Supervised Machine Learning II Classification

T5 Bootcamp by SDAIA

Machine Learning II: Classification

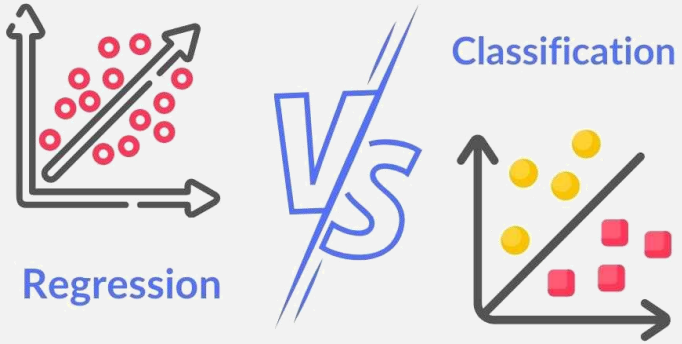
SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority

Agenda

- Supervised Learning - Classification
- Standardization & Normalization
- Binary Classification
- Imbalanced Dataset
- Performance Metrics
- Multiclass/Multilabel Classification
- Versatile Models
- Overfitting & Regularization
- Ensemble Learning

Supervised Learning

- Was it trained with human supervision? (supervised, unsupervised, semisupervised, and Reinforcement Learning)
- Supervised Learning: When the training data you feed to the algorithm includes the desired solutions, called labels
- Tasks : **Classification**, Regression
- Algorithms: K-Nearest Neighbors, Linear Regression, Logistic Regression, Support Vector Machines (SVMs), Decision Trees and Random Forests, Neural Networks



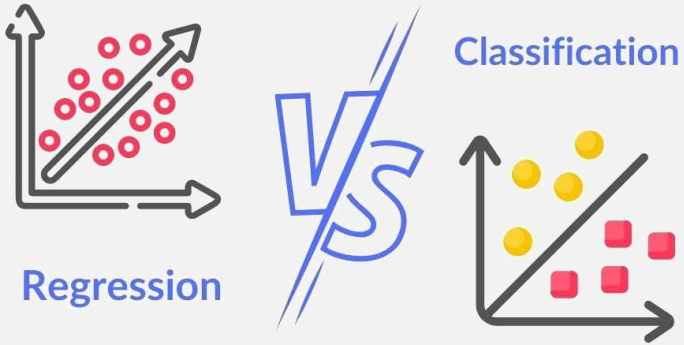
Classification

Classification is a process in machine learning where an algorithm is trained on a dataset with known labels, learning to recognize patterns and features. Once trained, it can categorize new, unseen data into these learned labels, effectively sorting the data based on its input features.

Common Uses:

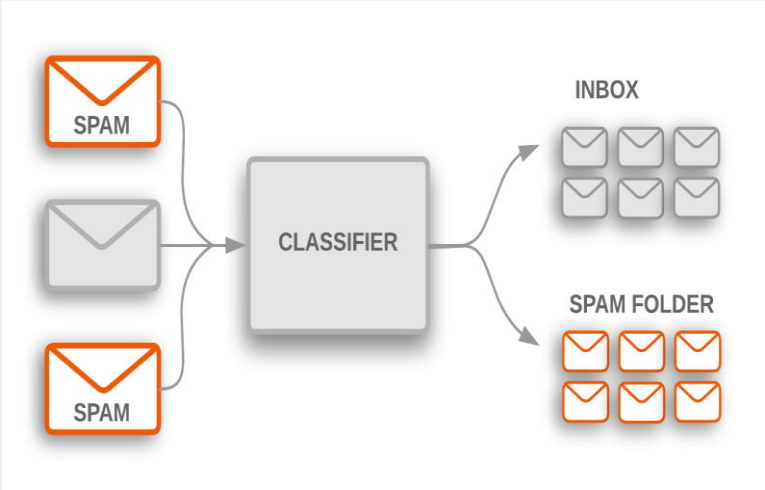
- Spam Email Detection: Classifying emails as spam or not spam.
- Disease Diagnosis: Detects diseases in patients.
- Sentiment Analysis: Evaluates emotions in written text.
- Image Classification: Labels images by content.
- Churn Prediction : Anticipates customer retention or turnover.

Key Algorithms: Employs techniques like logistic regression, decision trees, random forests, SVMs, K-NN, and neural networks for varied prediction complexity.



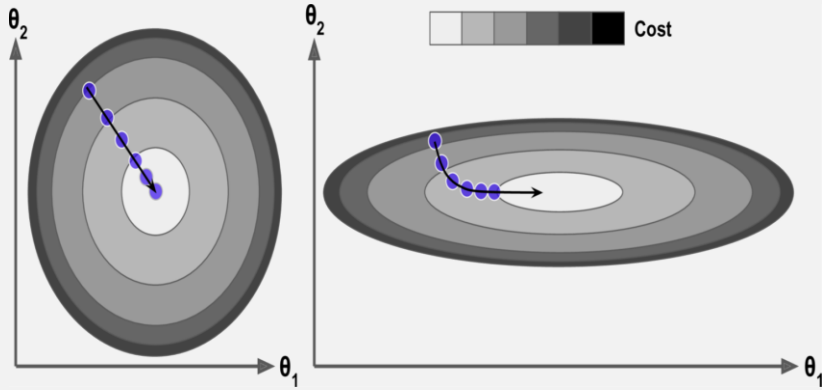
Spam Filter Example

- Spam filter is a Machine Learning program. It flags spam or not given examples of spam emails (e.g., flagged by users) and examples of regular (nonspam, also called “ham”) emails.
- **performance measure P** needs to be defined: the ratio of correctly classified emails. (**accuracy**).



Feature Scaling (Normalization)

Feature scaling normalizes numerical data in a dataset to a common scale, It adjusts numerical data in a dataset to a common scale, ensuring that features contributes equally to the analysis without being biased by their original magnitude.



The advantages of Feature Scaling

- Enhances algorithm performance by ensuring that features contribute equally to the learning process.
- Mitigates impact of feature magnitudes on model training by preventing features with larger scales from dominating the learning process.
- Faster convergence
- Easier to interpret model coefficients and understand their relative importance
- Increase ML stability
- Facilitates regularization

Feature Scaling (Normalization)

Methods of Feature Scaling

Min-Max Scaler:

- Scales features to a specified range, typically between 0 and 1, preserving the relative distances between data points.
- Suitable for algorithms sensitive to feature scaling, such as neural networks, but may be affected by outliers.

MinMax Scaler	$\frac{x_i - \min(\boldsymbol{x})}{\max(\boldsymbol{x}) - \min(\boldsymbol{x})}$

Standard Scaler:

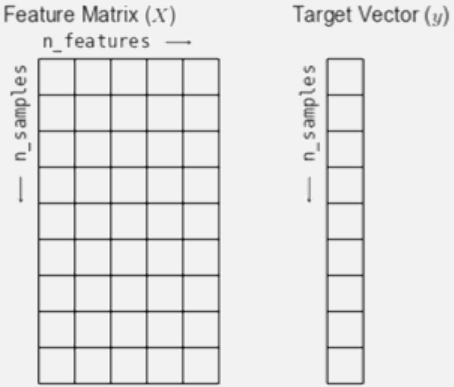
- Transforms features to have a mean of 0 and a standard deviation of 1, ensuring each feature has a similar scale.
- Robust to outliers and suitable for algorithms relying on normal distribution assumptions, like linear regression and logistic regression.

Standard Scaler	$\frac{x_i - \text{mean}(\boldsymbol{x})}{\text{stdev}(\boldsymbol{x})}$

Notation in ML

The following is common notations that is used in ML:

- **m** : size of the dataset (images)
- **$X^{(i)}$** : the feature values (no label) of the ith instance, (28 x 28 pixels) = 784 (X0, X1, …X784)
- **$y^{(i)}$** is its label (the desired output value for that instance) (example digit 5)
- **X** is a matrix containing all the feature values
- **Y** is a vector containing all labels
- **h** is your system's prediction function, also called a hypothesis. When your system is given an instance's feature vector x(i), it outputs a predicted value **$\hat{y}(i) = h(x(i))$** for that instance ($\hat{y}$ is pronounced "y-hat").



Numbers characters

Classification

Classification Use Case

MNIST DATA SET – Hello World

Let's understand the data used in our use-case!!

- Size : 70,000 smallimages of digits handwritten by high school students and employees of the US Census Bureau.
- (28 pixel x 28 pixel) = 784
- The label: [0-9]
- "Hello World" of ML
- Scikit-Learn provides many helper functions to download popular datasets including MNIST data set

Data Link: http://yann.lecun.com/exdb/mnist/

Follow lab:

T5B3ML101N01-0424-MNIST_DummyClassifier

To get to know the data and build a dummy classifier

Get the data

The first step of tackling any new dataset is always to import the data and explore its Raw form
Find below the code for fetching the data and exploring its shape:

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
dict_keys(['data', 'target', 'feature_names',
'DESCR', 'details', 'categories', 'url'])
```

- A **DESCR** key describing the dataset
- A **data key** containing an array with one row per instance and one column per feature
- A **target key** containing an array with the labels

```
X, y = mnist["data"], mnist["target"]
X.shape
(70000, 784)
y.shape
(70000,)
```

m =70,000 images x =784 features (28×28 pixels)
Each feature simply represents one pixel's intensity, from 0 (white) to 255 (black)

Peak at the data

- **Grab an instance's feature vector and reshape it to a 28×28 array**

```
import matplotlib as mpl
import matplotlib.pyplot as plt
some_digit = X[0] # Change the zero and get another digit
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = mpl.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```

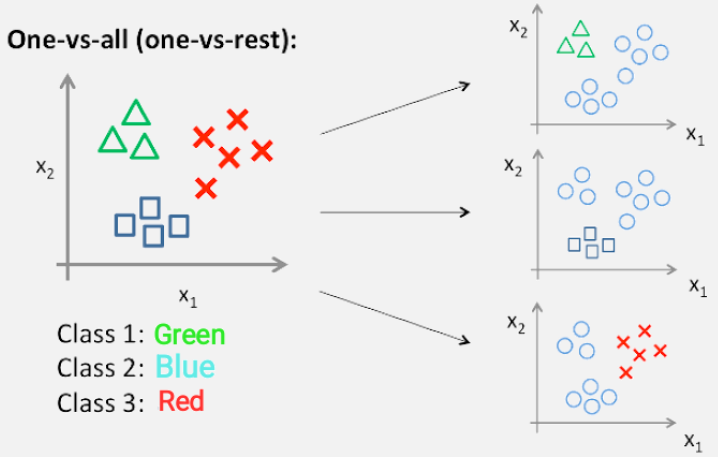
Training a Binary Classifier

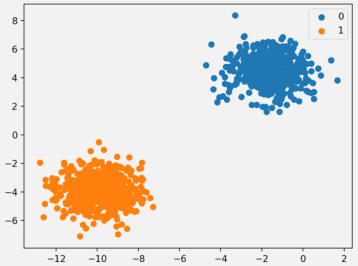
Binary classifier: ML algorithm used to classify input data into one of two categories or classes.

