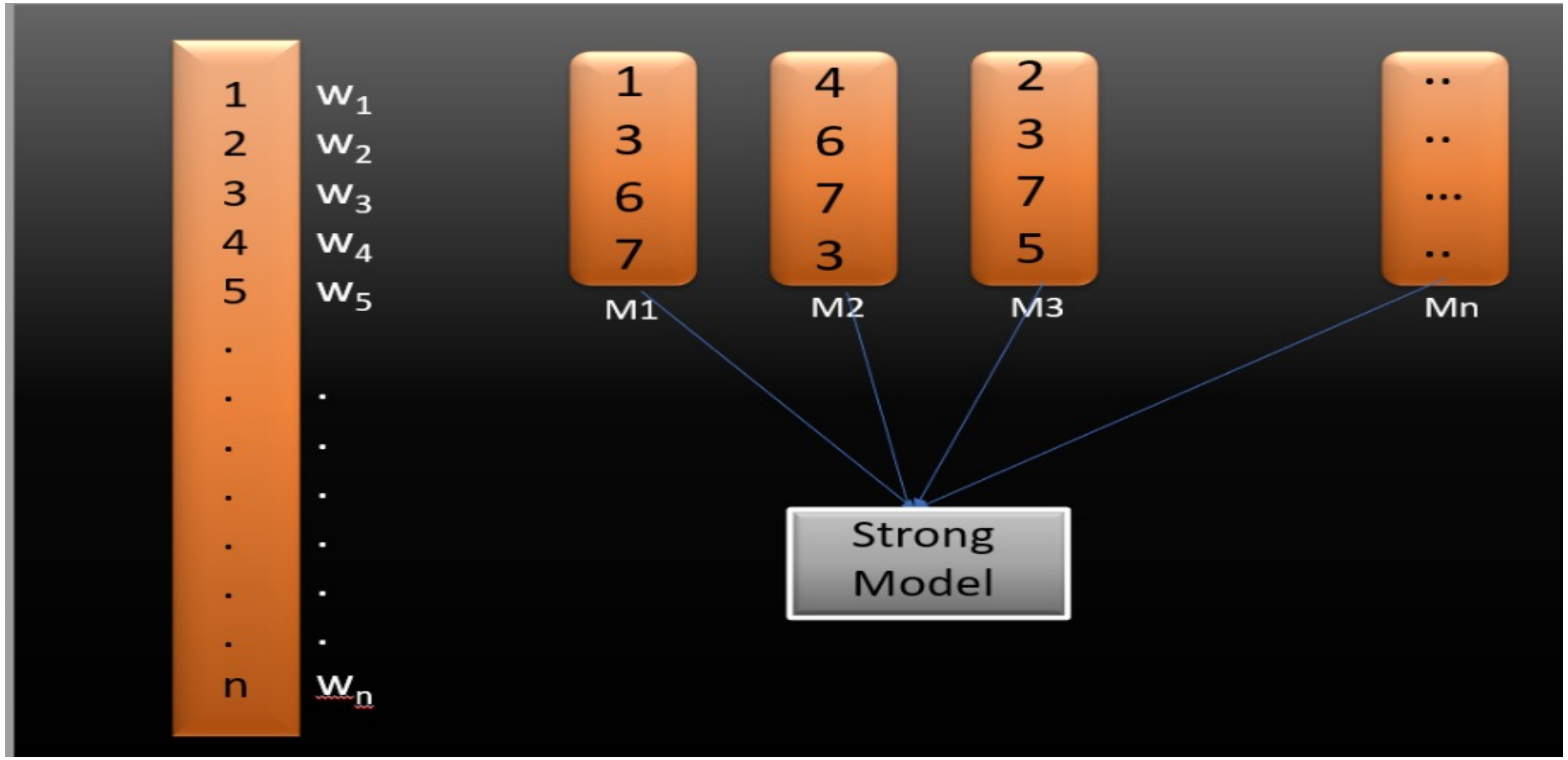
Boosting

Boosting is an ensemble modeling technique that attempts to build a strong learner from the number of weak learners. It is done by building a model by using weak models in series. Boosting combines weak learners (usually decision trees with only one split, called decision stumps) sequentially, so that each new tree corrects the errors of the previous one.Firstly, a model is built from the training data. Then the second model is built which tries to correct the errors present in the first model. This procedure is continued, and models are added until either the complete training data set is predicted correctly, or the maximum number of models are added.


1 2 3 4 5 . . . . n
w_1 w_2 w_3 w_4 w_5 . . . . . . w_n
1 3 6 7
M1
4 6 7 3
M2
2 3 7 5
M3
Mn
Strong Model

Boosting

- Suppose you have n data points and 2 output classes (0 and 1). You want to create a model to detect the class of the test data. Now what we do is randomly select observations from the training dataset and feed them to model 1 (M1), we also assume that initially, all the observations have an equal weight that means an equal probability of getting selected.
- Remember in ensembling techniques the weak learners combine to make a strong model so here M1, M2, M3….Mn all are weak learners.
- Since M1 is a weak learner, it will surely misclassify some of the observations. Now before feeding the observations to M2 what we do is update the weights of the observations which are wrongly classified. In Boosting techniques, when an observation is wrongly classified, its weight gets updated and for those which are correctly classified, their weights get decreased. The probability of selecting a wrongly classified observation gets increased hence in the next model only those observations get selected which were misclassified in model 1.
- Similarly, it happens with M2, the wrongly classified weights are again updated and then fed to M3. This procedure is continued until and unless the errors are minimized, and the dataset is predicted correctly. Now when the new datapoint comes in (Test data) it passes through all the models (weak learners) and the class which gets the highest vote is the output for our test data.

Gradient Boosting

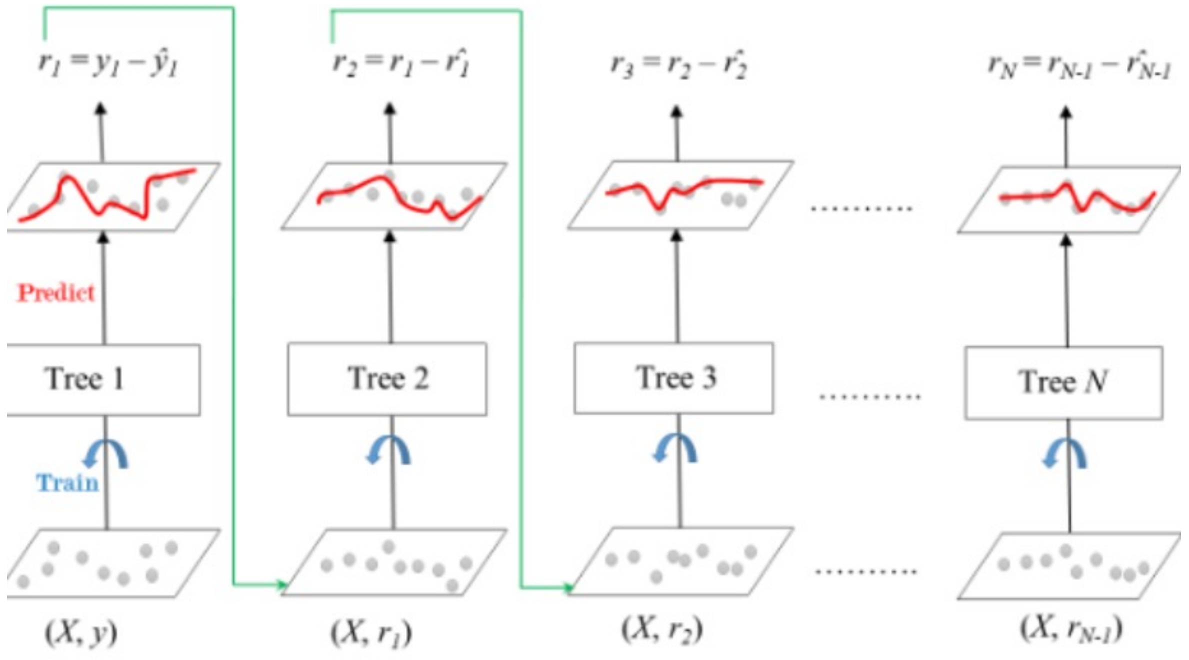
It is a boosting technique that builds a final model from the sum of several weak learning algorithms that were trained on the same dataset. It operates on the idea of stagewise addition.

The first weak learner in the gradient boosting algorithm will not be trained on the dataset; instead, it will simply return the mean of the relevant column.

The residual for the first weak learner algorithm's output will then be calculated and used as the output column or target column for the next weak learning algorithm that will be trained.

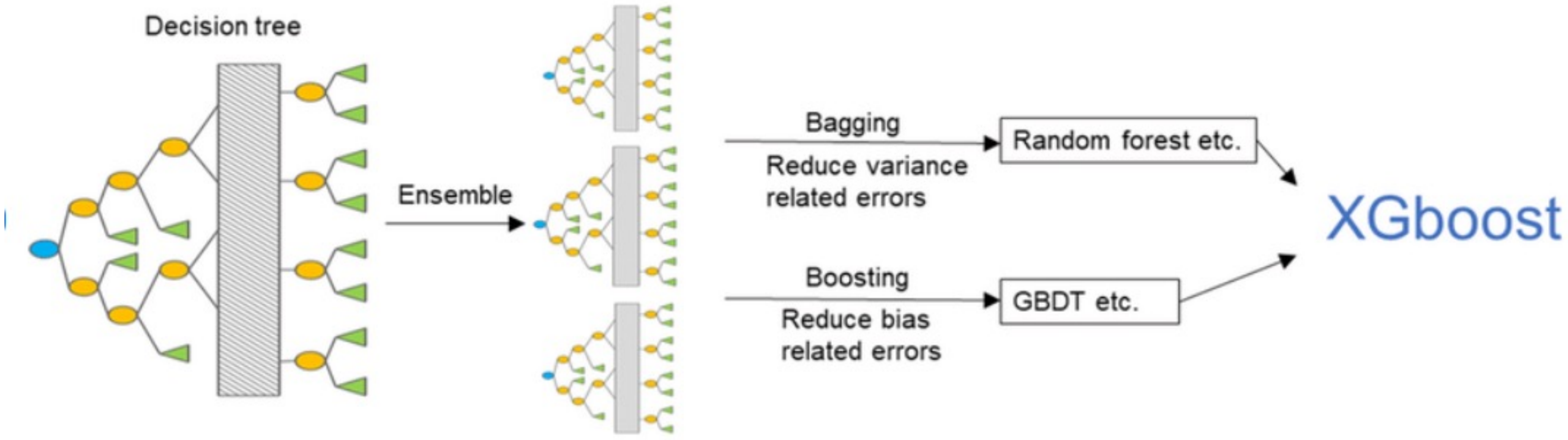
The second weak learner will be trained using the same methodology, and the residuals will be computed and utilized as an output column once more for the third weak learner, and so on until we achieve zero residuals.

The dataset for gradient boosting must be in the form of numerical or categorical data. GBM minimizes a loss function by fitting subsequent models to the previous models' residuals (the differences between the actual values and the predictions). This approach leads to a more robust model with each iteration.



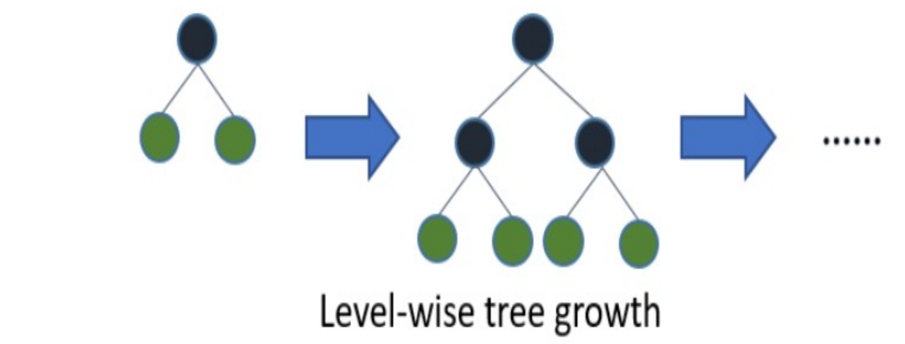
XGBoost- Extreme Gradient Boosting

- Extends gradient boosting with regularization and parallel processing for efficiency.
- Employs techniques like pruning to prevent overfitting.
- XGBoost optimizes the learning process, paying attention to features that contribute the most to reducing errors.
- It may penalize complex models, preventing overfitting and improving generalization.

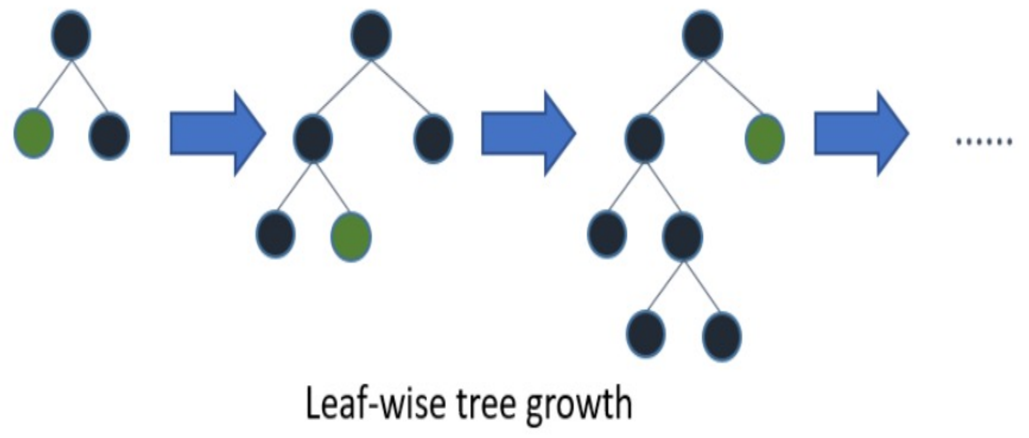


Light GBM

- Light GBM is a fast, distributed, high-performance gradient boosting framework based on decision tree algorithm, used for ranking, classification and many other machine learning tasks.
- Since it is based on decision tree algorithms, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise. So when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms. Also, it is surprisingly very fast, hence the word 'Light'.
- Leaf wise splits lead to increase in complexity and may lead to overfitting and it can be overcome by specifying another parameter max-depth which specifies the depth to which splitting will occur.
- Uses histogram-based learning, which speeds up the training process.
- Splits data into bins and builds histograms for efficient feature selection.
- LightGBM efficiently identifies common patterns in the data, speeding up the learning process.
- It may use histograms to quickly identify the size range where most errors occur and focus on refining predictions in that range.



Level-wise tree growth in XGBOOST.



Leaf wise tree growth in Light GBM.

ADABOOST

AdaBoost specifically focuses on improving the performance of weak learners (individual models that are slightly better than random guessing) by sequentially training them on different subsets of the data and giving more weight to the misclassified samples.

How does the AdaBoost algorithm work?

1. **Initialization**: Each sample in the training dataset is assigned an equal weight initially. These weights determine the importance of each instance during the training process.
2. **Training Weak Learners**: AdaBoost starts by training a weak learner on the training data. A weak learner is typically a simple model like a decision tree with limited depth (a “stump”) or a linear classifier. The weak learner’s performance might be just slightly better than random guessing.
3. **Weighted Error**: After training the weak learner, it’s evaluated on the training data. The samples that the weak learner misclassifies are assigned higher weights, making them more influential in subsequent iterations.
4. **Compute Alpha**: An alpha value is computed based on the weighted error of the weak learner. The alpha value indicates how much trust should be given to the weak learner’s prediction. A smaller weighted error leads to a higher alpha.
5. **Update Weights**: The weights of the misclassified samples are updated, increasing their importance for the next iteration. Correctly classified samples retain their weights or may have their weights reduced.
6. **Normalization of Weights**: The sample weights are then normalized to ensure they sum up to 1. This step prevents the weights from becoming too large over iterations.
7. **Aggregate Predictions**: The weak learner’s prediction is combined with the predictions from previous weak learners, each weighted by its corresponding alpha value. This creates the ensemble prediction.
8. **Repeat**: Steps 2 to 7 are repeated for a specified number of iterations or until a certain level of accuracy is achieved.
9. **Final Prediction**: The final prediction is made by combining the weighted predictions of all the weak learners in the ensemble. The alpha values also contribute to each weak learner’s prediction weight.

ADABOOST

We can split the idea of AdaBoost into 3 big concept :

1. USED STUMP AS WEAK LEARNERS

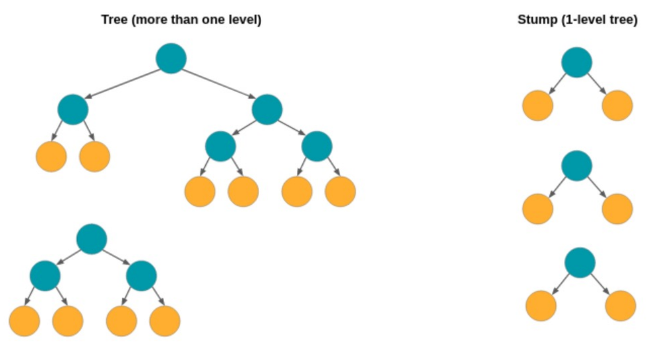
Weak learners is any model that has a accuracy better than random guessing even if it is just slightly better (e.g 0.51). In an Ensemble methods we combines multiple weak learners to make strong learner model. In AdaBoost, weak learners are used, a 1-level decision tree (Stump).The main idea when creating a weak classifier is to find the best stump that can separate data by minimizing overall errors.

