X8.shape}")
print(f"X9 shape: {X9.shape}")
print(f"X10 shape: {X10.shape}")
print(f"X11 shape: {X11.shape}")
print(f"Y shape: {Y.shape}")



import tensorflow as tf
import numpy as np

x0_train = np.array(X0)
x1_train = np.array(X1)
x2_train = np.array(X2)
x3_train = np.array(X3)
x4_train = np.array(X4)
x5_train = np.array(X5)
x6_train = np.array(X6)
x7_train = np.array(X7)
x8_train = np.array(X8)
x9_train = np.array(X9)
x10_train = np.array(X10)
x11_train = np.array(X11)
y_train = np.array(Y)

X_train = np.column_stack((x0_train, x1_train, x2_train, x3_train, x4_train,
x5_train, x6_train, x7_train, x8_train, x9_train, x10_train, x11_train))

X_train = (X_train - X_train.mean(axis=0)) / X_train.std(axis=0)
X_train = np.column_stack((np.ones(len(X_train)), X_train))

X = tf.constant(X_train, dtype=tf.float32)
y = tf.constant(y_train.reshape(-1, 1), dtype=tf.float32)

weights = tf.Variable(tf.random.normal(shape=(X.shape[1], 1), dtype=tf.float32))

#Hypothesis function
def hypothesis(X, weights):
    return tf.matmul(X, weights)

def mean_square_error(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_true - y_pred))

learning_rate = 0.01
epochs = 1000

#Gradient Descent
for i in range(epochs):
    with tf.GradientTape() as tape:
        y_pred = hypothesis(X, weights)
        loss = mean_square_error(y, y_pred)

    gradients = tape.gradient(loss, [weights])
    weights.assign_sub(learning_rate * gradients[0])

    if i % 100 == 0:
        print(f'Epoch {i}, Loss: {loss.numpy()}')

final_weights = weights.numpy()
print('Final Weights:', final_weights)

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

def plot_error_over_iterations(X_train, y_train, learning_rate=0.01,
epochs=1000):

    X_train_normalized = (X_train - X_train.mean(axis=0)) / X_train.std(axis=0)

    X_train_normalized = np.column_stack((np.ones(len(X_train_normalized)),
X_train_normalized))

    X = tf.constant(X_train_normalized, dtype=tf.float32)
    y = tf.constant(y_train.reshape(-1, 1), dtype=tf.float32)

    weights = tf.Variable(tf.random.normal(shape=(X.shape[1], 1),
dtype=tf.float32))

    #Hypothesis function
    def hypothesis(X, weights):
        return tf.matmul(X, weights)
    #Mean square error function
    def mean_square_error(y_true, y_pred):
        return tf.reduce_mean(tf.square(y_true - y_pred))

    errors = []
    iterations = []

    #Gradient Descent
    for i in range(epochs):
        with tf.GradientTape() as tape:
            y_pred = hypothesis(X, weights)
            loss = mean_square_error(y, y_pred)

        gradients = tape.gradient(loss, [weights])
        weights.assign_sub(learning_rate * gradients[0])

        errors.append(loss.numpy())
        iterations.append(i)

    #Graph
    plt.figure(figsize=(8, 6))
    plt.plot(iterations, errors, label='Error')
    plt.title('Error Over Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Error')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_error_over_iterations(np.column_stack((x0_train, x1_train, x2_train,
x3_train, x4_train, x5_train, x6_train, x7_train, x8_train, x9_train, x10_train,
x11_train)), y_train)

from sklearn.metrics import mean_squared_error, mean_absolute_error,r2_score

y_pred=hypothesis(X, weights)
print("mean squared error:" , mean_squared_error(y_train,y_pred), "\n")
print("r2_score : ", r2_score(y_train, y_pred))
```

Linear Regression Machine learning model implementation for all explanatory variables, using Python and TensorFlow.

```
import numpy as np
import math
import pandas as pd
```

```
from numpy import random
from sympy import symbols, diff
import numpy as np
from matplotlib import pyplot as plt

from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/drive')

mapping = {
    'no': 0,
    'yes': 1,
    'furnished': 2,
    'semi-furnished': 1,
    'unfurnished': 0}

mapping2 = {
    "no": 1,
    "yes": 0
    }

df = df.replace(mapping)
df['mainroad'] = df['mainroad'].replace(mapping2)

print(df)

df =
pd.read_csv(r"/content/drive/MyDrive/MachineLearning/Data_Sets/Housing777.csv")
print(df)

Input = np.array(df)