- Labels: positive and negative, or 1 and 0.
- Binary classifier analyzes the features of the input data and predicts which class the data belongs to.
- It is called "binary" because it makes a decision between only two possible outcomes.

Common examples:

- Spam email detection (spam or not spam)
- Disease diagnosis (positive or negative)
- Sentiment analysis (positive sentiment or negative sentiment).





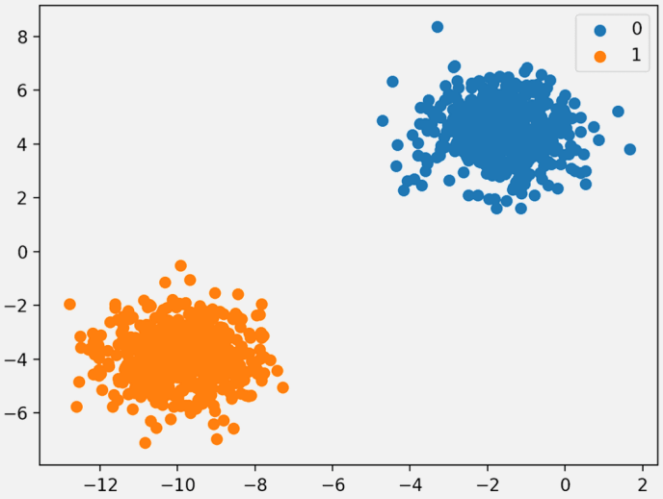
Training a Binary classifier

For simplification we will structure our problem into binary classification where we predict if a number is 5 or not:

```
y_train_5 = (y_train == 5) # True for all 5s, False for all other digits.
y_test_5 = (y_test == 5)
```

```
from sklearn.linear_model import LogisticRegression
log_clf = LogisticRegression(random_state=42)
log_clf.fit(X_train, y_train_5)
predicted_label  = log_clf.predict([some_digit])
print(predicted_label)
```

Classifier used is Logistic Regression Model



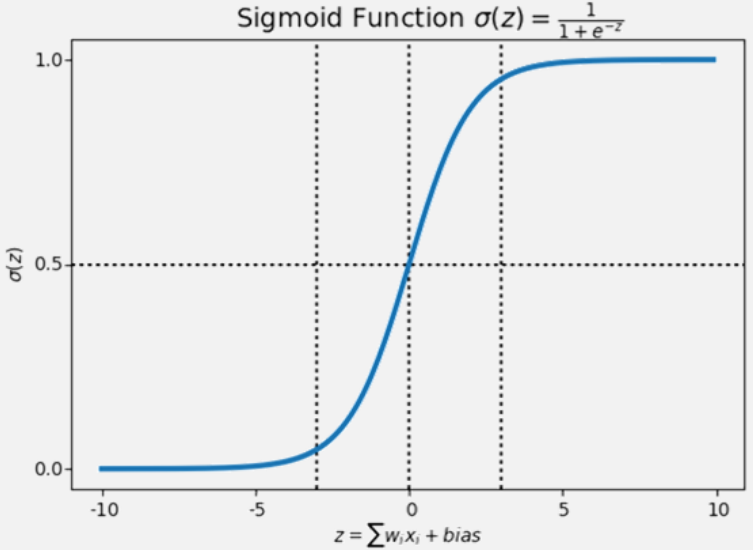
Follow lab:
T5B3ML101N02-0424-MNIST_Binary_Classifier
To run the code yourself and compare models

Logistic Regression

Estimates the probability of belonging to class for an instance(0-1)

Outputs the logistic of the *linear regression model*.

$$\sigma(t) = \frac{1}{1 + \exp(-t)} \qquad \hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma\left(\mathbf{x}^T \boldsymbol{\theta}\right)$$



If the probability > threshold then 1

Default threshold is 0.5

$\hat{y}$ = 1 if $\hat{p} \geq 0.5$ otherwise 0

Cost function:

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

Logistic cost function partial derivatives:

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

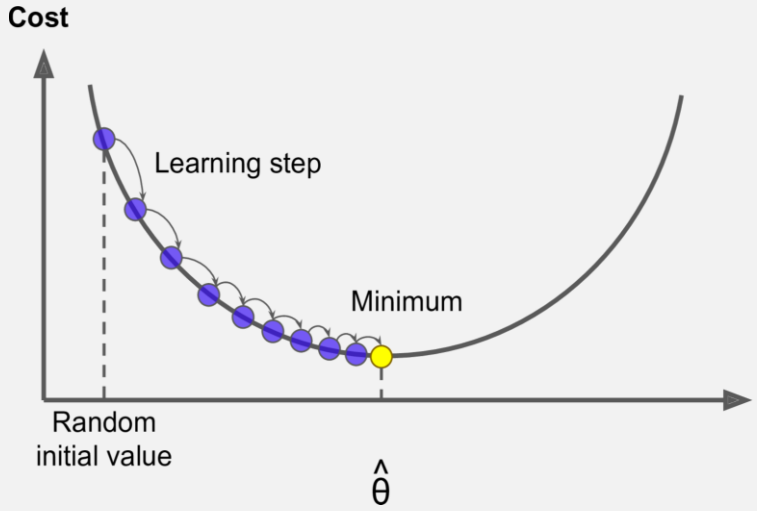
No known closed-form equation to compute best **θ.**

The cost function is convex, so Gradient Descent finds the global minimum (alpha not too large and train long enough).

Gradient Descent Refresher

Gradient Descent algorithm finds optimal solutions to a wide range of problems. It tweak parameters iteratively to minimize a cost function J(**θ**).

Start by random values for **θ** (*random initialization*), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function until it *converges* to a minimum



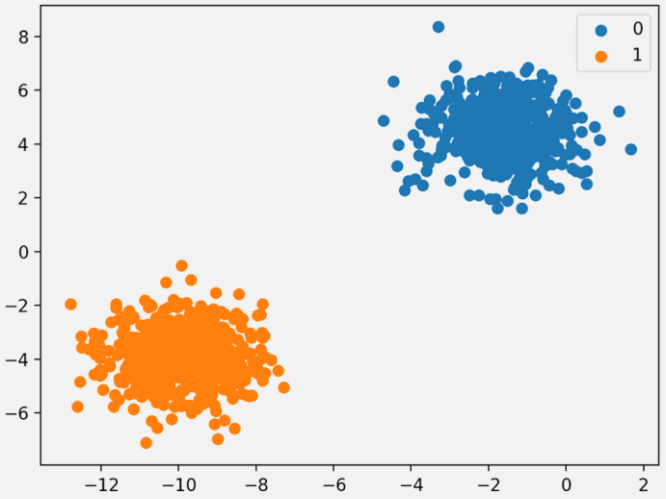
Training an SVM Binary classifier

We will now try another more advanced algorithm called Support vector Machine(SVM) to compare its performance on the same task:

```
y_train_5 = (y_train == 5) # True for all 5s, False for all other digits.
y_test_5 = (y_test == 5)
```

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
predicted_label  = sgd_clf.predict([some_digit])
print(predicted_label) >> array([False])
```

Classifier used is SVM Model



Follow lab:
T5B3ML101N02-0424-MNIST_Binary_Classifier
To run the code yourself and compare models

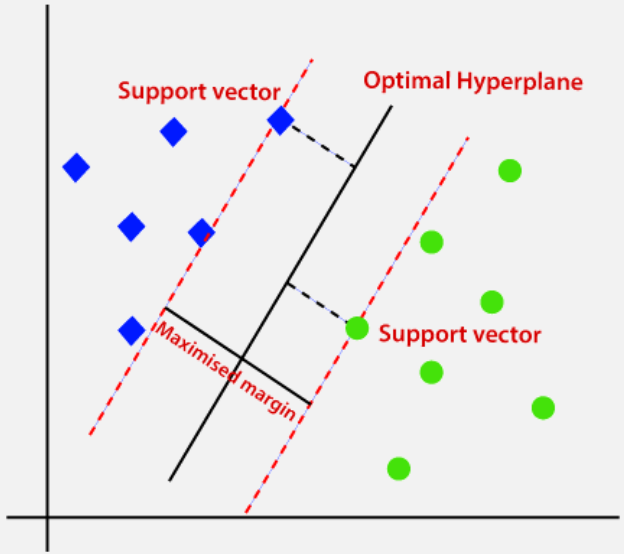
Support Vector Machine

SVM (Support Vector Machine) is a type of supervised learning algorithm used for both classification and regression tasks, though it is mostly used in classification problems.

The main idea behind an SVM classifier is to find the best boundary (or hyperplane) that separates classes in the feature space.

Core Components:

- **Hyperplane:** A decision boundary separating classes, optimized to be as far from the nearest data points of each class as possible.
- **Margin:** Distance between the hyperplane and the nearest data points. SVM aims to maximize this margin to increase model robustness.
- **Support Vectors:** Data points closest to the hyperplane, crucial in defining the margin.
- **Kernel Trick:** A method for handling non-linear data by projecting it into higher dimensions where it is linearly separable.



Using Cross-Validation to measure Accuracy

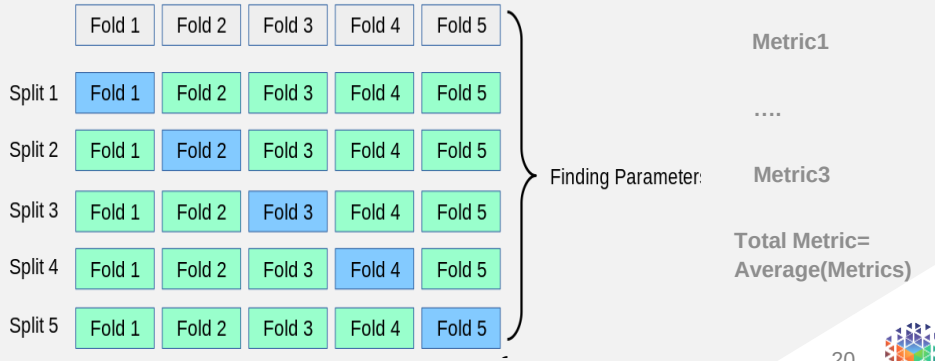
There are two main ways to split the data to ensure the model works on unseen data:

Train Test Split where you exclude part of the data from the training to test on

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Split the data into buckets and iterate on them to ensure Robustness but it takes more computation

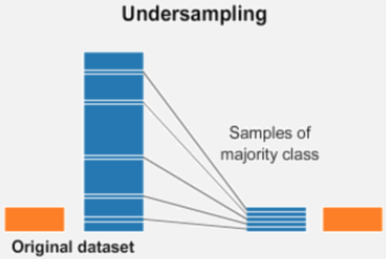
```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.96355, 0.93795, 0.95615])
```

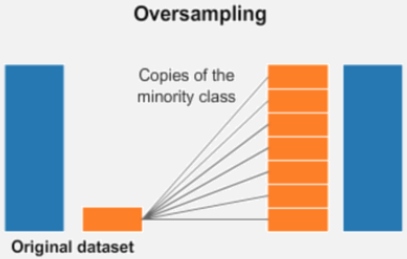


Is the data balanced?

Skewed datasets (Imbalanced Dataset): some classes are much more frequent than others.
Accuracy is not always the preferred performance measure

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
 def fit(self, X, y=None):
                pass
 def predict(self, X):
                return np.zeros((len(X), 1), dtype=bool)
```






```
never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.91125, 0.90855, 0.90915])
```

