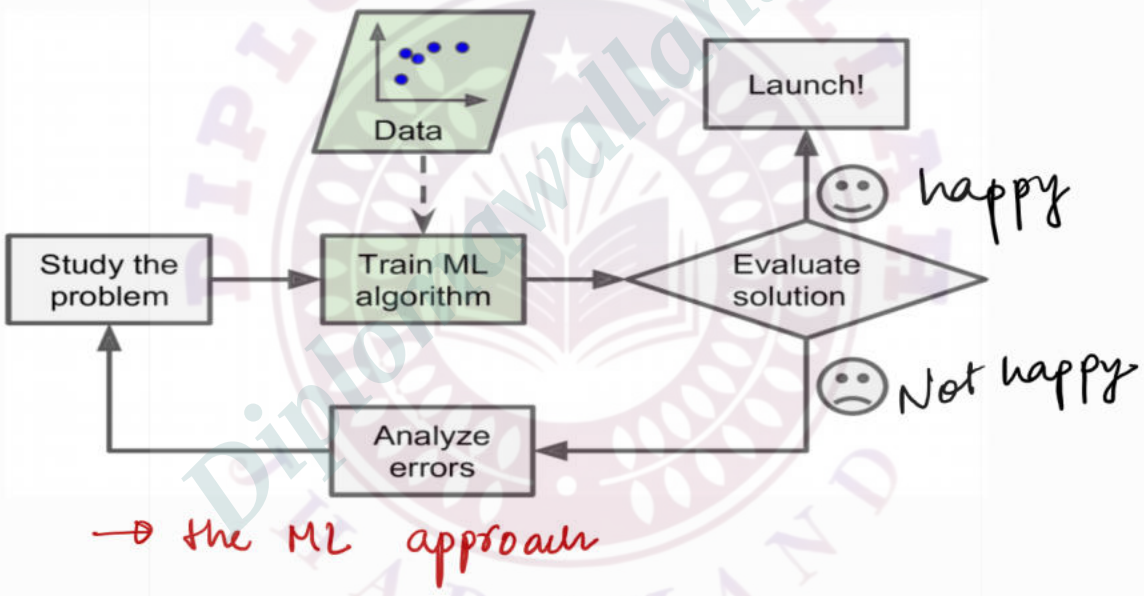
Machine learning

Def:- ML is the science of programming computers so they can learn from data.

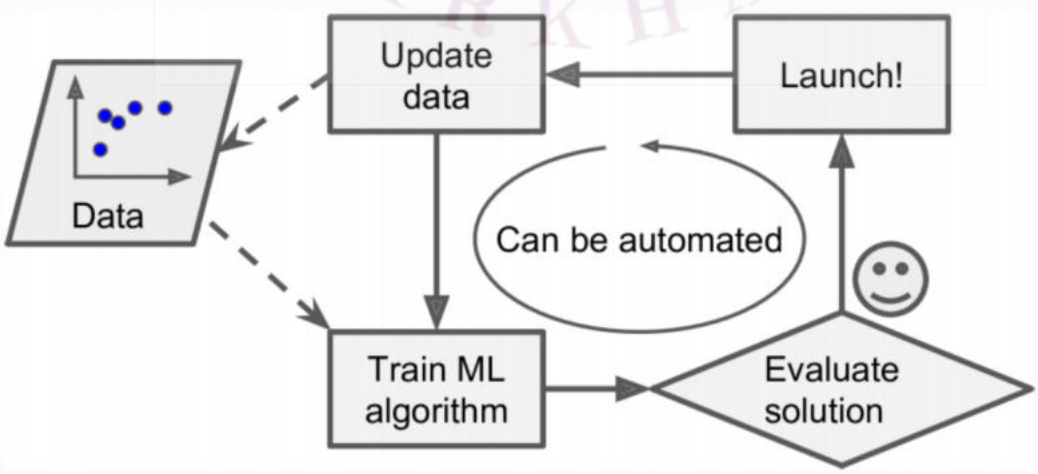
or

ML is the field of study that gives computers the ability to learn without being explicitly programmed

* the examples that the system uses to learn are called training set.

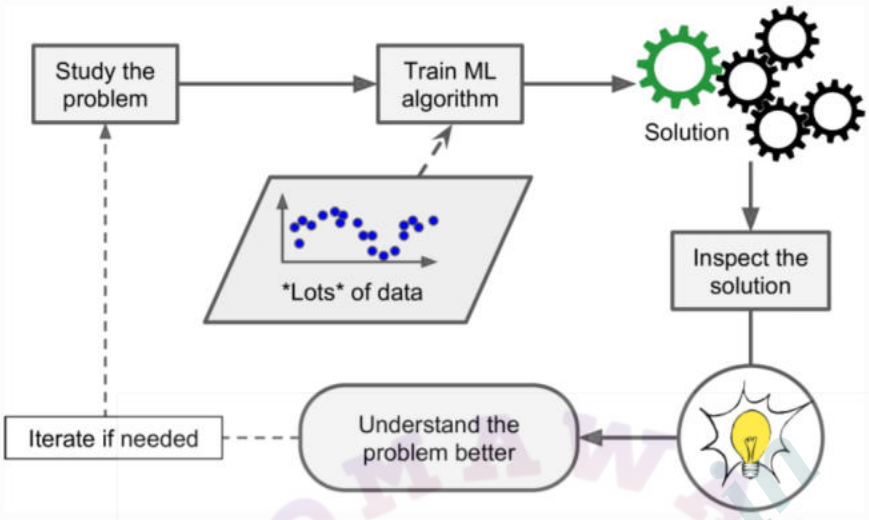


→ the ML approach



→ Automatically adapting to change

* Applying ML techniques to dig into large amount of data can help discover patterns that were not immediately apparent. This is called data mining.



→ ML can helps humans learn

To summarize, Machine Learning is great for:

- Problems for which existing solutions require a lot of hand-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better.
- Complex problems for which there is no good solution at all using a traditional approach: the best Machine Learning techniques can find a solution.
- Fluctuating environments: a Machine Learning system can adapt to new data.
- Getting insights about complex problems and large amounts of data.

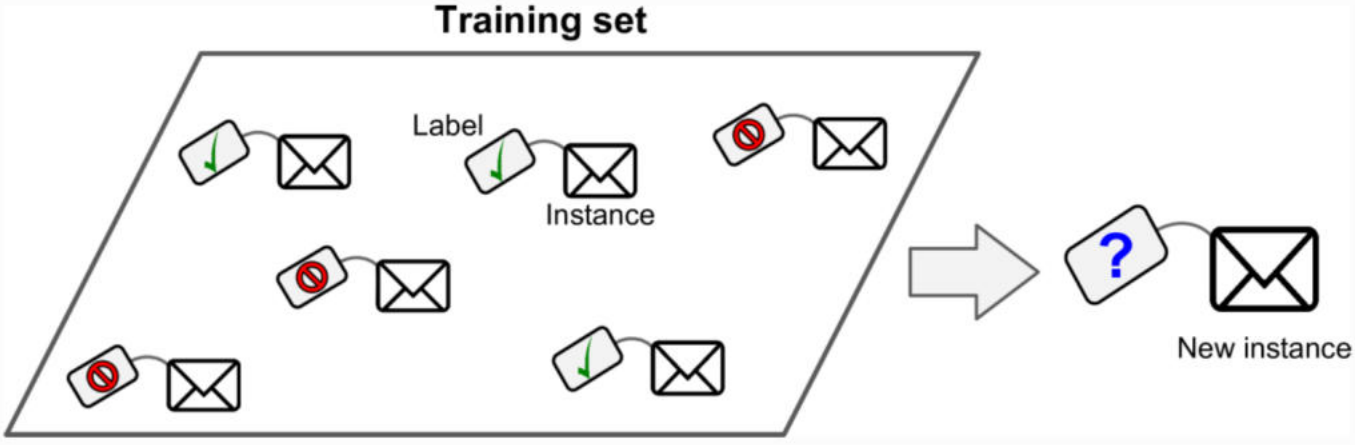
→ Types of ML systems

↳ Supervised, Unsupervised, semisupervised and Reinforcement learning

↳ Online versus batch learning

↳ Instance based versus model based learning

→ Supervised learning :- In this the training set you feed to the algorithms includes the desired soln called labels.



→ classification :- eg: spam filter

→ regression :- eg:- price of car, price of stock.

→ algorithms of Supervised learning are :-

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural networks

→ Unsupervised Learning :- In this the training data is unlabeled. the system tries to learn without a teacher.

→ unlabeled data

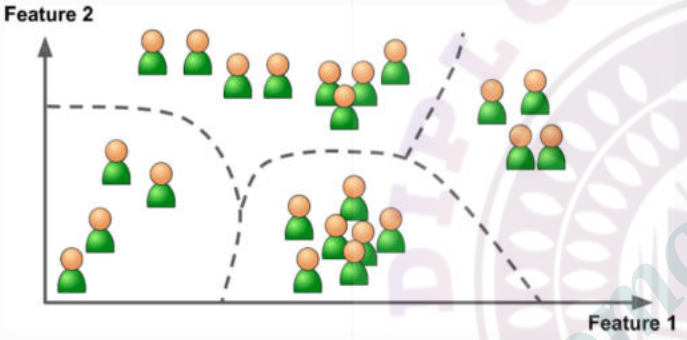
→ algorithms of unsupervised learning:-

Clustering
- K-Means
- DBSCAN
- Hierarchical Cluster Analysis (HCA)

Anomaly detection and novelty detection
- One-class SVM
- Isolation Forest

Visualization and dimensionality reduction
- Principal Component Analysis (PCA)
- Kernel PCA
- Locally-Linear Embedding (LLE)
- t-distributed Stochastic Neighbor Embedding (t-SNE)

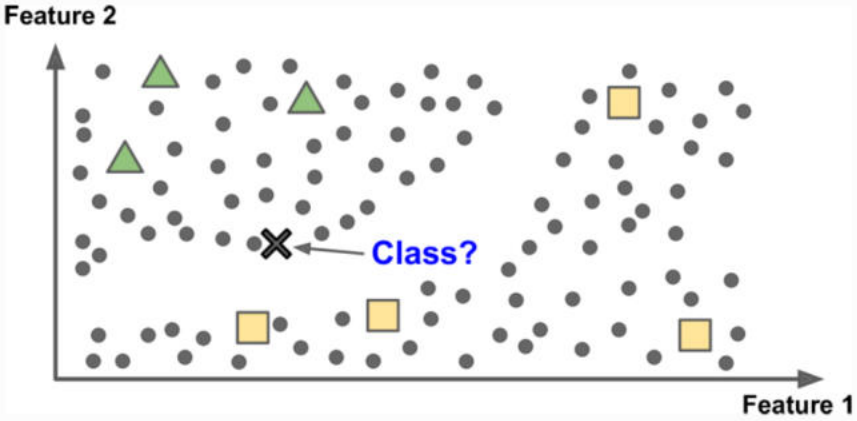
Association rule learning
- Apriori
- Eclat



→ Clustering

→ Anomaly Detection

→ Semi Supervised Learning :- Some algorithms can deal with partially labeled instances and those are called semi supervised learning.



eg:- google photos are using unsupervised learning

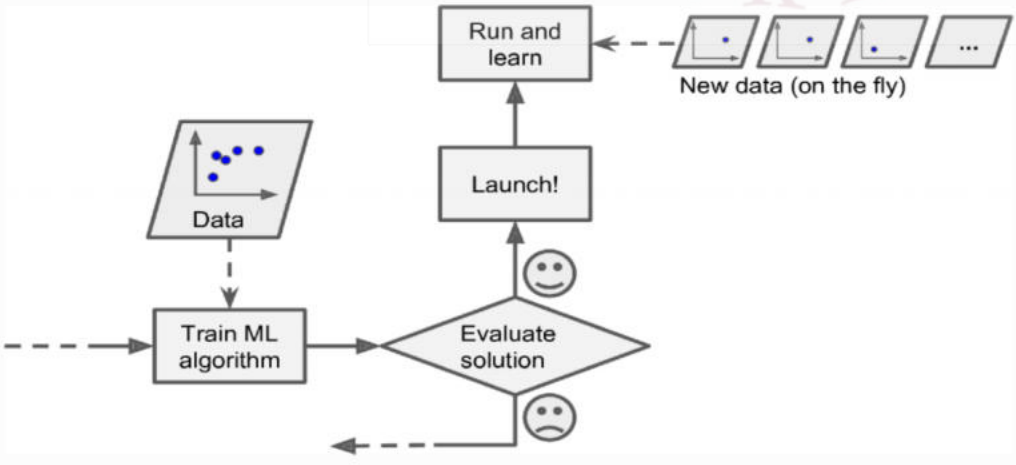
Semisupervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

Batch Learning :- also known as offline learning

In batch learning, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called offline learning.
If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one.

Online Learning :-

In online learning, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called mini-batches. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives.



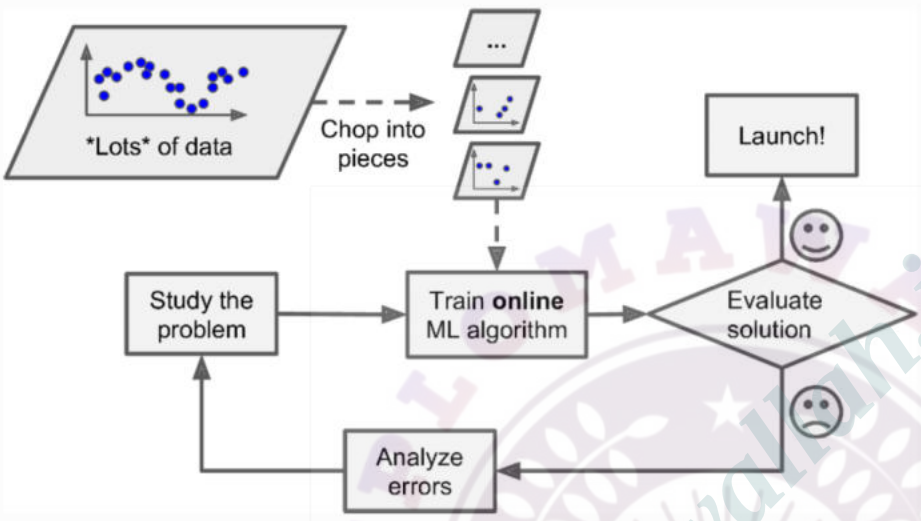
In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in

→ Outcare of online learning

↳ use for huge dataset that cannot fit in main memory

↳ use when continuous flow of data [eg: stock market]

↳ use when we have limited computing resources.



→ using online learning to handle huge datasets

→ Main Challenges of Machine Learning

(1) Insufficient Quatity of training data

(2) Non representing Training Data : it is crucial to use a training set that is representative of the cases you want to generalize to.

(3) Poor quality Data:-

Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poorquality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple of examples of when you'd want to clean up training data:

(3) Irrelevant Features:

(4) Overfitting the training data :-

Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. Here are possible solutions:

- Simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data, or by constraining the model.
- Gather more training data.
- Reduce the noise in the training data (e.g., fix data errors and remove outliers).

(5) Underfitting the training Data :-

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (e.g., reduce the regularization hyperparameter).

Note :-

The system will not perform well if your training set is too small, or if the data is not representative, is noisy, or is polluted with irrelevant features (garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit).

: Chapter-2 :

→ Some basic term that should be remembered (for better understanding of code visit sklearn documentation).

→ Pipelines :-

A sequence of data processing components is called a data pipeline. Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Components typically run asynchronously. Each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, some time later, the next component in the pipeline pulls this data and spits out its own output. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.
On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale and the overall system's performance drops

→ Performance Measure :-

A typical performance measure for regression problems is the root Mean Square Error [RMSE]. it gives an idea of how much error the system typically makes in its predi-ctions, with a higher weight for large errors.

$$RMSE(X,h) :- \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left[h(x^{(i)}) - y^{(i)}\right]^2}$$

$$MSE(X,h) :- \frac{1}{n}\sum_{i=1}^{n}\left[h(x^{(i)}) - y^{(i)}\right]^2$$

Mean squared error

n → the number of instances in the dataset

$x^{(i)}$ → is a vector of all feature values [excluding labels] of the i^{th} instances in the dataset

$y^{(i)}$ → the desired output value for that instances.

$h(x)$ → System prediction function also called a hypothesis.

X → is a matrix containg all feature values (excluding labels of all instances in the dataset.

→ RMSE, MSE, MAE all are cost function

$$MAE(X,h) :- \frac{1}{n}\sum_{i=1}^{n} |h(x^{(i)}) - y^{(i)}|$$

→ all these RMSE, MSE, MAE are ways to measure the distance b/w two vectors: the vector of predictions and the vector of target values

→ RMSE, MSE are more sensitive to outliers than the MAE.

→ We have three kind of data that which we pass to ML algo and those are

1. training data
2. testing data
3. validation data

→ for spliting training and testing data set we can use this code in python

```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

[x_train, y_train] [x_test, y_test] ← dataset ↓ size of test set

→ For converting the numpy arrays into pandas Dataframe the code will be] housing → old data

```
new_data = pd.DataFrame[data, columns = housing.columns,
                                index = housing.index]
```

→ Scikit learn provides a handy class to take care of missing values: SimpleImputer. Here is how to use it.

→ from sklearn.impute SimpleImputer
→ imputer = SimpleImputer(strategy = "median")

imputer → model variable
strategy → attribute
"median" → it can be mean, mode also

eg:- lets take we have a data set variable new_data so

→ imputer.fit(new_data)

→ imputer.statistics_] to see all median values for corresponding columns of new_data

→ X = imputer.transform(new_data)

X ↓ numpy array that filled with median value

→ Most ML algorithm prefers to work with numeric value. so for converting categorical to numeric value the sklearn provide OrdinalEncoder class.

→ from sklearn.preprocessing import OrdinalEncoder

→ Ordinal_encoder = OrdinalEncoder()

instance model variable created

eg: lets we have dataset "new_data" and in which we have 'A' as a categorical variable

→ new_data_encoded = Ordinal_encoder.fit_transform(new_data["A"])

↓

numeric variable or instance of numeric data.

→ Ordinal_encoder.categories_] to get all categori-cal class of that data

→ **One Hot Encoder :-** is a technique that we use to represent categorical variables as numerical values in a machine learning model. In this it created the binary attribute per category : one attribute equal to '1' when the category is same otherwise '0'. the new attribute are sometimes called dummy attributes. Sklearn provides a OneHotEncoder class to convert categorical values into one-hot

vectors.

→ from sklearn.preprocessing import OneHotEncoder

→ cat_encoder = OneHotEncoder()

eg:- Dataset = new_data → categorical column data

→ new_data_1hot = cat_encoder.fit_transform(new_data)

→ Notice that the output is a SciPy sparse matrix, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After onehot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the nonzero elements. You can use it mostly like a normal 2D array, but if you really want to convert it to a (dense) NumPy array, just call the toarray() method:

→ new_data_1hot.toarray()

↳
```
array([[1, 0, 0, 0, ... 0]
       [1, 0, 0, 0, ... 0]
       [                  ]])
```
} output numpy array.

→ if a categorical attribute has a large number of possible categories then one-hot-encoder will result in a large of input features. this may slow down training and degrade performance.

→ Custom Transformers:- to create custom transformers using sklearn in python you can extend the 'TransformerMixin' class and 'BaseEstimator' class from sklearn. these base classes provide the necessary framework for your custom transformer to integrate seamlessly

with sklearn pipelines and other tools.

```
import numpy as np
from sklearn.base import BaseEstimator, TransformerMixin
```

} step 1

Step 2: Define Your Custom Transformer

Create a class that inherits from BaseEstimator and TransformerMixin, and implement the required methods: __init__, fit, and transform.

Here is an example of a custom transformer that adds a specified constant to all elements of an input array:

python Copy code

```python
class AddConstantTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, constant=1):
        self.constant = constant

    def fit(self, X, y=None):
        # fit method doesn't need to do anything for this transforme
        return self

    def transform(self, X):
        # Add the constant to all elements of X
        return X + self.constant
```


→ You can now use this transformer in a scikit-learn pipeline or directly on data.

→ Feature Scaling :- One of the most important transformations you need to apply to your data is feature scaling. ML algo don't perform well when the input numerical attributes have different scales. Note that scaling the target values is generally not required.

→ two common ways to get all attr. to have same scale min-max scaling and standardization.

→ min-max scaling :- Values are shifted and rescaled so

that they end up ranging from 0 to 1.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

x_{min} → min value of x

x_{max} → max value of x

→ Standardization :- rescales data to have a mean(μ) of '0' and standard deviation (σ) of 1 (unit variance).

$$X_{changed} = \frac{X - \mu}{\sigma}$$

generally standardization is recommended. and it does not bound the values in specific range. sklearn provides a transformer called StandardScaler for standardization.

→ Transformation pipelines :- there are many data transform-ation steps that need to be executed in the right order. for that pipeline become very important. sklearn provide Pipeline class to help with such sequences of transformations.

→ from sklearn.pipeline import Pipeline

→ from sklearn.preprocessing import StandardScaler

pipline creation for transformation →

```
→ pipe_line = Pipeline([
        ("std_scaler", StandardScaler()),
        ("                              ),
    ])
```

When you call the pipeline's fit() method, it calls fit_transform() sequentially on all transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it calls the fit() method.

→ Column transformer :- we have handled the categorical columns and the numerical columns seperately. it would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformers to each column.

→ sklearn have 'ColumnTransformer' class.

```
from sklearn.compose import ColumnTransformer

full_pipline = ColumnTransformer([
    ("num", num_pipline, num_data),
    ("categorical", cat_pipeline, cat_data)
])
```

→ it applies each transformer to the appropriate columns and concatenates the output along second axis (the transformers must return same number of rows).

→ Cross validation :-

Cross-validation is a technique used to assess the performance of a machine learning model. It involves partitioning the data into subsets, training the model on some subsets (training set), and evaluating it on the remaining subsets (validation set). This process is repeated multiple times to ensure the model's performance is consistent and not dependent on a particular split of the data.

In scikit-learn (sklearn), there are several cross-validation techniques available, including K-Fold, Stratified K-Fold, Leave-One-Out, and more.

K-fold:-

How K-Fold Cross-Validation Works

→ Data Splitting:

- The dataset is divided into k equal-sized subsets or "folds".
- For example, if k=5, the dataset is divided into 5 folds.

→ Training and Validation:

- The model is trained and validated k times.
- In each iteration, one of the k folds is used as the validation set, and the remaining k-1 folds are used as the training set.
- This process is repeated k times, with each fold being used exactly once as the validation set.

→ Performance Estimation:

- The performance metric (e.g., accuracy, precision, recall) is computed for each iteration. The final performance estimate is obtained by averaging the performance metrics from all k iterations.

→ advantage:
- ① More reliable performance estimate
- ② Efficient use of Data
- ③ Reduce overfitting.

Stratified k fold:-

Stratified K-Fold Cross-Validation is an extension of K-Fold Cross-Validation that ensures each fold is representative of the entire dataset, particularly with regard to the distribution of the target variable. This is particularly useful when dealing with imbalanced datasets where some classes are underrepresented.

How Stratified K-Fold Cross-Validation Works

→ Data Splitting:

- The dataset is divided into k folds, similar to K-Fold Cross-Validation.
- However, the splitting is done in such a way that each fold maintains the same proportion of classes as in the entire dataset.

→ Training and Validation:

- The process is similar to K-Fold Cross-Validation.
- In each iteration, one of the k folds is used as the validation set, and the remaining k-1 folds are used as the training set.

- This ensures that each fold is a good representative of the entire dataset, especially in terms of class distribution.

→ Performance Estimation:

- The performance metric (e.g., accuracy, precision, recall) is computed for each iteration. The final performance estimate is obtained by averaging the performance metrics from all k iterations.

advantages :
① maintains class Distribution
② More reliable Performance estimate

→ leave-one-out :-

Leave-One-Out Cross-Validation (LOOCV) is a special case of K-Fold Cross-Validation where the number of folds equals the number of data points in the dataset. In LOOCV, each data point is used once as a validation set while the rest of the data points are used as the training set. This process is repeated for each data point.

→ Hyperparameter tuning :-

→ For finding best parameters for model or fine tune the model we have to use technique for that and those are ① grid search. ② Randomized Search

① grid search :-

All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values.

For example, the following code searches for the best combination of hyperparameter values for the RandomForestRegressor:

```
from sklearn.model_selection import GridSearchCV
param_grid = [
        {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
        {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]
forest_reg = RandomForestRegressor()
```

→ for this we have 3×4 = 12 combin
→ for this 2×3 = 6
→ total 12+6 = 18 combinatio

```
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

→ this will run 5 types for validation

→ total round of training
⇒ 18 × 5 = 90

housing_prepared → x_train
housing_labels → y_train

→ after successfull run you will get best combination of parameters.

② Randomized search :-

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but when the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead. This class can be used in much the same way as the GridSearchCV class, but instead of trying out all possible combinations, it evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration. This approach has two main benefits:

- If you let the randomized search run for, say, 1,000 iterations, this approach will explore 1,000 different values for each hyperparameter (instead of just a few values per hyperparameter with the grid search approach).
- Simply by setting the number of iterations, you have more control over the computing budget you want to allocate to hyperparameter search.

Chapter 3 :-

→ One of the popular dataset that we will use in this chapter is 'MNIST'. Sklearn provides many helper function to download popular datasets. MNIST is one of them.

→ from sklearn.datasets import fetch_openml
→ mnist = fetch_openml('mnist_784') } → pandas dataframe
→ mnist.keys()
↳ dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details', 'categories', 'url'])

→ Performance Measures :-

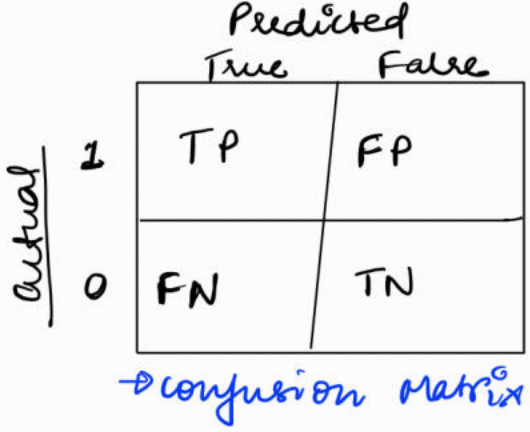
→ Accuracy is generally not preferred performance measure for classifiers, especially when you are dealing with skewed datasets [when some classes are much more frequent than others].

→ Confusion Matrix :-

A much better way to evaluate the performance of a classifier is to look at the confusion matrix. The general idea is to count the number of times instances of class A are classified as class B. For example, to know the number of times the classifier confused images of 5s with 3s, you would look in the fifth row and third column of the confusion matrix.

→ To use of confusion matrix just pass it the target classes and predicted classes.

→ from sklearn.metrics import confusion_matrix
→ confusion_matrix[targeted_class, pred_class]



→ confusion matrix

TP → True Positive [when we correctly classify positive class]

FP → False positive [when we unable to classify positive class]

FN :- False Negative [when we predict positive but it actually is Negative]

TN :- True Negative [when we predict negative and it actual is Negative]

→ Precision is the ratio of correctly predicted positive observation to the total predicted positive.

→ confusion matrix have lots of information but sometime you may prefer a more concise matric. An interesting one to look at is the accuracy of the positive predictions; this is called precision of the classifier.

$$\text{Precision} = \frac{TP}{TP + FP}$$

TP is the number of true positives, and FP is the number of false positives.

→ A trivial way to have perfect precision is to make one single positive prediction and ensure it is correct (precision = 1/1 = 100%). But this would not be very useful, since the classifier would ignore all but one positive instance. So precision is typically used along with another metric named recall, also called sensitivity or the true positive rate (TPR): this is the ratio of positive instances that are correctly detected by the classifier (Equation 3-2).