Y, X0, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11  = np.hsplit(Input, [1, 2, 3,
4, 5, 6, 7, 8, 9,10,11,12])

print(f"X0 shape: {X0.shape}")
print(f"X1 shape: {X1.shape}")
print(f"X2 shape: {X2.shape}")
print(f"X3 shape: {X3.shape}")
print(f"X4 shape: {X4.shape}")
print(f"X5 shape: {X5.shape}")
print(f"X6 shape: {X6.shape}")
print(f"X7 shape: {X7.shape}")
print(f"X8 shape: {X8.shape}")
print(f"X9 shape: {X9.shape}")
print(f"X10 shape: {X10.shape}")
print(f"X11 shape: {X11.shape}")
print(f"Y shape: {Y.shape}")



import tensorflow as tf
import numpy as np

x0_train = np.array(X0)
x1_train = np.array(X1)
x2_train = np.array(X2)
x3_train = np.array(X3)
x4_train = np.array(X4)
x5_train = np.array(X5)
x6_train = np.array(X6)
x7_train = np.array(X7)
x8_train = np.array(X8)
x9_train = np.array(X9)
x10_train = np.array(X10)
x11_train = np.array(X11)
y_train = np.array(Y)

X_train = np.column_stack((x0_train, x1_train, x2_train, x3_train, x4_train,
x5_train, x6_train, x7_train, x8_train, x9_train, x10_train, x11_train))

X_train = (X_train - X_train.mean(axis=0)) / X_train.std(axis=0)

X_train = np.column_stack((np.ones(len(X_train)), X_train))

X = tf.constant(X_train, dtype=tf.float32)
y = tf.constant(y_train.reshape(-1, 1), dtype=tf.float32)

weights = tf.Variable(tf.random.normal(shape=(X.shape[1], 1), dtype=tf.float32))

#Hypothesis function
def hypothesis(X, weights):
    return tf.matmul(X, weights)

#Mean square error function
def mean_square_error(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_true - y_pred))

learning_rate = 0.01
epochs = 1000

#Gradient descent
for i in range(epochs):
    with tf.GradientTape() as tape:
        y_pred = hypothesis(X, weights)
        loss = mean_square_error(y, y_pred)

    gradients = tape.gradient(loss, [weights])
    weights.assign_sub(learning_rate * gradients[0])

    if i % 100 == 0:
        print(f'Epoch {i}, Loss: {loss.numpy()}')

final_weights = weights.numpy()
print('Final Weights:', final_weights)

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

def plot_error_over_iterations(X_train, y_train, learning_rate=0.01,
epochs=1000):

    #Normalize the features
    X_train_normalized = (X_train - X_train.mean(axis=0)) / X_train.std(axis=0)

    X_train_normalized = np.column_stack((np.ones(len(X_train_normalized)),
X_train_normalized))

    #TensorFlow constants
    X = tf.constant(X_train_normalized, dtype=tf.float32)
    y = tf.constant(y_train.reshape(-1, 1), dtype=tf.float32)

    weights = tf.Variable(tf.random.normal(shape=(X.shape[1], 1),
dtype=tf.float32))

    #Hypothesis function
    def hypothesis(X, weights):
        return tf.matmul(X, weights)

    #Mean square error function
    def mean_square_error(y_true, y_pred):
        return tf.reduce_mean(tf.square(y_true - y_pred))

    errors = []
    iterations = []

    #Gradient descent
    for i in range(epochs):
        with tf.GradientTape() as tape:
            y_pred = hypothesis(X, weights)
            loss = mean_square_error(y, y_pred)

        gradients = tape.gradient(loss, [weights])
        weights.assign_sub(learning_rate * gradients[0])

        errors.append(loss.numpy())
        iterations.append(i)

    #Error over iterations
    plt.figure(figsize=(8, 6))
    plt.plot(iterations, errors, label='Error')
    plt.title('Error Over Iterations')
    plt.xlabel('Iterations')
    plt.ylabel('Error')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_error_over_iterations(np.column_stack((x0_train, x1_train, x2_train,
x3_train, x4_train, x5_train, x6_train, x7_train, x8_train, x9_train, x10_train,
x11_train)), y_train)

from sklearn.metrics import mean_squared_error, mean_absolute_error,r2_score

y_pred=hypothesis(X, weights)
print("mean squared error:" , mean_squared_error(y_train,y_pred), "\n")
print("r2_score : ", r2_score(y_train, y_pred))

import seaborn as sns

def bivariate_num_col(col):
    sns.set_palette("tab10")
    fig, ax = plt.subplots(1, 2, figsize=(20, 10))
    sns.regplot(x=df[col], y=df['price'], ax=ax[0])
    ax[0].set_title(f'Regression Plot: {col} vs Price')
    sns.scatterplot(x=col, y='price', data=df, ax=ax[1])
    ax[1].set_title(f'Scatter Plot of {col} vs price')