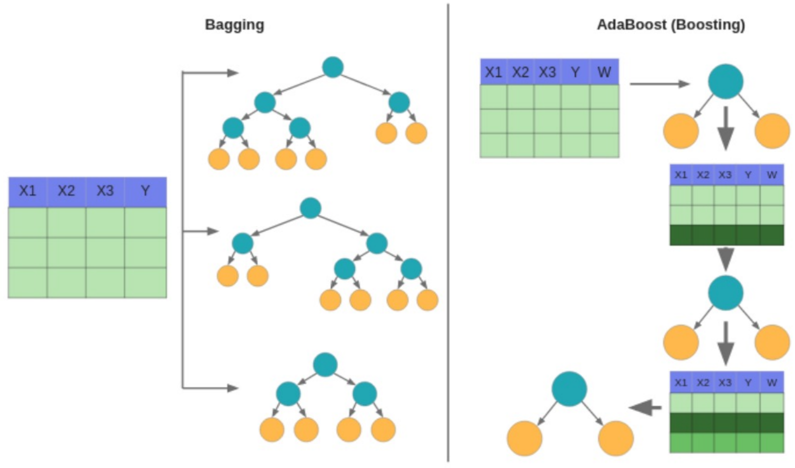
2. INFLUENCE THE NEXT STUMP

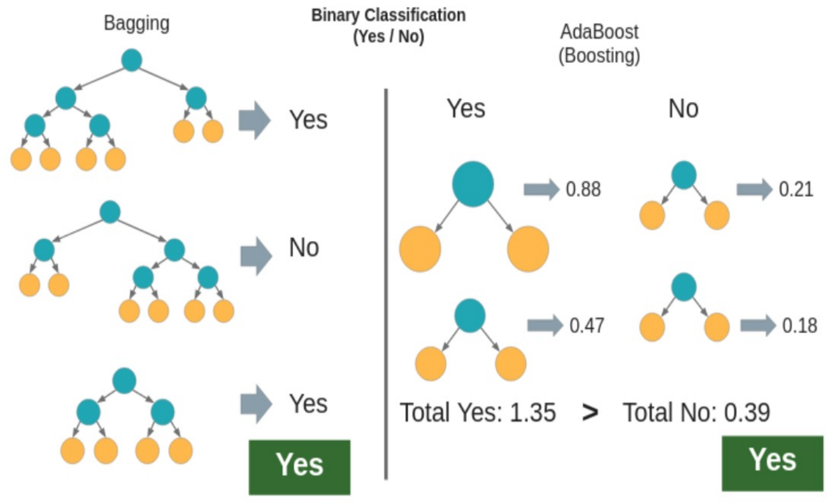
Unlike bagging, which makes models in parallel, Boosting does training sequentially, which means that each stump (weak learner) is affected by the previous stump. The way Stump affects the next stump is by giving different weights to the data that will be used in the next stump making process. This weighting is based on error calculations, if a data is incorrectly predicted in the first stump, then the data will be given a greater weight in the next stump-making process.

3. WEIGHTED VOTE

In AdaBoost algorithm, each stump has a different weight, the weight for each stump is based on the resulting error rate. The smaller errors generated by a stump, the greater the weight of the stump. The weight of each stump is used in the voting process, if the greater the total weight obtained by one of the classes, then that class will be used as the final class.






Bagging
Binary Classification (Yes / No)
AdaBoost (Boosting)
Yes
No
Yes
Yes
Yes
No
0.88
0.21
0.47
0.18
Total Yes: 1.35 > Total No: 0.39
Yes

In Simple Terms

1. Gradient Boosting:
   Idea: Builds trees sequentially, with each tree correcting errors of the previous one.
   Example:
   Suppose you're teaching a student a concept. Each time the student makes a mistake, you focus on explaining that specific mistake in the next iteration.

2. XGBoost (Extreme Gradient Boosting):
   Enhancements over Gradient Boosting:
     - Regularization terms to control overfitting.
     - Parallel processing to improve speed.
   Example:
   Imagine the student learning not only from you but also consulting multiple textbooks simultaneously to get a comprehensive understanding.

3. AdaBoost (Adaptive Boosting):
   Idea: Focuses on the weaknesses of the model by giving more weight to misclassified points.
   Example:
   If a student struggles with a particular topic, you'd spend more time on that topic to ensure they grasp it well.

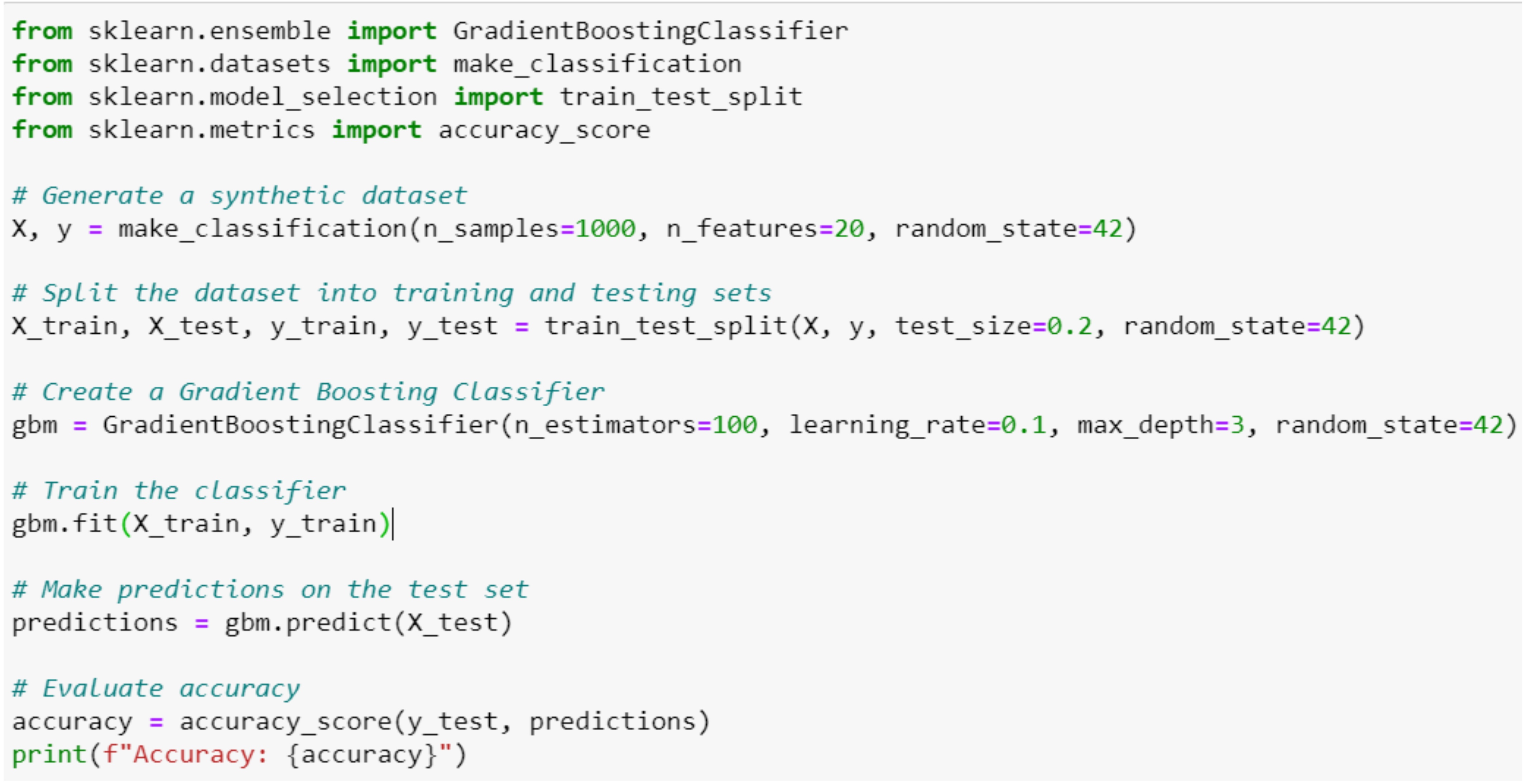
4. LightGBM:
   Key Feature: Gradient boosting framework that uses tree-based learning algorithms but is designed for distributed and efficient training.
   Example:
   Consider a scenario where multiple students are learning together, and they collaborate efficiently to cover a diverse range of topics.

5. CatBoost:
   Key Feature: Gradient boosting library that handles categorical features automatically.
   Example:
   Suppose the students have different learning styles, and you adapt your teaching strategy based on each student's unique preferences.

Analogy:
   If you're trying to climb a hill (optimize a model), these algorithms are like different paths. Gradient Boosting is the straightforward climb, XGBoost is the climb with additional tools and helpers, AdaBoost is adjusting based on where you stumble, LightGBM is the climb with friends who divide tasks efficiently, and CatBoost is a climb where each step is tailored to your unique strengths.

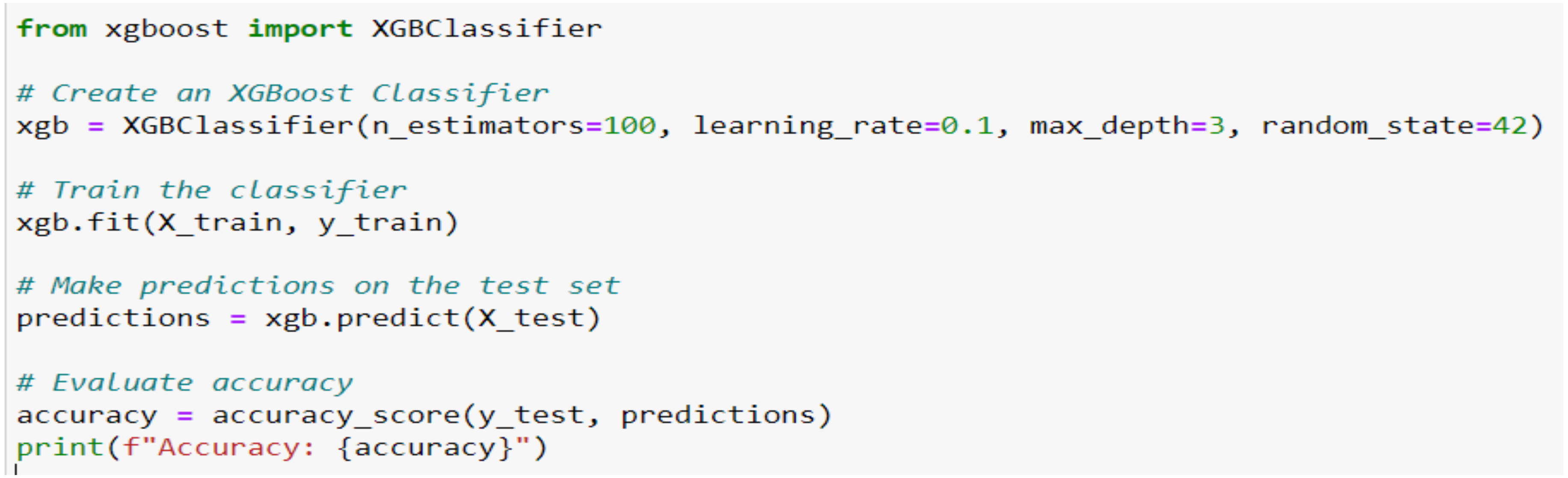
CatBoost – The growth of decision trees inside CatBoost is the primary distinction that sets it apart from and improves upon competitors. The decision trees that are created in CatBoost are symmetric. As there is a unique sort of approach for handling categorical datasets, CatBoost works very well on categorical datasets compared to any other algorithm in the field of machine learning. The categorical features in CatBoost are encoded based on the output columns. As a result, the output column’s weight will be taken into account while training or encoding the categorical features, increasing its accuracy on categorical datasets.

Gradient Boosting



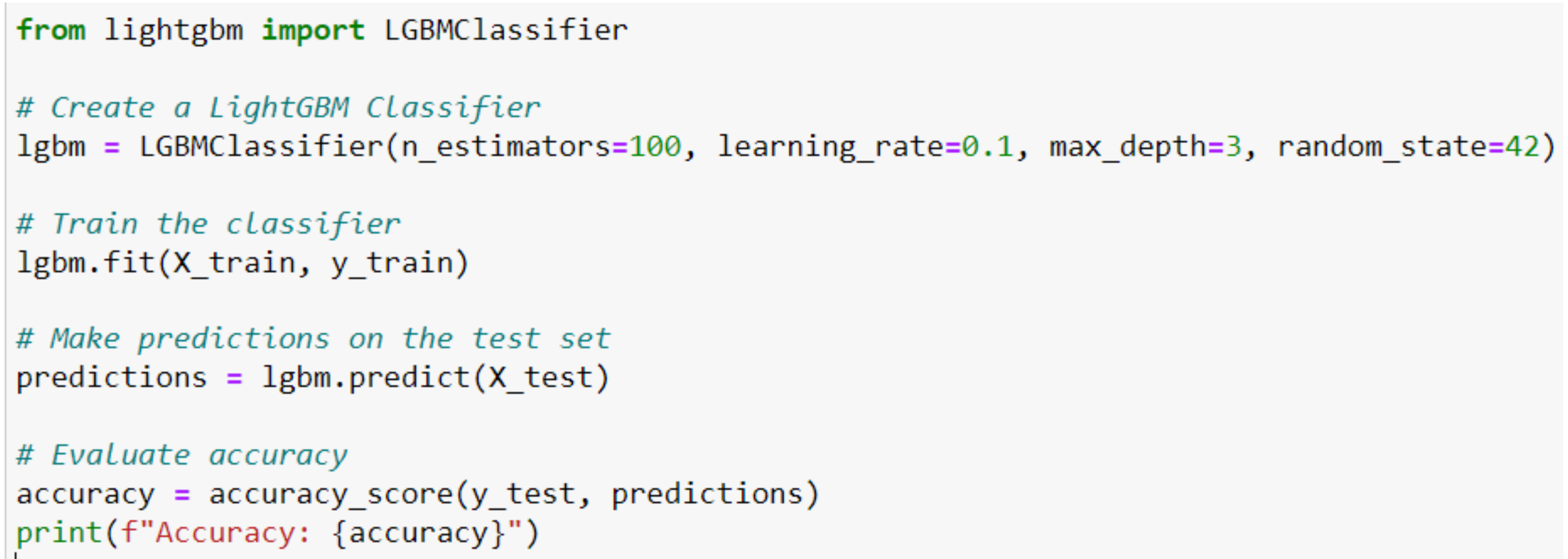
```python
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20, random_state=42)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Gradient Boosting Classifier
gbm = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3, random_state=42)

# Train the classifier
gbm.fit(X_train, y_train)

# Make predictions on the test set
predictions = gbm.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy}")
```

XGBoost



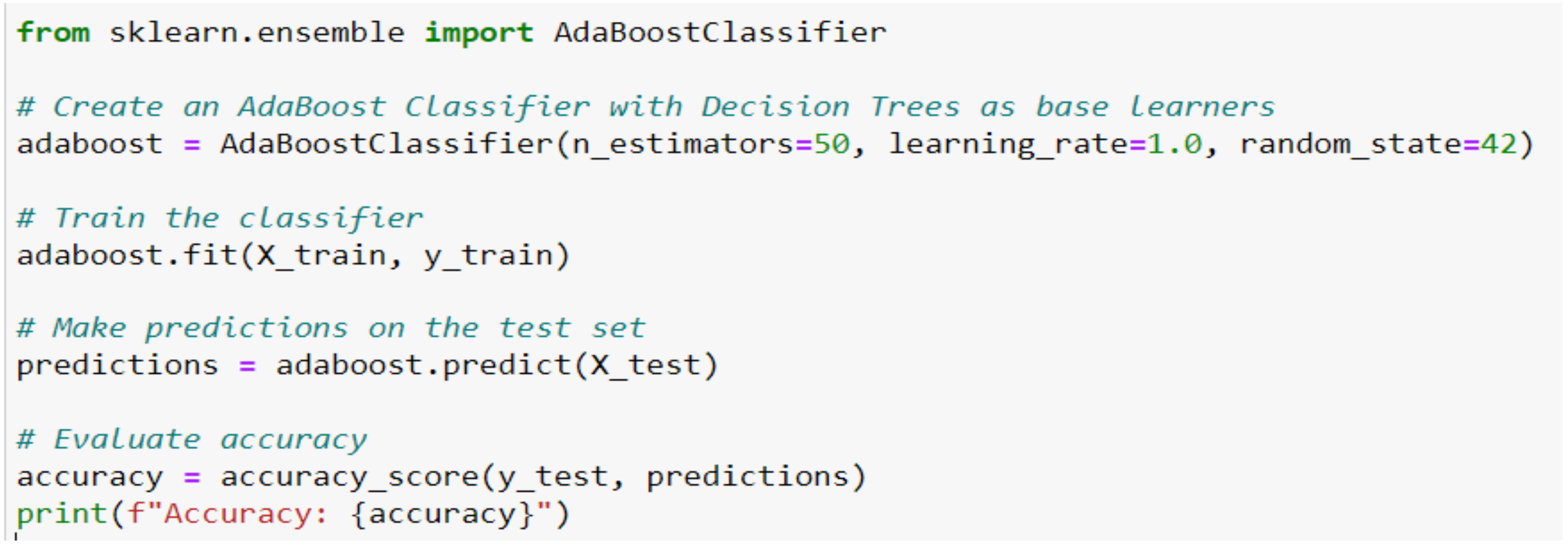
```python
from xgboost import XGBClassifier

# Create an XGBoost Classifier
xgb = XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=3, random_state=42)

# Train the classifier
xgb.fit(X_train, y_train)

# Make predictions on the test set
predictions = xgb.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy}")
```