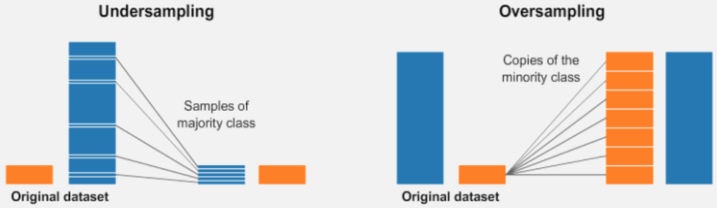
Skewed Data (Imbalanced Data)

Metric for imbalance Data:

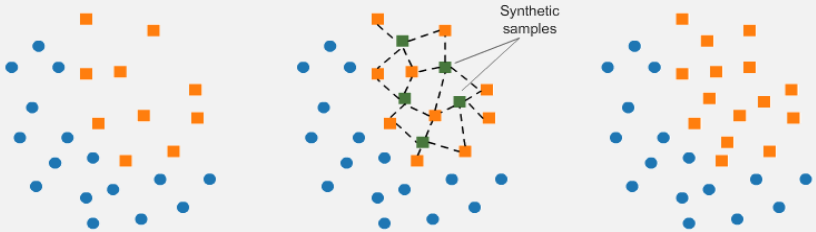
- F1 Score and Precision/Recall is preferred over accuracy in unbalanced data as accuracy is considered naïve in nature and misrepresents the error in this case.

Imbalance Handling:

Over-sampling and under-sampling



SMOTE (Synthetic Minority Oversampling): Its an intelligent way of oversampling that generates synthetic examples of the minority class to balance class distribution in a dataset.




```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote= smote.fit_resample(X_train, y_train)
```

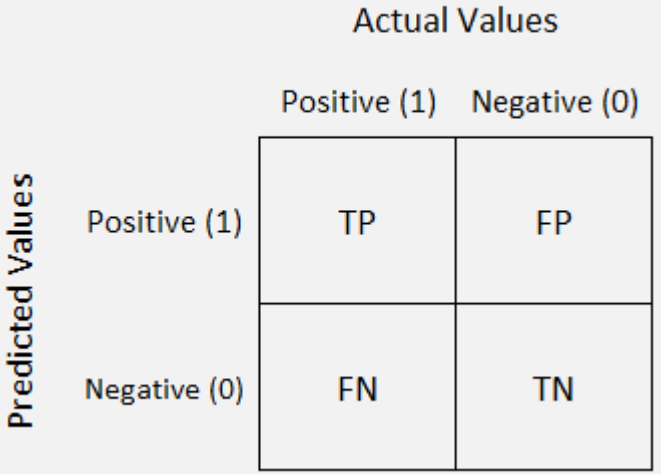
Confusion Matrix

- A table used to evaluate the performance of a classification model's predictions.
- Rows and columns, with each row representing the actual class labels and each column representing the predicted class labels.
- The main diagonal of the matrix represents the correct predictions, while off-diagonal elements represent errors made by the model.

The four cells of a confusion matrix typically represent:

- **True Positives (TP)**: The number of instances correctly predicted as positive.
- **False Positives (FP)**: The number of instances incorrectly predicted as positive.
- **True Negatives (TN)**: The number of instances correctly predicted as negative.
- **False Negatives (FN)**: The number of instances incorrectly predicted as negative.

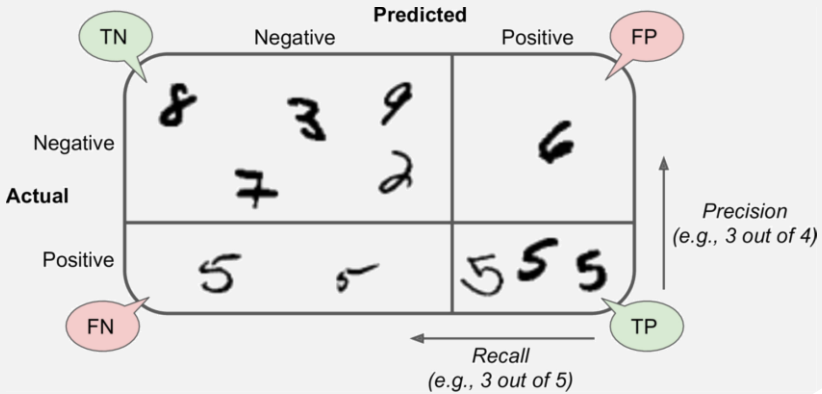
Used to calculate **Accuracy**, **precision**, **recall**, and **F1 score**.



Confusion Matrix

- **False**: count the number of times A was classified as B and VV.

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
  array([[53057, 1522],
  [ 1325, 4096]])
```



Performance Metrics

- **Recall:** It is the ability of a model to capture the positives in the data set, essentially capturing how many actual positives were not missed.

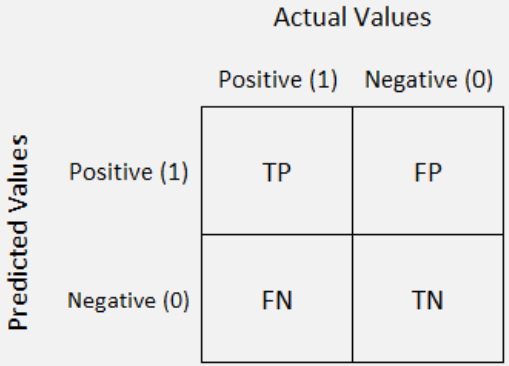
$$\text{recall} = \frac{TP}{TP + FN}$$

- **Precision:** It measures the accuracy of the model's positive predictions, indicating the count of correct predictions when the model predicts something as positive.

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

- **F1 Score:** The F1 Score combines precision and recall into a single metric, providing a single score(harmonic mean) that balances both the model's precision and its ability to recall all relevant samples.

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



Performance Metrics for Classification

- Accuracy : % of correct predictions (naive measure)
- Precision: % of correct positive from all +ve predicted
- Recall: % of correct predicted +ve samples

```
from sklearn.metrics import precision_score, recall_score
precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

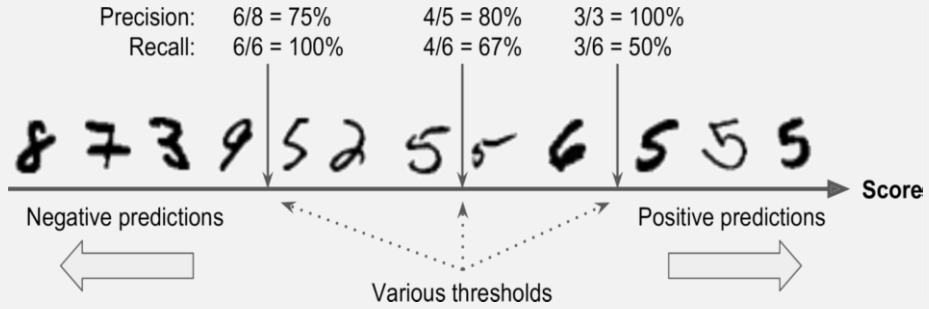
- F1 Score: Harmonic mean

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

Precision-Recall Trade-off

- SGDClassifier makes its classification decisions based on a decision function that compares a score to a threshold.
- Scikit-Learn does not let you set the threshold directly. (access to the decision scores. predict() ➔ decision_function()

```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
array([2412.53175101])
threshold = 0
y_some_digit_pred = (y_scores > threshold)
array([ True])
threshold = 8000
y_some_digit_pred = (y_scores > threshold)
array([False])
```



Improving precision often comes at the expense of recall, meaning that as a model becomes more accurate in its positive predictions, it may also miss more actual positives, demonstrating the inherent tradeoff between these two metrics.

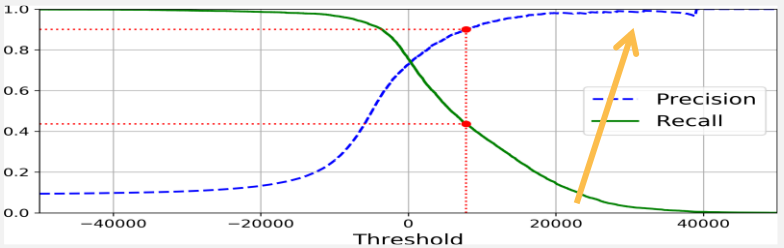
What is the best threshold value?

