Pending

* Overfitting in Decision Trees and prevention techniques
* Out-of-Bag (OOB) error: concept and calculation
* Choosing F1-score over Accuracy: when and why
* Importance of Precision in model evaluation
* Importance of Recall in model evaluation
* Common formats for saving ML models

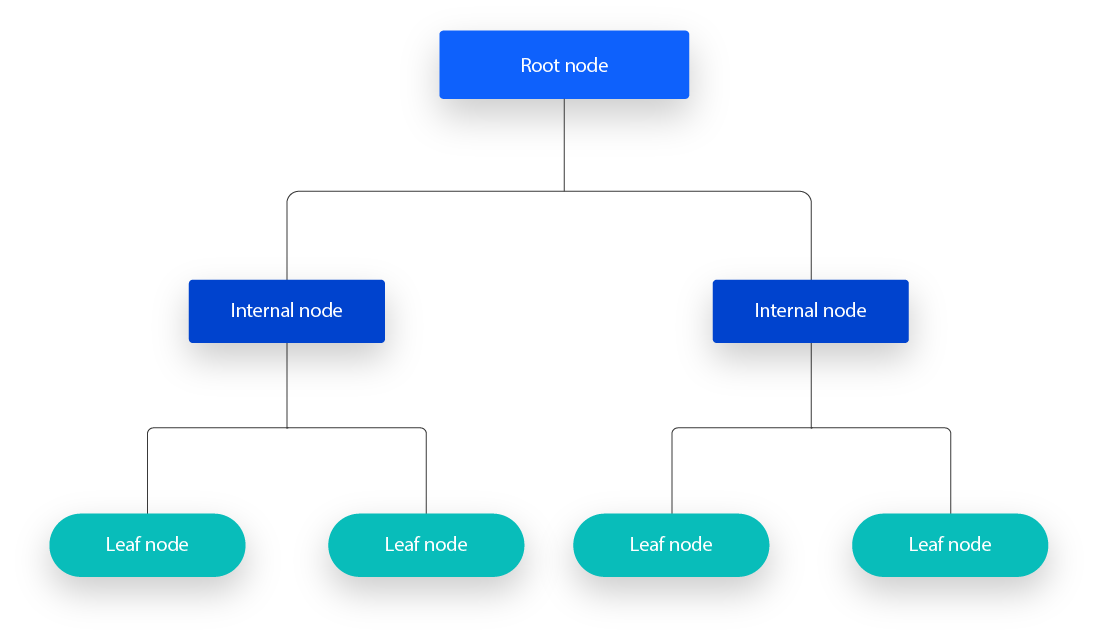
# Syllabus

1. Classification Algorithms (Continuation):
   1. Implement Decision Tree
   2. Implement Random Forest
   3. Implement Support Vector Machine
2. Evaluation Metrics for Classification
3. Learn how to save models.
4. Hyperparameter Tuning Techniques ( study any algorithms )
5. Learn and try out all algorithms and concepts mentioned above

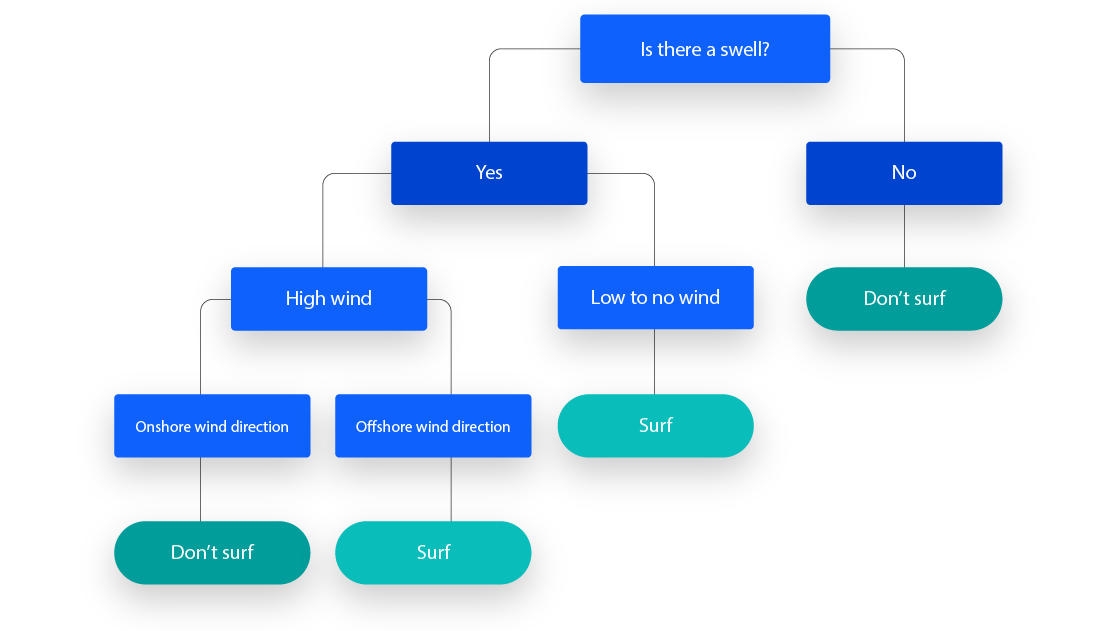
## **Decision Trees**

A decision tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. It has a hierarchical, tree structure, which consists of a root node, branches, internal nodes and leaf nodes.

