**Data Understanding**

the data set includes credit card transactions made by European cardholders over a period of two days in September 2013. Out of a total of **2,84,807 transactions, 492 were fraudulent.** This data set is highly unbalanced, **with the positive class (frauds) accounting for just 0.172% of the total transactions.** The data set has also been modified with Principal Component Analysis (PCA) to maintain confidentiality. Apart from ‘time’ and ‘amount’, all the other features **(V1, V2, V3, up to V28)** are the principal components obtained using PCA. The feature 'time' contains the seconds elapsed between the first transaction in the data set and the subsequent transactions. The feature 'amount' is the transaction amount. The **feature 'class' represents class labelling**, and it takes the value 1 in cases of fraud and 0 in others.  
  
The distribution plots of the variables were Gaussian, which might indicate the effects of transformations that had already occurred on the data set.

# Project Pipeline

The project pipeline can be briefly summarized in the following four steps:

* **Data Understanding:** Here, you need to load the data and understand the features present in it. This would help you choose the features that you will need for your final model.
* **Exploratory data analytics (EDA):** Normally, in this step, you need to perform univariate and bivariate analyses of the data, followed by feature transformations, if necessary. For the current data set, because Gaussian variables are used, you do not need to perform Z-scaling. However, you can check if there is any **skewness** in the data and try to mitigate it, as it might cause problems during the model-building phase.  
    
  Can you think why skewness can be an issue while modelling? Well, some of the data points in a skewed distribution towards the tail may act as outliers for the machine learning models which are sensitive to outliers and hence that may cause a problem. Also, if the values of any independent feature are skewed, depending on the model, skewness may affect model assumptions or may impair the interpretation of feature importance.

* **Train/Test Split:** Now you are familiar with the train/test split, which you can perform in order to check the performance of your models with unseen data. Here, for validation, you can use the k-fold cross-validation method. You need to choose an appropriate k value so that the minority class is correctly represented in the test folds.
* **Model-Building/Hyperparameter Tuning:** This is the final step at which you can try different models and fine-tune their hyperparameters until you get the desired level of performance.

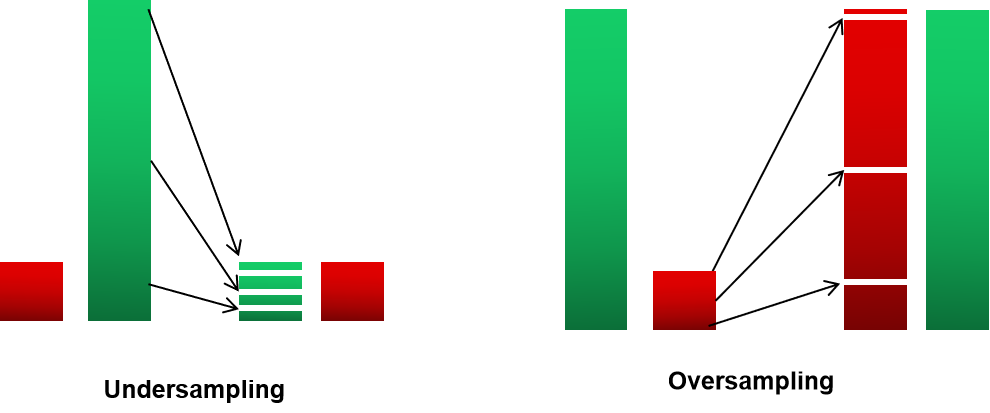
# Class Imbalance

Once you load the data, you can easily observe the disparity between the fraudulent and non-fraudulent cases. In machine learning terms, this situation is known as ‘**class imbalance**’

As you observed, the data shows a high class imbalance. Over 2,00,000 cases are mapped to 0, but hardly 500 cases are mapped to 1. Any machine learning algorithm would work well when there is equal representation of each of the classes. However, in this case, no matter which model you build, the underlying algorithm will learn more about the non-fraudulent cases rather than the fraudulent ones. Therefore, the loss function optimisation will be heavily biased to the former type of data. This is known as the **‘minority class problem’.**

