Miscellaneous text

The purpose of the code below is to find a reasonable value for the n\_estimators parameter in a random forest model. The code executes a loop over a range of values for n\_estimators, training a random forest model with each value and using cross-validation to evaluate the model's performance. The mean and standard deviation scores are plotted against the different values of n\_estimators, allowing the user to study how the model's performance changes as n\_estimators is varied. The max\_depth and max\_features parameters are also set in the model, with max\_depth controlling the maximum depth of the tree models in the random forest and max\_features determining the number of features to consider when looking for the best split.

The code is then using cross-validation to evaluate the model's performance, and storing the mean and standard deviation of the scores. Finally, it is plotting these mean and standard deviation scores against the different values of n\_estimators.

### Explaining what cros\_val\_score() is:

The cross\_val\_score function is a utility function provided by scikit-learn that allows you to evaluate the performance of a machine learning model using cross-validation. It takes the model to be evaluated, the feature data (X), the target data (y), the number of folds (cv), and other optional parameters.

In this case, the cross\_val\_score function is being used to evaluate the performance of a random forest regressor (regressor) on the feature data X and target data y. The cv parameter specifies that the data should be split into 5 folds, and the n\_jobs parameter specifies that the computation should be parallelized across all available cores (-1 means use all cores).In cross-validation, the data is split into folds, where each fold is a subset of the data. The number of folds is specified using the cv parameter in the cross\_val\_score function. For example, if the cv parameter is set to 5, the data will be split into 5 folds. This means that the model will be trained on 4 folds of the data and tested on the remaining fold, 5 times, with a different fold used for testing each time. The final score for the model will be the average of these 5 scores. For example, if the data consists of 100 observations, and the cv parameter is set to 5, the data will be split into 5 folds of 20 observations each. The model will be trained on 4 folds (80 observations) and tested on the remaining fold (20 observations), 5 times, with a different fold used for testing each time. The final score for the model will be the average of these 5 scores.

The scores variable stores the cross-validation scores for each fold. The mean of these scores is calculated using scores.mean(), and the standard deviation is calculated using scores.std(). These values are then appended to the n\_estimators, mean\_scores, and std\_scores lists, respectively. These lists will be used later to plot the mean and standard deviation scores against the different values of n\_estimators.

Code for cross validation:

n\_estimators = []

mean\_scores = []

std\_scores = []

# Split the data into features (X) and target variable (y)

X = df.drop(columns=['Borough','Year','Crime rate'], axis=1)

y = df['Crime rate']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=30)

for i in range(10,200,10):

# Create a RandomForestRegressor object

regressor = RandomForestRegressor(n\_estimators = i, max\_features = 'auto', max\_depth = 30,random\_state=30)

# Use cross\_val\_score to evaluate the model

scores = cross\_val\_score(regressor, X, y, cv=5, n\_jobs=-1)

n\_estimators.append(i)

mean\_scores.append(scores.mean())

std\_scores.append(scores.std())

# Plot the mean scores

fig = px.line(x=n\_estimators, y=mean\_scores, title='Mean scores for different n\_estimators')

fig.update\_layout(xaxis\_title='n\_estimators', yaxis\_title='Mean score')

fig.show()

# Plot the standard deviation scores

fig = px.line(x=n\_estimators, y=std\_scores, title='Standard deviation scores for different n\_estimators')

fig.update\_layout(xaxis\_title='n\_estimators', yaxis\_title='Standard deviation score')

fig.show()

Based on the mean scores plot, it seems that the best number of estimators for the random forest model is around 110. After this point, the mean score seems to continuously fall off, indicating diminishing returns in model performance. Using values above 200 would likely be computationally expensive, and therefore not practical.

The standard deviation plot shows that the standard deviation of the scores is very low at any number of estimators. This is generally a good thing, as it indicates that the model is producing consistent results and is not too sensitive to the specific training data used.

2016\_2018 data:  
The random\_state parameter is a seed value used to initialize the random number generator for various operations in machine learning algorithms. It is used to ensure that the results of these operations are reproducible, meaning that they will be the same every time the algorithm is run with the same seed value.

In the context of a random forest model, the random\_state parameter can be used to specify the seed value used to generate the random sampling of features and observations when building the individual trees in the forest. This can be useful for ensuring that the model produces the same results every time it is trained, which can be useful for debugging or for comparing different models.

It is important to note that the random\_state parameter does not determine the final predictions made by the model, but rather the process by which the model is trained. Therefore, changing the random\_state value may result in a different model being trained, but the resulting predictions will depend on the data and the model's hyperparameters, rather than the random\_state value. Hence, we will keep the most important hyperparameters constant, tese being; .

