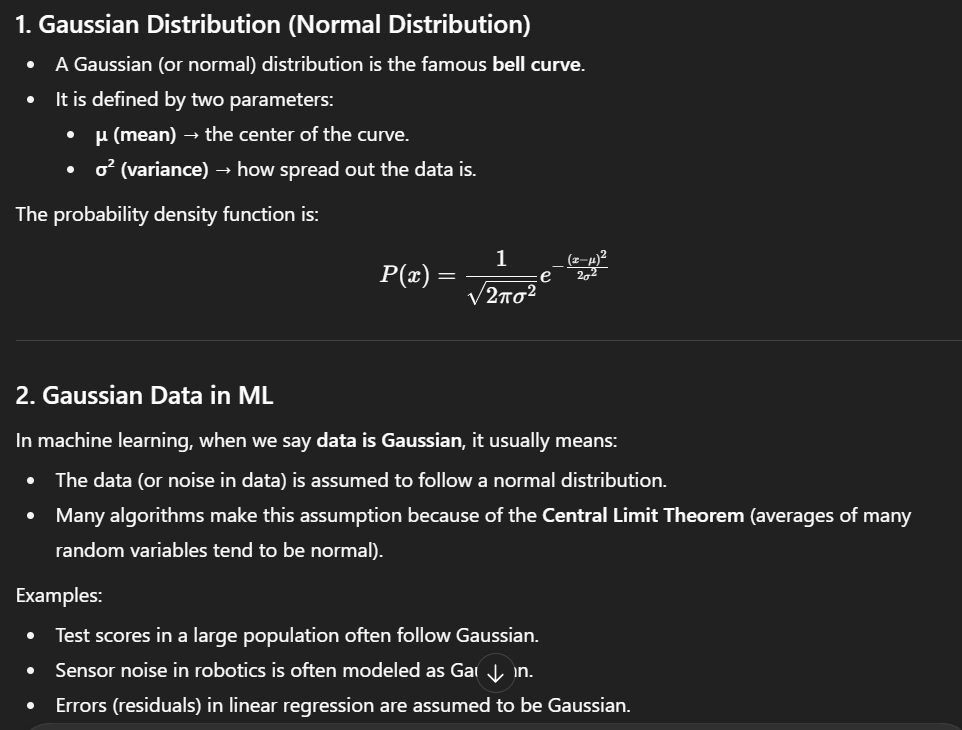
As you saw, the data set includes credit card transactions made by European cardholders over 2 days in September 2013. Out of a total of**284,807 transactions, 492 were fraudulent.**This data set is highly unbalanced, **with the positive class (frauds) accounting for only 0.172% of the total transactions.** 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 labeling**, and it takes the value of 1 in the cases of fraud and 0 in others.



**Note**: The fold Snehanshu specified in the video (at 02:49) should be 10. When k = 10, you would have 50 data points in each test fold (if the total number of target points is 500).

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):** Usually, 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 need not 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.  
  
Now, let’s discuss why skewness may be an issue while modeling. Some of the data points in a skewed distribution toward the tail may act as outliers for the machine learning models that are sensitive to outliers; hence, this 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.

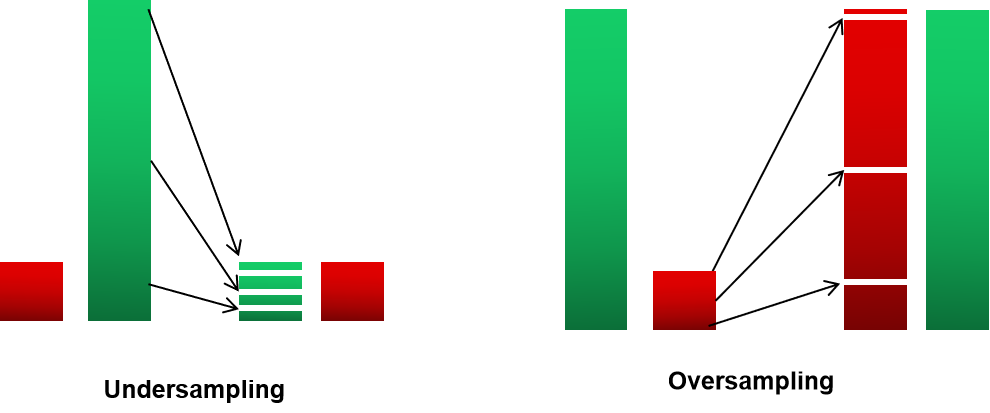
●     **Train/Test split:** Now, you are familiar with the train/test split that you can perform to assess the performance of your models with unseen data. Here, for validation, you can use the k-fold cross-validation method. You need to choose an appropriate k value so that the minority class is correctly represented in the test folds.

