Housing Prices

Rohan Pradhan

Submitted for the Degree of Master of Science in

Artificial Intelligence

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

December 12, 2022

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 11022

Student Name: Rohan Pradhan

Date of Submission: 12 December 2022

Signature:

Pradhan

Acknowledgements

I would like to thank my supervisor Dr Argyrios Deligkas for his expert guidance. He has been of immense help to me and played a major role in shaping this project.

I would also like to thank my family and friends for all the love and support that they have always given me.

Abstract

The objective of this project is to apply multiple machine learning models to different housing datasets in order to predict house prices and compare their performance.

We will implement the following regression models from scratch using `Python`: **K-Nearest Neighbors**, **Random Forest**, **Ridge**, and **Kernelized Ridge**. Thereafter, we will apply each of these models to three housing datasets, namely, the Boston Housing Dataset, the California Housing Dataset, and the Ames Housing Dataset. Their performance is then visualized and compared.

Finally, we implement **Linear Regression** in the **Online Mode** where the datapoints are available to us one at a time. We make a prediction for every datapoint that we observe and then see the actual label for that datapoint.

This report describes the background knowledge needed to implement the above models, the steps taken to preprocess the three datasets and the analysis of the results of their application.

Contents

1 Introduction

1.1 Predictive Modelling

The problem of predicting house prices is a type of predictive modelling problem. Predictive modelling is a statistical technique using machine learning and data mining to predict and forecast likely future outcomes with the aid of historical and existing data. It works by analyzing current and historical data and projecting what it learns on a model generated to forecast likely outcomes. [2]

Predicting the price of a house using its features has been studied greatly over the years. This project applies multiple regression models on three housing datasets and evaluates their results.

1.2 Supervised Learning

When a machine learning algorithm is given pairs of inputs and desired outputs, it learns how to produce the desired output given an input. Based on what it has learned, the algorithm is now able to provide an output for an input it has not seen before. This process is known as **supervised learning.**

An example of supervised learning would be spam classification. The algorithm is provided with a bunch of emails (the input) along with the information about whether they're spam or not (the output). Then, given a new email, the algorithm will try to predict whether it is spam or not.

As described in [16], the term *supervised* is used because a "teacher" provides supervision to the algorithms in the form of the desired outputs for each example that they learn from.

1.3 Regression Problem

A regression problem or regression predictive modelling is the task of approximating a mapping function $\mathbf{f}$ from input variables $\mathbf{X}$ to a continuous output variable $\mathbf{y}$.[4] In simple terms, it is a problem where we predict a real number value for a datapoint rather than categorizing it.

$$f : X \rightarrow y, \quad y \in \mathbb{R}$$

1.4 Training and Test Sets

Before we can apply our algorithm/model to new and unseen data, we need a way to measure its accuracy. This measure will let us know how much we can trust the predictions of our model.

The data that we used to build our model cannot be used to evaluate it as the model would remember the data very well and always predict the correct output/label for every data point. This fails to indicate how good our model is at generalizing its predictions i.e. how well it can perform on unseen data.

In order to achieve that, we assess our model by applying it to new unseen data for which we already know the true labels. We do this by splitting the labeled data that we have acquired into two parts. One part of it is used to build our model. This part is called the *training set*. The other part is used to evaluate how well our model can perform. This part is called the *test set*.

In Python's `scikit-learn` library, the `train_test_split` function helps us to shuffle the dataset and split it into training and test sets. By default, it extracts 75% of the data into the training set and 25% into the test set. We can modify this split proportion by changing the parameters of the function but in general, a test set containing 25% of the data is considered a good rule of thumb. [16]

2 Background Research

2.1 K-Nearest Neighbors Regression

2.1.1 The Basic Concept

The K-Nearest Neighbors Algorithm is one of the simplest machine learning algorithms. It does not have any training phase i.e. predictions for a new data point can be made straight away. In order to build the model we just need to store the training set. [16]

To make a prediction for a new data point, the k-nearest neighbor algorithm finds the 'k' closest data points to it in the training set. Hence the name, "nearest neighbors". [16]

2.1.2 The Distance Function

Euclidean Distance

The most common way to measure the distance between two data points is to find the Euclidean Distance between them.
In a two-dimensional Euclidean space, let x be a point with coordinates (x_1, x_2) and y be another point with coordinates (y_1, y_2). Then the Euclidean distance between the two points x and y is given by: [7]

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

Therefore for two points x and y in n-dimensional space, the Euclidean distance will be: [7]

$$d(x, y) = \sum_{i=1}^{n} \sqrt{(x_i - y_i)^2}$$

Manhattan Distance

The Manhattan Distance is another common distance measure. For two points x and y in n-dimensional space, the Manhattan distance is calculated as [3]:

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Minkowski Distance

The Minkowski Distance is a generalization of the Euclidean Distance and the Manhattan Distance. [3] For two points x and y in n-dimensional space, the Minkowski distance is calculated as:

$$d(x, y) = \sum_{i=1}^{n} (|x_i - y_i|^p)^{\frac{1}{p}}$$

where p is the order parameter.

- When $p = 1$, we get the Manhattan Distance.
- When $p = 2$, we get the Euclidean Distance.

2.1.3 Prediction

To predict the label for a test data point (x_0, y_0) in a regression problem, we take the following steps:

- Compute N_0, the set of K nearest neighbors of x_0 in the training set.
- Predict the label $\hat{y}$ as:

  $$\hat{y} = \frac{1}{K} \sum_{x_i \in N_0} y_i$$

  i.e. the predicted label $\hat{y}$ is the mean of its K nearest neighbors. [8]

2.1.4 Weighted Prediction

In weighted prediction, we multiply the nearest labels with certain weights instead of taking their mean. The weights are based on how close they are to the test data point.

For example, in a 3-Nearest Neighbor Setting, the weight vector is calculated as follows:
We take an array of positive integers in descending order from the number of neighbors to 1:

$$w = [3, 2, 1]$$

Then we divide this vector by the sum of its elements to get our weights:

$$w = \left[\frac{3}{6}, \frac{2}{6}, \frac{1}{6}\right]$$

Note: The sum of elements of the weight vector w must be equal to 1.

As we can see above in w, the labels closer to the test data point are assigned larger weights while the ones farther away are assigned smaller weights.

For Weighted K-Nearest Neighbors Regression, we take the following steps to predict the label for a test data point (x_0, y_0):

- Compute v, an ordered vector of the K-Nearest Neighbor labels of x_0 in the training set.
- Predict the label $\hat{y}$ as:

$$\hat{y} = w \cdot v$$

2.2 Random Forest Regression

2.2.1 Regression Trees

A regression tree is a machine-learning approach that involves dividing the predictor space into a number of simple regions. A prediction for a given observation is made by taking the mean of the training observations in the region to which it belongs.

Let $\mathcal{D} = \{(x_n, y_n) \in \mathcal{X} \times \mathcal{Y}\}_{n=1}^{N}$ be a labelled dataset.

A regression tree divides the predictor space $\mathcal{X}$ into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$.[12]

For a new test object x_0 and a trained regression tree, the prediction rule is as follows [8]:

$$\hat{y} = \frac{1}{|R_{J_0}|} \sum_{n:x_n \in R_{J_0}} y_n \qquad |R_{J_0}| = \sum_{n=1}^{N} 1[x_n \in R_{J_0}]$$

Loss Function

The loss function of the regression tree is its Mean Square Error (MSE) given by [8]:

$$MSE(R_j) = \frac{1}{|R_j|} \sum_{n:x_n \in R_j} (y_n - \hat{y}_{R_j})^2$$

We build the regions $R = \{R_j\}_{j=1}^{J}$ that minimize the loss function above.

Building the Tree

1. **Top Down Approach:** We start at the top of the tree with all the data in one region. [14]
2. For each leaf node (region):

```
For each feature $x_j$ and split point $s$:
        Calculate the reduction in loss
        (information gain) if we split there
```

3. **Greedy Approach:** Then we choose the best (j, s) combination to split and create new child nodes/regions.[14]
4. **Recursive Step:** We repeat from Step 2 until a certain stopping criterion is met. [14]

Finding the Best Split

Suppose we need to split a region R_j further into R_{j_1} and R_{j_2} such that $R_{j_1} \cup R_{j_2} = R_j$.

The best split is the one that gives us the maximum information gain, defined as: [8]

$$\Delta = |R_j| MSE(R_j) - (|R_{j_1}| MSE(R_{j_1}) + |R_{j_2}| MSE(R_{j_2}))$$

Stopping Criteria

We keep splitting the tree at each node until we reach a node that has fewer than some minimum number of data points/ observations. [8]

$$\max\{|R_j|\}_{j=1}^{J} \geq n_{stop}$$

2.2.2 Ensemble Methods

Ensemble methods are a machine learning technique in which we get our predictions from multiple models and aggregate them to get the final prediction. They help to reduce over-fitting.

Bagging and Bootstrapping

Bagging is done in the ideal scenario in which we have B separate training datasets. We use each dataset to train for a prediction function. The prediction functions are $\hat{f}^1$, $\hat{f}^2$,. . .,$\hat{f}^B$ and we aggregate them to make the final prediction as: [6]

$$\hat{f}_{avg}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x)$$

Each prediction function is generated by a decision tree with high variance. Aggregating these predictions helps us to reduce the variance while the bias is not affected.

However, in the real world, data is expensive and we generally do not have access to several training datasets. [6] The solution to this is Bootstrapping.

Bootstrapping is the process of sampling multiple datasets from a single training dataset. [6]

Suppose, we have a training dataset $\mathcal{D}$ with ‘n’ observations.

We generate B different bootstrapped training sets independently.

Each bootstrapped training set is generated by sampling ‘n’ observations from $\mathcal{D}$ with replacement.

Then we construct B regression trees using the B bootstrapped training sets and average the predictions as in *bagging.* [6]

Random Forests

Random Forests can be considered to be an improvement over bagged trees because they help in decorrelating the trees. The decision trees produced by bagging are highly correlated due to strong attributes and as a result, the variance reduction by bagging is inhibited.[12]

Random Forests solve this problem by considering only a small sample of attributes at each node instead of all the attributes. As a general rule of thumb, if there are p attributes, Random Forest only considers $\lceil p/3 \rceil$ random attributes at each node. [6]

Therefore, the Random Forest method follows the procedure below: [6]

- We generate B decision trees with the help of B bootstrapped training sets
- Now, instead of considering all the attributes at each node, we only consider $\lceil p/3 \rceil$ random attributes at each node.
- We make the final prediction by taking the mean of the predictions from the B decision trees.

2.3 Ridge Regression

2.3.1 Linear Regression

Before we talk about Ridge Regression, it is necessary for us to first understand Linear Regression.

Let there be an input vector $X^T = (X_1, X_2, \dots, X_p)$. We want to predict a real-valued output Y. A Linear Regression model predicts the output as a linear function of the input: [11]

$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j. \tag{1}$$

Consider the training data to be $(x_1, y_1) \dots (x_N, y_N)$. Our goal is to estimate the parameters β. A popular way to do this is the *least squares method*, where we pick the coefficients $\beta = (\beta_0, \beta_1, \dots, \beta_p)^T$ to minimize the residual sum of squares: [11]

$$RSS(\beta) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

$$RSS(\beta) = \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij} \beta_j \right)^2. \tag{2}$$

To minimize the RSS, we denote the $N \times (p+1)$ matrix where each row is an input vector with a 1 in the first position (to account for the intercept) by $\mathbf{X}$. Likewise, the N-vector of training outputs is denoted by $\mathbf{y}$. Then we can write the residual sum of squares as: [11]

$$RSS(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta). \tag{3}$$

Differentiating with respect to β we get:

$$\frac{\partial RSS}{\partial \beta} = -2\mathbf{X}^T (\mathbf{y} - \mathbf{X}\beta). \tag{4}$$

We assume $\mathbf{X}$ to have full column rank which implies that $\mathbf{X}^T\mathbf{X}$ is positive definite. And now, we set the first derivative to zero:

$$\mathbf{X}^T (\mathbf{y} - \mathbf{X}\beta) = 0 \tag{5}$$

Thus solving for β gives us the unique solution: [11]

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{6}$$

Now that we have calculated the coefficients $\hat{\beta}$, we can write the prediction function as follows:

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{7}$$

2.3.2 Why Ridge Regression?

Ridge Regression is an alternative linear model used to reduce overfitting. The coefficients (β) are chosen not only so that they predict well on the training data, but also to fit an additional constraint.