LIGHT GBM



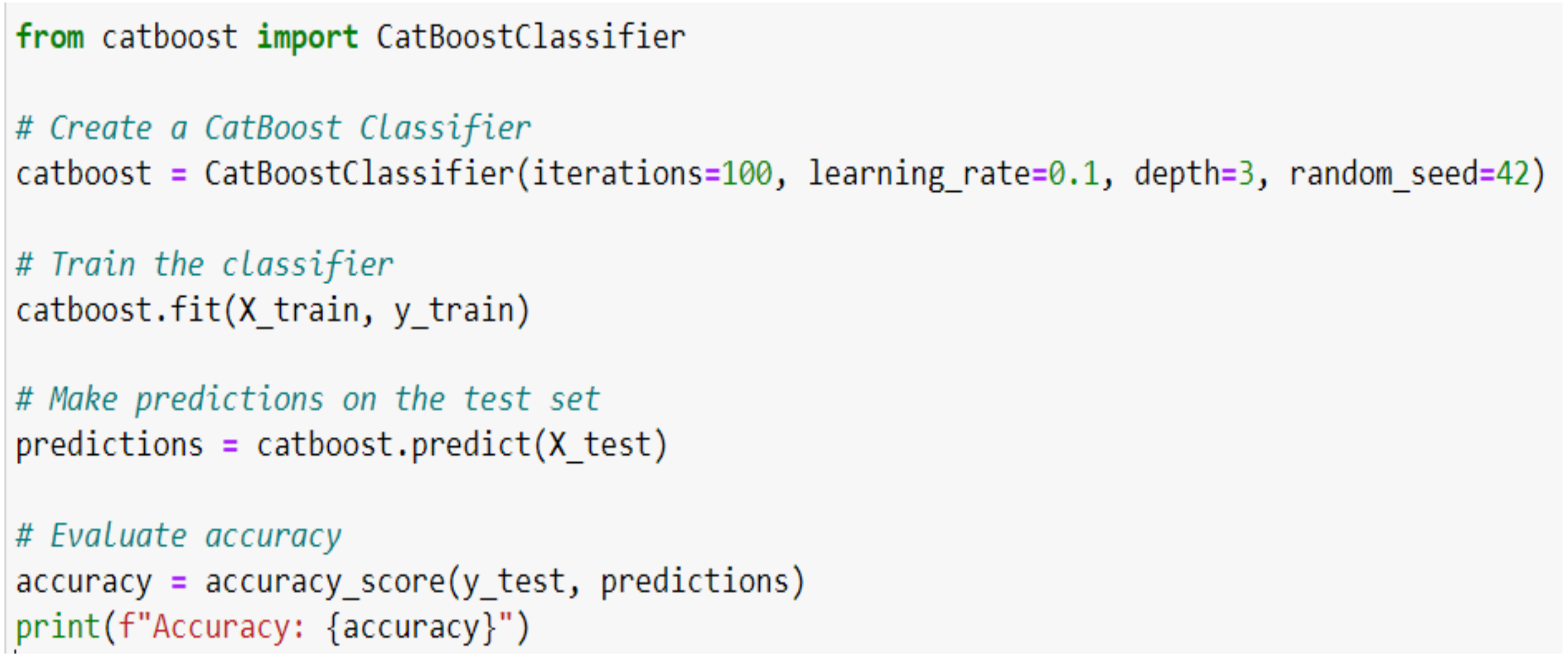
```python
from lightgbm import LGBMClassifier

# Create a LightGBM Classifier
lgbm = LGBMClassifier(n_estimators=100, learning_rate=0.1, max_depth=3, random_state=42)

# Train the classifier
lgbm.fit(X_train, y_train)

# Make predictions on the test set
predictions = lgbm.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy}")
```

ADABOOST



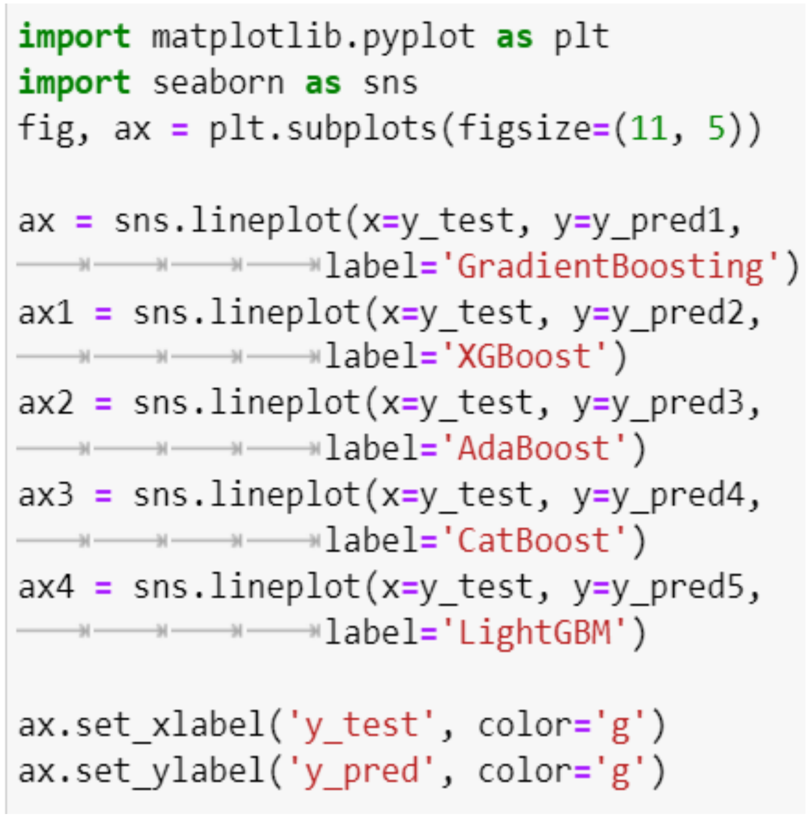
```python
from sklearn.ensemble import AdaBoostClassifier

# Create an AdaBoost Classifier with Decision Trees as base learners
adaboost = AdaBoostClassifier(n_estimators=50, learning_rate=1.0, random_state=42)

# Train the classifier
adaboost.fit(X_train, y_train)

# Make predictions on the test set
predictions = adaboost.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy}")
```

CATBOOST



```python
from catboost import CatBoostClassifier

# Create a CatBoost Classifier
catboost = CatBoostClassifier(iterations=100, learning_rate=0.1, depth=3, random_seed=42)

# Train the classifier
catboost.fit(X_train, y_train)

# Make predictions on the test set
predictions = catboost.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy}")
```

To visualize the performance of all the algorithms on the same data, we can also plot the graph between the y_test and y_pred of all the algorithms.



```python
import matplotlib.pyplot as plt
import seaborn as sns
fig, ax = plt.subplots(figsize=(11, 5))

ax = sns.lineplot(x=y_test, y=y_pred1,
                  label='GradientBoosting')
ax1 = sns.lineplot(x=y_test, y=y_pred2,
                  label='XGBoost')
ax2 = sns.lineplot(x=y_test, y=y_pred3,
                  label='AdaBoost')
ax3 = sns.lineplot(x=y_test, y=y_pred4,
                  label='CatBoost')
ax4 = sns.lineplot(x=y_test, y=y_pred5,
                  label='LightGBM')

ax.set_xlabel('y_test', color='g')
ax.set_ylabel('y_pred', color='g')
```

Extra Trees Model (Extremely Randomized Trees)

- Similar to Random Forests, ExtraTrees is an ensemble ML approach that trains numerous decision trees and aggregates the results from the group of decision trees to output a prediction. However, there are few differences between Extra Trees and Random Forest.
- Random Forest uses **bagging** to select different variations of the training data to ensure decision trees are sufficiently different. However, Extra Trees uses the entire dataset to train decision trees. As such, to ensure sufficient differences between individual decision trees, it RANDOMLY SELECTS the values at which to split a feature and create child nodes. In contrast, in a Random Forest, we use an algorithm to greedy search and select the value at which to split a feature.
- There are three main hyperparameters to tune in the algorithm; they are
    - ➢ The number of decision trees in the ensemble,
    - ➢ The number of input features to randomly select and consider for each split point, and
    - ➢ The minimum number of samples required in a node to create a new split point.
- The random selection of split points makes the decision trees in the ensemble less correlated, although this increases the variance of the algorithm. This increase in variance can be countered by increasing the number of trees used in the ensemble.

Extra Trees vs Random Forest

The two ensembles have a lot in common. Both of them are composed of a large number of decision trees, where the final decision is obtained taking into account the prediction of every tree. Specifically, by majority vote in classification problems, and by the arithmetic mean in regression problems.

The main differences between Extra Trees and Random Forest are the following:

- Random forest uses bootstrap replicas, that is to say, it subsamples the input data with replacement, whereas Extra Trees use the whole original dataset. In the Extra Trees sklearn implementation there is an optional parameter that allows users to bootstrap replicas, but by default, it uses the entire input sample. This may increase variance because bootstrapping makes it more diversified.
- Another difference is the selection of cut points in order to split nodes. Random Forest chooses the optimum split while Extra Trees chooses it randomly. However, once the split points are selected, the two algorithms choose the best one between all the subset of features. Therefore, Extra Trees adds randomization but still has optimization.

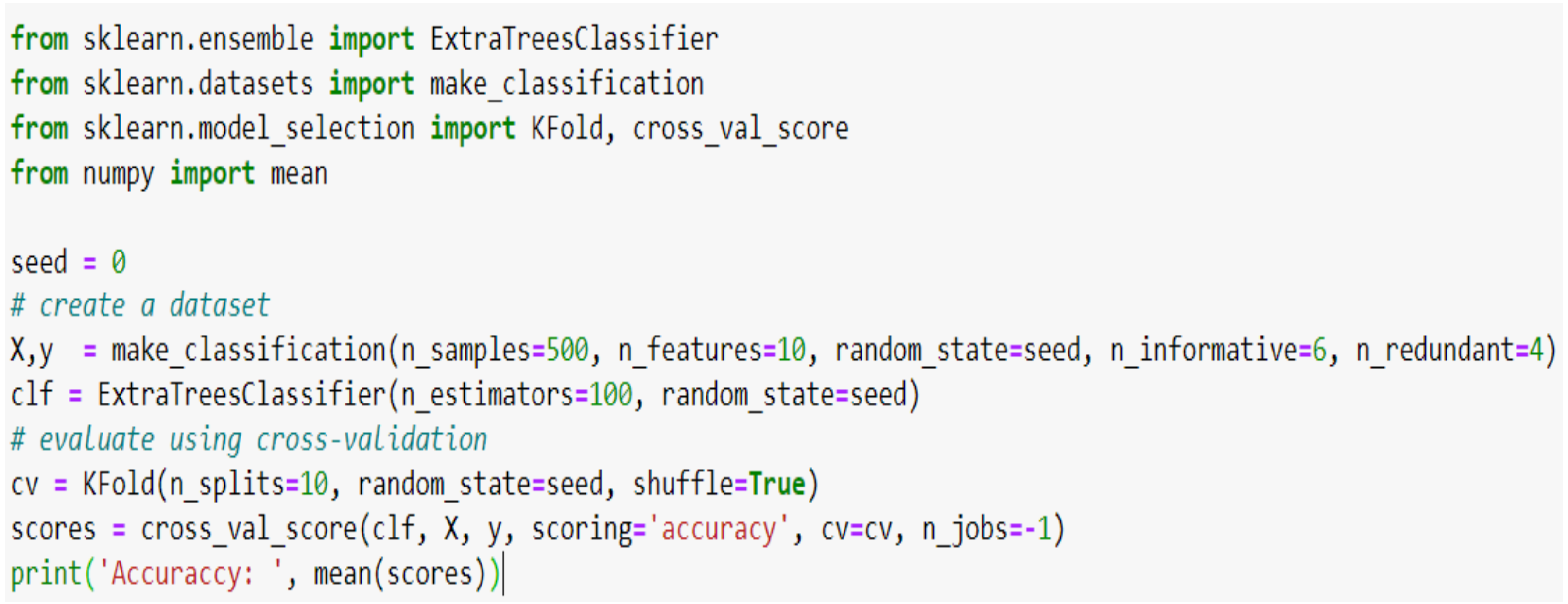
These differences motivate the reduction of both bias and variance. On one hand, using the whole original sample instead of a bootstrap replica will reduce bias. On the other hand, choosing randomly the split point of each node will reduce variance.

In terms of computational cost, and therefore execution time, the Extra Trees algorithm is faster. This algorithm saves time because the whole procedure is the same, but it randomly chooses the split point and does not calculate the optimal one.

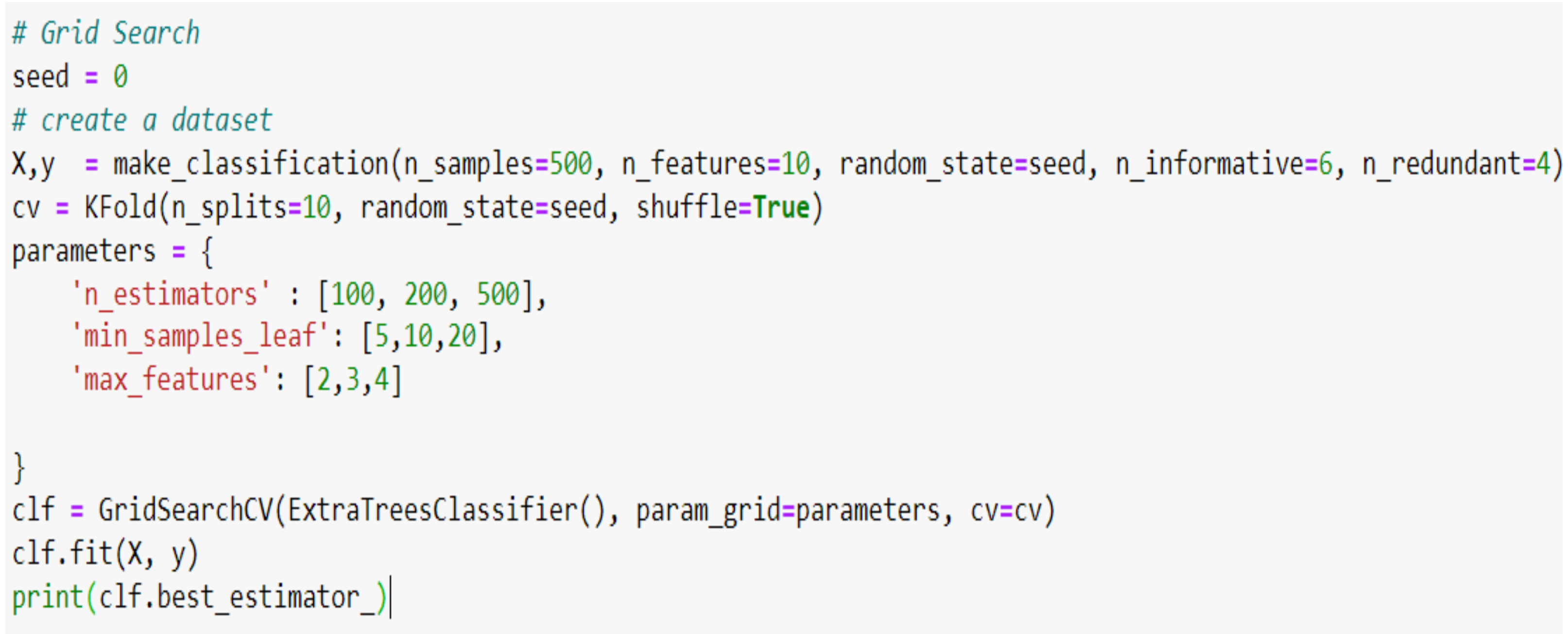
Note--

The paper that introduced the Extra Trees model conducts a bias-variance analysis of different tree based models. **From the paper we see on most classification and regression tasks (six were analyzed) ExtraTrees have higher bias and lower variance than Random Forest.** However, the paper goes on to say this is because the randomization in extra trees works to include irrelevant features into the model. As such, when irrelevant feature were excluded, say via a feature selection pre-modelling step, Extra Trees get a bias score similar to that of Random Forest.