●     **Model building / hyperparameter tuning:** This is the final step where you can try different models and fine-tune their hyperparameters until you get the desired level of performance.  
  
As you observed, the data shows a high class imbalance. Over 200,000 cases are mapped to 0, but hardly 500 cases are mapped to 1. Any machine learning algorithm would work well when there is an equal representation of each of the classes. However, in this case, regardless of the model you build, the underlying algorithm will learn more about non-fraudulent cases rather than fraudulent ones. Therefore, the loss function optimization will be heavily biased toward the former type of data. This is called the**minority class problem.**

You can use certain methods to mitigate this problem. They are as follows:

●     **Undersampling**:In this method, you can select fewer data points from the majority class for your model building process. If you have only 500 data points in the minority class, you will 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.

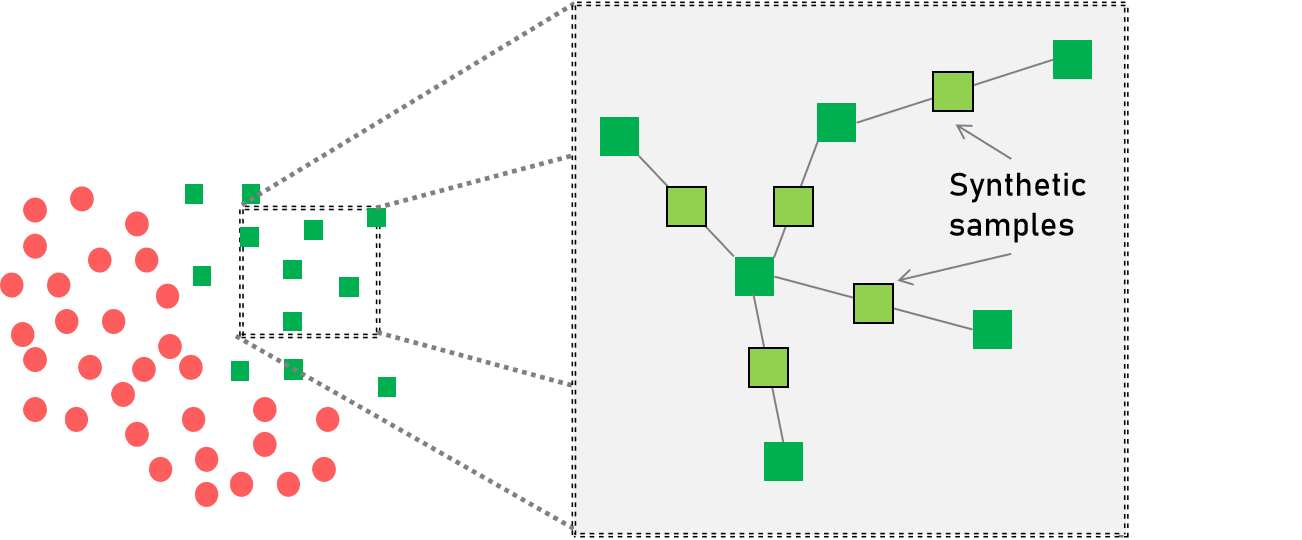
●     **Oversampling**: Using this method, you can assign weights to randomly chosendata points from the minority class. In this way, the occurrence of each data point will be multiplied by the assigned weight, and the machine learning algorithm will now be able to focus on this class while optimizing the loss function. However, this method does not add any new information and may even exaggerate the existing information considerably.