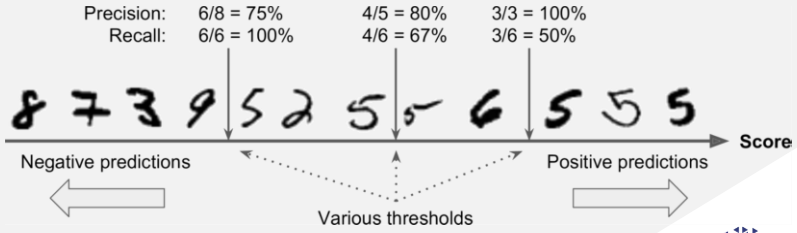
- Get all instances scores (training set) using the cross_val_predict() function to return decision scores and not predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
method="decision_function")
```

- Draw the precision vs recall for all possible thresholds

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
        plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
        plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
        [...] # highlight the threshold, add the legend, axis label and grid
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



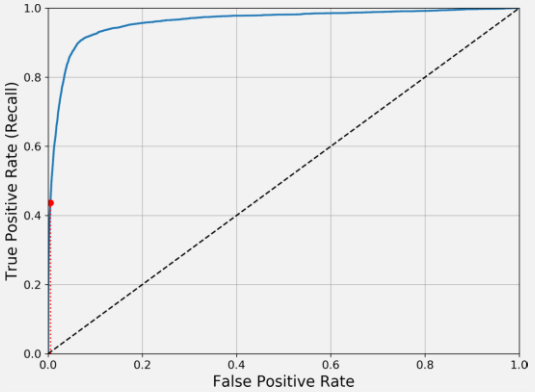


High precision classifier is not very useful if its recall is too low.
High precision at what recall?

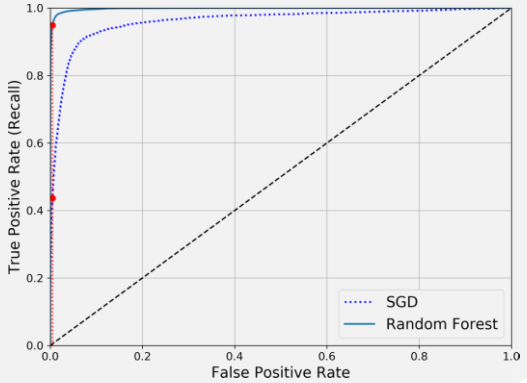
ROC Curve

- Receiver Operating Characteristic (ROC) curve: used with binary classifiers.
- Similar to the precision/recall curve, instead it is plotting TPR (recall) vs FPR
- FPR: ratio of negative instances that are incorrectly classified as positive (1-TNR(specificty))
- ROC curve plots sensitivity (recall) versus 1 – specificity.

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
def plot_roc_curve(fpr, tpr, label=None):
                    plt.plot(fpr, tpr, linewidth=2, label=label)
                    plt.plot([0, 1], [0, 1], 'k--') # dashed diagonal
                    [...] # Add axis labels and grid
plot_roc_curve(fpr, tpr)
plt.show()
```



Tradeoff: the higher the recall (TPR), the more false positives (FPR) the classifier produces.
Dotted line represents the ROC curve of a purely random classifier
Good classifier stays as far away from that line as possible



ROC AUC or PR Curve?

- **Rule of thumb:** use PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives, and the ROC curve otherwise.
- **For example**: looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s).
- **In contrast,** the PR curve makes it clear that the classifier has room for improvement (the curve could be closer to the top left corner).

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
method="predict_proba")
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()
roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

Precision: 99% and Recall 86.6% (Not too bad)

All Metrics

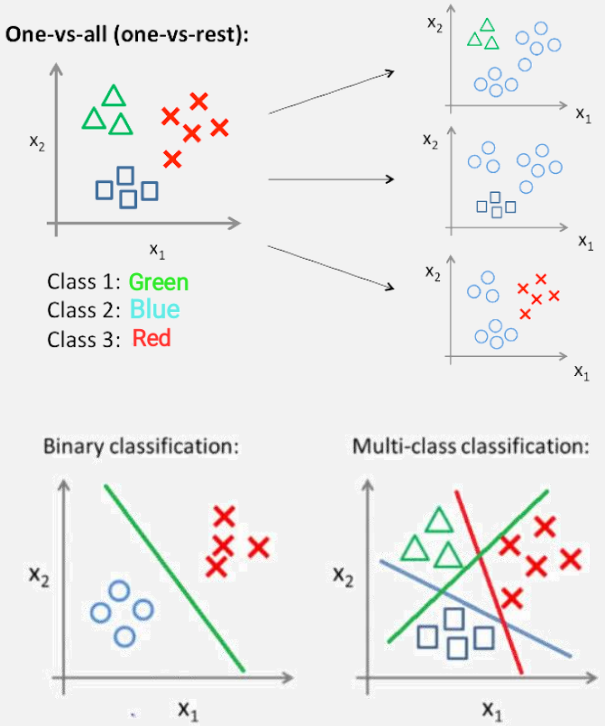
		Condition (as determined by "Gold standard")			
	Total population	Condition positive	Condition negative	Prevalence = Σ Condition positive / Σ Total population	
Test outcome	Test outcome positive	**True positive**	**False positive** (Type I error)	Positive predictive value (PPV, Precision) = Σ True positive / Σ Test outcome positive	False discovery rate (FDR) = Σ False positive / Σ Test outcome positive
	Test outcome negative	**False negative** (Type II error)	**True negative**	False omission rate (FOR) = Σ False negative / Σ Test outcome negative	Negative predictive value (NPV) = Σ True negative / Σ Test outcome negative
	Positive likelihood ratio (**LR+**) = TPR/FPR	True positive rate (TPR, Sensitivity, Recall) = Σ True positive / Σ Condition positive	False positive rate (FPR, Fall-out) = Σ False positive / Σ Condition negative	Accuracy (ACC) = Σ True positive + Σ True negative / Σ Total population	
	Negative likelihood ratio (**LR−**) = FNR/TNR	False negative rate (FNR) = Σ False negative / Σ Condition positive	True negative rate (TNR, Specificity, SPC) = Σ True negative / Σ Condition negative		
	Diagnostic odds ratio (**DOR**) = LR+/LR−				

Multiclass Classification

- **Binary Classifier** (Support Vector Machine classifiers or or Logistic regression)
- **Multiclass classifier (Multinomial)**: distinguishes between more than two classes. Ex: Random Forest classifiers or naive Bayes)
- **One vs All (rest) OvR**
- **One vs One (OvO)**

Follow lab:
T5B3ML101N03-0424-MNIST_Multiclass_Classification

To run the code yourself and compare models



Multiclass Classification example

Let's try SVM classifier

```
from sklearn.svm import SVC
svm_clf = SVC()
svm_clf.fit (_train, y_train)  #y_train not y_train_5
svm._clf.predict([some_digit])
array([5], dtype=uint8)
```

Under the hood Scikit-Learn uses OvO (45 binary classifiers) classifiers)

```
some_digit_scores =
svm_clf.decision_function([some_digit])
#10 scores per instance
some_digit_scores
array([[2.9249,7.0230, 3.9364, 0.9011,5.9694, 9.5,
1.9071, 8.0275, -0.1320, 4.9421]])
np.argmax(some_digit_scores)
5
sgd_clf.classes_
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
sgd_clf.classes_[5]
5
```

Multiclass Classification example

```
from sklearn.multiclass import OneVsOneClassifier
ovo_clf = OneVsOneClassifier(SVC())
ovo_clf.fit(X_train, y_train)
ovo_clf.predict([some_digit])
array([5], dtype=uint8)
len(ovo_clf.estimators_)
45
```

```
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC())
ovr_clf.fit(X_train, y_train)
ovr_clf.predict([some_digit])
array([5], dtype=uint8)
len(ovr_clf.estimators_)
10
```


Multiclass Classifier

Training a SGDClassifier to predict multiclass:

```
sgd_clf.fit(X_train, y_train)
sgd _clf.predict([some_digit])
array([5], dtype=uint8)

sgd.decision_function([some_digit])
array([[-15955.22, -38080.96, -13326.66, 573.52, -17680.68, 2412.53, -25526.86,-12290.15, -7946.05,-10631.25]])

forest_clf.predict_proba([some_digit])
array([[0. , 0. , 0.01, 0.08, 0. , 0.9 , 0. , 0. , 0. , 0.01]])
```

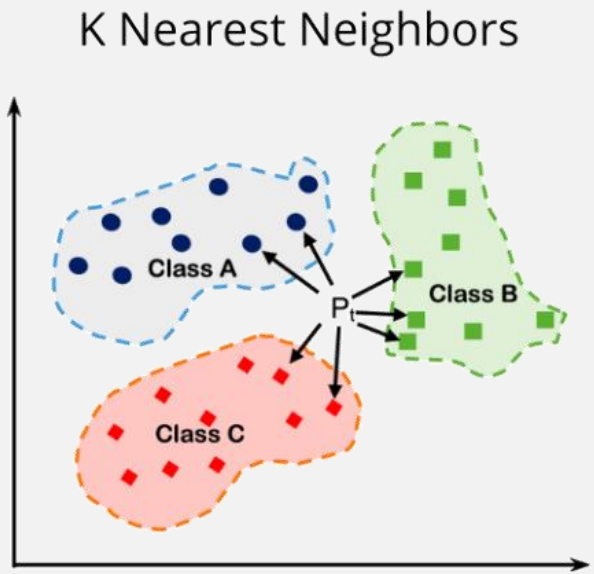
```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8489802 , 0.87129356, 0.86988048])

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([0.89707059, 0.8960948 , 0.90693604])
```

KNN Classifier

K-Nearest Neighbors (KNN) is a straightforward machine learning algorithm used for classification and regression. It predicts the label of a new point by examining the 'K' closest labeled points, using majority vote or average for the prediction.

Advantages of KNN:

- **Flexible to Data:** Assumes nothing about data distribution.
- **Simple:** Easy to understand and implement.
- **Versatile:** Effective for classification and regression.
- **Handles Multi-class:** Directly predicts multiple classes.
- **Dynamic Adaptation:** Adjusts to changes in data without needing a predefined model.



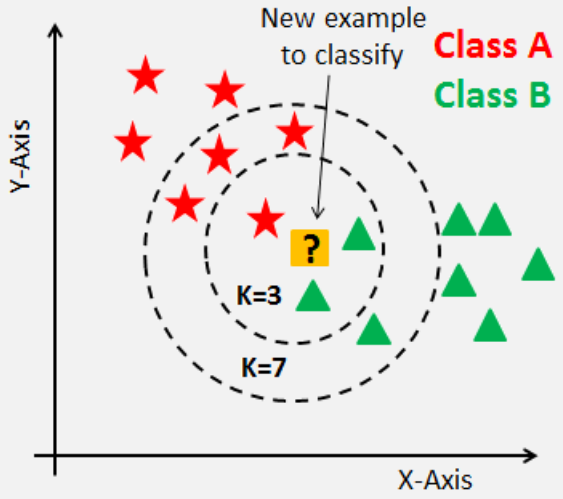
Follow lab: T5B3ML101N04-0424-KNN_Classifier
To run the code yourself and compare models

KNN Multilabel Classification

Multilabel classifier outputs multiple classes for each instance.