Now, we can use certain methods to mitigate this problem. They are as follows:

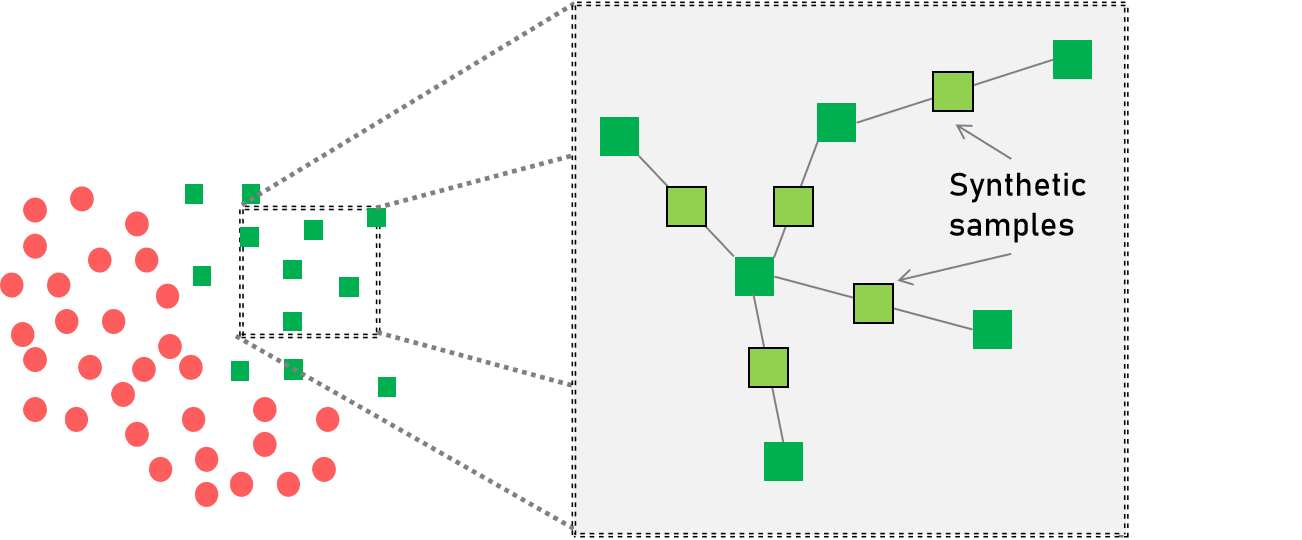
* **Undersampling:** In this method, you have the choice of selecting fewer data points from the majority class for your model-building process. In case you have only 500 data points in the minority class, you will also have to take 500 data points from the majority class; this will make the classes somewhat balanced. However, in practice, this method is not effective because you will lose over 99% of the original data.
* **Oversampling:** Using this method, you can assign weights to randomly chosendata points from the minority class. This way, the occurrence of each data point will be multiplied by the assigned weight, and the machine learning algorithm will now be able to focus on this class while optimising the loss function. However, this method does not add any new information and may even exaggerate the existing information quite a bit



Undersampling and Oversampling

* **Synthetic Minority Over-Sampling Technique (SMOTE):** In this process, you can generate new data points, which lie vectorially between two data points that belong to the minority class. These data points are randomly chosen and then assigned to the minority class. This method uses K-nearest neighbours to create random synthetic samples. The steps in this process are as follows:

1. Randomly selecting a minority point A
2. The k nearest neighbours for that data point belonging to the same are found and then a random point, B form the k\_neighbours is selected.
3. Specifying a random value in the range [0, 1] as λ.
4. Generating and placing a synthetic sample between the two points A and B on the vector located at λ% from the original point A.



**New synthetic sample points are added**

**in between the two homogenous class points.**

* **ADAptive SYNthetic (ADASYN):** This is similar to SMOTE, with a minor change in the generation of synthetic sample points for minority data points. For a particular data point, the number of synthetic samples that it will add will have a density distribution, whereas, for SMOTE, the distribution will be uniform. The aim here is to create synthetic data for minority examples that are harder to learn, rather than the easier ones.

