B22EE036_PRML_PA2_Report

Programming Assignment 2

Question 1: Decision Tree

Task Outline

- **Dataset :** The Titanic dataset.
- **Overview:** The Titanic dataset is a well-known dataset that provides information about the passengers on the Titanic, a passenger ship that sunk in the North Atlantic Ocean on April 15, 1912 after colliding with an iceberg.
- **Feature Description:**

Variable	Description	Comments
PassengerId	Unique identifier for each passenger	Used in the kaggle submission
Survived	Survival status (0 = No, 1 = Yes)	Rows with NaNs used for test prediction for Kaggle
Pclass	Ticket class (1 = 1st class, 2 = 2nd class, 3 = 3rd class)	
Name	Passenger's name	It containis the title
Sex	Passenger's sex:	string "male" or "female"
Age	Passenger's age in years	
SibSp	Number of siblings and/or spouses aboard the Titanic	
Parch	Number of parents and/or children aboard the Titanic	
Ticket	Ticket code,	The companions have same ticket code
Fare	Fare corresponding to the ticket code	Repeated for passengers with the same ticket code
Cabin	Cabin label and number	Some passengers have more than one cabin some missing

https://medium.com/@praoiticica/titanic-data-cleaning-and-feature-engineering-9f122752097f

- **Dimensions :** `1309*12`
- **Objective :** To implement a decision tree classifier from scratch and train it on the Titanic dataset.

Task 1.1.1 Data Visualization

▼ Data Fetching:

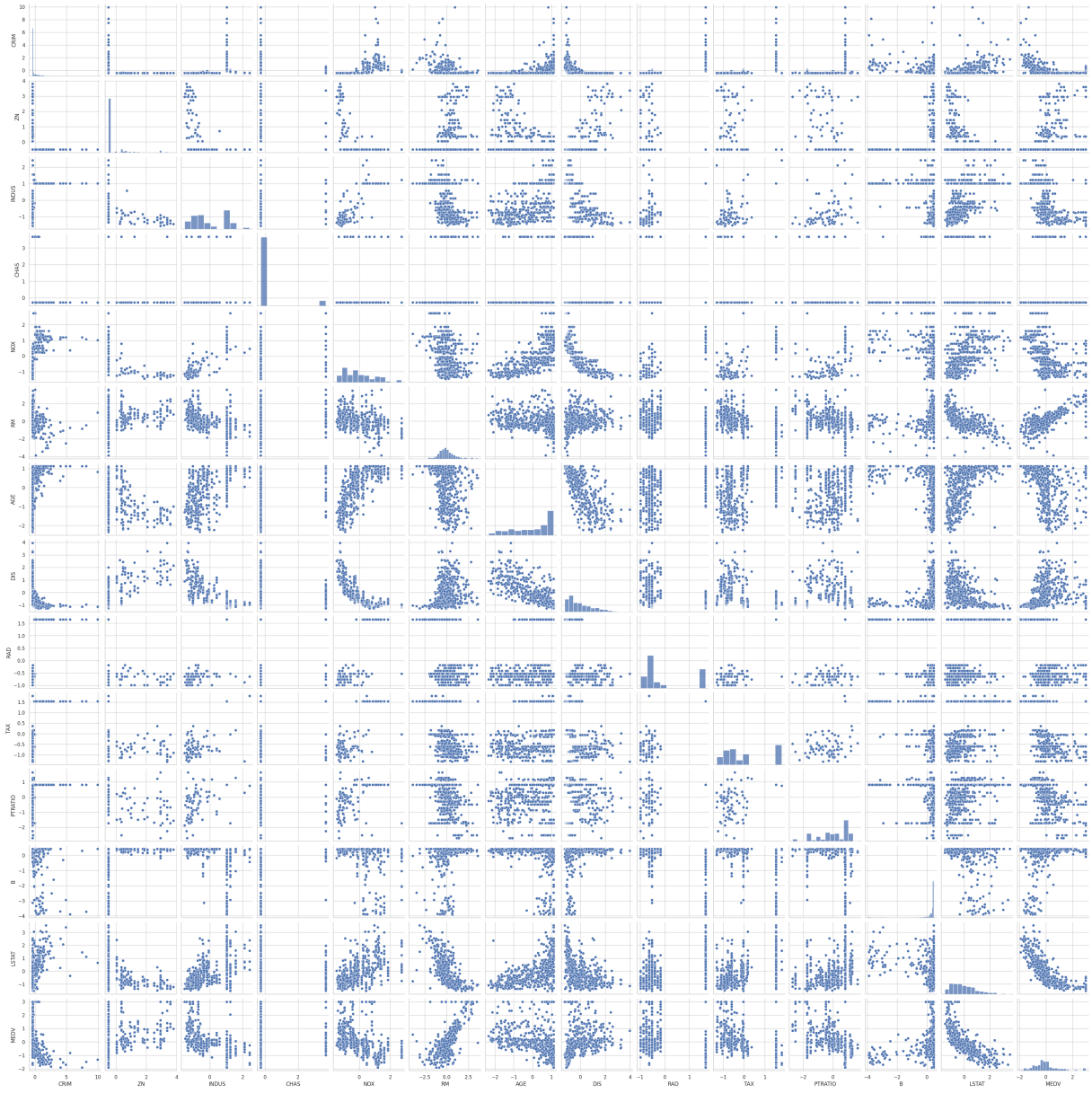
- The data was fetched using the raw source file url to make the by-pass cleaning the data.

▼ Dataset description

The data has the following numerical and categorical features:

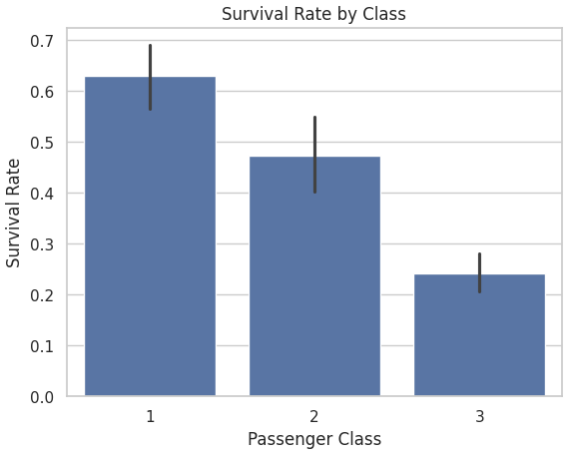
- Numerical Features :
- Categorical Features :
  - Nominal and ordinal data are the types of categorical data. In ordinal data the categories can be ranked or ordered whereas nominal data is just descriptive and doesn’t inherently provide a ranking or an order.
  - Nominal Features : Name, Sex, Embarked, Cabin, Ticket

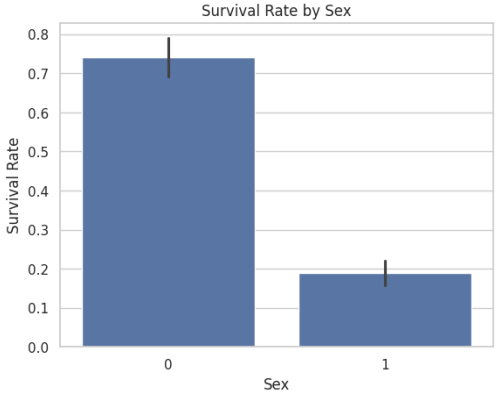
▼ Pairplot

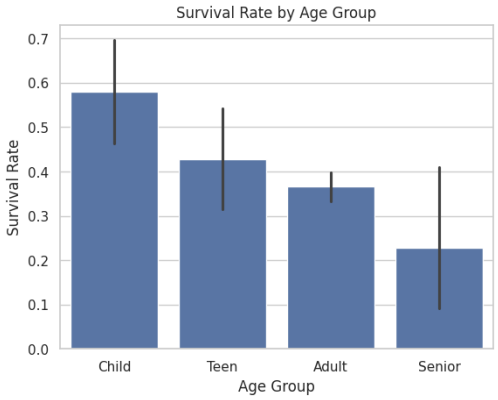


Pairplot showing scatterplot between all feature combinations, while the diagonals show the distribution of a feature.