Undersampling and Oversampling

**Synthetic Minority Over-Sampling TEchnique (SMOTE)**: In this process, you can generate new data points that 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 neighbors to create random synthetic samples. The steps in this process are as follows:

* A minority point A is randomly selected.
* The k nearest neighbors for that data point belonging to the same group are found, and then, a random point B from k\_neighbours is selected.
* A random value in the range [0, 1] is specified as λ.
* A synthetic sample is generated and placed between the two points A and B on the vector located at λ% from the original point A.



**New synthetic sample points are added 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:

* It lowers the bias introduced by the class imbalance.
* It adaptively shifts the classification decision boundary toward difficult examples.

**from** **imblearn.over\_sampling** **import** SMOTE, ADASYN

X\_smote, y\_amote = SMOTE().fit\_resample(X, y)

X\_ada, y\_ada = ADASYN().fit\_resample(X, y)

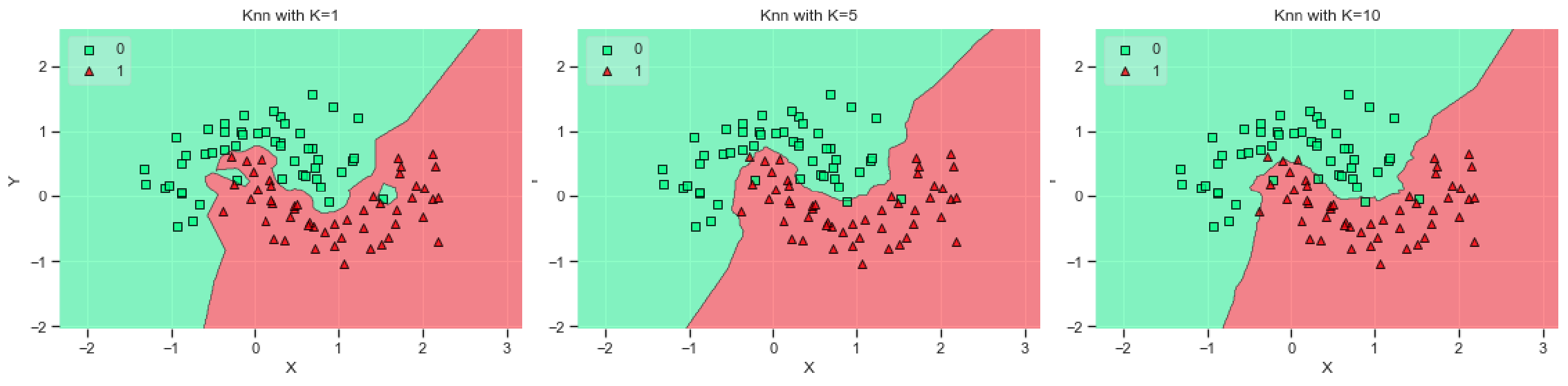
**Introduction to KNN**

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

The **k** in KNN specifies the **number of neighbors** that the algorithm should focus on. Suppose k = 3. Then, for a particular test data, the algorithm observes the three nearest neighbors 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. The k in k-means stands for the number of clusters that can have any number of data points, whereas the k in KNN stands for the number of points that the model would consider to make predictions.

What should the ideal k value be? And what happens if you change it? You will be able to answer these with the help of the diagrams included in the text below that show the separation boundary between two 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.