To sum it up, ADASYN offers the following advantages:

1. It lowers the bias introduced by the class imbalance.
2. It adaptively shifts the classification decision boundary towards difficult examples.

You learnt how the 'class\_weight' parameter helps you in oversampling for a random forest classifier. You can also try out the 'class\_weight' parameters for the other classification algorithms as well.

But as you can recall, oversampling doesn't add any new information to the data - it just repeats it. So let's also take a look at how the other SMOTE and ADASYN can be implemented using Python.

# Introduction to KNN

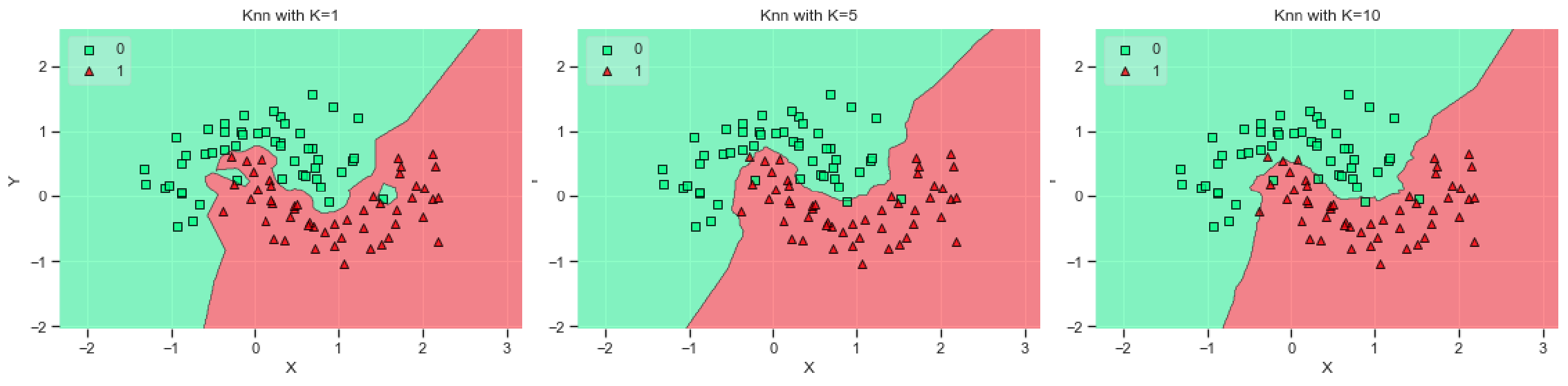
**KNN:** K-nearest neighbour is a simple, **supervised machine learning algorithm** used for both classification and regression tasks. It performs these tasks by identifying the neighbours that are nearest to a data point. For classification tasks, it takes the majority vote and for regression tasks, it takes the average value from the neighbours.

The **k** in KNN specifies the **number of neighbours** that the algorithm should focus on. For example, if k = 3, then, for a particular test data, the algorithm observes the three nearest neighbours and takes the majority vote from them. Depending on the majority of the classes from the three nearby points, the algorithm classifies the test data.

the k value should be an **odd number** because you have to take the majority vote from the nearest neighbours by breaking the ties.

Also, note that the k in k-means is different from the k in KNN. k in k-means stands for the number of clusters which can have any number of data points while k in KNN stands for the number of points that the model would consider to make predictions.

But what should the ideal k value be? And what happens if we change it? Let’s understand this with the help of the following diagrams which show the separation boundary between 2 classes for different values of 'k'.