As you can see from the diagram below, a decision tree starts with a root node, which does not have any incoming branches. The outgoing branches from the root node then feed into the internal nodes, also known as decision nodes. Based on the available features, both node types conduct evaluations to form homogenous subsets, which are denoted by leaf nodes, or terminal nodes. The leaf nodes represent all the possible outcomes within the dataset.



As an example, let’s imagine that you were trying to assess whether or not you should go surf, you may use the following decision rules to make a choice:



This type of flowchart structure also creates an easy to digest representation of decision-making, allowing different groups across an organization to better understand why a decision was made.

Decision tree learning employs a divide and conquer strategy by conducting a greedy search to identify the optimal split points within a tree. This process of splitting is then repeated in a top-down, recursive manner until all, or the majority of records have been classified under specific class labels.

Whether or not all data points are classified as homogenous sets is largely dependent on the complexity of the decision tree. Smaller trees are more easily able to attain pure leaf nodes—i.e. data points in a single class. However, as a tree grows in size, it becomes increasingly difficult to maintain this purity, and it usually results in too little data falling within a given subtree. When this occurs, it is known as data fragmentation, and it can often lead to [overfitting](https://www.ibm.com/think/topics/overfitting).

As a result, decision trees have preference for small trees, which is consistent with the principle of parsimony in Occam’s Razor; that is, “entities should not be multiplied beyond necessity.” Said differently, decision trees should add complexity only if necessary, as the simplest explanation is often the best. To reduce complexity and prevent overfitting, pruning is usually employed; this is a process, which removes branches that split on features with low importance. The model’s fit can then be evaluated through the process of cross-validation.

Another way that decision trees can maintain their accuracy is by forming an ensemble via a [random forest](https://www.ibm.com/think/topics/random-forest) algorithm; this classifier predicts more accurate results, particularly when the individual trees are uncorrelated with each other.

## **Types of decision trees**

Hunt’s algorithm, which was developed in the 1960s to model human learning in Psychology, forms the foundation of many popular decision tree algorithms, such as the following:

**- ID3:**Ross Quinlan is credited within the development of ID3, which is shorthand for “Iterative Dichotomiser 3.” This algorithm leverages entropy and information gain as metrics to evaluate candidate splits. Some of Quinlan’s research on this algorithm from 1986 can be found [here](https://hunch.net/~coms-4771/quinlan.pdf).

**- C4.5:**This algorithm is considered a later iteration of ID3, which was also developed by Quinlan. It can use information gain or gain ratios to evaluate split points within the decision trees.

**- CART:**The term, CART, is an abbreviation for “classification and regression trees” and was introduced by Leo Breiman. This algorithm typically utilizes Gini impurity to identify the ideal attribute to split on. Gini impurity measures how often a randomly chosen attribute is misclassified. When evaluating using Gini impurity, a lower value is more ideal.