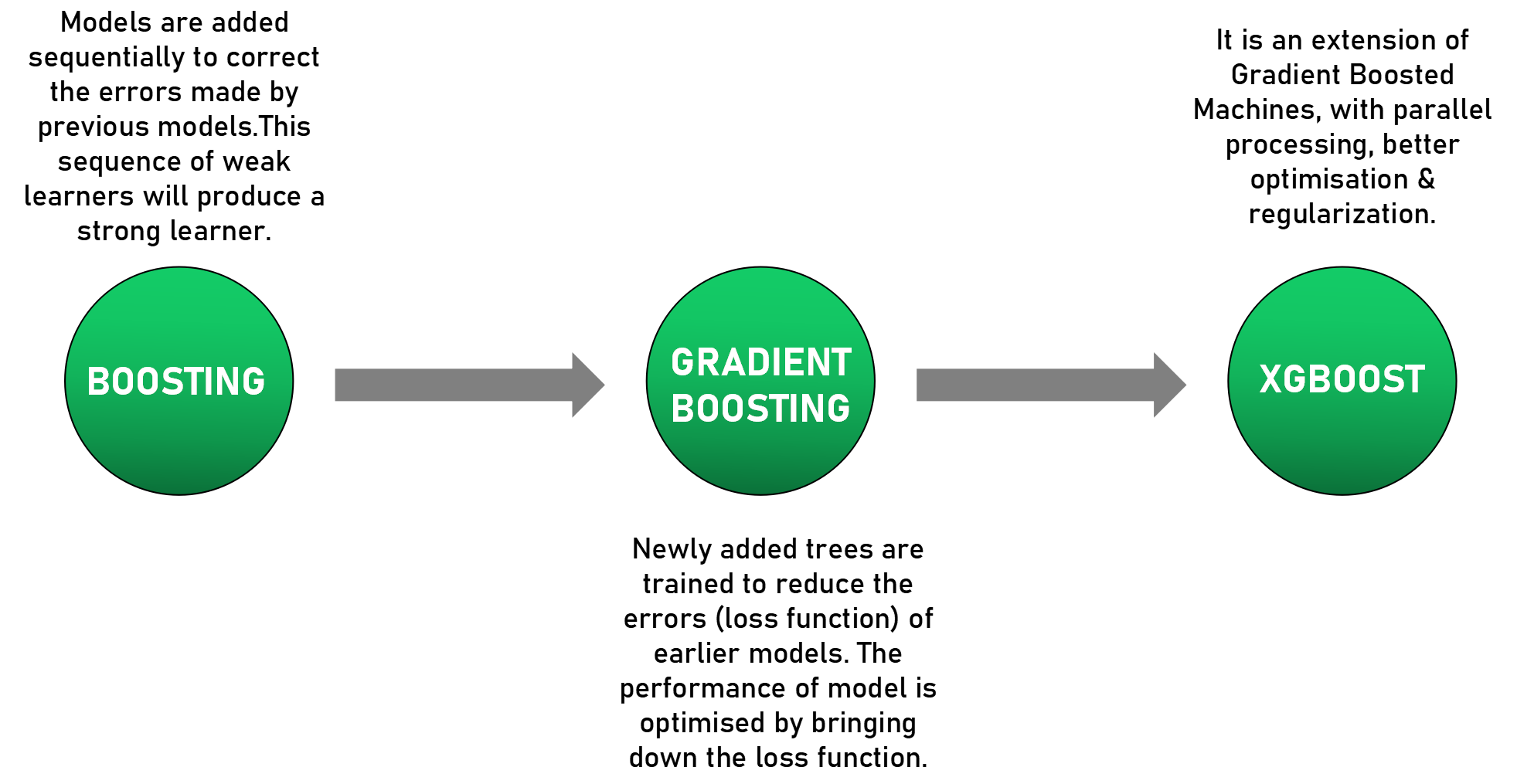
●     When**k = 1**for a data point, the model observes the **immediate neighbor**,i.e., it understands the noise as well, which causes it to overfit the data.

●     When **k = 5**, the model observes the **five nearest neighbors** for a data point. Therefore, the decision boundary starts becoming smooth.

●     When **k = 10**, the model observes the **10 nearest neighbors** 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 in terms of structured and unstructured data. For structured or tabular data, XGBoost, since its inception, 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 discuss what we have covered so far:

●     **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 achieves this by adding more weight to cases with incorrect predictions. By following this approach, the ensemble model will correct itself while learning by focusing on cases/data points that are hard to predict correctly.

●     We discussed the concept of **gradient boosting**. Previously, you learned about gradient descent. The same principle applies here as well, wherein the newly added trees are trained to **reduce the errors (loss function)** made by the earlier models. Overall, in gradient boosting, you are optimizing the performance of the boosted model by bringing down the loss function.

