Hyperiondev

TASK

Supervised Learning – Random Forests

Visit our website

Introduction

Decision trees are effective methods for predicting behaviours. However, they are prone to overfitting, which reduces their ability to perform well on new data. To address this issue, we introduce ensemble methods such as bootstrapping and bagging. These techniques combine multiple models to create more robust and accurate models. For instance, bagging (used in random forest) trains each tree on random subsets of data, while boosting improves accuracy by focusing on the instances where previous trees made errors. These methods help enhance the overall predictive power of decision trees.

ENSEMBLE METHODS

Ensemble techniques tackle overfitting by treating a tree's predictions as votes towards labels or combining predictions to get an average. Rather than trying to create one classifier that makes perfect predictions, ensemble methods aggregate the predictions of multiple classifiers into a single, improved prediction. Ensemble methods can be applied to various algorithms beyond decision trees, but they are particularly beneficial for tree-based models due to their susceptibility to overfitting.

Decision trees are widely used because they are easy to understand, apply, interpret, and visualise. However, despite these advantages, decision trees often lack robustness. This lack of robustness means that small changes, or perturbations, in the training data can lead to drastically different predictions during testing. This issue is known as high variance, a term used to describe models that are overly sensitive to the specific data they were trained on. High variance models tend to fit the training data too closely (overfitting), capturing noise or irrelevant patterns, which makes them unreliable when faced with new or unseen data.

In machine learning, we aim for models that can capture general patterns in the data, allowing them to perform well on new datasets. A model with high variance fails to generalise, meaning even a minor difference in the training sample can result in vastly different outcomes during testing.

There are different types of ensemble methods, but two of the most common for decision trees are:

- **Bagging:** This method, short for **bootstrap aggregating**, involves training multiple decision trees on different subsets of the data. A well-known example is random forest, which uses bagging to create a collection of decision trees that work together to make better predictions.
- **Boosting:** This technique builds trees sequentially, with each tree correcting the mistakes of the previous one. Boosting aims to create a strong model by focusing on instances the earlier trees struggled to predict.

Ensemble methods like bagging and boosting offer a way to overcome the high variance problem in decision trees by combining the strengths of multiple models, leading to better generalisation and improved predictive performance.

BOOTSTRAPPING

In bootstrapping, samples are drawn repeatedly from a dataset. If the dataset contains N instances, the bootstrapped samples are typically of smaller or the same size as N, and sampling is done with replacement – meaning the same instance can be selected more than once. This results in a bootstrap dataset, where some instances from the original data may appear multiple times, while others might not appear at all. The process is repeated to generate multiple bootstrap datasets.

This technique is important because it introduces diversity among the samples, allowing models trained on them to generalise better. Bootstrapping helps estimate the model's performance by simulating different training datasets, reducing overfitting, and improving the model's robustness, particularly when the available data is limited.

```
# Function to sample from a dataset with replacement
def subsample(dataset, ratio=0.7):
    sample = list()
    n_sample = round(len(dataset) * ratio)

    while len(sample) < n_sample:
        index = randrange(len(dataset))
        sample.append(dataset[index])
    return sample
```

This function simulates the bootstrapping process:

- `dataset` is the original dataset from which samples are drawn.
- `ratio` determines how large the new sample should be relative to the original.
- A new list `sample` is created to store the bootstrapped data.
- `randrange(len(dataset))` randomly selects an index, and the corresponding instance is added to the sample list.
- The same index can be chosen multiple times (sampling with replacement).

This function is useful for understanding how random samples are generated in the background when we use bagging or random forest algorithms. In practice, you won't usually implement this function yourself, as libraries like `scikit-learn` perform this

internally. However, seeing the process in code helps to demystify what happens when models train on randomly drawn subsets of data.