We try to reduce the magnitude of the coefficients to be very small i.e. as close to zero. This means that we try to have every feature decrease its effect on the outcome while still making a good prediction. The constraint used by Ridge Regression is called L2 *regularization*. Regularization means explicitly restricting a model so as to reduce overfitting. [16]

In Ridge Regression, we minimize a penalized residual sum of squares: [11]

$$RSS_{ridge}(\beta) = \sum_{i=1}^{N}\left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2. \tag{8}$$

$\lambda\sum_{j=1}^{p}\beta_j^2$ is called the *shrinkage penalty* and λ is called the *regularization parameter*.

In matrix form, (8) is written as:

$$RSS_{ridge}(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\beta^T\beta. \tag{9}$$

Solving for the coefficients β, we get the unique solution: [11]

$$\hat{\beta}_{ridge} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{10}$$

Thus, we write the prediction function as

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta_{ridge}} = \mathbf{X}(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{11}$$

2.4 Kernelized Ridge Regression

2.4.1 What is a Kernel?

A kernel is a function that quantifies the similarity between two vectors. It returns a scalar that happens to be equivalent to the dot product of the two vectors in some n-dimensional space. In simple terms, a kernel function tells us how close the two vectors are in some n-dimensional space without having to map the vectors to that space.

Let there be two vectors x, $z \in \mathcal{X}$(some feature space).

Let $\phi : \mathcal{X} \to \mathcal{H}$ be a mapping from $\mathcal{X}$ to some higher dimensional space equipped with dot product $\mathcal{H}$ (formally, a Hilbert space).
Then a kernel function is defined as: [22]

$$k(x, z) = \langle \phi(x), \phi(z) \rangle$$

2.4.2 The Kernel Trick

If we write our machine learning algorithm in such a way that our original feature space x only appears in dot products, we can replace all the dot products with kernels. This allows us to operate in the original feature space without having to transform it into a higher dimensional space. We do not need to know what $\phi(x)$ is which helps us save computational costs. This is known as **the kernel trick**. [15]

2.4.3 Examples of Kernels

Some common types of kernels are: [15]

Linear Kernel

If $\phi(x) = x$, we get the linear kernel defined by:

$$k(x, x') = \langle (x), (x') \rangle = x^T x'.$$

Polynomial Kernel

$$k(x, x') = (\gamma \langle (x), (x') \rangle + c)^d.$$

Radial Kernel

$$k(x, x') = \exp(-\gamma ||x - x'||^2).$$

2.4.4 Kernelizing Ridge Regression

In this section, we apply the kernel trick to the ridge regression model. More details can be found in [15] page 492.

Solution: Primal Form

Suppose we have a feature vector $x \in \mathbb{R}^D$ and its corresponding $N \times D$ design matrix. For the ridge regression problem, we want to minimize the following loss function:[15]

$$RSS(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda||\beta||^2. \tag{12}$$

And the optimal solution is given by:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{13}$$

which can be written as:

$$\hat{\beta} = (\sum_i \mathbf{x}_i\mathbf{x}_i^T + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{14}$$

Solution: Dual Form

We observe that the primary solution is not in the form of dot products. Using the matrix identity $\mathbf{A}(\mathbf{A}^T\mathbf{A}+\lambda\mathbf{I})^{-1} = (\mathbf{A}\mathbf{A}^T+\lambda\mathbf{I})^{-1}\mathbf{A}$, [13] we rewrite equation (13) as:

$$\hat{\beta} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})^{-1}\mathbf{y}. \tag{15}$$

We can now partially kernelize this equation by replacing $\mathbf{X}\mathbf{X}^T$ with the Gram Matrix $\mathbf{K}$ giving us

$$\hat{\beta} = \mathbf{X}^T(\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{y}. \tag{16}$$

Let us define a dual variable $\mathbf{r}$ such that:

$$\mathbf{r} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{y}.$$

Then, we can write (16) as follows:

$$\hat{\beta} = \mathbf{X}^T\mathbf{r} = \sum_{i=1}^{N} r_i\mathbf{x}_i. \tag{17}$$

Finally, the prediction function is defined as:

$$\hat{\mathbf{y}} = \hat{\beta}^T \mathbf{x} = \sum_{i=1}^{N} r_i \mathbf{x}_i^T \mathbf{x} = \sum_{i=1}^{N} r_i k(\mathbf{x}_i, \mathbf{x}). \tag{18}$$

2.5 Performance Metric

Just like the `scikit-learn` library, we measure the performance of our models by computing the coefficient of determination, also known as the R^2 score.

As described in [20], the R^2 score represents the proportion of variance (of the label) that has been explained by the independent variables in the model. It provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of explained variance.

The best R^2 score that we can obtain is 1.0. The score can be negative for a model that completely fails to fit the data. A score of 0.0 is achieved when there is a constant model which always predicts the average label without considering the input.

Let there be n samples in the test set. Let $\hat{y}_i$ be the predicted label for the i-th sample and let y_i be its corresponding true label. Then the test R^2 score is given by:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

$\bar{y}$ is the average label in the test set given by $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$

2.6 Online Prediction

In an online setting, the data points are usually available to us one at a time as opposed to in a group (batch learning). We get a single data point, make our prediction, and then observe the true label. Consequently, we measure our loss and then receive the next data point.

Here we try to implement Linear Regression in an online setting. In order to do so, we follow the workflow below: [18]

- Initialize the weights vector to 0 at time $t = 1$

$$\mathbf{w}_1 = [0, 0, ..., 0]$$

- Choose a learning rate $\eta > 0$
- For $t = 1, \ldots, T$:
  - Get $\mathbf{x}_t \in \mathbb{R}^n$ and add it to the training set.
  - Perform feature scaling on the training set with `StandardScaler`
  - Predict the output with $\hat{y}_t = \mathbf{w}_t \cdot \mathbf{x}_t \in \mathbb{R}$
  - Observe true label $y_t \in \mathbb{R}$
  - Update the weights as $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta(\mathbf{w}_t \cdot \mathbf{x}_t - y_t)\mathbf{x}_t$
- Calculate cumulative loss as the sum of individual losses:

$$\mathbf{L} = \sum_{t=1}^{T} (\hat{y}_t - y_t)^2$$

This algorithm is known as the *Widrow-Hoff (WH)* or *Least Mean Squares (LMS)* Algorithm. [18]

3 Data Preprocessing

3.1 Boston Housing Dataset

The Boston housing data set contains information collected by the U.S Census Service concerning housing in the area of Boston, Massachusetts, USA [17]. It has only 506 samples and is a fairly simple dataset to preprocess. A complete description of its columns can be found at `http://lib.stat.cmu.edu/datasets/boston`

- **Loading the Data:** We load the data into a `pandas` DataFrame and call it `boston`

- **Check missing values:** Then we check if the dataset contains any missing values:

  ```
  boston.isnull().sum()
  ```

  We get the following output and observe that the dataset has no missing values:

  ```
  CRIM       0
  ZN         0
  INDUS      0
  CHAS       0
  NOX        0
  RM         0
  AGE        0
  DIS        0
  RAD        0
  TAX        0
  PTRATIO    0
  B          0
  LSTAT      0
  MEDV       0
  dtype: int64
  ```

  Thus, we can move on to the next step without needing any further action.

- **Correlation matrix:** We plot the correlation matrix for the dataset to observe which features have the most influence on the price (`MEDV`).

```
fig, ax = plt.subplots(figsize = (15,10))
ax = sns.heatmap(boston.corr(), annot = True);
```

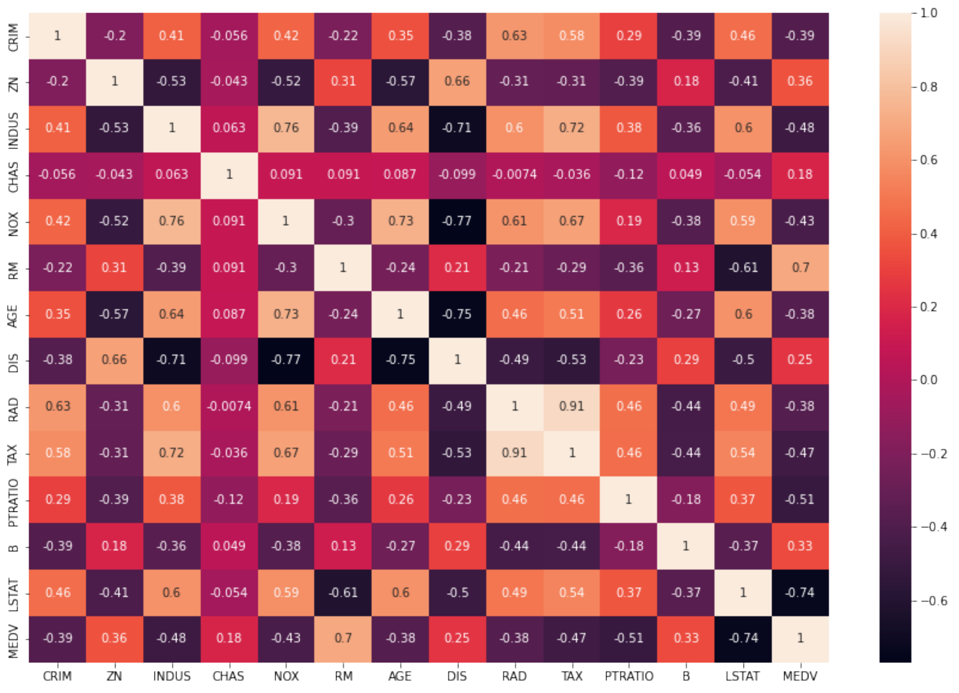


Figure 1: Correlation Matrix for the Boston Housing Dataset

We see that feature `RM` has the most influence on the price of the house (`MEDV`). This is expected as the price of a house would increase with the number of rooms it has.

- **Splitting into Features and Labels:** Now, we split `boston` into features (`X`) and labels (`y`). `MEDV` is our target column, so we split it accordingly.

```
X_boston = boston.drop('MEDV', axis = 1)
y_boston = boston['MEDV']
```

- **Training and Test Sets:** Next, we split `X` and `y` into training and test sets:

```
X_train, X_test, y_train, y_test =
train_test_split(X_boston,
                 y_boston,
                 random_state = 42)
```

- **Feature Scaling:** Then we scale the features so that they are measured on the same scale. We do so with the help of the `StandardScaler` class in `sklearn`'s `preprocessing` library:

  ```
  from sklearn.preprocessing import StandardScaler
  scaler = StandardScaler()

  X_train_scaled = scaler.fit_transform(X_train)
  X_test_scaled = scaler.transform(X_test)
  ```

  Then convert the scaled columns back into a `pandas` dataframe:

  ```
  X_train_scaled_df = pd.DataFrame(X_train_scaled,
  columns = X_train.columns)

  X_test_scaled_df = pd.DataFrame(X_test_scaled,
  columns = X_test.columns)
  ```

  `StandardScaler` resets the indices of the samples in the `X_train` and `X_test`. So, it is crucial that we reset the indices for `y_train` and `y_test` as well so that they correspond to each other:

  ```
  y_train.reset_index(drop = True, inplace=True)
  y_test.reset_index(drop = True, inplace=True)
  ```

Now that we have preprocessed the dataset, we are ready to apply our models to it.