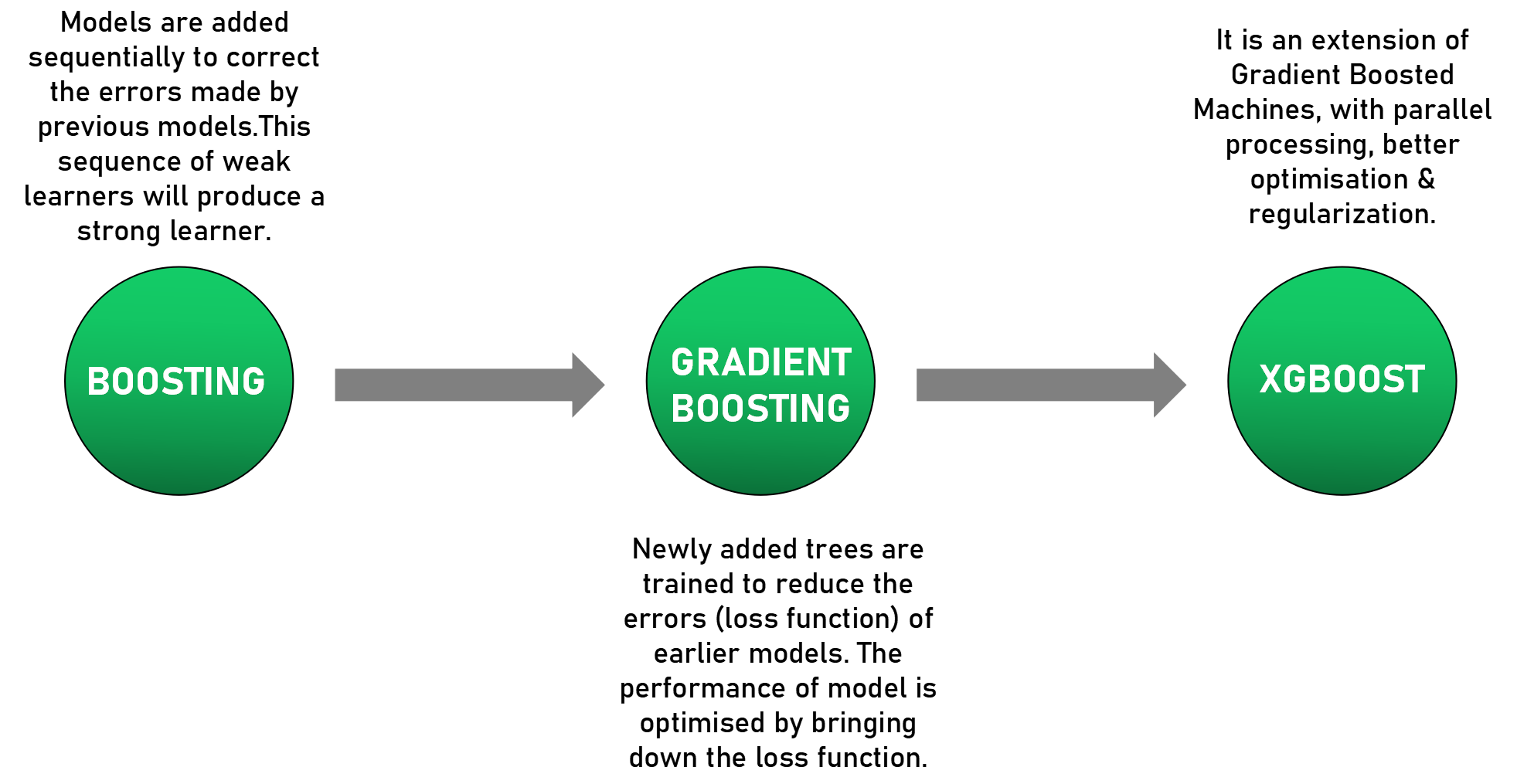
KNN

As you can see, an increase in the k-value causes the decision boundary between the two classes to become smooth.

* When **k = 1** for a data point, the model is observing the **immediate neighbour**, i.e., it is understanding the noise as well causing it to overfit the data.
* When **k = 5**, the model observes the **five nearest neighbours** for a data point. Therefore, the decision boundary starts becoming smooth.
* When **k = 10**, the model observes the **10 nearest neighbours** for a data point. Here, the decision boundary becomes smoother.

# XGBoost

XGBoost stands for **‘eXtreme Gradient Boosting'**. It is a decision-tree-based ensemble ML algorithm that uses a **gradient boosting framework**. It is a highly effective and widely used machine learning method and has applications for structured as well as unstructured data. For structured or tabular data, XGBoost, since its inception, XGBoost has dominated most of the **Kaggle** competitions and was even used in challenges such as the **Netflix Prize**.