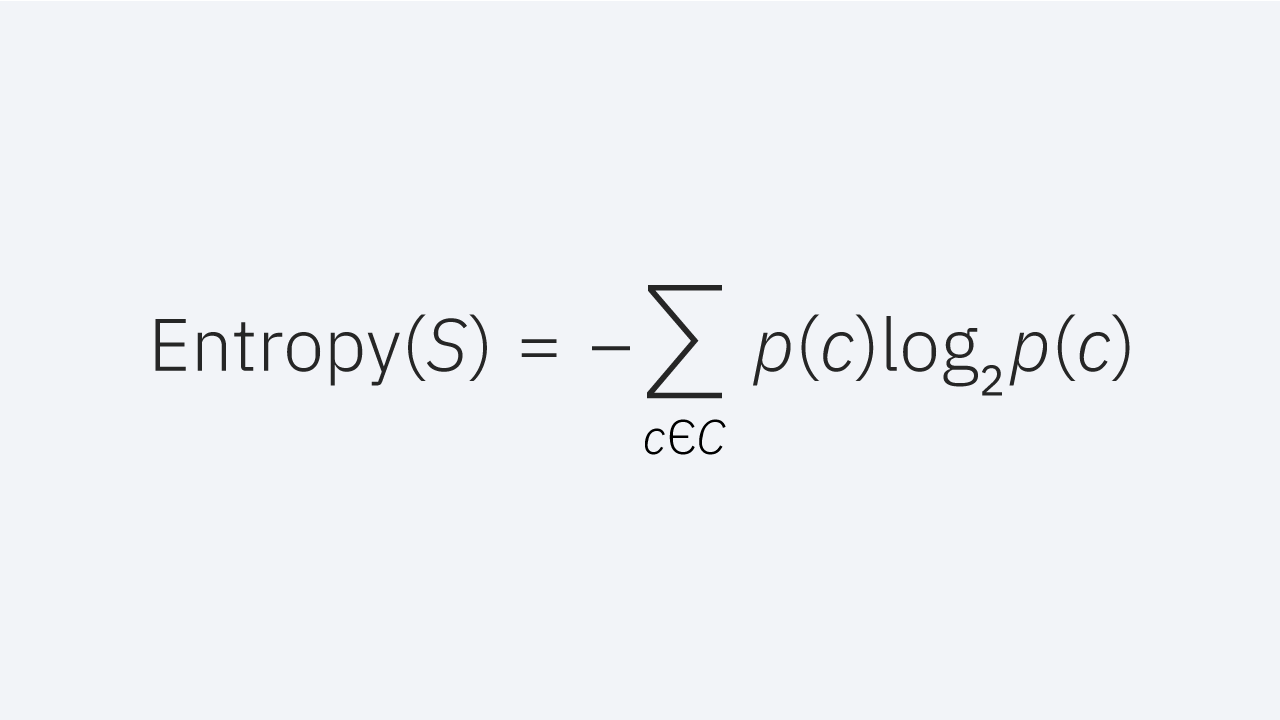
Bottom of Form

## **How to choose the best attribute at each node**

While there are multiple ways to select the best attribute at each node, two methods, information gain and Gini impurity, act as popular splitting criterion for decision tree models. They help to evaluate the quality of each test condition and how well it will be able to classify samples into a class.

### **Entropy and information gain**

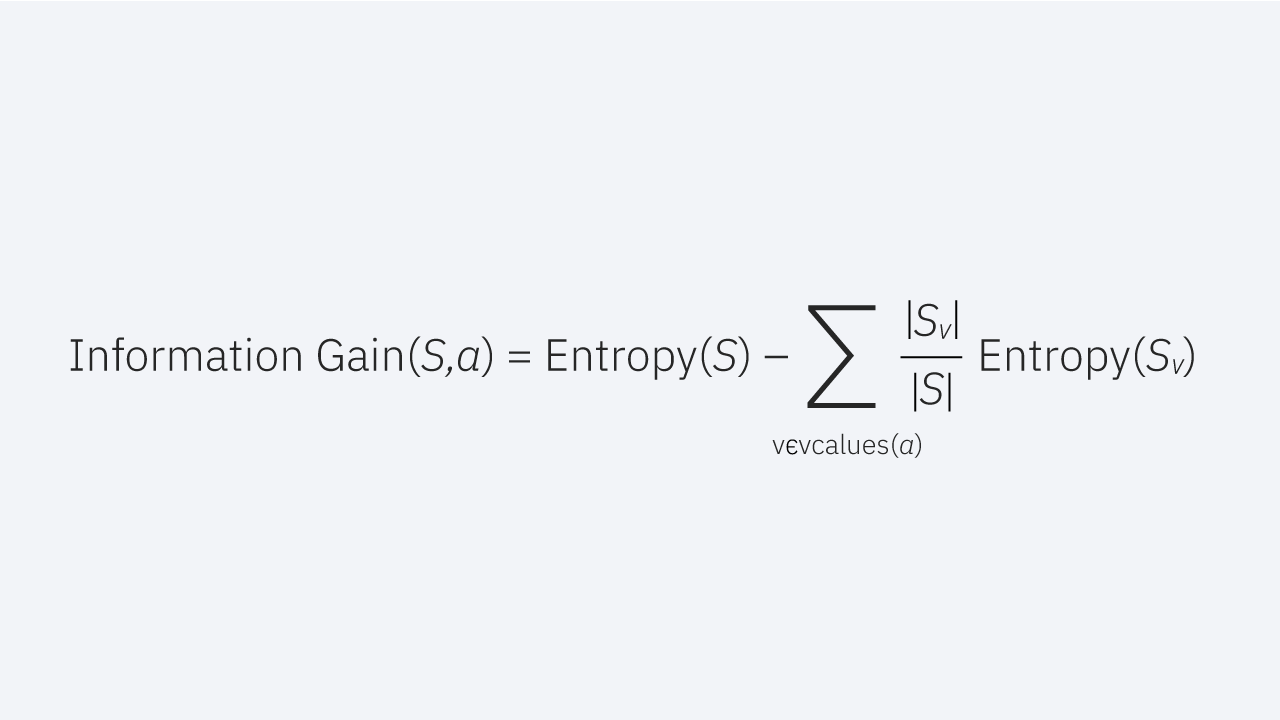
It’s difficult to explain information gain without first discussing entropy. Entropy is a concept that stems from information theory, which measures the impurity of the sample values. It is defined with by the following formula, where:



* S represents the data set that entropy is calculated
* c represents the classes in set, S
* p(c) represents the proportion of data points that belong to class c to the number of total data points in set, S

Entropy values can fall between 0 and 1. If all samples in data set, S, belong to one class, then entropy will equal zero. If half of the samples are classified as one class and the other half are in another class, entropy will be at its highest at 1. In order to select the best feature to split on and find the optimal decision tree, the attribute with the smallest amount of entropy should be used.