●     **XGBoost** is an extended version of**gradient boosting**, wherein it uses more accurate approximations to tune the model and find the best fit. The added features in this are listed below:

* **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 using the second-order derivative instead of the first-order derivative.
* **Advanced regularization (Lasso and Ridge** **)** to penalize the model based on the number of trees and depth of the model: Higher the number of trees, the 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.

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

**🌱 Gradient Boosting (normal version)**

* Think of it as building a model step by step, adding **small decision trees** (weak learners) one after another.
* Each new tree tries to fix the mistakes made by the previous trees.

**⚡ What makes XGBoost “on steroids”**

1. **Second-order derivatives (curvature information)**
   * Normal gradient boosting just looks at the *slope* (first derivative) of the error to decide how to improve.
   * XGBoost looks at both the *slope* **and** the *curvature* (second derivative).
   * This gives a clearer picture of where the “valley” (best solution) is, so it can converge (find the best fit) faster and more accurately.

👉 Like walking downhill:

* + Gradient boosting knows the **direction** (down).
  + XGBoost also knows how **steep or curved** the slope is, so it avoids zigzagging and reaches the bottom quicker.

1. **Regularization (penalizing complexity)**
   * If the model makes trees that are **too deep** or **too many**, it can overfit (memorize instead of generalize).
   * XGBoost uses Lasso (L1) and Ridge (L2) penalties to keep the trees **simpler**.
   * Before splitting a node, it checks: *“Will this split really reduce error by enough?”*
     + If **yes** → make the split.
     + If **no** → skip it, keep the tree smaller.

👉 This means XGBoost makes **leaner, smarter trees** instead of overcomplicated ones.

1. **Parallel & distributed computing**
   * Gradient boosting usually builds trees **one after another**, which can be slow.
   * XGBoost speeds this up by using **multi-core processors** (parallelism) and even running on **clusters** (distributed).
   * Result → much **faster training**, especially with big datasets.

**💡 In short**

* **Normal Gradient Boosting** = good, but can be slow and overfit easily.
* **XGBoost** = same idea, but with:  
  ✅ Smarter math (uses slope + curvature)  
  ✅ Built-in control to prevent overfitting  
  ✅ Much faster because it uses modern CPUs/GPUs efficiently

👉 That’s why people call it **“gradient boosting on steroids”**.

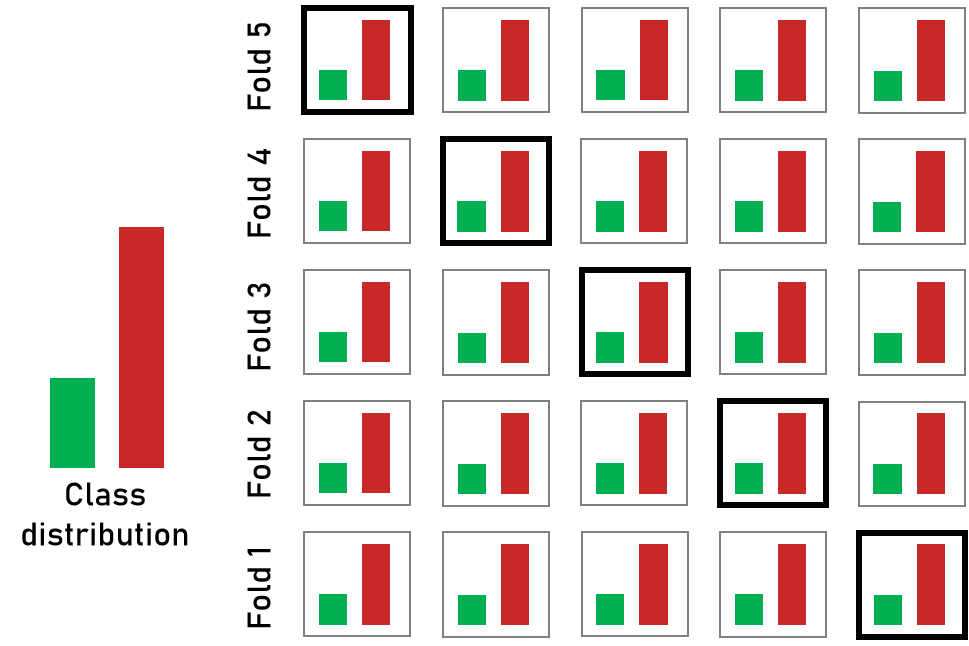
* **logistic regression** works best when data is **linearly separable** and **interpretable**. However, the biggest problem occurs mostly when the data has a good amount of **overlap between the classes**present in it.
* As in logistic regression, **KNN** is highly interpretable. However, there is an issue while computing the neighbors of a particular test point. You need to find the distance between all the data points and the test data point to come up with the classification of a single neighbor. This task will require a lot of computation and is not a good choice when you have a **large amount of data**.
* The **decision tree model** is the first choice when you want the output to be **intuitive** and want to explain the results to individuals without a technical background and other stakeholders. You can find the exact node at which a point has been classified and the reason for it. However, when the tree fits all the samples in the training data perfectly,**overfitting** occurs. **Decision trees** check data in many ways. Hence,**decision trees tend to overfit if left unchecked**.
* 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, **neural networks / LSTM** should be preferred to extract features from the unstructured features.
* In the situation described above, apart from the insight that **pixelated values** are highly correlated with each other, you do not have other features available. Since most of the supervised algorithms focus on these features, you will have to use models that can extract features from the present unstructured data.