▼ General Trend of Survival vs Categorical Features:

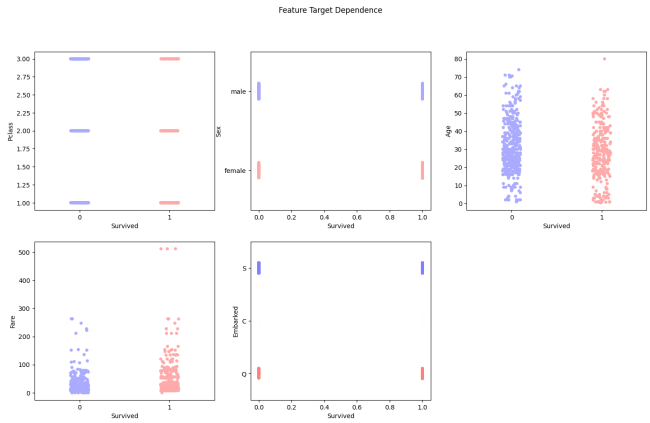






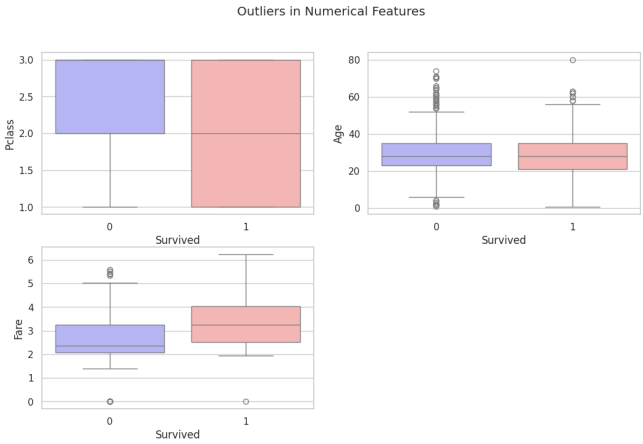
- It can be clearly gleamed that the survival rate is higher for female and passengers of higher class, this is inline with the historical context that women were prioritized while being boarded into lifeboats.
- Also, the survival rate seems to be higher for children since they were prioritized during the rescue too.

▼ Visualizing the feature-target dependence:



- While this doesn't really tell us much about categorical data like `Pclass`, `Sex` and `Embarked`. It reinforces my earlier inferences drawn from the bar plots for the features `Age` and `Fare`.

▼ Checking for outliers in Numerical Features:



- `Pclass`: The attribute has values of 1, 2, and 3, with an absence of outliers.
- `Age`: The plots show some data values above 60. This is indicative of elderly people traveling in the ship rather than outliers. Hence, these data points are considered and not removed.
- `Fare`: Fare values can be large, as shown in the plot, and this is reasonable enough. Hence, this is also considered.

The dataset under review has no outliers.

⇒ Therefore, while there are statistical outliers, there aren't any outliers which are skewing the data in an undesirable direction, all the points considered are relevant to our analysis. Hence, we don't need to discard any of the data points to account for the skewedness brought about by the outliers.

Task 1.1.2 Preprocessing and Feature Engineering

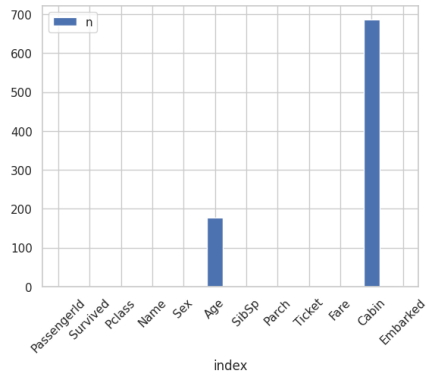
▼ Dropping Columns:

- At the first sight, we can see that the "`PassengerId`" column can be dropped since it just serves the purpose of indexing and is not a useful feature.
- Other features like `Name` and `Ticket` were dropped too.
- There was a useful feature `Title` extracted from the `Name` feature, but `Ticket` didn't seem to be a useful feature from my research.

▼ Data Cleaning:

- Missing values in the data were filled with a statistical measure.
  - For Age: The median was chosen to not let the extremities of old age in the data affect the predictions.

- Embarked: This column represents the destination, and in my opinion the most frequently occurring destination spot is the appropriate way to fill the missing values. Rather, the missing values could also be filled with the correct destinations based on the other.



▼ Train-test-val splitting the dataset:

- I have split the dataset into train, test, and validation splits in the ratio 70:20:10.
- I have used the train set to initially train the decision tree, the validation set to tune the hyperparameters like max_depth of the tree, min_sample_size of the split (samples in a split) and minimum information gain.
- I have also used it to conduct a bunch of experiments, a couple of which I've included in the final code.

▼ Feature Engineering:

- Note that the "Name" column wasn't dropped in the earlier sections. We process the name to extract the title of a person. For example, titles like "Dr.", etc etc have a higher probability of surviving. The titles extracted from each name are then one-hot encoded into an ordered dictionary.
- Created a new feature `'FamilySize'` from `'SibSp'` and `'Parch'` as the summation of the both of them. There is a significant correlation between survival rates and the number of family members onboard. This could be due to the higher ability to be informed of the crisis happening with the help of family members. Large families are grouped together, giving them a better rate of survival.
- `'Fare'` is highly skewed and I created a new feature which is calculated as the log of the original, which removes some of the skewness in the variable.

Task 1.2 Cost Function

▼ Entropy:

- In decision trees, entropy is **a measure of impurity used to evaluate the homogeneity of a dataset/decision tree node**.
- The formula for calculating the entropy of a node containing a subset of the training data, for a general classification problem is:

$$I_H = -\sum_{j=1}^{c} p_j log(p_j)$$

- where p_j is the probability or class percentage of a class $j \in C$, in a node.

▼ Information gain as the cost function:

- At every node of a decision tree, the decision for which feature to split the node on is made by seeing which split results in the **maximum decrease in entropy upon splitting**.
- We calculate the weighted average of the entropy of children nodes (with number of samples as the weights for each split) that result from splitting on a feature, and subtract it from the entropy of the parent node. We choose the feature which has the highest possible resultant value of this calculation. (In other words, we maximize information gain.)
- Here is my implementation of the function to calculate the entropy of a node based on the target label values:

```
def calculate_entropy(y):
    _, counts = np.unique(y, return_counts=True)
    probabilities = counts / counts.sum()
    entropy = sum(probabilities * -np.log2(probabilities))
    return entropy
```

Task 1.3 conversion of variables from categorical to continuous

- My final implementation of the `conTocat()` function involves splitting the dataset into two partitions on the basis of a decision of the form `Feature <= value` or `Feature > value`.
- In my implementation all of the features were made numerical and continuous to some degree.
- I have made the above choice since making decisions of the form : `'Sex = female'` has resulted in a significantly lower performance than one hot encoding the feature Sex and letting the tree chose a decision boundary of the form : `Sex <= 0.5`.
- In my opinion, the above is a result of letting the model choose free decision boundaries which divide the entire feature space into two regions rather than

isolating a particular feature and classifying the rest as otherwise, effectively over-fitting to the current feature space.

Task 1.4 Implementation of the training function

I have implemented all the necessary helper functions for making a decision tree.

▼ a) Get the attribute that leads to the best split

- I have implemented a `identify_splits(X)` function that yields all of the possible unique splits in the data. Then the `best_split(X)` function takes into account all of these possible splits and chooses the split resulting the highest information gain as the best split for a given dataset X. Note that I have already concatenated the Y label or target information into the X variable which is why this function only takes in X as the argument.

```
def best_split(X, y, feature_types):
    possible_splits = identify_splits(X)
    best_gain = -1
    best_split_column = None
    best_split_value = None
    for split_column in possible_splits:
        for split_point in possible_splits[split_column]:
            print(f"Split Column : {split_column}, Split Point : {split_point}")
            gain = cost_function(X, y, split_column, split_point, feature_types)
            if gain > best_gain:
                best_gain = gain
                best_split_column = split_column
                best_split_value = split_point

    return best_gain, best_split_column, best_split_value
```

- This function calls upon a helper function `identify_splits(X)` which returns all the possible splits given a subdataset `X`.
- To get the attribute that leads to the best possible split, we use the `cost_function` method defined earlier here which gives the gain (or, depending on the convention used, decrease) in the entropy value due to a split on a given feature/column.

```
def cost_function(X, y, split_column, split_value, feature_types):
    print(split_column, split_value)
    left_X, right_X, y_left, y_right = conTocat(X, y, split_column, split_value, fe
    if len(y_left) == 0 or len(y_right) == 0:
        return 0  # No gain if split is not possible

    total_entropy = calculate_entropy(y)
    p_left = len(y_left) / len(y)
    p_right = len(y_right) / len(y)
```

```
    left_entropy = calculate_entropy(y_left)
    right_entropy = calculate_entropy(y_right)
    gain = total_entropy - (p_left * left_entropy + p_right * right_entropy)

    return gain
```

b) Split the node on a selected attribute

- Please read below.

c) Repeat these steps on the newly-created split

- I have done both b) and c) together in the function, `build_tree` which takes in a dataset X, y and the feature types for X and constructs a decision tree recursively while keeping track of BOTH : a) maximum depth specified that should not be exceeded and b) splitting should result in at least more than a specified minimal gain value to ensure that the model has not plateaud in its training. This minimal gain value is a tunable parameter.

```
def build_tree(X, y, feature_types, current_depth=0, max_depth=20, min_samples_spli
    if len(y) <= min_samples_split or current_depth >= max_depth:
        return y.mode()[0]

    gain, split_column, split_point = best_split(X, y, feature_types)
    if gain <= min_gain:
        return y.mode()[0]

    left_X, right_X, y_left, y_right = conTocat(X, y, split_column, split_point, fe
    if len(y_left) == 0 or len(y_right) == 0:
        return y.mode()[0]

    question = "{} <= {:.2f}".format(split_column, split_point) if feature_types[sp
    subtree = {question: []}

    yes_answer = build_tree(left_X, y_left, feature_types, current_depth + 1, max_d
    no_answer = build_tree(right_X, y_right, feature_types, current_depth + 1, max_

    if yes_answer == no_answer:
        subtree = yes_answer
    else:
        subtree[question].append(yes_answer)
        subtree[question].append(no_answer)

    return subtree
```

Additional Remarks:

- The wording in Task 4 (b) wasn't very clear, it seems to be implying at pre-pruning (early stopping) so I've chosen to implement that.