Let's use K-Nearest Neighbor (KNN) for predicting 2 labels:

```
from sklearn.neighbors import KNeighborsClassifier
y_train_large = (y_train >= 7) # large or not
y_train_odd = (y_train % 2 == 1) # odd or not
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
knn_clf.predict([some_digit])
array([[False, True]])
y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel,
cv=3)

f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```



Follow lab: T5B3ML101N04-0424-KNN_Classifier
To run the code yourself and compare models

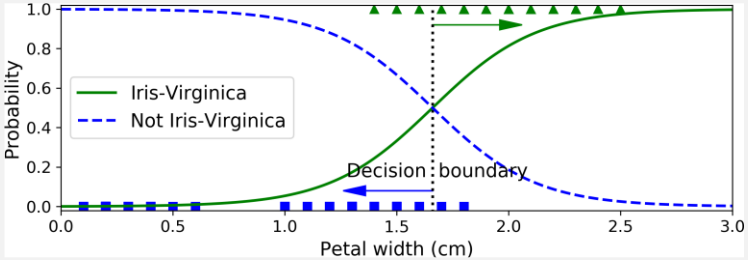
Logistic Regression Decision Boundaries

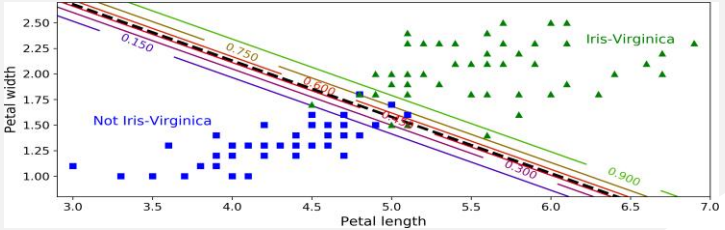
The data used this time is the IRIS dataset it is consists of 150 rows of data representing three species of iris flowers Setosa, Versicolor, and Virginica each described by four features: sepal length, sepal width, petal length, and petal width.

```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names',
'filename']
X = iris["data"][:, 3:] # petal width
y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Follow lab: T5B3ML101N05-0424-Logistic_Regression

To run the code yourself and compare models





Softmax Regression (Multinomial Logistic Regression)

Generalization of logistic regression to support multiple classes (no need to multiple binary classifiers)

1. for instance x, compute score $s_k(x)$ for each class k $s_k(x) = x^T\theta^k$
2. Apply softmax function to estimate each class probability (normalized exponential)

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

3. Pick the class with the highest probability. $\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\left(\boldsymbol{\theta}^{(k)}\right)^T \mathbf{x}\right)$

Objective: Train a model that estimates high prob for the target class and low prob for others

Cost function: Cross Entropy

Each class has a dedicated parameter vector $\theta^{(k)}$. $J(\boldsymbol{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$

Predict only one class at a time (not multioutput, mutually exclusive)

With K=2, cross entropy is only a Logistic Regression cost function

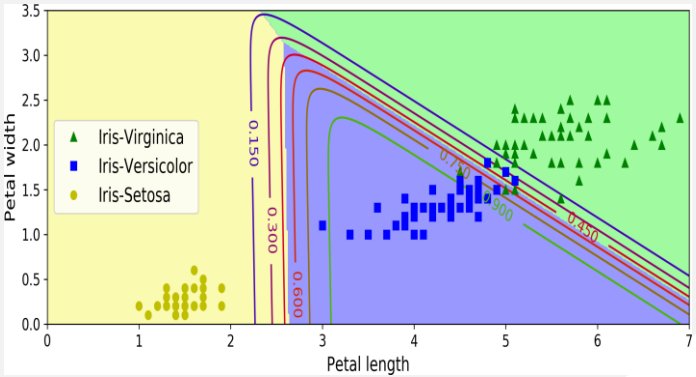
Follow lab: T5B3ML101N06-0424-Softmax_Regression

To run the code yourself and compare models

Softmax Regression in python

Predicting multiclass using Logistic Regression

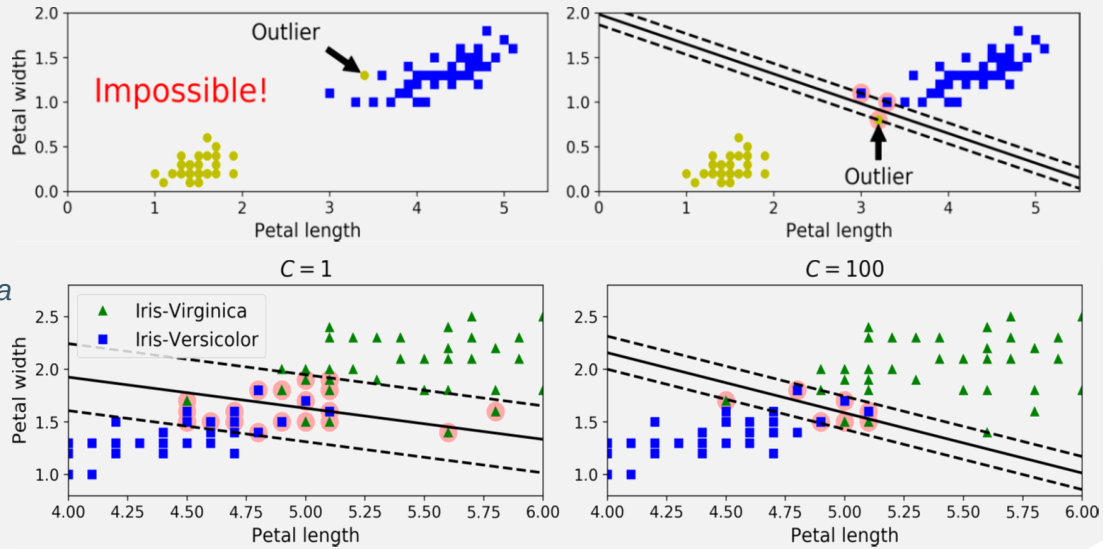
```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]
softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
softmax_reg.predict([[5, 2]])
array([2])
softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```



Hard Margin Classifier vs Soft Margin Classifier

Hard Margin Classifier: Its only goal is to strictly separates classes without misclassifications, ideal for clearly distinct datasets.
Soft Margin Classifier: It Allows some misclassifications to better handle complex datasets without clear separation in order to hopefully generalize better.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica
svm_clf = Pipeline([
("scaler", StandardScaler()),
("linear_svc", LinearSVC(C=1, loss="hinge")), ])
svm_clf.fit(X, y)
svm_clf.predict([[5.5, 1.7]])
array([1.])
```



Linear SVM vs non-linear SVM

Linear SVM

Linear SVM is used when the data can be separated by a straight line (or hyperplane in higher dimensions).

It aims to find the optimal hyperplane that maximizes the margin between two classes.

The key characteristics include:

- **Suitability:** Ideal for linearly separable datasets where classes can be separated without any errors using a straight line.
- **Computation:** Generally, more computationally efficient than its non-linear counterparts, especially suitable for high-dimensional data.
- :**Implementation:** Simpler to implement and understand. It involves solving a relatively simple optimization problem.

Follow lab: T5B3ML101N07-0424-Linear_SVM_Classification

To run the code yourself and compare models

Non-linear SVM

Non-linear SVM is used when the data cannot be separated by a straight line(the decision boundary between classes is non-linear).

It uses kernel functions (e.g., polynomial, RBF, etc.) to project data into a higher-dimensional space for linear separation.

The characteristics include:

- **Suitability:** Best for datasets where classes cannot be linearly separated in the original feature space.
- **Computation:** They require more processing due to the complexity of mapping data into higher dimensions.
- **Implementation:** Offers versatility with different kernels to fit various data patterns, though choosing and tuning these kernels demands more expertise.

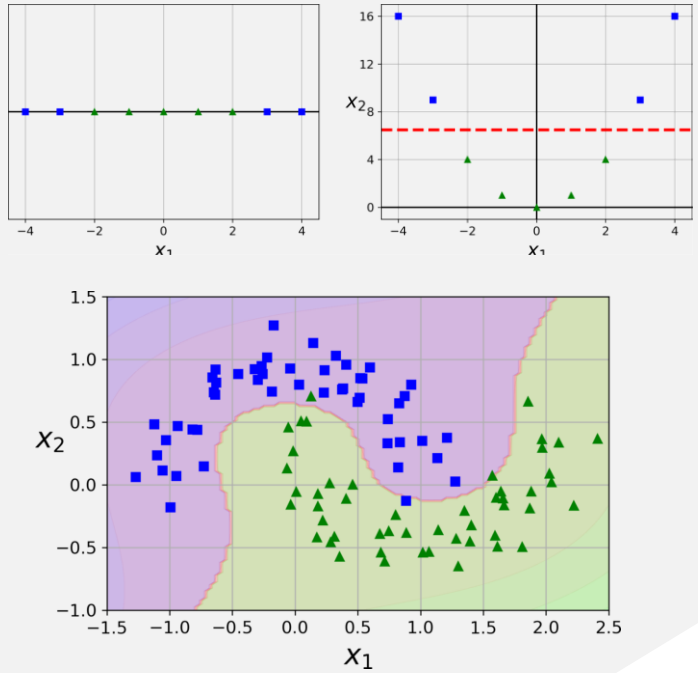
Follow lab: T5B3ML101N08-0424-NonLinear_SVM_Classification

To run the code yourself and compare models

Nonlinear SVM Classification

This code creates a pipeline for a Polynomial SVM Classifier and fits it to the data to show its effect on classifying non-linear problems.

```
from sklearn.datasets import make_moons
    from sklearn.pipeline import Pipeline
    from sklearn.preprocessing import PolynomialFeatures
    polynomial_svm_clf = Pipeline([
                ("poly_features", PolynomialFeatures(degree=3)),
                ("scaler", StandardScaler()),
                ("svm_clf", LinearSVC(C=10, loss="hinge"))])
    polynomial_svm_clf.fit(X, y)
```



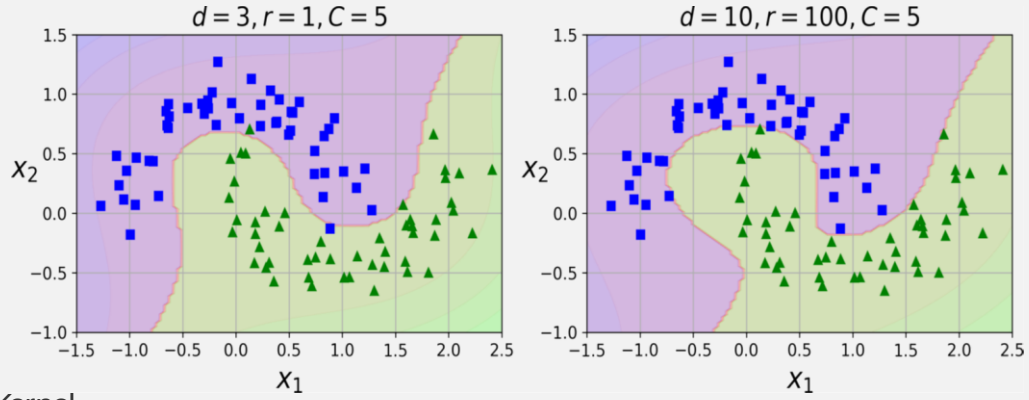
Follow lab: T5B3ML101N08-0424-NonLinear_SVM_Classification

To run the code yourself and compare models

Polynomial Kernel

Another way to introduce non linearity into the model is to have it create its own polynomial features using the poly kernel.