3.2 California Housing Dataset

The California Housing Dataset contains housing data drawn from the 1990 U.S. Census. A complete description of its columns can be found at `https://developers.google.com/machine-learning/crash-course/california-housing-data-description`

- **Loading the Data:** We load the data into a `pandas` DataFrame and call it `california`.
- **Check missing values:** First, we check for any missing values in the dataset:

  ```
  california.isna().sum()
  ```

  We get the following output where we observe that 207 rows have missing values:

  ```
  longitude              0
  latitude               0
  housing_median_age     0
  total_rooms            0
  total_bedrooms       207
  population             0
  households             0
  median_income          0
  median_house_value     0
  ocean_proximity        0
  dtype: int64
  ```

  Since the size of this dataset is very large, we choose to remove the rows with missing values:

  ```
  california = california.dropna()
  ```

- **Correlation Matrix**: We plot the correlation matrix for the dataset to see the influence the features have on the price of the house (`median_house_value`).

  ```
  fig, ax = plt.subplots(figsize=(15,10))
  ax = sns.heatmap(california.corr(),annot=True);
  ```

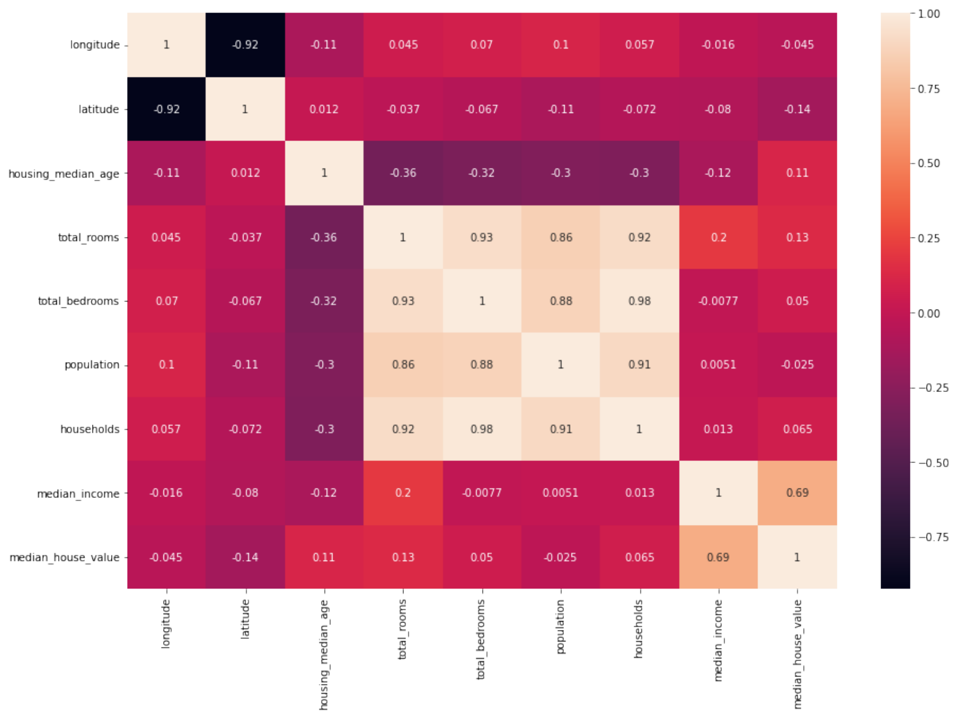


Figure 2: Correlation Matrix for the California Housing Dataset

We observe that `median_income` has the most influence on the `median_house_value` which makes sense considering that people with a high income are more likely to live in expensive homes.

- **Handling Categorical Variables:** We notice that `ocean_proximity` is a categorical variable so we encode it with help of dummy variables:

```
california['ocean_proximity'].value_counts()
```

```
<1H OCEAN     9034
INLAND        6496
NEAR OCEAN    2628
NEAR BAY      2270
ISLAND           5
Name: ocean_proximity, dtype: int64
```

Encoding:

```
california = pd.get_dummies(california)
california.head()
```

This is what we get after encoding the categorical variable `ocean_proximity`:

ocean_proximity_<1H OCEAN	ocean_proximity_INLAND	ocean_proximity_ISLAND	ocean_proximity_NEAR BAY	ocean_proximity_NEAR OCEAN
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0

- **Splitting into Features and Labels:** Now, we split `california` into features (`X`) and labels (`y`). `median_house_value` is our target column, so we split it accordingly.

```
X_california = california.drop('median_house_value', axis = 1)
y_california = california['median_house_value']
```

- **Training and Test Sets** Next, we split `X` and `y` into training and test sets:

```
X_train, X_test, y_train, y_test =
train_test_split(X_california,
                 y_california,
                 random_state = 42)
```

- **Feature Scaling:** We use `StandardScaler` to scale the features. However, in this instance, we want to fit `StandardScaler` only up to the `median_income` column. This is because the rest of the columns are dummy variables for the categorical feature `ocean_proximity`.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train.iloc[:, :n])
```

Here, `n` = 1 + index of `median_income`.

Similarly, we only want to transform the columns up to the `median_income` column. We do so with a helper function:

```
def custom_transform(X):
    temp = np.copy(X)
    temp[:, :n] = scaler.transform(temp[:, :n])
    return temp

X_train_scaled = custom_transform(X_train)
X_test_scaled = custom_transform(X_test)
```

Then convert the scaled columns back into a `pandas` dataframe:

```
X_train_scaled_df = pd.DataFrame(X_train_scaled,
columns = X_train.columns)

X_test_scaled_df = pd.DataFrame(X_test_scaled,
columns = X_test.columns)
```

As noted earlier, `StandardScaler` resets the indices of the samples in the `X_train` and `X_test`. So, it is crucial that we reset the indices for `y_train` and `y_test` as well so that they correspond to each other:

```
y_train.reset_index(drop = True, inplace=True)
y_test.reset_index(drop = True, inplace=True)
```

Now that we have preprocessed the dataset, we are ready to apply our models to it.

3.3 Ames Housing Dataset

The Ames Housing Dataset contains information on the houses sold in Ames, Iowa, USA from 2006 to 2010. A complete description of the dataset and its features can be found at `http://jse.amstat.org/v19n3/decock/DataDocumentation.txt`

- **Loading the Data:** We load the data into a `pandas` DataFrame and call it `ames`.

- **Check missing values:** We check for any missing values in the dataset. Since this dataset contains a lot of missing values, we calculate the percentage of missing values for each feature.

  `Pool QC` has 2917 missing values. According to the description of the dataset, a missing value for this feature indicates that the house does not have a pool. Therefore, we fill in the missing values with the value 'Np' which indicates so.

  ```
  ames['Pool QC'].fillna('Np', inplace=True)
  ```

  `Misc Feature` has 2824 missing values. According to the description of the dataset, a missing value for this feature indicates that the house does not have any miscellaneous features. Therefore, we fill in the missing values with the value 'None' which indicates so.

  ```
  ames['Misc Feature'].fillna('None', inplace=True)
  ```

  `Alley`, `Fence`, and `Fireplace Qu` have 2732, 2358, and 1422 missing values respectively. We fill in the missing values with the value 'No Alley', 'No Fence', and 'No Fireplace' which indicate so.

  ```
  ames['Alley'].fillna('No Alley', inplace = True)
  ames['Fence'].fillna('No Fence', inplace = True)
  ames['Fireplace Qu'].fillna('No Fireplace', inplace = True)
  ```

  `Lot Frontage` denotes the linear feet of the street connected to the property. A missing value for it means that there are no streets connected to the house. Hence we fill in `0.0` for the missing values.

  Missing values for `Garage Type`, `Garage Finish`, `Garage Qual` and `Garage Cond` denote that the house doesn't have a garage. So we fill in the missing values with 'No Garage'.

```
for col in ['Garage Type', 'Garage Finish',
'Garage Qual', 'Garage Cond']:
    ames[col].fillna('No Garage', inplace=True)
```

Missing values for `Garage Cars`, `Garage Area`, and `Garage Yr Blt` denote that the house doesn't have a garage. So we fill in the missing values with `0`.

```
ames['Garage Cars'].fillna(0, inplace = True)
ames['Garage Area'].fillna(0.0, inplace = True)
ames['Garage Yr Blt'].fillna(0, inplace = True)
```

Missing values for `Bsmt Exposure`, `BsmtFin Type 2`, `BsmtFin Type 1`, `Bsmt Qual`, and `Bsmt Cond` denote that the house does not have a basement. So we fill in the missing values with `No Basement`.

```
for col in ['Bsmt Exposure', 'BsmtFin Type 2',
'BsmtFin Type 1', 'Bsmt Qual', 'Bsmt Cond']:
    ames[col].fillna('No Basement', inplace = True)
```

Missing values for `Bsmt Half Bath`, `Bsmt Full Bath`, `Total Bsmt SF`, `Bsmt Unf SF`, `BsmtFin SF 2`, and `BsmtFin SF 1` are filled with `0`.

```
for col in ['Bsmt Half Bath', 'Bsmt Full Bath',
'Total Bsmt SF','Bsmt Unf SF', 'BsmtFin SF 2', 'BsmtFin SF 1']:
    ames[col].fillna(0, inplace = True)
```

Missing values for `Mas Vnr Area`, and `Mas Vnr Type` are filled with `0` and `None` respectively.

```
ames['Mas Vnr Area'].fillna(0, inplace = True)
ames['Mas Vnr Type'].fillna('None', inplace = True)
```

The `Electrical` column has only one missing value. So, we fill it with the mode of the column.

```
ames['Electrical'].fillna(ames['Electrical'].mode()[0], inplace=True)
```

- **Removing Irrelevant Columns:** Then, we remove some unnecessary columns like `order` and `PID` which do not have any effect on the price of the house.

```
ames.drop(columns=['Order', 'PID'], axis = 1, inplace = True)
```

- **Correlation Matrix:** We plot the correlation matrix for the dataset to see which features highly influence `SalePrice`.

  ```
  fig, ax = plt.subplots(figsize=(30,30))
  ax = sns.heatmap(ames.corr(), annot = True);
  ```

  The Figure is available in page 28.

  Enlarged image can be viewed at `https://drive.google.com/file/d/1M4YTX1QUpoYtpTLlSc_34_Nyu4LPJG9M/view?usp=sharing`

  We observe that `Overall Qual` and `Gr Liv Area` have the most effect on the `SalePrice`. This is expected as the price would normally be higher for a house with high overall quality and a large living area.

- **Handling Ordinal Variables:** Next, we deal with ordinal variables. In order to encode them, we map them to certain numbers for each category. Variables having the same categories are encoded using the same mapping.

  `Exter Cond`, `Exter Qual`, `Kitchen Qual`, and `Heating QC` have the same categories and so use the same mapping.

  ```
  ordinal_mapping_1 = {'Ex': 4, 'Gd': 3, 'TA': 2, 'Fa': 1, 'Po': 0}

  ames['Exter Cond'] = ames['Exter Cond'].map(ordinal_mapping_1)
  ames['Exter Qual'] = ames['Exter Qual'].map(ordinal_mapping_1)
  ames['Kitchen Qual'] = ames['Kitchen Qual'].map(ordinal_mapping_1)
  ames['Heating QC'] = ames['Heating QC'].map(ordinal_mapping_1)
  ```

  Similarly for `Bsmt Cond` and `Bsmt Qual`, we have:

  ```
  ordinal_mapping_2 = {'Ex': 5, 'Gd':4, 'TA':3, 'Fa':2,
  'Po':1, 'No Basement':0}

  ames['Bsmt Cond'] = ames['Bsmt Cond'].map(ordinal_mapping_2)
  ames['Bsmt Qual'] = ames['Bsmt Qual'].map(ordinal_mapping_2)
  ```

  For `BsmtFin Type 1` and `BsmtFin Type 2`, we have:

  ```
  ordinal_mapping_3 = {'GLQ':6, 'ALQ':5, 'BLQ':4, 'Rec':3,
  'LwQ':2, 'Unf':1, 'No Basement':0}
  ```

```
ames['BsmtFin Type 1'] = ames['BsmtFin Type 1'].map(ordinal_mapping_3)
ames['BsmtFin Type 2'] = ames['BsmtFin Type 2'].map(ordinal_mapping_3)
```

For `Garage Qual` and `Garage Cond`, we have:

```
ordinal_mapping_4 = {'Ex': 5, 'Gd':4, 'TA':3,
'Fa':2, 'Po':1, 'No Garage':0}

ames['Garage Qual'] = ames['Garage Qual'].map(ordinal_mapping_4)
ames['Garage Cond'] = ames['Garage Cond'].map(ordinal_mapping_4)
```

The rest of the ordinal features each have unique categories and hence are mapped as follows:

```
ames['Bsmt Exposure'] = ames['Bsmt Exposure'].map({'Gd':4,
Av':3, 'Mn':2, 'No':1, 'No Basement':0})

ames['Garage Finish'] = ames['Garage Finish'].map({'Fin':3,
'RFn':2, 'Unf':1, 'No Garage':0})

ames['Land Slope'] = ames['Land Slope'].map({'Sev':2, 'Mod':1, 'Gtl':0})

ames['Pool QC'] = ames['Pool QC'].map({'Ex': 4, 'Gd':3,
'TA':2, 'Fa':1, 'Np':0})

ames['Fireplace Qu'] = ames['Fireplace Qu'].map({'Ex': 5, 'Gd':4,
'TA':3, 'Fa':2, 'Po':1, 'No Fireplace':0})

ames['Fence'] = ames['Fence'].map({'GdPrv':4, 'MnPrv':3,
'GdWo':2, 'MnWw':1, 'No Fence':0})

ames['Functional'] = ames['Functional'].map({'Typ':7, 'Min1':6,
'Min2':5, 'Mod':4, 'Maj1':3, 'Maj2':2, 'Sev':1, 'Sal':0})

ames['Central Air'] = ames['Central Air'].map({'Y':1, 'N':0})
```

- **Handling Non-Ordinal Variables:** The other non-ordinal categorical variables are encoded with the help of dummy variables.

```
ames = pd.get_dummies(ames)
```

- **Splitting into Features and Labels:** Now, we split `ames` into features (`X`) and labels (`y`). `SalePrice` is our target column, so we split it accordingly.

