

**School of InfoComm Technology**

**Machine Learning**

Diploma in Data Science (DS)

Diploma in Information Technology (IT)

October 2023 Semester

**INDIVIDUAL ASSIGNMENT 2**

(40% of Machine Learning Module)

# Deadline for Submission:

**Presentation: 28th Jan 2024 (Sunday), 2359 Hours**

**Report: 8th Feb 2024 (Thursday), 2359 Hours**

|  |  |  |
| --- | --- | --- |
| Student Name | : | Markell Wong |
| Student Number | : | S10242300 |

**Penalty for late submission:**

10% of the marks will be deducted every day after the deadline.

**NO** submission will be accepted after 15th Feb 2024, 23:59.

# Introduction

In classification problems, our goal is to categorize or label input data into predefined classes or categories. Some examples of classification tasks are email spam detection, image recognitions etc. For these problems, we can use models such as Logistic Regression, Decision Trees and many other advanced machine learning models.

In regression problems, our goal is to predict continuous numerical values based on input features. Some examples of regression tasks are forecasting stock prices, predicting house prices etc. For these problems, we can use models such as Linear Regression, Support Vector Regression and more!

When we build up models, we typically want to follow the universal machine learning workflow. We want to first state concisely our problem, for example, if it is binary or multiclass classification. Then we choose a measure of success given the nature of our problem. This can be like precision for imbalanced datasets etc. Next we decide on a evaluation protocol, will we be using cross validation? Or it a hold out test set enough?

We then prepare the data, familiarizing ourself with its characteristics before developing a baseline model and a model that does better than the baseline. Next we develop a model that overfits.

The universal tension in machine learning is between optimization and generalization; the ideal model is one that stands right at the border between under-fitting and over-fitting; between under-capacity and over-capacity. To figure out where this border lies, first you must cross it.

Then, repeatedly modify your model, train it, evaluate on your validation data (not the test data, at this point), modify it again, and repeat, until the model is as good as it can get.

For this project, I tried to follow the workflow as closely as I could.

# Parameter Tuning Methods

Firstly, I will go through the different methods I used to tune the hyperparameter values.

**Manual Tuning**

Manual tuning is where I tuned each hyperparameter one by one, plotting the test and train curves, to visualize where convergence or deviation occurs, then selecting the best value. With manual tuning, I had more control over the tuning process, but it led to a longer and less efficient process.

**GridSearchCV**

GridSearch essentially tries every combination of hyperparameter values defined in the feature space. This allows it to find the absolute best values but it is extremely computationally expensive and inefficient especially when the feature space is huge or many hyperparameters are being tested.

**Bayesian Optimization**

Compared to the two other methods which are quite inefficient and slow, Bayesian optimization chooses different current values based on the performance of past values. Essentially, it can learn from its past iterations, making it much more efficient.

A diagram of a process

Description automatically generated

Process of Bayesian Optimization

This is the overall process. So, we first start with a random set of hyperparameter values, we then evaluate the values with the model and based on the evaluation results, we start training a Gaussian process regression. We then do a calculation of the acquisition function and use the function to identify the next set of hyperparameter values. We try to minimize the function to decide new values to be evaluated.

A graph with purple dots

Description automatically generated

Let’s say we are doing Bayesian optimization for a regression model based on the mean-squared error metric. Our objective function, f(x), would be mean-squared error and we aim to minimize it. First, we put in some values and get the predictions of the model. Input1 is our hyperparameter value.

We start with an initial set of hyperparameter values (x) and their corresponding evaluations using the objective function f(x).

A yellow line with purple dots

Description automatically generated

We then train the Gaussian process regressor. Here, we train multiple regression functions with different kernels and tails, the blue line indicates the mean of all the regression functions while the yellow range indicates the uncertainty of the model and is the standard deviation of the model’s predictions. [This is further explained in Gaussian Process Classifier]

As we can see, the regions where there are data samples have no uncertainty. The further the sample points are away from each other, the uncertainty range increases

A screenshot of a graph

Description automatically generated

Next, we find the activation function. An activation function is a mathematical function describing the gain or potential optimization volume. In the figure above, we use a common activation function called lower confidence bound. Our goal is to find the minimal point of the activation function. The next sample value of choice is where the acquisition function is minimized. From our figure, as we see by the lowest point of our activation function, we sample somewhere between 5 and 6.

The acquisition function is used to select the next hyperparameter value to evaluate, balancing exploration and exploitation. The selected configuration is then evaluated using the objective function and the uncertainty range is updated. We keep refining the uncertainty range and selecting new hyperparameter values.

A screenshot of a graph

Description automatically generated

Then, we start each iteration. For every iteration, we add more sample points (pink dot) from the previous iteration and retrain our model and as we can see in the figure above, the sample value has now completely shifted to 10 due to the high uncertainty ranges from the previous iteration and we find a new minimum point. We can see also see that the uncertainty (yellow range) of the Gaussian regressor changes after every iteration. The acquisition function is used to select the next point to evaluate based on the current uncertainties.

This process is then repeated iteratively until we reach the defined number of iterations.

In the end, we get an optimized set of parameters that minimizes our MSE.

Pros

Bayesian optimization explores the feature space intelligently, reducing the need for many evaluations. Its use of probabilistic models like Gaussian process allows it to better explore the feature space, searching for regions where improvements are likely. This makes it effective, and less time-consuming.

Cons

In Bayesian optimization, the Gaussian process relies on the assumption that nearby points in the feature space have similar metric values. This works well when the metric is continuous and differentiable. However, if the metric is not continuous or smooth, optimization is unable to model the underlying behaviour. Non-smooth metrics are when the metric suddenly changes even with close hyperparameter values.

Additionally, Bayesian optimization is unable to work with categorical hyperparameters as it uses a continuous landscape.

Hence, I will use GridSearch and Manual tuning in conjunction with Bayesian optimization.

Images Credit: <https://www.youtube.com/watch?v=M-NTkxfd7-8&t=508s>

# HR Analytics

## Problem understanding

Problem Statement: HR would like to use machine learning to help make identifying employees who are most likely to be promoted easier.

The target variable is the feature “is\_promoted”, it is a categorical nominal variable type ( 0 and 1). This means that this is a binary classification problem.

## Summary of data cleaning and transformation process

From Assignment 1:

For missing data, I used random sampling to impute null values. Three features, ‘length\_of\_service’, ‘no\_of\_trainings’ and ‘age’ had outliers hence I applied windsorization to replace the outliers. Next, I ensured the class frequencies were balanced by down-sampling the data.

Next, as we are creating models for this assignment, I train-test split the data before any transformation or scaling to prevent any data leakage on the test dataset.

Data leakage is when information from outside the training dataset is used to create the model. This additional information can allow the model to learn or know something that it otherwise would not know and lead to overestimation of model performance and poor generalization to new data

For distribution transformation, I did log transform on ‘age’, ‘length\_of\_service’ and ‘avg\_training\_score’. I then grouped rare categories for ‘region’ feature.

Did encoding for categorical variables, ordinal encoding for ‘education’, count encoding for ‘department’, ‘gender’ and ‘recruitment\_channel’ and mean encoding for ‘region’. Lastly, did min max scalling and removed length\_of\_service and department.

**Approach**

Since we balanced the classes in the previous assignment, we can safely use accuracy as a metric.

As our dataset is not too small, we do not need to do K-fold validation and can instead evaluate the models with a hold-out test set. For each model, we will try to follow the universal machine learning workflow of scaling the model to overfitting then regularizing and tuning hyperparameters.

## Building, evaluating and improving the model(s)

**Decision Trees**

**A diagram of wind direction

Description automatically generated**

[**https://www.ibm.com/topics/decision-trees**](https://www.ibm.com/topics/decision-trees)

The first model I built was a simple decision tree classifier. They are non-parametric (do not make any assumptions about the data) supervised learning models. Decision trees use splits done in a top-down, recursive manner until the majority or all classes are classified. Optimal splits are found using greedy search where it selects the best option at the moment.