- There can be two ways to approach pre-pruning:
  - Minimum Threshold Gain: Instead of building the tree until we hit 0 information gain, another approach could be to set it as a parameter based on the dataset being worked with and tune it as a hyperparameter.
  - Validation Error: Another approach could be to continue building up the tree until we see a spike in the Val error, and stopping there.
- I've clarified with my lab instructor regarding whether I should be implementing pre-pruning and he reckoned it should be fine to let the model train until we see no significant information gain.

Task 1.5 Inference function

- I have written a `Infer` function that computes the result given a decision tree and a sample. Another function, `predict`, takes in multiple samples and computes the results for all of them.

```
def infer(example, tree):
    if not isinstance(tree, dict):
        return tree

    question = list(tree.keys())[0]
    feature_name, comparison, value = question.split(" ", 2)

    if comparison == "<=":
        if example[feature_name] <= float(value):
            answer = tree[question][0]
        else:
            answer = tree[question][1]
    else:
        if str(example[feature_name]) == value:
            answer = tree[question][0]
        else:
            answer = tree[question][1]

    return infer(example, answer)

def predict(X, tree):
    predictions = []
    for index, row in X.iterrows():
        prediction = infer(row, tree)
        predictions.append(prediction)
    return predictions
```

Task 1.6 Results on train and test splits

▼ Validation split (for experiment 2):

- Overall Accuracy: 0.8202247191011236
- Class-wise Accuracy: [0.9009901 0.71428571]
- For survived and not survived respectively.

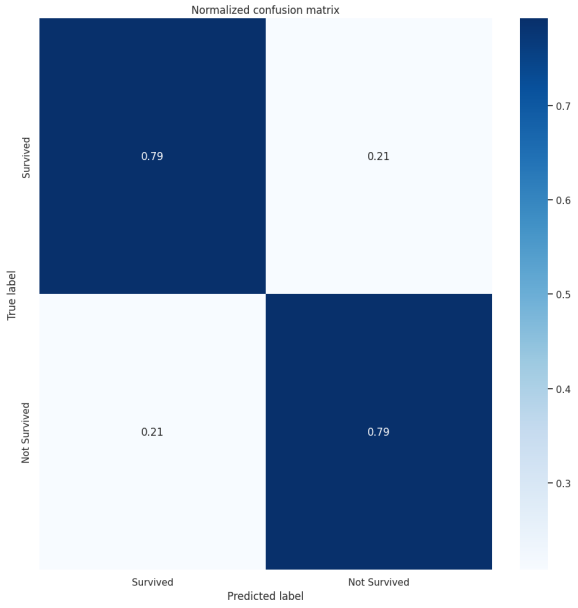
▼ Test split:

Overall Accuracy: 0.8

💡 Hyper-parameter Tuning:

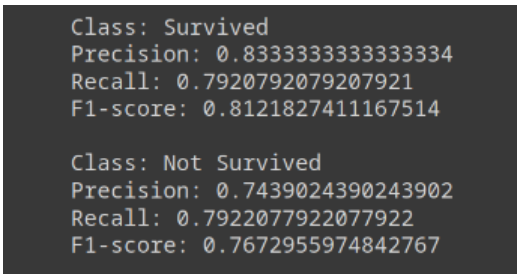
Inside the Experiment sections in my notebook, I have included a couple of experiments on the effect of different kind of tunable hyper-parameters on the decision tree accuracy in order to increase the score. Namely, I have varied the max depth and I found that higher max-depth with a sizeable min-sample-split (around 50) gave me a better accuracy on the validation set.

Task 1.7 Confusion Matrix



- The confusion matrix indicates a balanced classification model with a 79% accuracy rate for both correctly predicting survival and non-survival. However, there is a 21% chance of error in wrongly predicting the survival status, either as false negatives (predicting death when the individual survived) or false positives (predicting survival when the individual did not survive).

Task 1.8 Calculating Other Metrics of performance



- The “Survived” class has high precision, indicating there’s a better chance that the model can correctly classify samples which survived.
  - It has about the same recall for both “Survived” and “Not Survived”, meaning the model generally has a better memory, which is expected for a decision tree.
- The F1-score which balances precision and recall is suggesting a good overall performance for the model which can be improved by a little more tinkering around. (Currently, on it !)

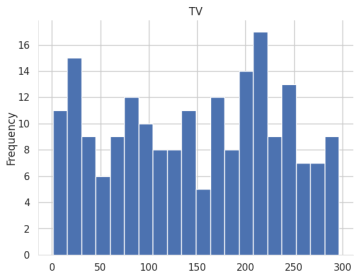
Summary

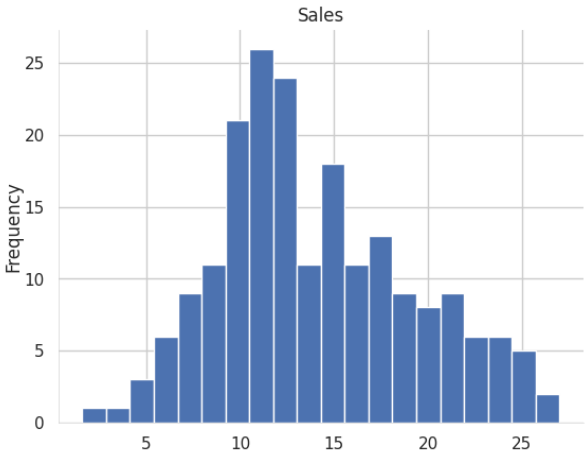
- In summary, the decision tree classifier was successfully trained while completing all the tasks step by step.
- I've also discussed about implementing pre-pruning

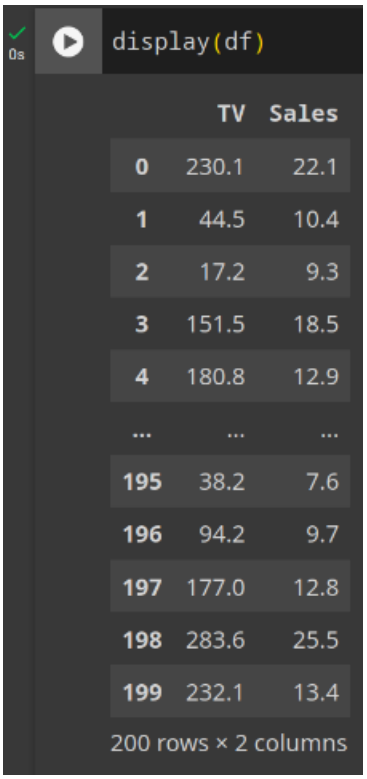
Question 2: (Linear Regression 1)

Task Outline:

- Dataset : TV Marketing Data.
- Overview:





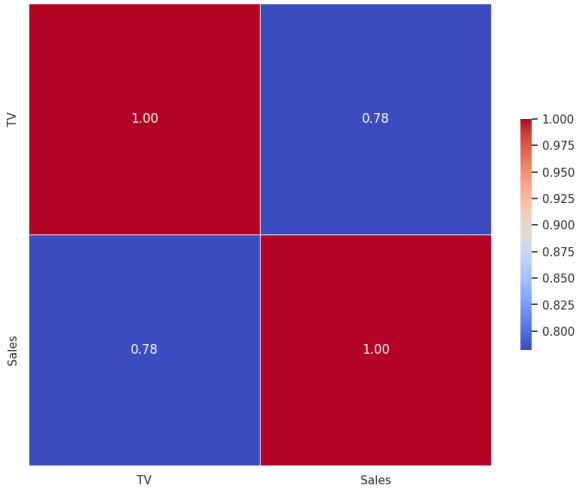


Dataset

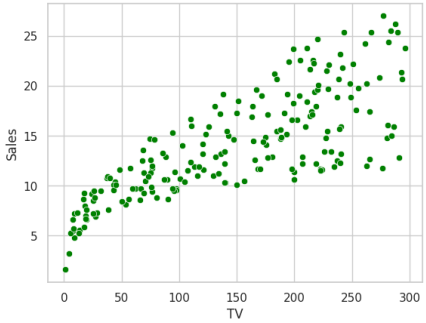
- Feature Description:
  - The only feature is the `TV Marketing Budget`, and the target variable is the `Sales`.
- Dimensions : `200 rows × 2 columns`
- Objective : To implement a single-variate linear regression from scratch on the dataset.