  ```
  X_ames = ames.drop('SalePrice', axis = 1)
  y_ames = ames['SalePrice']
  ```

- **Training and Test Sets:** Next, we split `X` and `y` into training and test sets:

  ```
  X_train, X_test, y_train, y_test =
  train_test_split(X_ames,
                   y_ames,
                   random_state = 42)
  ```

- **Feature Scaling:** We use `StandardScaler` to scale the features. However, here we want to fit `StandardScaler` only up to the `Yr Sold` column. This is because the rest of the columns are dummy variables for the categorical features.

  ```
  from sklearn.preprocessing import StandardScaler
  scaler = StandardScaler()
  scaler.fit(X_train.iloc[:, :n])
  ```

  Here, `n` = 1 + index of `Yr Sold`.

  Similarly, we only want to transform the columns up to the `Yr Sold` column. We do so with a helper function:

  ```
  def custom_transform(X):
      temp = np.copy(X)
      temp[:, :n] = scaler.transform(temp[:, :n])
      return temp

  X_train_scaled = custom_transform(X_train)
  X_test_scaled = custom_transform(X_test)
  ```

  Then convert the scaled columns back into a `pandas` dataframe:

  ```
  X_train_scaled_df = pd.DataFrame(X_train_scaled,
  columns = X_train.columns)

  X_test_scaled_df = pd.DataFrame(X_test_scaled,
  columns = X_test.columns)
  ```

As noted earlier, `StandardScaler` resets the indices of the samples in the `X_train` and `X_test`. So, it is crucial that we reset the indices for `y_train` and `y_test` as well so that they correspond to each other:

```
y_train.reset_index(drop = True, inplace=True)
y_test.reset_index(drop = True, inplace=True)
```

Now that we have preprocessed the dataset, we are ready to apply our models to it.

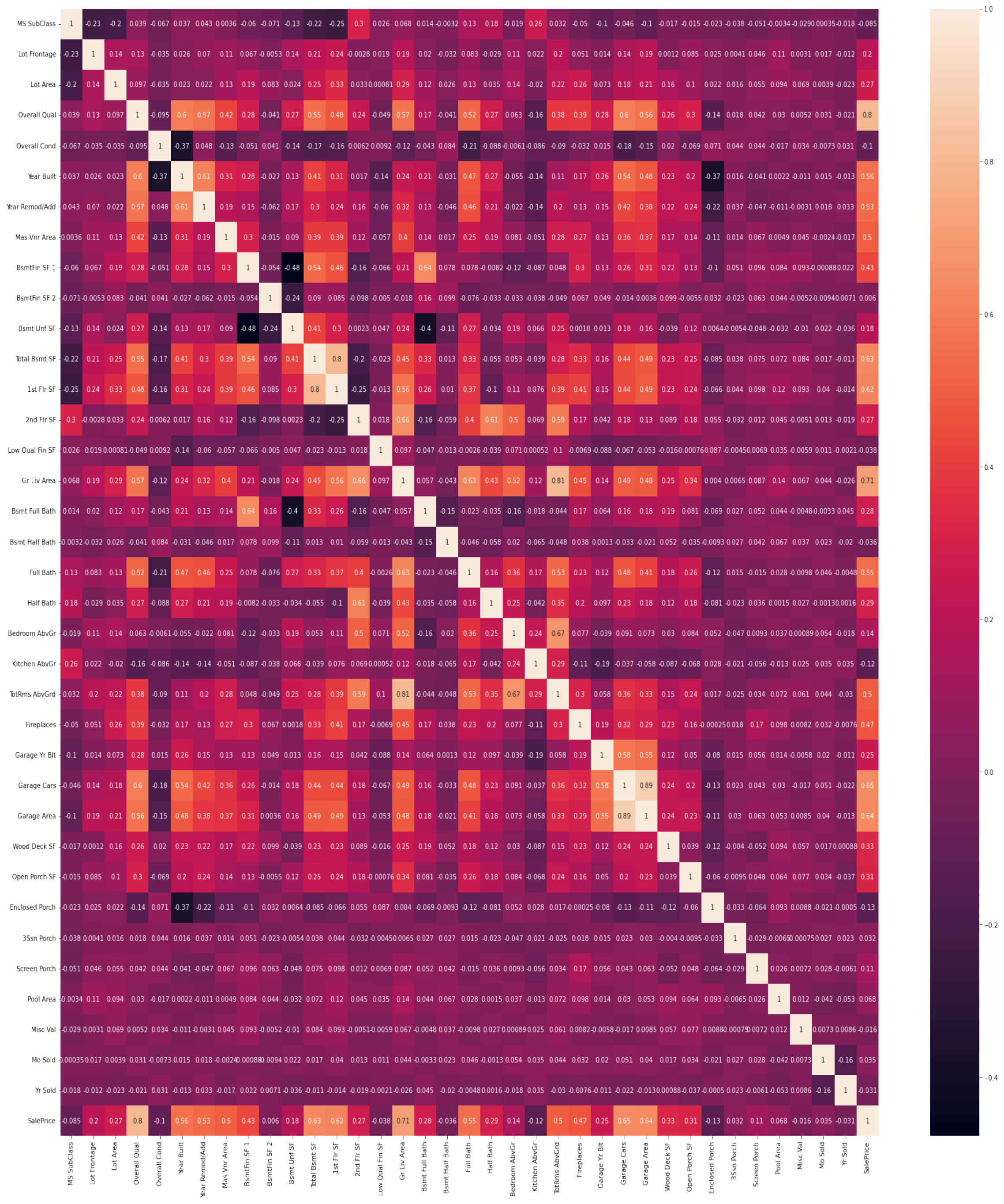


Figure 3: Correlation Matrix for the Ames Housing Dataset

4 Applying our Models

4.1 Boston Housing Dataset

4.1.1 K-Nearest Neighbors

We apply the K-Nearest Neighbors model to the Boston Housing Dataset with a bunch of possible combinations of parameters and end up with the following result:

```
Best value for n_neighbors: 2
Best distance function: 2
Weighted: False
Best Score: 0.7742672278926411
```

We find that the model with the best performance for this dataset is an **Unweighted 2-Nearest Neighbors Model** using **Euclidean Distance**.

Now, we take the model above and vary the number of neighbors to see how it affects the R^2 score:

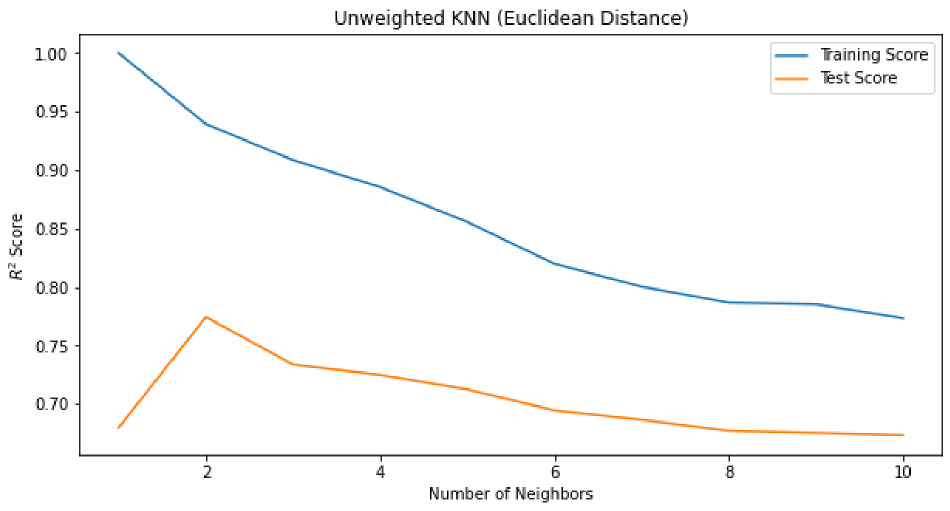


Figure 4: R^2 Score vs Number of Neighbors

In the graph above, we observe that when we only consider one nearest neighbor, the model tends to overfit. As the number of neighbors is increased beyond 2, the model starts to underfit and the training and the test scores both start to decrease.

4.1.2 Random Forest

First, we calculate the maximum number of features to be considered at each node. As a general rule of thumb, it is calculated as:

```
p = int(np.ceil(X_train.shape[1]/3))
```

where `X_train.shape[1]` is the number of features in the dataset.

Then we apply the Random Forest Model to the Boston Housing Dataset with a range of possible combinations of its parameters and search for the model with the best performance. We end up with the following result:

```
Best value for n_estimators: 100
Best value for min_samples_split: 5
Best value for max_depth: 25
Best Score: 0.8517834812317369
```

Now, we take the model above and vary the number of trees to see how it affects the score:

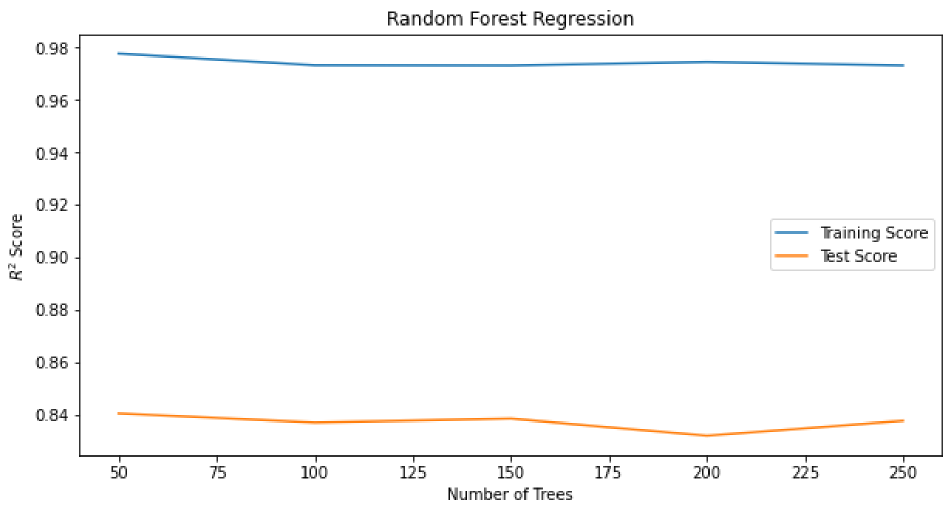


Figure 5: R^2 Score vs Number of Trees

We observe that `n_estimators` does not drastically affect the model performance. This means that choosing a large number of trees in a random forest model is not the best idea. This helps us to save computational complexity costs [19].

Now, we vary the number of minimum samples required in a node for splitting to see how it affects the score:

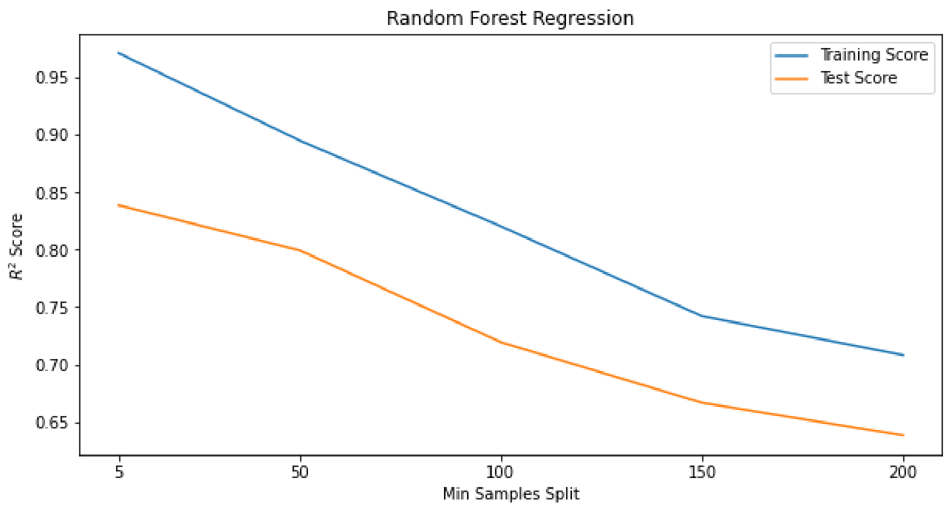


Figure 6: R^2 Score vs Minimum Samples Per Split

Here, we observe that as `min_samples_split` increases the model starts to underfit due to the fact that the minimum number of samples required for splitting a node is so high that there are no significant splits observed. Therefore, we must not choose very high values for `min_samples_split` [19].

Now, we take the model above and vary the maximum depth of the tree to see how it affects the score:

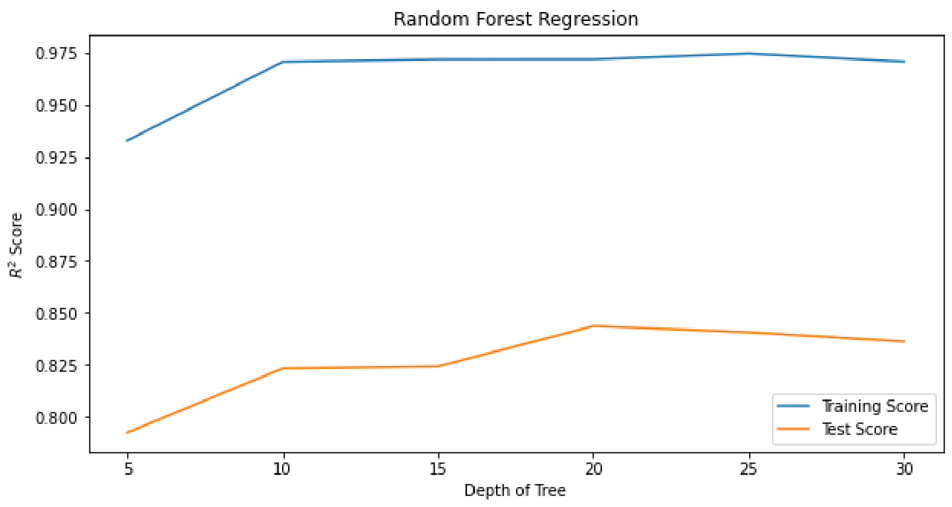


Figure 7: R^2 Score vs Depth of the Tree

We observe that the test scores increase as the depth of the tree increases up to a certain extent. After that, we see that the test score starts decreasing. There it's not necessary for us to set a very high depth for the trees [19].

4.1.3 Ridge

We fit our training data to a Ridge Regression Model with different values of the regularization parameter `alpha` and get the following result:

```
Best value of alpha: 0.1
Best Score: 0.6844008362928345
```

Below is a graph that shows how the scores are affected with respect to the regularization parameter alpha:

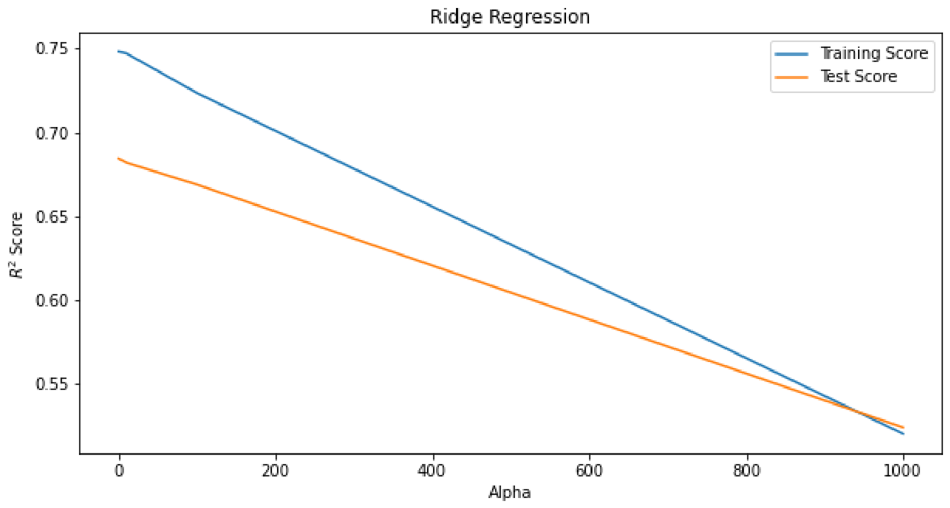


Figure 8: R^2 Score vs Value of Alpha

We observe that as the value of `alpha` is increased, the model starts to underfit which leads to poor training and test scores. Therefore, we must choose a small value for the regularization parameter `alpha`.

4.1.4 Kernelized Ridge

We fit our training data to the Kernelized Ridge Regression Model using three different kernels. The R^2 scores achieved are as follows:

- Linear Kernel: `-7.010432219940345`
- Polynomial Kernel: We try a bunch of possible combinations for its parameters and end up with the following:

```
        Best Degree: 3
        Best Coef0: 2
        Best Score: 0.8286687606504263
```

- Radial Basis Kernel: `0.7326765716896875`

4.1.5 Comparing Model Performance

Let us plot the scores achieved by our various models on the Boston Housing Dataset:

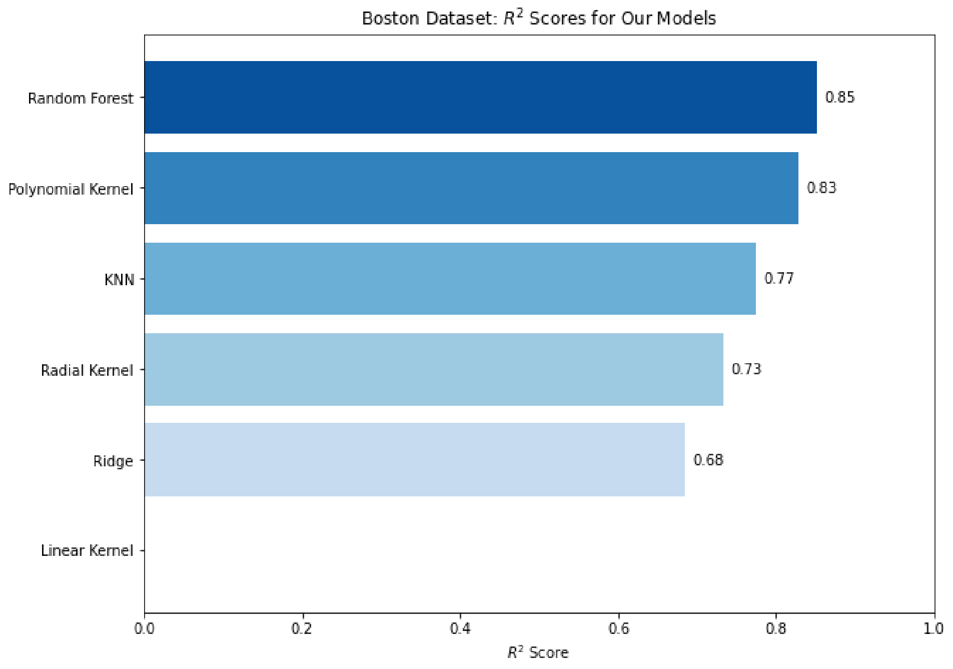


Figure 9: Comparing our Models (R^2 Scores)

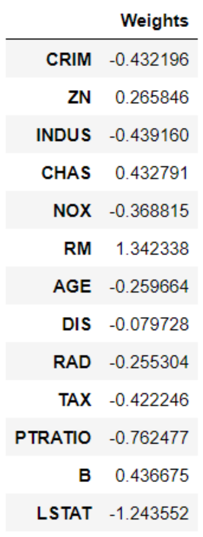
We see that the **Random Forest Regression** model performs the best on the Boston Housing Dataset. It seems to be able to generalize its predictions with a R^2 score ≈ 0.85. However, the model is not interpretable. It makes good predictions we cannot clearly see how it is making them.

Kernelized Ridge Regression using the **Polynomial Kernel** with a test R^2 score ≈ 0.83 performs well too but suffers from the same problem of being not interpretable.

The **K-Nearest Neighbors** model performs okay with an R^2 Score ≈ 0.77. This method is easy to understand and is a good baseline method to try before moving on to more complex models [16].

The **Kernelized Ridge model** with **Radial Basis Kernel** gives us a fair generalization accuracy as well with a test R^2 score around 0.73.

The **Ridge Regression** model does not perform very well on this dataset but it is interpretable. By looking at the weights of the features, we can see which features have a significant effect on the price.



	Weights
CRIM	-0.432196
ZN	0.265846
INDUS	-0.439160
CHAS	0.432791
NOX	-0.368815
RM	1.342338
AGE	-0.259664
DIS	-0.079728
RAD	-0.255304
TAX	-0.422246
PTRATIO	-0.762477
B	0.436675
LSTAT	-1.243552

We can see that the number of rooms `RM` has the most positive effect on the price `MEDV`. This makes sense as the price of the house would increase with the number of rooms it has.

The percentage of lower status population `LSTAT` has the most negative effect on `MEDV`. This is expected too. Similarly, the other weights seem sensible as well.
Finally, **Kernelized Ridge with the Linear Kernel** fails to capture the patterns in the data and gives us a very poor test R^2 score ≈ -7.01

4.1.6 Online Prediction

In order to simulate an online scenario, we assume that the 506 samples of the Boston Housing Dataset are available to us one at a time. We run a `for` loop from `t = 1,...,506` where `t` represents an instance of time (which could be hours, days, weeks, etc.)

Then, we make a prediction for each sample that we observe based on the samples that we have observed so far. After every prediction, we also observe the true label, calculate our loss and update the weights.

```
for t in range(T):
    # Observe the sample at time t and make a prediction
    prediction = olr.predict(X_boston.iloc[t])
    # Store the prediction that we have made
    y_preds.append(prediction)
    # Calculate Loss and Update Weights
    olr.update_weights(y_boston[t], prediction)
```

Here `olr` is an instance of the class `OnlineLinearRegression`. Its implementation can be found in the file `OnlineLR.py`.

After we have made a prediction for each of the samples we have received, we plot a graph of our loss over time:

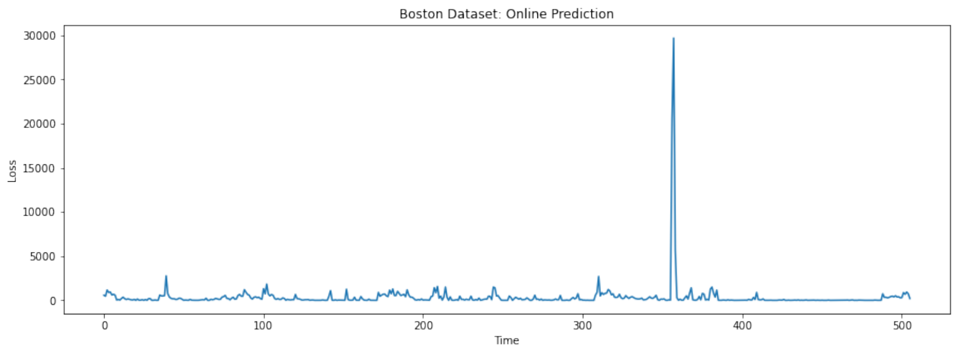


Figure 10: Loss Over Time (Online Prediction)

Also, our Cumulative Loss (L) = `178449.73573892968`

4.2 California Housing Dataset

4.2.1 K-Nearest Neighbors

We apply the K-Nearest Neighbors model to the California Housing Dataset with a bunch of possible combinations of parameters and end up with the following result:

```
That took 137.0 minute(s).

Best value for n_neighbors: 5
Best distance function: 1
Weighted: True
Best Score:0.7346569978090304
```

We find that the model with the best performance for this dataset is an **Weighted 5-Nearest Neighbors Model** using **Manhattan Distance**.

Now, we take the model above and vary the number of neighbors to see how it affects the R^2 score:

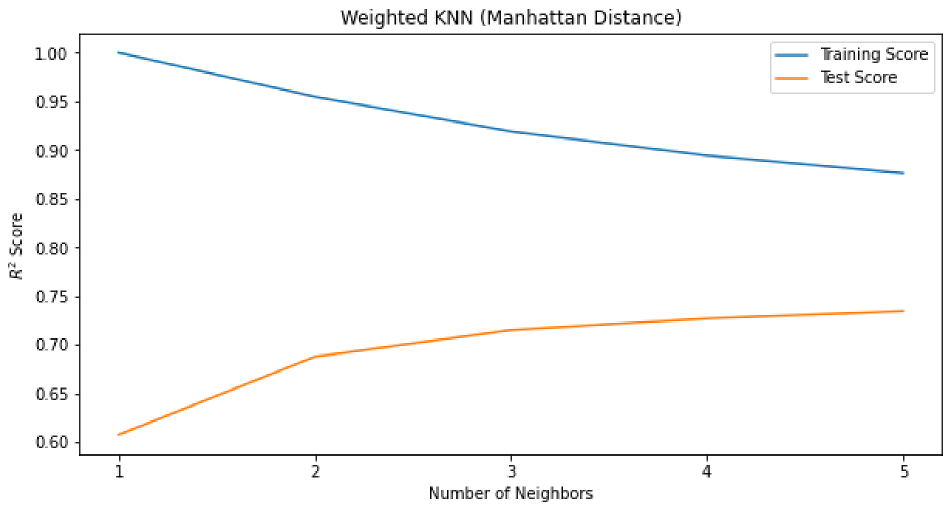


Figure 11: R^2 Scores vs Number of Neighbors

In the graph above, we observe that when we only consider one nearest neighbor, the model tends to overfit. As the number of neighbors is increased beyond 2, the model starts to underfit and the training and the test scores both start to decrease.

4.2.2 Random Forest

First, we calculate the maximum number of features to be considered at each node. As a general rule of thumb, it is calculated as:

```
    p = int(np.ceil(X_train.shape[1]/3))
```

where `X_train.shape[1]` is the number of features in the dataset.

Then we apply the Random Forest Model to the Boston Housing Dataset with a range of possible combinations of its parameters and search for the model with the best performance. To avoid complexity costs, we set `n_estimators` = 100, as we've seen earlier that the number of trees does not drastically affect the performance. We end up with the following result:

```
    That took 224.0 minute(s).