**Hyperparameter Tuning**

You also learned about model evaluation in Machine Learning modules, in which we focused on **hyperparameters**and **cross-validation**. You can consider hyperparameters as model controls/settings. Since the ideal settings of a model used for a particular data set will differ from those of models used for other data sets, you need to tune a model every time you use it to obtain the best results. In the video given below, let’s revisit the topics covered and understand the reasons for performing hyperparameter tuning.

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

**Important:**   
Also, this **hold-out approach (train-test-val)** should be preferred when you have enough data points in both classes. However, 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 remaining 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.



**Stratified K-Fold cross-validation**

An extension of K-Fold Cross-Validation is **Stratified K-Fold Cross-Validation**, in which you rearrange data to ensure that each fold is a good representative of all the strata of the data. For **imbalanced data**, such as the one that we will focus 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.

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 obtain a score on the test data. From the hyperparameter value obtained, we will select a nearby range on which the model might perform well.

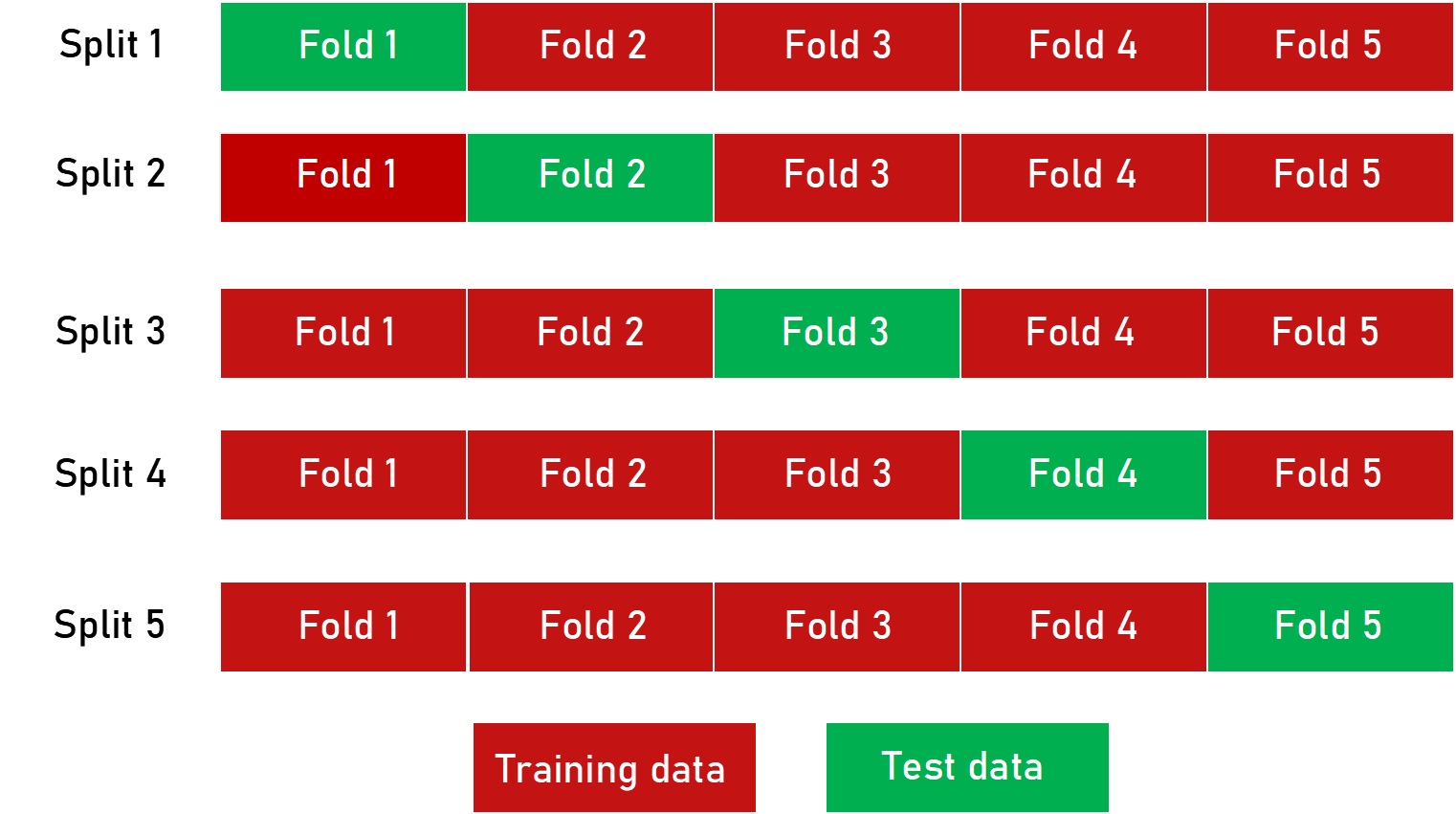
Next, you will take a look at more samples within that range to find the best value within that grid. This is an iterative process, which will continue until we obtain the exact value at which the model is performing the best.

Randomized search CV is similar to GridSearchCV 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 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 RandomizedSearchCV and GridSearchCV, respectively.