Equation 3-2. Recall

$$\text{Recall} = \frac{TP}{TP + FN}$$

→ Recall is the ratio of correctly predicted positive observation to all observation in the actual class.

FN is, of course, the number of false negatives.

→ from sklearn.metrics import precision_score, recall_score

→ It is often convenient to combine precision and recall into a single metric called the F_1 score, in particular if you need a simple way to compare two classifiers. The F_1 score is the harmonic mean of precision and recall. Where as the regular mean treats all values equally, the harmonic mean gives much more weight to low value. As a result, the classifier only get a high F_1 score if both recall and precision are high.

$$F_1 \text{ score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{Precision} \times \text{recall}}{\text{Precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

→ ROC curve :- [Receiver operating characteristic]

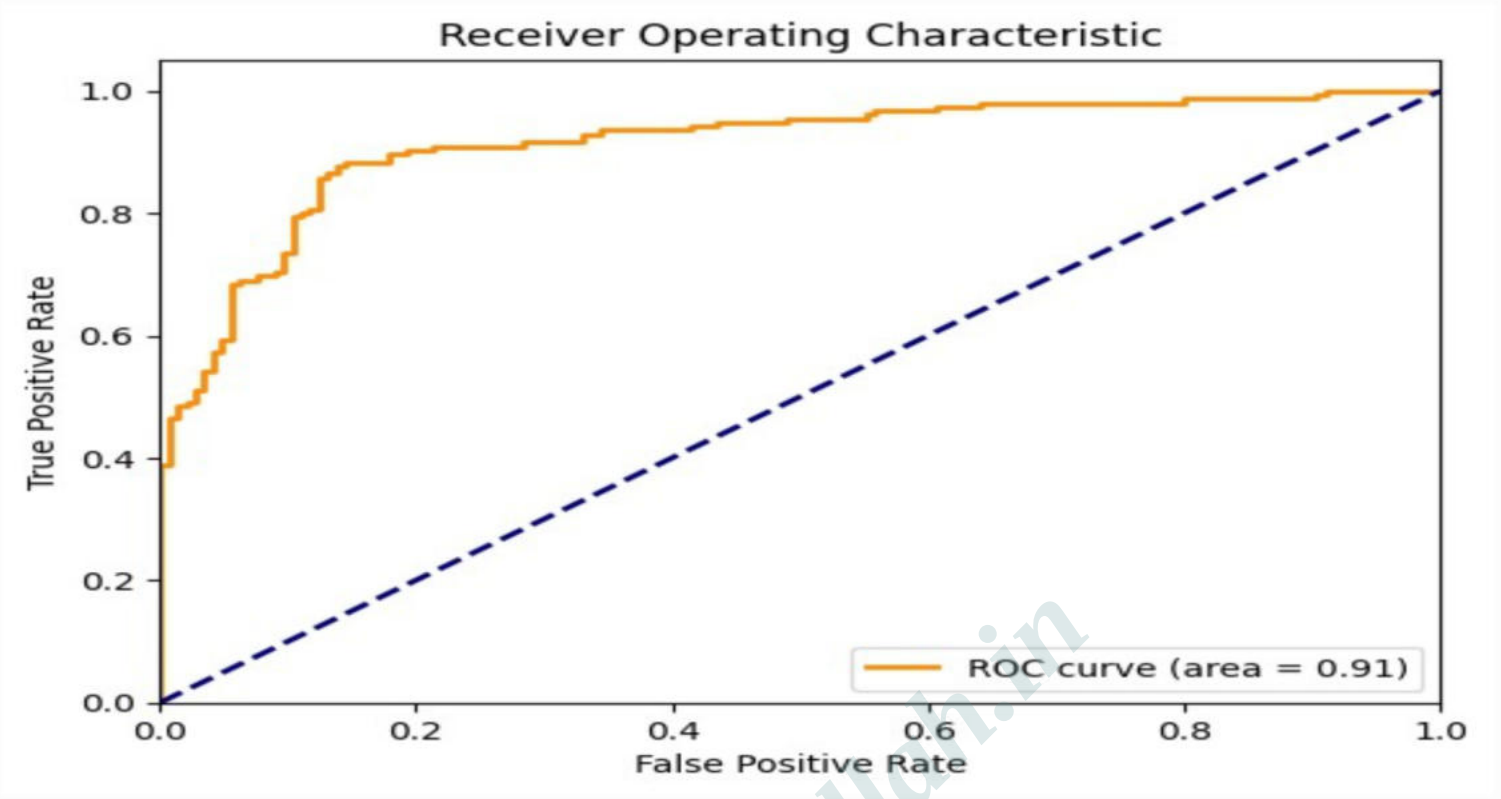
↳ it is common tool used with binary classifiers.

→ the ROC curve plots the True Positive rate [TPR] [another name for recall) against false positive rate [FPR]. The FPR is the ratio of negative instance that are incorrectly classified as positive. it equal to 1 - True negative rate [TNR], which is the ratio of negative instances that are correctly classified as negative. TNR is also called specificity.

$$FPR = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{TP + FN}$$

→ plot TPR vs FPR : Plot the TPR [y-axis] against the FPR [x-axis]



→ ROC curve

→ Multiclass classification :. binary classifiers distinguish between two classes, multiclass classifiers (also called multinomial classifiers) can distinguish b/w more than two classes.

→ some algorithms [such as SGD classifiers, Random forest classifiers, and naive Bayes classifiers] are capable of handling multiple classes natively. Others such as Logistic Regression or Support Vector Machine] are strictly binary classifiers.

→

there are various strategies that you can use to perform multiclass classification with multiple binary classifiers. One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on). Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score. This is called the one-versus-the-rest (OvR)

strategy (also called one-versus-all).

Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. This is called the one-versus-one (OvO) strategy. If there are N classes, you need to train N × (N – 1) / 2 classifiers. For the MNIST problem, this means training 45 binary classifiers! When you want to classify an image, you have to run the image through all 45 classifiers and see which class wins the most duels. The main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish.

imp

Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set. For these algorithms OvO is preferred because it is faster to train many classifiers on small training sets than to train few classifiers on large training sets. For most binary classification algorithms, however, OvR is preferred.

Note:- Scaling input can increases accuracy.

→ Error Analysis :-

You'd explore data preparation options, try out multiple models (shortlisting the best ones and fine-tuning their hyperparameters using GridSearchCV), and automate as much as possible. Here, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

→ First, look at the confusion matrix, you need to make prediction using the cross validation, then call the confusion matrix.

eg: → y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)

sgd classifier model

data

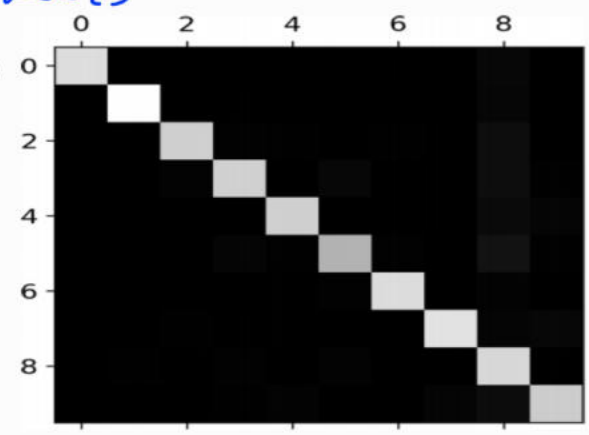
→ conf_m = confusion_matrix(y_train, y_train_pred)

→ conf_m

```
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

That's a lot of numbers. It's often more convenient to look at an image representation of the confusion matrix, using Matplotlib's matshow() function:

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



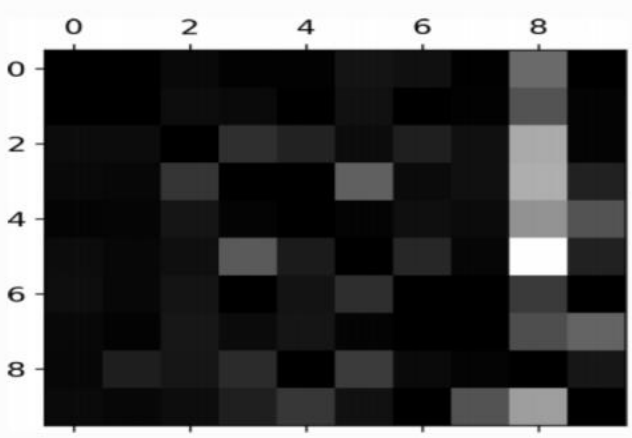
This confusion matrix looks pretty good, since most images are on the main diagonal, which means that they were classified correctly. The 5s look slightly darker than the other digits, which could mean that there are fewer images of 5s in the dataset or that the classifier does not perform as well on 5s as on other digits. In fact, you can verify that both are the case.

Let's focus the plot on the errors. First, you need to divide each value in the confusion matrix by the number of images in the corresponding class so that you can compare error rates instead of absolute numbers of errors (which would make abundant classes look unfairly bad):

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

Fill the diagonal with zeros to keep only the errors, and plot the result:

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



you can clearly see the kinds of errors the classifier makes.

↳ Remember that rows represent actual classes, while colm represent predicted classes. the colum for class 8 is quite bright, which tells you that many images get misclassified as 8s. However the row for class 8 is not that bad, telling you that actual 8s in general get properly classified as 8s.

→

Analyzing the confusion matrix often gives you insights into ways to improve your classifier. Looking at this plot, it seems that your efforts should be spent on reducing the false 8s. For example, you could try to gather more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them from real 8s.

→ Multilabel classification :-

Until now each instance has always been assigned to just one class. In some cases you may want your classifier to output multiple classes for each instance. Consider a facerecognition classifier: what should it do if it recognizes several people in the same picture? It should attach one tag per person it recognizes. Say the classifier has been trained to recognize three faces, Alice, Bob, and Charlie. Then when the classifier is shown a picture of Alice and Charlie, it should output [1, 0, 1] (meaning "Alice yes, Bob no, Charlie yes"). Such a classification system that outputs multiple binary tags is called a multilabel classification system.

→ Multioutput classification :- it can be also recognised as multioutput - multiclass classification. it is simply a generalization of multilabel classification where each level can be multiclass.

→

To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). It is thus an example of a multioutput classification system.

-: chapter - 4 :-

→ However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameter for your task. Understanding what's under the hood will also help you debug issues and perform error analysis more efficiently.

→ Linear Regression :-

$$h_{(\theta_0, \theta_1)}(x) = \theta_0 + \theta_1 x$$

↳ this model is just a linear function of the input feature x. θ_0 and θ_1 are model parameters.

→ More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term (also called the intercept term).

$$\hat{y} = h(x)$$

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

↳ Linear Regression model prediction in this equation:

- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots \theta_n$).

→ This can be written much more concisely using a vectorized form,

$$\hat{y} = h_\theta(x) = \theta \cdot x$$

↳ linear regression mode prediction (vectorized form)

→ In ML, vectors are often represented as column vectors, which are 2D arrays with a single column. if θ and x are column vectors, then the prediction is $\hat{y} = \theta^T x$, where θ^T is the transpose of θ [a row vector instead of a column vector] and $\theta^T x$ is the matrix multiplication of θ^T and x. It is of course the same prediction. except that it is now represented as a single cell matrix rather than a scalar value.

→

to train a Linear Regression model, we need to find the value of θ that minimizes the RMSE. In practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root).

→ the MSE of a linear Regression hypothesis h_θ on a training set X is calculated using.

$$MSE(\theta) = \frac{1}{m} \sum_{i=1}^{m} [\underbrace{\theta^T x^{(i)}}_{\text{predicted value}} - \underbrace{y^{(i)}}_{\text{actual value}}]^2$$

→ Normal equation :-

To find the value of θ that minimizes the cost function, there is a closed-form solution —in other words, a mathematical equation that gives the result directly. This is called the Normal Equation.

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

- θ^T is the value of θ that minimizes the cost funct.
- y is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

→ performing Linear Regression using Sklearn is simple

```
→ from sklearn.linear_model import LinearRegression
→ lin_reg = LinearRegression()
→ lin_reg.fit(x_train, y_train)
→ lin_reg.intercept_ , lin_reg.coef_
```

y_train → target data

x_train → Input feature

lin_reg.intercept_ → Intercept = b

lin_reg.coef_ → slope = m

for $y = mx + b$

↳ I am adding slides which show how linear regression work and what are the math concepts behind it.

SLOPE

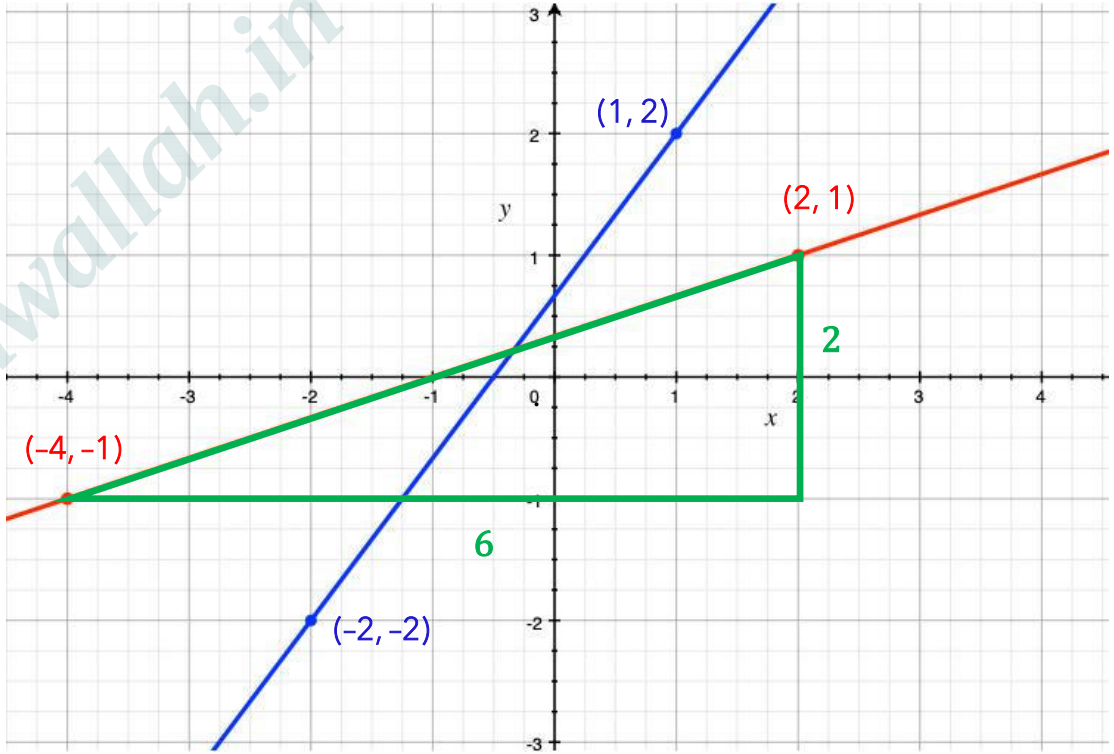
$P_1 = [4, 2]$, $P_2 = [-2, -2]$

$$m = \frac{2-(-2)}{1-(-2)} = \frac{4}{3}$$

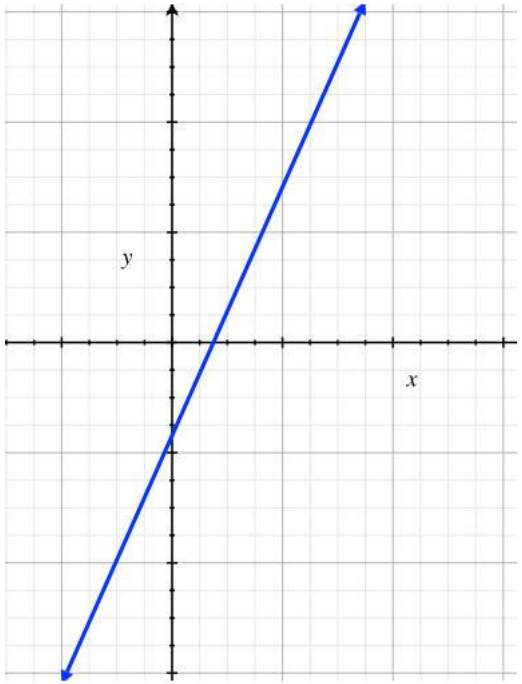
$$\text{Slope} = \frac{\text{Vertical change}}{\text{Horizontal change}}$$

$$m = \frac{1-(-1)}{2-(-4)} = \frac{2}{6} = \frac{1}{3}$$

$P_1[2, 1]$, $P_2[-4, -1]$

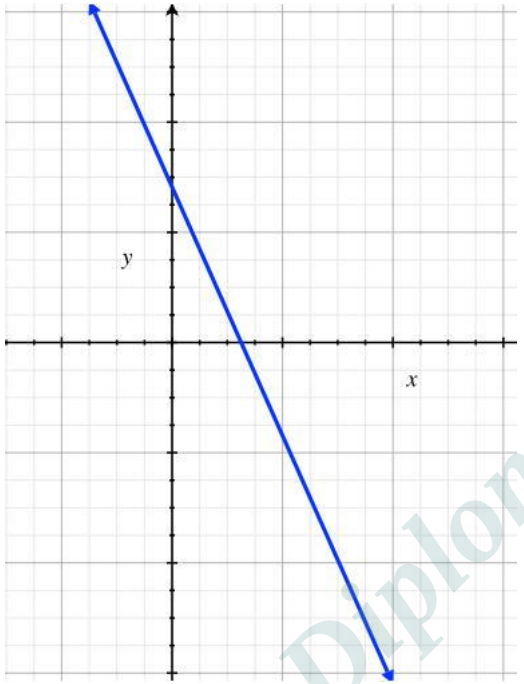


Positive slope



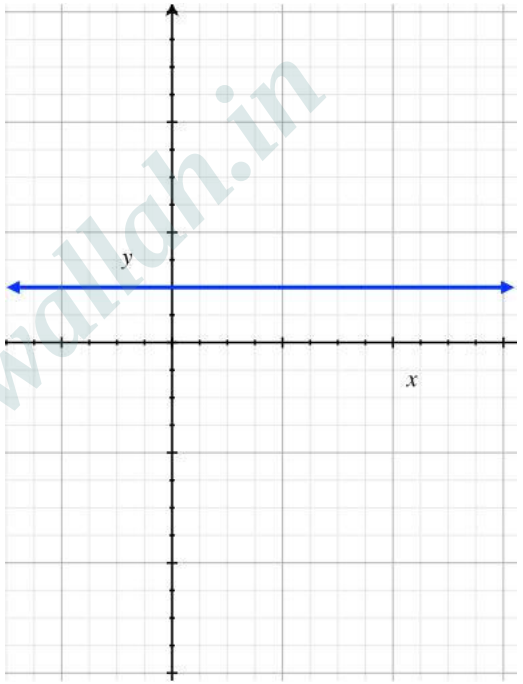
$y = mx + b$
$m > 0$

Negative slope



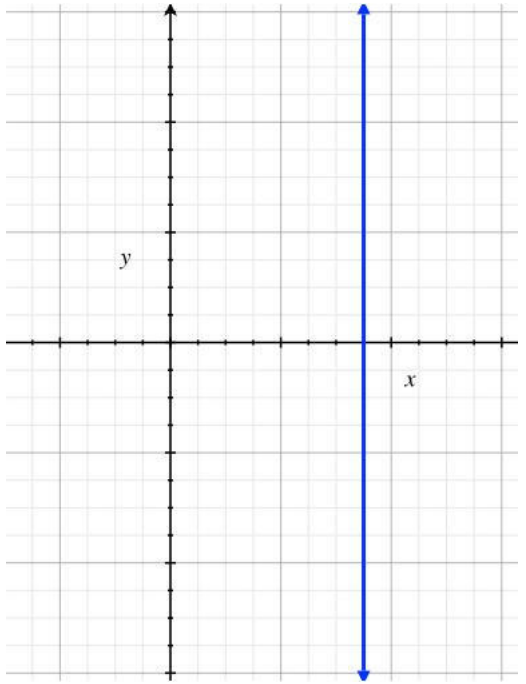
$y = mx + b$
$m < 0$

Zero slope



$y = b$

Undefined slope



$x = a$

LINEAR EQUATIONS

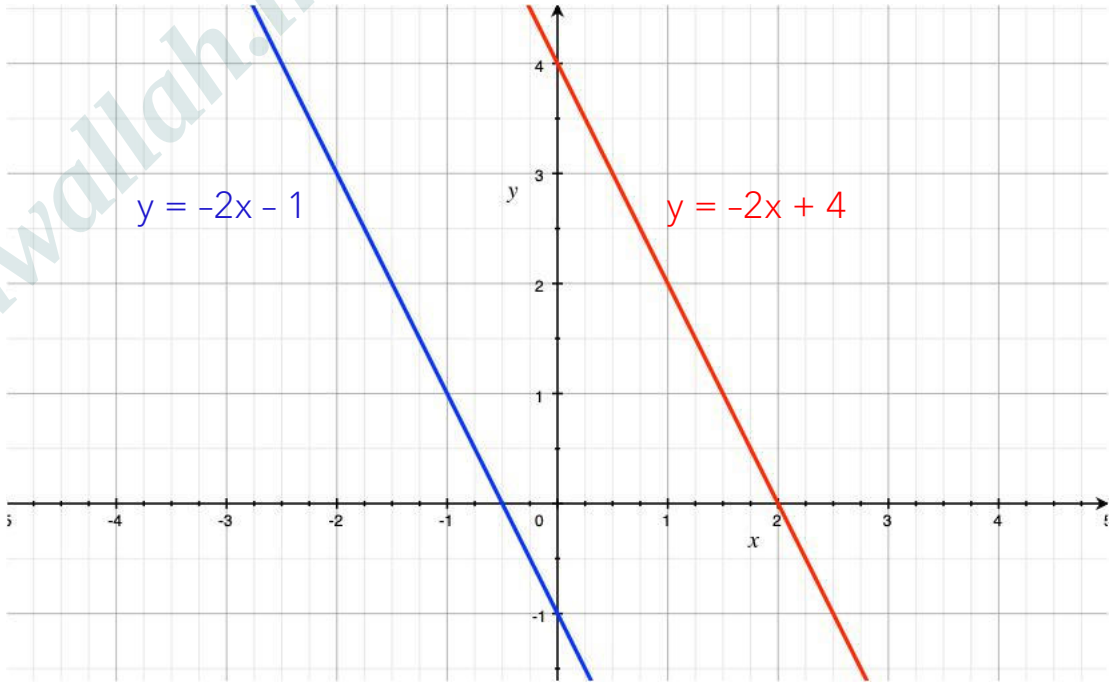
- Slope-intercept form: $y = mx + b$

- Standard form: $Ax + By = C$, where A and B are not both 0.

PARALLEL LINES

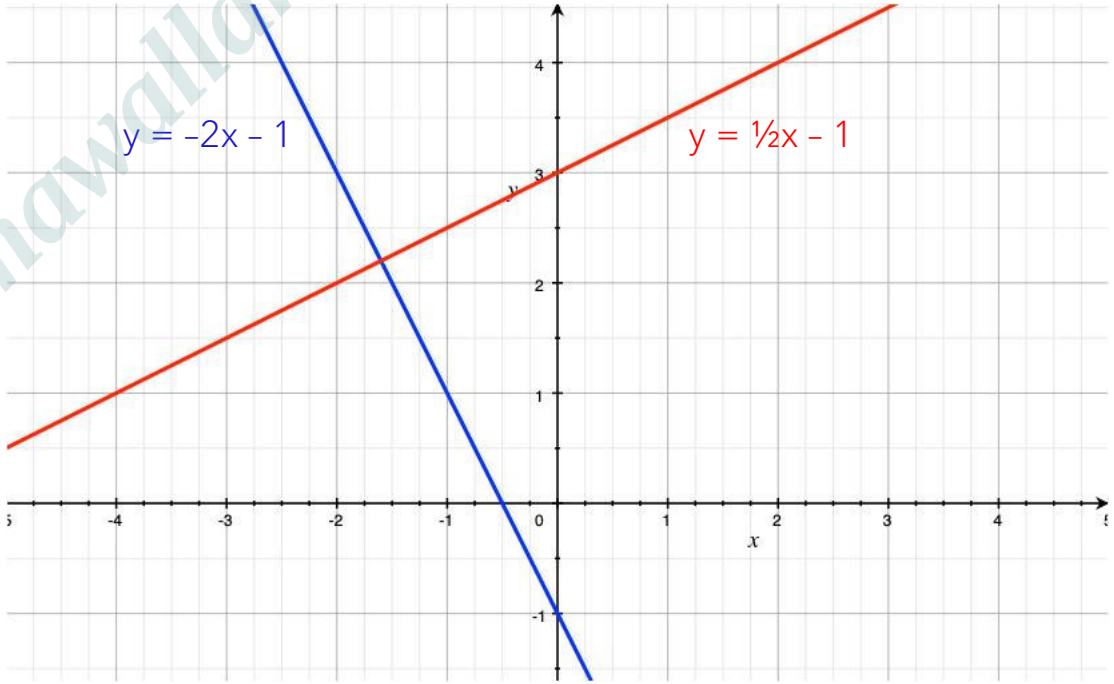
Same slope (positive, negative, zero)

or both vertical



PERPENDICULAR LINES

- Product of slopes is –1, one is vertical and the other horizontal lines



EXAMPLE

Passes through (–2, 6) and parallel to

$y = \frac{2}{3}x - \frac{5}{3}$

$$m = \frac{2}{3}$$

$$y = \frac{2}{3}x + b$$

$$\left(\frac{2}{3}\right)(-2) + b = 6$$

$$-\frac{4}{3} + b = 6$$

$$b = \frac{4}{3} + \frac{18}{3} = \frac{22}{3}$$

$$\boldsymbol{y = \frac{2}{3}x + \frac{22}{3}}$$

Passes through (–2, 6) and perpendicular to

$y = \frac{2}{3}x - \frac{5}{3}$

$$m = -\frac{3}{2}$$

$$y = -\frac{3}{2}x + b$$

$$\left(-\frac{3}{2}\right)(-2) + b = 6$$

$$3 + b = 6$$

$$b = 6 - 3 = 3$$

$$\boldsymbol{y = -\frac{3}{2}x + 3}$$

BREAK-EVEN ANALYSIS

- Linear cost function, $C(x) = mx + b$

  m is the marginal cost, b is the fixed cost, x is the number of items produced

- Revenue function, $R(x) = px$

  p is the price per unit and x is the number of units sold

- Profit function, $P(x) = R(x) - C(x)$

- Break-even point: The point where $R(x) = C(x)$

  Occurs where the two lines intersect

EXAMPLE

The cost to produce x widgets is given by C(x) = 105x + 6000 and each widget sells for $250. Determine the break-even quantity.

Solution:

R(x) = 250x

250x = 105x + 6000

145x = 6000

x ~ 41.38

R(41) = 250(41) = 10,250

and

C(41) = 105(41) + 6000 = 10,305

R(42) = 250(42) = 10,500

and

C(42) = 105(42) + 6000 = 10,410

Note: Selling 41 widgets is not enough.

The breakeven quantity is 42 widgets.

LEAST SQUARES LINE

Minimize the sum of the squares of the vertical distances from the data points to the line

$$y = mx + b$$

Data points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

Imp

$$m = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

and

Imp

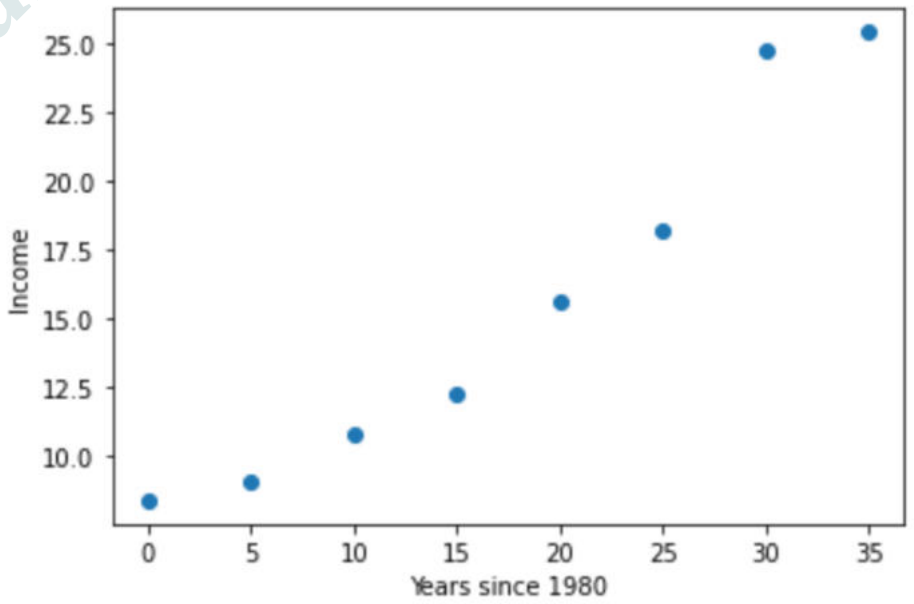
$$b = \frac{\sum y - m(\sum x)}{n}$$

SCATTERPLOT

Income from side business

Year	Income
1980	8,414
1985	9,124
1990	10,806
1995	12,321
2000	15,638
2005	18,242
2010	24,792
2015	25,436

Let x represent the number of years since 1980 and y represent the income in thousands of dollars



LEAST SQUARES CALCULATIONS

Year	Income
1980	8,414
1985	9,124
1990	10,806
1995	12,321
2000	15,638
2005	18,242
2010	24,792
2015	25,436

Least Squares Calculations				
x	y	xy	x^2	y^2
0	8.414	0	0	70.795396
5	9.124	45.62	25	83.247376
10	10.806	108.06	100	116.769636
15	12.321	184.815	225	151.807041
20	15.638	312.76	400	244.547044
25	18.242	456.05	625	332.770564
30	24.792	743.76	900	614.643264
35	25.436	890.26	1225	646.990096
140	**124.773**	**2741.325**	**3500**	**2261.57042**

$$m = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

$$= \frac{8(2741.325) - (140)(124.773)}{8(3500) - (140)^2}$$

$$= 0.5312$$

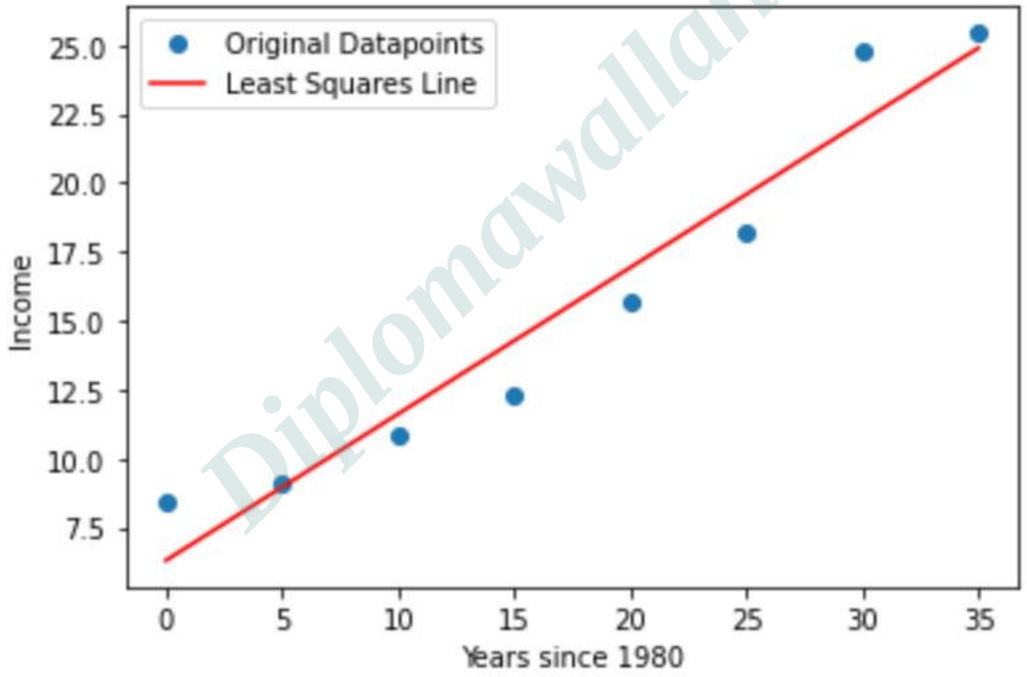
$$b = \frac{\sum y - m(\sum x)}{n}$$

$$= \frac{124.773 - (0.5312)(140)}{8}$$

$$= 6.3$$

$$\boldsymbol{y = 0.5312x + 6.3}$$

GRAPH OF LEAST SQUARES LINE



LEAST SQUARES LINE PREDICTION

Use the least squares line $y = 0.5312x + 6.3$ to predict income in 2025

Recall, x is the number of years since 1980, so $x = 45$ corresponds to 2025

$$y = (0.5312)(45) + 6.3 = 30.204$$

Since y is in thousands of dollars, the predicted income in 2025 is $30,204

CORRELATION COEFFICIENT

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{n(\sum x^2) - (\sum x)^2} \cdot \sqrt{n(\sum y^2) - (\sum y)^2}}$$

$$= \frac{8(2741.325)(140)(124.773)}{\sqrt{8(3500) - (140)^2} \cdot \sqrt{8(2261.57042) - (124.773)^2}}$$

$$= \mathbf{0.9691}$$

Least Squares Calculations				
x	y	xy	x^2	y^2
0	8.414	0	0	70.795396
5	9.124	45.62	25	83.247376
10	10.806	108.06	100	116.769636
15	12.321	184.815	225	151.807041
20	15.638	312.76	400	244.547044
25	18.242	456.05	625	332.770564
30	24.792	743.76	900	614.643264
35	25.436	890.26	1225	646.990096
140	**124.773**	**2741.325**	**3500**	**2261.57042**

Prediction | Forecast

Python code :-