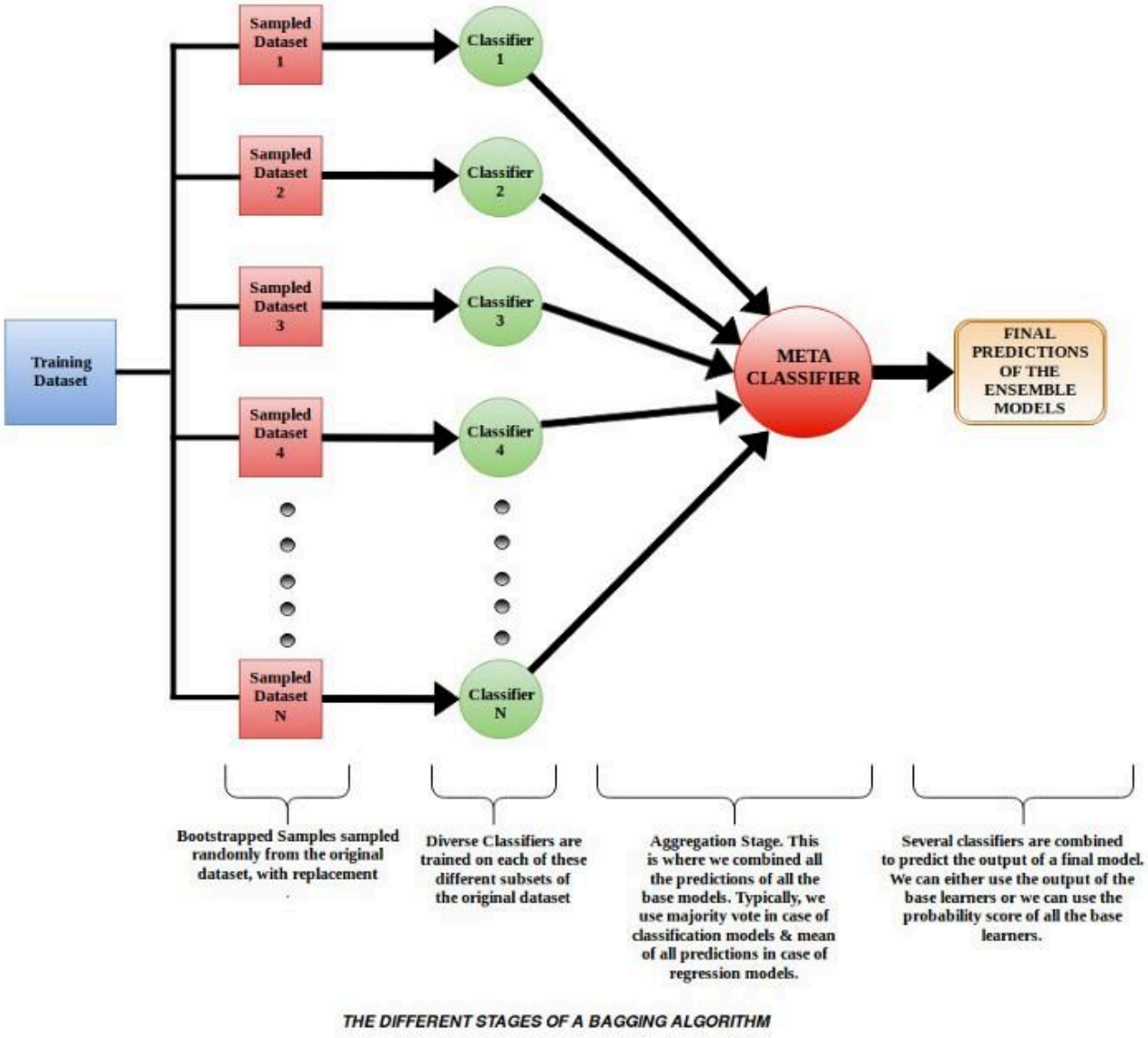
Out-of-bag error estimation

An important extension of the bootstrapping process is the **out-of-bag (OOB)** error estimation, which provides a method for evaluating the test error of a bagged model. Since each bootstrap sample leaves out about one-third of the data, the instances that are not included in the sample are known as OOB samples. These unseen instances can be used to estimate how well the model performs on new data. By assessing the model's performance on OOB samples, we get an approximation of test error without needing a separate validation/test set, making OOB error estimation a more efficient and practical alternative to traditional testing methods.