A screenshot of a computer screen

Description automatically generated

A graph with blue bars

Description automatically generated

Based on our correlation and mutual information analysis in assignment 1, our data lacked strong linear and non-linear relationships with the target variable. Decision trees are able to capture complex relationships beyond linear and non-linear due to its non-parametric nature.

A screenshot of a computer code

Description automatically generated

I first build a very simple model with a max\_depth of 3.

A diagram of a company structure

Description automatically generated

As seen above, some of the splits had a very high number of samples for both classes (eg: [993, 1849]) and gini impurity values for some of the splits were very high at 0.454. For reference, the maximum for binary classification is 0.5. This shows that our tree is unable to split definitively and that the splits did not provide enough information to differentiate the classes.

When looking at train and test accuracy, training accuracy: 0.7416, testing accuracy: 0.7372. While the model is not performing too badly, we can tune some hyperparameters to improve performance.

Max\_Depth

I first tune the max\_depth of the tree. Max\_depth is a crucial hyperparameter that refers to the length of the longest path from the root node to leaf node. It influences the model’s complexity and can be adjusted to improve the model’s generalization of new data.

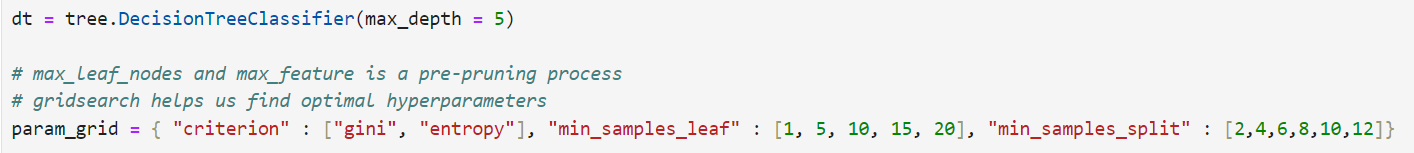
A graph of a performance

Description automatically generated with medium confidence

Here, I manually tune the max\_depth. At around max\_depth = 5, the testing accuracy starts to decrease while training accuracy continues increasing. This shows that overfitting starts after max depth 5. As max\_depth 5 has the highest test accuracy before the model starts to overfit, we will increase the max\_depth for 3 to 5.

Tuning max\_depth improve model performance by quite a lot, train accuracy: 0.7671, test accuracy: 0.7572.

Next, I tuned criterion, min\_samples\_leaf and min\_samples\_split using GridSearch. These hyperparameters are commonly tuned to achieve better performance, control overfitting and to control the complexity of the tree. By tuning these hyperparameters, I looked to improve model performance on test data. There were other hyperparameters like max\_leaf\_nodes and max\_features, however I did not define them as they look to prevent overfitting.



Criterion:

Used to determine the loss function used for impurity in the decision tree nodes. The choice of criteria directly affects how our model makes decisions at each node, with different criterias resulting in different tree structures. By experimenting with criterion, we try to find the best loss function for our splits to improve model performance.

Feature space: Gini and Entropy

Min\_samples\_leaf:

Sets the minimum number of samples required in a leaf node. It helps prevent the creation of nodes that have far too few too samples, which might capture noise in the data. Typically, increasing this value can lead to more robust models and better generalization. The default value is 1, hence I will trying higher values to see if it improves model generalization.

Feature space: [1, 5, 10, 15, 20]

Min\_samples\_split:

Sets the minimum number of samples a node must have to split. If a node has fewer samples than the minimum, it will not split. By tuning this, I hoped to better control the quality of the splits, improving model generalization. Here, I tried higher minimum values, the default is 2.

Feature space: [2, 4, 6, 8, 10, 12]

GridSearch results:

{'criterion': 'gini', 'min\_samples\_leaf': 20, 'min\_samples\_split': 2}

When evaluating with k-fold validation, the train\_accuracy is 0.7613 and test accuracy is 0.7601. We can see a slight improvement from the previous iteration.

Next, I did polynomial expansion to try to capture any non-linear relationships in the data. By doing this, I looked to improve model generalization.

A screenshot of a computer

Description automatically generated

Based on the feature importance, I expanded on the top 3 features at a degree of 3. However, the model performance deteriorated with polynomial features, train accuracy: 0.7556, test accuracy: 0.7443.

**Random Forest Classifier**

A diagram of a sample

Description automatically generated

<https://medium.com/@roiyeho/random-forests-98892261dc49>

Random Forest Classifier is a bagging ensemble method that builds multiple decision trees. It uses bootstrap and out-of-bag sampling as well as feature bagging to introduce diversity among individual decision trees, creating an uncorrelated forest of decision trees. After training, individual trees are then brought together for a majority vote (hard voting), where cross validation is done for finalization.

I decided to build random forest as it is able to build multiple decision trees at once, making it more robust and generalize better. It is also able to introduce feature randomness, further improving generalization. By using random forest, I looked to improve on decision tree’s performance.

A screenshot of a computer program

Description automatically generated

Base model performance: train accuracy: 0.9925, test accuracy: 0.7222

As we can see from the variance between train and test, our model is heavily overfitting at a 0.2703 accuracy difference. We will need to introduce some regularization to reduce the variance.

To counter overfitting, I tuned regularization and model complexity hyperparameters, namely: max\_depth, n\_estimators, min\_samples\_split, min\_samples\_leaf and max\_leaf\_nodes.

A screen shot of a computer code

Description automatically generated

Max\_depth: By defining the max\_depth, we can set a limit of the maximum depth of the tree. The default value is none, allowing the tree to expand without control. Hence I defined a range of (5, 32) to prevent and control the trees from becoming too deep.

N\_estimators:

Controls the number of decision tree models in the forest. A higher number of trees can lead to higher model complexity, resulting in overfitting. Our base model’s value is 20 hence I defined a smaller range of values (10, 20). By using a feature space with smaller values, I looked to decrease the model complexity.

Min\_samples\_split: Default is 2, by defining a feature space with larger values (2, 20), I looked to further limit decision tree growth, lowering model complexity.

Min\_samples\_leaf: Default is 1, by defining a feature space with larger values (2,20), I looked to make the model more robust.

Max\_leaf\_nodes:

Restricts the number of leaf nodes in the tree. As the default value is none, the tree is able to grow without control, hence I defined a range (5,32) to control the maximum number of leaf nodes, helping control the complexity of the model.

Bayesian Optimization Output:

The decimal value outputs of Bayesian optimization need to be round down as the hyperparameters only take integer values.

Best Hyperparameters: {'max\_depth': 18, 'max\_leaf\_nodes': 30, 'min\_samples\_leaf': 17, 'min\_samples\_split': 17, 'n\_estimators': 19}

The optimization function decreased n\_estimators and increased min\_samples\_split and min\_samples\_leaf from its default values.

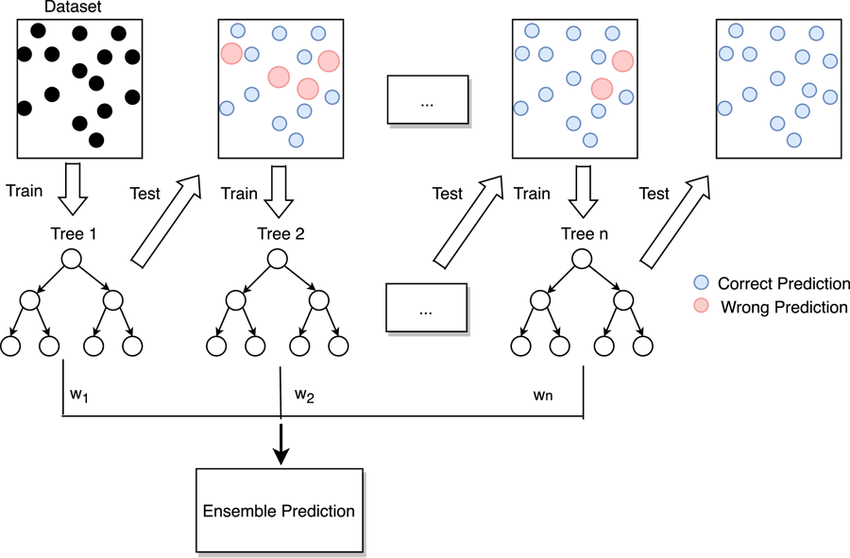
The training accuracy is now 0.7651 and the test accuracy is 0.7544. As we can see, variance between train and test scores have been reduced. The random forest model now has better performance than the decision tree.