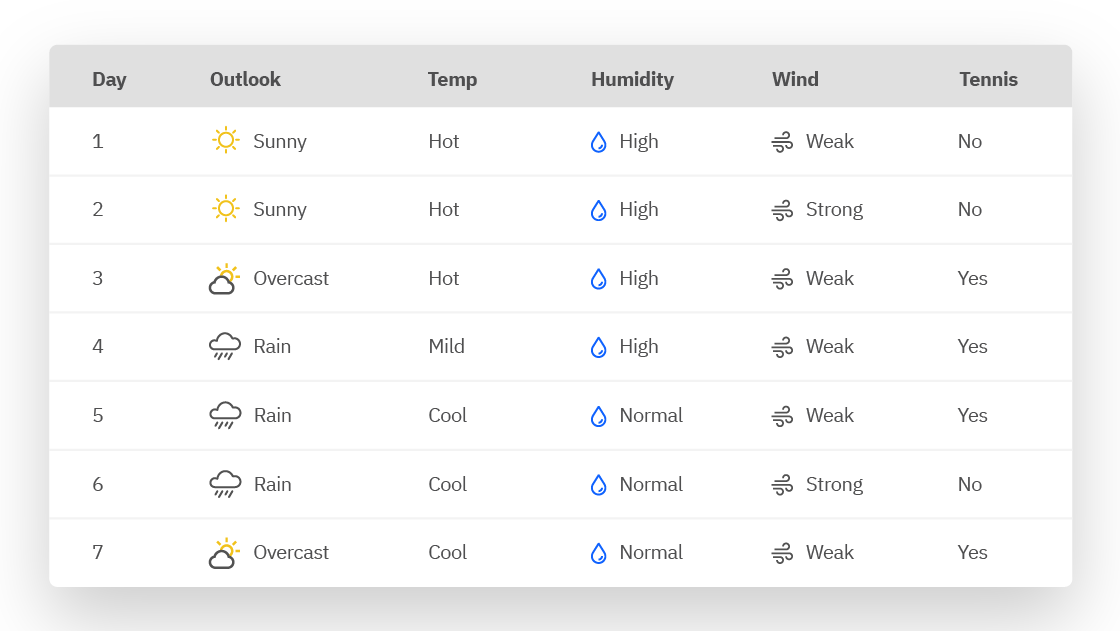
Information gain represents the difference in entropy before and after a split on a given attribute. The attribute with the highest information gain will produce the best split as it’s doing the best job at classifying the training data according to its target classification. Information gain is usually represented with the following formula,



where

* a represents a specific attribute or class label
* Entropy(S) is the entropy of dataset, S
* |Sv|/|S| represents the proportion of the values in Sv to the number of values in dataset, S.

Let’s walk through an example to solidify these concepts. Imagine that we have the following arbitrary dataset:



For this dataset, the entropy is 0.94. This can be calculated by finding the proportion of days where “Play Tennis” is “Yes”, which is 9/14, and the proportion of days where “Play Tennis” is “No”, which is 5/14. Then, these values can be plugged into the entropy formula above.

Entropy (Tennis) = -(9/14) log2(9/14) – (5/14) log2 (5/14) = 0.94

We can then compute the information gain for each of the attributes individually. For example, the information gain for the attribute, “Humidity” would be the following:

Gain (Tennis, Humidity) = (0.94)-(7/14)\*(0.985) – (7/14)\*(0.592) = 0.151

As a recap,

- 7/14 represents the proportion of values where humidity equals “high” to the total number of humidity values. In this case, the number of values where humidity equals “high” is the same as the number of values where humidity equals “normal”.