To understand different tree boosting algorithms, let’s recap:

* **AdaBoost** is an iterative way of adding weak learners to form the final model. For this, each model is trained to correct the errors made by the previous one. This sequential model does this by adding more weight to cases with incorrect predictions. By this approach, the ensemble model will correct itself while learning by focusing on cases/data points that are hard to predict correctly.
* Next, we will talk about **Gradient Boosting**. You learned about gradient descent in the earlier module. The same principle applies here as well, where the newly added trees are trained to **reduce the errors (loss function)** of earlier models. So, overall, in gradient boosting, we are optimizing the performance of the boosted model by bringing down the loss function.
* **XGBoost** is an extended version of **gradient boosting**, where it uses more accurate approximations to tune the model and find the best fit. The added features here are:
  + **Optimization through second-order derivatives**: The second-order partial derivative of the loss function provides a more detailed picture of the gradient direction. Thus, you can easily find the global minima in comparison to doing it using the first-order derivative.
  + **Advanced regularisation (Lasso and Ridge)** to penalize the model based on the number of trees and the depth of the model: Higher the number of trees, higher will be the number of nodes in each tree and greater will be the penalty attached to it. Whenever you need to add a new node, you will need to check for a minimum reduction in the loss. If there is no significant reduction, you will not create the node.
  + Fast learning through parallel and distributed computing enables quicker model exploration.

Because of parallel processing (speed) and model performance, we can say that XGBoost is gradient boosting on steroids.

# Model Selection and Understanding - I

Now that you are aware of most of the ML algorithms, you know that there is no single algorithm for solving all problems. You can test out different models on your data and tune them in order to find the best fit. However, sometimes, the steps for testing each model for the data will take a lot of computational resources and time. You have to understand the type of data available with you and identify which model will be the best fit for it.

**logistic regression** works best when the data is **linearly separable** and **needs to be interpretable**. However, the biggest problem is mostly when the data will have a good amount of **overlap between the classes** present in it.

As in logistic regression, **KNN** is also highly interpretable. But there is a problem while computing the neighbours of a particular test point. You have to find the distance between all the data points and test data point to come up with the classification of a single neighbour. This task will need a lot of computation and is not a good choice when you have **large amounts of data**.

The **decision tree model** is the first choice when we want the output to be **intuitive** and want to explain the results to people without a technical background and other stakeholders. We can find the exact node at which a point has been classified and the reason behind it. But when the tree fits all the samples in the training data perfectly, **overfitting** occurs. **Decision tree** checks the data in many ways. Hence, **decision trees tend to overfit if left unchecked**.

However, working with large data becomes challenging because quadratic computing requires a lot of training time for large data sets.

# Model Selection and Understanding - II

Imagine you are working as a data scientist at a startup in the **healthcare domain**. The data you are working on regularly is **tabular/structured**, and you have comfortably applied the decision tree on it to predict whether the person has skin cancer or not. The available features are not highly correlated such as the **skin texture, radius of the skin cells, smoothness**, etc. But now you decide that you need to work with **images** rather than the structured data.

apart from the insight that **pixelated values** are highly correlated with each other, we do not have other features available. Because most of the supervised algorithms focus on these features, you will have to use models that can extract features from the present unstructured data.