Polynomial expansion

Next, I experimented to see if I could further improve the generalization of the model using polynomial expansion. I defined the top 3 features based on feature importance, 'KPIs\_met >80%', 'avg\_training\_score', and 'previous\_year\_rating' and expanded them to a degree of 3.

After polynomial expansion, my train accuracy was 0.797 and test accuracy was 0.7611. While test scores were now slightly better, the variance was increased hence I decided not to use polynomial expansion.

**Gradient Boosting Classifier**

****

<https://www.researchgate.net/figure/Flow-diagram-of-gradient-boosting-machine-learning-method-The-ensemble-classifiers_fig1_351542039>

The gradient boosting classifer combines multiple weak learners (typically decision trees) as a progression of the previous model. As seen in the figure above, boosting is where multiple learners are built sequentially hence errors committed by the previous model is learnt by the next model, improving generalization with every model. At the end, the model takes the predictions of each learner with weights assigned to minimize error and generates an output (soft voting). This process is repeated until the model is able to get the best accuracy.

I decided to use gradient boosting to see if I could improve on random forest’s model performance. In gradient boosting, better-performing models are prioritized and models are built upon each other, allowing continuous improvement. In random forest, hard voting is used hence even base models with low confidence, contribute the same to the final output. Additionally, gradient boosting runs multiple iterations.

A screenshot of a computer code

Description automatically generated

I started off with a relatively simple model, n\_estimators = 100, learning\_rate = 1.0 and max\_depth = 1. The train accuracy was 0.7319 and test accuracy was 0.7158. As we can see, the model underperforms compared to the other models tested so far. We will need to scale up the model till overfitting.

I first tuned tree-specific model complexity hyperparameters namely: max\_depth and n\_estimators

A close-up of red text

Description automatically generated

Our base gradient boosting classifier model had a max\_depth of 1 and n\_estimators of 100. As we want to increase model complexity, I defined the feature space of both hyperparameters to have larger values.

GridSearch output: max\_depth: 10, n\_estimators: 100

Testing the new values: train accuracy: 0.9957, test accuracy: 0.7161. As you can see, our model has a very large variance between train and test scores hence experiencing heavy overfitting after increasing the max\_depth from 1 to 10. We can now regularize the model.

Next, I tuned the boosting regularization parameters namely: learning\_rate and subsample using Bayesian optimization as they were continuous variables.

Learning rate can be considered the most important hyperparameter of gradient boosting. It controls the contribution of each weak learner by adjusting the shrinkage factor. Smaller learning rates decreases how much a weak learner can contribute to the output hence requiring more weak learners to achieve good performance. But, it also makes the model more robust and less prone to overfitting.

Our model currently has a learning rate of 1.0. I defined the feature space of (0.001, 1). By defining a feature space with smaller values, we look to reduce model overfitting.

Subsample controls the proportion of the data that is used to train each tree rather than the entire dataset. The subset is typically sampled without replacement. Here, we can reduce the subsample rate, meaning that a smaller fraction of the data is used to train each weak leaner which introduces diversity in the ensemble, leading to a more robust model that generalizes better to unseen data, reducing overfitting.

The default rate of subsample is 1.0. To reduce overfitting, I defined a feature space with smaller values (0.5, 1.0).

Bayesian output

Best Hyperparameters: {'learning\_rate': 0.0027152900679729124, 'subsample': 0.5479190369866818}

After reducing the learning rate and subsample values, we get a train accuracy of 0.8084 and a test accuracy of 0.7500. As we can see, the variance between train and test has been reduced greatly. Moreover, test accuracy has improved showing increase in generalisation ability.

Lastly, I tuned miscellaneous parameters such as criterion and loss. Here, I try to further improve model generalization selecting a more suitable splitting criteria and loss function for the model.

For criterion, I tried friedman\_mse and squared\_error. Between the two, friedman\_mse is the default and is typically preferred over squared\_error as it is designed to provide better sensitivity to errors made by earlier weak leaners and can lead to improved performance. However, we will still try both to see which is more suitable for our model

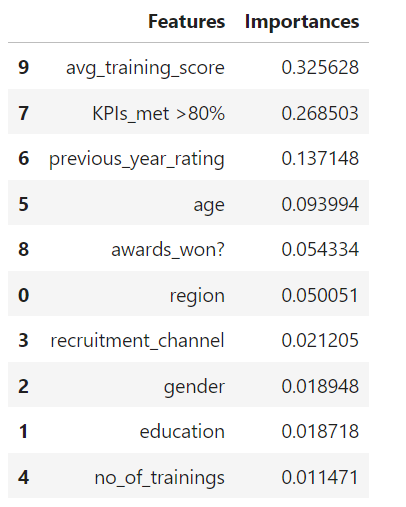
Loss function is essentially the metric we use the evaluate the differences between predicted and actual values. The ensemble aims to minimize the loss function. Here, I tried log\_loss and exponential. Log loss is the default function for GBC, exponential is used to emphasize misclassifications on subsequent models in the ensemble.

GridSearch output: {'criterion': 'squared\_error', 'loss': 'exponential'}

Now, train accuracy: 0.8067 and test accuracy: 0.7511. As we can see, the variance is slightly reduced and test accuracy is slightly increased after changing the criterion and loss.

Feature Selection

Next, I tried to select only the important features to see if I was able to further reduce the variance.

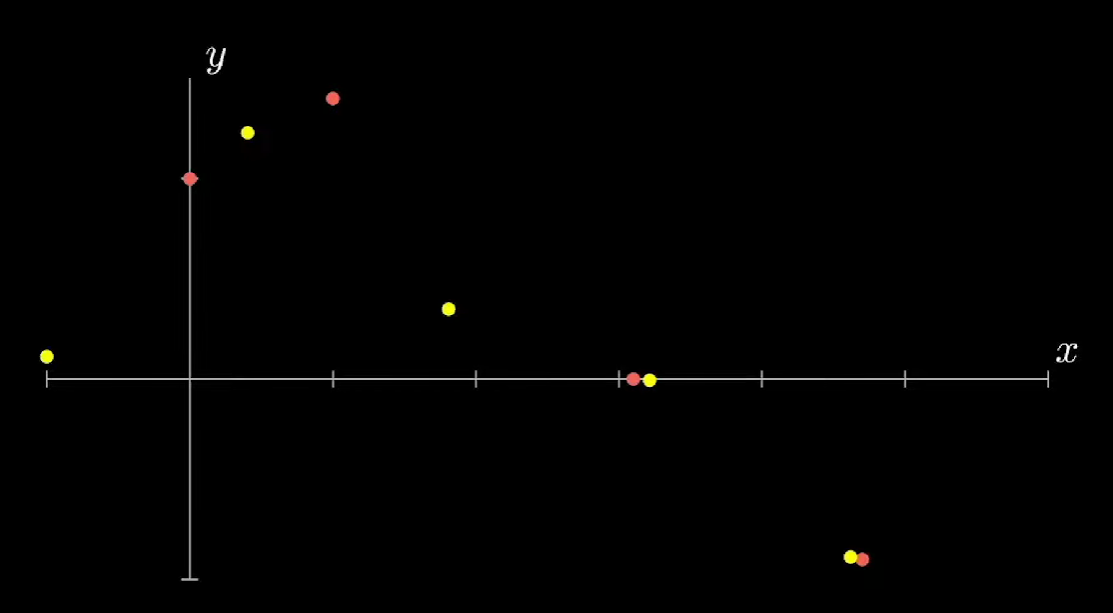


I only selected the features with importance > 0.05. After testing the model on the new dataset, my train accuracy is 0.8013 and test accuracy is 0.7493. While variance is reduced slightly, the slight deterioriation

I only selected the feature with importance > 0.05. After testing the model on the new dataset, my train accuracy is 0.8013 and test accuracy is 0.7493. While variance is reduced slightly, there was also a slight deterioration in test set (explain why you didn’t choose)