Thus, bootstrapping not only enhances model stability by reducing variance but also, through OOB error estimation, offers a reliable way to evaluate a model's generalisation ability without wasting valuable data on a separate test set.

BAGGING

As discussed earlier, bootstrapping is a technique where multiple samples are drawn from the original dataset with replacement, creating different training sets for model building. Bagging takes bootstrapping a step further by combining the predictions from these models to enhance accuracy and stability. It can be applied to both classification and regression problems, making it a versatile tool in machine learning.



The different stages of a bagging algorithm (Paul, 2018)

Bagging for classification

In classification tasks, after bootstrapping has generated several samples, each sample is used to train a separate model (typically decision trees). Each model is trained on a unique bootstrapped dataset, allowing the models to capture different aspects of the data. Once all models are trained, they make predictions independently. The final prediction is made by applying majority voting:

- Each model votes for a class label based on its training.
- The class label that receives the most votes is selected as the final prediction.

This voting mechanism helps reduce the variance of individual models by smoothing out any overfitting that may have occurred during training on different bootstrapped samples. Explore the **`BaggingClassifier`** from sklearn to experiment with implementing bagging practically.

Bagging for regression

For regression tasks, bootstrapping creates several samples of the data, and each model trained on these samples produces a numerical prediction. Unlike classification, where voting is used, regression problems require the averaging of the predictions from all models:

- Each model trained on a bootstrapped sample predicts a value.
- The final prediction is obtained by averaging the predictions from all models.

Averaging these predictions reduces the variability that might arise from individual models, ensuring a more stable and accurate prediction. Explore **`BaggingRegressor`** from sklearn to become familiar with the parameters and attributes of this ensemble class.

Table 1: Differences between bagging and bootstrapping

	Bootstrapping a Decision Tree	Bagging Decision Trees
Definition	Training a single decision tree on a bootstrapped sample.	Creating an ensemble of multiple decision trees using bootstrapping.
Purpose	Assess the performance and variability of one model.	Reduce variance and improve predictive performance.
Process	One bootstrapped sample, one decision tree.	Multiple bootstrapped samples, multiple decision trees.
Final Output	Single prediction from one decision tree.	Aggregate predictions from an ensemble of trees.
Use Cases	Evaluation of model stability.	Improved accuracy and robustness of predictions.

RANDOM FORESTS

Random forests are one of the most widely used bagging methods in machine learning, providing an improvement over traditional bagged trees by introducing a process known as decorrelation. **Decorrelation** is the process of reducing the similarity between the models in the ensemble to ensure they don't all make the same mistakes. This helps to increase the overall diversity of the models, making the final predictions more reliable.

Like regular bagging, random forests build multiple decision trees using bootstrapped training samples. However, a key difference is that, instead of allowing all features to be considered at every split, random forests randomly select a subset of features for each tree at each split. This randomisation reduces the correlation between the trees, meaning they become more independent of each other. Because of this decorrelation, the strong predictor variables that could dominate every tree in regular bagging are not always present in each tree, leading to more varied models.

By decorrelating the trees, random forests ensure that the trees in the ensemble are less likely to make the same errors. When you average the predictions from these less-correlated trees, the result is a model that has a much lower variance than if the trees were highly correlated, making it more robust and accurate.

BOOSTING

Boosting is an ensemble learning technique that aims to improve the accuracy of models by combining several weaker models, typically decision trees, into a single strong model. Unlike bagging, where models are trained independently, boosting works **sequentially**, with each new model focusing on the errors made by the previous ones. The process begins by training the first model on the entire dataset, and then the misclassified or poorly predicted examples are given higher importance. The next model in the sequence is trained to correct these mistakes. This cycle continues, with each model being adjusted to improve performance. By the end, all the models' predictions are combined to make a more accurate overall prediction.