    plt.tight_layout()
    plt.show()

numeric_cols=[]
cat_cols=[]
price_col=df.price
for column in df.columns:
    if pd.api.types.is_numeric_dtype(df[column]):
        print(f"The column '{column}' is numeric.")
        numeric_cols.append(column)
    else:
        print(f"The column '{column}' is not numeric.")
        cat_cols.append(column)

for col in numeric_cols[1:]:
    print(f'Bivariate analysis between {col} and price')
    bivariate_num_col(col)
```

Works cited.

Abhishek Bhatia. "Product of a Vector and Its Transpose (Projections)." *Mathematics Stack Exchange*, 9 July 2016, math.stackexchange.com/questions/1853808/product-of-a-vector-and-its-transpose-projections. Accessed 1 Oct. 2023.

Andrew Ng, Online, Stanford. "Stanford CS229: Machine Learning Course, Lecture 1 - Andrew Ng (Autumn 2018)." *YouTube*, YouTube Video, 17 Apr. 2020, www.youtube.com/watch?v=jGwO_UgTS7I&ab_channel=StanfordOnline. Accessed 3 Jun. 2023.

Andrew Ng, Online, Stanford. "Stanford CS229: Machine Learning - Linear Regression and Gradient Descent | Lecture 2 (Autumn 2018)." *YouTube*, YouTube Video, 17 Apr. 2020, www.youtube.com/watch?v=4b4MUYve_U8&t=3771s&ab_channel=StanfordOnline. Accessed 3 Jun. 2023.

"Britannica | Arithmetic Mean." *Encyclopædia Britannica*, 2023, www.britannica.com/science/arithmetic-mean. Accessed 1 Oct. 2023

Duchi, John. *Properties of the Trace and Matrix Derivatives*. Stanford University. Accessed 21 Nov. 2023

Gish, H. (n.d.). A probabilistic approach to the understanding and training of neural network classifiers. International Conference on Acoustics, Speech, and Signal Processing. doi:10.1109/icassp.1990.115636

Ng, Andrew, and Tengyu Ma. *CS229 Lecture Notes*. 2023, cs229.stanford.edu/main_notes.pdf. Accessed 05 Nov. 2023.

Hamza, Syed Muhammad. “Probabilistic Justification for Using Specific Loss Function in Different Types of Machine….” *Analytics Vidhya*, 21 Mar. 2021, medium.com/analytics-vidhya/probabilistic-justification-for-using-specific-loss-function-in-different-types-of-machine-e60fda8146b2. Accessed 21 Nov. 2023.

Heckert, Alan. “PSEUDO INVERSE.” *Pseudo Inverse*, 21 Jan. 2009, www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/pseudinv.htm. Accessed 1 Oct. 2023.

Hodson, Timothy O. “Root-Mean-Square Error (RMSE) or Mean Absolute Error (MAE): When to Use Them or Not.” *Geoscientific Model Development*, vol. 15, no. 14, July 2022, pp. 5481–87, https://doi.org/10.5194/gmd-15-5481-2022, Accessed 5 Oct. 2023.

IBM.” *Ibm.com*, “What Is Gradient Descent? | 2023, www.ibm.com/topics/gradient-descent#:~:text=Gradient%20descent%20is%20an%20optimization,each%20iteration%20of%20parameter%20updates. Accessed 22 May. 2023.

Kumar, Ronit. “Inspirit IA, Online Lecture.” Received by Michal Borowski, 7 June 2023.

Ostmeyer, J., & Cowell, L. (2019). Machine learning on sequential data using a recurrent weighted average. Neurocomputing, 331, 281–288. doi:10.1016/j.neucom.2018.11.066. Accessed 7 Sep. 2023.

Shah, Khushi. “Exploratory Analysis Using Univariate, Bivariate, and Multivariate Analysis Techniques.” *Analytics Vidhya*, 19 Apr. 2021, www.analyticsvidhya.com/blog/2021/04/exploratory-analysis-using-univariate-bivariate-and-multivariate-analysis-techniques/. Accessed 4 Sep. 2023.

Su, Xiaogang, et al. “Linear Regression.” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 4, no. 3, Wiley-Blackwell, Feb. 2012, pp. 275–94, https://doi.org/10.1002/wics.1198. Accessed 10 Oct. 2023.

Yasser H , M . “Housing Prices Dataset.” *Www.kaggle.com*, 2001, www.kaggle.com/datasets/yasserh/housing-prices-dataset. Accessed 22 Aug. 2022.