    Best value for min_samples_split: 5
    Best value for max_depth: 20
    Best Score: 0.8251726491389899
```

Now, we vary the number of minimum samples required in a node for splitting to see how it affects the score:

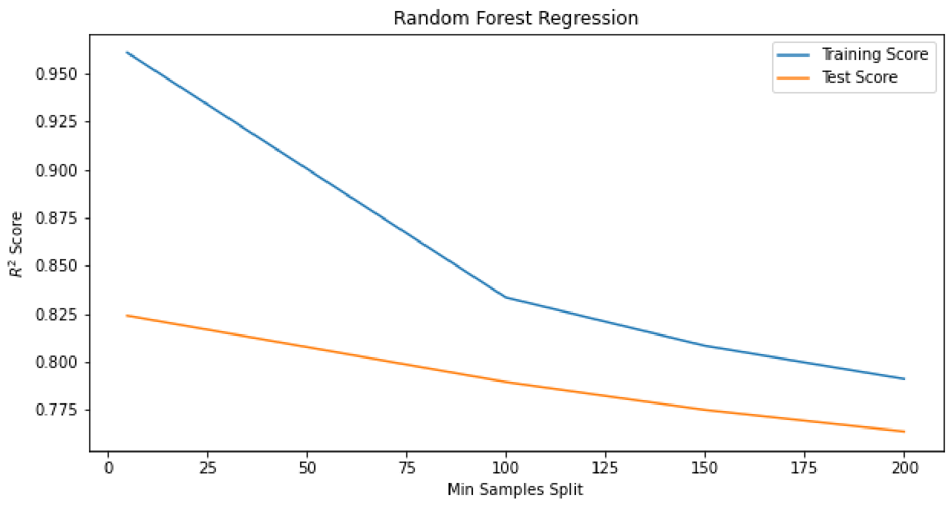


Figure 12: R^2 Scores vs Min Samples Per Split

Here, we observe that as `min_samples_split` increases the model starts to underfit due to the fact that the minimum number of samples required for splitting a node is so high that there are no significant splits observed. Therefore, we must not choose very high values for `min_samples_split` [19].

Now, we take the model above and vary the maximum depth of the tree to see how it affects the score:

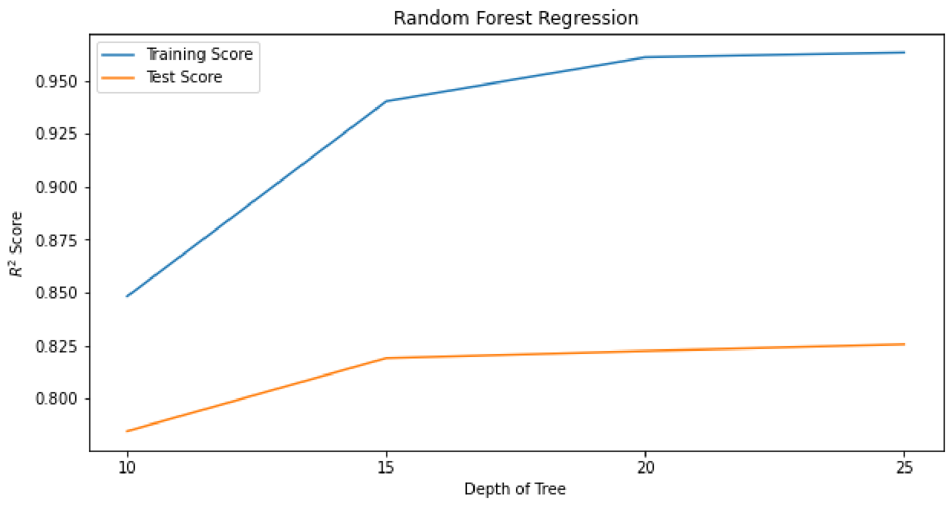


Figure 13: R^2 Scores vs Depth of the Tree

We observe that the test scores increase as the depth of the tree increases up to a certain extent. After that, we see do not see a drastic difference in the test score. [19].

4.2.3 Ridge

We fit our training data to a Ridge Regression Model with different values of the regularization parameter `alpha` and get the following result:

```
Best value of alpha: 10
Best Score: 0.6553892965223076
```

Below is a graph that shows how the scores are affected with respect to the regularization parameter alpha:

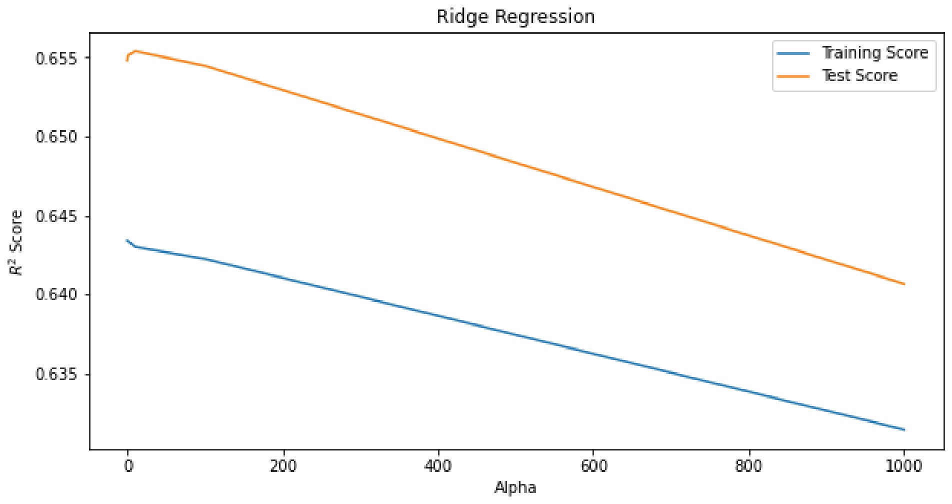


Figure 14: R^2 Scores vs Alpha

We observe that as the value of `alpha` is increased, the model starts to underfit which leads to poor training and test scores. Therefore, we must choose a small value for the regularization parameter `alpha`.

4.2.4 Kernelized Ridge

We fit our training data to the Kernelized Ridge Regression Model using three different kernels. The R^2 scores achieved are as follows:

- Linear Kernel: `0.6541877462647631`
- Polynomial Kernel: We try a bunch of possible combinations for its parameters and end up with the following:

```
That took 29.0 minute(s).
Best Degree: 4
Best Coef0: 2
Best Score: 0.7539057131726862
```

- Radial Basis Kernel: `0.7539057131726862`

4.2.5 Comparing Models

Similarly, let us plot the scores achieved by our various models on the California Housing Dataset:

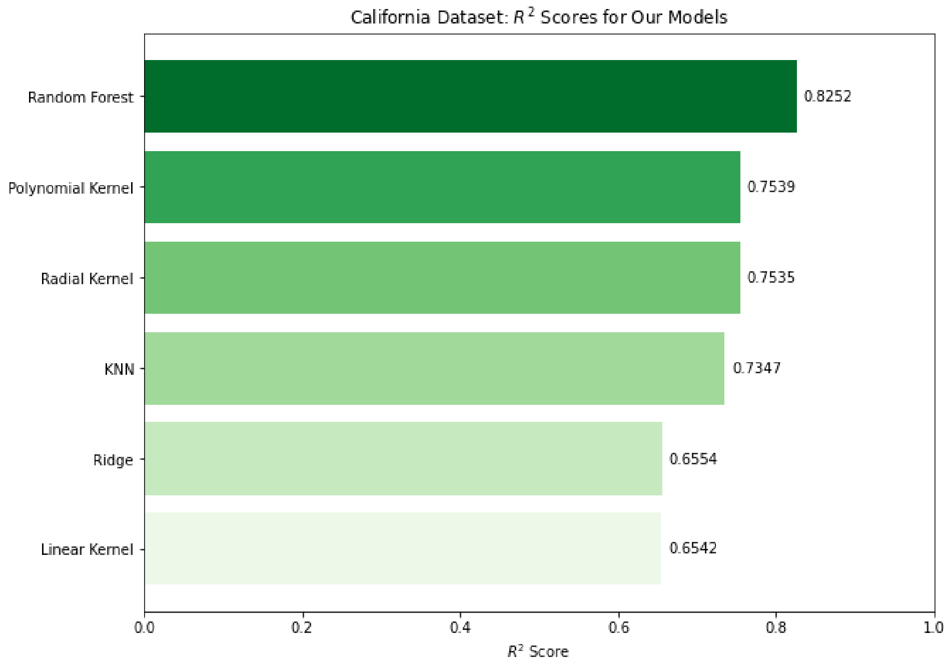


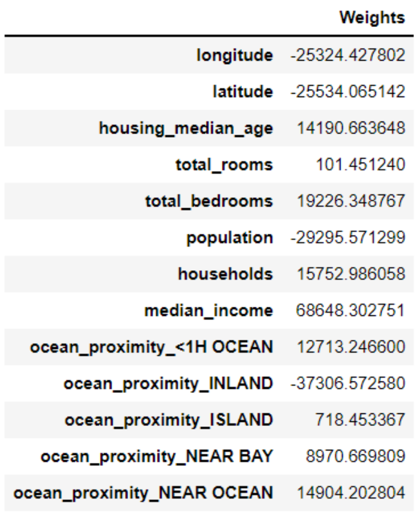
Figure 15: Comparing our Models (R^2 Scores)

We see that the **Random Forest Regression** model performs the best on the California Housing Dataset as well. It seems to be able to generalize its predictions with a R^2 score ≈ 0.82. However, again the model is not interpretable. The model is also very time-consuming to fit and takes a toll on the CPU.

Kernelized Ridge Regression using the **Polynomial Kernel** and the **Radial Kernel** with a test R^2 scores ≈ 0.75 perform fairly well but suffer from the same problem of being not interpretable.

The **K-Nearest Neighbors** model performs okay with an R^2 Score ≈ 0.73. It is a good baseline method to try before moving on to more complex models [16].

The **Ridge Regression** model does not perform too well on this dataset but it is interpretable. By looking at the weights of the features, we can see which features have a significant effect on the price.



	Weights
longitude	-25324.427802
latitude	-25534.065142
housing_median_age	14190.663648
total_rooms	101.451240
total_bedrooms	19226.348767
population	-29295.571299
households	15752.986058
median_income	68648.302751
ocean_proximity_<1H OCEAN	12713.246600
ocean_proximity_INLAND	-37306.572580
ocean_proximity_ISLAND	718.453367
ocean_proximity_NEAR BAY	8970.669809
ocean_proximity_NEAR OCEAN	14904.202804

We can see that `median_income` has the most positive effect on the price `median_house_value`. This makes sense as people with higher incomes would be living in more expensive houses.

The `ocean_proximity_INLAND` has the most negative effect on `median_house_value`. This is expected too as houses closer to the ocean are more expensive. Similarly, the other weights seem sensible as well.

Finally, **Kernelized Ridge with the Linear Kernel** performs okay with R^2 score ≈ 0.65

4.2.6 Online Prediction

In order to simulate an online scenario, we assume that the 20433 samples of the California Housing Dataset are available to us one at a time. We run a `for` loop from `t = 1,...,20433` where `t` represents an instance of time (which could be hours, days, weeks, etc.)

Then, we make a prediction for each sample that we observe based on the samples that we have observed so far. After every prediction, we also observe the true label, calculate our loss and update the weights.

```
for t in range(X_california.shape[0]):
    # Observe the sample at time t and make a prediction
    prediction = olr.predict(X_california.iloc[t])
    # Store the prediction that we have made
    y_preds.append(prediction)
    # Calculate Loss and Update Weights
    olr.update_weights(y_california.iloc[t], prediction)
```

Here `olr` is an instance of the class `OnlineLinearRegression`. Its implementation can be found in the file `OnlineLR.py`.

After we have made a prediction for each of the samples we have received, we plot a graph of our loss over time:

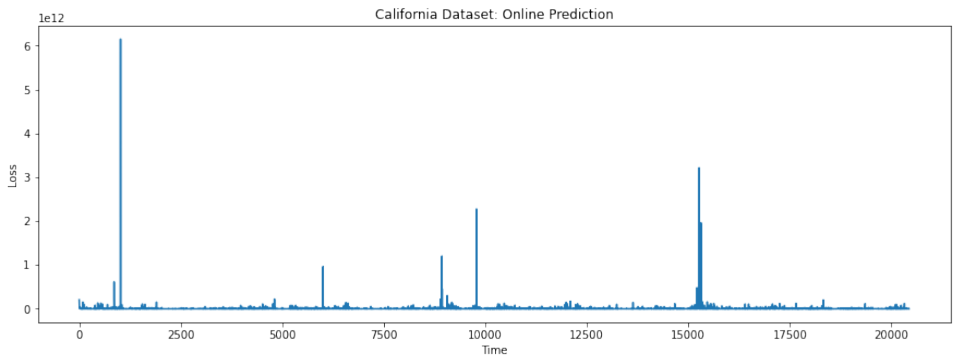


Figure 16: Loss over Time (Online Prediction)

Also, our Cumulative Loss (L) = `106900299526861.34`

4.3 Ames Housing Dataset

4.3.1 K-Nearest Neighbors

We apply the K-Nearest Neighbors model to the Ames Housing Dataset with a bunch of possible combinations of parameters and end up with the following result:

```
That took 7.0 minute(s).