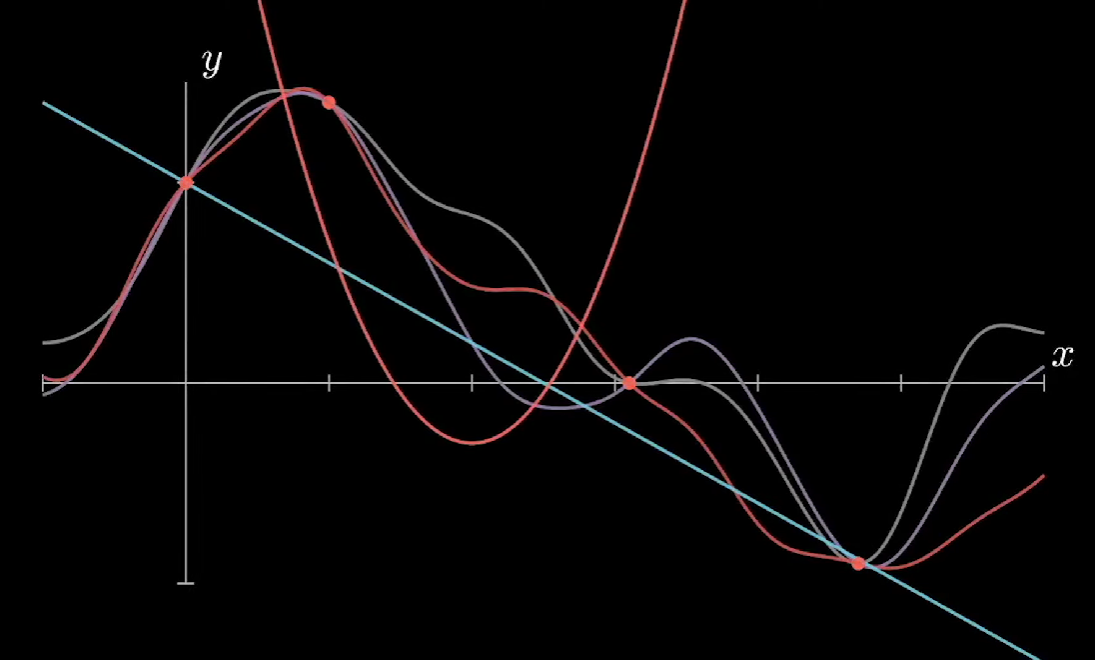
**Gaussian Process Classifier**

Gaussian Process is a non-parametric and probabilistic model that defines a distribution over functions.



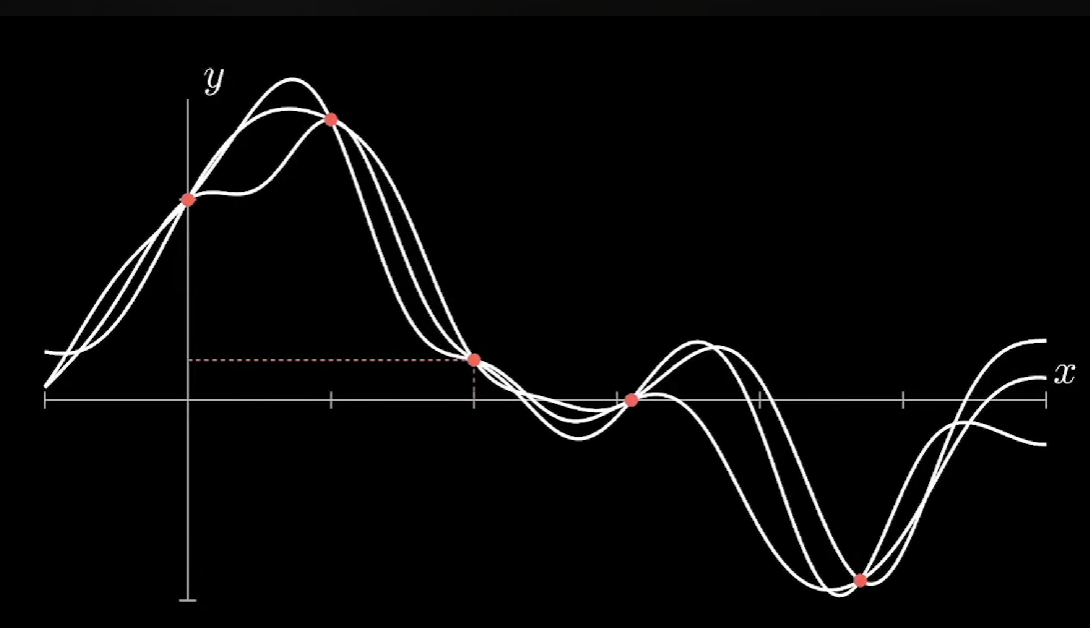
<https://www.youtube.com/watch?v=iDzaoEwd0N0>

Lets say we have the following dataset of regression targets, y indexed by independent variable x. Then, we try to predict a function that can predict every point.



To determine the functions that are able to predict every point, gaussian process considers all the functions and adds a probability to each of them.

A graph of a graph

Description automatically generated with medium confidence 

Now, we have a probability distribution of all the functions. When we add more data points, we want to update our function probability distribution accordingly. To do that, we calculate the probability of a function under the condition that new data is observed, this is the conditional probability.

Afterwards, we combine the Gaussian process and the conditional probability calculation to create an algorithm which calculates and adapts our function probability distribution iteratively to the observed values. In the end, we have to choose a single function based on the probability measure. One way to do it is the mean of all functions, however, we will also need to quantify uncertainty as we defined a probability measure.

A graph of a wave

Description automatically generated with medium confidence

Hence, in the end we will not only have the mean of all the functions (line at the centre of uncertainty range), but we will also have a range where we expect most of our functions to be.

The above process is done for regression but classification is no different. Instead of a continuous output variable y, we have discrete classes as our output. The Gaussian process provides a probabilistic prediction for the different classes for a given input, and the predicted output is the class with the highest probability.

I decided to use Gaussian processes as they have a completely different structure as compared to the decision-tree based models we used earlier. Here, I wanted to experiment if something different like gaussian process was more suitable for the data.

**Base Model**

A screenshot of a computer code

Description automatically generated

For the base model, I used a relatively simple model with radial basis function and length\_scale = 1. N\_restarts\_optimizer was not defined.

A screenshot of a computer code

Description automatically generated

The first model performs quite well with a 0.7447 test accuracy. As gaussian process does not have an explicit model complexity hyperparameter, we cannot scale the model up. Instead, we will try to improve generalization by using different kernels

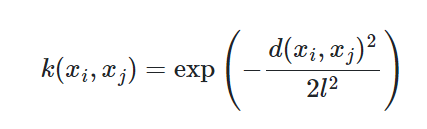
**Tuning**

First, I tried different kernel types to find the one most suitable for my model. The properties of Gaussian process that is used to fit the data is strongly controlled by a covariance function (kernel). Kernels define underlying assumptions about the relationship between data points such as assumptions that similar data points should have similar target values.

There are stationary and non-stationary kernels. Stationary kernels depend only on the distance between data points and not their absolute values while non-stationary kernels additionally depend on the exact values of the data points.

I tried 4 other kernels, namely DotProduct, Matern, RationalQuadratic and WhiteKernel.

Radial Basis Function (RBF): The one originally tested, it is a stationary kernel and is given by:



where d(xi, xj) is the Euclidean distance. It is characterized by its smoothness and ability to model functions that vary smoothly over the input space

DotProduct: A non-stationary kernel and is given by:

A black and white image of a mathematical equation

Description automatically generated

As it models linear relationships between x and y, it is particularly useful when underlying function is expected to be linear or close to linear

Matern: A stationary kernel and a generalization of RBF. It adds a “v” parameter to control the smoothness of the resulting function.

## RationalQuadratic: Another generalization of RBF that includes a scale mixture of RBF kernels. Is particularly useful when dealing with data that exhibit short-term and long-term variations.

## 

## Out of all the kernels, Matern (index 2) producted the best test performance at a train accuracy of 0.7628 and test accuracy of 0.7504.