EXTRA TREES MODEL



```python
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import KFold, cross_val_score
from numpy import mean

seed = 0
# create a dataset
X,y  = make_classification(n_samples=500, n_features=10, random_state=seed, n_informative=6, n_redundant=4)
clf = ExtraTreesClassifier(n_estimators=100, random_state=seed)
# evaluate using cross-validation
cv = KFold(n_splits=10, random_state=seed, shuffle=True)
scores = cross_val_score(clf, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
print('Accuraccy: ', mean(scores))
```

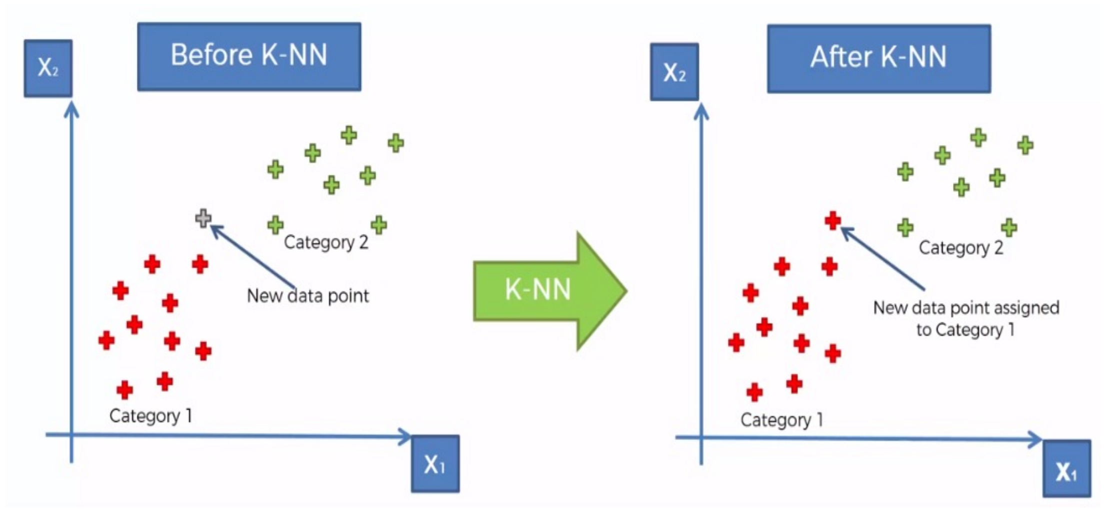


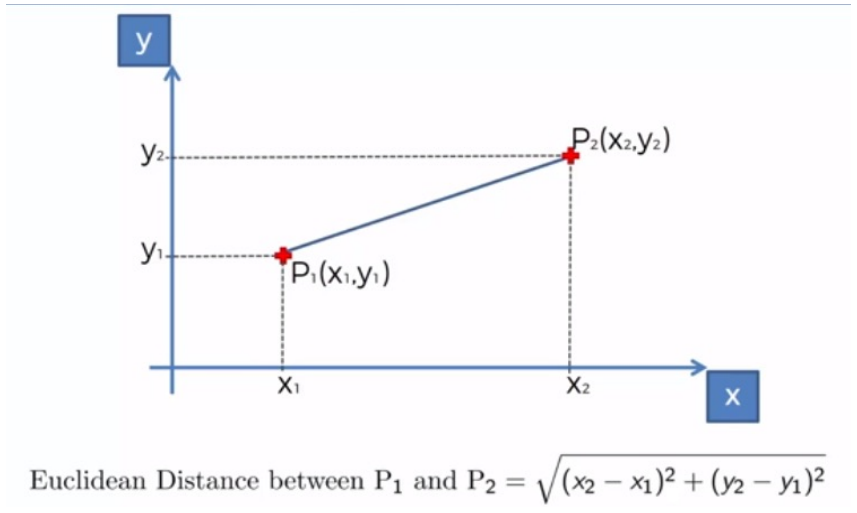
```python
# Grid Search
seed = 0
# create a dataset
X,y  = make_classification(n_samples=500, n_features=10, random_state=seed, n_informative=6, n_redundant=4)
cv = KFold(n_splits=10, random_state=seed, shuffle=True)
parameters = {
    'n_estimators' : [100, 200, 500],
    'min_samples_leaf': [5,10,20],
    'max_features': [2,3,4]

}
clf = GridSearchCV(ExtraTreesClassifier(), param_grid=parameters, cv=cv)
clf.fit(X, y)
print(clf.best_estimator_)
```

Let's look at these parameters more closely from the implementation perspective.

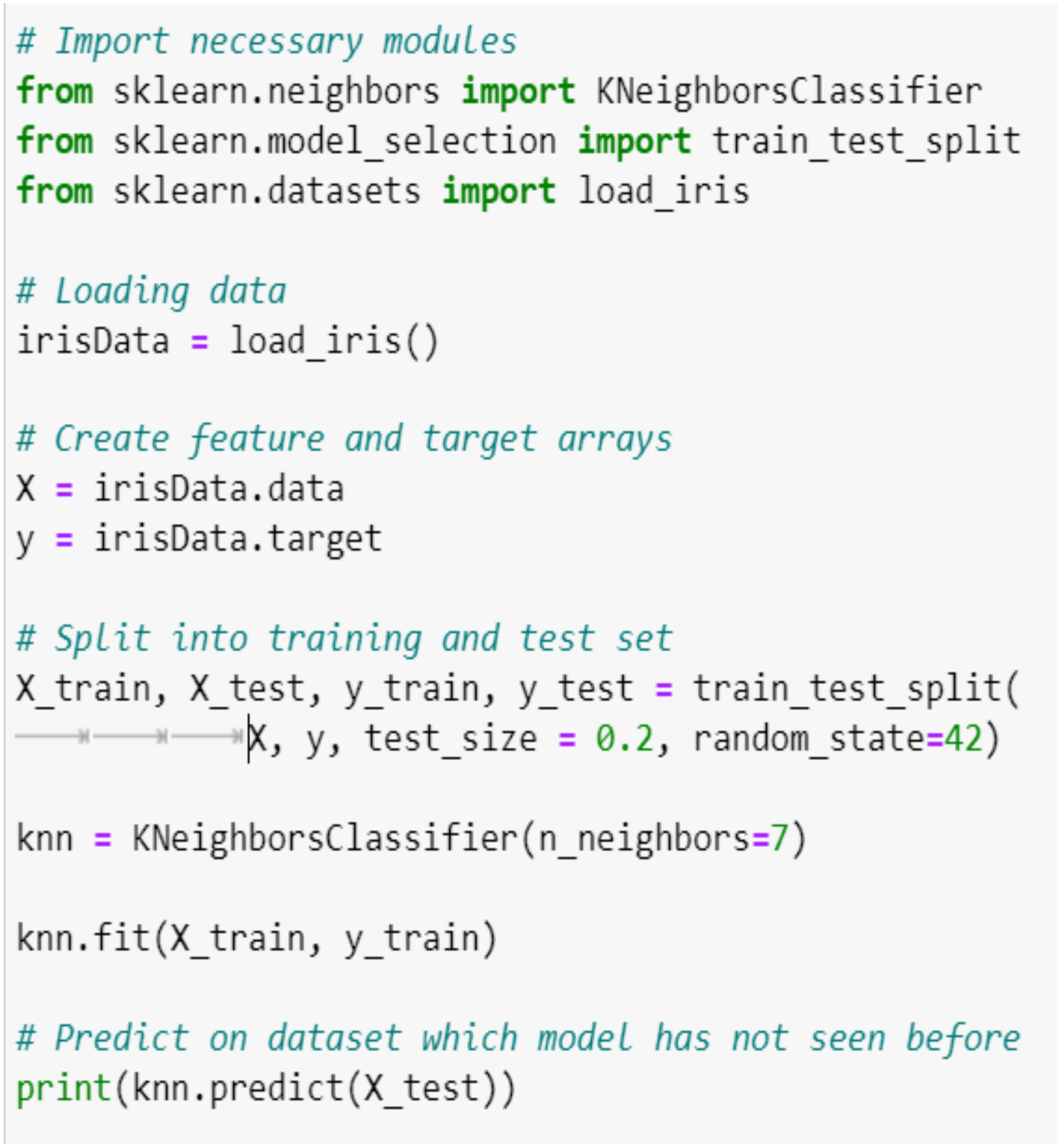
- **K** is the max_feature in Scikit-learn documentation and refers to the number of features to be considered at each decision node. The higher the value of K, more features are considered at each decision node, and hence lower the bias of the model. However, too high a value of K reduces randomization, negating the effect of the ensemble.
- **nmin** maps to min_sample_leaf, and is a minimum number of samples required to be at a leaf node. The higher its value, the less likely the model is to overfit. Smaller numbers of samples result in more splits and a deeper, more specialized tree.
- **M** maps to n_estimators, and is a number of trees in the forest. The higher its value, the lower the variance of the model.

KNN Model-Classification



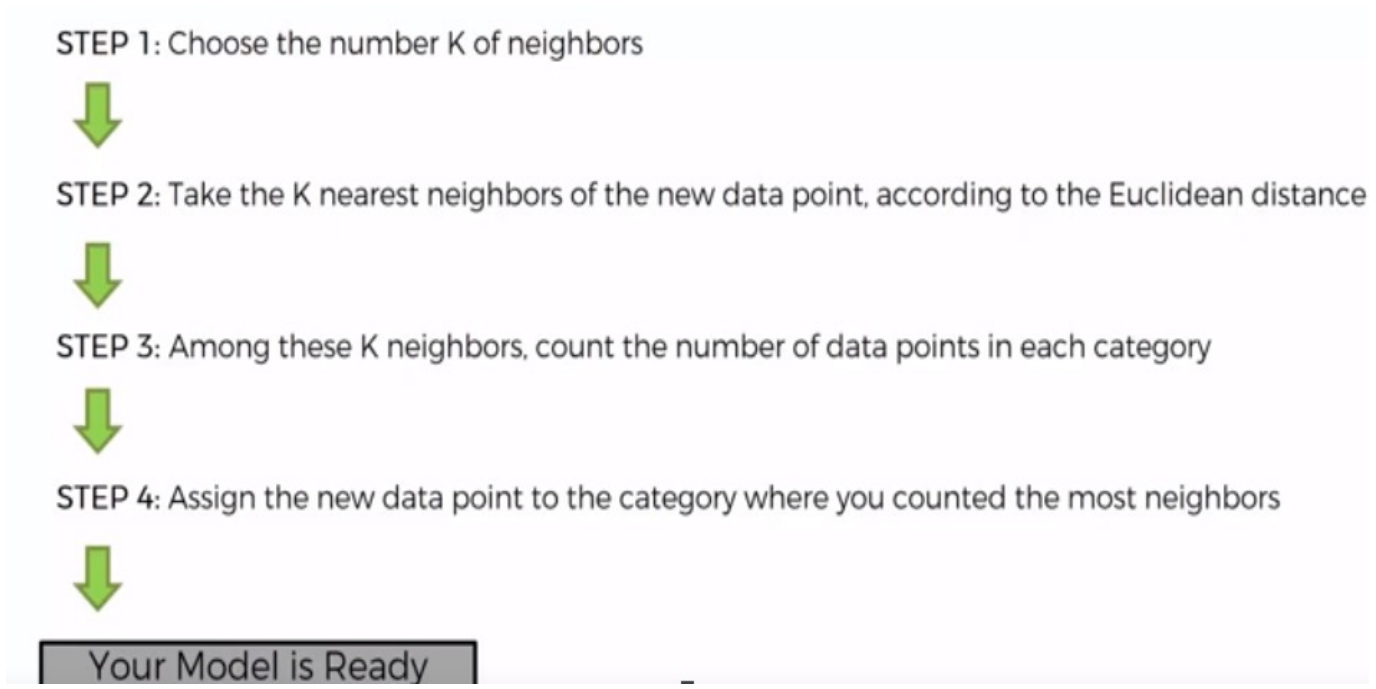


Euclidean Distance between P_1 and $P_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$



```
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Loading data
irisData = load_iris()

# Create feature and target arrays
X = irisData.data
y = irisData.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(
			X, y, test_size = 0.2, random_state=42)

knn = KNeighborsClassifier(n_neighbors=7)

knn.fit(X_train, y_train)

# Predict on dataset which model has not seen before
print(knn.predict(X_test))
```

KNN Model-Classification



STEP 1: Choose the number K of neighbors

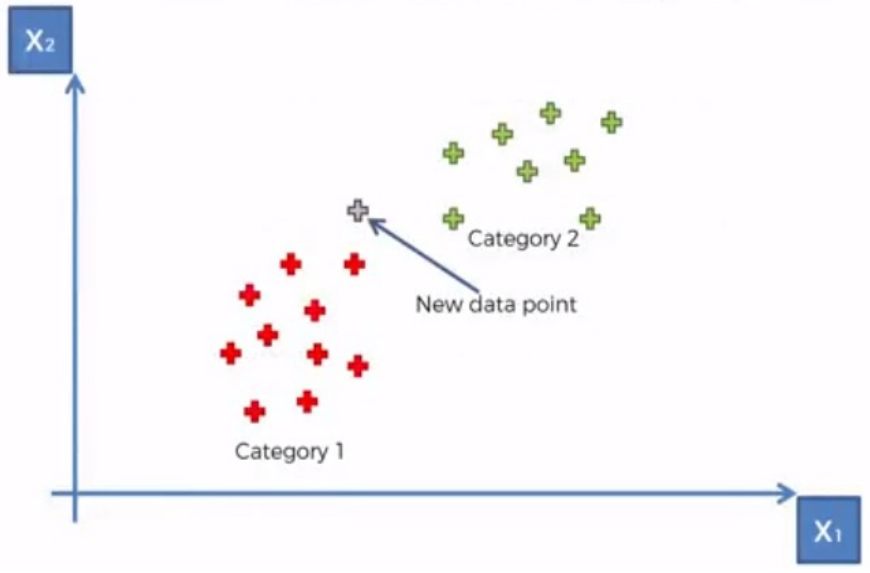
STEP 2: Take the K nearest neighbors of the new data point, according to the Euclidean distance

STEP 3: Among these K neighbors, count the number of data points in each category

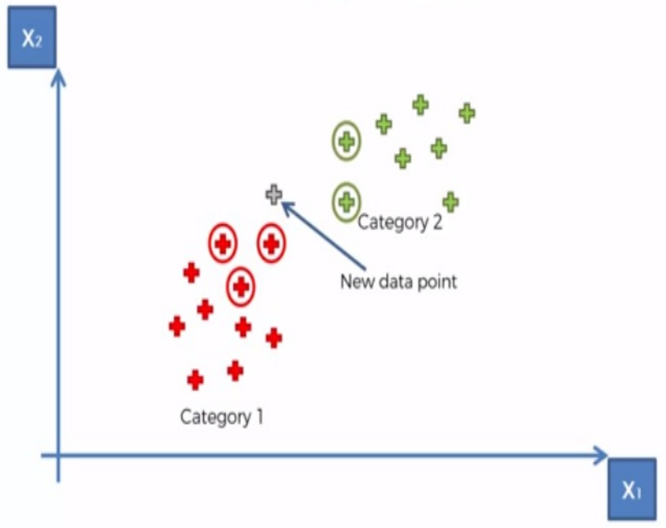
STEP 4: Assign the new data point to the category where you counted the most neighbors

Your Model is Ready

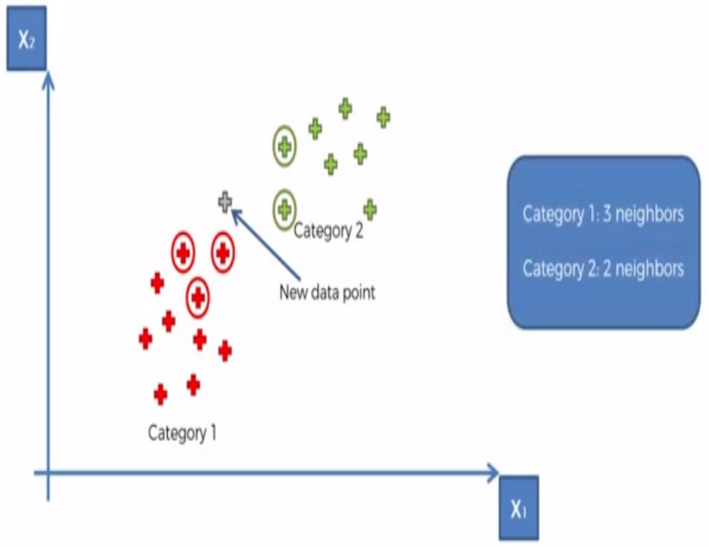
STEP 1: Choose the number K of neighbors: K = 5



STEP 2: Take the K = 5 nearest neighbors of the new data point, according to the Euclidean distance



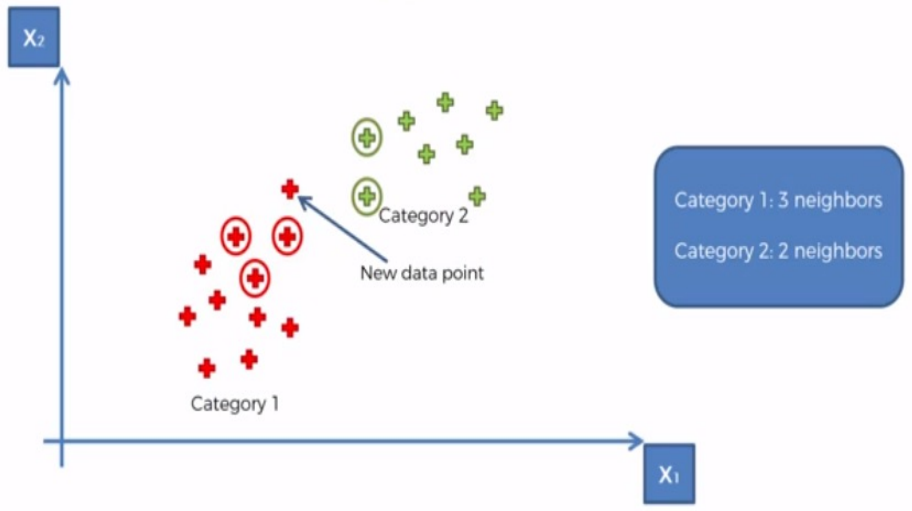
STEP 3: Among these K neighbors, count the number of data points in each category



Category 1: 3 neighbors

Category 2: 2 neighbors

STEP 4: Assign the new data point to the category where you counted the most neighbors



Category 1: 3 neighbors

Category 2: 2 neighbors

SVM-Classification

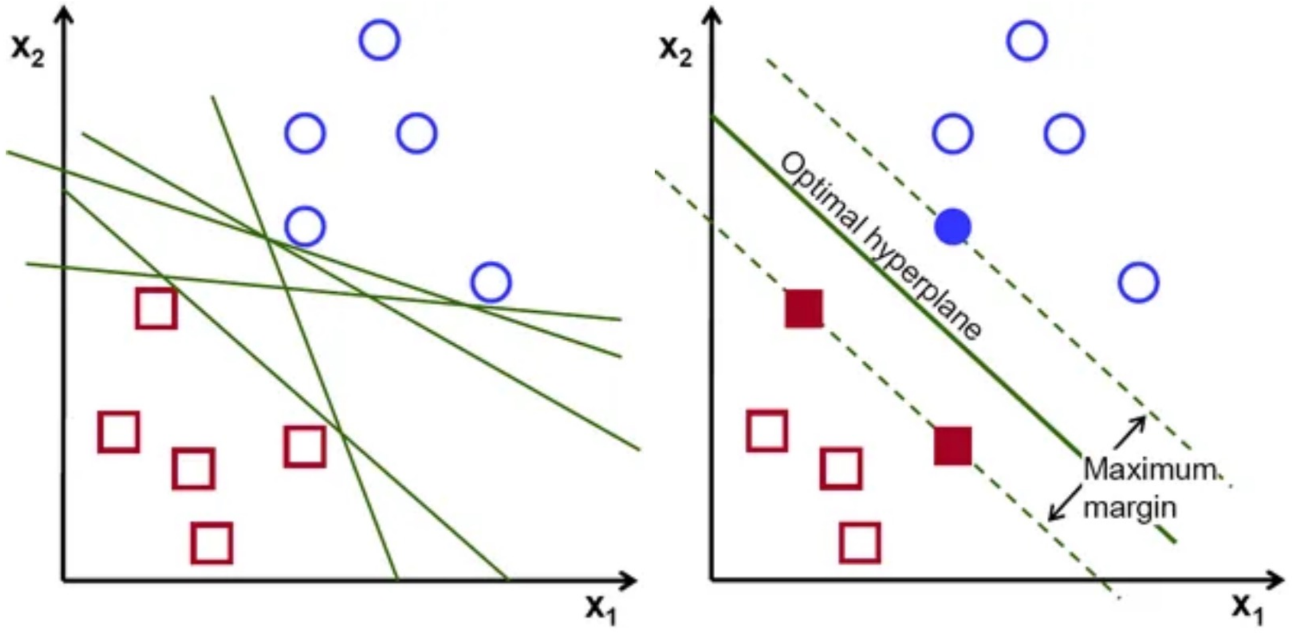
Hyperplane in Machine Learning

In Machine Learning, a hyperplane is a decision boundary that divides the input space into two or more regions, each corresponding to a different class or output label. In a 2D space, a hyperplane is a straight line that divides the space into two halves. In a 3D space, however, a hyperplane is a plane that divides the space into two halves. Meanwhile in higher-dimensional spaces, a hyperplane is a subspace of one dimension less than the input space.