Task 2.1:

▼ Data Exploration and Visualization:



Correlation Matrix



Scatter Plot

- The red cells represent the correlation of a variable with itself and the blue cells represent the correlation of the TV budget with Sales. While the relationship isn't perfectly linear. The measure 0.78 indicates a positive and a fairly strong linear relationship between the target and feature.

▼ Statistical Overview:

- Here, are the statistical measures of the dataset.

	TV	Sales
count	200.000000	200.000000
mean	147.042500	14.022500
std	85.854236	5.217457
min	0.700000	1.600000
25%	74.375000	10.375000
50%	149.750000	12.900000
75%	218.825000	17.400000
max	296.400000	27.000000

Task 2.2:

💡 There were no missing values in the dataset.

▼ Normalization

- I ran the experiment twice with two types of normalization:

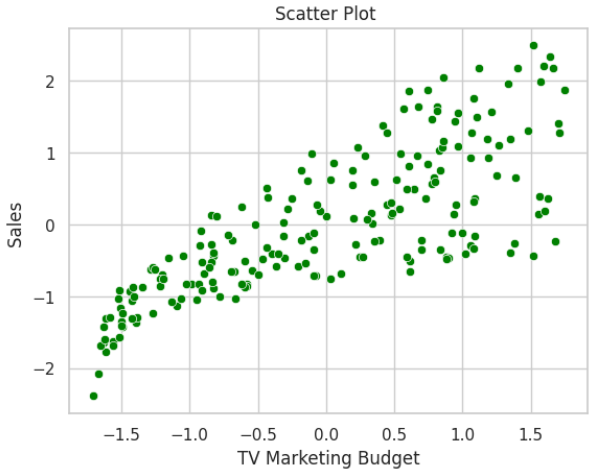
- Standard Normalization:
  - $z = \frac{(x-\mu)}{\sigma}$
- Min-Max Normalization:
  - $x_{\text{norm}} = \frac{(x-x_{\min})}{(x_{\max}-x_{\min})}$

▼ Standard Normalization:

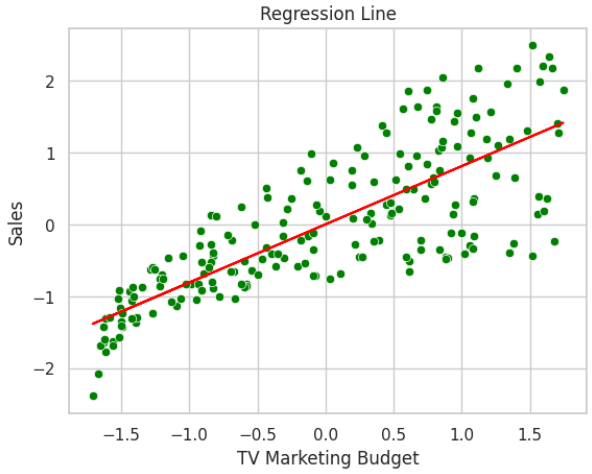
- Below is the scatter plot generated after normalizing the data with standard normalization.
- The data was centered around the mean and normalized using the standard deviation.
- A snippet of the training loop is shown below.

```
Iteration : 984, dw : -3.0580923460199473e-09, db : -3.318129684903859e-09
New Weights : w 0.8108310335874189 and b: 0.004382197851685596
Iteration : 985, Computed Loss : 0.38597192359436033
Iteration : 985, dw : -2.99792034530455e-09, db : -3.2551595460783566e-09
New Weights : w 0.8108310336173982 and b: 0.004382197884237191
Iteration : 986, Computed Loss : 0.38597192359436033
Iteration : 986, dw : -2.938934717811037e-09, db : -3.193381949151419e-09
New Weights : w 0.8108310336467875 and b: 0.004382197916171011
Iteration : 987, Computed Loss : 0.3859719235943604
Iteration : 987, dw : -2.88111225987819 34e-09, db : -3.132774399616789e-09
New Weights : w 0.8108310336755987 and b: 0.004382197947498755
Iteration : 988. Computed Loss : 0.38597192359436033

> Loss converges around 0.38.
```

Gradient Descent Logs



Scatter Plot for Standardized Data.



Regression Line for Standardized Data.

▼ Min-Max Normalization:

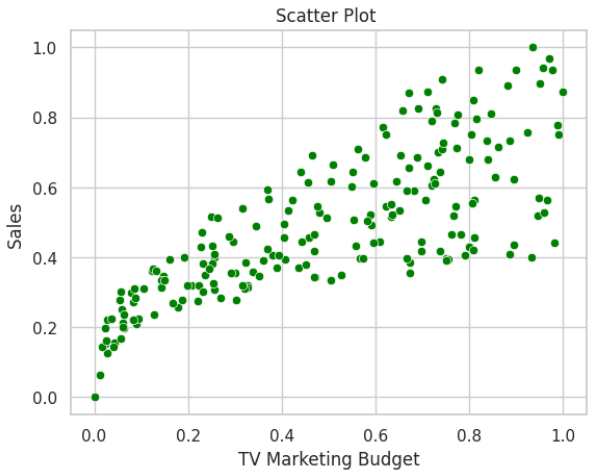
- Below is the scatter plot generated after normalizing the data with min-max normalization.
- The data was centered around the minimum and normalized using the difference between the maximum sample and the minimum sample.

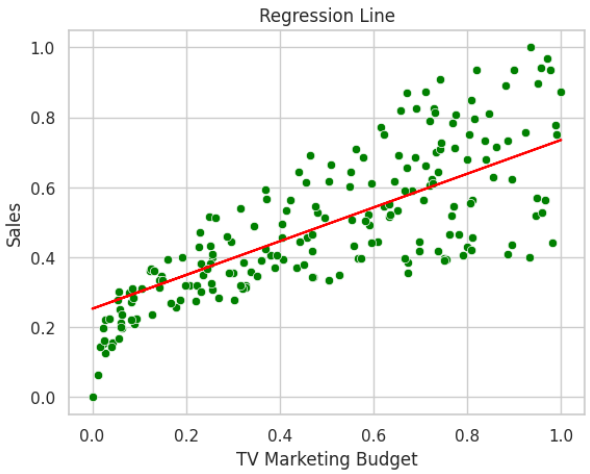
- A snippet of the training loop is shown below.

```
New Weights : w 0.4820388136116926 and b: 0.2532429927649858
Iteration : 996, Computed Loss : 0.016947884479212017
Iteration : 996, dw : -0.012832676964914047, db : 0.006607577480305646
New Weights : w 0.48216714038134173 and b: 0.25317691699018274
Iteration : 997, Computed Loss : 0.016945802561642145
Iteration : 997, dw : -0.012814700760234183, db : 0.006598321487896334
New Weights : w 0.48229528738894406 and b: 0.2531109337753038
Iteration : 998, Computed Loss : 0.01694372647274909
Iteration : 998, dw : -0.012796749736892543, db : 0.006589078461411588
New Weights : w 0.482423254886313 and b: 0.2530450429906897
Iteration : 999, Computed Loss : 0.01694165619621449
Iteration : 999, dw : -0.012778823859614657, db : 0.006579848382688694
New Weights : w 0.48255104312490915 and b: 0.2529792445068628
Iteration : 1000, Computed Loss : 0.01693959171576567
Iteration : 1000, dw : -0.01276092309317547, db : 0.006570631233590416
New Weights : w 0.4826786523558409 and b: 0.25291353819452694
```

Loss converges around 0.38 for standardized data and converges around 0.016 for min-max normalized data.

Gradient Descent Logs





Scatter Plot for Normalized Data

Regression Line for Normalized Data

▼ Train-Test Split:

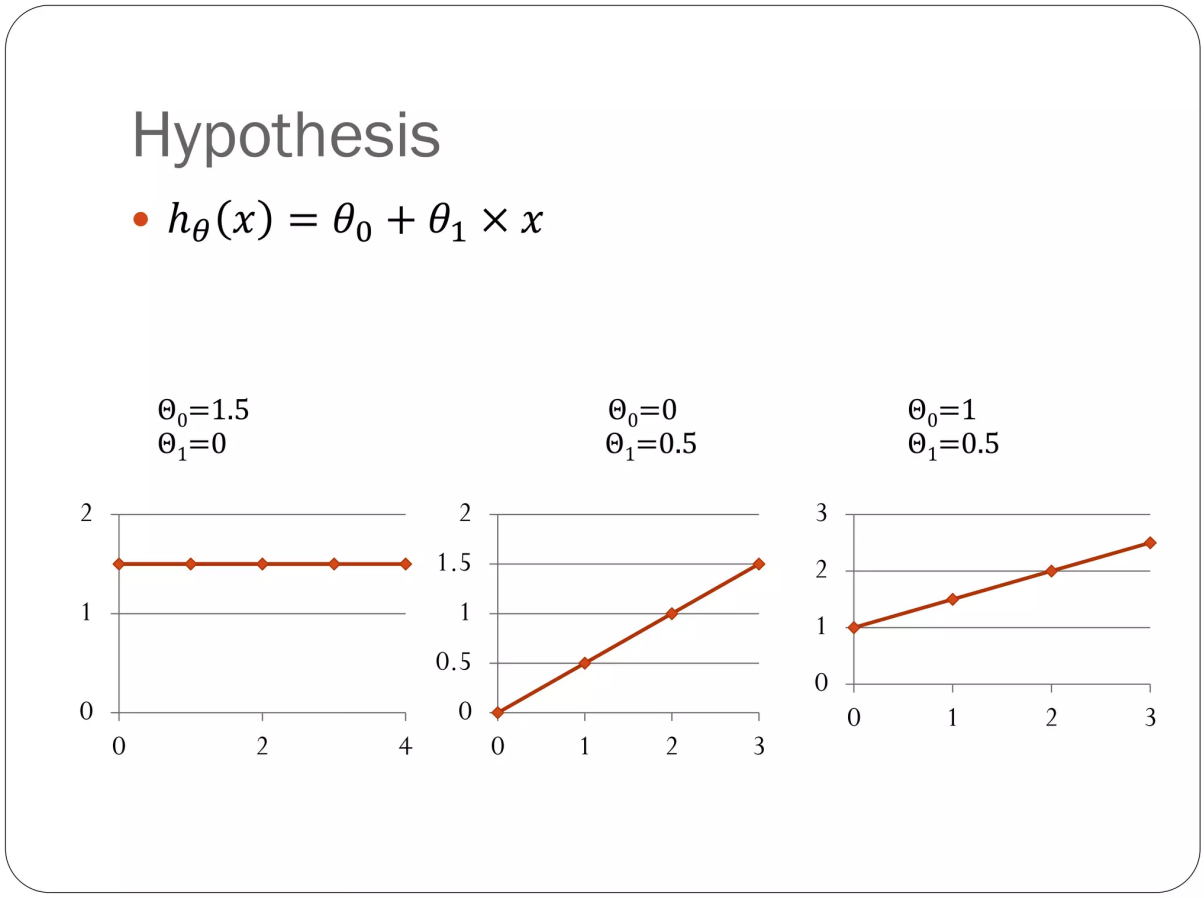
- The split was performed as instrucuted.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Task 2.3

- All of the following functions were implemented from scratch. Sources referred to are linked at the end of the report.

▼ Model Hypothesis:



- Initially a model of the form $y = w \cdot x + b$ was hypothesized. w and b were intialized as 0 both. The initialization could have been done randomly too.

▼ Cost Function:

- The cost function used was the *mean square entropy*. This cost function is generally preferred over other ones like mean absolute error since it's differentiable and more sensitive to outliers.
- The cost function is given by: $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

```
def mean_squared_error(y_hat, y_predicted):
    mse = np.mean((y_predicted-y_hat)**2)
    return mse
```

▼ Gradient Descent:

- The training loop or performing gradient descent involves multiple steps. The helper functions I broke it down to were:
  - compute_gradients : Takes in the X, y_predicted and y_original and returns the gradients required to update the parameters w and b.

```
def compute_gradients(x, y_hat, y_pred):
    db = np.mean(-2*(y_hat - y_pred))          # derivative of loss function
    dw = np.mean(-2*(x*(y_hat - y_pred)))
    return dw, db
```

- weight_update : The gradients from the previous step are used to now update the parameters by performing the learning step/parameter update step, given by:
  - $\theta_{\text{new}} = \theta_{\text{old}} - \alpha \cdot \nabla_\theta J(\theta)$

```
def weight_update(w, b, db, dw, learning_rate):
    w_new = w - learning_rate*dw
    b_new = b - learning_rate*db
    return w_new, b_new
```

- The above equation holds true for multivariate linear regression and in this case it simplifies to a simple $learningrate * gradient$ since there is no vectorization involved.

- The final gradient descent function uses all of these helper functions to calculate the gradients of w and b (gradients calculated here are to minimize the mean_absolute_error), and keeps updating for a given number of epochs.
- The tunable parameters here are the learning rate and the number of epochs, though the model is almost guaranteed to converge after a few minimum number of epochs unless the learning rate is too high in which case it keeps oscillating.

```
def gradient_descent(w, b, X_train, y_train, epochs, learning_rate):
    for i in range(epochs):
      # for x, y in enumerate(zip(X_train, y_train)): #will add batch training if t
        y_pred = w*X_train + b
        mse = mean_squared_error(y_train, y_pred)
        print(f"Iteration : {i+1}, Computed Loss : {mse}")
        dw, db = compute_gradients(X_train, y_train,y_pred)
        print(f"Iteration : {i+1}, dw : {dw}, db : {db}")
        w, b = weight_update(w, b, db, dw, learning_rate)
        print(f"New Weights : w {w} and b: {b}")
    return w, b
```

- Plotted Regression lines were shown above.

Task 2.4

▼ Metrics

- Here are the computed mean square and mean absolute errors on test set.

```
[97] mse = mean_squared_error(y_test, y_test_pred)
     mae = mean_absolute_error(y_test, y_test_pred)
     print("MSE:", mse)
     print("MAE:", mae)

     MSE: 0.01585809761071221
     MAE: 0.09514397976339875
```

Question 3: (Linear Regression 2)

Task Outline:

- **Dataset :** Boston House Prices Dataset.
- **Overview:** The given dataset is a very well known dataset in machine learning and one of the oldest datasets. It contains the median housing price data and various factors affecting it in the Boston suburbs.
- **Feature Description:**

7.2.1. Boston house prices dataset

Data Set Characteristics:

Number of Instances:	506
Number of Attributes:	13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.
Attribute Information (in order):	• CRIM per capita crime rate by town • ZN proportion of residential land zoned for lots over 25,000 sq.ft. • INDUS proportion of non-retail business acres per town • CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise) • NOX nitric oxides concentration (parts per 10 million) • RM average number of rooms per dwelling • AGE proportion of owner-occupied units built prior to 1940 • DIS weighted distances to five Boston employment centres • RAD index of accessibility to radial highways • TAX full-value property-tax rate per $10,000 • PTRATIO pupil-teacher ratio by town • B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town • LSTAT % lower status of the population • MEDV Median value of owner-occupied homes in $1000's
Missing Attribute Values:	None
Creator:	Harrison, D. and Rubinfeld, D.L.

Source : https://towardsdatascience.com/things-you-didnt-know-about-the-boston-housing-dataset-2e87a6f960e8

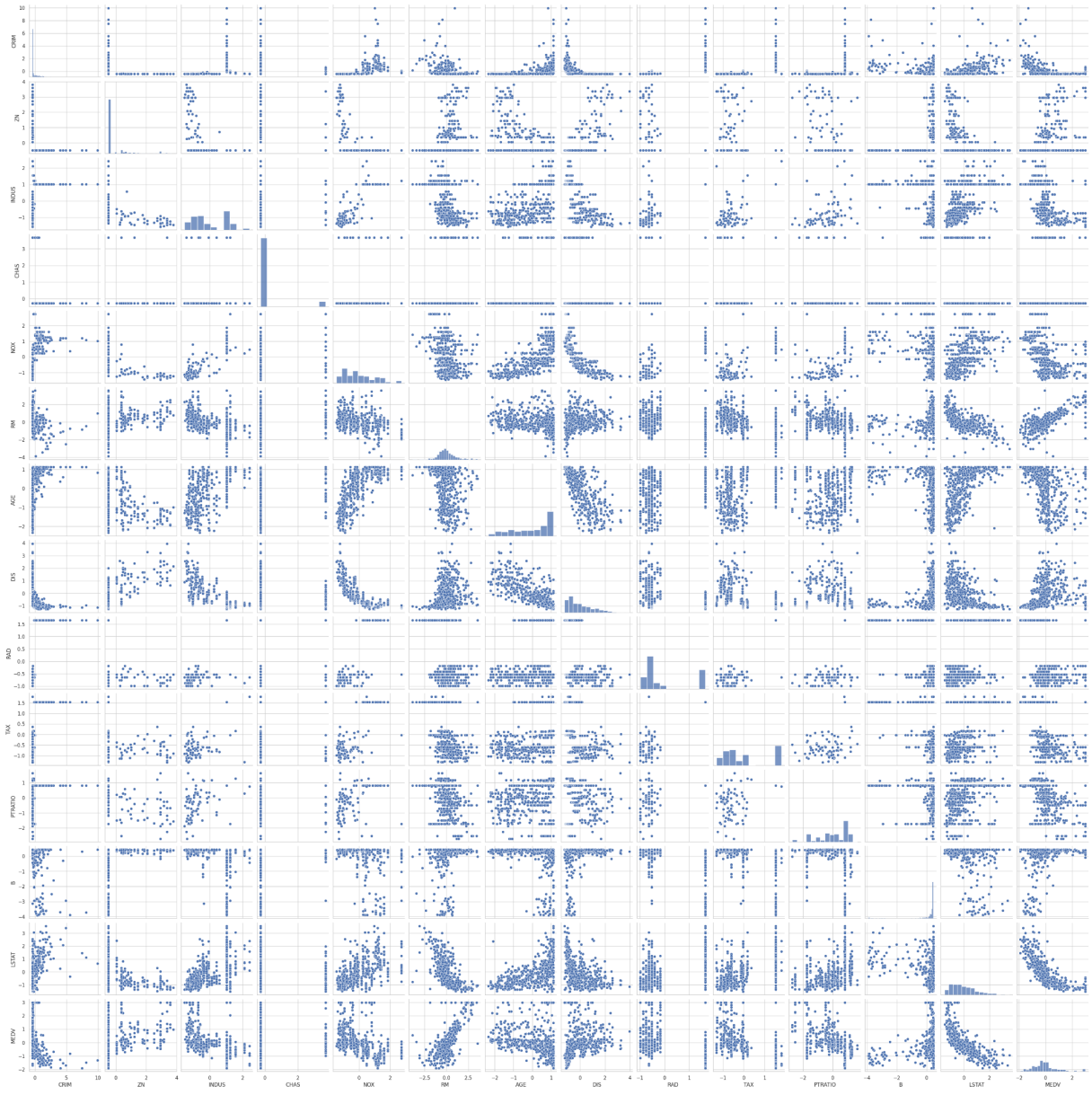
- **Dimensions :** `506 rows × 13 columns`