```
from scipy import stats
x = [0,5,10,15,20,25,30,35]
y = [8.414,9.124,10.806,12.321,15.638,18.242,24.792,25.436]
slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)
print("slope = ", slope)
print("intercept = ", intercept)
print("correlation coefficient = ", r_value)
```

```
slope =  0.5312357142857143
intercept =  6.300000000000001
correlation coefficient =  0.969080175464345 9
```

AVERAGE RATE OF CHANGE

The average rate of change of $f(x)$ with respect to x as x changes from a to b is

$$\frac{f(b)-f(a)}{b-a}$$

Based on population projections for 2000 to 2050, the projected Hispanic population (in millions) for a certain country can be modeled by the exponential function

$$H(t) = 37.791(1.021)^t$$

where $t = 0$ corresponds to 2000 and $0 \le t \le 50$. Use H to estimate the average rate of change in the Hispanic population from 2000 to 2010.

The years 2000 and 2010 correspond to $t = 0$ and $t = 10$, respectively

$$\frac{H(10)-H(0)}{10-0} = \frac{37.791(1.021)^{10} - 37.791(1.021)^0}{10}$$

$$\approx \frac{8.73}{10} = 0.873$$

Never round until the last step

Tip: Use technology

```
(37.791*1.021**10-37.791*1.021**0)/10
0.8729653294860398
```

Based on this model, the Hispanic population increased at an average rate of approximately 873,000 people per year between 2000 and 2010

INSTANTANEOUS RATE OF CHANGE

Suppose a car is stopped at a traffic light. When the light turns green, the car begins to move along a straight road. Assume that the distance traveled by the car is given by $s(t) = 3t^2$, for $0 \le t \le 15$ where t is time in seconds and $s(t)$ is distance traveled in feet.

How do we find the exact velocity of the car at say, $t = 10$?

Interval	Average velocity
$t = 10$ to $t = 10.1$	$\frac{s(10.1) - s(10)}{10.1 - 10} = \frac{306.03 - 300}{0.1} = 60.3$
$t = 10$ to $t = 10.01$	$\frac{s(10.01) - s(10)}{10.01 - 10} = \frac{300.6003 - 300}{0.01} = 60.03$
$t = 10$ to $t = 10.001$	$\frac{s(10.001) - s(10)}{10.001 - 10} = \frac{300.060003 - 300}{0.001} = 60.003$

Table suggests that the velocity at $t = 10$ is 60 ft/sec.

Velocity represents both how fast something is moving and its direction, so ***velocity can be negative.***

Consider the following where h is small but not 0

$$\frac{s(10+h) - s(10)}{(10+h) - 10} = \frac{s(10+h) - s(10)}{h}$$

$$\frac{s(10+h) - s(10)}{h} = \frac{3(10+h)^2 - 3(10)^2}{h}$$

$$= \frac{3(100 + 20h + h^2) - 300}{h}$$

$$= \frac{300 + 60h + 3h^2 - 300}{h}$$

$$= \frac{60h + 3h^2}{h} = \frac{h(60 + 3h)}{h} = 60 + 3h$$

$$\lim_{h \to 0} \frac{s(10+h) - s(10)}{h} = \lim_{h \to 0} (60 + 3h) = 60 \text{ ft/sec}$$

INSTANTANEOUS RATE OF CHANGE

The instantaneous rate of change for a function f when $x = a$ is

$$\lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

provided this limit exists

Difference Quotient

$$\frac{f(a+h) - f(a)}{h}$$

Alternate Form

The instantaneous rate of change for a function f when $x = a$ can be written as

$$\lim_{b \to a} \frac{f(b) - f(a)}{b - a}$$

provided this limit exists

EXAMPLE

Suppose the total profit in hundreds of dollars from selling x items is given by $P(x) = 2x^2 - 5x + 6$. Find and interpret the following:

(a) The average rate of change of profit from $x = 2$ to $x = 4$
(b) The average rate of change of profit from $x = 2$ to $x = 3$
(c) The instantaneous rate of change of profit with respect to the number produced when $x = 2$

$$\frac{P(4) - P(2)}{4 - 2} = \frac{(2(4)^2 - 5(4) + 6) - (2(2)^2 - 5(2) + 6)}{2}$$

$$= \frac{18 - 4}{2} = 7$$

The average rate of change of profit from $x = 2$ to $x = 4$ is $\$700$ per item

$$\frac{P(3) - P(2)}{3 - 2} = \frac{(2(3)^2 - 5(3) + 6) - (2(2)^2 - 5(2) + 6)}{1}$$

$$= 9 - 4 = 5$$

The average rate of change of profit from $x = 2$ to $x = 3$ is $\$500$ per item

$$\lim_{h \to 0} \frac{P(2 + h) - P(2)}{h} = \lim_{h \to 0} \frac{(2(2 + h)^2 - 5(2 + h) + 6) - 4}{h}$$

$$= \lim_{h \to 0} \frac{(8 + 8h + 2h^2 - 10 - 5h + 6) - 4}{h}$$

$$= \lim_{h \to 0} \frac{2h^2 + 3h}{h}$$

$$= \lim_{h \to 0} (2h + 3) = 3$$

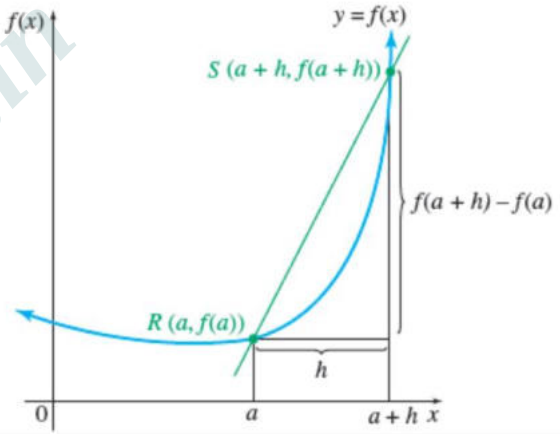
The instantaneous rate of change of profit with respect to the number of items produced when $x = 2$ is $\$300$ per item

SECANT AND TANGENT LINES

The slope of the secant line of the graph of $y = f(x)$ containing the points $(a, f(a))$ and $(a + h, f(a + h))$ is given by

$$\frac{f(a+h) - f(a)}{h}$$

Slope of secant line = average rate of change

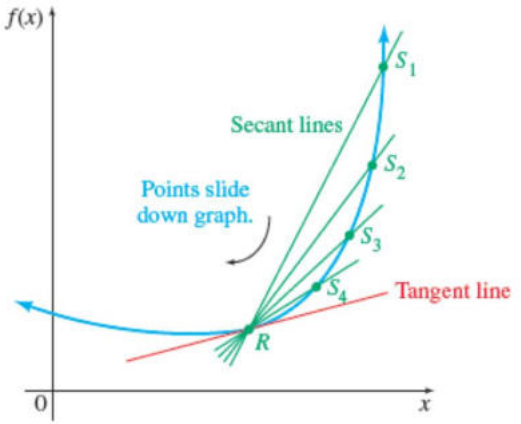


The slope of the tangent line of the graph of $y = f(x)$ at the point $(a, f(a))$ is given by

$$\lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

provided this limit exists. If this limit does not exist, then there is no tangent at the point.

Slope of tangent line = instantaneous rate of change



DEFINITION OF THE DERIVATIVE

The derivative of the function f at x is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

The function $f'(x)$ represents the instantaneous rate of change of $y = f(x)$ with respect to x

The function $f'(x)$ represents the slope of the graph at any point x

If $f'(x)$ is evaluated at the point $x = a$, then it represents the slope of the curve, or the slope of the tangent line at that point

APPLICATIONS OF DERIVATIVES

- Rate of Change of Quantities
- Increasing and Decreasing Functions
- Maxima and Minima

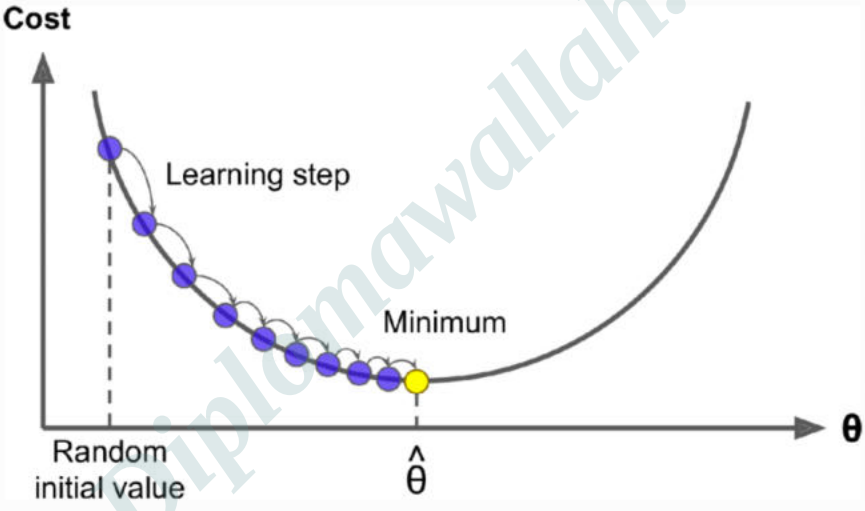
↳ all these math is important to understand the rest of the part of this chapter.

→ Gradient Descent :-

- Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.

- Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient of the error function with regard to the parameter vector θ, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

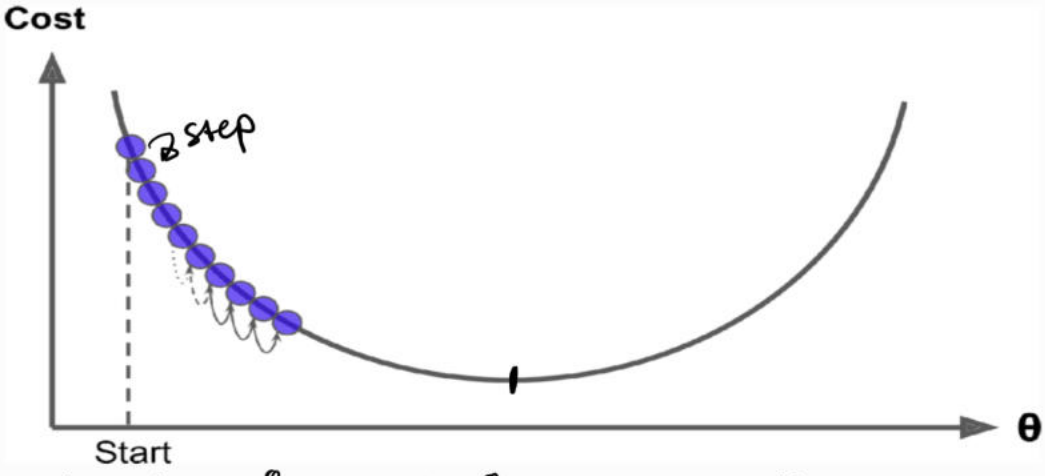
- Concretely, you start by filling θ with random values (this is called random initialization). Then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum



↳ In this depiction of gradient Descent, the model parameters are initialized randomly and get tweaked to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually gets smaller as the parameters approach the minimum.

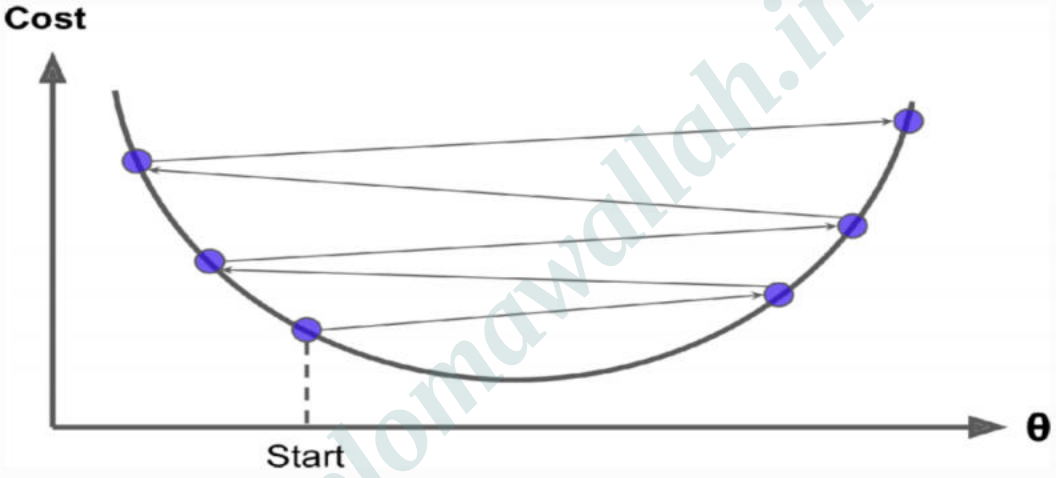
→

An important parameter in Gradient Descent is the size of the steps, determined by the learning rate hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time.



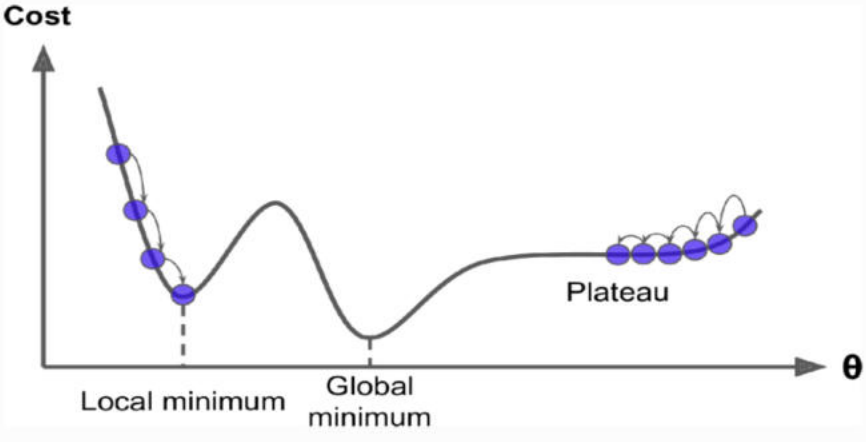
↳ learning rate (step) is too small

On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution .



↳ learning rate (step) is too large

Finally, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult. Figure shows the two main challenges with Gradient Descent. If the random initialization starts the algorithm on the left, then it will converge to a local minimum, which is not as good as the global minimum. If it starts on the right, then it will take a very long time to cross the plateau. And if you stop too early, you will never reach the global minimum.



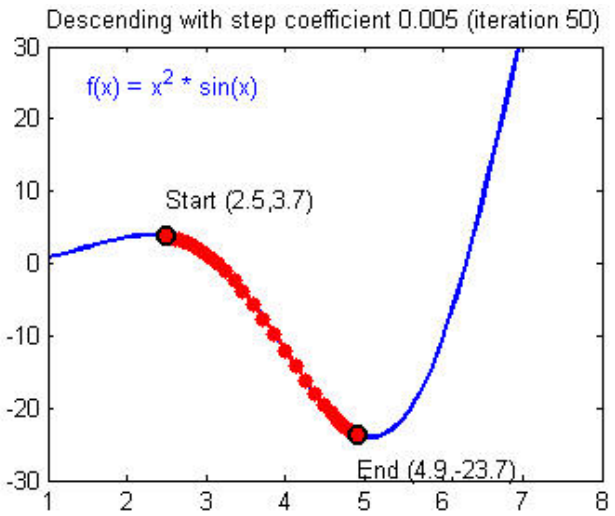
↳ gradient descent pitfalls.

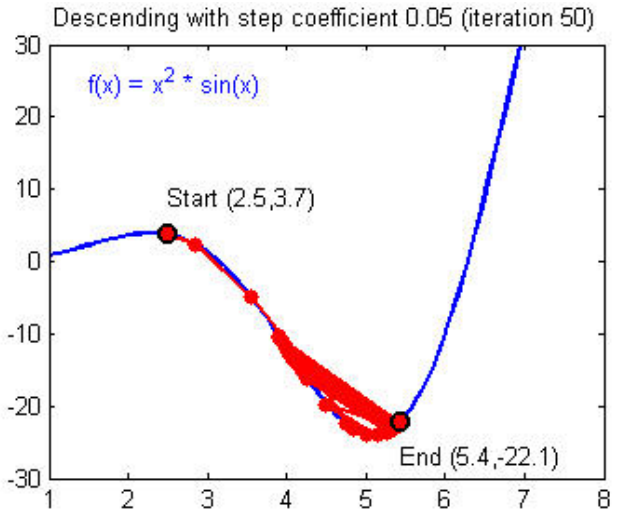
VISUALIZING GD : LEARNING RATE AND LOSS FUNCTION

$$Loss = \frac{\sum_{i=1}^{m} (Prediction_i - Actual_i)^2}{2 \times m}$$

Target: Find optimal model parameters to minimize the Loss

$$W_{new} = W_{old} - \eta \frac{d}{dW} Loss(W_{old})$$





Fortunately, the MSE cost function for a Linear Regression model happens to be a convex function, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve. This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly. These two facts have a great consequence: Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

Note :-

When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

Batch Gradient Descent :-

To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter θj. In other words, you need to calculate how much the cost function will change if you change θj just a little bit. This is called a partial derivative. It is like asking "What is the slope of the mountain under my feet if I face east?" and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions). Below equation computes the partial derivative of the cost function with regard to parameter θj, noted ∂ MSE(θ) / ∂θj

$$MSE(\theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2$$

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)}) \cdot x^{(i)}$$

Instead of computing these partial derivatives individually, you can use below equation to compute them all in one go. The gradient vector $\nabla_\theta MSE(\theta)$, contains all the partial derivatives of the cost function (one for each parameter).

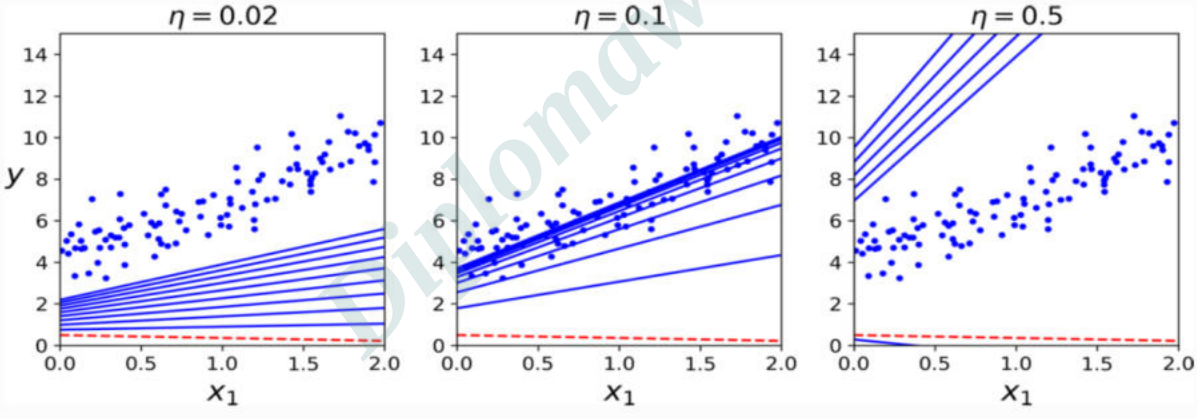
$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} X^T (X\theta - y)$$

Note :-

Notice that this formula involves calculations over the full training set X, at each Gradient Descent step! This is why the algorithm is called Batch Gradient Descent: it uses the whole batch of training data at every step (actually, Full Gradient Descent would probably be a better name). As a result it is terribly slow on very large training sets (but we will see much faster Gradient Descent algorithms shortly). However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation or SVD decomposition.

↳ Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. this means subtracting $\nabla_\theta MSE(\theta)$ from θ. this is where the learning rate η comes into play. multiply the gradient vector by η to determine the size of the downhill step.

$$\theta^{\text{next step}} = \theta - \eta \nabla_\theta MSE(\theta)$$



↳ gradient descent with various learning rates

↳

On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution. On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

↳ to find a good learning rate you can use grid search.

→

Convergence Rate :-

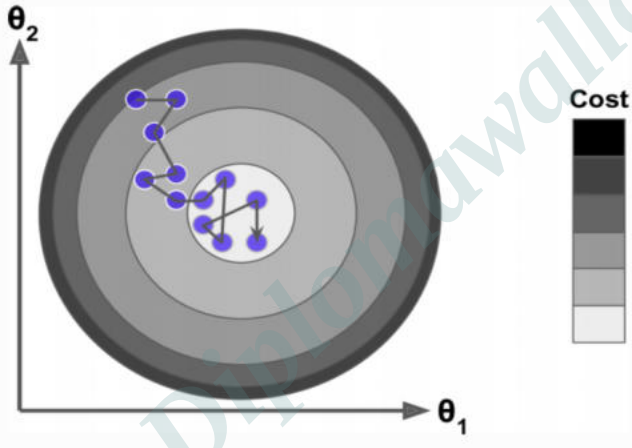
When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), Batch Gradient Descent with a fixed learning rate will eventually converge to the optimal

solution, but you may have to wait a while: it can take O(1/ϵ) iterations to reach the optimum within a range of ϵ, depending on theshape of the cost function. If you divide the tolerance by 10 to have a more precise solution, then the algorithm may have to run about 10 times longer.

→ Stochastic gradient Descent :-

- The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, Stochastic Gradient Descent picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.

- On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see Figure). So once the algorithm stops, the final parameter values are good, but not optimal.



↳ with stochastic gradient Descent, each training step is much faster but also much more stochastic than when using Batch gradient Descent.

↳

When the cost function is very irregular (as in Figure 4-6), this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

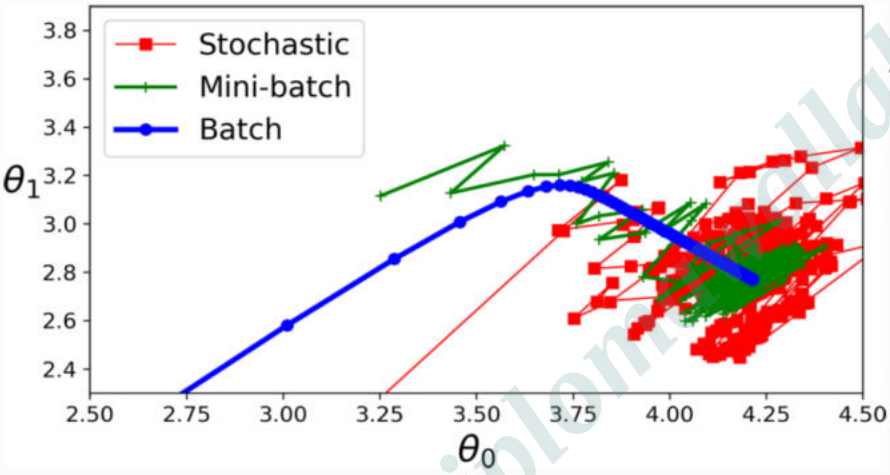
↳

To perform Linear Regression using Stochastic GD with Scikit-Learn, you can use the SGDRegressor class, which defaults to optimizing the squared error cost function.

→ Mini-Batch gradient Descent :-

instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in stochastic GD), Min-Batch GD computes the gradient on small random sets of instance called mini-batches.

↳ the main advantage of Mini-batch GD over stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.



→ gradient Descent paths in parameter space

↳ Figure shows the paths taken by the three Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around.

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

m → number of training instance

n → number of features.

↳ there is almost no difference after training.

Polynomial Regression :-

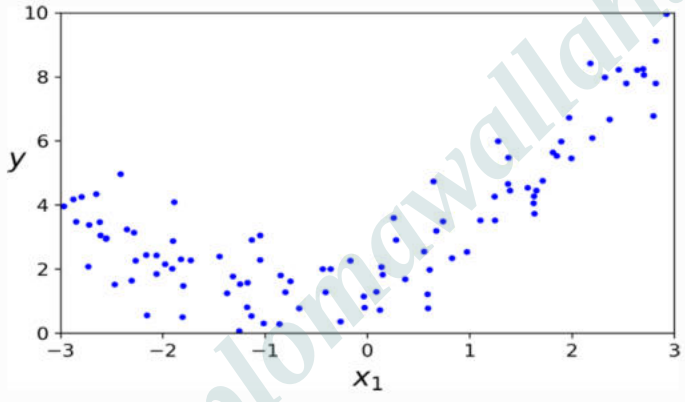
What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.

- the Normal Equation can only perform Linear Regression, the Gradient Descent algorithms can be used to train many other models, as we will see.

- A quadratic equation is of the form $y = ax^2 + bx + c$.

- code for polynomial regression