Boosting is particularly effective in reducing bias, making it useful when simpler models struggle to capture complex patterns in the data. Boosting can be effectively applied to both regression and classification problems by focusing on improving the model's performance in each step through sequential learning.

Boosting in classification

In classification problems, boosting works by focusing on correcting misclassifications. Initially, a weak classifier is trained, and the misclassified examples are given higher weights. The next classifier is trained to pay more attention to these hard-to-classify examples. This process continues, with each new model correcting the mistakes of the previous ones. The final prediction is based on the weighted vote or average of all the classifiers. Popular boosting algorithms used for classification include **`AdaBoostClassifier`** and **`GradientBoostingClassifier`**.

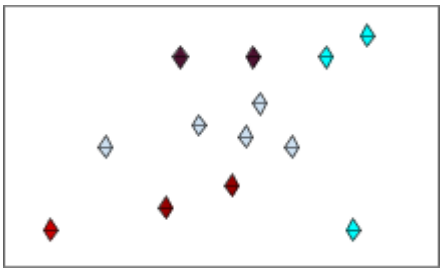
Boosting in regression

In regression tasks, boosting works by minimising the errors between the predicted and actual values. The first model is trained to predict the target variable, and the residuals (the differences between the actual and predicted values) are calculated. The next model is trained to predict these residuals. This process is repeated, with each subsequent model learning from the errors of the previous models. The final prediction is the sum of the predictions from all the models, making the overall prediction more accurate. A common boosting algorithm used for regression is **`GradientBoostingRegressor`**.

In both cases, boosting helps by iteratively reducing errors, either by correcting misclassifications in classification tasks or minimising residuals in regression problems.

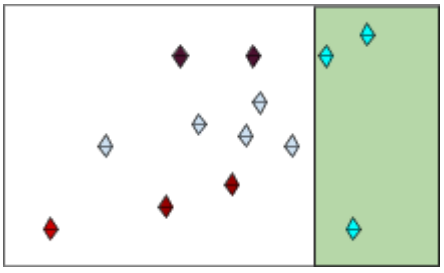
The steps for boosting are as follows:

1. A sample is drawn from a dataset.



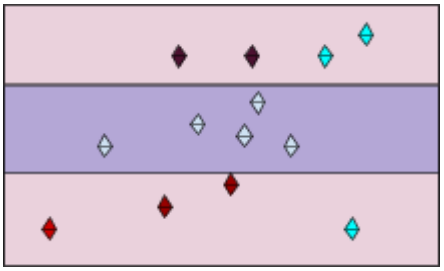
2. An initial model is trained on this sample, with each instance having equal weighting.

3. The initial model generates predictions for the whole dataset.



4. The error of the initial model is computed and the weighting of all incorrectly predicted instances is increased.

5. A second model is trained on a sample of this weighted version of the dataset.



6. In the end, all models are weighted and the mean or majority of all models' predictions are taken.

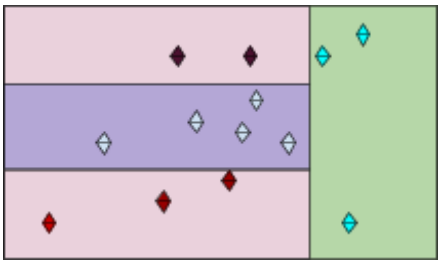


Table 2: Summary of differences between bootstrapping and boosting

	Bootstrapping	Boosting
Purpose	Reduce variance (overfitting) by training models on different subsets of the data.	Reduce bias (underfitting) by training models sequentially and correcting mistakes.
How Models are Trained	Models are trained independently on different bootstrapped samples.	Models are trained sequentially, where each model focuses on the mistakes of the previous one.
Sampling	Data is sampled with replacement to create different datasets for each model.	All models are trained on the same data, but weights are adjusted after each iteration.
Focus of Training	No focus on mistakes of previous models; each model works on its own dataset.	Each new model focuses on correcting errors made by previous models.
Algorithm Examples	Random forest (bagging), bagged decision trees.	AdaBoost, Gradient Boosting, XGBoost.