- Objective : To implement a multi-variate linear regression from scratch on the given dataset.
- MEDV refers to the median value of owner occupied houses in the Boston Suburbs. It is the target variable for this task. It’ll be referred to by MEDV for the rest of the report.

Task 3.1:

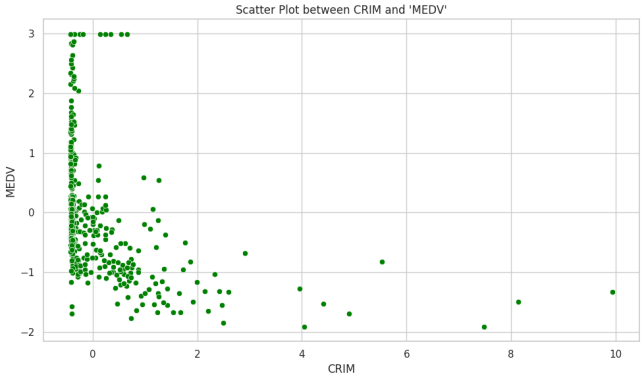
▼ Data Visualization and Exploration:

- A pairplot was plotted to visualize relationships between all the variables with each other. Histograms and density plots are along the diagonal, they show the distribution of each variable.



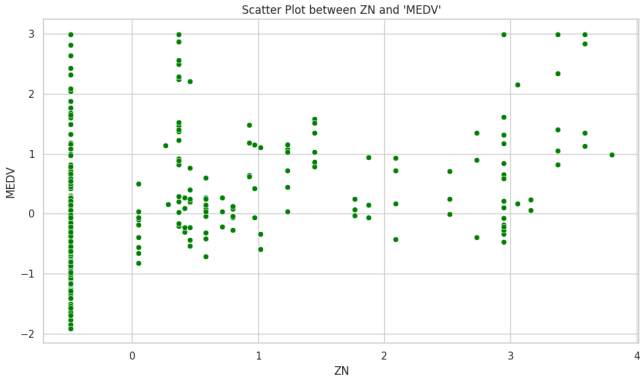
▼ Scatter Plots for each feature against the target variable.

- The scatter plots have all been standardized but the inferences have been drawn while keeping the original data in mind too.



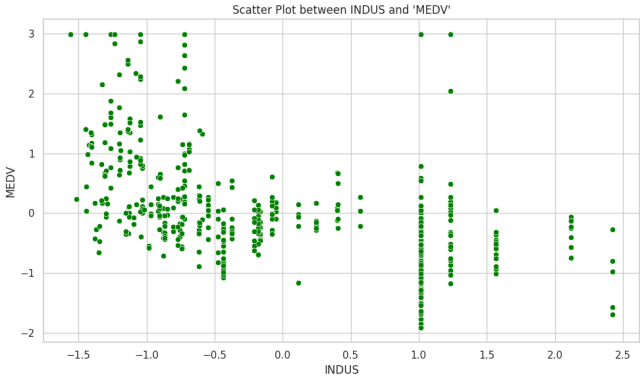
Crime Rate vs Median Value

- The data points are densely packed at lower crime rates, indicating that most of the areas have a low crime rate. MEDV is inversely related to crime rate.



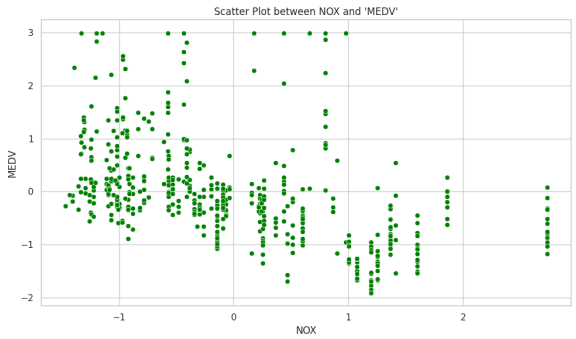
Residential land zoned for lots over 25,000 sq.ft.

- Most of the points are isolated at 0 implying there were many areas with no or minimal zoning for residential lots. There is no clear trend as such.



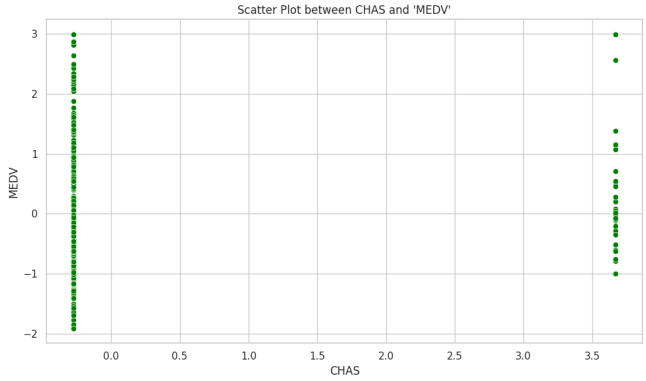
Proportion of non-retail business acres per town.

- Dispersed data, no clear trend.



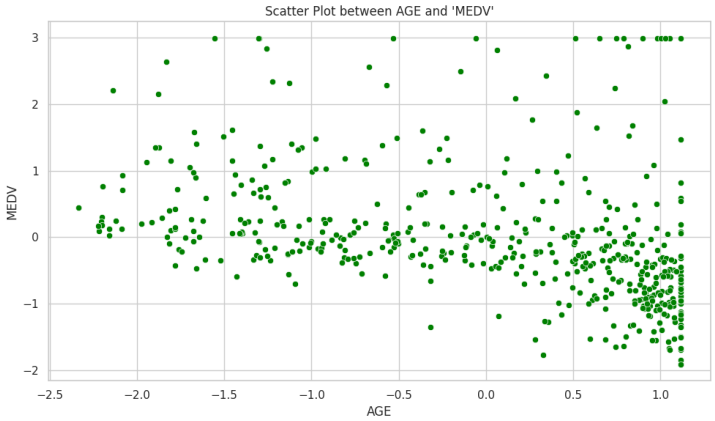
Nitric Oxide Concentration.

- Homes with higher polution have a lower MEDV. Possible negative non linear trend.



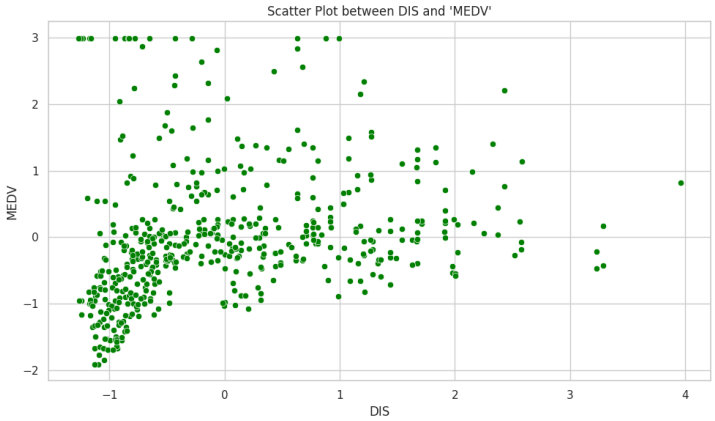
Charles River Dummy Variable

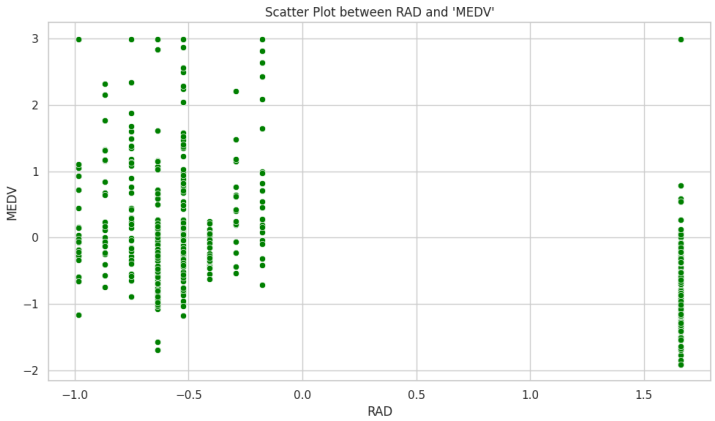
- Being by the riverside seems to have a slightly higher chance of MEDV, but the data is too symmetric for that to have any real significance.