Best value for n_neighbors: 5
Best distance function: 1
Weighted: True
Best Score: 0.8672412854684298
```

We find that the model with the best performance for this dataset is an **Weighted 5-Nearest Neighbors Model** using **Manhattan Distance**. Now, we take the model above and vary the number of neighbors to see how it affects the R^2 score:

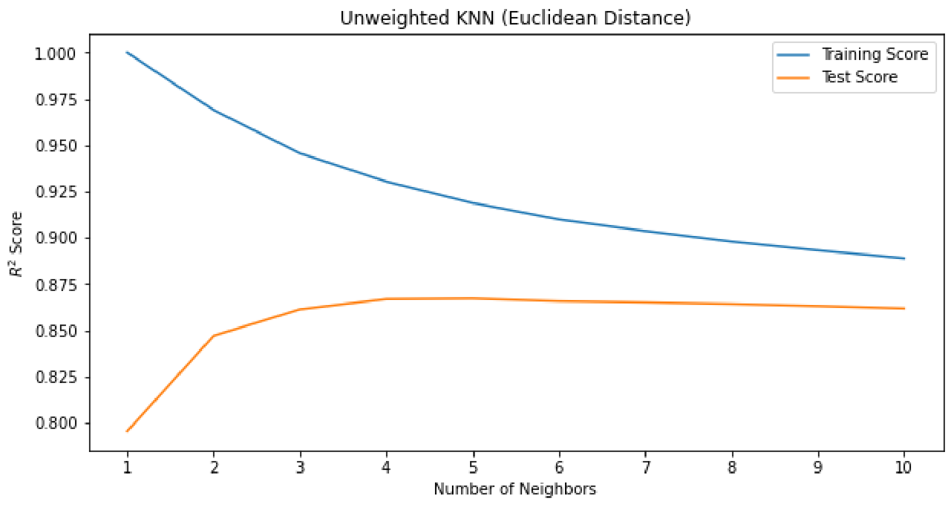


Figure 17: R^2 Scores vs Number of Neighbors

In the graph above, we observe that s the number of neighbors is increased beyond 5, there is no significant change in the test score. The model on the whole tends to underfit as the number of neighbors increases.

4.3.2 Random Forest

First, we calculate the maximum number of features to be considered at each node. As a general rule of thumb, it is calculated as:

```
p = int(np.ceil(X_train.shape[1]/3))
```

where `X_train.shape[1]` is the number of features in the dataset.

Then we apply the Random Forest Model to the Ames Housing Dataset with a range of possible combinations of its parameters and search for the model with the best performance. To avoid complexity costs, we set `n_estimators` = 100, as we've seen earlier that the number of trees does not drastically affect the performance. We end up with the following result:

```
That took 77.0 minute(s).

Best value for min_samples_split: 5
Best value for max_depth: 20
Best Score: 0.9138114734441379
```

Now, we vary the number of minimum samples required in a node for splitting to see how it affects the score:

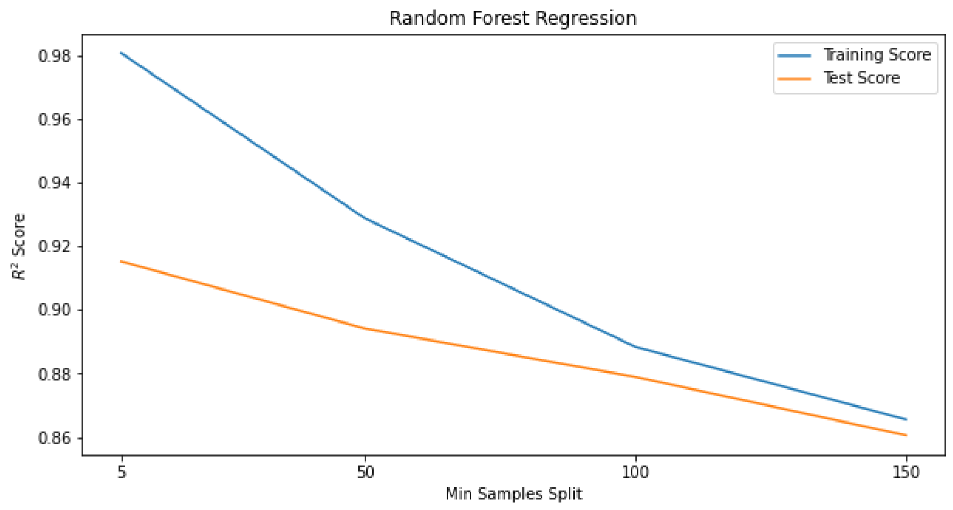


Figure 18: R^2 Scores vs Minimum Samples Per Split

Here, we observe that as `min_samples_split` increases the model starts to underfit due to the fact that the minimum number of samples required for splitting a node is so high that there are no significant splits observed. Therefore, we must not choose very high values for `min_samples_split` [19].

Now, we take the model above and vary the maximum depth of the tree to see how it affects the score:

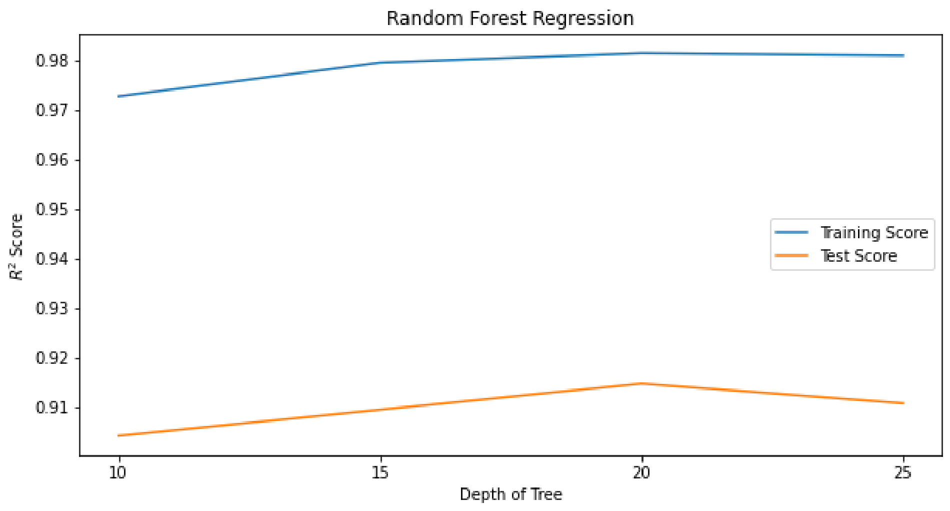


Figure 19: R^2 Scores vs Depth of Tree

We observe that the test scores increase as the depth of the tree increases up to a certain extent. After that, we see that the test score starts decreasing. There it's not necessary for us to set a very high depth for the trees [19].

4.3.3 Ridge

We fit our training data to a Ridge Regression Model with different values of the regularization parameter `alpha` and get the following result:

```
Best value of alpha: 10
Best Score: 0.8819893181169245
```

Below is a graph that shows how the scores are affected with respect to the regularization parameter alpha:

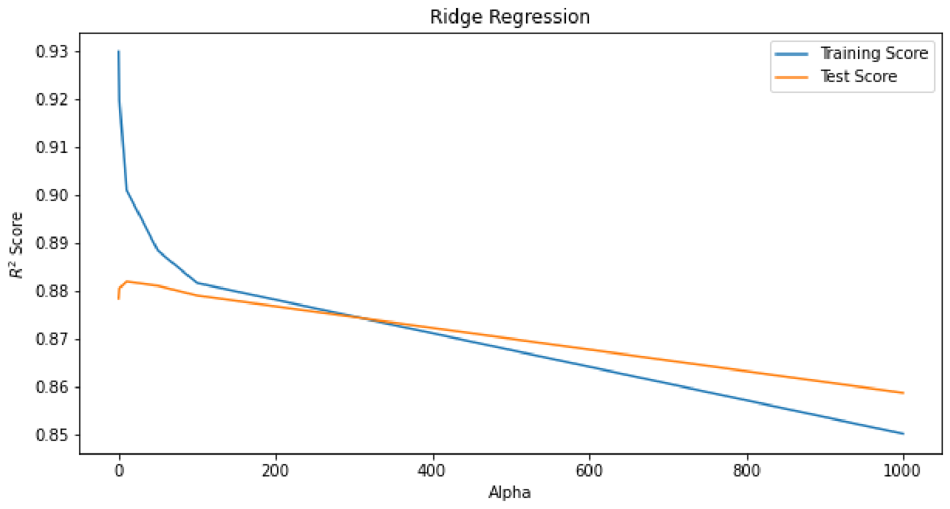


Figure 20: R^2 Scores vs Alpha

We observe that as the value of `alpha` is increased, the model starts to severely underfit which leads to poor training and test scores. After a certain point, interestingly, the test score becomes higher than the training score. Therefore, we must choose a small value for the regularization parameter `alpha`.

4.3.4 Kernelized Ridge

We fit our training data to the Kernelized Ridge Regression Model using three different kernels. The R^2 scores achieved are as follows:

- Linear Kernel: `0.8824086152840763`
- Polynomial Kernel: We try a bunch of possible combinations for its parameters and end up with the following:

```
Best Degree: 2
Best Coef0: 1
Best Score: 0.8986135374542726
```

- Radial Basis Kernel: `0.9028491386883942`

4.3.5 Comparing Models

Let us plot the scores achieved by our various models on the Ames Housing Dataset:

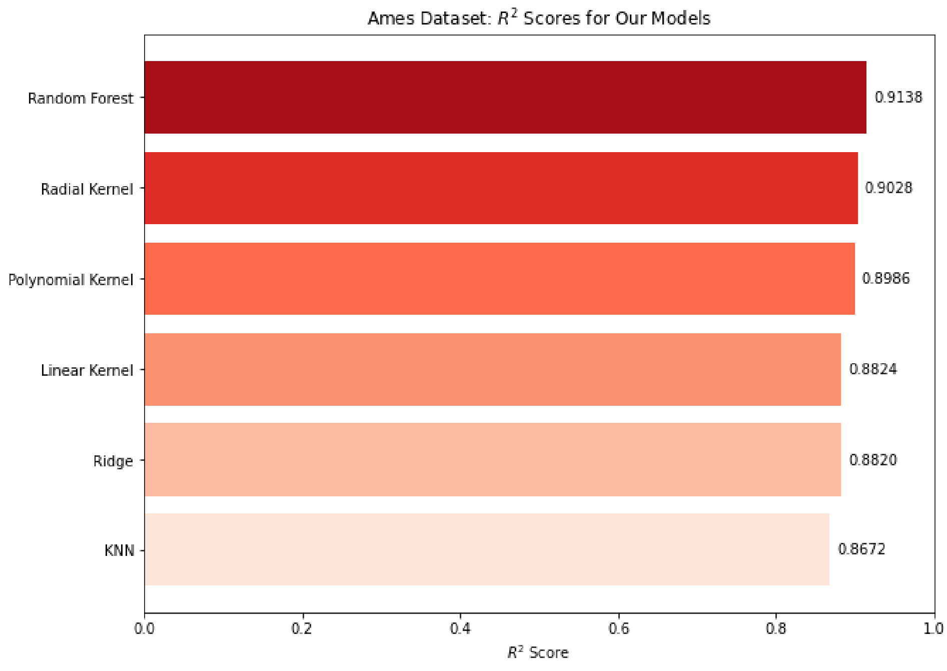


Figure 21: Comparing our Models (R^2 Scores)

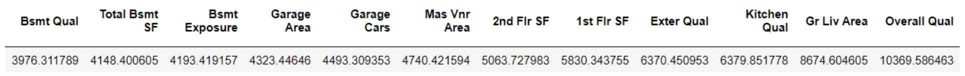
For the Ames Housing Dataset as well, **Random Forest Regression** is the most powerful model in terms of performance with a R^2 score ≈ 0.91. However, since this is a large dataset, it takes a lot of time to fit the model. And again, the model is complex but not interpretable.

Kernelized Ridge Regression using the **Polynomial Kernel** and **Radial Kernel** with a test R^2 score ≈ 0.90 perform very well too but suffer from being not interpretable. These models are much quicker than the `Random Forest` model.

The **K-Nearest Neighbors** model performs well with an R^2 Score ≈ 0.86. This method is easy to understand but very time-consuming due to multiple distance calculations [16].

Kernelized Ridge with the Linear Kernel performs well with a R^2 score ≈ 0.88.

The **Ridge Regression** model does a good job as well on this dataset and it is interpretable. By looking at the weights of the features, we can see which features have a significant effect on the price.



Bsmt Qual	Total Bsmt SF	Bsmt Exposure	Garage Area	Garage Cars	Mas Vnr Area	2nd Flr SF	1st Flr SF	Exter Qual	Kitchen Qual	Gr Liv Area	Overall Qual
3976.311789	4148.400605	4193.419157	4323.44646	4493.309353	4740.421594	5063.727983	5830.343755	6370.450953	6379.851778	8674.604605	10369.586463

Not surprisingly, we observe that the overall quality of the house `Overall Qual` and the size of the living area `Gr Living Area` have the most positive effect on the price `SalePrice`.

4.3.6 Online Prediction

In order to simulate an online scenario, we assume that the 2930 samples of the Ames Housing Dataset are available to us one at a time. We run a `for` loop from `t = 1,...,2930` where `t` represents an instance of time (which could be hours, days, weeks, etc.)

Then, we make a prediction for each sample that we observe based on the samples that we have observed so far. After every prediction, we also observe the true label, calculate our loss and update the weights.