HYPERPARAMETERS

Hyperparameter tuning is about adjusting certain settings in a machine learning model to help it perform better. These settings, called hyperparameters, need to be chosen before the model starts learning from the data. Unlike the model parameters, which the model learns by looking at the data, hyperparameters are like rules you set in advance to guide the learning process.

- **Model parameters** are the values the model uses to make predictions, and they get updated during the training process. For example, in a decision tree, this could be how the tree decides to split the data at each step.
- **Hyperparameters** are settings that control how the model learns, like how big the tree can grow or how many trees to build in a forest.

Some examples of hyperparameters include:

- **Max depth:** This limits how deep a decision tree can go. If a tree is too shallow, it won't learn enough. If it's too deep, it might memorise the data instead of learning general patterns.
- **Number of trees:** In models like random forest, this is how many decision trees you want to create. Too few trees might not give enough variety in predictions, while too many can make the model overly complicated.
- **Learning rate:** Think of this as how quickly the model learns from its mistakes. If it learns too slowly, it might take forever to get good. If it learns too quickly, it might skip over important patterns.

Choosing the right hyperparameters is crucial because it can make your model much better at solving the problem. If the settings are off, the model may either underfit (be too simple to capture the data) or overfit (become too complex and memorise the data, which makes it perform poorly on new data).

There are several ways to tune hyperparameters to optimise the performance of a machine learning model. Some options include:

- **Grid search:** This method tries out all possible combinations of hyperparameters (e.g., different learning rates, tree depths, etc.) to find the best one.
- **Random search:** Instead of trying every possible combination, random search picks hyperparameter values at random to find good ones faster.
- **Bayesian optimization:** This is a more advanced method that focuses the search on the most promising hyperparameters based on past results.

To check which hyperparameters work best to optimise performance, we use techniques like **cross-validation**, where we test the model on different pieces of the data. This helps ensure the model works well, not just on the training data but also on unseen data.

CROSS-VALIDATION

Cross-validation is a technique used to evaluate how well a machine learning model performs on data it hasn't seen before. Instead of just splitting the data into one training and one testing set, cross-validation splits the data into several parts (called folds), so that every data point gets a chance to be used for both training and testing. This ensures that the model doesn't just perform well on one specific set of data but can generalise to new data. Here's how it works:

1. **Split** the dataset into several smaller groups called folds. Typically, 5 or 10 folds are used.
2. **Train** the model on all the folds except one. The fold that's left out is used for testing.
3. **Repeat** this process until each fold has been used for testing once.
4. **Average** the results from each round to get a more reliable performance score (like accuracy, or mean squared error for regression).

By doing this, cross-validation helps to assess how well the model will perform on unseen data.