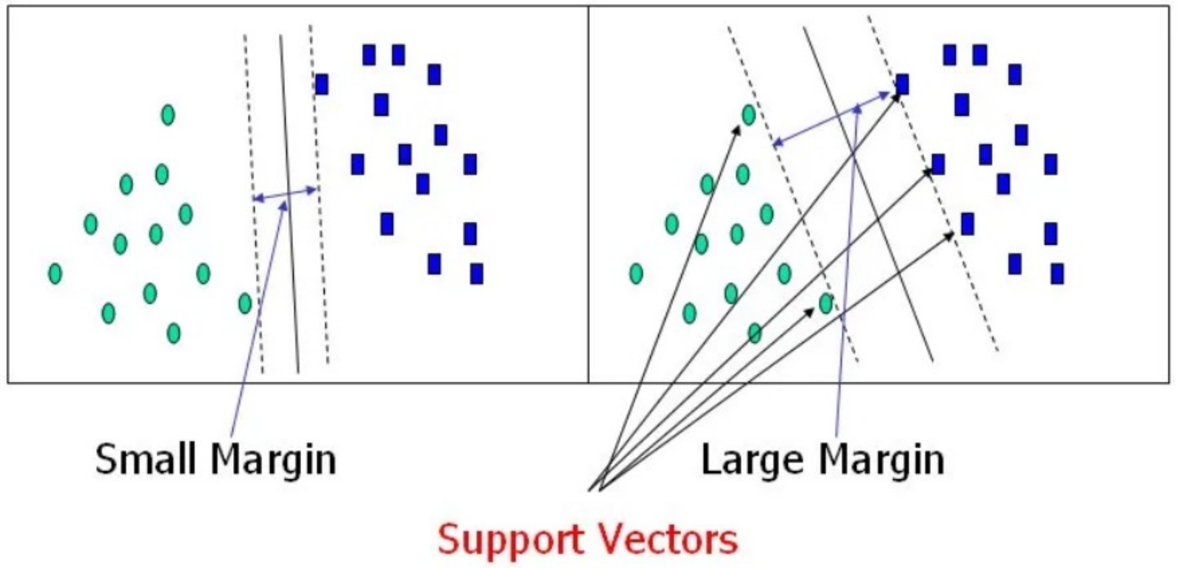
Supporting hyperplane

A supporting hyperplane is a hyperplane that touches at least one data point from each class. In a two-class classification problem, there may be multiple hyperplanes that separate the classes, but only one of these — the maximum margin hyperplane — has the maximum distance between the hyperplane and the nearest data points from each class. This maximum distance is called the margin. The maximum margin hyperplane is often preferred because it has the largest separation between the classes and is therefore less prone to overfitting and more generalizable to unseen data.

SVM-Classification(Linear)

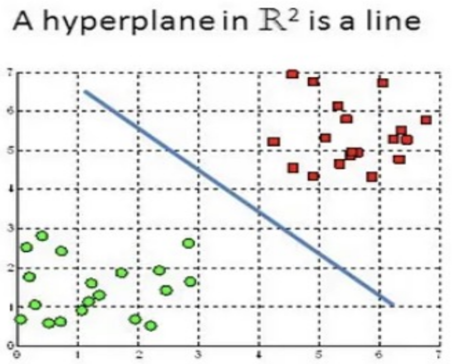


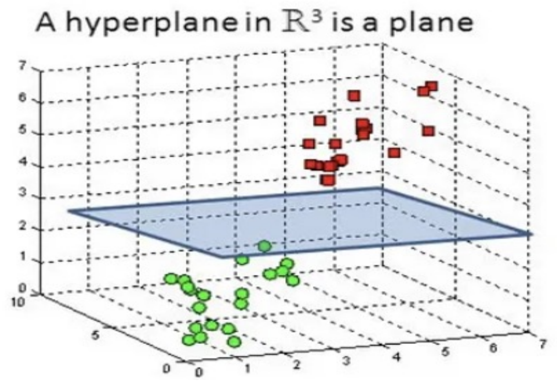
Possible hyperplanes



Support Vectors

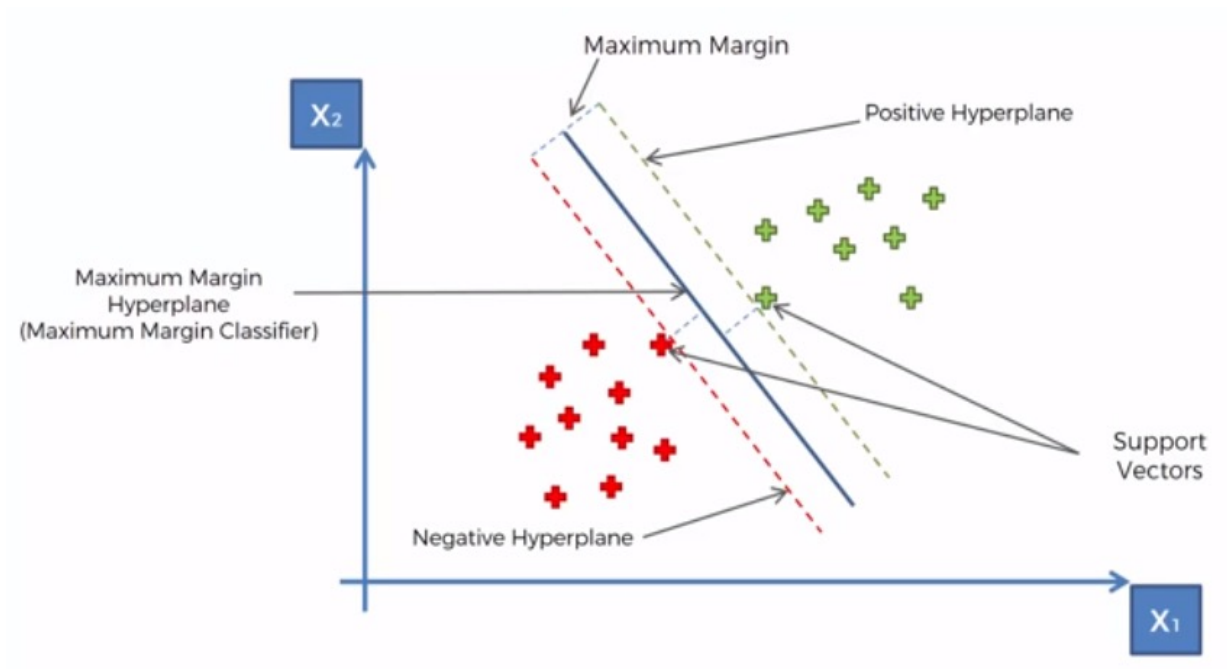
Hyperplanes and Support Vectors

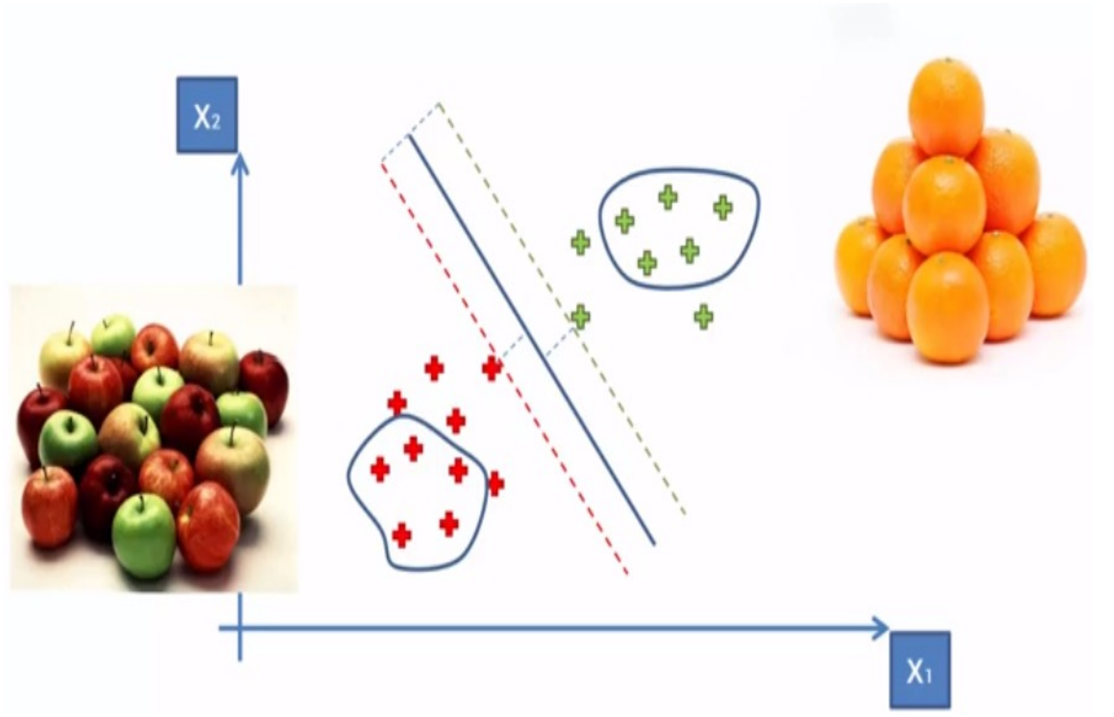


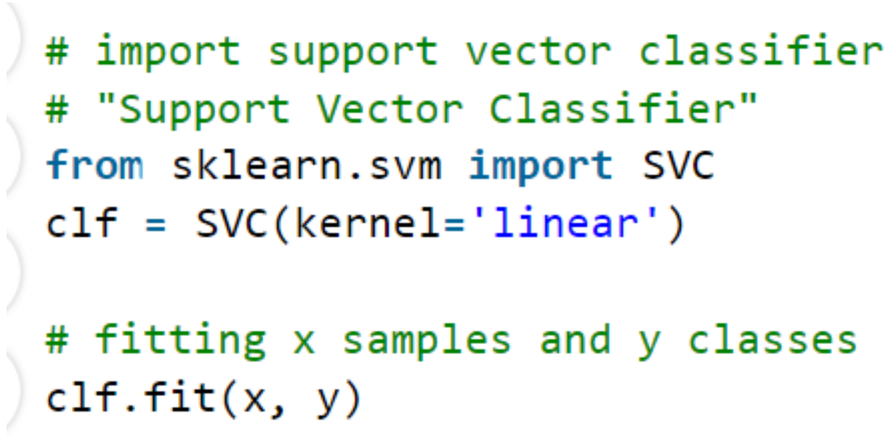


- **Hyperplanes** are decision boundaries that help classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3
- **Support vectors** are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

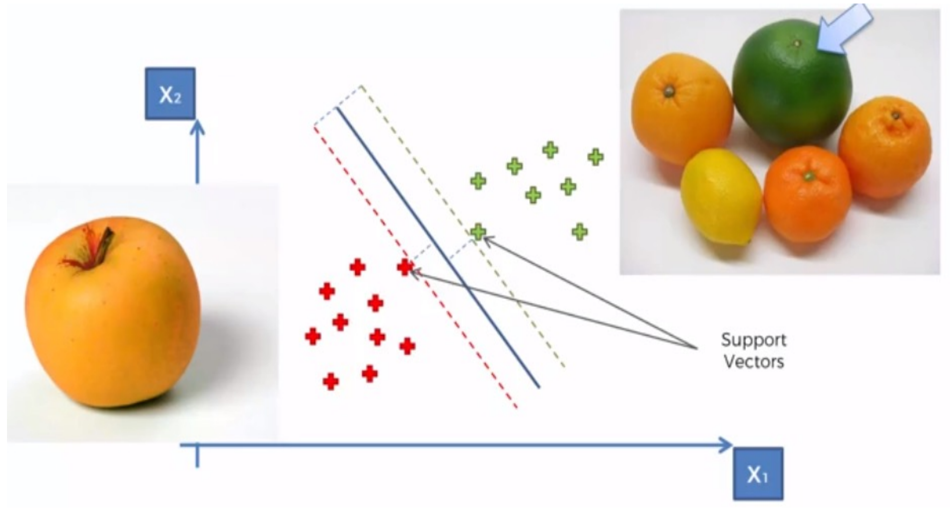
SVM-Classification (Linear)





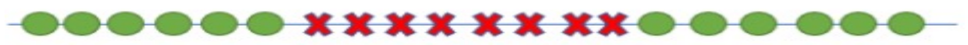


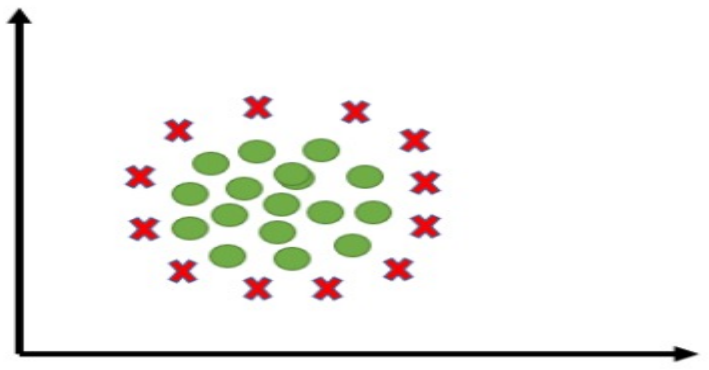
```
# import support vector classifier
# "Support Vector Classifier"
from sklearn.svm import SVC
clf = SVC(kernel='linear')

# fitting x samples and y classes
clf.fit(x, y)
```



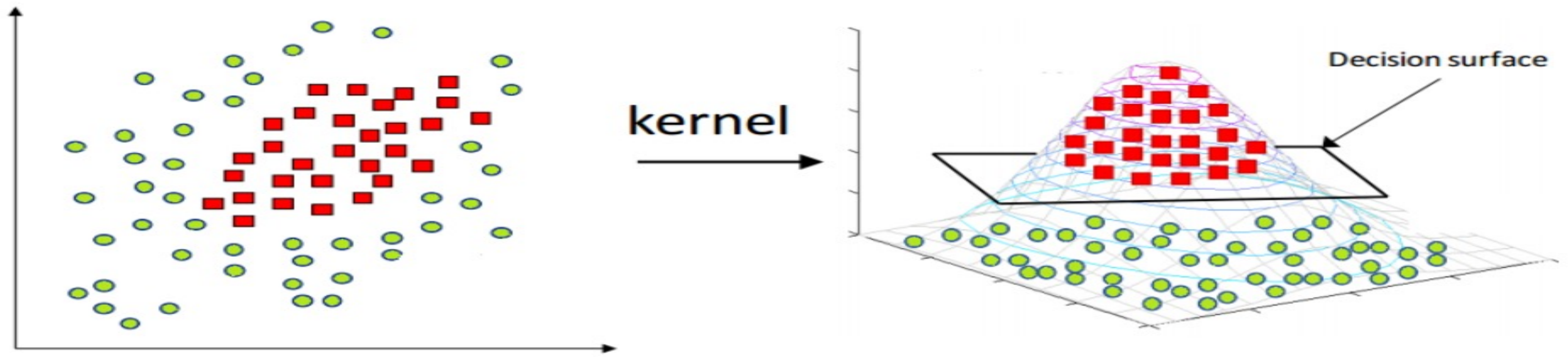
SVM-Non Linear (Kernel)

The most interesting feature of SVM is that it can even work with a non-linear dataset and for this, we use "Kernel Trick" which makes it easier to classifies the points. Suppose we have a dataset like this:

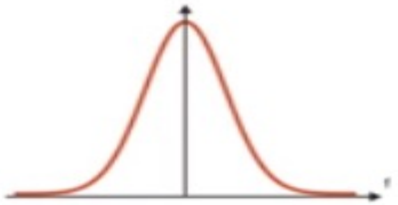




Here we see we cannot draw a single line or say hyperplane which can classify the points correctly. So what we do is try converting this lower dimension space to a higher dimension space using some quadratic functions which will allow us to find a decision boundary that clearly divides the data points. These functions which help us do this are called Kernels and which kernel to use is purely determined by hyperparameter tuning.