## Length Scale

## Next, I tuned length scale. The length scale controls the characteristic length over which the function values in the Gaussian process are correlated. It influences the smoothness and flexibility of the Gaussian process model.

## A larger length scale allows the model to capture broader trends and variations in the data, making it more suited for modelling smooth functions. A smaller length scale allows the model to capture fine-grained details and rapid changes in the data, making it more flexible but potentially more prone to fitting noise.

## The default length scale is 1.0. As I did not have a clear idea of the model’s functions, I tried both higher and lower length scale values.

## 

## A screenshot of a computer Description automatically generated

## However, there seemed to be no model performance differences between the length scales. This could be because underlying patterns in the data are relatively smooth and the length scale was already within a reasonable range, hence adjusting it did not have a significant impact.

## N\_restarts\_optimizer

## n\_restarts\_optimizer parameter determines how many times the optimization process should be restarted with different initial hyperparameter values. Each restart begins the optimization process from a different starting point. More restarts can increase the likelihood of finding a global optimum, but it also incurs a computational cost.

## The default value is 0, hence only one run is performed, hence I increased the number of runs to 5.

## 

## However, there were no changes in model performance, this could mean that the model has already reached the global optimum.

## Feature Engineering

## Next, I looked at the feature importance of the model. As .feature\_importances cannot be done for Gaussian Processes, I used permutation importance to evaluate feature importance. Essentially, it permutates the values of each feature and observes the impact on model performance.

## 

## Here, I tried polynomial expansion to see if I could still improve model performance by expanding on the data’s non-linear relationships. I expanded on the top 3 most important features of the model, 'KPIs\_met >80%', 'avg\_training\_score' and 'previous\_year\_rating'.

## A screenshot of a computer code Description automatically generated

## The polynomial expansion improved model generalization, with a train accuracy of 0.7727 and a test accuracy of 0.7622.

## Summary

I will go through the problem, models created as well as hyperparameters tuned and feature engineering used for each model.

This problem is a binary classification problem as the target variable is a categorical nominal variable type (eg: 0 and 1)

I built 4 different types of models: Decision Trees, Random Forest, Gradient Boosting Classifier and Gaussian Process Classifier.

For Decision trees, I started with training accuracy of 0.7416 and 0.7372. I then increased max\_depth from 3 to 5 and increased min\_samples\_leaf from default 1 to 20. Final train accuracy: 0.7613, test accuracy: 0.7601

For Random Forest, I started with a heavily overfitted model with training accuracy 0.9925, test accuracy: 0.7222. I then defined the max\_depth = 18, defined max\_leaf\_nodes to 30 as both were orignally none. I also increased min\_samples\_leaf from default 1 to 17, increased min\_samples\_split from default 2 to 17 and reduced n\_estimators from 20 to 19.

Afterwards, my final model had a training accuracy of 0.7651 and test of 0.7544, hence variance was greatly reduced and model generalisation to new data was improved.

For Gradient Boosting Classifier, I started with a slightly underperforming model at train accuracy 0.7319 and test accuracy 0.7158. I then increased max\_depth from 1 to 10 to scale up the model. The model accuract was 0.9957 train and 0.7161 test.

I then regularized the model by reducing learning rate from 1.0 to 0.0027, reduced subsample from 1.0 to 0.5479. Accuracy after this was train: 0.8084 and test: 0.7500.

Lastly, I switched splitting criteria from Friedman\_mse to squared\_error and loss function from log\_loss to exponential. The final model accuracy was train: 0.8067 and 0.7511.

For Gaussian Process, I started with a train accuracy of 0.7551 and test accuracy of 0.7447. I then changed kernels from default radial basis function to Matern and introduce polynomial expansion for the top 3 most important features based on permutation importance. The final model accuracy was train 0.7727 and test 0.7622.

Chosen Model

For this problem, I chose to use Gaussian Process as it was able to best generalise to new, unseen data and had little variance between train and test scores.

# Airbnb

## Problem understanding and the approaches

## A property management firm would like to optimize its pricing strategy for short-term rentals by predicting optimal prices for their Airbnb listings.

## The target variable is the feature “price”, it is a continuous variable. This is a regression problem.

## Summary of data cleaning and transformation process

Features last\_review, reviews\_per\_month and name had missing values. For last\_review, I grouped the data and replaced the null values with "No review". For review\_per\_month, I replaced the null values with 0 and for education, I decided to drop the rows. Features price, minimum\_nights, number\_of\_reviews, reviews\_per\_month and calculated\_host\_listings\_count had outliers. I used windsorization to replace the outliers.

Afterwards, I removed ID as it was a unique identifier column, host\_name as it is better represented by host\_id, latitude and longitude as they are no longer meaningful after scaling and neighbourhood\_group as it only had one value. I then created more features using TFIDF to increase the number of features the model can learn from. Similarly to the HR dataset, I train-test split before any transformation to prevent data leakage.

Next, I did distribution transformation on reviews\_per\_month. I also encoded object values by using target encoding for neighbourhood and count encoding for features room\_type and last\_review.

I then scaled my dataset to be all of the same range using Min-max scaling. I also did correlation analysis where I dropped reviews\_per\_month due to high multicollinearity and poor correlation with the target variable.

**Approach:**

For this problem, we will use Root Mean Squared Error and R-Squared

R-Squared is a statistical measure that represents the percentage of variance in price that the independent variables explain, providing us with a measure of how well the model fits the observed data.

As our dataset is not too small, we do not need to do K-fold validation and can instead evaluate the models with a hold-out test set. For each model, we will try to follow the universal machine learning workflow of scaling the model to overfitting then regularizing and tuning hyperparameters.

## Build, evaluating and improving the model(s)

## Multi-layer perceptron

## Introduction to how an Multilayer Perceptron works but without complicated math | by SangGyu An | CodeX | Medium

## <https://medium.com/codex/introduction-to-how-an-multilayer-perceptron-works-but-without-complicated-math-a423979897ac>

## Multi-layer perceptrons are a type of artificial neural network architecture that consists of multiple layers of nodes. It is a multi-layer structure containing input, hidden and output layers. It uses feed forward activation and backpropagation to calculate errors and update connection weights to optimize model performance. It also uses activation functions, which is a mapping of summed weighted inputs to the output of the neuron and governs the threshold at which the neuron is activated as well as the strength of the output signal.

## 

## A colorful chart with yellow squares Description automatically generated with medium confidence

## A graph with blue and white bars Description automatically generated

## From assignment 1, my correlation and mutual information analysis found that some features have both strong linear and non-linear relationships. MLP’s complex architectures allow it to capture linear and non-linear relationships hence I decided to use MLPs.

## Base Model

## 

## I first created a relatively simple model, using all the default hyperparameter values.

## 

## As we can see, the model is not performing very well and shows potential underfitting as test MSE is better than train MSE.

## 

## This could be due to the model inability to converge as stated in this warning here. Convergence warnings indicate that the optimization algorithm used during model training is having difficulty reaching a stable solution. We must first address the model’s convergence.

## Tuning

## First, I tuned the training parameters to improve convergence speed, namely learning rate, activation function and solver for weight optimization.

## Setting the learning rate of your neural network.

## <https://www.jeremyjordan.me/nn-learning-rate/>

## Learning rate determines the size of gradient descent. As seen above, if the learning rate is too high, it will overshoot the minimum point, preventing it from converging. Similarly, too small a learning rate will take too long to reach the minimum.

## The choice of activation functions can impact how quickly a model converges. For example, ReLU can mitigate vanishing gradients problem, allowing for more efficient training and convergence. Similarly, the of solver for weight optimisation can impact the speed of convergence. Different solver need to be tested to see which is the most suitable.

## A close-up of a logistic Description automatically generated

## In the feature space, I included identity, logistic and tanh activation functions alongside ReLU which is the default. For solvers, I included lbfgs and sgd with adam as the default. For learning rates, I tried both increasing and decreasing learning rates to see which is able to allow the model to converge.