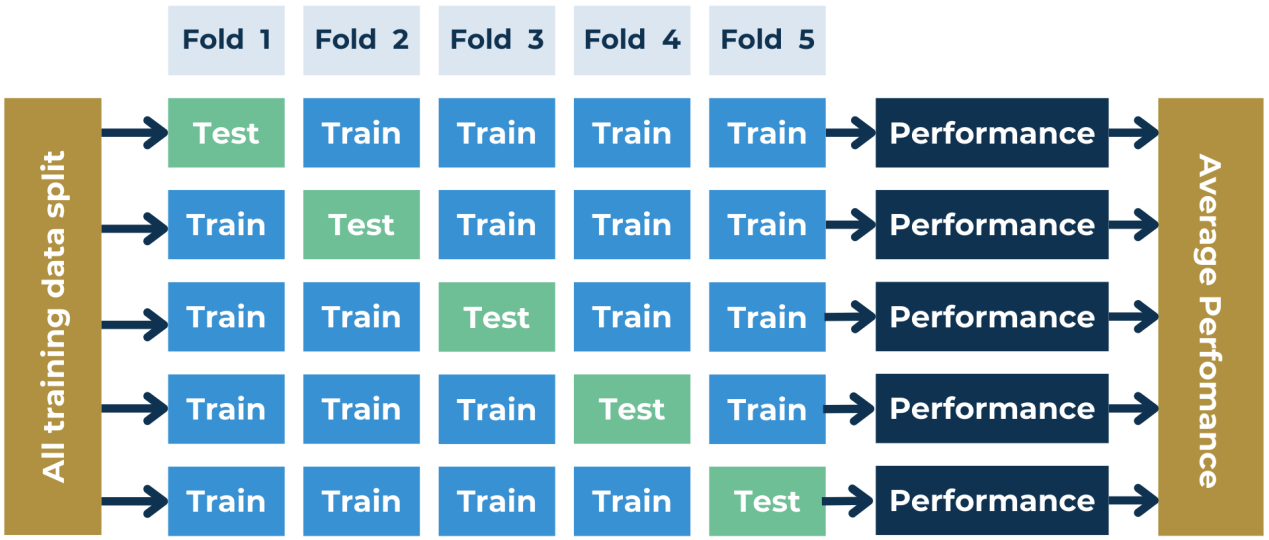


Figure demonstrating cross-validation

GridSearchCV

GridSearchCV builds on cross-validation and is used to **tune hyperparameters**. After splitting the data using cross-validation, GridSearchCV goes a step further and tries different combinations of hyperparameters to find the best settings for the model. Here's how it works:

1. **Define a set of hyperparameters:** For example, if you're working with a decision tree, you might want to try different tree depths or numbers of splits.
2. **Test all combinations:** GridSearchCV will automatically try every combination of the hyperparameters you specified. For example, it will test each combination of maximum tree depth, number of trees, or learning rates.
3. **Use cross-validation:** For each combination of hyperparameters, the model is trained and evaluated using cross-validation.
4. **Pick the best combination:** Once all combinations are tested, GridSearchCV chooses the set of hyperparameters that give the best performance based on the cross-validation results.

GridSearchCV automates the process of finding the best hyperparameters, saving time and ensuring that your model is tuned to perform as well as possible. It uses cross-validation to make sure that the best set of hyperparameters is selected based on how well the model performs on different parts of the data.

GridSearchCV example

During cross-validation, different combinations of hyperparameters are tested to find the optimal set that results in the best performance across the folds.

We start off with creating a parameter grid that contains values for the hyperparameters we would like to test the combinations of. Open the `Random_Forests_diabetes.ipynb` file in your folder to see the complete example below.

```
# Hyperparameter tuning for random forest using GridSearchCV
from sklearn.model_selection import GridSearchCV

params = {
    'max_depth': [2,3,5,10,20],
    'min_samples_split': [5,10,20,50,100,200],
    'max_features': [1,2,3,4,5,6,7]
}
```

We then use **GridSearchCV()** to fit our random forest model, the estimator called **rf**, on 4 folds using all possible combinations of our hyperparameters in params.

```
# Instantiate the grid search model
grid_search = GridSearchCV(estimator=rf,
                           param_grid=params,
                           cv = 4,
                           n_jobs=-1, verbose=1, scoring="accuracy")

grid_search.fit(X_train, y_train)
```

After fitting, we can obtain the configuration of the best-performing random forest model using **grid_search.best_estimator_**.

```
# Check best-scoring hyperparameters
print(grid_search.best_score_)
rf_best = grid_search.best_estimator_
rf_best
```

Output:

```
0.7858070757324488
```

```
RandomForestClassifier(max_depth=10, max_features=2, min_samples_split=5,
                       random_state=42)
```

In this case, the final model that is robust and generalises well to the data has a maximum depth of 10 levels, considers only 2 features at each split, and will not split a node further if it has less than 5 samples.

FEATURE SELECTION

When working with real-world data, not all features (or columns) in your dataset will be equally useful for building a predictive model. Some features may be irrelevant, redundant, or even harmful to the model's performance. Feature selection is the process of identifying and selecting the most important features to use in model training. This can improve model performance, reduce overfitting, and make the model easier to interpret.