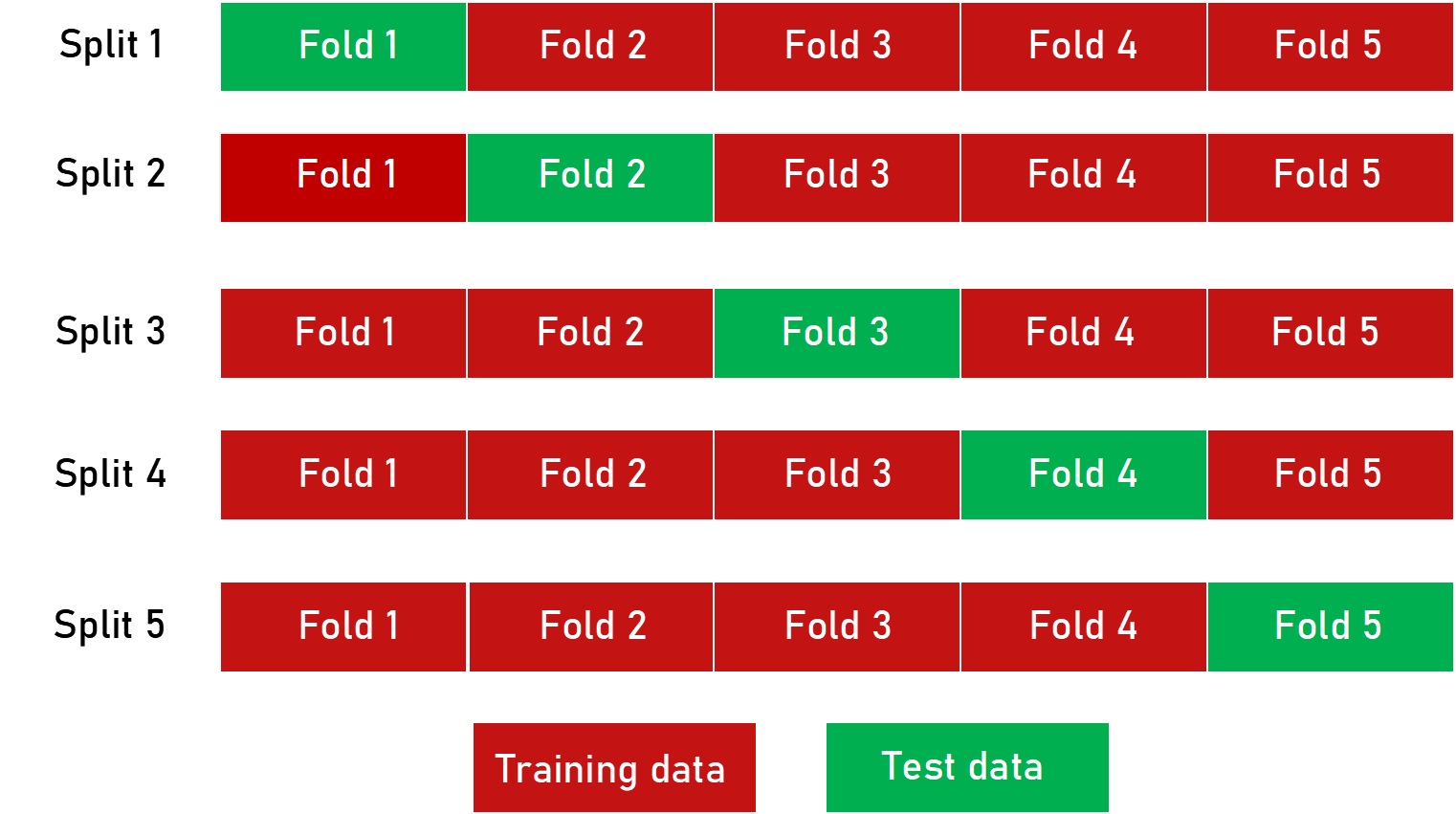
the thumb rule is that whenever you have **structured data**, you can use high-performing models such as **random forest/XGBoost.** However, when you have **unstructured data** such as an image/text, it is better to use **neural networks/LSTM** to extract the features from the unstructured features.

# Hyperparameter Tuning

You studied model evaluation in Machine Learning - II, in which we focused on **Hyperparameters and Cross-Validation**. You can consider hyperparameters as model controls/settings. Because the ideal settings of a model used for a particular data set will differ from those of models used for other data sets, we need to tune the model every time we use it to come up with the best results. Let’s revise the topics covered and understand why we perform hyperparameter tuning in the video given below.