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



Follow lab: T5B3ML101N09-0424-Polynomial_Kernel

To run the code yourself and compare models

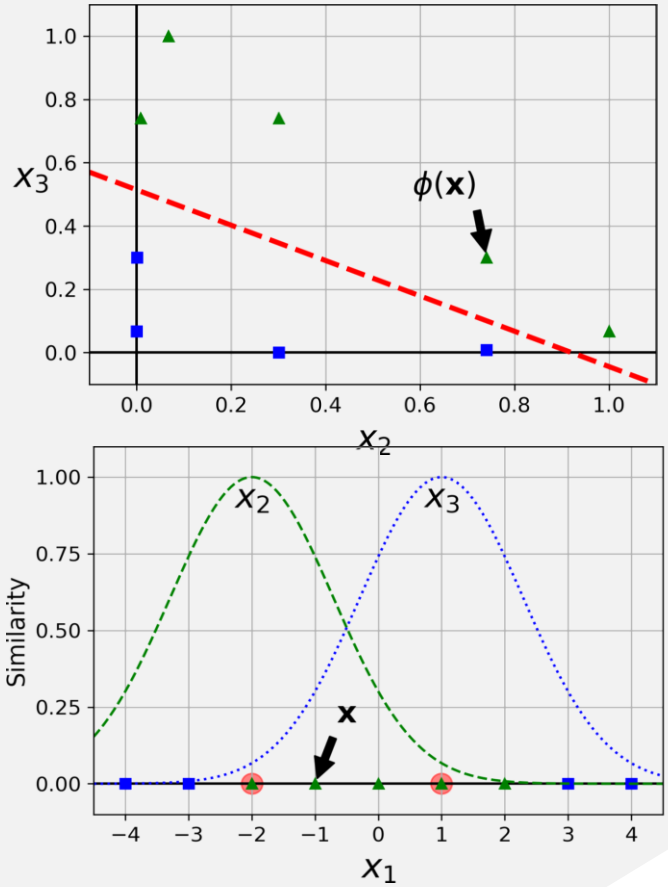
Similarity Features

Another useful non linear kernel that can be used is Gaussian RBF (Radial Basis Function) by transforming input features into higher dimensions to enable non-linear classification or regression

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

Ex: γ =0.3

```
rbf_kernel_svm_clf = Pipeline([
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
            ])
rbf_kernel_svm_clf.fit(X, y)
```



Follow lab: T5B3ML101N10-0424-Gaussian_RBF_Kernel

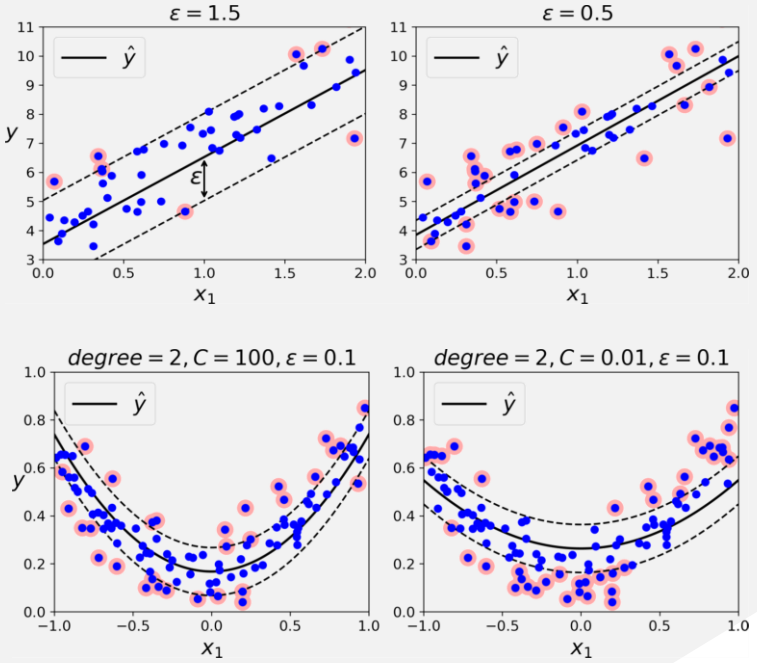
To run the code yourself and compare models

SVM Regressor

SVM algorithm is versatile (Linear and Nonlinear Regression)

- SVM Classifier: fits the largest possible street between two classes while limiting margin violations
- SVM Regressor: 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 : Street width, Scale the data

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



Note: SVM can be used as an outlier detector

Follow lab: T5B3ML101N11-0424-SVM_Regression

To run the code yourself and compare models

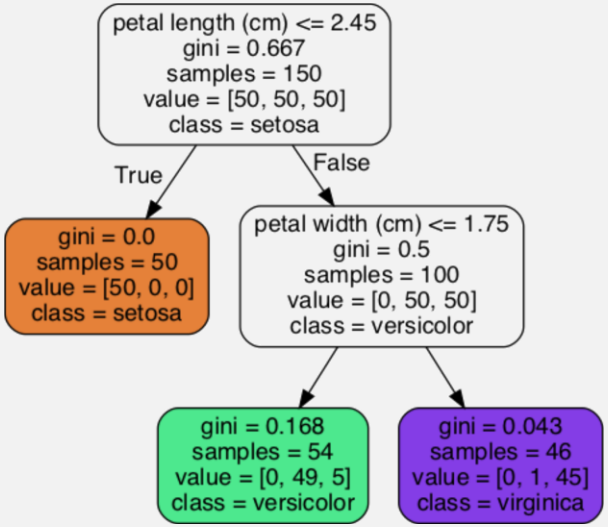
Decision Trees

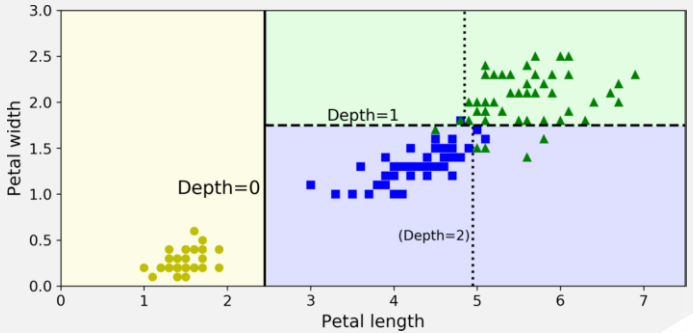
Decision Tree is a machine learning algorithm that builds a model in the form of a tree structure to make predictions by dividing a dataset into subsets based on feature values.

Versatile ML algorithm (classification and Regression)