```
m = 100
X = 6 * np.random.rand(m, 1)
y = 0.5 * X**2 + X + 2 + np.random.rand(m, 1)
```



- generated nonlinear and noisy dataset

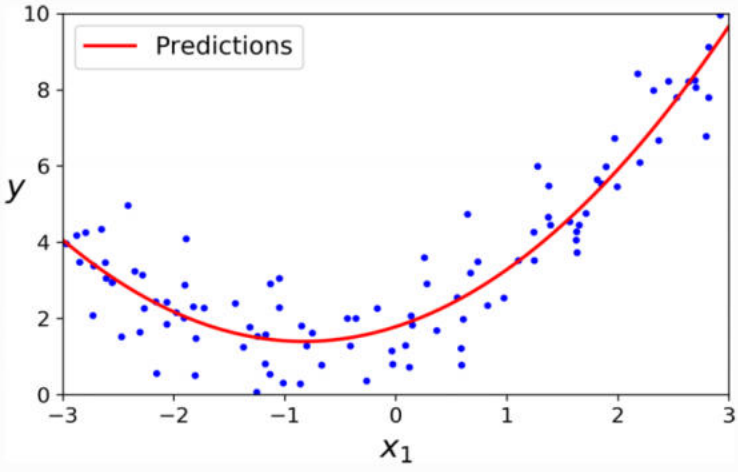
- Here straight line will never fit this data properly so lets use scikit-learns PolynomialFeatures class to transform our training data, adding the square (second degree polynomial) of each feature in the training set as a new feature.

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

- X_poly now contains the original feature of x plus the square of this feature. Now you can fit a linear

Regressor model to this extended training data

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```



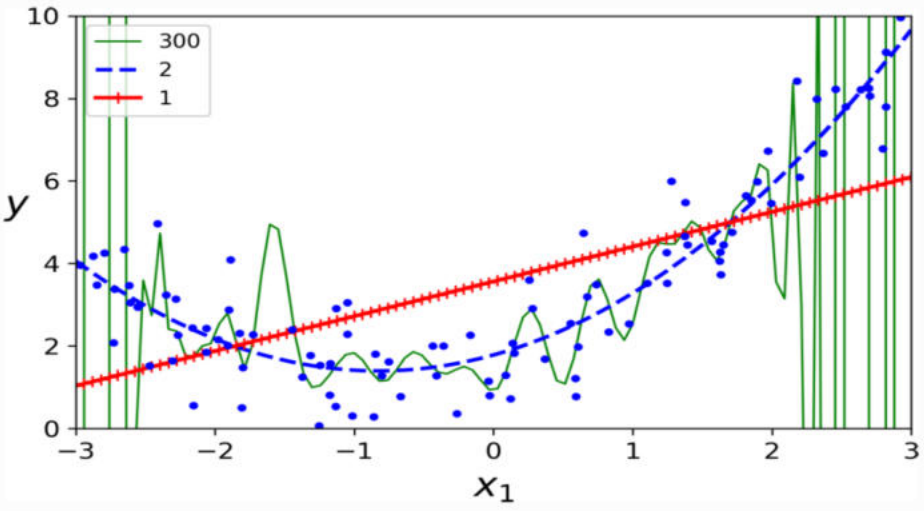
↳ the equation of line $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$

when in fact the original function was

$\hat{y} = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{gaussian noise.}$

→ Learning Curves :-

If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression. For example, Figure applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.



This high-degree Polynomial Regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model, which makes sense because the data was generated using a quadratic model. But in general you won't know what function generated the data, so how can you decide how complex your model should be? How can you tell that your model is overfitting or underfitting the data?

to get an estimate of a model's generalization performance you can use cross validation. if a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. if it performs poorly on both then it is underfitting. this is one way to tell when a model is too simple or too complex.

Another way to tell is to look at the learning curves: these are plots of the model's performance on the training set and the validation set as a function of the training set size (or the training iteration). To generate the plots, train the model several times on different sized subsets of the training set. The following code defines a function that, given some training data, plots the learning curves of a model:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

Let's look at the learning curves of the plain Linear Regression model (a straight line; see Figure 4-15):

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

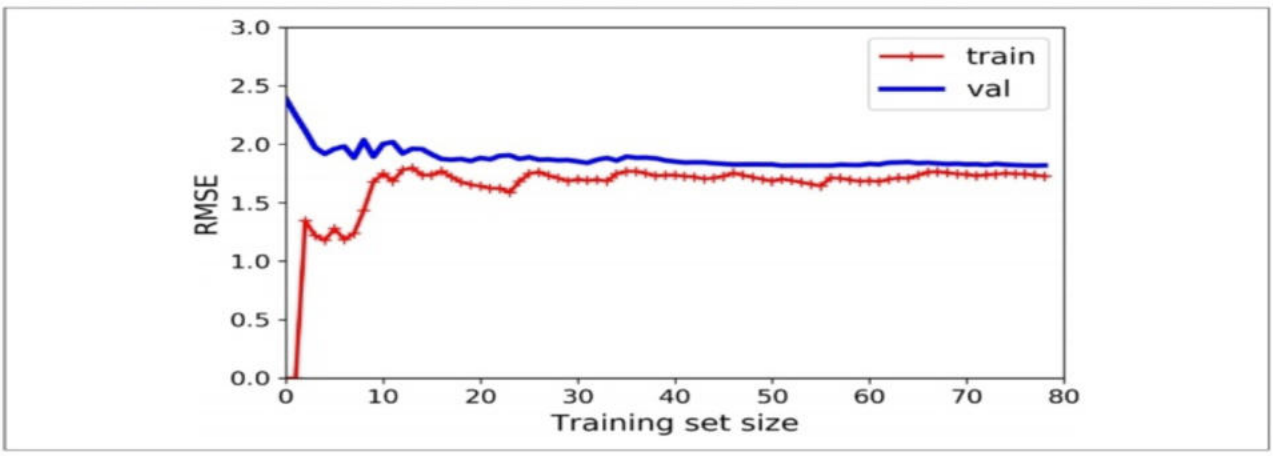


Figure 4-15. Learning curves

↳ if your model is underfitting the training data, adding more training examples will not help. You need to use a more complex model or come up with better features.

↳ draw this learning curves for different model and check whether it perform well or not.

↳ One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

Note :-

↳ Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model complexity increases its bias and reduces its variance.

→ Regularized Linear Models :-

a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.
For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at Ridge Regression, Lasso Regression, and Elastic Net, which implement three different ways to constrain the weights.

① Ridge Regression :- Ridge Regression is a regularized version of linear Regression. a regularization term equal to $\alpha \sum_{i=1}^{n} \theta_i^2$ is added to the cost function. this forces the learning algorithm to not only fit the data but also keep the model weights as small as possible

the hyperparameter 'α' controls how much you want to regularize the model. if α=0, then Ridge regression is just linear Regression.

$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

↓ cost function.

Note :-

→ It is important to scale the data (e.g., using a StandardScaler) before performing Ridge Regression, as it is sensitive to the scale of the input features. This is true of most regularized models.

→ from sklearn.linear_model import SGDRegressor

→ model = SGDRegressor(penalty = "l2")

↳ Ridge Regression

② Lasso Regression : least Absolute Shrinkage and selection Operator Regression (usually simply called Lasso Regression).

→ just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ1 norm of the weight vector instead of half the square of the ℓ2 norm.

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

→

An important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero).Lasso Regression automatically performs feature selection and outputs a sparse model (i.e., with few nonzero feature weights).

Here is a small Scikit-Learn example using the Lasso class:

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

↳ you could instead use SGDRegressor(penalty='l1')

③ Elastic Net :-

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio r. When r= 0, Elastic Net is equivalent to Ridge Regression, and when r = 1, it is equivalent to Lasso Regression.

$$J(\theta) = MSE(\theta) + r \alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2$$

so generally you should avoid plain Linear Regression. Ridge is a good default, but if you suspect that only a few features are useful, you should prefer Lasso or Elastic Net because they tend to reduce the useless features' weights down to zero,Elastic Net is preferred over Lasso because Lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

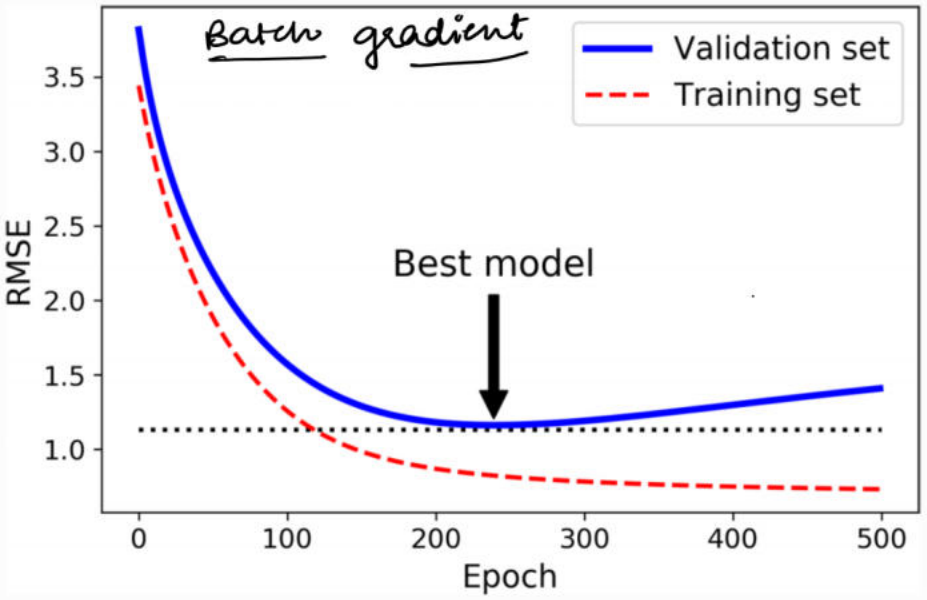
Here is a short example that uses Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds to the mix ratio *r*):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

X, y → training Data

Early stopping :-

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called early stopping.



→ Early stopping regularization

Note :-

With Stochastic and Mini-batch Gradient Descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.

```python
from sklearn.base import clone
from sklearn.pipeline import Pipeline
from sklearn.linear_model import SGDRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures,StandardScaler
import numpy as np
from math import inf
#data prepration
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)


X_train,X_val,y_train,y_val=train_test_split(X,y,test_size=0.2,random_state=42)


poly_pipe=Pipeline([
    ("poly_feat",PolynomialFeatures(degree=90,include_bias=False)),
    ("STd",StandardScaler())
])

x_poly_scaled=poly_pipe.fit_transform(X_train)

x_val_scaled=poly_pipe.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, warm_start=True,
 penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None

for epoch in range(1000):
    sgd_reg.fit(x_poly_scaled,y_train)
    y_poly_pred=sgd_reg.predict(x_val_scaled)
    error=mean_squared_error(y_val,y_poly_pred)
    if error<minimum_val_error:
        minimum_val_error=error
        best_epoch=epoch
        best_model=clone(sgd_reg)
```

✓ 1.6s

↳ Implementation of early stopping

→ **Logistic regression :-** this regression algorithm is used for classification task.

→

Logistic Regression (also called Logit Regression) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"), and otherwise it predicts that it does not (i.e., it belongs to the negative class, labeled "0"). This makes it a binary classifier.

↳ Just like Linear Regression model, Logistic regression model computes a weighted sum of input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the logistic of the Result.

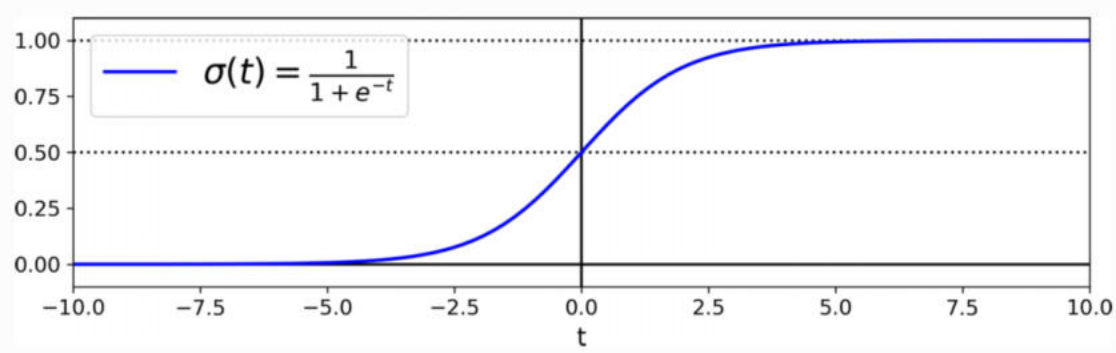
$$\hat{p} = h_\theta(x) = \sigma(x^T\theta)$$

→ logistic regression model estimated probability (Vectorized form).

σ → sigmoid function that output number between 0 and 1.

$$\sigma(t) = \frac{1}{1+e^{[-t]}}$$

$$\hat{p} = \frac{1}{1+e^{-x^T\theta}}$$



→ logistic function

→

Once the Logistic Regression model has estimated the probability $\tilde{p} = h\theta(x)$ that an instance x belongs to the positive class, it can make its prediction ŷ easily

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

↳ Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so logistic Regression model predicts 1 if $x^T\theta$ is positive and 0 if it is negative.

→

The score t is often called the logit. The name comes from the fact that the logit function, defined as logit(p) = log(p / (1 − p)), is the inverse of the logistic function. Indeed, if you compute the logit of the estimated probability p, you will find that the result is t. The logit is also called the log-odds, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class.

$$\sigma(t) = p = \frac{1}{1+e^{-t}} \Rightarrow 1+e^{-t} = \frac{1}{p}$$

$$\Rightarrow e^{-t} = \frac{1-p}{p} \quad \Big| \Rightarrow \log(e^{-t}) = \log\left(\frac{1-p}{p}\right)$$

$$-t = \log(1-p/p)$$

$$\Rightarrow \boxed{t = \log \frac{p}{1-p}}$$

→

The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances (y = 1) and low probabilities for negative instances (y = 0). This idea is captured by the cost function shown in Equation for a single training instance x

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1-\hat{p}) & \text{if } y = 0 \end{cases}$$

→ cost function

→

This cost function makes sense because −log(t) grows very large when t approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance. On the other hand, −log(t) is close to 0 when t is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

↳ the cost function over the whole training set is the average cost over all training instances. it can be written in a single expression called log loss,

$$J(\theta) = \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \hat{p}^{(i)}\right) \right]$$

↳ logistic regression cost function

→ this cost function is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).

→ logistic cost function partial derivatives

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left[\sigma\left[\theta^T x^{(i)}\right] - y^{(i)} \right) x_j^{(i)}$$

→ for each instance it computes the prediction error and multiplies it by the jth feature value, and then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives, you can use it in the Batch Gradient Descent algorithm.

```
# Logistic Regression
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split


data=load_iris()
new_data=pd.DataFrame(data["data"])
new_data.columns=data.feature_names
new_data['target']=data["target"]

X=new_data.iloc[:,:4]
y=(new_data["target"]==2)
y=pd.factorize(y)
y=y[0]

model=LogisticRegression()

X_train,x_test,y_train,y_test=train_test_split(X,y)
model.fit(X_train,y_train)

y_pred=model.predict(x_test)

accuracy_score(y_test,y_pred)
```

```
[68]  ✓ 0.0s
...  0.9736842105263158
```

→ Implementation of logistic regression

→ **Softmax Regression** :- the logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers this is called softmax Regression, or Multinomial logistic Regression.

→

when given an instance x, the Softmax Regression model first computes a score sk(x) for each class k, then estimates the probability of each class by applying the softmax function (also called the normalized exponential) to the scores. The equation to compute sk(x) should look familiar, as it is just like the equation for Linear Regression prediction •

$$s_k(x) = x^T \theta^{(k)}$$

softmax score for class k

↳ Note that each class has its own dedicated parameter vector $\theta^{(k)}$. all these vectors are typically stored as rows in a parameter matrix.

→

Once you have computed the score of every class for the instance x, you can estimate the probability $\hat{p}k$ that the instance belongs to class k by running the scores through the softmax function (Equation). The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

$$\hat{p}_k = \sigma(s(x))_k = \frac{e^{[s_k(x)]}}{\sum_{j=1}^{N} e^{[s_j(x)]}}$$

N → No of classes

s(x) = is a vector containing the scores of each class for the instance x.

$\sigma[s(x)]_k$ ⇒ estimated probability that the instance x belongs to class k, given the scores of each class for that instance.

→

Just like the Logistic Regression classifier, the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score)

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(s(x))_k = \underset{k}{\operatorname{argmax}}\ s_k(x) = \underset{k}{\operatorname{argmax}}\ \left((\theta^{(k)})^T x\right)$$

↳ the argmax operator returns the value of a variable that maximizes a function.

Note :-

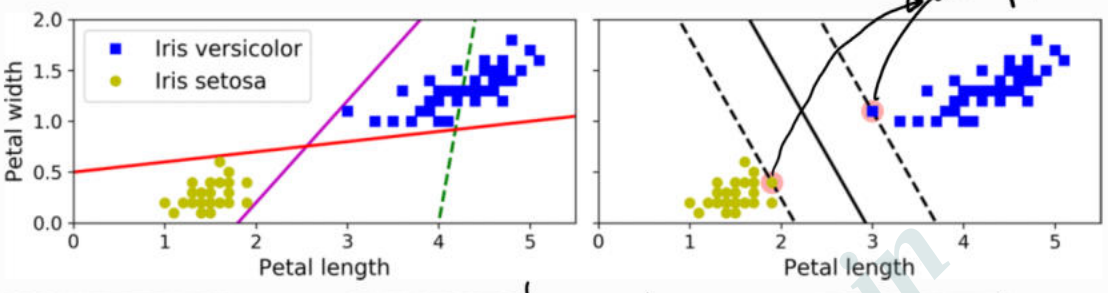
→ The Softmax Regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different types of plants. You cannot use it to recognize multiple people in one picture.

:- chapter-5 :-

(Support Vector Machines)

→ A Support Vector Machine (SVM) is a powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex small- or medium-sized datasets.

→ Linear SVM :-



support vectors

↳ through SVM

↳ large margin classification

through linear regression

→ the left plot shows the decision boundaries of three possible linear classifiers. the model whose decision boundry is represented by the dashed line is so bad it does not even seperate the classes properly the other two models work perfectly on the training set, but this decision boundries come so close to the instances that these models will not perform well on new instances.

→ the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called large margin classification.

→ always try to do feature scaling before using SVM algo.

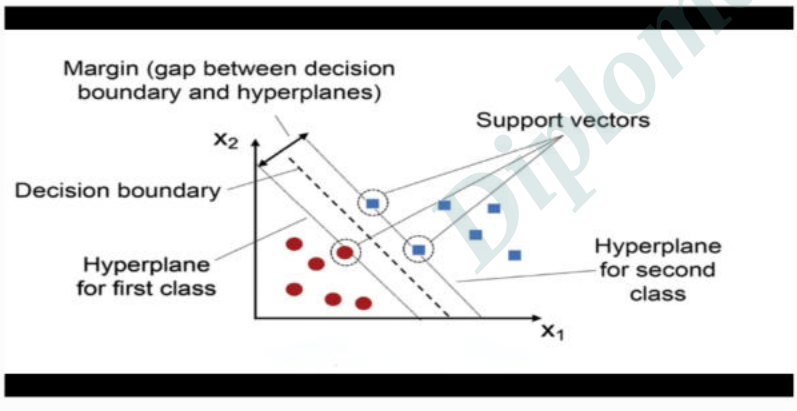
↳ SVMs work by finding the optimal hyperplane that seperates data points into different classes.

Key terms :-

① Hyperplane :-

A hyperplane is a decision boundary that separates data points into different classes in a high-dimensional space. In two-dimensional space, a hyperplane is simply a line that separates the data points into two classes. In three-dimensional space, a hyperplane is a plane that separates the data points into two classes. Similarly, in N-dimensional space, a hyperplane has (N-1)-dimensions.

↳ it can be used to make prediction on new data points by evaluating which side of the hyperplane they fall on. Data points on one side of the hyperplane are classified as belonging to one class, while data points on the other side of the hyperplane are classifieds as belonging to another class.



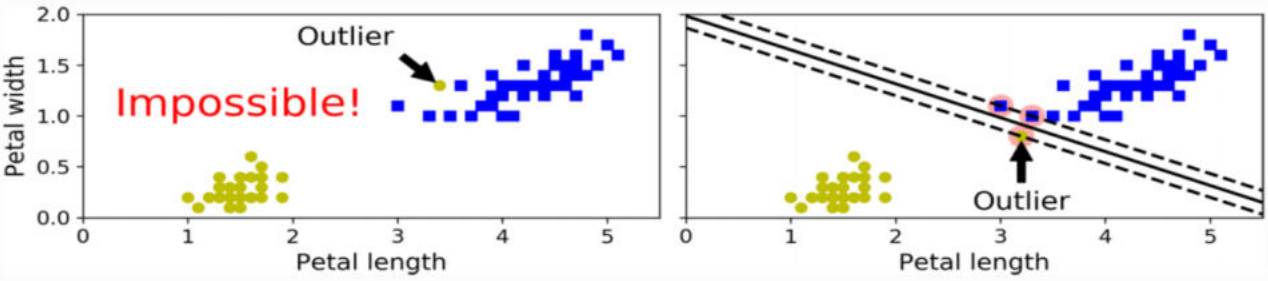
Support Vectors :- they are the data points that lie closest to the decision boundary (hyperplane) in SVM.

② Margin :-

A margin is the distance between the decision boundary (hyperplane) and the closest data points from each class. The goal of SVMs is to maximize this margin while minimizing classification errors. A larger margin indicates a greater degree of confidence in the classification, as it means that there is a larger gap between the decision boundary and the closest data points from each class. The margin is a measure of how well-separated the classes are in feature space. SVMs are designed to find the hyperplane that maximizes this margin, which is why they are sometimes referred to as maximum-margin classifiers.

→ Soft Margin classification :-

If we strictly impose that all instances must be off the street and on the right side, this is called hard margin classification. There are two main issues with hard margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers.

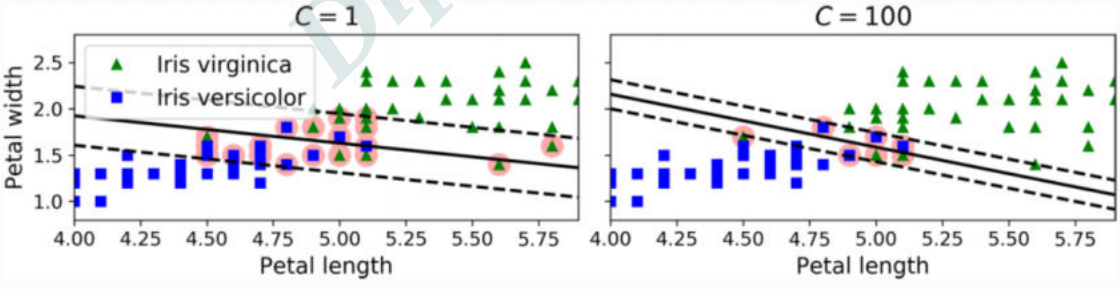


↳ hard margin sensitivity to outliers.

on the left, it is impossible to find a hard margin; on the right, the decision boundary ends up very different from the one we saw in without the outlier, and it will probably not generalize as well.

→ To avoid these issues, use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the margin violations (i.e., instances that end up in the middle of the street or even on the wrong side). This is called soft margin classification.

→ When creating an SVM model using Scikit-Learn, we can specify a number of hyperparameters. C is one of those hyperparameters. If we set it to a low value, then we end up with the model on the left of Figure. With a high value, we get the model on the right. Margin violations are bad. It's usually better to have few of them. However, in this case the model on the left has a lot of margin violations but will probably generalize better.



↳ large margin (left) vs few margin violation (Right)

↳ if your SVM model is overfitting, you can try regularizing it by reducing C.

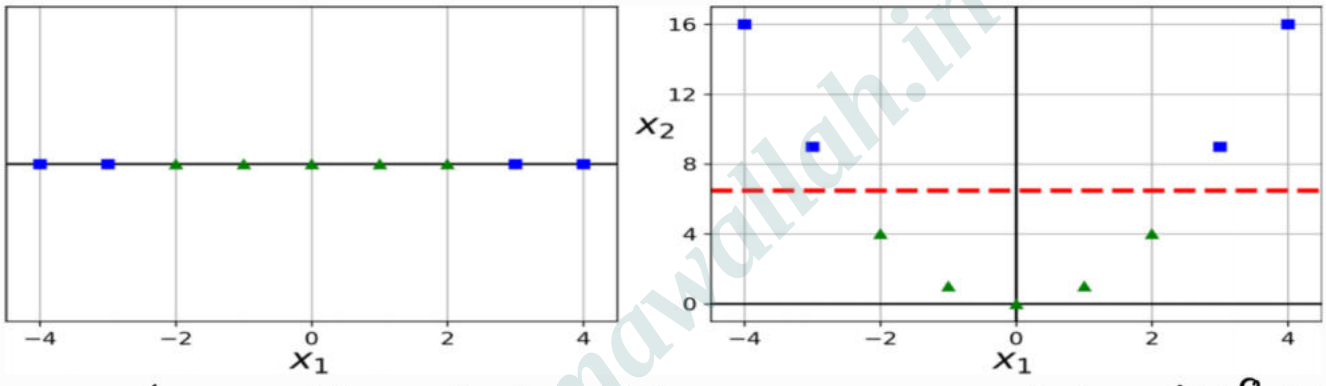
↳ Unlike logistic regression classifiers, SVM classifiers do not output probabilities for each class.

↳ In sklearn we have LinearSVC and we can use this model directly just by importing.

↳ Instead of using the LinearSVC class, we could use the SVC class with a linear kernel. When creating the SVC model, we would write SVC(kernel="linear", C=1). Or we could use the SGDClassifier class, with SGDClassifier(loss="hinge", alpha=1/(m*C)). This applies regular Stochastic Gradient Descent to train a linear SVM classifier. It does not converge as fast as the LinearSVC class, but it can be useful to handle online classification tasks or huge datasets that do not fit in memory (out-of-core training).

→ Non linear SVM classification :-

→ Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features, in some cases this can result in a linearly separable dataset.



↳ adding features to make a dataset linearly seperable

→ Consider the left plot in Figure: it represents a simple dataset with just one feature, x1. This dataset is not linearly separable, as you can see. But if you add a second feature x2= $(x1)^2$ the resulting 2D dataset is perfectly linearly separable.

→ **Polynomial kernel :-** Adding polynomial features is simple to implement and can work great with all sorts of ML algorithms (not just SVM). at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow. fortunetly, when using SVMs you can apply almost miraculous mathematical technique called the kernel trick.

↳

The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features because you don't actually add any features. This trick is implemented by the SVC class.