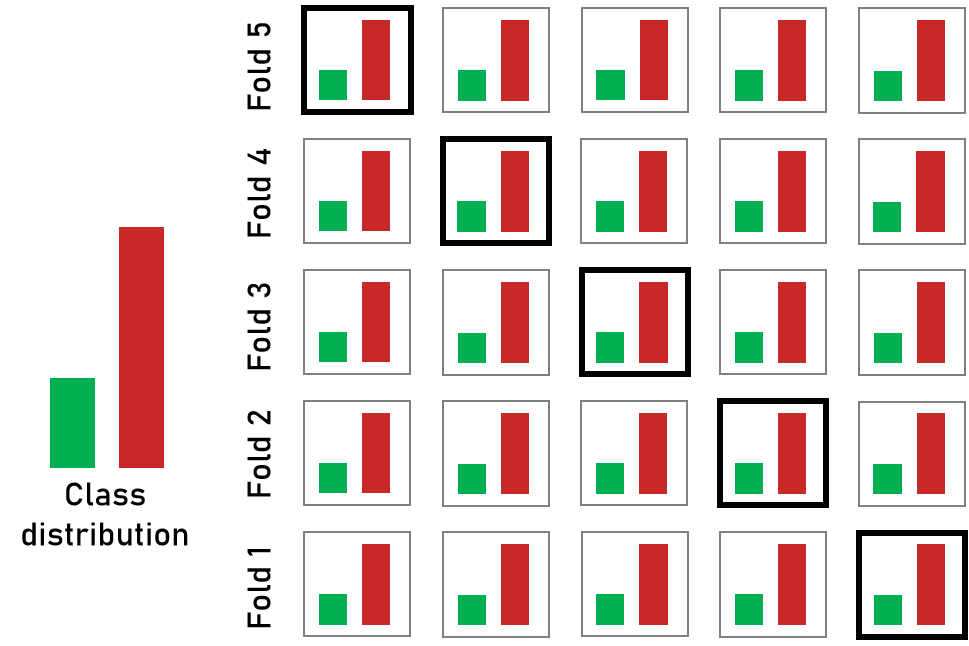
Normally, in the machine learning process, we divide the data into **train, test and validation sets** to evaluate the model’s performance. However, the test and validation sets may cause **variance** to increase when the performance of a particular test set might differ from that of another test set.

Also, this **hold-out approach (train-test-val)** is better when you have enough data points in both the classes. But when the data is imbalanced or less, it is better to use **K-Fold Cross Validation** for evaluating the performance when the data set is randomly split into ‘k’ groups. Out of these groups, one will be used as the **test set** and the rest of the groups will be used as **train sets.** To evaluate the performance, the model will be trained on **k-1 groups** and then scored using the test set. This process will be iterated until each unique group has been used as the test set.



**K-Fold Cross-Validation**

An extension of K-Fold Cross Validation is **Stratified K-Fold Cross Validation**, in which we rearrange the data to ensure that each fold is a good representative of all the strata of the data. For **imbalanced data**, such as the one we will be focusing on in this Capstone, it is important that the class distribution in each fold is the same as that in the original data set. **Stratification** ensures that each fold is representative of all the strata of the data.



**Stratified K-Fold cross-validation**

**Grid Search** can be thought of as an exhaustive search of hyperparameters for selecting the ideal hyperparameters for a model.

We will set up a **grid of the hyperparameter values**, and for each parameter combination, we will train a model and get a score on the test data. From the hyperparameter value obtained, we will select a nearby range on which the model might perform well. Next, we will look at more samples within that range to find the best value within that grid. This is an iterative process, which will continue until we obtain the exact value at which the model is performing the best.

**Randomized search CV** is similar to grid search CV but randomly takes samples of parameter combinations from all possible parameter combinations.

When you have a small data set, the computation time will be manageable to test out different hyperparameter combinations. In this scenario, it is advised to use a grid search.

However, with large data sets, high dimensions will require a prolonged computation time to train and test each combination. In this scenario, it is advised to use a randomized search because the sampling will be random and not uniform.

**For hyperparameter tuning, random and grid search** are the two methods available in **scikit-learn** in the form **of RandomiszedSearchCV and GridSearchCV,** respectively**.**