The aim of the code is to find the combination of random\_state's between the 'test\_split()' and 'RandomRegressor()' functions, that optimize the model's score(). In general, as the score of the model (as measured by a performance metric like mean squared error or mean absolute error) increases, it can be assumed that the model is making more accurate predictions. This can in turn suggests that the borough-attribute importance scores are more reliable, as they are based on a model that is performing well.

The following code is designed to evaluate the performance of a random forest model using a range of different random states for both the test set split and the random forest model itself. To achieve this, the code first creates an empty dataframe called results\_df, which will be used to store the resulting scores.

Next, the code splits the data into features (stored in the variable X) and the target variable (stored in the variable y). It then starts a timer using the time library, which will be used to track the elapsed time of the loop.

The code then begins a loop that iterates over a range of random states for the test set split (stored in the variable random\_state\_test\_split). For each iteration, the code splits the data into training and testing sets using the train\_test\_split() function from scikit-learn, with the test set size set to 20% of the total data and the random state specified by the current iteration.

The code then begins another loop that iterates over a range of random states for the random forest model (stored in the variable random\_state\_rf). For each iteration, the code creates a new random forest regressor using the RandomForestRegressor() function from scikit-learn, with the specified random state. It then trains the model on the training data and makes predictions on the test data.

The code then evaluates the model's performance using the score() method, which returns a score indicating the model's accuracy on the test data. This method returns the coefficient of determination (R^2) of the prediction, which is a measure of how well the model fits the data. This score is stored in the variable regressor\_score and used to create a new dataframe (df1) containing the current random state for the test set split, the current random state for the random forest model, and the resulting score. This dataframe is then appended to the results\_df dataframe using the pd.concat() function.

After the inner loop has completed, the code displays the results\_df dataframe and calculates the elapsed time of the loop using the time library. The resulting dataframe will be used to find the maximum regressor score, with its corresponding random\_stae parameters for the relevant functions.

Choosing hyperparameters for randomforestregressor:  
The RandomForestRegressor() function from scikit-learn, has the hyperparameter's n\_estimators = 40, max\_features = 'auto', and max\_depth = 30. The value of 40 for n\_estimators was chosen because it strikes a good balance between accuracy and computational efficiency. A larger value for n\_estimators generally leads to better performance, but it also increases the computational time required to train the model. The choice to set max\_features to 'auto' was made because it is default value that can help to prevent overfitting and improve the generalization performance of the model. The value of 30 for max\_depth was chosen in order to build a model with good performance without overfitting. A deeper tree can lead to better performance on the training data, but it can also increase the risk of overfitting.

The RandomForestRegressor() function from scikit-learn, has the hyperparameter's n\_estimators = 50, max\_features = 'auto', and max\_depth = 40. The value of 50 for n\_estimators was chosen because it strikes a good balance between accuracy and computational efficiency. A larger value for n\_estimators generally leads to better performance, but it also increases the computational time required to train the model. The choice to set max\_features to 'auto' was made because it is default value that can help to prevent overfitting and improve the generalization performance of the model. The value of 40 for max\_depth was chosen in order to build a model with good performance without overfitting. A deeper tree can lead to better performance on the training data, but it can also increase the risk of overfitting. We increased the n\_estimators and max\_depth for the 1999\_2021 dataset as this data set had a lot more data.

### Evaluation of Random Forest Regressor

The `regressor.score()` method is used to evaluate the performance of a `RandomForestRegressor` model on the test data. This method returns the coefficient of determination (R^2) of the prediction, which is a measure of how well the model fits the data.

A value of `regressor\_score` for the R^2 means that the model explains 90.7% of the variance in the target variable. This is generally considered a good fit, as it indicates that the model is able to accurately predict the values of the target variable based on the features in the data.

However, the R^2 value is not always the best metric for evaluating the performance of a model, especially when the target variable is not normally distributed or has heteroscedasticity (i.e. the variance is not constant across the range of values). In these cases, other metrics such as mean absolute error (MAE) or mean squared error (MSE) may be more appropriate.

If the scatter plot does not show a clear relationship between the number of bedrooms and the price of the house, but the number of bedrooms has a high feature importance score when used to predict house prices, this may indicate that the number of bedrooms is not a strong predictor of the price of the house when considered in isolation, but it becomes important when combined with other features in the model.

In a machine learning model, each feature is considered in combination with all of the other features in the model, and the model learns how to weight the importance of each feature based on its contribution to the overall prediction. It is possible that the number of bedrooms may not have a strong relationship with the price of the house when considered on its own, but when combined with other features in the model, it becomes a strong predictor.

To understand why the number of bedrooms is important in the model, you may want to examine the other features in the model and how they interact with the number of bedrooms. You could also try training a simpler model that only includes the number of bedrooms as a feature and see how well it performs at predicting house prices. This can help you better understand the role that the number of bedrooms plays in the model.