```
for t in range(X_ames.shape[0]):
    # Observe the sample at time t and make a prediction
    prediction = olr.predict(X_ames.iloc[t])
    # Store the prediction that we have made
    y_preds.append(prediction)
    # Calculate Loss and Update Weights
    olr.update_weights(y_ames.iloc[t], prediction)
```

Here `olr` is an instance of the class `OnlineLinearRegression`. Its implementation can be found in the file `OnlineLR.py`.

After we have made a prediction for each of the samples we have received, we plot a graph of our loss over time:

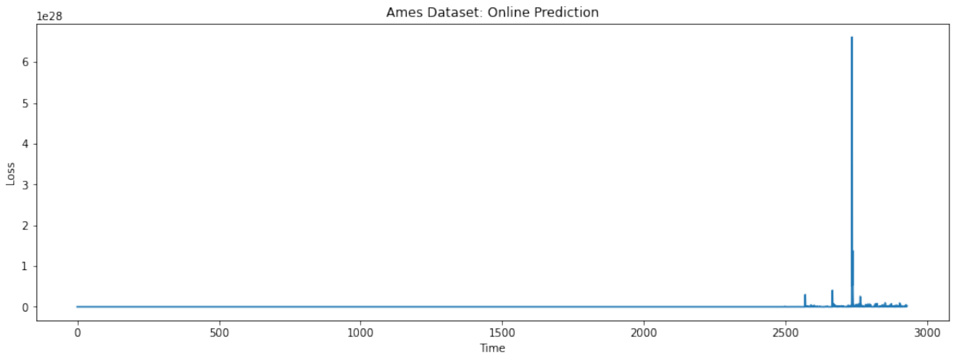


Figure 22: Loss over Time (Online Prediction)

Also, our Cumulative Loss (L) = `1.3427072367480432e+29`

5 Future Work and Conclusion

We observed that our models worked pretty well on the three datasets that we have chosen. The Random Forest Regression model achieved high predictive accuracy scores on all the datasets. This makes it a great choice if making the correct prediction is our goal. The models based on Ridge Regression were useful in observing how the individual features influence the price of the house. Finally, the K-Nearest Neighbors model provided us with a good and simple start to analysing the datasets.

In terms of future work on this project, we could explore the following possibilities:

5.1 Implementing More Models

We could implement more regressing models such as Lasso Regression, Support Vector Regression, and Neural Network Regression and then analyse how well these models perform on the three datasets.

5.2 Using other Datasets

We could apply the models that we have implemented to more housing data, and measure their performance on those datasets.

5.3 Prediction with Expert Advice

For the online mode of prediction, we could try to implement Prediction with Expert Advice where our model has access to the predictions made by a number of experts before our model makes its prediction. Our goal is to perform as well as or close to the best expert. [5]

6 Self-Assessment

Now, I will be discussing my experience throughout the course of this project. I learned many things and had to tackle a few problems too, but sticking to a proper plan along with regular meetings with my supervisor helped my steady progress.

I began my project by reading online articles on popular machine-learning models for regression problems. Then after a discussion with my supervisor, I decided to implement a K-Nearest Neighbors Regression model to kick-start my project. This would be followed by Random Forest Regression, Ridge Regression, and Kernelized Ridge Regression models.

I used the book 'The Elements of Statistical Learning' by T. Hastie, R, Tibshirani, and J. Friedman [11] to develop the mathematical knowledge required to write this report as well as implement the algorithms in Python.

After implementing and testing the models on simple data, I moved on to collect and preprocess the datasets that I would be using for this project namely, the Boston Housing Dataset, the California Housing Dataset, and the Ames Housing Dataset. The Boston and California Datasets were very straightforward to preprocess. However, preprocessing the Ames Housing Dataset with multiple missing values and categorical variables was a tedious task.

Applying our models to the California Housing Dataset –as discussed later in the Professional Issues section too– was a draining task for my CPU due to its sheer size. As a result slow models like the K-Nearest Neighbors Regression and the Random Forest Regression took a very long time to fit.

The next challenging task was to implement prediction in the online mode. I struggled a bit initially to find proper resources in order to approach this task but my supervisor referred to me some online papers which were very helpful to help me finish tackle this problem.

I decided to choose `Python` as the programming language to implement my algorithms as I am familiar with it and have used it across most of my modules. Its rich open-source libraries for machine learning like `numpy`, `pandas`, and `matplotlib` are an essential part of my project. I have used `Jupyter Notebook` as the editor for the analysis tasks. It makes showcasing my work easier as both the code and the results can be seen simultaneously.

In order to write this report, I have used `LaTeX`. The editor I used was `Overleaf` which is cloud based and easy to use. It helps us to see the code and its output simultaneously which makes it a great tool for writing reports.

I was able to successfully complete this project in time due to a well laid out plan, setting weekly goals and timely meetings with my supervisor.

References

[1] John Aldrich. Correlations Genuine and Spurious in Pearson and Yule. *Statistical Science*, 10(4):364 – 376, 1995.

[2] Rami Ali. Predictive Modeling: Types, Benefits, and Algorithms. `https://www.netsuite.com/portal/resource/articles/financial-management/predictive-modeling.shtml`.

[3] Jason Brownlee. 4 Distance Measures for Machine Learning. `https://machinelearningmastery.com/distance-measures-for-machine-learning/`.

[4] Jason Brownlee. Difference Between Classification and Regression in Machine Learning. `https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/`.

[5] Nicolò Cesa-Bianchi and Gábor Lugosi. Section 1. In *Prediction, Learning, and Games*, 2006.

[6] Yun Kuen Chueng. CS5100J Data Analysis Week 6 - Ensemble Methods. `https://moodle2122.royalholloway.ac.uk/mod/resource/view.php?id=733286`.

[7] D. Cohen, T. Lee, and D. Sklar. *Precalculus: A Problems-Oriented Approach*. Number p. 2 in Precalculus: A Problems-oriented Approach. Cengage Learning, 2004.

[8] Nicolo Colombo. CS5100J Data Analysis Week 3 - Algorithms 1. `https://moodle2122.royalholloway.ac.uk/mod/resource/view.php?id=716969`.

[9] Nathan Green. Correlation is Not Causation. `https://www.theguardian.com/science/blog/2012/jan/06/correlation-causation`.

[10] David Harrison and Daniel L. Rubenfield. Hedonic housing prices and the demand for clean air. `https://www.researchgate.net/publication/4974606_Hedonic_housing_prices_and_the_demand_for_clean_air`.

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, second edition, 2009.

[12] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R.* Springer, New York, first edition, 2013.

[13] Yuri Kalnishkan. Kernel Methods. http://onlineprediction.net/index.html?n=Main.KernelMethods#toc5.

[14] Herman Kamper. Regression Trees. https://github.com/kamperh/data414/blob/main/slides/dt_2_regression_trees-crop.pdf.

[15] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* The MIT Press, Cambridge, Massachusetts, first edition, 2012.

[16] Andreas C. Müller and Sarah Guido. *Introduction to Machine Learning with Python.* O' Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, 2016.

[17] University of Toronto. Boston Housing Dataset. https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html.

[18] Rob Schapire and Maryam Bahrani. Cos 511: Theoretical machine learning. https://www.cs.princeton.edu/courses/archive/spring18/cos511/scribe_notes/0411.pdf.

[19] Sharoon Saxena. A Beginner's Guide to Random Forest Hyperparameter Tuning. https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/.

[20] scikit-learn user guide. Section 3.3.4.1. R^2 score. https://scikit-learn.org/stable/modules/model_evaluation.html#r2-score-the-coefficient-of-determination.

[21] Edward R Tufte. *The Cognitive Style of PowerPoint: Pitching Out Corrupts Within.* Graphics Press, Cheshire, Connecticut, second edition, 2006.

[22] Vladimir Vovk. CS5920 Machine Learning Week 7 - Kernel Methods. https://moodle2122.royalholloway.ac.uk/mod/resource/view.php?id=584362.

A Professional Issues

In this section, I will be stating some issues that I faced while developing the project as well as some that have been of concern to me.

A.1 Ethical Problem

The Boston Housing Dataset used in this project has an ethical problem. The authors of this dataset have engineered a non-invertible variable 'B', assuming that racial segregation had a positive effect on house prices. [10] This dataset reflects the scenario of the past and therefore we must be mindful about this before jumping to any conclusions.

A.2 Long Runtimes

K-Nearest Neighbors and Random Forest are slow models and are very time consuming when working with large datasets. In this project, the California Housing Dataset is very large and is prone to long runtimes when applying a model to it. For example, when finding the best KNN model for the California dataset, we record the following run time:

```
That took 137.0 minute(s).
```

And similarly for the Random Forest Model:

```
That took 224.0 minute(s).
```

A.3 Correlation Does Not Imply Causation

"Correlation does not imply causation" refers to the inability to legitimately deduce a cause-and-effect relationship between two events or variables solely on the basis of an observed association or correlation between them. [21] To describe this in simple terms, just because two things correlate does not necessarily mean that one causes the other. [9]

The idea that "correlation implies causation" is an example of a questionable-cause logical fallacy, in which two events occurring together are taken to have established a cause-and-effect relationship.[1]

A.4 Lack of Resources

Lack of relevant and comprehensible resources for the online mode of prediction part of the project was a problem that I faced. Most of the articles and papers I did find were a bit too obscure for my level of understanding.

B How To Use My Project

All the code used in the project can be found in the submitted `MyProject.zip` folder.

The code for implementing the regression models can be found in:

- `KNN.py` for the K-Nearest Neighbors Regression Model
- `RFR.py` for the Random Forest Regression Model
- `RR.py` for the Ridge Regression Model
- `KernelizedRR.py` for the Kernelized Ridge Regression Model
- `OnlineLR.py` for the Online Linear Regression Model

The python notebooks containing the analysis for the three datasets can found under the names:

- `Boston Housing Dataset - Analysis.ipnyb`
- `California Housing Dataset - Analysis.ipnyb`
- `Ames Housing Dataset - Analysis.ipnyb`

To view the analysis, just open the files and run the cells in a serial manner. The previous outputs should still be visible for the cells.

Advisory: *Some cells with very long runtimes have the runtime displayed in its output. So please consider that before running the cell again.*

The code has the following dependencies: `NumPy`, `pandas`, `matplotlib`, `seaborn`, and `sklearn` which can be installed as follows:

- For `NumPy`: `pip install --user numpy`
- For `pandas`: `pip install --user pandas`
- For `matplotlib`: `pip install --user matplotlib`
- For `seaborn`: `pip install --user seaborn`
- For `sklearn`: `pip install --user scikit-learn`

Private Github Repository

All the code and plots for this project should be provided along with this report. Alternatively, they can also be found at: `https://github.com/rohanprad/housing-prices`. To get access to this repository, please send an email to:

`rohanprad09@gmail.com`

Links in this Report

In case you are unable to visit a link in this report by clicking on it, please copy the entire URL and paste it in the browser's search bar.

Python Version

This project was developed using `Python 3.9.7`. Please ensure that you are using `Python 3` or above to successfully execute all the code.