**K-Fold Cross-Validation.**

**Model evaluation:**  accuracy is not always the correct metric for solving classification problems. There are other metrics such as **precision, recall, confusion matrix, F1 score**and **the AUC-ROC score**.

the default threshold of 0.5 is not always the ideal threshold to find the best classification label of the test point. Since the ROC curve is measured at all thresholds, the best threshold would be one at which the **TPR is high and FPR is low,** **i.e., misclassifications are low.**

After determining the optimal threshold, you can calculate the F1 score of this classifier to measure the precision and recall at the selected threshold.

Finding the best F1 score is not the last step. This score depends on both **precision and recall.**So, depending on the use case, you have to account for what you need–high precision or high recall.

You learnt how the model was a bad classifier when it labelled every transaction as non-fraudulent with a high accuracy. But what if the scenario is the opposite?

If the model labels all data points as fraudulent, then your recall becomes 1.0. However, at the same time, your precision is compromised. **Precision** is the ability of a classification model to identify only the relevant data points. When you **increase the recall, you will also decrease the precision.**

You can maximise the recall or precision at the expense of another metric, which depends on whether you want a high precision or a high recall for your use case.

**For banks with a smaller average transaction value, you would want a high precision**because you only want to label relevant transactions as fraudulent. For every transaction that is flagged as fraudulent, you can add the human element to verify whether the transaction was made by calling the customer. However, when the precision is low such tasks are a burden because the human element has to be increased.