```
→ from sklearn.svm import SVC
  poly_kernel_svm_clf = Pipeline([
                          ("scaler", StandardScaler()),
                          ("SVC", SVC(kernel="poly", C=5,
                                      degree=3, coef0=1))
                          ])
```

↳ the hyperparameter "coef0" controls how much the model is influenced by high degree polynomial vs low-degree polynomials.

↳ gaussian RBF Kernal also used to talkel with non linear datasets.

$$\phi_\gamma(x, \ell) = \exp(-\gamma \|x - \ell\|^2)$$

$\gamma \rightarrow$ hyperparameter
$\ell \rightarrow$ landmark

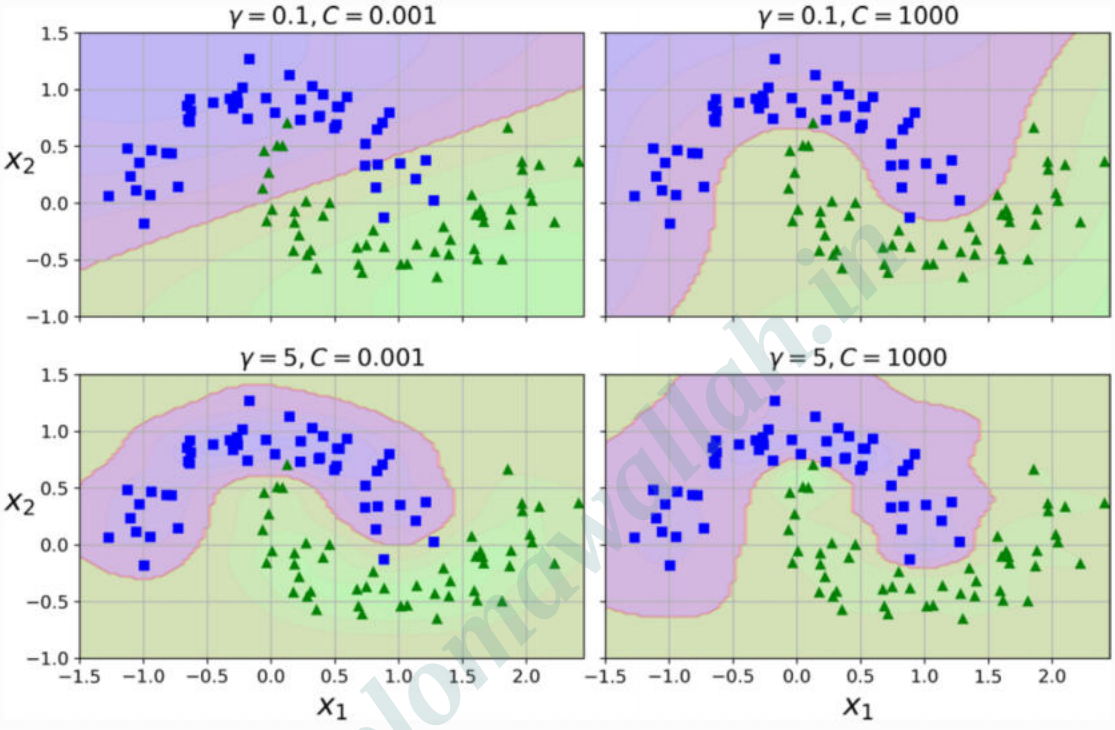
eg:- instance $x_1 = -1$ it is located at a distance of 1 from first landmark and 2 from second landmark so new feature will be

$$x_2 = e^{[-0.3 \times 1^2]} = 0.74$$
$$x_3 = e^{[-0.3 \times 2^2]} = 0.30$$

→ new features

↳ if you want to use in SVM then just change kernal="rbf".

↳ 'γ' acts like a regularization hyperparameter if your model is overfitting, you should reduce it; if it is underfitting you should increase it. (similar to C hyperparameter).



↳ SVM classifiers using an RBf kernal

↳ Note :-

→ Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. String kernels are sometimes used when classifying text documents or DNA sequences (e.g., using the string subsequence kernel or kernels based on the Levenshtein distance).

→ With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first (remember that LinearSVC is much faster than SVC(kernel="linear")), especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should also try the Gaussian RBF kernel; it works well in most cases. Then if you have spare time and computing power, you can experiment with a few other kernels, using cross-validation and grid search. You'd want to experiment like that especially if there are kernels specialized for your training set's data structure.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No → liblinear library
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes → libsvm library

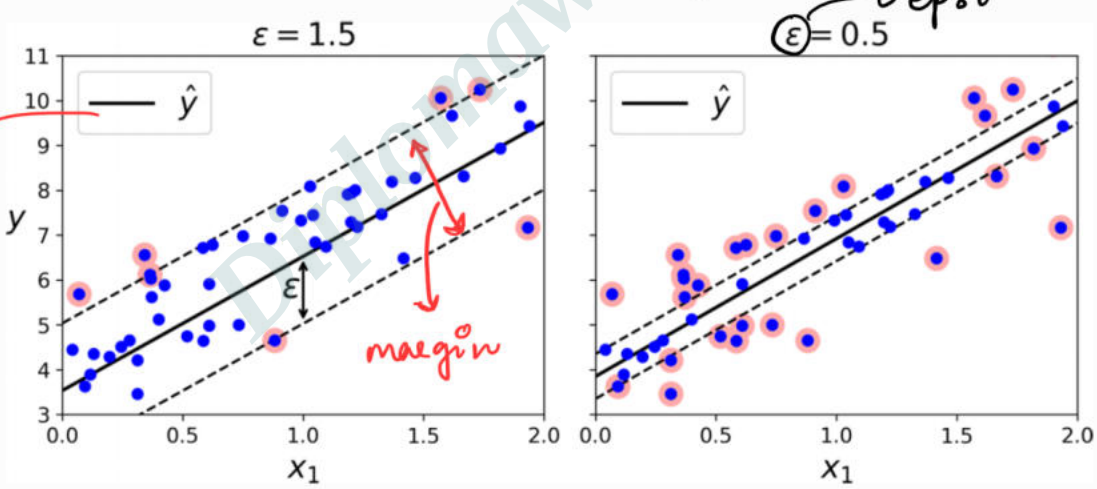
SVM Regression :- Svm support linear and non linear regression.

→

To use SVMs for regression instead of classification, the trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street). The width of the street is controlled by a hyperparameter, ϵ.

→ two linear SVM Regression models trained on same random linear data, one with a large margin ($\epsilon = 1.5$) and other with a small margin ($\epsilon = 0.5$)

→ epsi



Note:- margin = 2ϵ

→

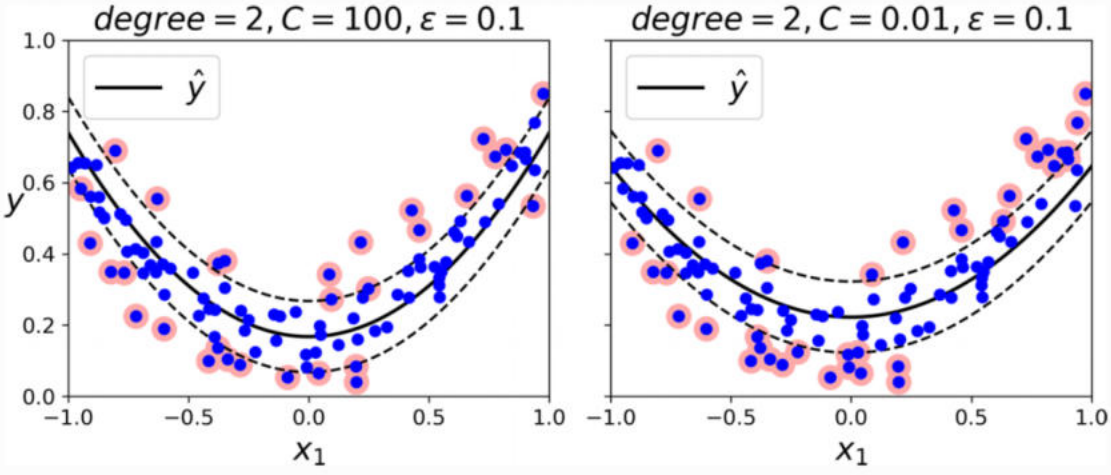
Adding more training instances within the margin does not affect the model's predictions; thus, the model is said to be ϵ-insensitive.

→ you can use sklearn to implement SVM regression by importing LinearSVR class.

→ from sklearn.svm import LinearSVR
→ svm_reg = LinearSVR(epsilon=1.5)
→ svm_reg.fit(X, Y)

ϵ → epsilon

To tackle nonlinear regression tasks, you can use a kernelized SVM model. Figure shows SVM Regression on a random quadratic training set, using a second-degree polynomial kernel. There is little regularization in the left plot (i.e., a large C value), and much more regularization in the right plot (i.e., a small C value).



→ from sklearn.svm import SVR

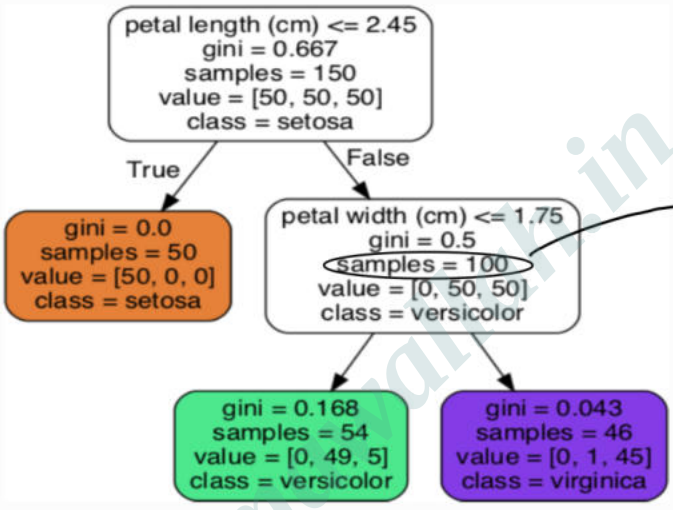
→ svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)

The SVR class is the regression equivalent of the SVC class, and the LinearSVR class is the regression equivalent of the LinearSVC class. The LinearSVR class scales linearly with the size of the training set (just like the LinearSVC class), while the SVR class gets much too slow when the training set grows large (just like the SVC class).

-: chapter - 6 :- [Decision Trees]

→ Decision tree can perform both classification and regression tasks, and even multioutput tasks. it is powerfull algo, it can fit more complex datasets.

→ lets build a decision tree and see how it make predictions. We are using iris dataset for this and feature petal length and petal width.



→ Training instances

→

Suppose you find an iris flower and you want to classify it. You start at the root node (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a leaf so it does not ask any questions: simply look at the predicted class for that node, and the Decision Tree predicts that your flower is an Iris setosa (class=setosa).

→ Now suppose you find another flower, and this time the petal length is greater than 2.45 cm. You must move down to the root's right child node (depth 1, right), which is not a leaf node, so the node asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an Iris versicolor (depth 2, left). If not, it is likely an Iris virginica (depth 2, right).

→

Decision Trees is that they require very little data preparation. In fact, they don't require feature scaling or centering at all.

→ node's gini attribute measures its impurity: a node is "pure" (gini=0) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to Iris setosa training instances, it is pure and its gini score is 0.

$$\text{gini index } (G_i) = 1 - \sum_{K=1}^{n} p_{i,K}^{2}$$

$p_{i,K}$ is the ratio of class K instances among the training instances in the i^{th} node.

eg:- g_i, score for depth-2 left node

$$g_i = 1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$$

Note:→

Scikit-Learn uses the CART algorithm, which produces only binary trees: nonleaf nodes always have two children (i.e., questions only have yes/no answers). However, other algorithms such as ID3 can produce Decision Trees with nodes that have more than two children

→

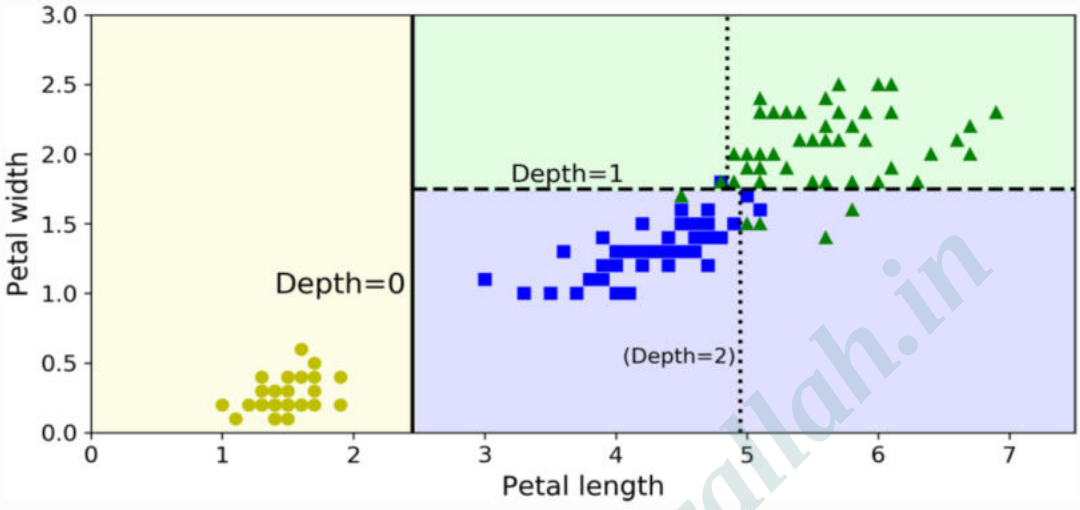


Figure shows this Decision Tree's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the lefthand area is pure (only Iris setosa), it cannot be split any further. However, the righthand area is impure, so the depth-1 right node splits it at petal width =1.75 cm (represented by the dashed line). Since max_depth was set to 2, the DecisionTree stops right there. If you set max_depth to 3, then the two depth-2 nodes would each add another decision boundary (represented by the dotted lines).

→ Decision trees are intuitive, and their decisions are easy to interpret. Such models are often called white box models.

→

A Decision Tree can also estimate the probability that an instance belongs to a particular class k. First it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class k in this node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the Decision Tree should output the following probabilities: 0% for Iris setosa (0/54), 90.7% for Iris versicolor (49/54), and 9.3% for Iris virginica (5/54). And if you ask it to predict the class, it should output Iris versicolor (class 1) because it has the highest probability.

CART Algorithm :-

Scikit-Learn uses the Classification and Regression Tree (CART) algorithm to train Decision Trees (also called "growing" trees). The algorithm works by first splitting thetraining set into two subsets using a single feature k and a threshold tk(e.g., "petal length ≤ 2.45 cm"). How does it choose k and tk ? It searches for the pair (k, tk) that produces the purest subsets (weighted by their size) .

→ cart cost function that algorithm try to minimize

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

- G_{left} , G_{right} measures impurity of left/right subset
- $m_{left/right}$ is the number of instances in the left/right subset
- m → total subset

→ Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the max_depth hyperparameter), or if it cannot find a split that will reduce impurity.

→ CART algo is an greedy algo. it greedily searches for an optimum split at the top level, then repeats the proces at each subsequent level.

→ Finding the optimal tree to be an NP-complete problem it requires $O(e^m)$ time.

→ prediction requires $O(\log_2 m)$ time complexity for even larger training set.

→ Entropy :- We can use entropy instead of gini by setting the hyperparameter criterion to "entropy". In ML, entropy used as an impurity measures a set's entropy is zero

when it contains instances of only one class.

$$\text{Entropy } (\Delta H) = -\sum_{\substack{k=1 \\ P_{i,k} \neq 0}}^{n} P_{i,k} \log_2 (P_{i,k})$$

eg: for depth 2 left node of above tree is

$$\Delta H = -49/54 \log_2 49/54 - 5/54 \log_2 (5/54)$$
$$\approx 0.445$$

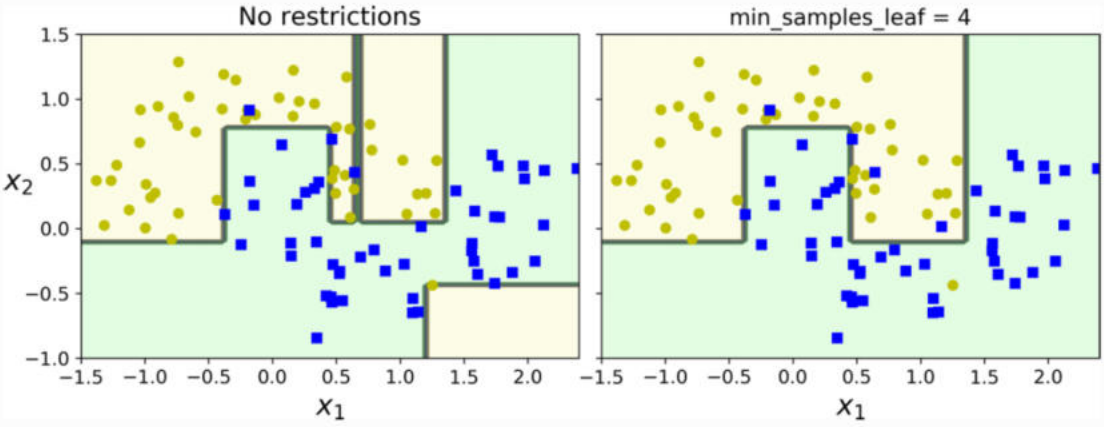
So, should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.

To avoid overfitting the training data, you need to restrict the Decision Tree's freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the Decision Tree. In Scikit-Learn, this is controlled by the max_depth hyperparameter (the default value is None, which means unlimited). Reducing max_depth will regularize the model and thus reduce the risk of overfitting.

The DecisionTreeClassifier class has a few other parameters that similarly restrict the shape of the Decision Tree: min_samples_split (the minimum number of samples a node must have before it can be split), min_samples_leaf (the minimum number of samples a leaf node must have), min_weight_fraction_leaf (same as min_samples_leaf but expressed as a fraction of the total number of weighted instances), max_leaf_nodes (the maximum number of leaf nodes), and max_features (the maximum number of features that are evaluated for splitting at each node). Increasing min_*hyperparameters or reducing max_* hyperparameters will regularize the model.

Other algorithms work by first training the Decision Tree without restrictions, then pruning (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the $\chi 2$ test (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the null hypothesis). If this probability, called the p-value, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

→ Figure shows two Decision Trees trained on the moons dataset

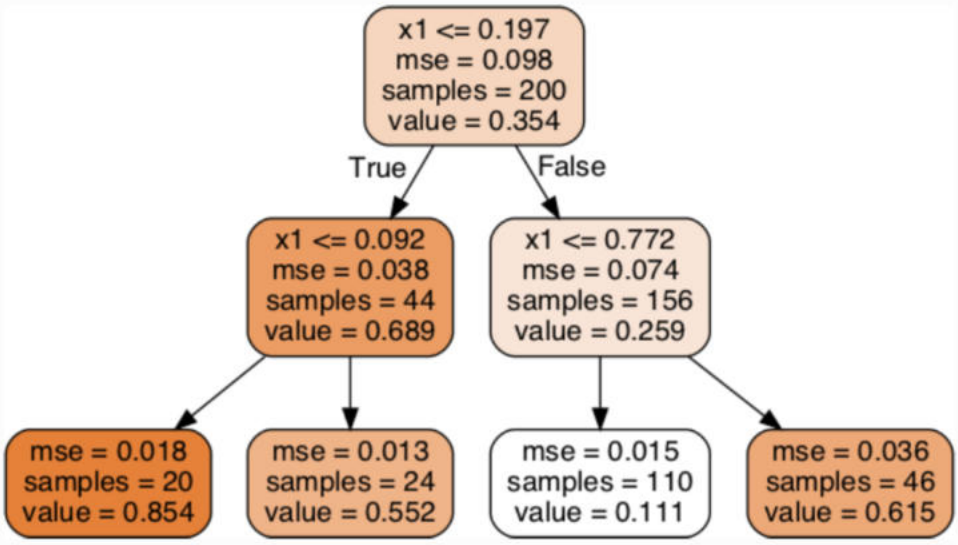


On the left the Decision Tree is trained with the default hyperparameters (i.e., no restrictions), and on the right it's trained with min_samples_leaf=4. It is quite obvious that the model on the left is overfitting, and the model on the right will probably generalize better.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris=load_iris()

x=iris["data"][:,:2]
y=iris["target"]
dt=DecisionTreeClassifier(max_depth=2)
dt.fit(x,y)
y=dt.predict([[5,1.2]])
if y==0:
    print("iris_setosa")
elif y==1:
    print("iris_versicolor")
else:
    print("iris_virginica")
```

```
[23]  ✓ 0.0s
...  iris_versicolor
```


→ **Regression :-** Decisions tree are also capable of performing regression tasks. Let's build a regression tree

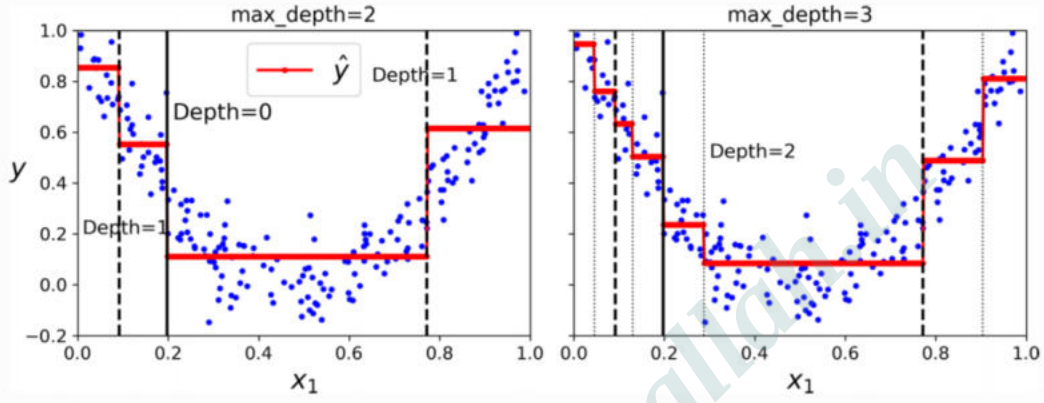


→ this tree look similar to classification tree that we built earlier. the main difference is that instead

of predicting class it predicts a value.

For example, suppose you want to make a prediction for a new instance with x1 = 0.6. You traverse the tree starting at the root, and you eventually reach the leaf node that predicts value=0.111. This prediction is the average target value of the 110 training instances associated with this leaf node, and it results in a mean squared error equal to 0.015 over these 110 instances.

This model's predictions are represented on the left in Figure . If you set max_depth=3, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.



↳ predictions of two decision Tree regression models.

The CART algorithm works mostly the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. Equation shows the cost function that the algorithm tries to minimize.

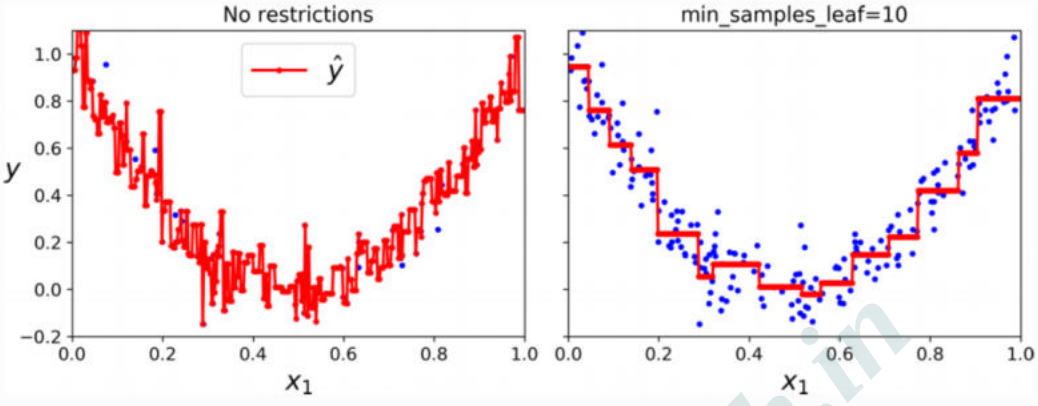
$$J(k, t_k) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right}$$

$$MSE_{node} = \sum_{i \in node} (\hat{y}_{node} - y^{(i)})^2$$

$$\hat{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y^{(i)}$$

Just like for classification tasks, Decision Tree are prone to overfitting when dealing with regression tasks.

→ Without any regularization (using the default hyperp-arameters), you get the prediction on lyt in below figure. the prediction are obviously overfitting the training very badly. just setting min_samples_leaf=10 results in a much reasonable model, shown in right of below figure.



without regularization

→ regularized a Decision Tree regressor

→ the main issue with Decision Trees is that they are very sensitive to small variations in the training data.

- : Chapter - 6 :-

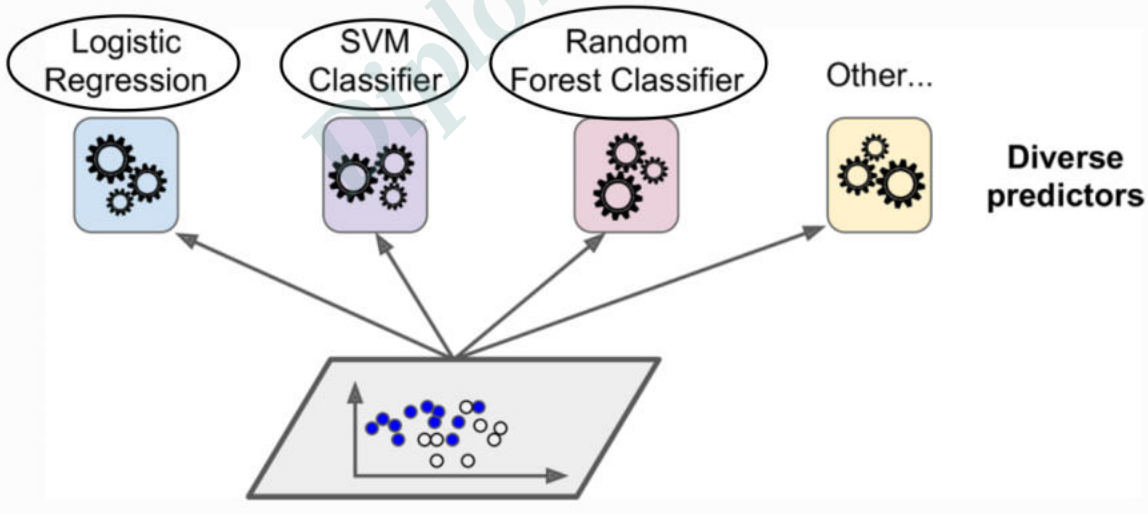
Ensemble Learning and Random Forests

→ Suppose you pose a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the wisdom of the crowd. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an ensemble; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method.