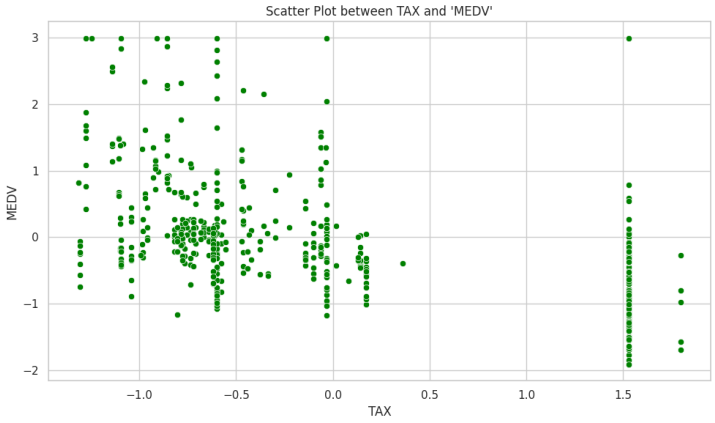


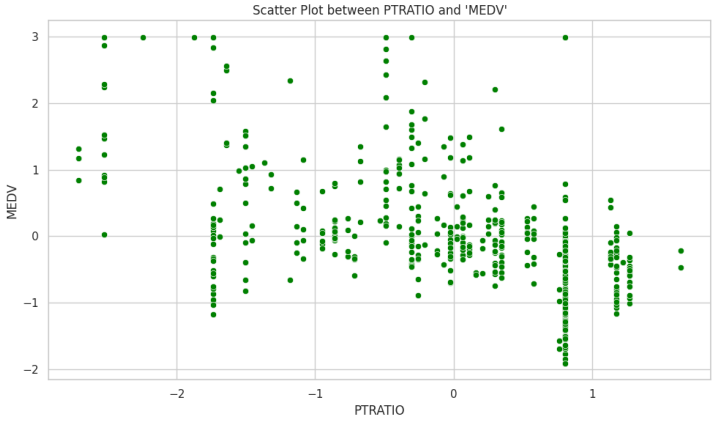
Age of Houses

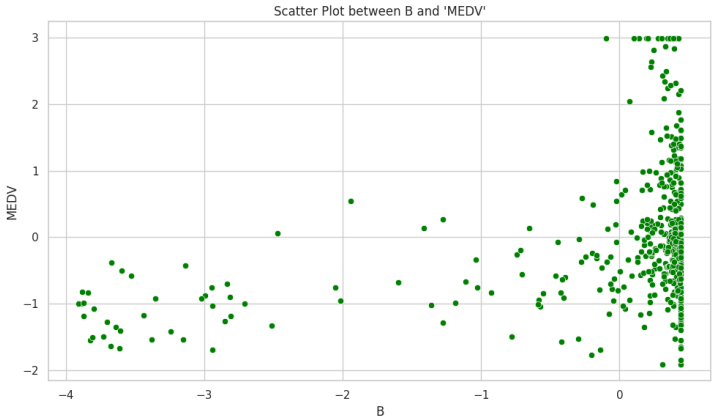
- Possible Inverse Relation, there seems to be a negative linear relation.

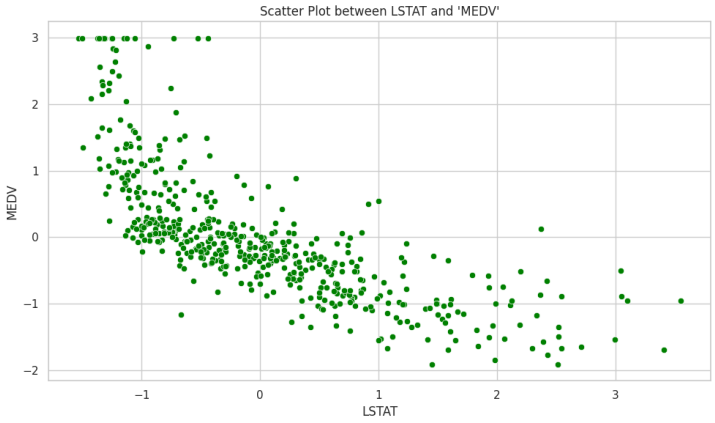





Scatter Plot between TAX and 'MEDV'
MEDV
TAX
3
2
1
0
−1
−2
−1.0
−0.5
0.0
0.5
1.0
1.5


Scatter Plot between PTRATIO and 'MEDV'
MEDV
PTRATIO
3
2
1
0
−1
−2
−2
−1
0
1


Scatter Plot between B and 'MEDV'
MEDV
B
3
2
1
0
−1
−2
−4
−3
−2
−1
0



▼ Statistical Overview:

- Here is the statistical overview of the dataset.

```
[106] data.describe()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
count	506	506	506	506	506	506	506	506	506	506	506	506	506	506
unique	504	26	76	2	81	446	356	412	9	66	46	357	455	229
top	0.01501	0.00	18.100	0	0.5380	5.7130	100.00	3.4952	24	666.0	20.20	396.90	7.79	50.00
freq	2	372	132	471	23	3	43	5	132	132	140	121	3	16

Task 3.2:

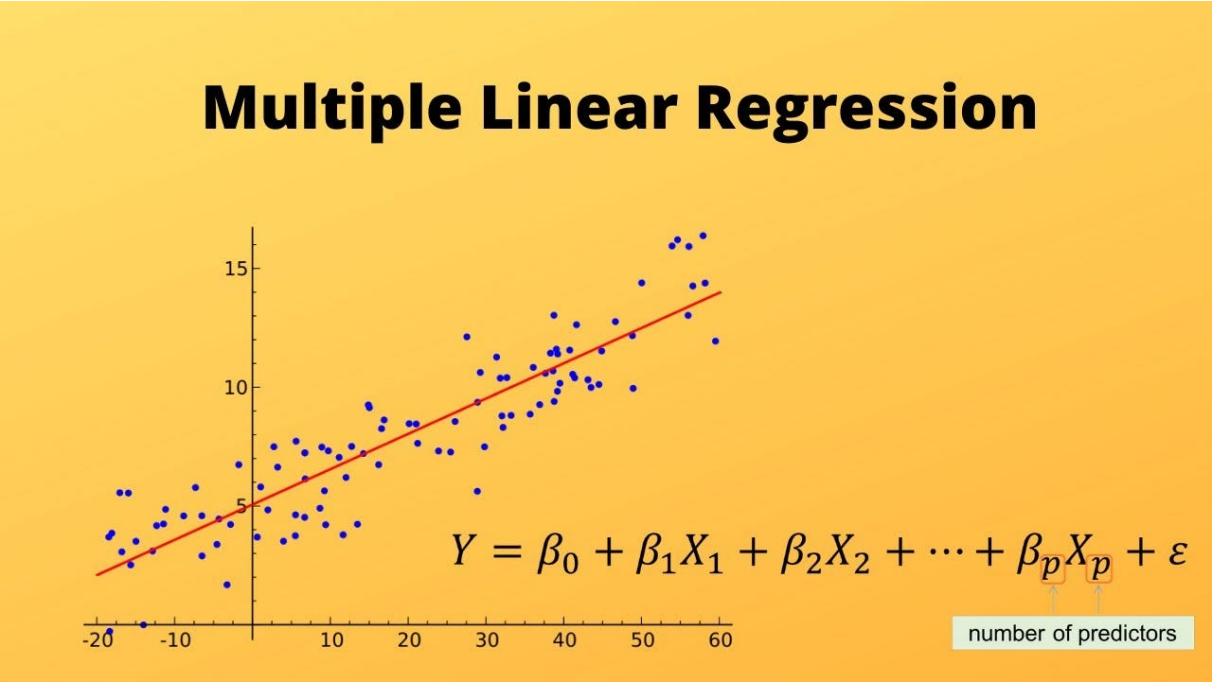
> 💡 There were no missing values in the dataset.

- This section is largely the same as above.
- The data has been normalized using min-max normalization.
- The train-test split was performed as instructed.

Task 3.3:

▼ Line Fitting

- In the case of multi-variable regression the hypothesis function is modelled as follows:



- The implementation details largely same as the above description provided. The equations only slightly differs with the introduction of vectorization of all the variables involved.

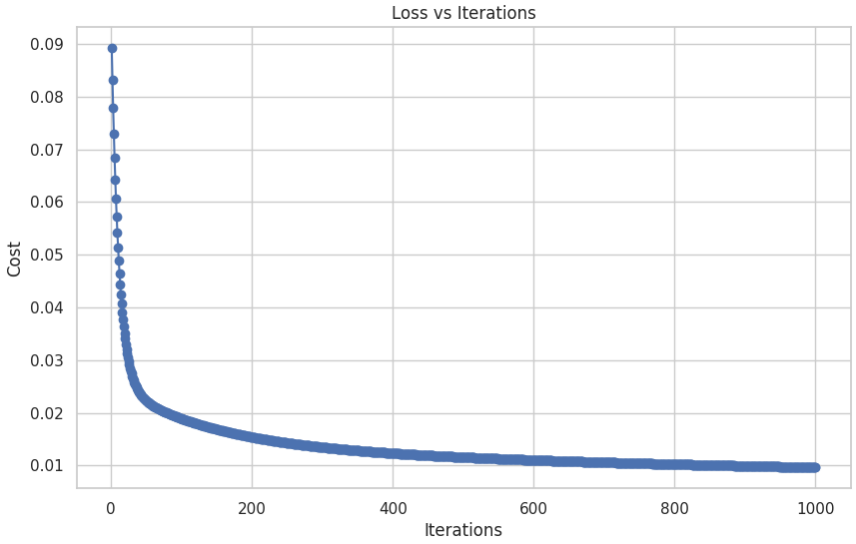
- The equation `y_pred = w*X_train + b` in the case of single variate regression now goes to `y_pred = np.dot(X, w) + b`.

- The training logs and the cost involved are attached here.