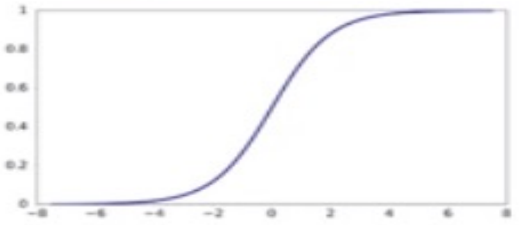


SVM-Non Linear (Different Types of Kernels)



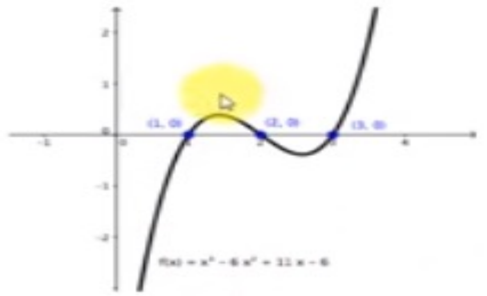
Gaussian RBF Kernel

$$K(\vec{x},\vec{l}^i) = e^{-\frac{\|\vec{x}-\vec{l}^i\|^2}{2\sigma^2}}$$



Sigmoid Kernel

$$K(X,Y) = \tanh(\gamma \cdot X^T Y + r)$$



Polynomial Kernel

$$K(X,Y) = (\gamma \cdot X^T Y + r)^d, \gamma > 0$$

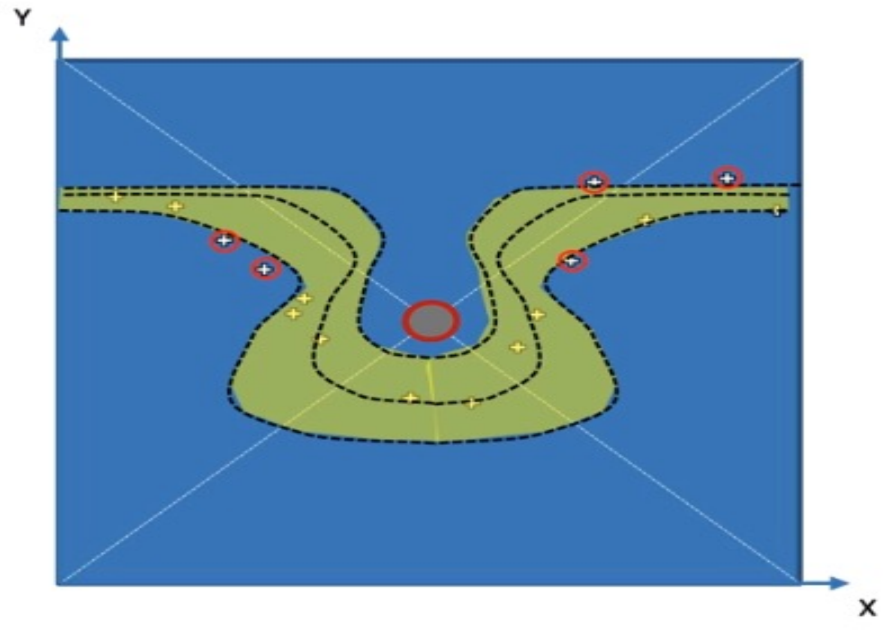
•**Sigmoid Kernel:** this function is equivalent to a two-layer, perceptron model of the neural network, which is used as an activation function for artificial neurons.

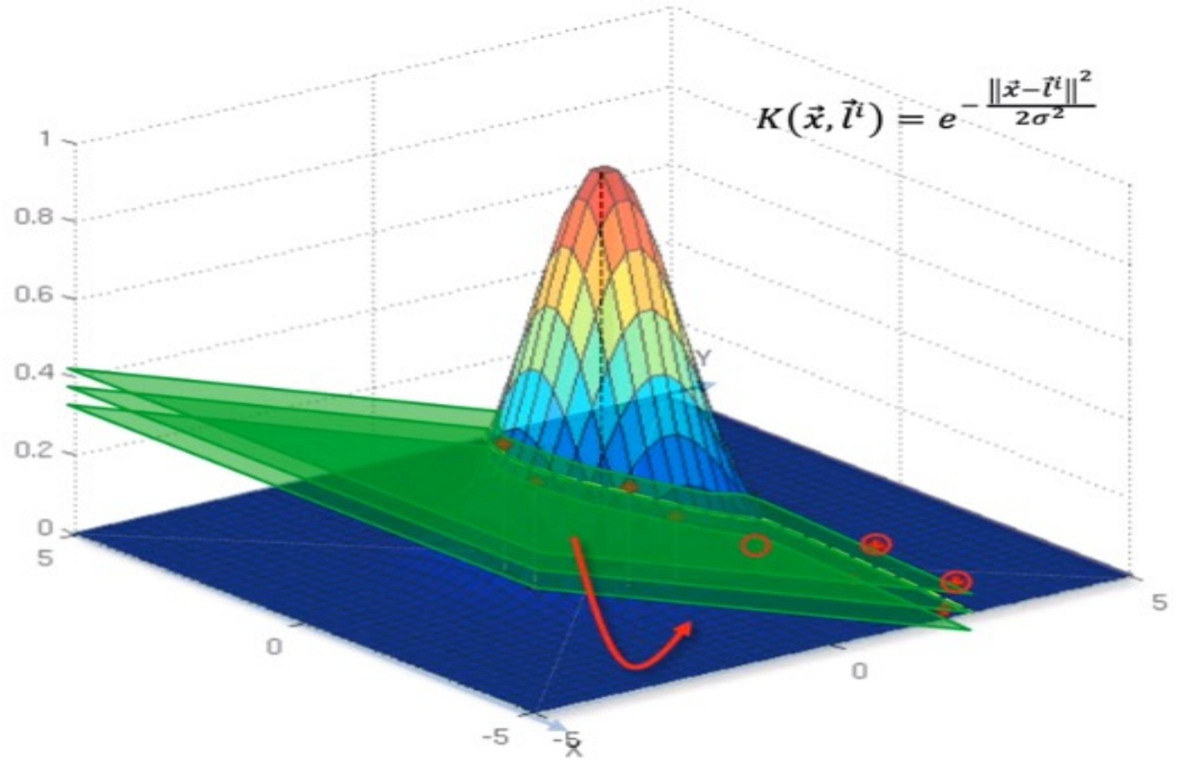
```
from sklearn.svm import SVC
classifier = SVC(kernel ='sigmoid')
classifier.fit(x_train, y_train) # training set in x, y axis
```

•**Polynomial Kernel:** It represents the similarity of vectors in the training set of data in a feature space over polynomials of the original variables used in the kernel.

```
from sklearn.svm import SVC
classifier = SVC(kernel ='poly', degree = 4)
classifier.fit(x_train, y_train) # training set in x, y axis
```

SVM-Non Linear (Gaussian Kernel Radial Basis Function (RBF))

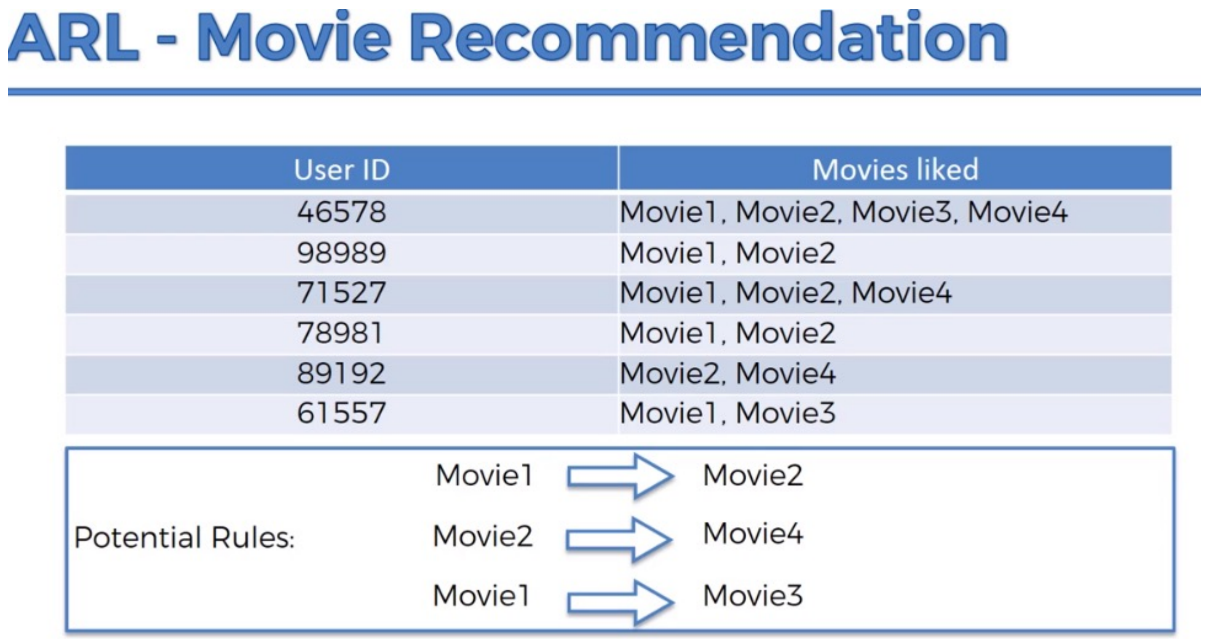




RBF Kernel- What it actually does is to create non-linear combinations of our features to lift your samples onto a higher-dimensional feature space where we can use a linear decision boundary to separate your classes It is the most used kernel in SVM classifications

```
from sklearn.svm import SVC
classifier = SVC(kernel ='rbf', random_state = 0)
# training set in x, y axis
classifier.fit(x_train, y_train)
```

APRIORI

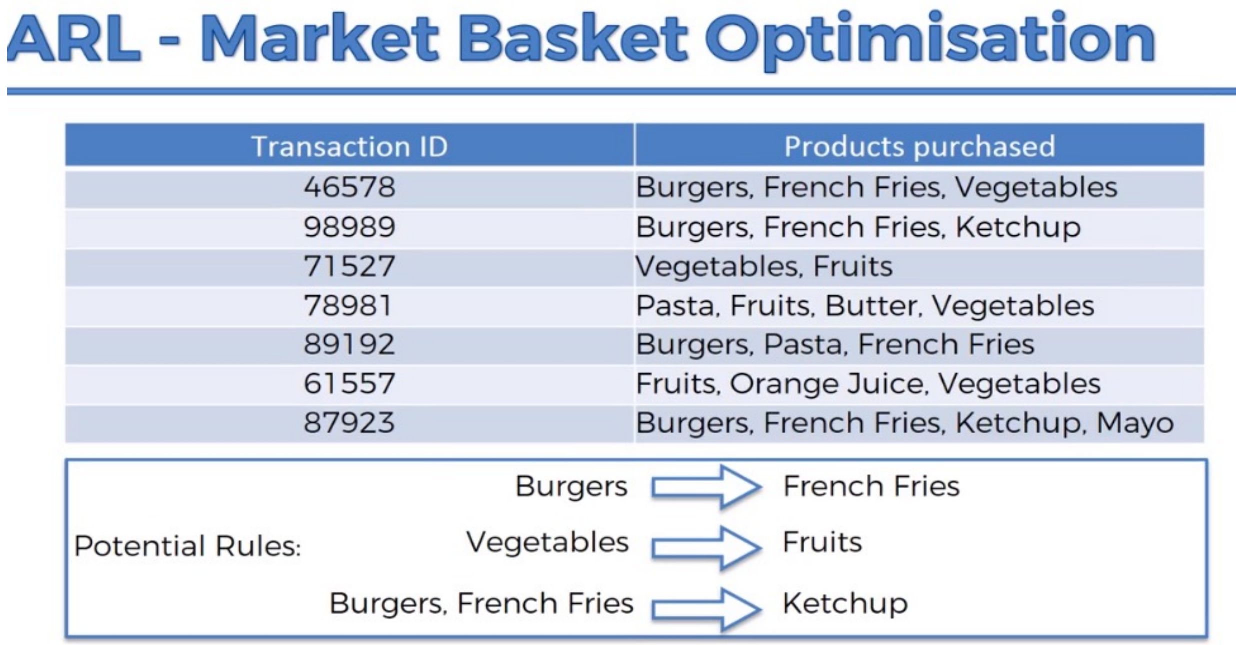


ARL - Movie Recommendation

User ID	Movies liked
46578	Movie1, Movie2, Movie3, Movie4
98989	Movie1, Movie2
71527	Movie1, Movie2, Movie4
78981	Movie1, Movie2
89192	Movie2, Movie4
61557	Movie1, Movie3

Potential Rules:

$$\text{Movie1} \Rightarrow \text{Movie2}$$
$$\text{Movie2} \Rightarrow \text{Movie4}$$
$$\text{Movie1} \Rightarrow \text{Movie3}$$



ARL - Market Basket Optimisation

Transaction ID	Products purchased
46578	Burgers, French Fries, Vegetables
98989	Burgers, French Fries, Ketchup
71527	Vegetables, Fruits
78981	Pasta, Fruits, Butter, Vegetables
89192	Burgers, Pasta, French Fries
61557	Fruits, Orange Juice, Vegetables
87923	Burgers, French Fries, Ketchup, Mayo

Potential Rules:

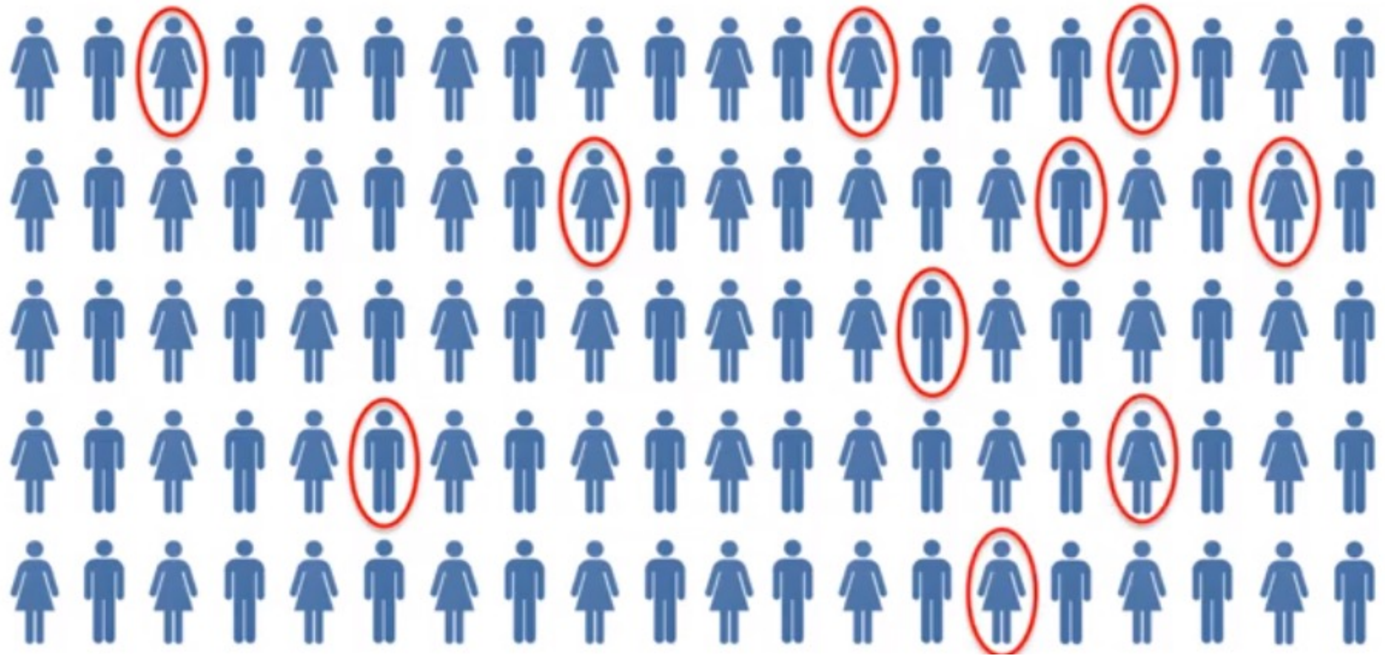
$$\text{Burgers} \Rightarrow \text{French Fries}$$
$$\text{Vegetables} \Rightarrow \text{Fruits}$$
$$\text{Burgers, French Fries} \Rightarrow \text{Ketchup}$$

1. APRIORI-Support

Support = 10 / 100 = 10%

Movie Recommendation: $$\text{support}(\boldsymbol{M}) = \frac{\#\ \text{user watchlists containing}\ \boldsymbol{M}}{\#\ \text{user watchlists}}$$

Market Basket Optimisation: $$\text{support}(\boldsymbol{I}) = \frac{\#\ \text{transactions containing}\ \boldsymbol{I}}{\#\ \text{transactions}}$$