Feature selection matters because it allows the model to focus on the most relevant features, reducing the chance of overfitting and improving both accuracy and computational efficiency. By using fewer but more meaningful features, the model becomes easier to understand and faster to train, while potentially performing better on new, unseen data.

There are several strategies for selecting features, and they generally fall into three main categories: **filter methods**, **wrapper methods**, and **embedded methods**.

Filter methods

Filter methods apply statistical techniques to each feature independently of the model to determine its relevance. These methods rank the features based on metrics like correlation, **Chi-square tests**, or **ANOVA F-tests**. For example, you can use a correlation matrix to identify which features are most strongly associated with the target variable:

```
import seaborn as sns
import matplotlib.pyplot as plt

corr_matrix = df.corr()
sns.heatmap(corr_matrix, annot=True)
plt.show()
```

Wrapper methods

Wrapper methods approach feature selection as a search problem, evaluating multiple combinations of features by training a model on each. One commonly used wrapper method is Recursive Feature Elimination (RFE), which recursively removes the least important features until the desired number is reached:

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()
rfe = RFE(model, n_features_to_select=5)
fit = rfe.fit(X_train, y_train)
print("Selected features:", X_train.columns[fit.support_])
```

Embedded methods

Embedded methods perform feature selection during the model training process itself. Tree-based models like random forests naturally fall into this category, as they include

built-in measures of feature importance. After training a random forest, we can inspect which features contributed most to the model:

```
from sklearn.linear_model import LassoCV

model = LassoCV()
model.fit(X_train, y_train)

# Select features with non-zero coefficients
selected_features = X_train.columns[model.coef_ != 0]
print("Selected features:", selected_features.tolist())
```

L1 regularisation can be a powerful embedded method when working with models where sparsity (few features) is preferred. This helps reduce complexity and improve generalisation.

In practice, filter methods are often a good starting point when working with large datasets, as they are fast and simple to apply. Wrapper methods, while more computationally intensive, allow for more fine-tuned control and can result in better performance. Embedded methods such as L1 regularisation or tree-based model importances integrate feature selection within model training and provide useful insights with minimal overhead.

Feature selection is an essential step in the machine learning pipeline. A thoughtful approach to selecting the most informative features can improve model robustness, reduce complexity, and lead to better generalisation performance.

Instructions

First, read and run the Jupyter notebooks in this task's folder before attempting the practical task. Feel free to write and run your own example code before doing the tasks in order to become more comfortable with the concepts covered within.

Also, explore the **`RandomForestClassifier`** documentation, including the **`feature_importances_` property**, to become familiar with the practical implementation of some of the concepts covered.

Practical task

1. Create a Jupyter notebook called **random_forest_titantic.ipynb**.

2. Load, clean, and preprocess the data from the **titanic.csv** file. You may reuse any code you may have used previously to clean and preprocess this dataset.
3. Create a **random forest** to predict the survival of passengers on the Titanic.
4. From the random forest model, determine which of the features is the one that contributes the most to predicting whether a passenger survives or not.
5. Tune the `n_estimators` (the number of trees in the model) and `max_depth` (the maximum depth of each tree) parameters.
6. Report the accuracy of all models and report which model performed the best, including the values for `n_estimators` and `max_depth` that the best model had.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Challenge

Use this opportunity to extend yourself by completing an optional challenge activity.

Follow these steps:

1. Add to your **random_forest_titantic.ipynb** Jupyter notebook by creating a bagged and boosted tree to predict the survival of passengers on the Titanic.
2. Tune the `n_estimators` (the number of trees in the model) and `max_depth` (the maximum depth of each tree) parameters for the bagged and boosted tree models.
3. Report the accuracy of all models and report which model performed the best, including the values for `n_estimators` and `max_depth` that the best model had.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

REFERENCES

Paul, S. (2018, November 30). *Ensemble learning: Bagging, boosting, stacking and cascading classifiers in machine learning using SKLEARN and MLEXTEND libraries.* **https://medium.com/@saugata.paul1010/ensemble-learning-bagging-boosting-stacking-and-cascading-classifiers-in-machine-learning-9c66cb271674**