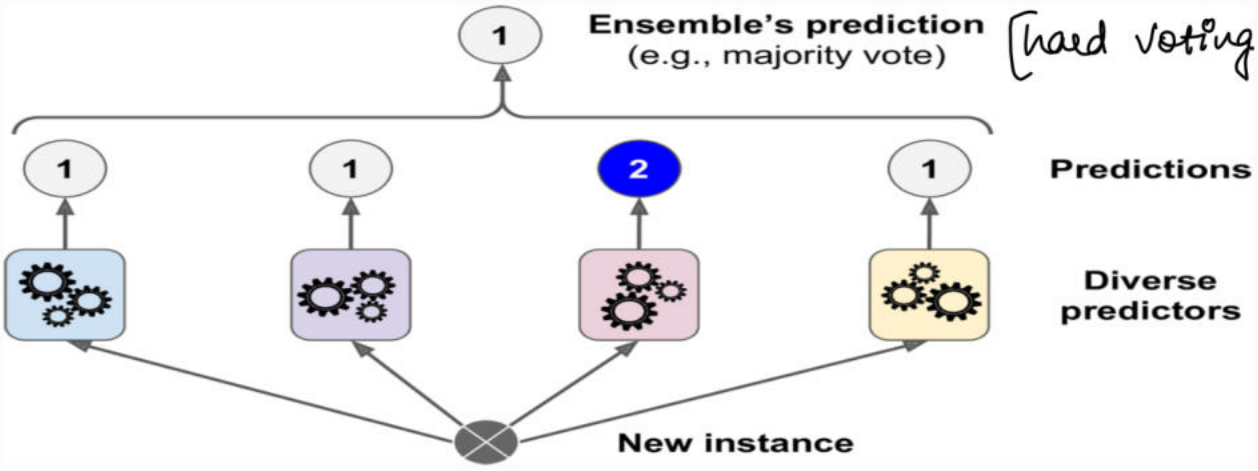
→ As an example of an Ensemble method, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you obtain the predictions of all the individual trees, then predict the class that gets the most votes (see the last exercise in Chapter 6). Such an ensemble of Decision Trees is called a Random Forest, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

→ Voting classifiers.

→ suppose you have trained a few classifiers, each one achieving about 80% accuracy. and those are below



→ a very simple way to create an better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a hard voting classifier.



[hard voting classifier]

↳ hard voting classifier

→ sometime this voting classifier achieves a higher accuracy than the best classifier in the ensemble.

→ Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

↳ Hard Voting

- Each classifier in the ensemble makes a prediction (class label), and the final prediction is determined by majority voting.
- The class that receives the most votes from the classifiers is chosen as the final output.

→ Soft Voting

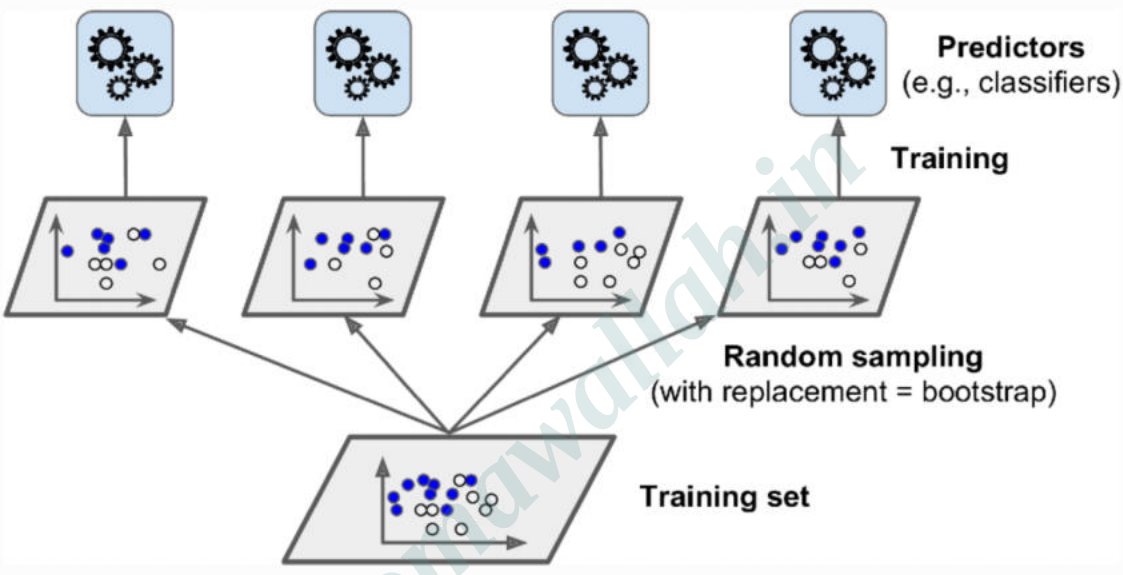
- Each classifier provides a probability estimate for each class (using methods like predict_proba()).
- The final prediction is made by averaging these probabilities and selecting the class with the highest average probability.

→ code

```
from sklearn.ensemble import RandomForestClassifier,VotingClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

# Generate the dataset
X, y = make_moons(n_samples=10000, noise=0.1, random_state=42)

x_train,x_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
r_f=RandomForestClassifier()
svc=SVC(probability=True)
lgl=LogisticRegression()
Voting_clf=VotingClassifier(estimators=[("random",r_f),("svc",svc),("lgl",lgl)],voting="soft")

Voting_clf.fit(x_train,y_train)
```

→ **Bagging and Pasting :-** Same training algo for every predictors and train them on different random subsets of the training set. When sampling is performed with replacement, this method is called bagging. When sampling is performed without replacement, it is called pasting.

→ both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in Figure



→ Once all predictors are trained, the ensembles can make a prediction for a new instance by simply aggregating the predictions of all predictors. the aggregation function is typically the statistical mode (most frequent prediction, just like a hard voting classifier) for classification or average for regression.

→

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf=BaggingClassifier(DecisionTreeClassifier(),
        n_estimators=100,max_samples=1000,
        bootstrap=True,n_jobs=-1)
```

→ Base algo

→ and these are hyperparameters.

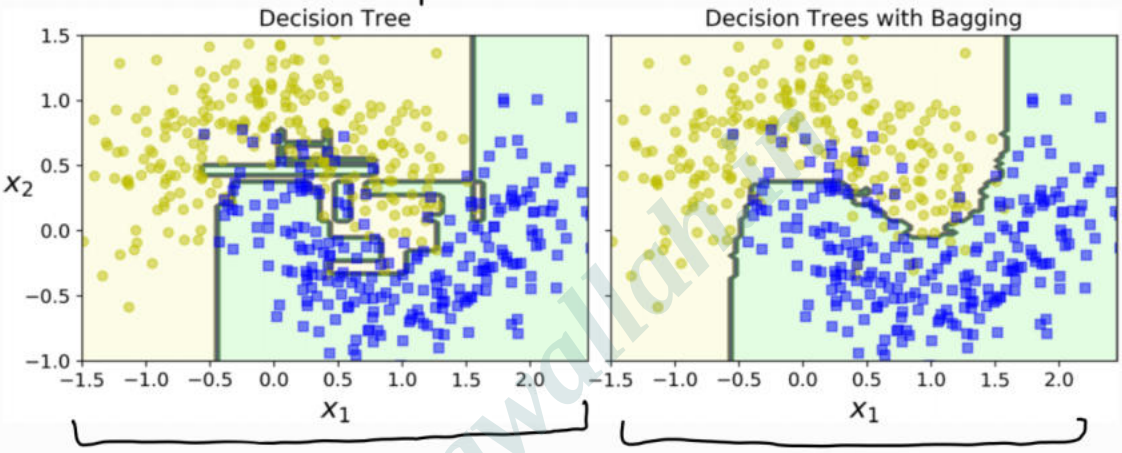
→ no of training sample given to single base model

→ no of base algo model

Note :-

→ the BaggingClassifier automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities [if it has predict_proba() method].

→ Below image compare single decision tree boundary with DT boundary of a bagging ensemble of 500 trees, both trained on same moon dataset. the ensemble predictions will likely generalize much better than single Decision Tree's predictions.



A single Decision Tree

bagging ensemble of 500 trees.

→ Random Patches and Random subspaces :-

the BaggingClassifier class supports sampling the features as well. Sampling is controlled by two hyperparameters: max_features and bootstrap_features. they work the same way as max_samples and bootstarp, but for feature sampling instead instance sampling. Thus, each predictor will be trained on a random subset of the input features.

→ this technique is particularly useful when you are dealing with high-dimensional inputs (such as images).

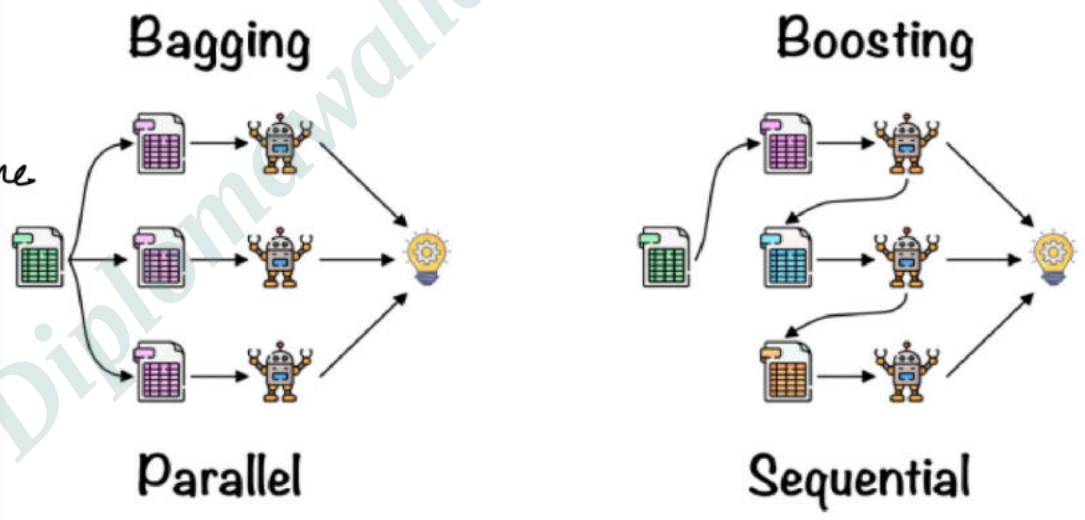
→ Sampling both training instance and features is called the Random Patches method.

→ Keeping all training instance [by setting bootstrap=False and max_samples=1.0] but sampling features [by bootstrap_features = True and and/or max_features to a value smaller than 1.0] is called the Random Subspace Method.

→ **Random Forests :-** (RFA)

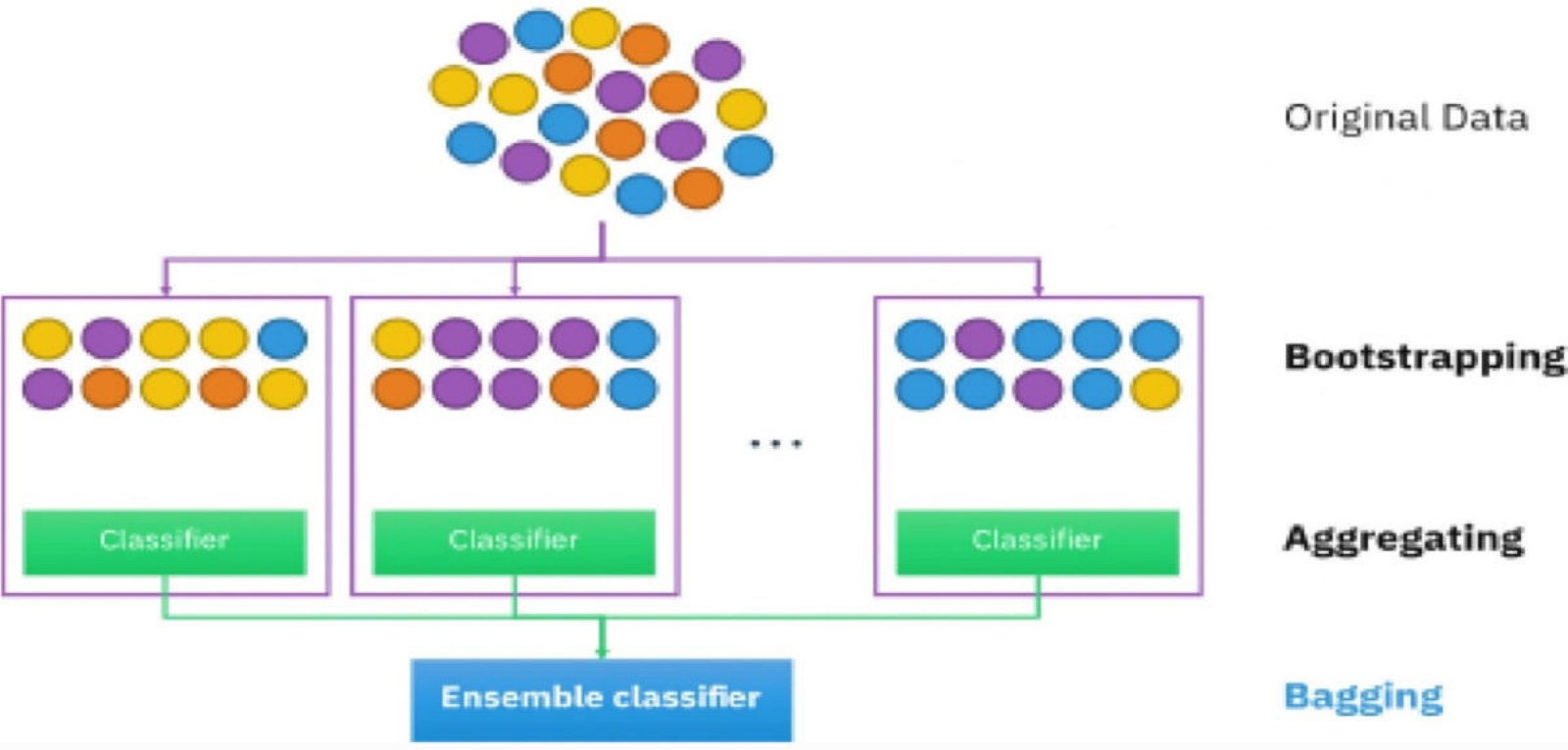
- RFA can handle data set containing continous var., as in case of regression and categorical var. as in the case of classification.

→ Random forest classifier works on the bagging principle.

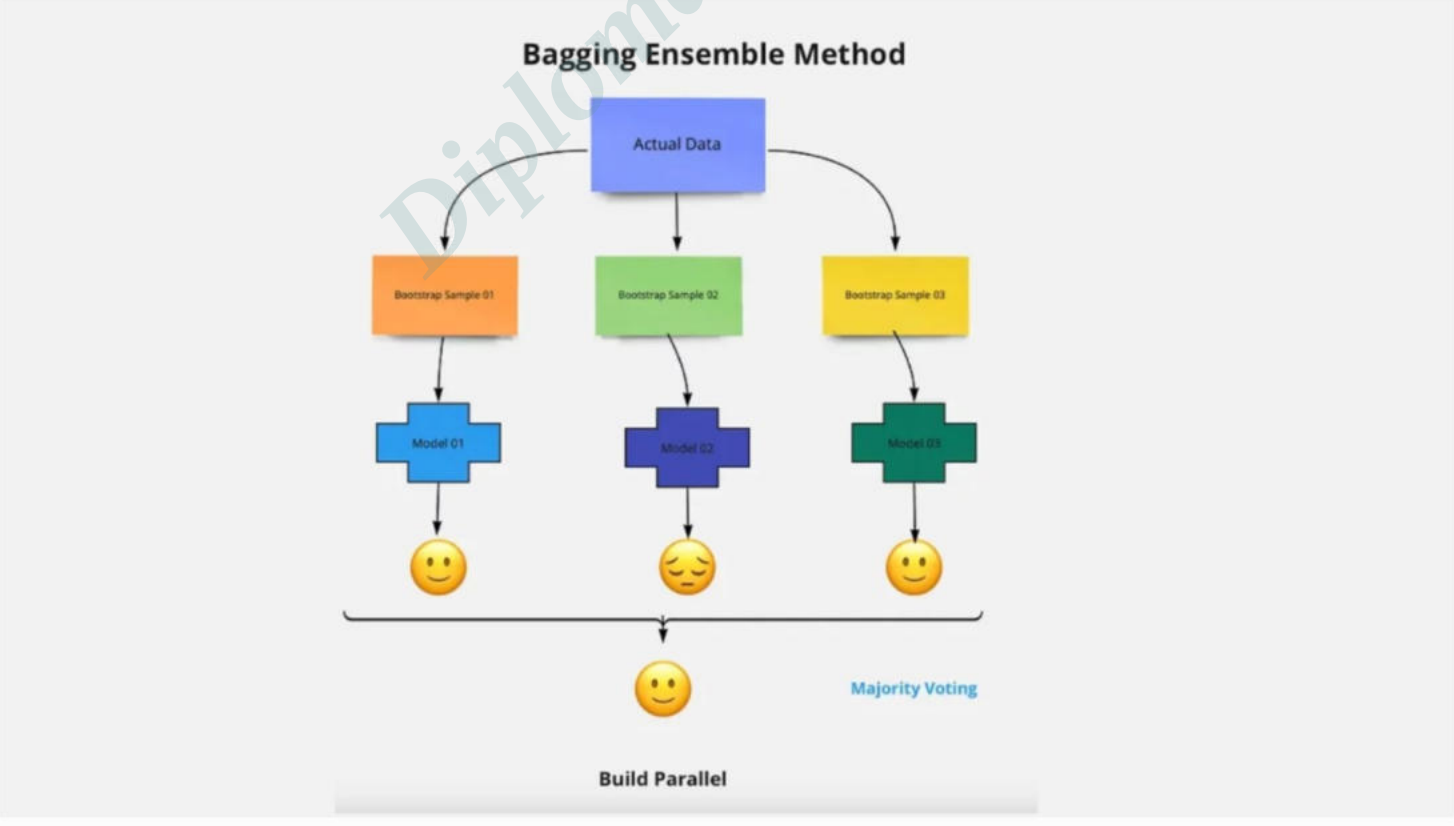


→ **How boosting works :-**

1. **Selection of Subset**: Bagging starts by choosing a random sample, or subset, from the entire dataset.
2. **Bootstrap Sampling**: Each model is then created from these samples, called Bootstrap Samples, which are taken from the original data with replacement. This process is known as row sampling.
3. **Bootstrapping**: The step of row sampling with replacement is referred to as bootstrapping.
4. **Independent Model Training**: Each model is trained independently on its corresponding Bootstrap Sample. This training process generates results for each model.
5. **Majority Voting**: The final output is determined by combining the results of all models through majority voting. The most commonly predicted outcome among the models is selected.
6. **Aggregation**: This step, which involves combining all the results and generating the final output based on majority voting, is known as aggregation.



Now let's look at an example by breaking it down with the help of the following figure. Here the bootstrap sample is taken from actual data (Bootstrap sample 01, Bootstrap sample 02, and Bootstrap sample 03) with a replacement which means there is a high possibility that each sample won't contain unique data. The model (Model 01, Model 02, and Model 03) obtained from this bootstrap sample is trained independently. Each model generates results as shown. Now the Happy emoji has a majority when compared to the Sad emoji. Thus based on majority voting final output is obtained as Happy emoji.



Steps Involved in Random Forest Algorithm

- **Step 1**: In this model, a subset of data points and a subset of features is selected for constructing each decision tree. Simply put, n random records and m features are taken from the

data set having k number of records.

- **Step 2**: Individual decision trees are constructed for each sample.
- **Step 3**: Each decision tree will generate an output.
- **Step 4**: Final output is considered based on Majority Voting or Averaging for Classification and regression, respectively.

→ Important features of Random Forest :-

- Diversity :- each decision tree in the Random forest is built from a different subset of data and feature. this diversity helps in reducing overfitting and improving the model generalization capability.

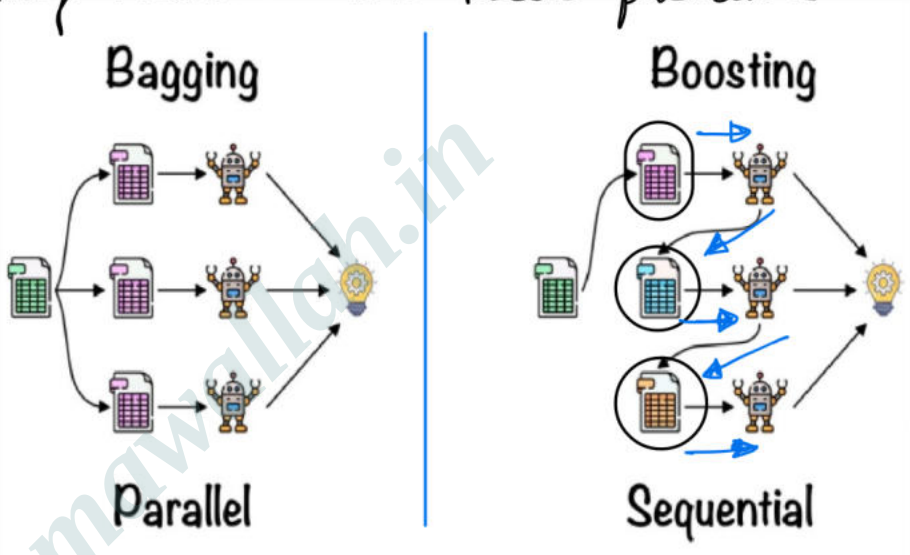
- Handling of Missing Values :- it can handle missing values internally by using surrogate splits or by averaging results from other trees that do not have missing values for the same data points.

- Out-of-Bag error estimation :- RF provides an internal mechanism for estimating the model error without the need for a seperate Validation set.

→ Similar to Random Forest ML algo we have extra tree classifier. by using sklearn you can implement, the main difference b/w RF and extra tree is that, in extra tree we set the random threshold for each each feature rather than searching for the best possible thresholds (like regular Decision Tree do.)

→ RF are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

→ **Boosting** :- Originally called hypothesis boosting, refers to any ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.



→ **AdaBoost** :-

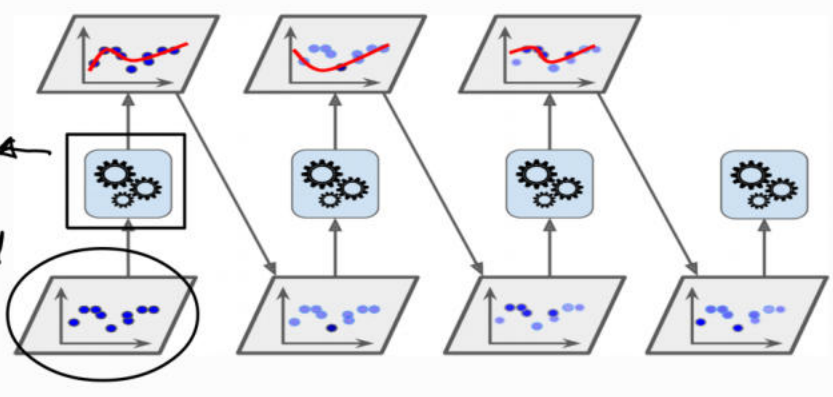
→ One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

→ For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (such as a Decision Tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances.Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on see Figure

↳ first classifier gets many instances wrong, so their weights get boosted. the 2nd clf. therefore does a better job on these instances, and so on

predictor 1

original data



Note:-

→ There is one important drawback to this sequential learning technique: it cannot be parallelized (or only partially), since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

→ Let take a closer look at the Adaboost algorithm. each instance weight $w^{(i)}$ initially set to $1/m$. A first predictor is trained, and its weighted error rate r_1 is computed on the training set:

$$r_j = \frac{\sum_{i=1}^{m} w^{(i)} \cdot \mathbb{1}\left[\hat{y}_j^{(i)} \neq y^{(i)}\right]}{\sum_{i=1}^{m} w^{(i)}}$$

→ $w^{(i)}$ → weight of the i^{th} training example

→ $\hat{y}_j^{(i)}$ → prediction of weak learner for the i^{th} data point

→ y_i → is the true label for i^{th} data point

→ $\mathbb{1}\left(\hat{y}_j^{(i)} \neq y^{(i)}\right)$ → is an indicator function that is 1 if the prediction $\hat{y}_j^{(i)}$ is incorrect otherwise 0.

→ the predictors weight α_j is computed by using weight error using an hyperparameter η. the more accurate the predictor is, the higher its weight will be. if it is just guessing randomly, then its weight will be

close to zero. However if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

$$\alpha_j = \eta \log\left[\frac{1-r_j}{r_j}\right]$$

↳ After this adaboost algorithm updates the instance weights, using below equation, which boosts the weight of the misclassified instances.

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y_j^{(i)} \\ w^{(i)} \times e^{\alpha_j} & \text{if } \hat{y}_j^{(i)} \neq y_j^{(i)} \end{cases}$$

for $i = 1, 2, 3 \cdots m$

↳ then all the instances weights are normalized by dividing $\sum_{i=1}^{m} w^{(i)}$

↳ Finally a new predictor is trained using the updated weights, and the whole process is repeated [the new predictor's weight is computed, and the instance weight are updated, then another predictor is trained and so on]. the algorithm stops when the desired number of predictors is reached, or a when a perfect predictor is found.

↳ To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights αj. The predicted class is the one that receives the majority of weighted votes

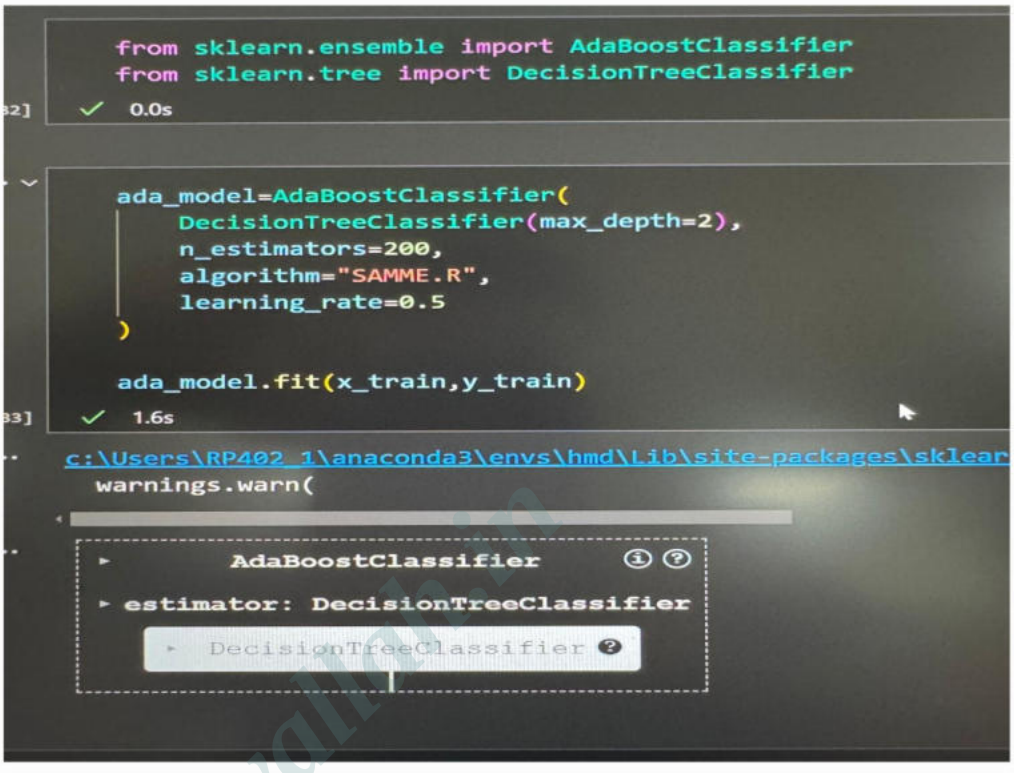
$$\hat{y}(x) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(x)=k}}^{N} \alpha_j$$

N → no of predictors

Notes :-

Scikit-Learn uses a multiclass version of AdaBoost called SAMME (which stands for Stagewise Additive Modeling using a Multiclass Exponential loss function). When there are just two classes, SAMME is equivalent to AdaBoost. If the predictors can estimate class probabilities (i.e., if they have a predict_proba() method), Scikit-Learn can use a variant of SAMME called SAMME.R (the R stands for "Real"), which relies on class probabilities rather than predictions and generally performs better.

Implementation of Adaboost classifier using Decision tree classifier.



Gradient Boosting :-

- Another very popular boosting algorithm is Gradient Boosting. Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual network errors made by the previous predictor.

- Let's go through a simple regression example, using Decision Trees as the base predictors (of course, Gradient Boosting also works great with regression tasks). This is called Gradient Tree Boosting, or Gradient Boosted Regression Trees (GBRT). First, let's fit a DecisionTreeRegressor to the training set (for example, a noisy quadratic training set):