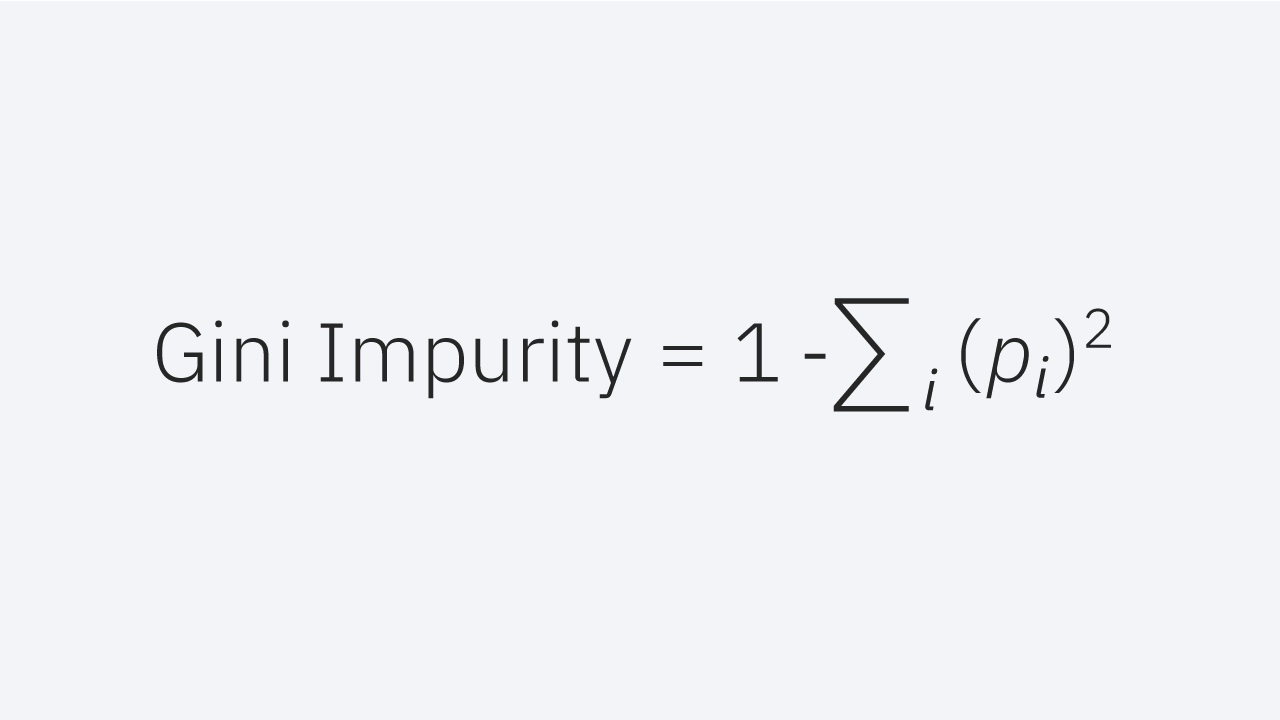
- 0.985 is the entropy when Humidity = “high”

- 0.59 is the entropy when Humidity = “normal”

Then, repeat the calculation for information gain for each attribute in the table above, and select the attribute with the highest information gain to be the first split point in the decision tree. In this case, outlook produces the highest information gain. From there, the process is repeated for each subtree.

### **Gini Impurity**

Gini impurity is the probability of incorrectly classifying random data point in the dataset if it were labeled based on the class distribution of the dataset. Similar to entropy, if set, S, is pure—i.e. belonging to one class) then, its impurity is zero. This is denoted by the following formula:



## **Advantages and disadvantages of decision trees**

While decision trees can be used in a variety of use cases, other algorithms typically outperform decision tree algorithms. That said, decision trees are particularly useful for [data mining](https://www.ibm.com/think/topics/data-mining) and knowledge discovery tasks. Let’s explore the key benefits and challenges of utilizing decision trees more below:

### **Advantages**

* **Easy to interpret:**The Boolean logic and visual representations of decision trees make them easier to understand and consume. The hierarchical nature of a decision tree also makes it easy to see which attributes are most important, which isn’t always clear with other algorithms, like [neural networks](https://www.ibm.com/think/topics/neural-networks).
* **Little to no data preparation required:**Decision trees have a number of characteristics, which make it more flexible than other classifiers. It can handle various data types—i.e. discrete or continuous values, and continuous values can be converted into categorical values through the use of thresholds. Additionally, it can also handle values with missing values, which can be problematic for other classifiers, like Naïve Bayes.
* **More flexible:** Decision trees can be leveraged for both classification and regression tasks, making it more flexible than some other algorithms. It’s also insensitive to underlying relationships between attributes; this means that if two variables are highly correlated, the algorithm will only choose one of the features to split on.

### **Disadvantages**

* **Prone to overfitting:** Complex decision trees tend to overfit and do not generalize well to new data. This scenario can be avoided through the processes of pre-pruning or post-pruning. Pre-pruning halts tree growth when there is insufficient data while post-pruning removes subtrees with inadequate data after tree construction.
* **High variance estimators:**Small variations within data can produce a very different decision tree. [Bagging](https://www.ibm.com/think/topics/bagging), or the averaging of estimates, can be a method of reducing variance of decision trees. However, this approach is limited as it can lead to highly correlated predictors.
* **More costly:**Given that decision trees take a greedy search approach during construction, they can be more expensive to train compared to other algorithms.

**What Is a Classification Tree?**

A classification tree (or decision tree classifier) is a type of decision tree used to predict categorical outcomes from a set of observations. Classification trees are created by recursively partitioning data based on measure of Gini impurity or information gain, with leaf nodes representing class labels and final possible outcomes.

[builtin desicion tree notes](https://builtin.com/data-science/classification-tree)

[IBM decisiontree](https://www.ibm.com/think/topics/decision-trees)

[desiciontree geekforgeeks](https://www.geeksforgeeks.org/machine-learning/decision-tree/)

## **Random Forest**

## **What is random forest?**

Random forest is a commonly-used machine learning algorithm, trademarked by Leo Breiman and Adele Cutler, that combines the output of multiple decision trees to reach a single result. Its ease of use and flexibility have fueled its adoption, as it handles both classification and regression problems.