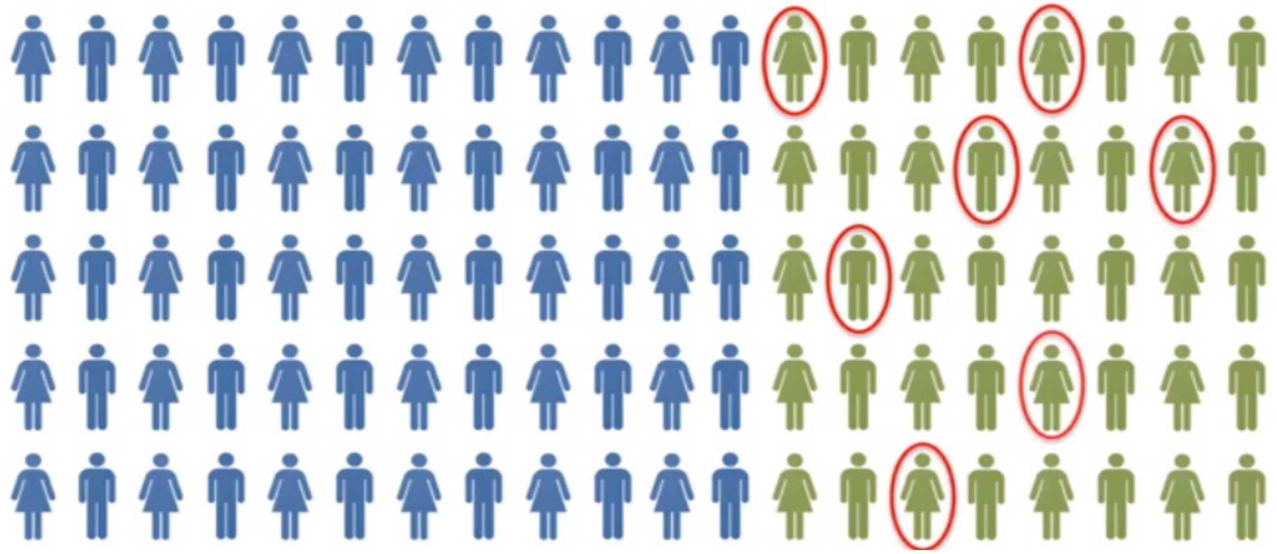


2. APRIORI-Confidence

Confidence = 7 / 40 = 17.5%

Movie Recommendation: $\text{confidence}(M_1 \rightarrow M_2) = \dfrac{\#\ \text{user watchlists containing } M_1 \text{ and } M_2}{\#\ \text{user watchlists containing } M_1}$

Market Basket Optimisation: $\text{confidence}(I_1 \rightarrow I_2) = \dfrac{\#\ \text{transactions containing } I_1 \text{ and } I_2}{\#\ \text{transactions containing } I_1}$

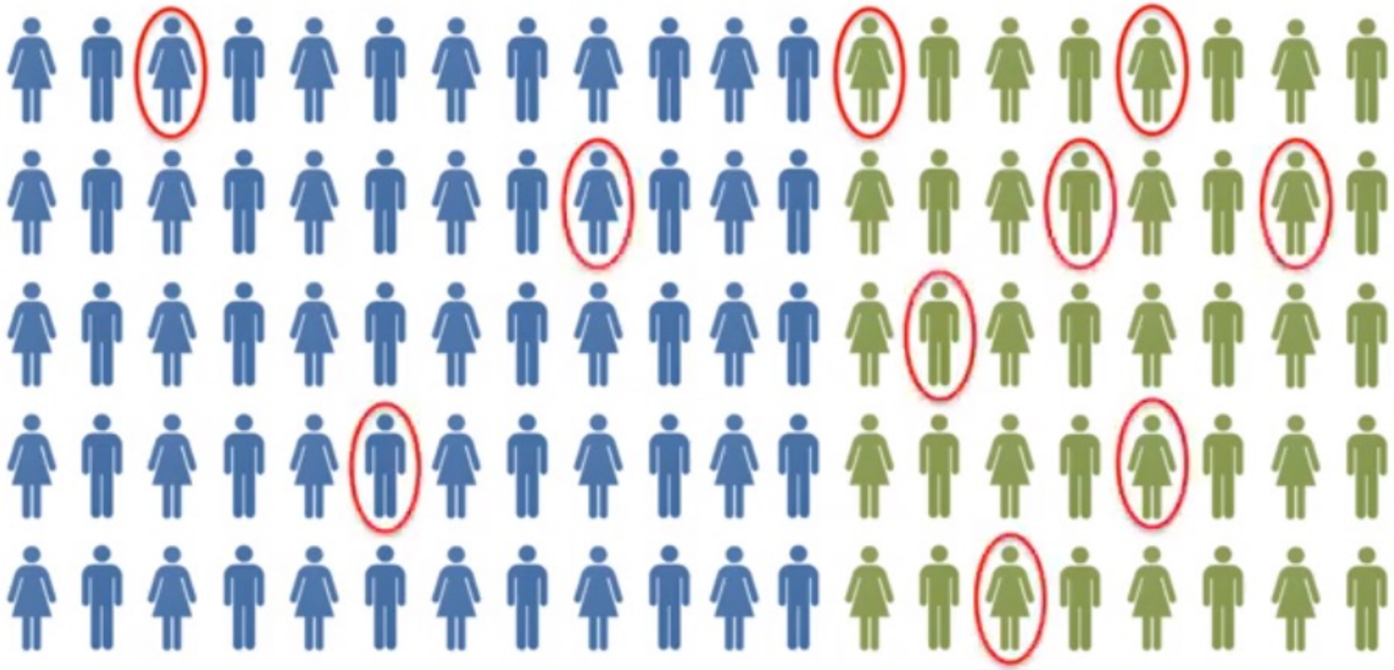


3. APRIORI-Lift

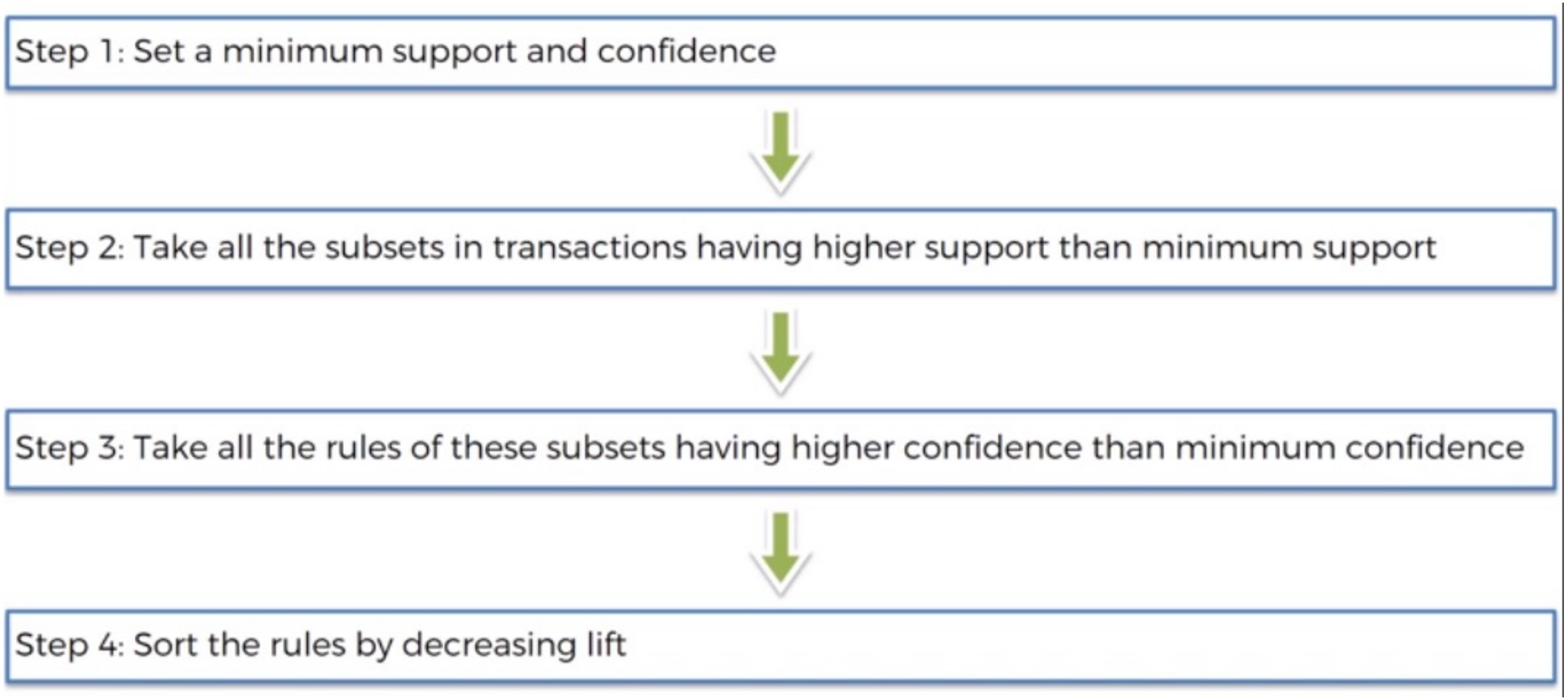
Lift = 17.5% / 10% = 1.75

Movie Recommendation: $\mathrm{lift}(M_1 \rightarrow M_2) = \dfrac{\mathrm{confidence}(M_1 \rightarrow M_2)}{\mathrm{support}(M_2)}$

Market Basket Optimisation: $\mathrm{lift}(I_1 \rightarrow I_2) = \dfrac{\mathrm{confidence}(I_1 \rightarrow I_2)}{\mathrm{support}(I_2)}$



APRIORI- Algorithm



Step 1: Set a minimum support and confidence

Step 2: Take all the subsets in transactions having higher support than minimum support

Step 3: Take all the rules of these subsets having higher confidence than minimum confidence

Step 4: Sort the rules by decreasing lift

ARL- Metrics

1. Antecedents and consequents: These columns represent the items involved in each association rule. Antecedents are the items found in the transactions before the rule, and consequents are the items predicted to occur after the rule.

2. Antecedent support and consequent support: These columns show the proportion of transactions that contain the antecedent or consequent items, respectively. Antecedent support is the percentage of transactions that contain the antecedent, and consequent support is the percentage that contains the consequent.

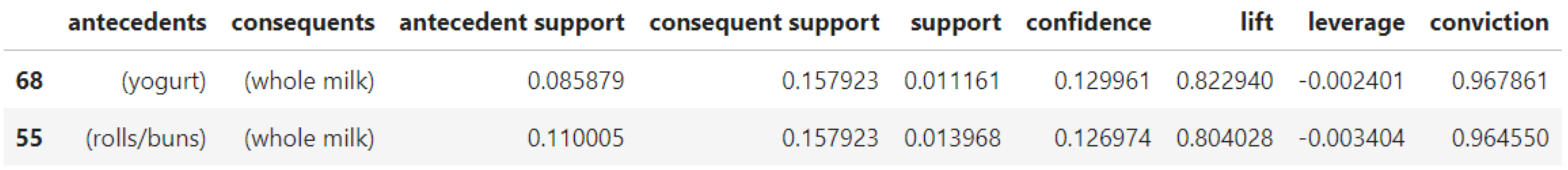
3. Support: Support indicates the proportion of transactions that contain both the antecedent and the consequent. It's a measure of how frequently the rule is observed in the dataset.

4. Confidence: Confidence is the probability of the consequent occurring given that the antecedent has occurred. It's calculated as `support(consequent and antecedent) / support(antecedent)`.

5. Lift: Lift measures how much more likely the consequent is to occur when the antecedent is present compared to when it's not. A lift greater than 1 indicates a positive association.

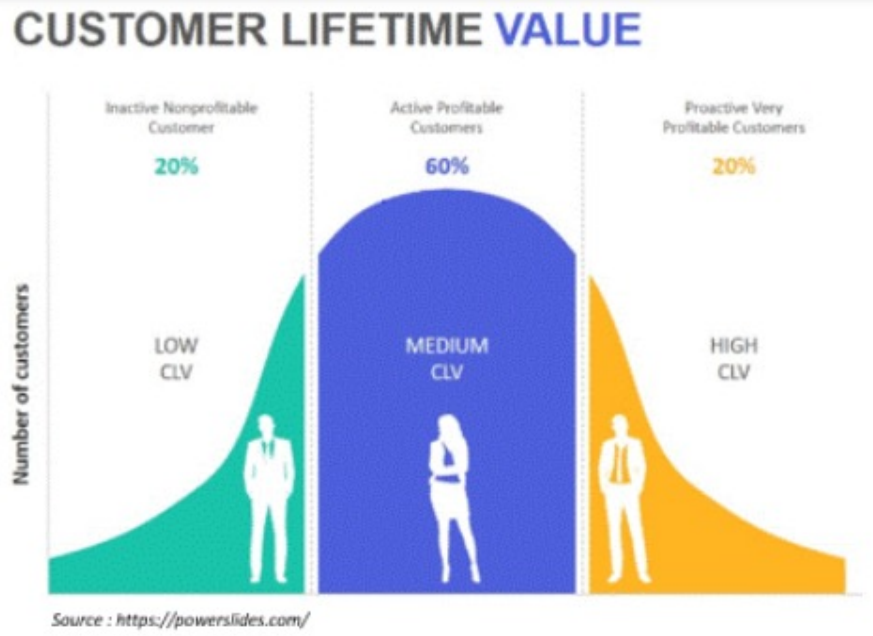
6. Leverage: Leverage measures the difference between the observed frequency of the antecedent and consequent occurring together and the frequency expected if they were independent.

7. Conviction: Conviction is the ratio of the expected frequency that the antecedent occurs without the consequent to the observed frequency of the antecedent occurring without the consequent. It helps identify how much more often the antecedent occurs without the consequent than expected if they were independent.



	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
68	(yogurt)	(whole milk)	0.085879	0.157923	0.011161	0.129961	0.822940	-0.002401	0.967861
55	(rolls/buns)	(whole milk)	0.110005	0.157923	0.013968	0.126974	0.804028	-0.003404	0.964550

RFM Analysis



In our stock index, NIFTY 50 defines how our stock market is performing similarly, in business, it's important to understand who your top customers are that give consistent and increasing revenue streams for your business.

One of the simple and effective methodologies which are generally used in calculating customer value over a time frame is RFM which is,

> Recency (R): How recently a customer has made a purchase
> Frequency (F): How often a customer makes a purchase
> Monetary Value (M): Dollar value of the purchases

RFM Analysis

Let's take a small example where a bank wants to identify key customers for retention/development/acquisition.

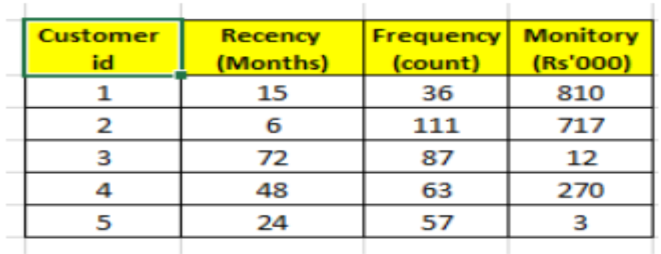
So in the above scenario, we need to score each customer who had recent transactions with the bank on three important metrics mentioned above R, F & M.Then create a scoring methodology to segment the customer base and apply for different marketing programs.

Process of RFM Analysis

Let's calculate the RFM score for 5 sample customers,

Step1: **Derive R, F & M from the transactions of the bank from the last 1 year**.

Preferably RFM is done for recent data and will be refreshed on a quarterly/half-yearly basis based on the business



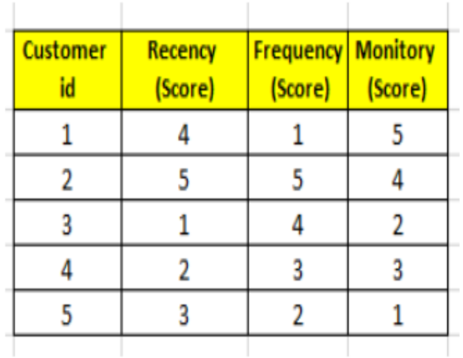
Customer id	Recency (Months)	Frequency (count)	Monitory (Rs'000)
1	15	36	810
2	6	111	717
3	72	87	12
4	48	63	270
5	24	57	3

Finding R, F and M are pretty simple. Let's say a customer deposited 10 K money on May 1st and deposited another 5 k on June 10th and if you are doing RFM analysis on July 1st. Now for this customer, the Recency will be 1 month because the last transaction was in June and Frequency will be 2 because he made two deposits in May and June and M will be 15 K

RFM Analysis

Step 2: **Derive score of each customer based on each parameter based on rank within the parameter**

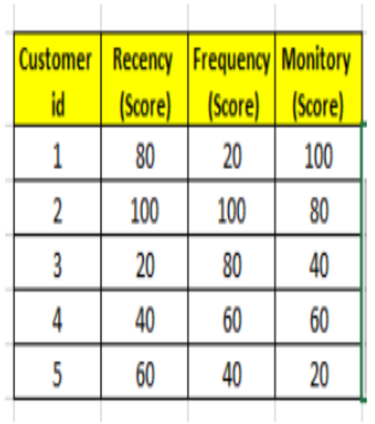
For Recency, smaller the better, because of the customer, we are on top of his mind and for Frequency & Monitory larger values are better



Customer id	Recency (Score)	Frequency (Score)	Monitory (Score)
1	4	1	5
2	5	5	4
3	1	4	2
4	2	3	3
5	3	2	1

Let's take the above table as an example, when compared to all customers, recency is best for customer 3 as he is ranked as number 1, whereas for frequency he is in 4th position, and in terms of the value he is in 2nd position.

Step 3: **Standardize score of each customer based on each parameter (0-100)**



Customer id	Recency (Score)	Frequency (Score)	Monitory (Score)
1	80	20	100
2	100	100	80
3	20	80	40
4	40	60	60
5	60	40	20

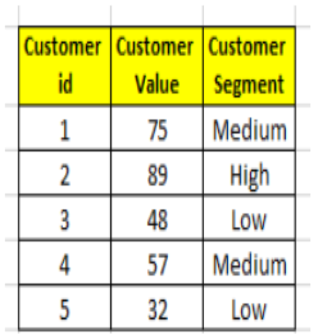
*Standardize = current value/Max(Value) * 100*

RFM Analysis

Step 4: **Derive weighted score across each parameter for each customer**

Consolidated Score = 0.15*R + 0.28*F + 0.57*M

Weights can be applied equally or we can provide specific weights for each parameter based on domain knowledge or business inputs. Here in the above case, we are giving more importance to Frequency and Monitory.