```
from sklearn.tree import DecisionTreeRegressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Next, we'll train a second DecisionTreeRegressor on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

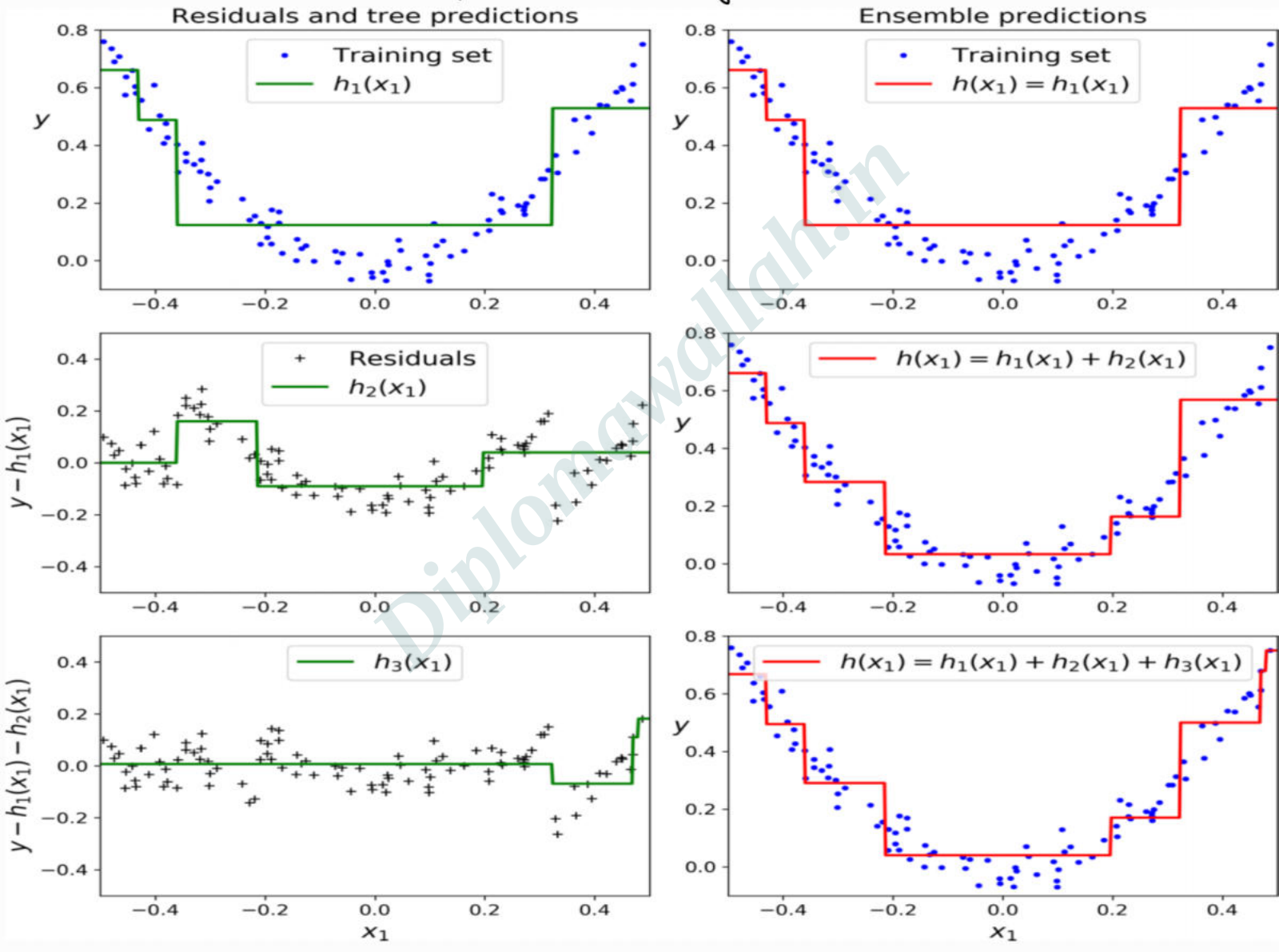
Then we train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

→ below figure represent the prediction of these three trees in left column, and ensemble prediction in right column.



In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions.

- In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right

you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

A simpler way to train GBRT ensembles is to use Scikit-Learn's GradientBoostingRegressor class. Much like the RandomForestRegressor class, it has hyperparameters to control the growth of Decision Trees (e.g., max_depth, min_samples_leaf), as well as hyperparameters to control the ensemble training, such as the number of trees (n_estimators).

The learning_rate hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.1, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called shrinkage.

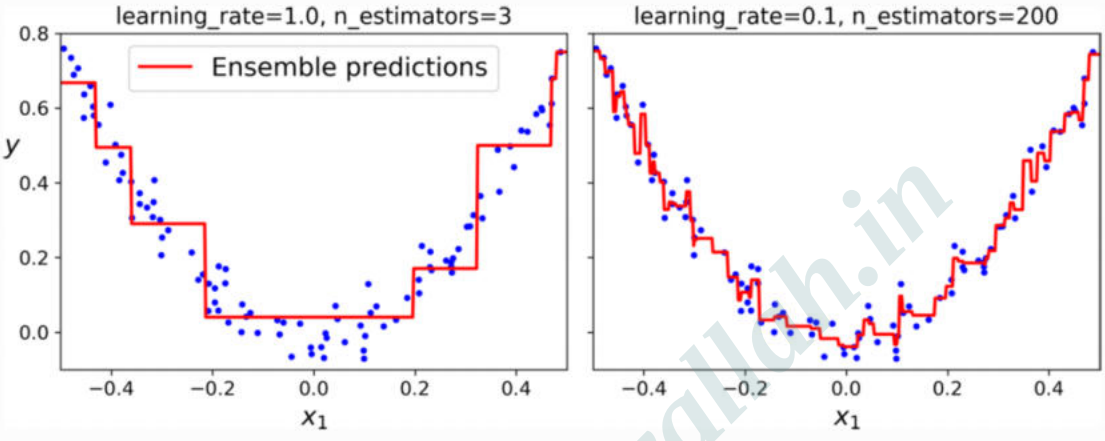
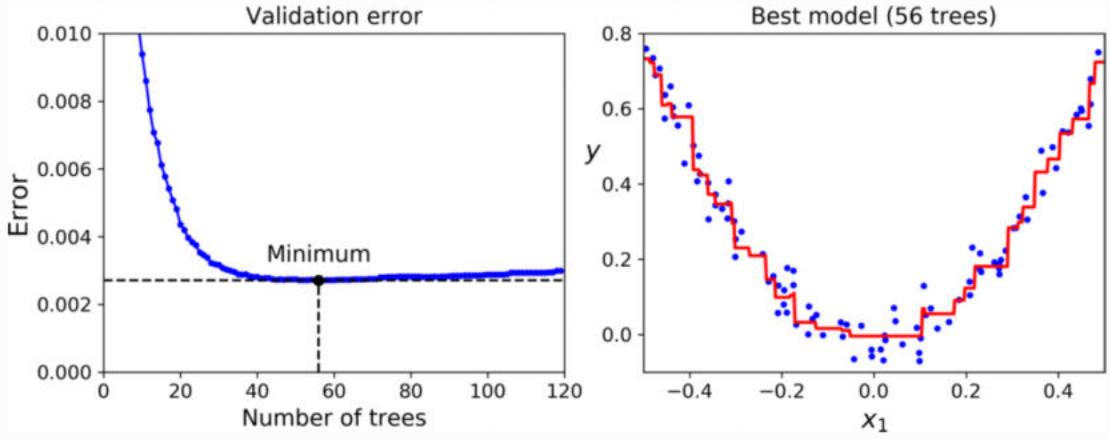


Figure shows two GBRT ensembles trained with a low learning rate: the one on the left does not have enough trees to fit the training set, while the one on the right has too many trees and overfits the training set.

In order to find the optimal number of trees, you can use early stopping. A simple way to implement this is to use the staged_predict() method: it returns an iterator over the predictions made by the ensemble at each stage of training (with one tree, two trees, etc.). The following code trains a GBRT ensemble with120 trees, then measures the validation error at each stage of training to find the opti-
mal number of trees, and finally trains another GBRT ensemble using the optimal number of trees:

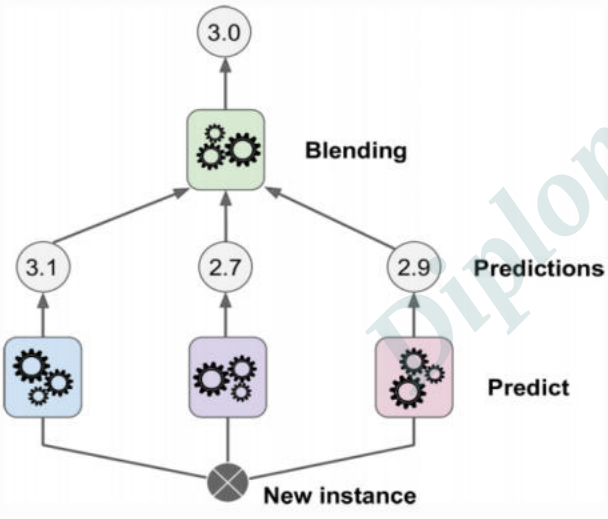
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
X_train, X_val, y_train, y_val = train_test_split(X, y)
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)
errors = [mean_squared_error(y_val, y_pred) for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1
gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

The validation errors are represented on the left of Figure, and the best model's predictions are represented on the right.



Stacking:-

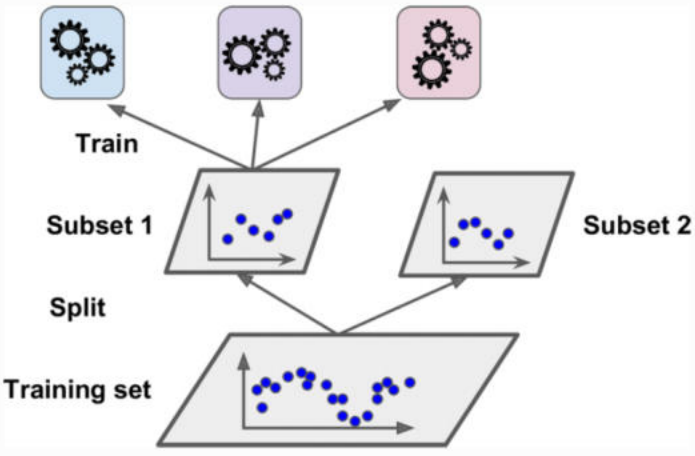
Instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? Figure shows such an ensemble performing a regression task on a new instance. Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor (called a blender, or a meta learner) takes these predictions as inputs and makes the.final prediction (3.0).



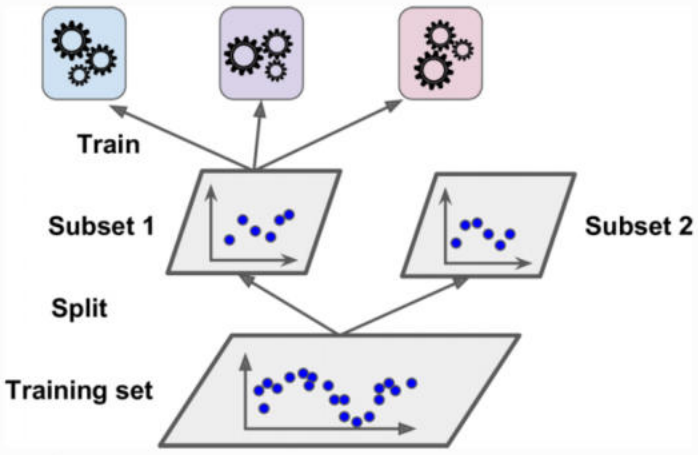
Aggregating predictions using a blending predictor.

To train the blender, a common approach is to use a hold-out set.19 Let's see how it works. First, the training set is split into two subsets. The first subset is used to train the predictors in the first layer (see Figure 1).

Next, the first layer's predictors are used to make predictions on the second (heldout) set (see Figure 2). This ensures that the predictions are "clean," since the predictors never saw these instances during training. For each instance in the hold-out set, there are three predicted values. We can create a new training set using these predicted values as input features (which makes this new training set 3D), and keeping the target values. The blender is trained on this new training set, so it learns to predict the target value, given the first layer's predictions.

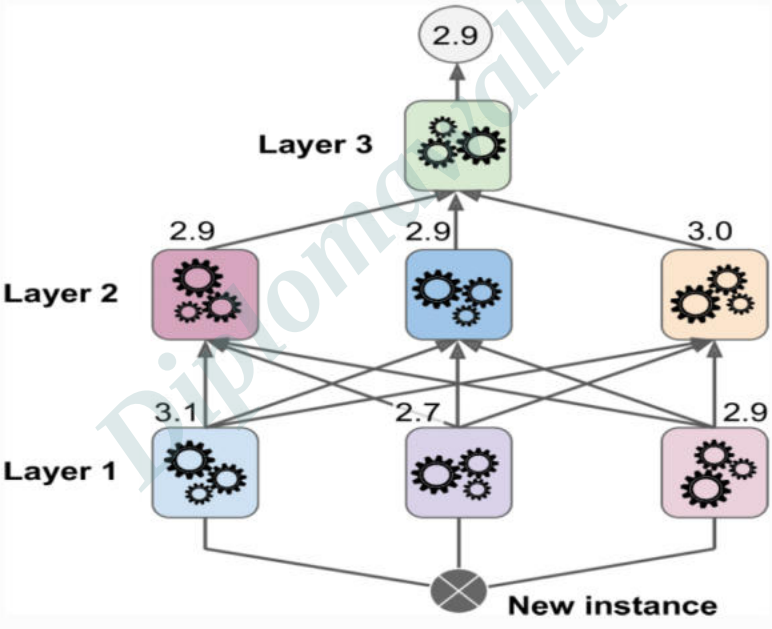


↳ figure 1



↳ figure 2

→ It is actually possible to train several different blenders this way (e.g., one using Linear Regression, another using Random Forest Regression), to get a whole layer of blenders. The trick is to split the training set into three subsets: the first one is used to train the first layer, the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one is used to create the training set to train the third layer (using predictions made by the predictors of the second layer). Once this is done, we can make a prediction for a new instance by going through each layer sequentially, as shown in Figure



:- Chapter-8 Dimensionality Reduction :-

→ Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, as we will see. This problem is often referred to as the curse of dimensionality.

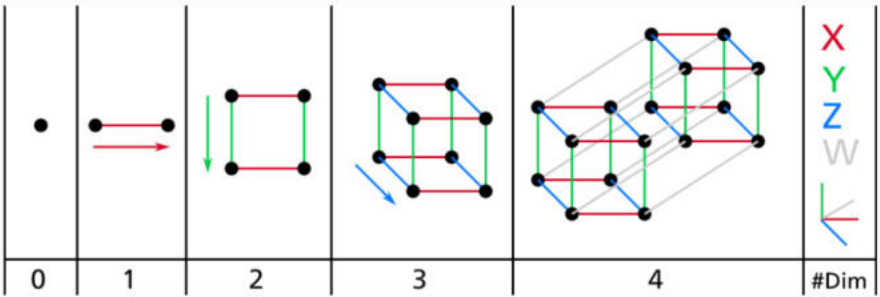
→ Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images, the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information.

→ Reducing dimensionality does cause some information loss (just like compressing an image to JPEG can degrade its quality), so even though it will speed up training, it may make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. So, if training is too slow, you should first try to train your system with the original data before considering using dimensionality reduction. In some cases, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance, but in general it won't; it will just speed up training

→ Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization (or DataViz). Reducing the number of dimensions down to two (or three) makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns,such as clusters.

The Curse of Dimensionality :-

→ We are so used to living in three dimensions that our intuition fails us when we try to imagine high-dimensionale space. Even a basic 4D hypercube is incredibly hard to picture in our minds.



→ figure represent :- point, segment, square, cube and tesseract (0D to 4D hypercubes).

→ the more dimensions, the training set has, the greater the risk of overfitting it.

→ In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instance.

→ Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features (significantly fewer than in the MNIST problem), you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

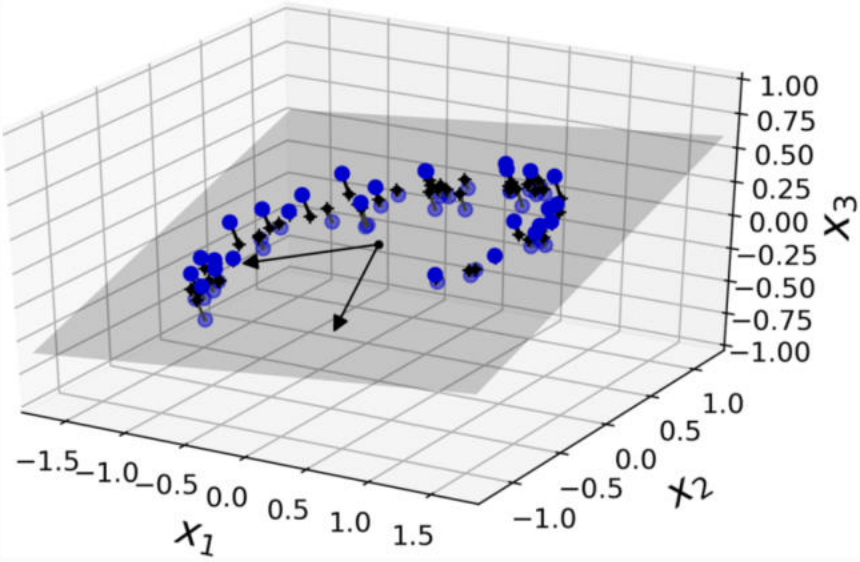
→ Approaches for Dimensionality Reduction :-

↳ Two main approaches to reducing Dimensionality
① Projection ② Manifold Learning

① Projection :-

→ In most real-world problems, training instances are not spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, all training instances lie within (or close to) a much lower-dimensional subspace of the high-dimensional space.

eg:- lets look at the figure 1.

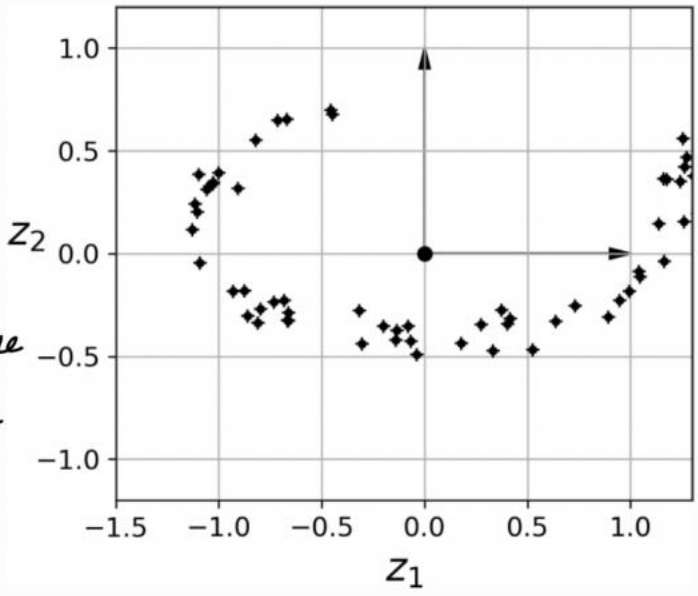


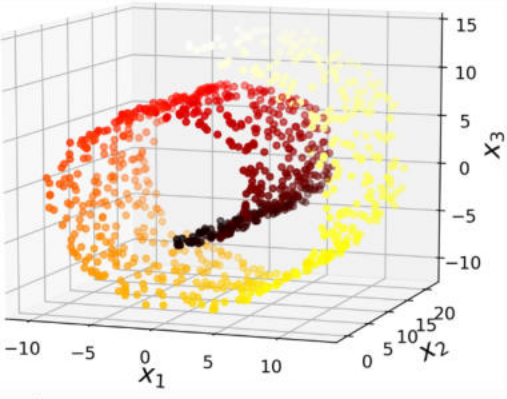
→ figure 1 (3D)

→ all training instance lie close to a plane. this is lower Dimension (2D) subspace of the high-Dimensional (3D) space. if we project every training instances perpendicularly onto the subspace, we get

2D new Dataset.

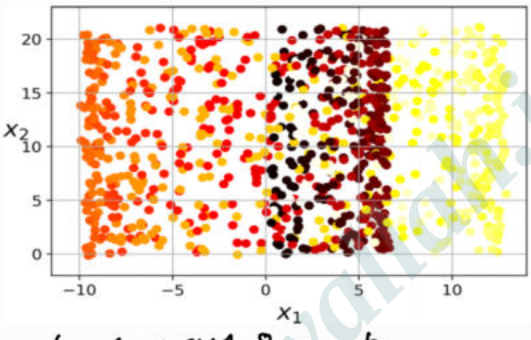
→ However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous Swill roll toy dataset.



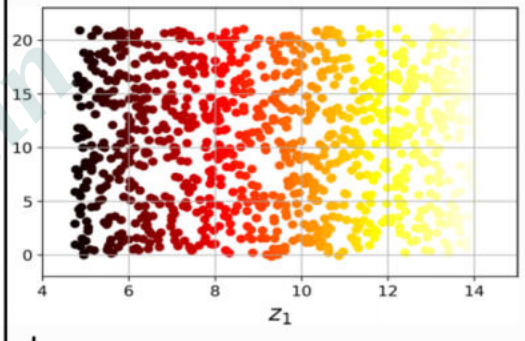


↳ Swiss roll Dataset

↳ 2D New Dataset



↳ Squashing by projecting onto a plane



↳ Unrolling the Swiss roll Dataset

② Manifold learning the swiss roll is an example of mani-fold.

→ 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an n-dimensional space (where d < n) that locally resembles a d-dimensional hyperplane. In the case of the Swiss roll, d = 2 and n = 3: it locally resembles a 2D plane, but it is rolled in the third dimension.

→ Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called Manifold Learning. It relies on the manifold assumption, also called the manifold hypothesis, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

→ Several algo. have been developed under the manifold learning framework each with its unique approach to

uncovering the underlying manifold structure.

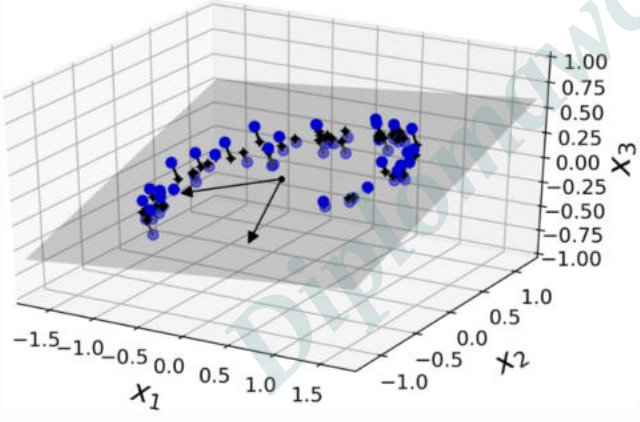
① Isometric feature Mapping (Isomap)

② locally linear embedding (LLE)

③ Laplacian Eigenmaps

④ t-Distributed Stochastic Neighbor

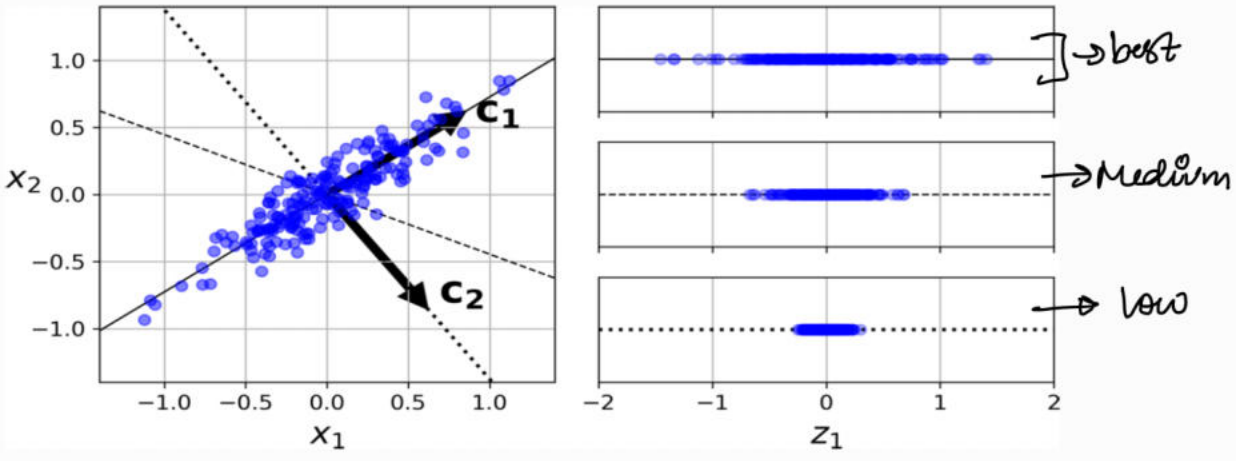
→ PCA [principal component Analysis (PCA)] :- is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.



→ like in this figure the PCA is that line where we project our data points.

→

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left in Figure, along with three different axes (i.e., 1D hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance, while the projection onto the dotted line preserves very little variance and the projection onto the dashed line preserves an intermediate amount of variance.



It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA.

Principal components :-

PCA identifies the axis that accounts for the largest amount of variance in the training set. In above Figure, it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

↳ the i^{th} axis is called i^{th} principal component (PC) of the Data. In above figure the first PC is the axis on which Vector c_1 lies, and the second PC is the axis on which Vector c_2 lies.

→ To find the principal components of a training set, there is a standard matrix factorization technique called Singular Value Decomposition (SVD). SVD decomposes the training set matrix X into the matrix multiplication of three matrices '$U \Sigma V^T$' where 'V' contains the unit vector that defines all the principal components that we are looking for,

$$V = \begin{bmatrix} | & | & | & | \\ c_1 & c_2 & c_3 & \cdots & c_n \\ | & | & | & | \end{bmatrix}$$

$$X = U \Sigma V^T$$

The following Python code uses NumPy's svd() function to obtain all the principal components of the training set, then extracts the two unit vectors that define the first two PCs:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

Note :-

PCA assumes that the dataset is centered around the origin. As we will see, Scikit-Learn's PCA classes take care of centering the data for you. If you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

→ Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible.