### **Decision trees**

Since the random forest model is made up of multiple decision trees, it would be helpful to start by describing the decision tree algorithm briefly. Decision trees start with a basic question, such as, “Should I surf?” From there, you can ask a series of questions to determine an answer, such as, “Is it a long period swell?” or “Is the wind blowing offshore?”. These questions make up the decision nodes in the tree, acting as a means to split the [data](https://www.ibm.com/consulting/analytics). Each question helps an individual to arrive at a final decision, which would be denoted by the leaf node. Observations that fit the criteria will follow the “Yes” branch and those that don’t will follow the alternate path. Decision trees seek to find the best split to subset the data, and they are typically trained through the Classification and Regression Tree (CART) algorithm. Metrics, such as Gini impurity, information gain, or mean square error (MSE), can be used to evaluate the quality of the split.

This decision tree is an example of a classification problem, where the class labels are "surf" and "don't surf."

While decision trees are common supervised learning algorithms, they can be prone to problems, such as bias and overfitting. However, when multiple decision trees form an ensemble in the random forest algorithm, they predict more accurate results, particularly when the individual trees are uncorrelated with each other.

### **Ensemble methods**

Ensemble learning methods are made up of a set of classifiers—e.g. decision trees—and their predictions are aggregated to identify the most popular result. The most well-known ensemble methods are bagging, also known as bootstrap aggregation, and boosting. In 1996, [Leo Breiman](https://link.springer.com/content/pdf/10.1007/BF00058655.pdf) introduced the bagging method; in this method, a random sample of data in a training set is selected with replacement—meaning that the individual data points can be chosen more than once. After several data samples are generated, these models are then trained independently, and depending on the type of task—i.e. regression or classification—the average or majority of those predictions yield a more accurate estimate. This approach is commonly used to reduce variance within a noisy dataset.

### **Random forest algorithm**

The random forest algorithm is an extension of the bagging method as it utilizes both bagging and feature randomness to create an uncorrelated forest of decision trees. Feature randomness, also known as feature bagging or “[the random subspace method](https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf)”, generates a random subset of features, which ensures low correlation among decision trees. This is a key difference between decision trees and random forests. While decision trees consider all the possible feature splits, random forests only select a subset of those features.

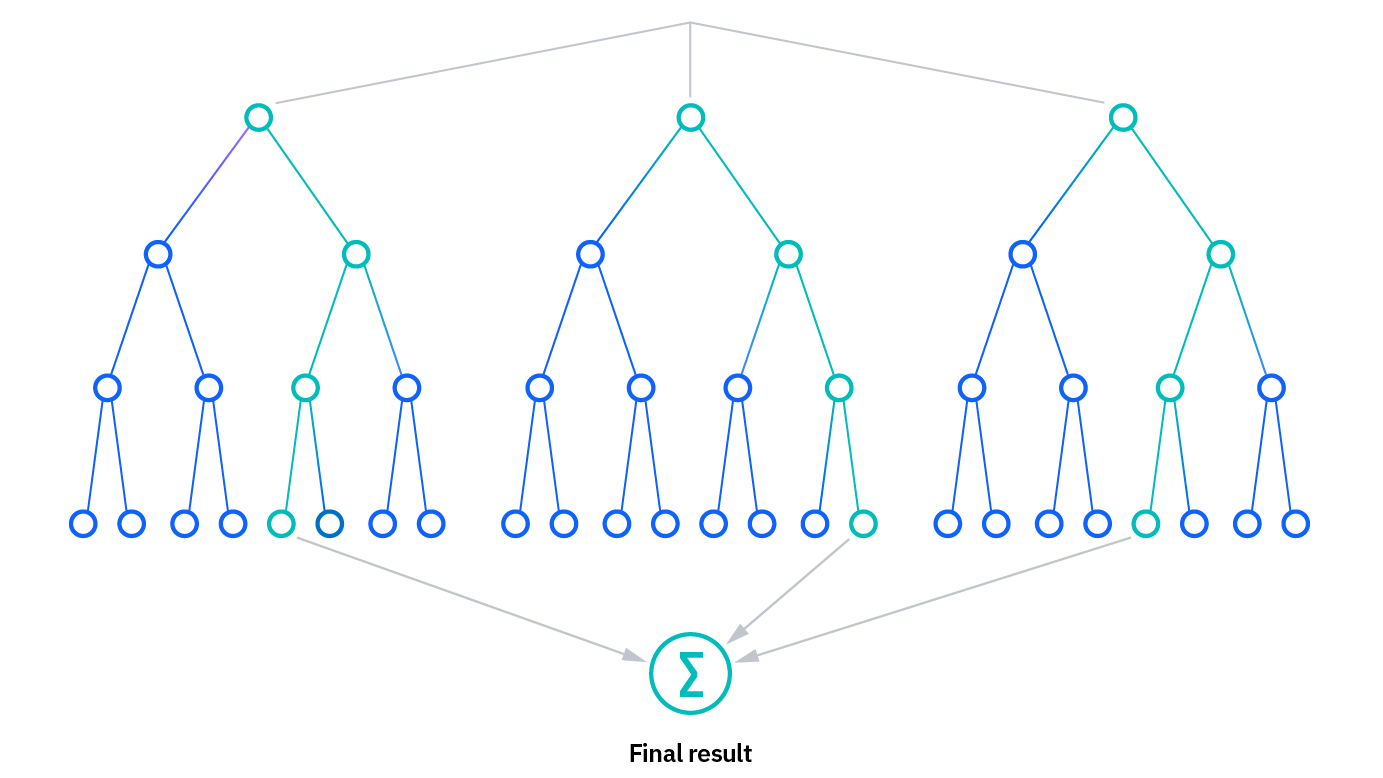
If we go back to the “should I surf?” example, the questions that I may ask to determine the prediction may not be as comprehensive as someone else’s set of questions. By accounting for all the potential variability in the data, we can reduce the risk of overfitting, bias, and overall variance, resulting in more precise predictions.