No need to scale or center the data

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
tree_clf.predict_proba([[5, 1.5]])
array([[0. , 0.90740741, 0.09259259]])
tree_clf.predict([[5, 1.5]])
array([1])
```





Follow lab: T5B3ML101N12-0424-Decision_Trees

To run the code yourself and compare models

Decision Tree Drawbacks

Overfitting: They can overly adapt to training data, capturing noise instead of useful patterns.

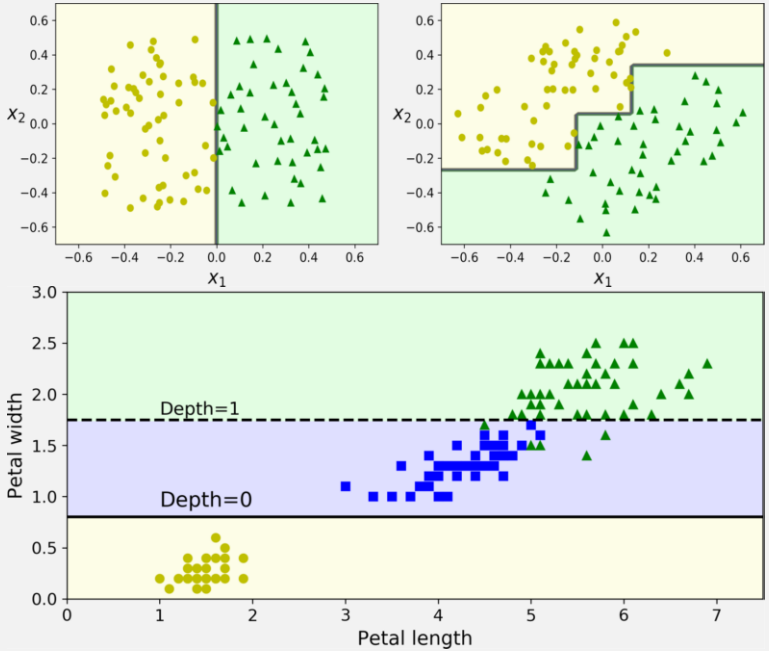
Axis-Aligned Boundaries: Their splits are aligned with data axes, which might not efficiently capture complex or diagonal boundaries.

Instability: Minor data changes can lead to significantly different trees, making them sensitive to data variations.

Binary Splits Limitation: They primarily use binary splits, which may not efficiently handle multi-category variables.

Inability to Capture Linear Relationships: Approximating linear relationships can lead to overly complex trees.

Bias with Imbalanced Data: They can be biased towards the majority class in imbalanced datasets, affecting performance for the minority class.

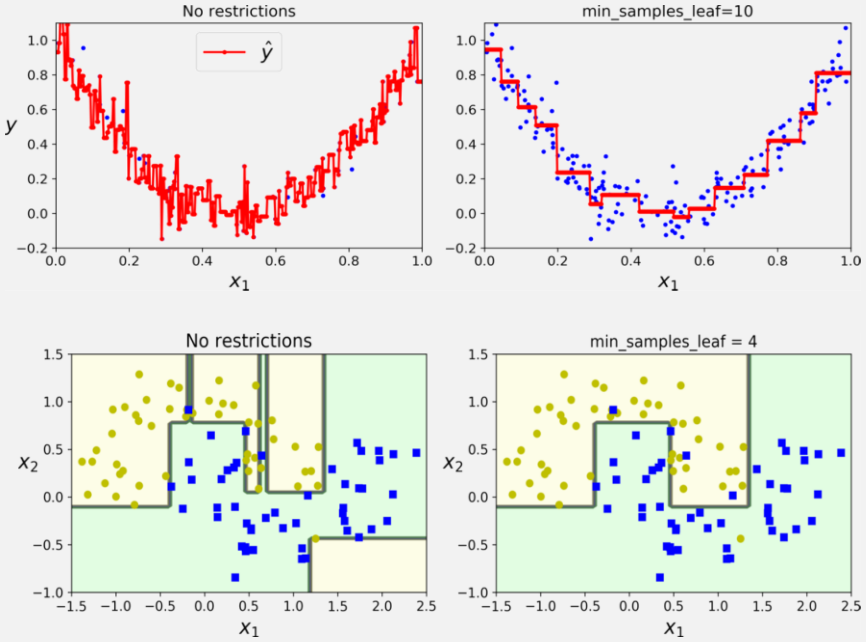


Decision Tree Regularization

Decision tree regularization involves techniques to prevent **Overfitting**, ensuring the model generalizes well to unseen data.

This can include:

- Limiting tree depth to prevent overly complex models.
- Pruning unnecessary branches to reduce overfitting.
- Setting a minimum number of samples required at a leaf node.
- Controlling the minimum number of samples required to split an internal node.
- Using cross-validation to find the optimal tree size.

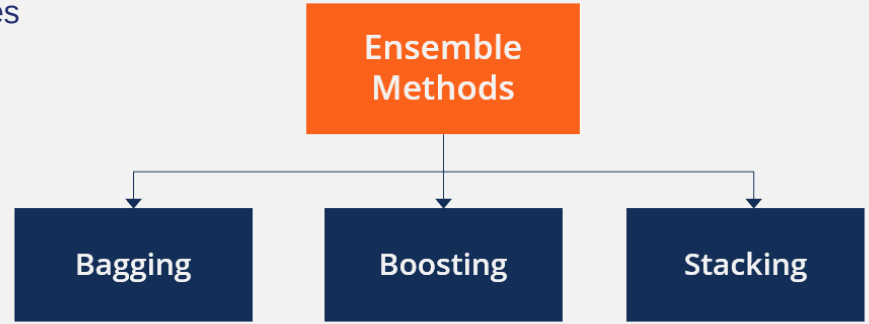


Ensemble Methods

Ensemble methods improve decision tree performance by combining multiple models to overcome individual weaknesses and solves many of the tree drawbacks.

Key benefits include:

- **Reducing Overfitting:** Techniques like Random Forests average multiple trees' predictions, smoothing decision boundaries and reducing sensitivity to noise.
- **Complex Boundaries:** They approximate complex decision boundaries more effectively than single trees.
- **Stability:** Combining models reduces the impact of data variations on the final prediction.
- **Handling Multi-Category Variables:** Aggregating insights from multiple trees improves handling of variables with many categories.
- **Imbalanced Data**: Some methods, like boosting, focus on harder-to-predict instances, improving minority class predictions.



Bagging and Pasting

Diverse Classifiers approach: train the same algorithm but with different random subsets of the training set

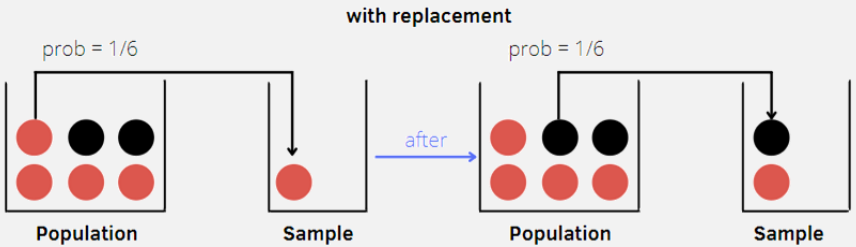
Bagging

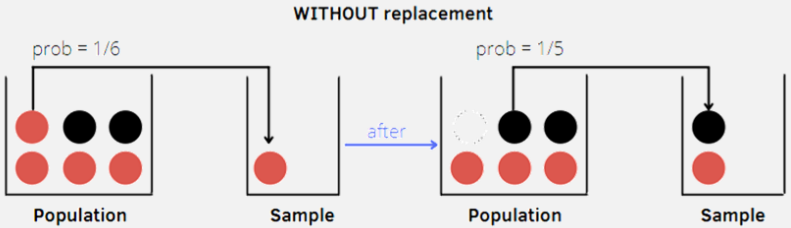
Involves training models on random subsets of the training data with replacement, allowing the same data points to be sampled multiple times in the same subset.

Short for Bootstrap Aggregating

Pasting

Similar to bagging but uses sampling without replacement, ensuring no data point appears more than once in the same subset.





Random Forest

- Random Forest: ensemble of Decision Trees (Bagging or Pasting, max_samples= training set size)
- BaggingClassifier ➔ DecisionTreeClassifier == RandomForestClassifier (highly optimized)

```
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
bag_clf = BaggingClassifier(DecisionTreeClassifier(splitter="random", max_leaf_nodes=16), n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

- Random Forest algorithm : extra randomness when growing trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features. (greater tree diversity, trades a higher bias for a lower variance, yielding an overall better model)

Follow lab: T5B3ML101N13-0424-Random_Forests_Binary, T5B3ML101N14-0424-Random_Forests

To run the code yourself and compare models

Feature Importance

- Weighted average: each node's weight is equal to the number of training samples that are associated with it.

```
from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rnd_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
    print(name, score)
```

```
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```


Voting Classifiers

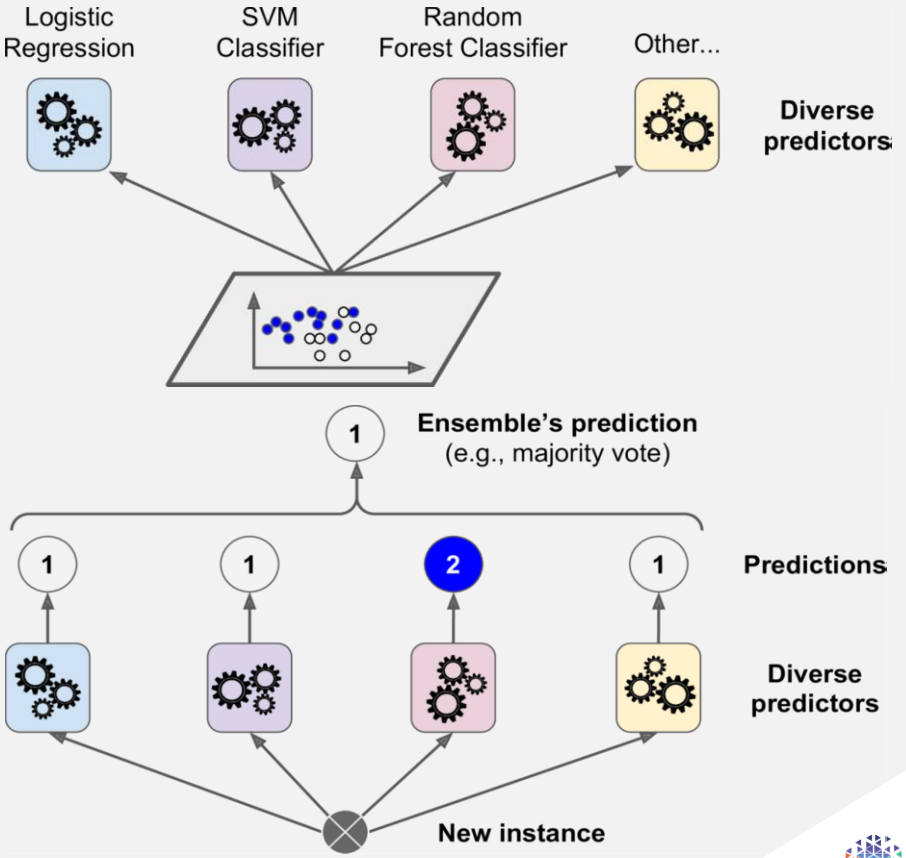
Combining multiple different models to get the best results

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()
voting_clf = VotingClassifier(
estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)], voting='hard')
voting_clf.fit(X_train, y_train)
from sklearn.metrics import accuracy_score
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

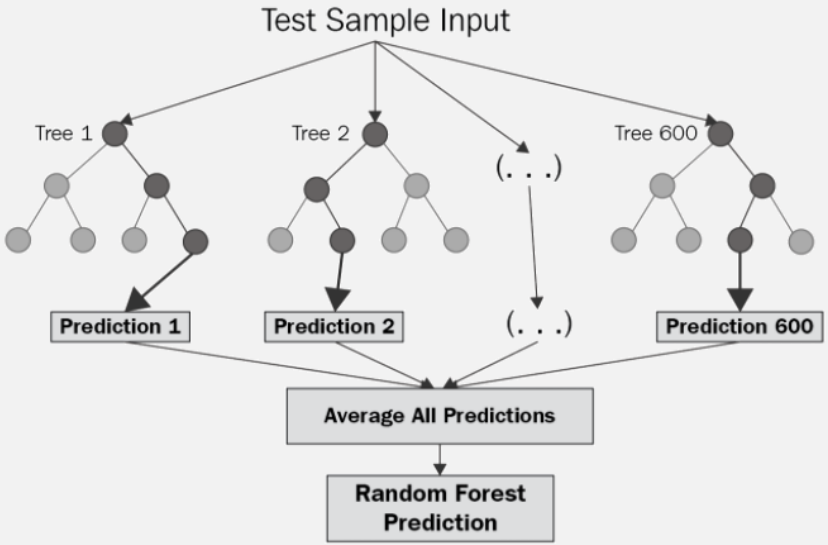
LogisticRegression 0.864, RandomForestClassifier 0.896, SVC 0.888
0.904

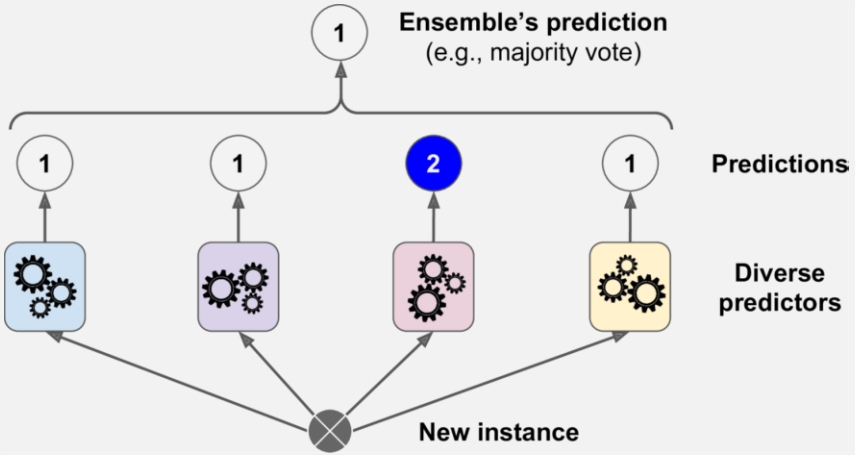
Follow lab: T5B3ML101N15-0424-Ensemble_Voting_Classifiers

To run the code yourself and compare models



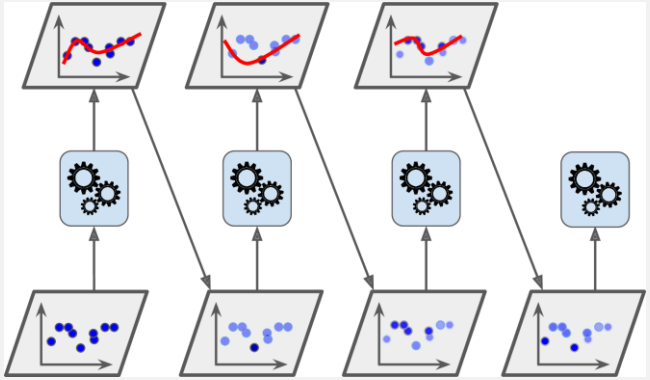
Random Forest VS Voting Classifiers





Boosting

- Ensemble method combining several weak learners into a strong learner
- Train predictors sequentially
- Each predictor is trying to correct its predecessor mistakes
- AdaBoost: predictors focusing on hard cases from previous predictor (underfitted)
- Base classifier ➔ increase misclassified instances (boosting) weights ➔another classifier…
- Predicted class receives the majority of weighted votes.
- Not parallelizable

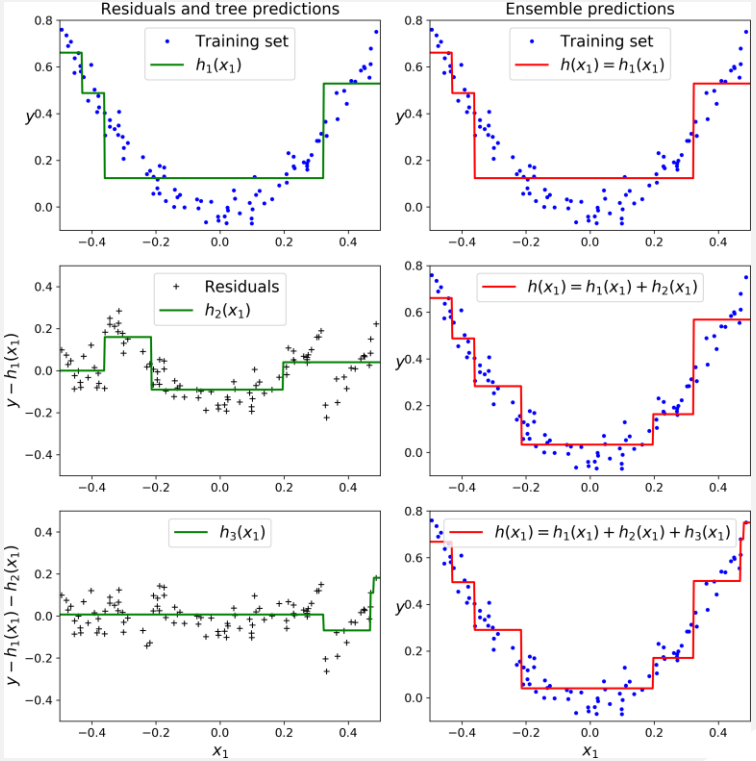