→ To project the training set onto the hyperplane and obtain a reduced dataset $X_{d\text{-proj}}$ of dimensionality d, compute the matrix multiplication of the training set matrix X by the matrix W_d, defined as the matrix containing the first d columns of V.

$$X_{d\text{-proj}} = X W_d$$

$X_{d\text{-proj}}$:- reduced dataset of dimension 'd'

W_d → first 'd' columns of matrix V.

→

The following Python code projects the training set onto the plane defined by the first two principal components:

```
→ W2 = Vt.T[:, :2]
→ X2D = X_centered.dot(W2)
```
] → dot product.

→ In sklearn code will look like

```
→ from sklearn.decomposition import PCA
→ pca = PCA(n_components = 2)
→ X2D = pca.fit_transform(X)
```

n_components :- dimension at which we want to reduce.

→

Another useful piece of information is the explained variance ratio of each principal component, available via the explained_variance_ratio_ variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in

→ `pca.explained_variance_ratio_`

↳ output

array([0.84248607, 0.14631839])

↳ this output tells you that 84% of the dataset's variance lies along the first PC, and 14.6% lies along the second PC. this leaves us that 1.2% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

→ Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will want to reduce the dimensionality down to 2or 3.
The following code performs PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
→ pca = PCA()
→ pca.fit(X_train)
→ cumsum = np.cumsum(pca.explained_variance_ratio_)
→ d = np.argmax(cumsum >= 0.95) + 1
```

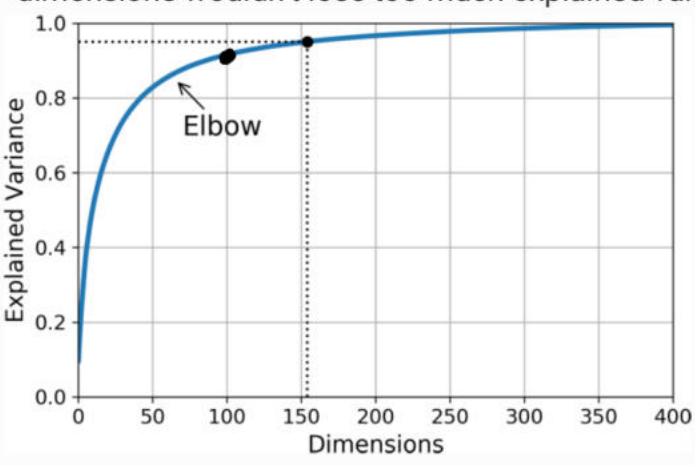
↳ you could then set n_component = d and run PCA again.

→ there is a much better option: instead of specifying the number of principal components you want to preserve, you can set n_components to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
→ pca = PCA(n_components=0.95)
→ X_reduced = pca.fit_transform(X_train)
```

→ Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot cumsum; see Figure). There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

cumsum → cumulative sum



→ explained variance as a function of the number of dimensions.

→ PCA for compression:-

After dimensionality reduction, the training set takes up much less space. As an example, try applying PCA to the MNIST dataset while preserving 95% of its variance. You should find that each instance will have just over 150 features, instead of the original 784 features. So, while most of the variance is preserved, the dataset is now less than 20% of its original size! This is a reasonable compression ratio, and you can see how this size reduction can speed up a classification algorithm (such as an SVM classifier) tremendously.

→ it is also possible to decompress the reduced dataset back to original dimensions by applying the inverse transformation of pca projection. this won't give you back the original data, since projection lost a bit of information, but it will likely be close to the original data.

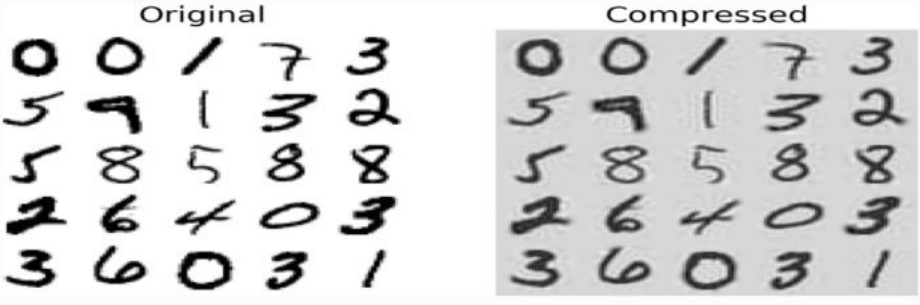
→ the mean squared distance between the original data and reconstructed data (compressed and then decompressed) is called reconstruction error.

→

The following code compresses the MNIST dataset down to 154 dimensions, then uses the inverse_transform() method to decompress it back to 784 dimensions:

```
→ pca = PCA(n_components = 154)
→ X_reduced = pca.fit_transform(X_train)
→ X_recovered = pca.inverse_transform(X_reduced)
```

→ Figure shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.



→ Mnist compression that preserves 95% variance

$$X_{recovered} = X_{d\text{-}proj} W_d^T$$

→ this equation use to decompress or PCA inverse transformation

If you set the svd_solver hyperparameter to "randomized", Scikit-Learn uses a stochastic algorithm called Randomized PCA that quickly finds an approximation of the first d principal components. Its computational complexity is. $O(m \times d^2) + O(d^3)$ instead of. $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when d is much smaller than n:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

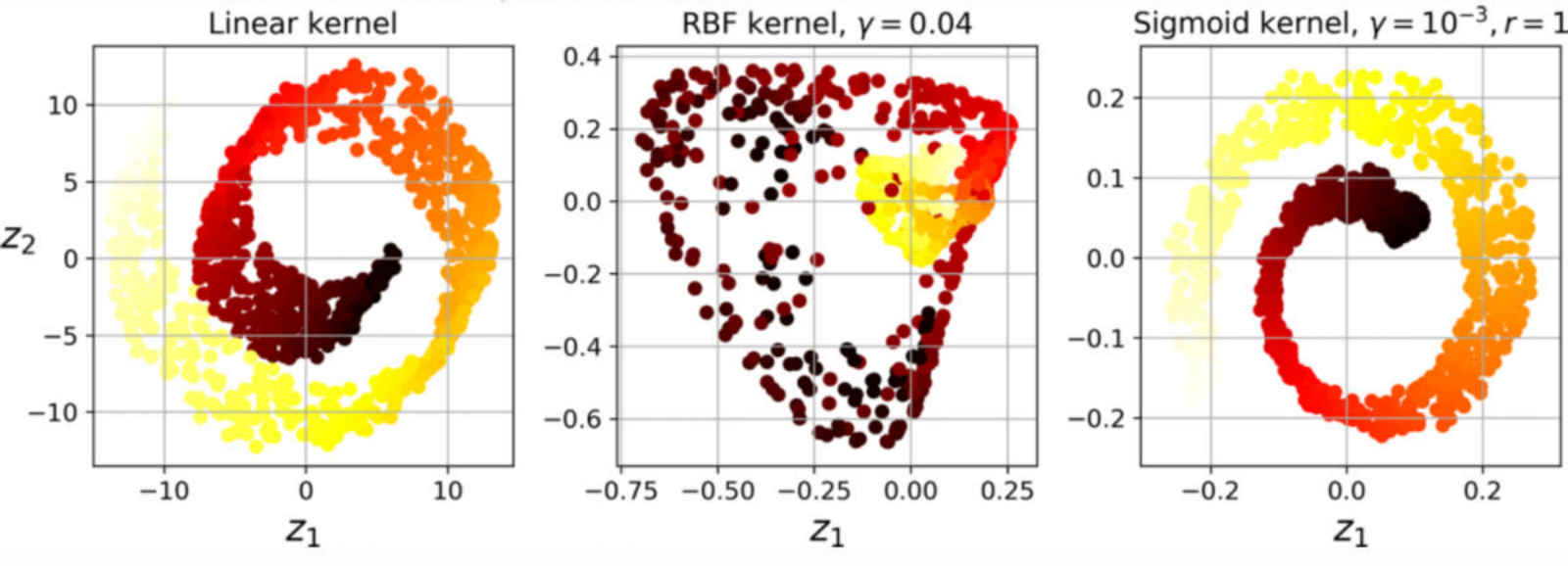
By default, svd_solver is actually set to "auto": Scikit-Learn automatically uses the randomized PCA algorithm if m or n is greater than 500 and d is less than 80% of m or n, or else it uses the full SVD approach. If you want to force Scikit-Learn to use full SVD, you can set the svd_solver hyperparameter to "full".

One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, Incremental PCA (IPCA) algorithms have been developed. They allow you to split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time.This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive)

Kernel PCA :-

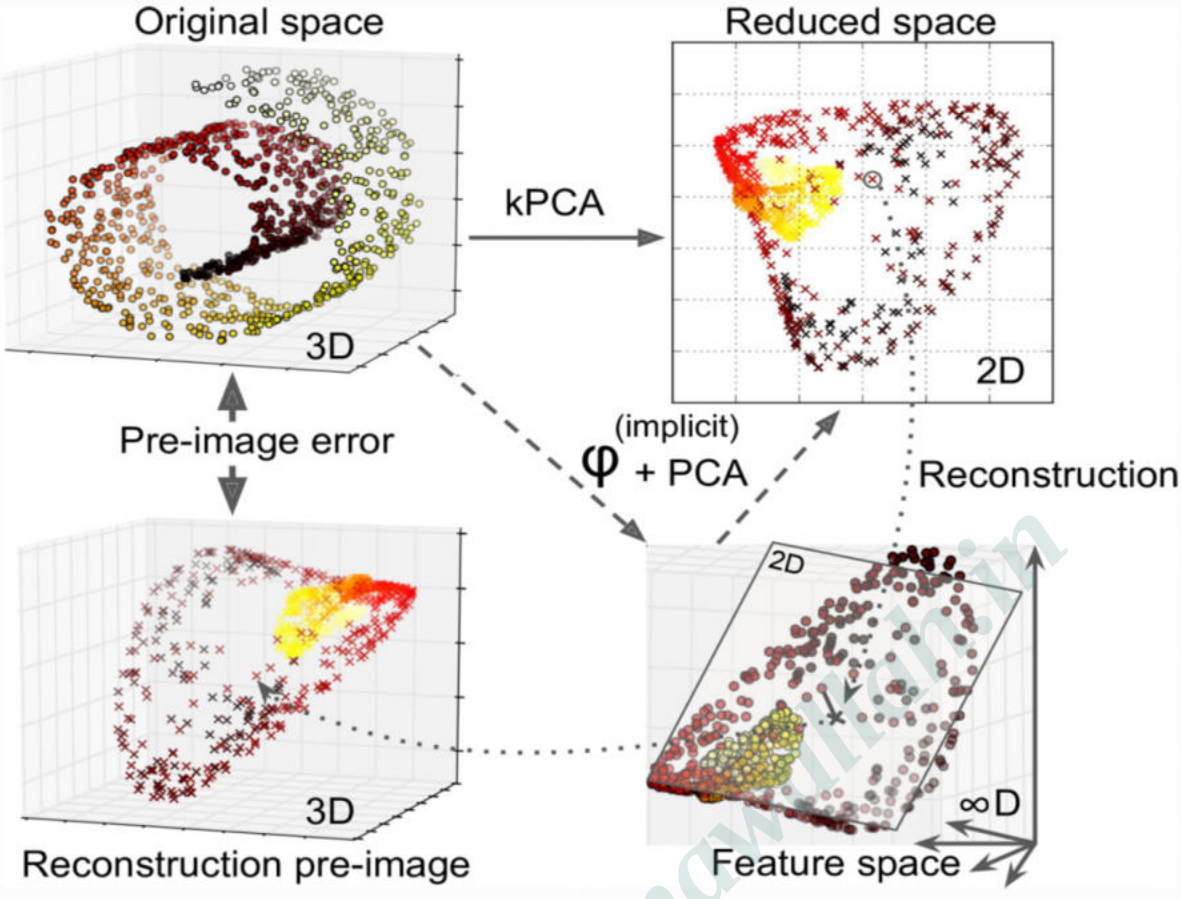
the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space. It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel PCA (KPCA).

```
from sklearn.decomposition import KernelPCA
rf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```



shows the Swiss roll, reduced to two dimensions using a linear kernel(equivalent to simply using the PCA class), an RBF kernel, and a sigmoid kernel.

→ Another approach, this time entirely unsupervised, is to select the kernel and hyperparameters that yield the lowest reconstruction error. Note that reconstruction is not as easy as with linear PCA. Here's why. Figure shows the original Swiss roll 3D dataset (top left) and the resulting 2D dataset after kPCA is applied using an RBF kernel (top right). Thanks to the kernel trick, this transformation is mathematically equivalent to using the feature map φ to map the training set to an infinitedimensional feature space (bottom right), then projecting the transformed training set down to 2D using linear PCA.



→ LLE [locally linear Embedding] :-

Locally Linear Embedding (LLE) is another powerful nonlinear dimensionality reduction (NLDR) technique. It is a Manifold Learning technique that does not rely on projections, like the previous algorithms do. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved. This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

→ Other Dimensionality Reduction Techniques :-

1. Random Projections
2. Multidimensional scaling (MDS)
3. Isomap
4. t-SNE
5. LDA

— : Chapter - 09 :-
— : Unsupervised learning Techniques :-

→ Although most of the applications of Machine Learning today are based on supervised learning (and as a result, this is where most of the investments go to), the vast majority of the available data is unlabeled: we have the input features X, but we do not have the labels y. The computer scientist Yann LeCun famously said that "if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake." In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

→ In this we will learn

Clustering

The goal is to group similar instances together into *clusters*. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

Anomaly detection

The objective is to learn what "normal" data looks like, and then use that to detect abnormal instances, such as defective items on a production line or a new trend in a time series.

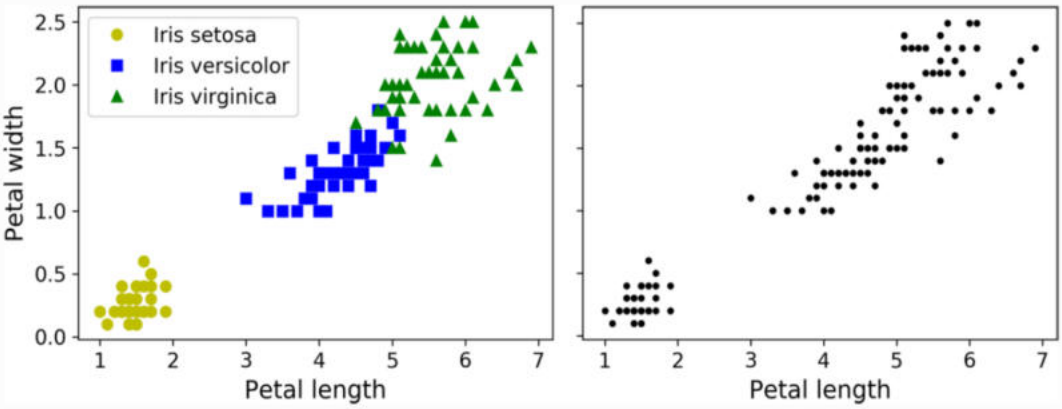
Density estimation

This is the task of estimating the *probability density function* (PDF) of the random process that generated the dataset. Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

→ ## clustering :-

- it is the task of identifying similar instances and assigning them to clusters, or groups of similar instances. is called clustering.

→ Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task. Consider Figure : on the left is the iris dataset , where each instance's species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as Logistic Regression, SVMs, or Random Forest classifiers are well suited. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore. This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two distinct sub-clusters. That said, the dataset has two additional features (sepal length and width), not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well (e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster).



→ classification left
VS clustering Right

→ clustering is used in a wide variety of applications, including these:

① For customer segmentation :- you can cluster your customers based on their purchases and their activity on your website

② Data Analysis :- When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster seperately.

③ As a dimensionality Reduction Technique :-

④ For search engine

⑤ For semi supervised learning

⑥ To segment an image.

→

There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a centroid. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

K Means Algorithm Overview

The **K-means algorithm** is a popular unsupervised machine learning method used for clustering data into K distinct groups based on feature similarity. Mathematically, it partitions a set of n data points in a d-dimensional space into K clusters $\{C_1, C_2, \ldots, C_K\}$ such that the within-cluster sum of squares (WCSS) is minimized. Below is a detailed mathematical exposition of the K-means algorithm.

1. Mathematical Formulation

1.1. Data Representation

Let $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ be a set of n data points, where each $\mathbf{x}_i \in \mathbb{R}^d$ is a d-dimensional vector.

1.2. Cluster Centers

The goal is to find K cluster centers (also called centroids) $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K\}$, where each $\boldsymbol{\mu}_k \in \mathbb{R}^d$, that best represent the clusters.

1.3. Objective Function

K-means aims to minimize the **within-cluster sum of squares (WCSS)**, which is defined as:

$$\text{WCSS} = \sum_{k=1}^{K} \sum_{\mathbf{x}_i \in C_k} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|^2$$

Where:

- C_k is the set of points assigned to cluster k.
- $\|\mathbf{x}_i - \boldsymbol{\mu}_k\|^2$ is the squared Euclidean distance between point $\mathbf{x}_i$ and centroid $\boldsymbol{\mu}_k$.

1.4. Optimization Problem

Formally, the K-means clustering problem can be stated as:

$$\min_{\{C_k\},\{\boldsymbol{\mu}_k\}} \sum_{k=1}^{K} \sum_{\mathbf{x}_i \in C_k} \|\mathbf{x}_i - \boldsymbol{\mu}_k\|^2$$

Subject to:

- $C_k \subseteq \mathcal{X}$ for all k.
- $\bigcup_{k=1}^{K} C_k = \mathcal{X}$.
- $C_k \cap C_{k'} = \emptyset$ for all $k \neq k'$.

2. Algorithm Steps

The K-means algorithm iteratively optimizes the objective function through two main steps: **Assignment** and **Update**.

2.1. Initialization

- **Random Initialization**: Select K distinct data points randomly from $\mathcal{X}$ as the initial centroids $\{\boldsymbol{\mu}_1^{(0)}, \boldsymbol{\mu}_2^{(0)}, \ldots, \boldsymbol{\mu}_K^{(0)}\}$.
- **Alternative Methods**: Techniques like K-means++ can be used to choose initial centroids to improve convergence and solution quality.

2.2. Assignment Step

Given the current centroids $\{\boldsymbol{\mu}_1^{(t)}, \boldsymbol{\mu}_2^{(t)}, \ldots, \boldsymbol{\mu}_K^{(t)}\}$ at iteration t, assign each data point to the nearest centroid:

$$C_k^{(t)} = \left\{ \mathbf{x}_i \in \mathcal{X} \;\middle|\; \|\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)}\|^2 \leq \|\mathbf{x}_i - \boldsymbol{\mu}_{k'}^{(t)}\|^2 \; \forall \, k' \in \{1, 2, \ldots, K\} \right\}$$

2.3. Update Step

Recompute the centroids as the mean of all data points assigned to each cluster:

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{1}{|C_k^{(t)}|} \sum_{\mathbf{x}_i \in C_k^{(t)}} \mathbf{x}_i \quad \text{for each } k = 1, 2, \ldots, K$$

2.4. Convergence Criteria

The algorithm repeats the Assignment and Update steps until one of the following conditions is met:

1. **Centroid Stabilization**: The centroids do not change significantly between iterations, i.e., $\|\boldsymbol{\mu}_k^{(t+1)} - \boldsymbol{\mu}_k^{(t)}\| < \epsilon$ for all k, where ϵ is a small threshold.
2. **Maximum Iterations**: A predefined maximum number of iterations is reached.
3. **No Change in Assignments**: The cluster assignments $\{C_k\}$ do not change between consecutive iterations.

3. Mathematical Properties

3.1. Convergence

K-means is guaranteed to converge to a local minimum of the WCSS objective function. However, it may not find the global minimum due to its dependence on the initial centroid positions.

3.2. Computational Complexity

Each iteration of the K-means algorithm has a computational complexity of $O(nKd)$, where:

- n is the number of data points.
- K is the number of clusters.
- d is the dimensionality of the data.

The total complexity depends on the number of iterations until convergence.

3.3. Optimality

K-means solves the clustering problem via **Lloyd's algorithm**, which is an instance of the **Expectation-Maximization (EM)** algorithm for Gaussian mixtures with equal spherical covariance and equal priors. However, K-means assumes clusters are convex and isotropic, which may not hold in all datasets.

4. Extensions and Variations

Several variations of the K-means algorithm have been proposed to address its limitations:

- **K-means++**: Improves initialization by spreading out the initial centroids, leading to better convergence properties.

- **Mini-Batch K-means**: Uses small random subsets (mini-batches) of data for updates, enhancing scalability for large datasets.
- **Kernel K-means**: Extends K-means to non-linear cluster boundaries by applying kernel functions.

5. Example

Consider a simple 2-dimensional dataset with $n = 4$ points:

$$\mathcal{X} = \{\mathbf{x}_1 = (1, 2),\ \mathbf{x}_2 = (1, 4),\ \mathbf{x}_3 = (1, 0),\ \mathbf{x}_4 = (10, 2)\}$$

Let $K = 2$.

Initialization:

- Suppose we randomly choose $\boldsymbol{\mu}_1^{(0)} = (1, 2)$ and $\boldsymbol{\mu}_2^{(0)} = (10, 2)$.

Assignment:

- $\mathbf{x}_1$ and $\mathbf{x}_2$ are closer to $\boldsymbol{\mu}_1^{(0)}$.
- $\mathbf{x}_3$ is closer to $\boldsymbol{\mu}_1^{(0)}$.
- $\mathbf{x}_4$ is assigned to $\boldsymbol{\mu}_2^{(0)}$.

Update:

- New $\boldsymbol{\mu}_1^{(1)} = \frac{1}{3}\left((1, 2) + (1, 4) + (1, 0)\right) = (1, 2)$
- New $\boldsymbol{\mu}_2^{(1)} = (10, 2)$ (unchanged)

Convergence:

- Since the centroids did not change, the algorithm converges with the final clusters:
  - $C_1 = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$
  - $C_2 = \{\mathbf{x}_4\}$

6. Limitations

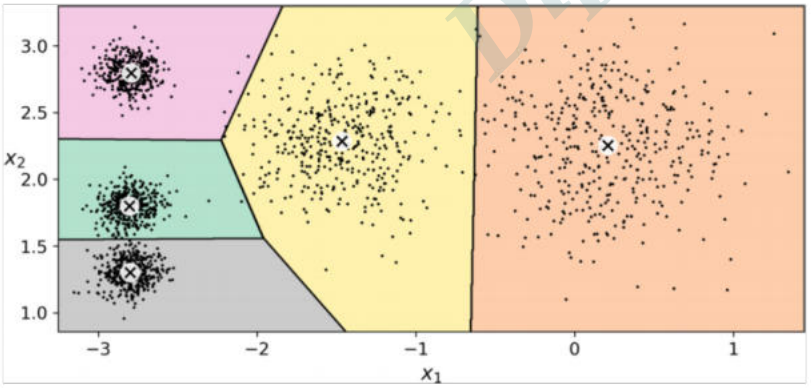
- **Choosing K**: Determining the optimal number of clusters K is non-trivial and often requires methods like the Elbow Method or Silhouette Analysis.

- **Sensitivity to Initialization**: Poor initial centroid selection can lead to suboptimal clustering.
- **Assumption of Spherical Clusters**: K-means works best when clusters are spherical and equally sized, which may not hold for all datasets.
- **Scalability**: While efficient for small to medium-sized datasets, K-means can be computationally intensive for very large datasets without modifications like Mini-Batch K-means.

7. Conclusion

Mathematically, K-means is an iterative optimization algorithm aimed at partitioning data into K clusters by minimizing the within-cluster variance. Its simplicity and efficiency make it a widely used clustering technique, though it comes with assumptions and limitations that must be considered in practical applications.

→ from sklearn.cluster import KMeans
→ K=5
→ kmeans = KMeans(n_clusters=k)
→ y_pred = kmeans.fit_predict(X)
→ code for KMeans



→ KMeans decision boundaries where each centroid is represented as X

The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled (especially near the boundary between the top-left cluster and the central cluster). Indeed, the K-Means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid. Instead of assigning each instance to a single cluster, which is called hard clustering, it can be useful to give each instance a score per cluster, which is called soft clustering. The score can be the distance between the instance and the centroid; conversely, it can be a similarity score (or affinity), such as the Gaussian Radial Basis Function.

The computational complexity of the algorithm is generally linear with regard to the number of instances m, the number of clusters k, and the number of dimensions n. However, this is only true when the data has a clustering structure. If it does not, then in the worstcase scenario the complexity can increase exponentially with the number of instances. In practice, this rarely happens, and K-Means is generally one of the fastest clustering algorithms.

→ Centre initialization method :-

If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the init hyperparameter to a NumPy array containing the list of centroids, and set n_init to 1:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the n_init hyperparameter: by default, it is equal to 10, which means that the whole algorithm described earlier runs 10 times when you call fit(), and ScikitLearn keeps the best solution. But how exactly does it know which solution is the best? It uses a performance metric! That metric is called the model's inertia, which is the mean squared distance between each instance and its closest centroid.

→ An important improvement to the K-Means algorithm, K-Means++, was proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii.3 They introduced a smarter initialization step that tends to select centroids that are distant from one another, and this improvement makes the K-Means algorithm much less likely to converge to a suboptimal solution. They showed that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution.

→ Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm typically by a factor of three or four and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the MiniBatchKMeans class. You can just use this class like the KMeans class:

```
from sklearn.cluster import MiniBatchKMeans
minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

→ Minibatch is much faster than normal KMeans and the difference increases with k. (k → no of centroids)

→ It is important to scale the input features before you run K-Means or the clusters may be very stretched and K-Means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally improves things.

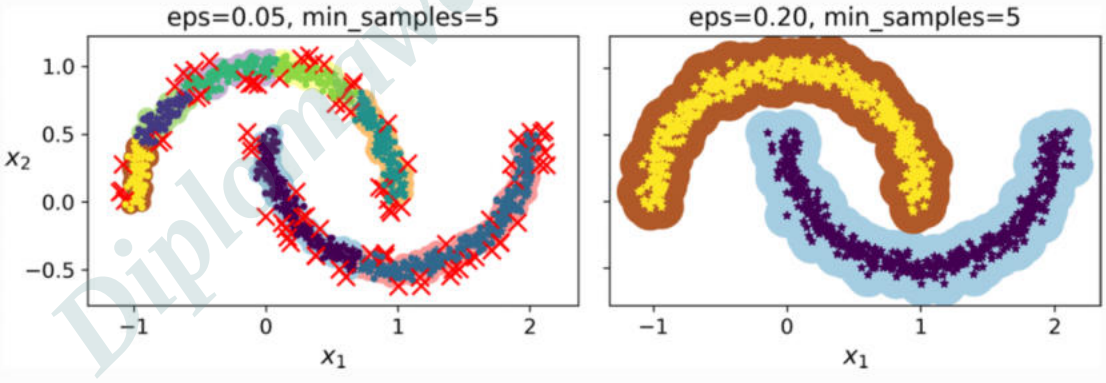
→ DBSCAN :-

This algorithm defines clusters as continuous regions of high density. Here is how it works:

- For each instance, the algorithm counts how many instances are located within a small distance ε (epsilon) from it. This region is called the instance's εneighborhood.
- If an instance has at least min_samples instances in its ε-neighborhood (including itself), then it is considered a core instance. In other words, core instances are those that are located in dense regions.
- All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly

→ this algorithm works well if all the clusters are dense enough and if they are well separated by low density regions.

→ This clustering is represented in the lefthand plot of . As you can see, it identified quite a lot of anomalies, plus seven different clusters. How disappointing! Fortunately, if we widen each instance's neighborhood by increasing eps to 0.2, we get the clustering on the right, which looks perfect. Let's continue with this model.



eps → neighborhood radius

→ DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (eps and min_samples). If the density varies significantly across the clusters, however, it can be impossible for it to capture all the clusters properly. Its computational complexity is roughly O(m log m), making it pretty close to linear with regard to the number of instances, but Scikit-Learn's implementation can require up to. $O(m^2)$ memory if eps is large.