Bottom of Form

## **How it works**

Random forest algorithms have three main hyperparameters, which need to be set before training. These include node size, the number of trees, and the number of features sampled. From there, the random forest classifier can be used to solve for regression or classification problems.

The random forest algorithm is made up of a collection of decision trees, and each tree in the ensemble is comprised of a data sample drawn from a training set with replacement, called the bootstrap sample. Of that training sample, one-third of it is set aside as test data, known as the out-of-bag (oob) sample, which we’ll come back to later. Another instance of randomness is then injected through feature bagging, adding more diversity to the dataset and reducing the correlation among decision trees. Depending on the type of problem, the determination of the prediction will vary. For a regression task, the individual decision trees will be averaged, and for a classification task, a majority vote—i.e. the most frequent categorical variable—will yield the predicted class. Finally, the oob sample is then used for cross-validation, finalizing that prediction.



## **Benefits and challenges of random forest**

There are a number of key advantages and challenges that the random forest algorithm presents when used for classification or regression problems. Some of them include:

### **Key Benefits**

* Reduced risk of overfitting: Decision trees run the risk of overfitting as they tend to tightly fit all the samples within training data. However, when there’s a robust number of decision trees in a random forest, the classifier won’t overfit the model since the averaging of uncorrelated trees lowers the overall variance and prediction error.
* Provides flexibility: Since random forest can handle both regression and classification tasks with a high degree of accuracy, it is a popular method among data scientists. Feature bagging also makes the random forest classifier an effective tool for estimating missing values as it maintains accuracy when a portion of the data is missing.
* Easy to determine feature importance: Random forest makes it easy to evaluate variable importance, or contribution, to the model. There are a few ways to evaluate feature importance. Gini importance and mean decrease in impurity (MDI) are usually used to measure how much the model’s accuracy decreases when a given variable is excluded. However, permutation importance, also known as mean decrease accuracy (MDA), is another importance measure. MDA identifies the average decrease in accuracy by randomly permutating the feature values in oob samples.

### **Key Challenges**

* Time-consuming process: Since random forest algorithms can handle large data sets, they can be provide more accurate predictions, but can be slow to process data as they are computing data for each individual decision tree.
* Requires more resources: Since random forests process larger data sets, they’ll require more resources to store that data.
* More complex: The prediction of a single decision tree is easier to interpret when compared to a forest of them.

## **Random forest applications**

The random forest algorithm has been applied across a number of industries, allowing them to make better business decisions. Some use cases include:

* Finance: It is a preferred algorithm over others as it reduces time spent on data management and pre-processing tasks. It can be used to evaluate customers with high credit risk, to detect fraud, and option pricing problems.
* Healthcare: The random forest algorithm has applications within [computational biology](https://www.cs.cmu.edu/~qyj/papersA08/11-rfbook.pdf), allowing doctors to tackle problems such as gene expression classification, biomarker discovery, and sequence annotation. As a result, doctors can make estimates around drug responses to specific medications.
* E-commerce: It can be used for recommendation engines for cross-sell purposes.

[IBM randomforest](https://www.ibm.com/think/topics/random-forest)

[Geekforgeeks Randomforest](https://www.geeksforgeeks.org/dsa/random-forest-classifier-using-scikit-learn/)