Customer id	Customer Value	Customer Segment
1	75	Medium
2	89	High
3	48	Low
4	57	Medium
5	32	Low

We just applied those weights to each customer.

For example,

*Customer 4 value = 0.15 *40 + 0.28 *60 + 0.57*60 = 57*

Then we segregated score as three segments,

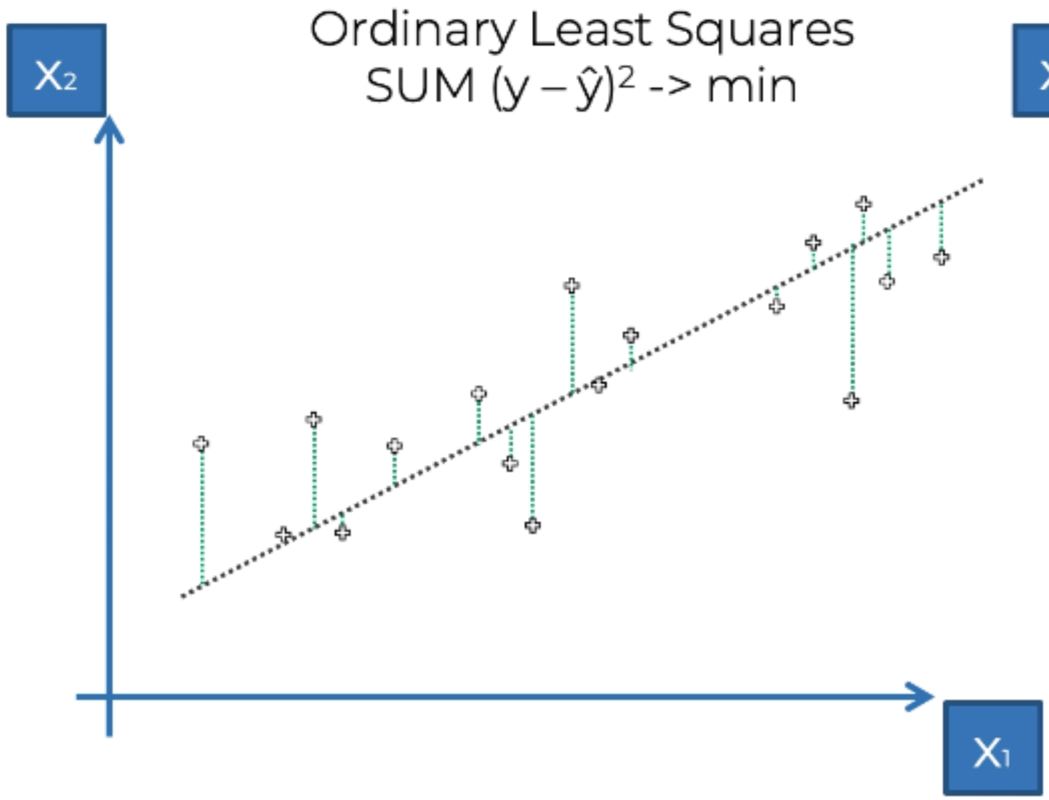
- 0 - 50 - Low valued customer
- 50 - 75 - Medium valued customer
- 76 - 100 - High valued customer

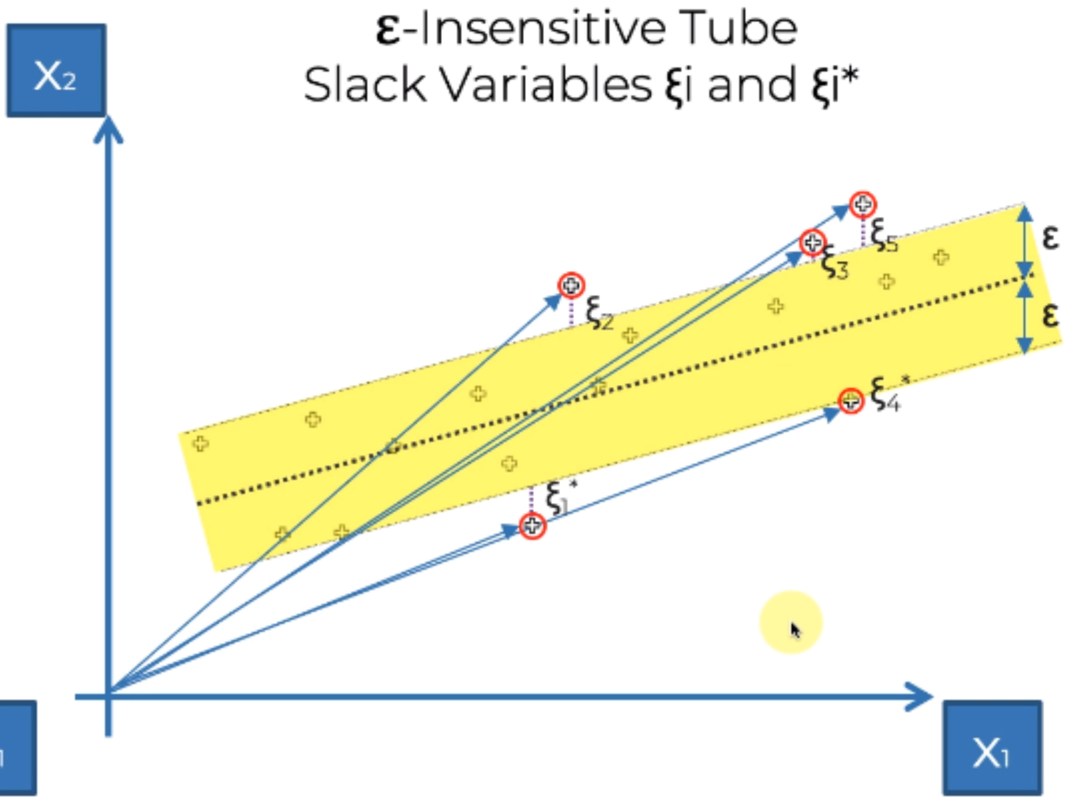
Now based on the above scores, a business can apply the differentiation strategy like retention/development/acquisition of different customer segments

Further, most we can profile these segments with additional features like demographics, spending pattern, and several products e.t.c; understand them a little deeper.

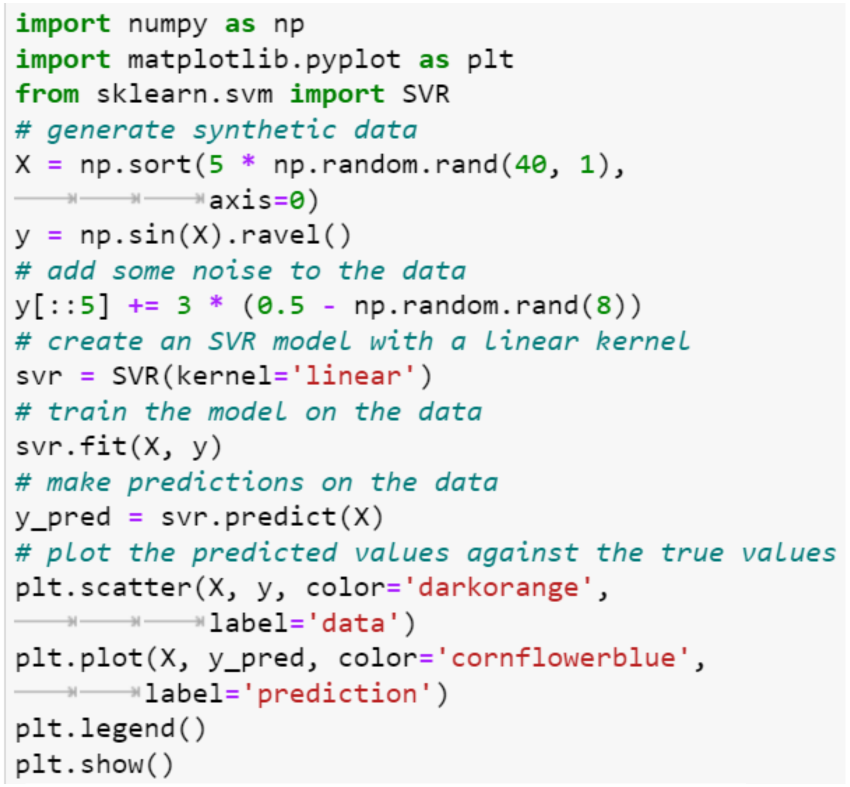
SVR-Support Vector Regression

$$\boxed{\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}(\xi_i + \xi_i^*) \rightarrow min}$$

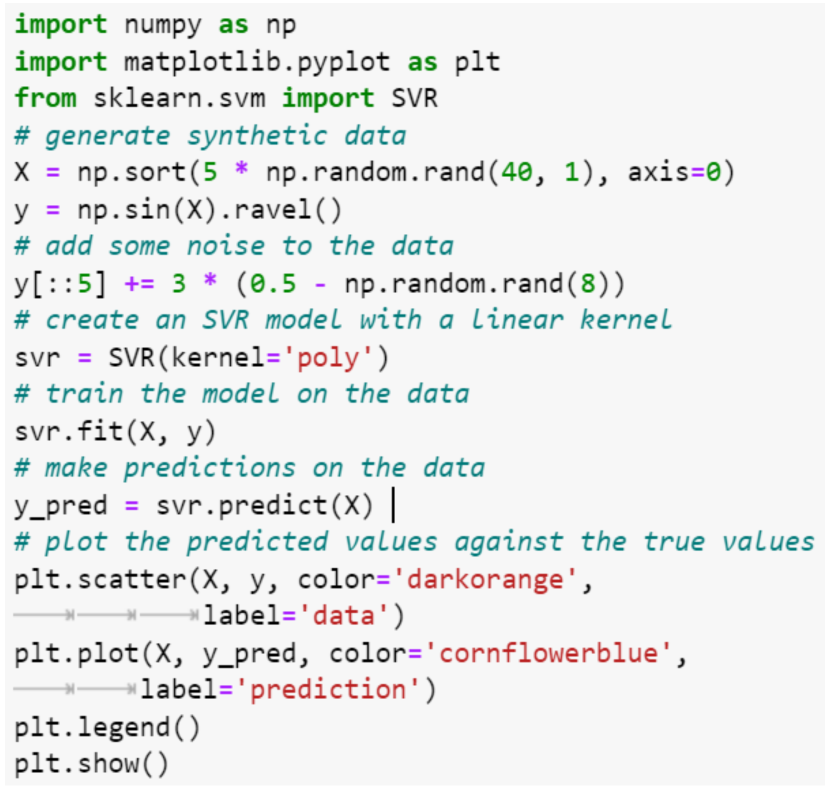




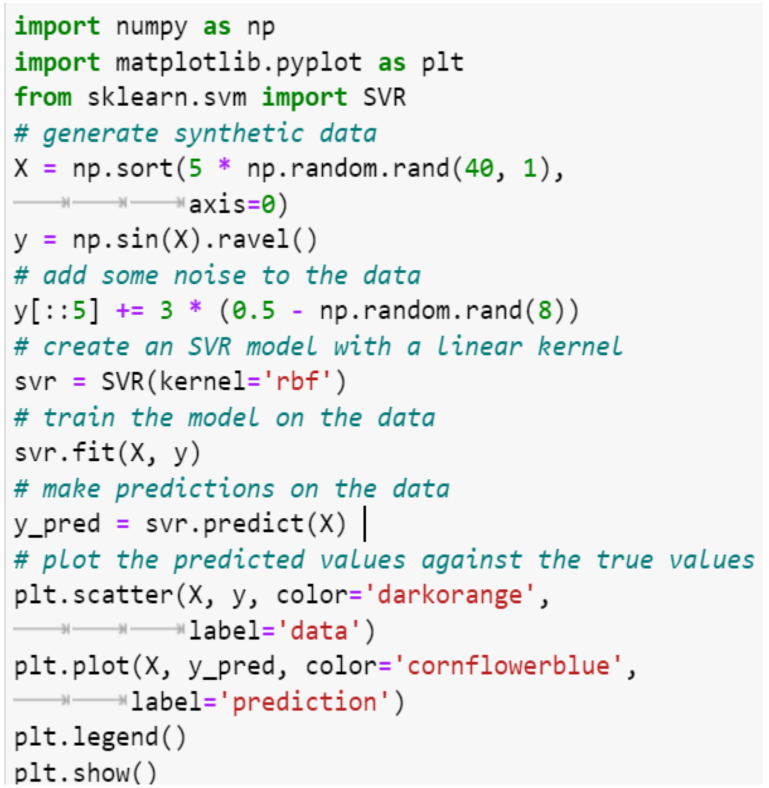
SVR-Support Vector Regression



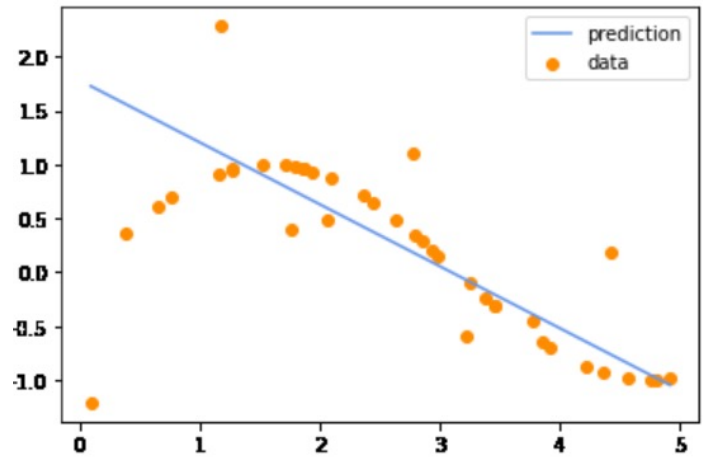
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR
# generate synthetic data
X = np.sort(5 * np.random.rand(40, 1),
            axis=0)
y = np.sin(X).ravel()
# add some noise to the data
y[::5] += 3 * (0.5 - np.random.rand(8))
# create an SVR model with a linear kernel
svr = SVR(kernel='linear')
# train the model on the data
svr.fit(X, y)
# make predictions on the data
y_pred = svr.predict(X)
# plot the predicted values against the true values
plt.scatter(X, y, color='darkorange',
            label='data')
plt.plot(X, y_pred, color='cornflowerblue',
        label='prediction')
plt.legend()
plt.show()
```



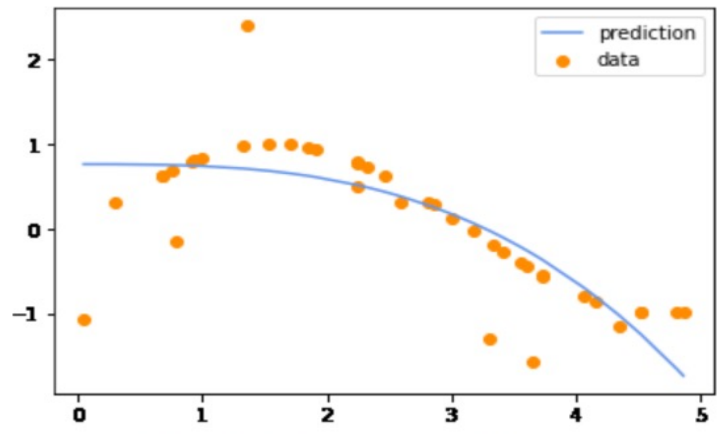
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR
# generate synthetic data
X = np.sort(5 * np.random.rand(40, 1), axis=0)
y = np.sin(X).ravel()
# add some noise to the data
y[::5] += 3 * (0.5 - np.random.rand(8))
# create an SVR model with a linear kernel
svr = SVR(kernel='poly')
# train the model on the data
svr.fit(X, y)
# make predictions on the data
y_pred = svr.predict(X)
# plot the predicted values against the true values
plt.scatter(X, y, color='darkorange',
            label='data')
plt.plot(X, y_pred, color='cornflowerblue',
        label='prediction')
plt.legend()
plt.show()
```



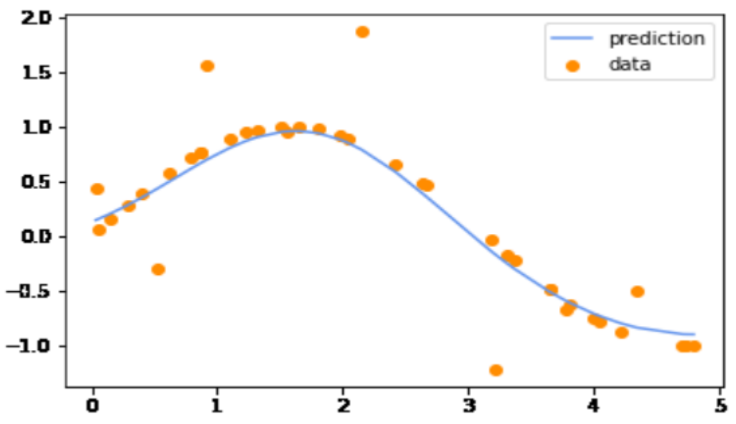
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR
# generate synthetic data
X = np.sort(5 * np.random.rand(40, 1),
            axis=0)
y = np.sin(X).ravel()
# add some noise to the data
y[::5] += 3 * (0.5 - np.random.rand(8))
# create an SVR model with a linear kernel
svr = SVR(kernel='rbf')
# train the model on the data
svr.fit(X, y)
# make predictions on the data
y_pred = svr.predict(X)
# plot the predicted values against the true values
plt.scatter(X, y, color='darkorange',
            label='data')
plt.plot(X, y_pred, color='cornflowerblue',
        label='prediction')
plt.legend()
plt.show()
```



Model fitted using Linear kernel



Model fitted using a polynomial kernel



Model fitted using RBF kernel