## GridSearch output:

{'activation': 'logistic', 'learning\_rate\_init': 0.1, 'solver': 'adam'}

## However, even after trying these values, I still received the Convergence warning.

## Iterations

## Next, I increased the number of iterations to see if the optimization process is too minimal causing the model’s convergence speed too be very slow. Here, I tried iterations 300, 400 and 500 as the default was 200.

## A graph with red and blue lines Description automatically generated

## A screenshot of a computer code Description automatically generated

## At iteration 500, I no longer received the warning hence I will adopt max\_iter = 500. However, our model experiences drastic overfitting.

## L2 Regularization

## To counter the overfitting, I introduced L2 regularization. L2 prevents overfitting by penalizing large weights, adding a penalty term to the loss function. During the training process, the optimizer aims to minimize the loss. This encourages the model to prioritize simpler solutions with smaller weights, which are less prone to overfitting. For our model, the strength of the regularization is denoted as alpha. As we are looking to counter overfitting, I defined the feature space to have larger alpha values.

## 

## A graph with red and blue lines Description automatically generated

## As we see above, a higher regularization strength seemed to reduce variance and improve model generalization to test data. Here, I continued increasing L2 strength.

## 

## A graph with red and blue lines Description automatically generated

## Here, I decided to use 0.40 as the variance was the least of all the alpha values. I also noted that the variance across alpha values is quite unstable. This might be due to the model optimization process still being a bit unstable. As the model is still overfitting, I decided to reduce model complexity.

## Hidden layer size

## 

## Next, I decreased the hidden layer size. Hidden layer size controls the number of nodes in a layer and the number of layers. By reducing the hidden layer size, we reduce the model complexity hence preventing the model from fitting too closely to training data. If the model fits too closely to the training data, it will not be able to generalize well to unseen data hence overfitting. In the figure above, I tried layer sizes smaller than 100 as that is the default.

## A screenshot of a computer Description automatically generated

## Hidden layer size (10,) [index 1] had the least variance out of my the values I tested. Although test performance took a little hit, overfitting has been reduced drastically hence I will use (10,).

## Feature Selection

## Next, I did feature selection to see if I can still reduce the variance. To find feature importance, I ran a OLS (Ordinary Least Squares) Regression Results report. It aims to model the relationship between the dependent variable and one or more independent variables by fitting a linear equation to the observed data. For feature importance, I will be basing it off the p-value which helps me assess the statistical significance of each independent variable’s contribution to the model.

## A graph with blue and white bars Description automatically generated

## I also supplemented the selection with the Mutual Information analysis done in assignment 1 to ensure no strong non-linear relationships are being removed. I decided to remove 'apt','mrt','near', 'room' as they had higher p value scores and relatively low mutual information scores.

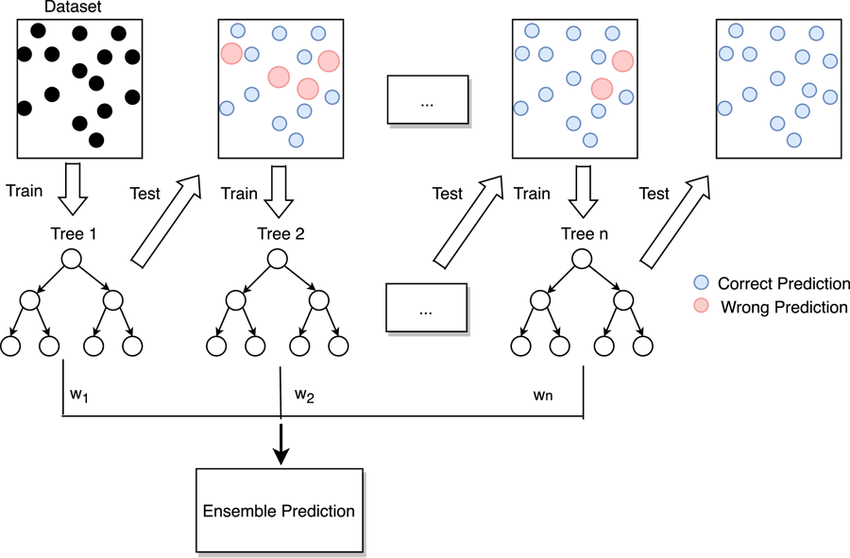
## A white background with black text Description automatically generated

## While the feature selection improved test scores, it increased variance hence I decided not to use it.

## Final MLP Model

## A white background with black text Description automatically generated

## Gradient Boosting Regressor

****

<https://www.researchgate.net/figure/Flow-diagram-of-gradient-boosting-machine-learning-method-The-ensemble-classifiers_fig1_351542039>

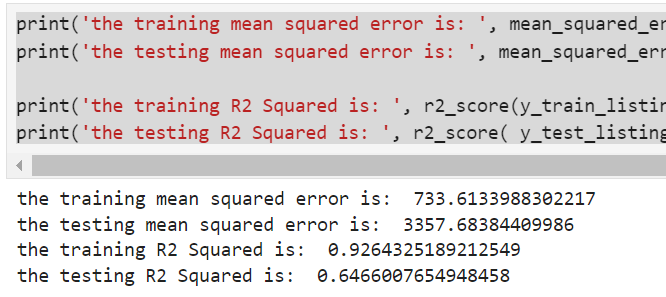
The gradient-boosting classifier combines multiple weak learners (typically decision trees) as a progression of the previous model. As seen in the figure above, boosting is where multiple learners are built sequentially hence errors committed by the previous model are learnt by the next model, improving generalization with every model. In regression, the predicted numerical value from each weak learner is assigned a weight which is proportional to the confidence and performance of the individual model. The final prediction is the weighted average of all the predictions (soft voting).

I decided to use gradient boosting as it has a completely different architecture from multi-layer perceptrons and allowed me to experiment with different model structure to deem which was most suitable for the data.

A screenshot of a computer

Description automatically generated

I first created a simple gradient boost model with all default hyperparameter values.



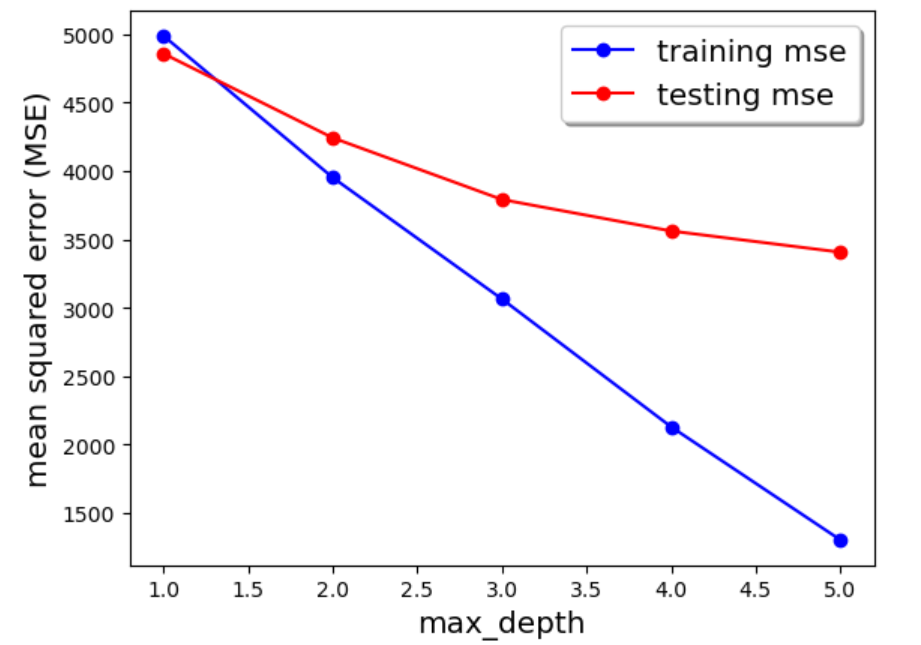
Immediately, the model experiences heavy overfitting as seen from the variance in training and test scores. To counter this, we can first reduce the model complexity.