```
[121] X_train_with_intercept = np.c_[np.ones((X_train.shape[0], 1)), X_train]

    # Initialize parameters
    n_features = X_train_with_intercept.shape[1]
    w, b = initialize_parameters(n_features)

    # Gradient Descent
    epochs = 1000
    learning_rate = 0.01
    w, b, cost_history = gradient_descent(X_train_with_intercept, y_train.values.reshape(-1,1), w, b, learning_rate, epochs)

    Iteration 0: Cost 0.08919140109323503
    Iteration 100: Cost 0.018865831592240506
    Iteration 200: Cost 0.015372800180982893
    Iteration 300: Cost 0.013461301557410807
    Iteration 400: Cost 0.012324491822615472
    Iteration 500: Cost 0.01157705268239519
    Iteration 600: Cost 0.011033122988917732
    Iteration 700: Cost 0.010602218251830116
    Iteration 800: Cost 0.010239557801959007
    Iteration 900: Cost 0.009922346665701116
```

Training Logs

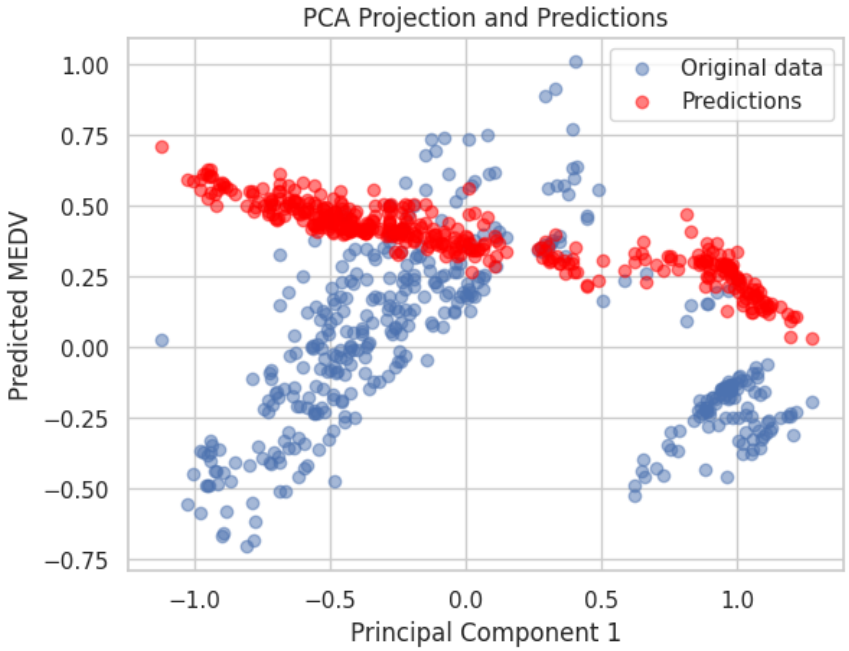
- The value of the cost function is plotted with the iterations, we can see that gradient descent minimizes it fairly quickly.



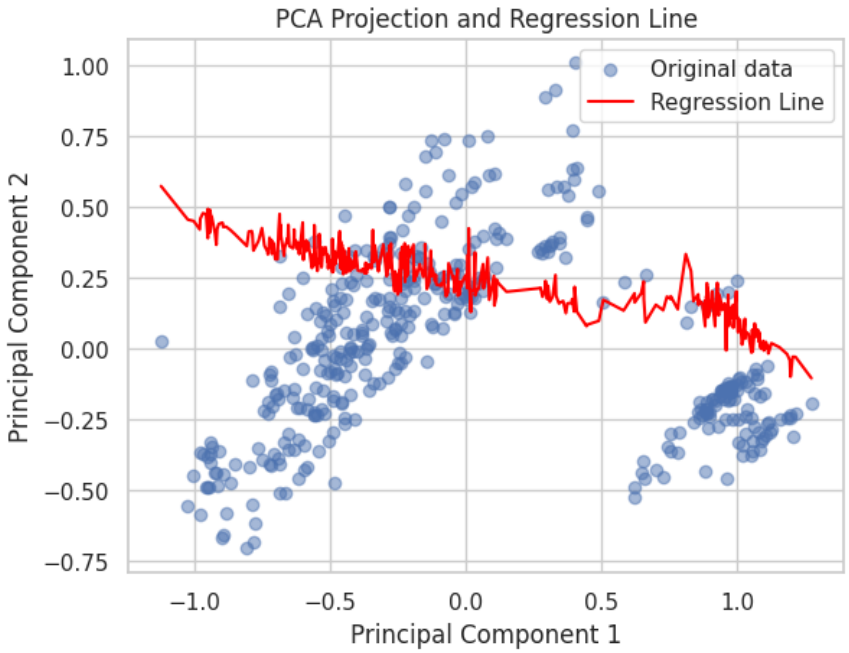
Loss vs Iterations

▼ Visualization the 13 Dimensional Regression line:

- Attempting to take the projections of the line and the feature space onto 2d space using sk-learn’s PCA gave the following plots:

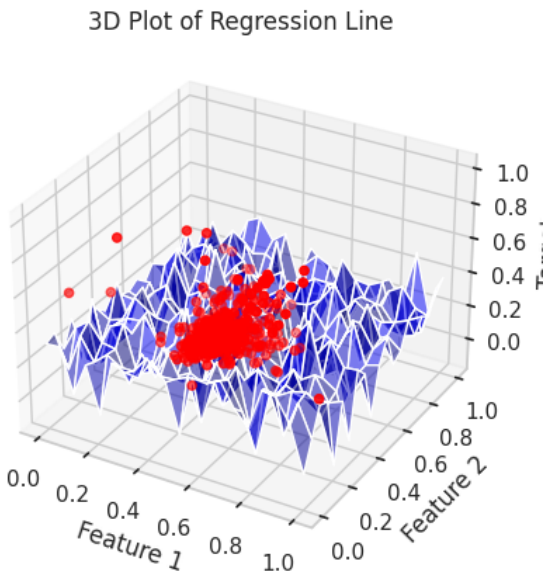


Attempt at plotting the points.



- For this to be the correct projection, the line should have captured the underlying features of the feature space, which it sort of does in this plot. This could be chalked up to an error while making the plot, I would like to give it another shot after studying PCA.

- Another approach is to pick the two most correlated features with the target variable and attempting to plot them along with the regression surface, this gives the following result.



Task 3.4

▼ Metrics

- Here are the computed mean square and mean absolute errors on test set after using the trained model to make predictions.

```
mse = mean_squared_error(y_test, preds)
mae = mean_absolute_error(y_test, preds)
print("MSE:", mse)
print("MAE:", mae)
```

```
MSE: 0.024680126927729813
MAE: 0.11055379736691438
```

References:

- https://www.ahmedbesbes.com/blog/kaggle-titanic-competition
- https://medium.com/@praoiticica/titanic-data-cleaning-and-feature-engineering-9f122752097f
- https://medium.com/analytics-vidhya/data-visualization-titanic-data-set-91531c3ab5a6
- I started making the report after finishing most of my work and hence cannot recall a lot of sources that I've used for learning, I will make sure to list them all henceforth.

THE END

K. K. N. SHYAM SATHVIK.