```
from sklearn.ensemble import AdaBoostClassifier # AdaBoostClassifer

ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
n_estimators=200, algorithm="SAMME.R", learning_rate=0.5)

ada_clf.fit(X_train, y_train)
```

Gradient Boosting

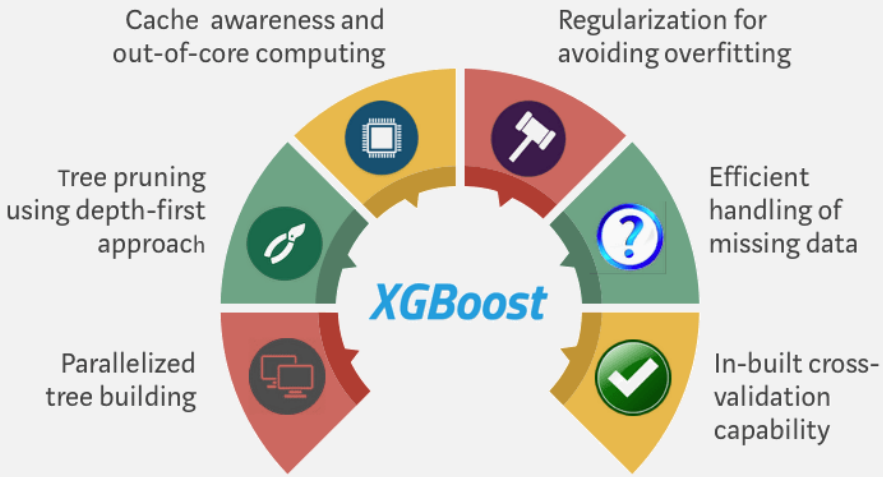
Similar to Adaboost but instead of tweaking weights, it fits the predictor on the residual errors made by previous predictor

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


y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
OR
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```



Gradient Boosting

```
from xgboost import XGBClassifier
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, KFold

iris = load_iris() x, y = iris.data, iris.target

xtrain, xtest, ytrain, ytest=train_test_split(x, y, test_size=0.15)
xgbc = XGBClassifier()

xgbc.fit(xtrain, ytrain)
```



Follow lab: T5B3ML101N16-0424-Ensemble_Gradient_Boosting

To run the code yourself and compare models

Stacking

Base-Level Models: Various models trained on the same data, their predictions become features for the next level.

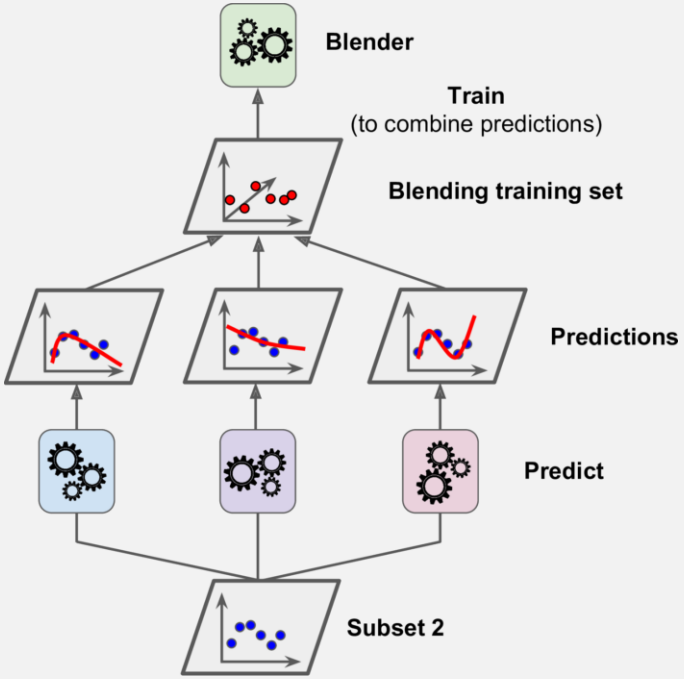
Meta-Model: Trains on the base models' predictions to make the final prediction.

Stacking potentially outperforming individual models and simpler ensembles. However, it's complex and demands careful tuning to prevent overfitting

.

Follow lab: T5B3ML101N16-0424-Ensemble_Stacking

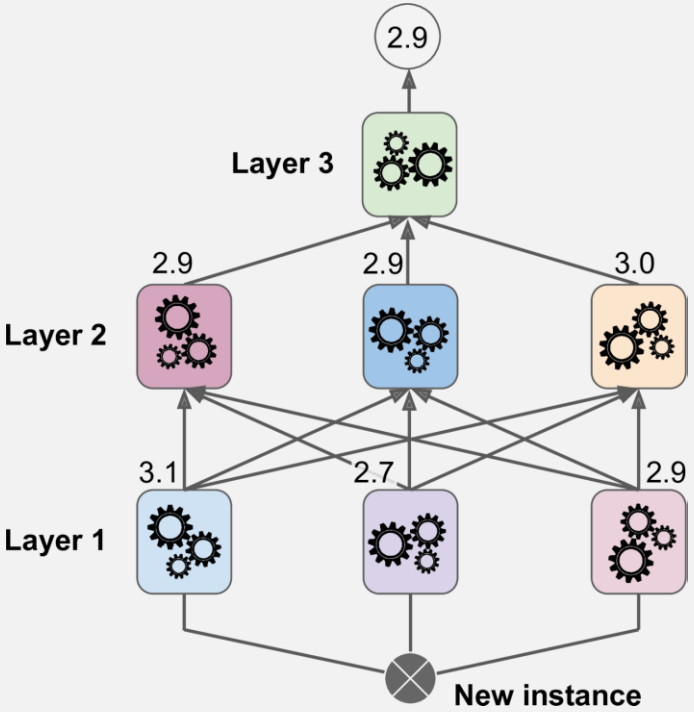
To run the code yourself and compare models



Prediction in a Multilayer stacking Ensemble

Multilayer Stacking Ensemble:

- Utilizes several layers of meta-models, each building on the previous layer's predictions.
- Enhances accuracy and performance by capturing complex patterns.
- Each successive layer refines predictions, aiming for improved precision.
- Requires careful architecture to prevent overfitting.
- Demands more computational resources and thoughtful tuning.



Error Analysis & Best Practices

Machine Learning Check list:

- Frame the Problem and Look at the Big Picture
- Get the Data
- Explore the Data
- Prepare the Data
- Build a baseline model for comparison
- Shortlist the promising model
- Fine-Tune the System (Hyper parameter tuning)
- Present Your solution

Error Analysis: one way to improve a model is to analyze the types of errors it makes and find reasons (not enough data for a certain class, model is not performing well to another class)

Thank You

SDAIA
الهيئة السعودية للبيانات والذكاء الاصطناعي
Saudi Data & AI Authority