**Tuning**

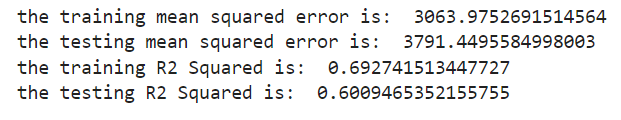
**Max depth**

I first tune the max\_depth of the individual weak learners. Max\_depth is a crucial hyperparameter that refers to the length of the longest path from the root node to leaf node. It influences the model’s complexity and can be reduced to prevent the model from learning too much from the train set.

As the default value for max\_depth is 6, I decided to define the feature space with smaller max\_depth values.



I decided to reduce the max\_depth to 3. While max\_depth 3 may not have the best test performance, max\_depth 4 and 5 experience too much variance between train and test for me to consider. On the other hand, for max\_depth 1 and 2, the test scores are too poor.



The model’s train MSE: 3063.97, test MSE: 3791.44

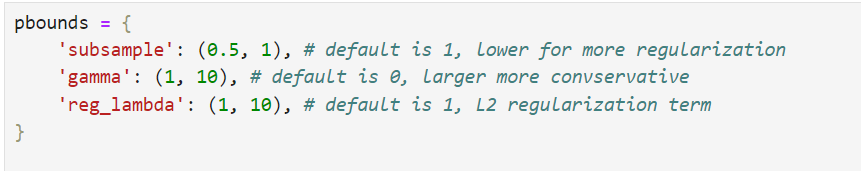
**Regularization**

As our model is still overfitting, I introduced some regularization hyperparameters namely subsample, gamma and reg\_lambda.

Subsample controls the proportion of the data that is used to train each tree rather than the entire dataset. The subset is typically sampled without replacement. Here, we can reduce the subsample rate, meaning that a smaller fraction of the data is used to train each weak leaner which introduces diversity in the ensemble, leading to a more robust model that generalizes better to unseen data, reducing overfitting.

Gamma controls the regularization on the decision tree’s leaf nodes. It is used to penalize individual weak learners for creating leaf nodes that do not significantly reduce the loss function. As more leaf nodes lead the expansion of the decision tree, gamma ensures that the model is not too complex, reducing the chances of overfitting. A larger gamma rate results in more conservative splitting.

Reg\_lambda is essentially the regularization strength of L2 regularization. It looks to penalize large weights and encourages the ensemble to use smaller weights hence simpler structures, a larger lambda value results in higher penalization (more in-depth discussion in MLP Tuning section).

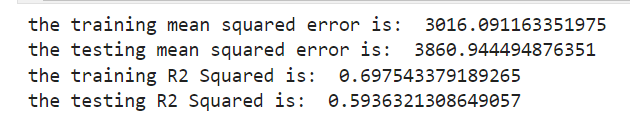


In my defined feature value space for Bayesian Optimization, I defined smaller than default values for subsample, larger than default values for gamma and larger than default values for reg\_lambda.

Bayesian Optimization Output:

Best Hyperparameters: {'gamma': 7.107174304893824, 'reg\_lambda': 1.0755955505222525, 'subsample': 0.9106722607689798}

Following the optimization output, I increased the gamma value from 0 to 7.107, slightly increased lambda value from 1 to 1.07 and decreased subsample from 1 to 0.9106.



## However, after introducing regularization, the variance between train and test increased. This could show that our feature space was not restrictive enough and we may need to introduce even more regularization hyperparameters.

## Even more regularization

## Here, I introduced an additional regularization hyperparameter, reg\_alpha. Reg\_alpha is the strength value of L1 Regularization. L1 regularization is similar to L2 regularization where large weights are penalized, however, L1 regularization is much harsher. L1 drive coefficient to exactly zero, creating sparsity in the model. L2 on the other hand penalizes but does not drive coefficients to zero. This means that L1 has a much greater regularization effect.

## 

## In this iteration, I further restricted the feature space of all the hyperparameters. As seen in the figure above, default values are no longer included in the feature space. By doing this, I ensure that there is minimum regularization effect. For reg\_alpha, since the default value is 0, I ensured that the values in the feature space were greater than it.

## Bayesian Optimization Output:

Best Hyperparameters: {'gamma': 6.0, 'reg\_alpha': 10.0, 'reg\_lambda': 5.0, 'subsample': 0.8}

## 

## Unfortunately, even after following the adding L1 regularization and restricting feature space, the variance was only reduced slightly and our model was still obviously overfitting.

## ETA / Learning Rate

## Here, I tried using the ensemble’s learning rate hyperparameter. Learning rate can be considered the most important hyperparameter of gradient boosting. It controls the contribution of each weak learner by adjusting the shrinkage factor. Smaller learning rates decreases how much a weak learner can contribute to the output hence requiring more weak learners to achieve good performance. But, it also makes the model more robust and less prone to overfitting.

## For our model, the default learning rate was 0.3, since I wanted to counter overfitting, I defined a feature space with smaller learning rate values.

## 

## A graph of a graph with red and blue lines Description automatically generated

## Here, I decided to use learning rate of 0.15. Learning rates 0.01 and 0.05 had poor testing scores while learning rate 0.2 had too much variance.

## 

## When comparing between learning rate 0.1 and 0.15, I also used R squared score to choose.

## Learning rate 0.1: train r2: 0.6006, test: 0.5561

## Learning rate 0.15: train r2: 0.6288, test: 0.5735

## When looking at variance between train and test r2 square values, there was not much difference between the two and learning rate 0.15 had the better test r-squared performance. Hence, I decided to adopt learning rate 0.15.

## Feature selection

## Lastly, I did some feature selection to further reduce overfitting. Based on p-values and mutual information scores, I decided to drop 'apt','mrt','near' and 'room' features.

## 

## However, the variance was slightly increased after feature selection. Notably, the test performance deteriorates slightly. This could show that the removed features helped the model generalize better to unseen data. Hence, I decided to keep the features.

## Final Gradient Boosting Model

## 

## Support Vector Regressor

## 

## <https://medium.com/it-paragon/support-vector-machine-regression-cf65348b6345>

## Support Vector Regressor are supervised machine learning algorithms that aims to find a function that approximates the relationship between input and target variables while minimizing prediction error. Unlike classification tasks, SVR seeks to find a hyperplane that best fits the data points in a continuous space (hence the curvy hyperplane in the figure above).

## SVR handles non-linear relationships by using a kernel function to map the data to a higher-dimensional space, making it powerful when there are complex relationships in the data. As the HR Analytics dataset has strong non-linear relationships, SVR is suitable for this problem.

## 

## I first started with a simple base model with the radial basis function kernel and a C value of 1.

## 

## As we can see, the model does not perform too well and shows signs of underfitting such as much larger testing scores compared to train scores. We can try to scale the model till overfitting first.

## Tuning

## C

## To increase model complexity, I decided to tune the C value.

## C is a regularization parameter that controls the trade-off between minimizing training error and minimizing the model complexity. C is a crucial hyperparameter that influences how much the SVR model penalizes deviations from desired output. The C value is inversely proportional to regularization strength, hence, we will look at increasing the C value.

## 

## When defining the feature space, I ensured to use values equal or greater than 1.

## 

## As we can see in the figure above, the larger the C, the better the model’s performance on both test and train sets. It shows that the regularization was indeed too strong, preventing the model from properly learning patterns from the data. This can be seen by the testing scores being better than the training scores, showing that the model is still underfitting.

## A green and black numbers Description automatically generated

## For the next iteration, I further increased the range of C values

## A graph of a training and testing Description automatically generated

## As we can see from the figure above, training finally outperforms test scores at around C = 200. However, as testing mse is still improving, I ran one more iteration.

## 

## Here, I ran even higher values of C to further scale the model.

## 

## Now, the model starts to overfit as seen at C = 300 when the test scores start to deteriorate while training scores improve. I decided to use C = 200 instead of 300 as the variance between train and test was just too large for C = 300 and there wasn’t any substantial improvement in test score over C = 200.

## 

## After tuning the C value, our model is now able to generalize much better as seen by the improvement of test scores. Our model experiences some slight overfitting, so we can introduce some regularization.

## Kernel

## Next, I decided to experiment with different kernel functions.

## While kernel functions primarily aim to handle non-linear relationships, certain kernels such as radial basis function (RBF) also introduce implicit regularization by considering only a subset of training data points (support vectors) in the higher-dimensional space. The regularization effect arises as not all data points contribute equally to the decision function and the SVR focuses on the most informative ones.