**For banks having a larger transaction value**, if the recall is low, i.e., it is unable to detect transactions that are labelled as non-fraudulent. So, consider the losses if the missed transaction was a high-value fraudulent one, for example, a transaction of $10,000?

Here, to save banks from high-value fraudulent transactions, **we need to focus on a high recall to detect actual fraudulent transactions.**

Now, you are ready to build your model for this capstone. In the next session, we will explain the problem statement, deliverables, deadlines, etc. of the project.

**🔹 First, what do Precision and Recall mean?**

* **Precision** = *Of all the transactions the model said are fraud, how many were truly fraud?*
* **Recall** = *Of all the actual fraud transactions, how many did the model catch?*

Think of it like:

* **Precision = how careful you are** (avoiding false alarms).
* **Recall = how complete you are** (catching as many frauds as possible).

**🔹 Case 1: Small-value transactions (need high precision)**

Imagine you’re at a bank where most fraudulent transactions are **$5–$20**.

* If the model keeps flagging **too many normal transactions** as fraud (low precision), a human agent has to **call customers every time** to confirm.
* That’s expensive, annoying, and unnecessary for such small amounts.

👉 So here, the bank prefers **high precision**:

* Only raise alarms when it’s *very likely* to be fraud.
* Even if you miss a few frauds (slightly lower recall), the financial loss is small.

**🔹 Case 2: Large-value transactions (need high recall)**

Now think of a bank where fraud happens on **big amounts, like $10,000**.

* If the model **misses just one fraud** (low recall), the bank could lose $10,000.
* That’s way worse than mistakenly calling a few extra customers.

👉 So here, the bank prefers **high recall**:

* Catch *every possible fraud*, even if some are false alarms.
* Precision may drop, but losing a $10,000 fraud is too risky.

✅ **Easy takeaway**:

* **High Precision = fewer false alarms → good for small-value fraud.**
* **High Recall = catch all frauds (even with some false alarms) → good for high-value fraud.**

**Summary**

You have come a long way! You have learnt different techniques in machine learning using which you should now be able to build your own model for credit card fraud detection.

The most important points to re-iterate are as follows:

**Class imbalances:**

* In **undersampling**, you select fewer data points from the majority class for your model building process to balance both classes.
* In **oversampling**, you **assign weights to randomly chosen data points** from the minority class. This is done so that the algorithm can focus on this class while optimising the loss function.
* **SMOTE**is a process using which you can **generate new data points**that lie vectorially between two data points that belong to the minority class.
* **ADASYN**is similar to SMOTE, with a minor change in the sense that the number of synthetic samples that it will add will have a**density distribution**. The aim here is to create synthetic data for minority examples that are harder to learn rather than the easier ones.

**Model selection and understanding:**

* **Logistic regression** works best when the data is **linearly separable** and **needs to be interpretable.**
* **KNN** is also highly interpretable but not preferred when you have a huge amount of data, as **it will consume a lot of computation.**
* **The decision tree model** is the first choice when you want the output to be **intuitive,**but they tend to overfit if left unchecked.
* **KNN**is a simple, **supervised machine learning algorithm** that is used for both classification and regression tasks. The k value in KNN should be an **odd number** because you have to take the majority vote from the nearest neighbours by breaking the ties.
* In **Gradient Boosted machines/trees**, newly added trees are trained to **reduce the errors (loss function)** of earlier models.
* **XGBoost** is an extended version of**gradient boosting**, with additional features such as regularisation and parallel tree learning algorithm for finding the best split.

**Hyperparameter tuning:**

* 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.
* **Stratified K-Fold Cross Validation**is an extension of K-Fold cross-validation,in which you rearrange the data to ensure that each fold is a good representative of all the strata of the data.
* 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, it is advised to use a randomised search because the sampling will be random and not uniform.

**Model evaluation:**

* Accuracy is not always the correct metric for solving classification problems of imbalanced data.
* Since the ROC curve is measured at all thresholds, the best threshold would be one at which the **TPR is high and FPR is low,** **i.e., misclassifications are low.**
* Depending on the use case, you have to account for what you need–high precision or high recall.