## 

## Other than the default RBF, I tried linear, polynomial and sigmoid kernel functions.

## The linear kernel is the simplest and represents a linear transformation, it is most effective for linearly separable or nearly linearly separable data.

## The polynomial kernel involves non-linear transformation that include polynomial terms. It introduces additional dimensions to the feature space, allowing the SVM to capture more complex relationships.

## The sigmoid kernel is based on the hyperbolic tangent function and is suitable for data where the relationship between features and target has a sigmoidal shape.

## A screenshot of a computer Description automatically generated

## After trying each, they were all unable to outperform the RBF kernel and none reduced the model’s variance, hence I will continue using RBF.

## Regularization

## To reduce the variance, I tuned epsilon and tolerance.

## Epsilon is a hyperparameter that defines the width of the margin of tolerance around the predicted value. The tuning of epsilon involves a trade-off between model flexibility and acceptance of errors, with a larger epsilon leading the model that does not fit the training data well but is flexible to noise and a smaller epsilon that results in closer fitting to train data but more sensitivity to noise.

## Tolerance is a hyperparameter that sets the tolerance for the optimization algorithm to converge. It is used to determine when the iterative optimization process should stop. A smaller tolerance (tol) may lead to a more accurate model on the training data, but there's a risk of overfitting if the model is too complex. Tight convergence criteria might result in a model that fits the training data noise rather than capturing the underlying patterns, potentially leading to poorer generalization on new, unseen data.

## As we are looking to reduce variance, I defined the feature space to be larger than the default epsilon and tolerance values.

## A white rectangular with green and blue text Description automatically generated

## Bayesian Optimization Output:

Best Hyperparameters: {'epsilon': 0.9999799958292386, 'tolerance': 0.044184370355385146}

A white background with black text

Description automatically generated

After tuning epsilon and tolerance, the model’s variance was reduced and the model’s ability to generalize is better.

**Feature Selection**

Next, I dropped 'apt','mrt','near' and 'room' features based on p-value and mutual information analysis. I looked to further reduce the model variance by reducing the dataset dimensionality.

A black text on a white background

Description automatically generated

After reducing the dimensionality, the model’s variance was further reduced.

## Summary

I will go through the problem, models created as well as hyperparameters tuned and feature engineering used for each model.

This is a regression problem as the target variable is the feature “price” is a continuous variable.

For this problem, I built 3 different types of models: Multi-layer perceptron, Gradient Boosting Regressor and Support Vector Regressor.

The metrics used for evaluation are Root Mean Squared Error and R-squared

For multi-layer perceptron, I started with train MSE of 5598 and test of 5390. The starting model also had convergence warnings to which I increased the learning rate from 0.001 to 0.1 and changed my activation function from reLU to logistic. I then increased my maximum iterations to 500, where I no longer received the convergence warning. To counter overfitting, I increased the L2 regularization strength from 0.0001 to 0.40 and decreased hidden\_layer\_size from (100,) to (10,).

My final MLP model's train MSE was 4742 and test MSE of 4812

For Gradient Boosting Classifier, I started with train MSE 733 and test MSE of 3357. To counter the model's heavy overfitting, I reduced the max\_depth from 6 to 3, increased gamma from 0 to 6, increased reg\_alpha from 0 to 10.0, increased reg\_lambda from 1 to 5 and decreased subsample from 1 to 0.8. As my model was still underfitting after regularization, I tuned learning rate and decreased the learning rate from 0.3 to 0.15

My final Gradient Boosting model's train MSE was 3700 and test MSE of 4051

For support vector regressor, I started with train MSE 7097 and test MSE 6507. The model showed signs of underfitting hence to scale the model, I increased the C value from 1 to 200. To further reduce variance, I increased epsilon from 0.1 to 0.9999 and tolerance from 1e-3 to 0.04418. Afterwards, for feature selection, I dropped 'apt', 'mrt;, 'near' and 'room' features.

My final support vector regressor model's train MSE was 4541 and test MSE of 4539.

**Choosing the best model**

Out of the three models, gradient boosting classifier had by far the best performance. Although the variance is a bit higher, the model shows a much stronger generalization ability. Hence, I will use Gradient Boosting.

# Conclusion

For the classification problem, I created 4 different models: Decision Trees, Random Forest, Gradient Boosting Classifier and Gaussian Process Classifier. I used accuracy as the metric as our classes are balanced

Decision Trees:

* Started with training accuracy of 0.7416 and test accuracy of 0.7372.
* Tuned hyperparameters by increasing max\_depth from 3 to 5 and min\_samples\_leaf from default 1 to 20.
* Achieved a final train accuracy of 0.7613 and test accuracy of 0.7601.

Random Forest:

* Initially, the model was heavily overfitted with training accuracy of 0.9925 and test accuracy of 0.7222.
* Regularized the model by defining max\_depth = 18, max\_leaf\_nodes = 30, min\_samples\_leaf = 17, min\_samples\_split = 17, and reducing n\_estimators from 20 to 19.
* Achieved a final train accuracy of 0.7651 and test accuracy of 0.7544.

Gradient Boosting Classifier:

* Started with a model having train accuracy of 0.7319 and test accuracy of 0.7158.
* Increased max\_depth from 1 to 10 for model scaling.
* Regularized the model by reducing learning rate, subsample, and changing splitting criteria and loss function.
* Achieved a final train accuracy of 0.8067 and test accuracy of 0.7511.

Gaussian Process Classifier:

* Started with a model having train accuracy of 0.7551 and test accuracy of 0.7447.
* Changed kernels from the default radial basis function to Matern and introduced polynomial expansion for the top 3 most important features.
* Achieved a final train accuracy of 0.7727 and test accuracy of 0.7622.

For the regression problem, I created 3 models. I used root mean squared error and R-squared

Multi-Layer Perceptron (MLP):

* Started with a train MSE of 5598 and test MSE of 5390.
* Addressed convergence warnings by increasing the learning rate from 0.001 to 0.1 and changing the activation function from ReLU to logistic.
* Increased maximum iterations to 500 to prevent convergence warnings.
* Applied L2 regularization by increasing the regularization strength from 0.0001 to 0.40.
* Counteracted overfitting by decreasing the hidden\_layer\_size from (100,) to (10,).
* Achieved a final train MSE of 4742 and test MSE of 4812.

Gradient Boosting Regressor:

* Started with a train MSE of 733 and test MSE of 3357.
* Addressed overfitting by reducing max\_depth from 6 to 3, increasing gamma from 0 to 6, reg\_alpha from 0 to 10.0, and reg\_lambda from 1 to 5.
* Decreased subsample from 1 to 0.8 to further reduce overfitting.
* Tuned learning rate by decreasing it from 0.3 to 0.15 to prevent underfitting.
* Achieved a final train MSE of 3700 and test MSE of 4051.

Support Vector Regressor (SVR):

* Started with a train MSE of 7097 and test MSE of 6507.
* Addressed underfitting by increasing the C value from 1 to 200.
* Further reduced variance by increasing epsilon from 0.1 to 0.9999 and tolerance from 1e-3 to 0.04418.
* Conducted feature selection by dropping 'apt', 'mrt', 'near', and 'room' features.
* Achieved a final train MSE of 4541 and test MSE of 4539.

# Reflection

For this project, I could have tried more models and techniques to better improve my model performance. For decision trees and random forests, I didn’t really follow the universal machine learning workflow as I didn’t scale them up all the way so I regretted that. Also, the report was extremely rushed for me where I did it in one day, 24 hours with no sleep. I feel that I could have explained some of the sections more coherently. This semester was a learning experience for me!

For the machine learning module, I think I could have definitely tried to utilize the CRISP-DM framework a bit more. I feel that it is a really good way to provide and communicate value and would be very useful in the workplace.

# Questions

# A screenshot of a web page Description automatically generated

# A screenshot of a computer Description automatically generated

# Answers

# A screenshot of a computer Description automatically generated

# A screenshot of a web page Description automatically generated

# A screenshot of a web browser Description automatically generated