## **Support Vector Machine**

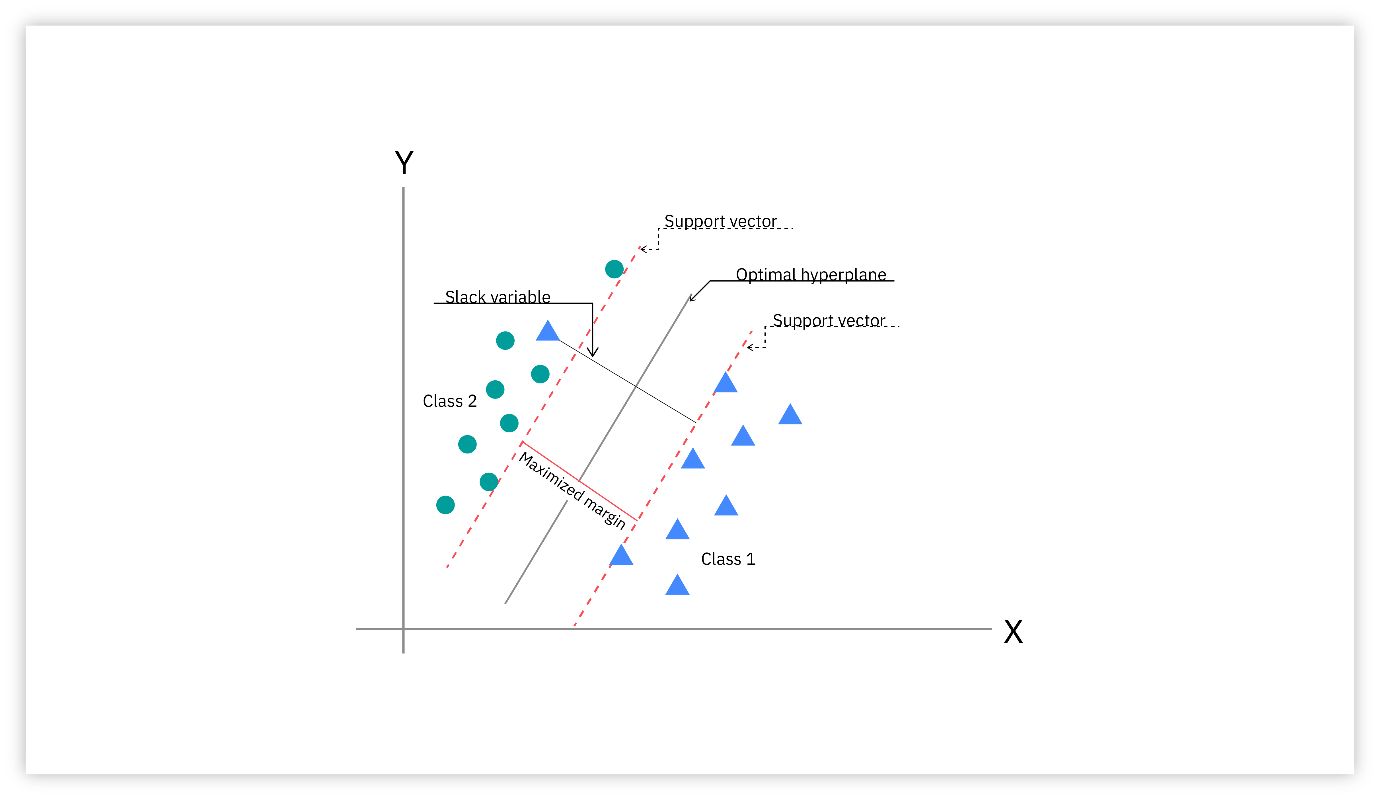
## **What are SVMs?**

A support vector machine (SVM) is a [supervised machine learning](https://www.ibm.com/topics/supervised-learning) algorithm that classifies data by finding an optimal line or hyperplane that maximizes the distance between each class in an N-dimensional space.

SVMs were developed in the 1990s by Vladimir N. Vapnik and his colleagues, and they published this work in a paper titled "Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing"1 in 1995.

SVMs are commonly used within classification problems. They distinguish between two classes by finding the optimal hyperplane that maximizes the margin between the closest data points of opposite classes. The number of features in the input data determine if the hyperplane is a line in a 2-D space or a plane in a n-dimensional space. Since multiple hyperplanes can be found to differentiate classes, maximizing the margin between points enables the algorithm to find the best decision boundary between classes. This, in turn, enables it to generalize well to new data and make accurate classification predictions. The lines that are adjacent to the optimal hyperplane are known as support vectors as these vectors run through the data points that determine the maximal margin.

The SVM algorithm is widely used in [machine learning](https://www.ibm.com/think/topics/machine-learning) as it can handle both linear and nonlinear classification tasks. However, when the data is not linearly separable, kernel functions are used to transform the data higher-dimensional space to enable linear separation. This application of kernel functions can be known as the “kernel trick”, and the choice of kernel function, such as linear kernels, polynomial kernels, radial basis function (RBF) kernels, or sigmoid kernels, depends on data characteristics and the specific use case.



## **Types of SVM classifiers**

### **Linear SVMs**

Linear SVMs are used with linearly separable data; this means that the data do not need to undergo any transformations to separate the data into different classes. The decision boundary and support vectors form the appearance of a street, and Professor Patrick Winston from MIT uses the analogy of "[fitting the widest possible street](https://ocw.mit.edu/courses/6-034-artificial-intelligence-fall-2010/resources/mit6_034f10_svm/)"2 (link resides outside ibm.com) to describe this quadratic optimization problem. Mathematically, this separating hyperplane can be represented as:

wx + b = 0

where w is the weight vector, x is the input vector, and b is the bias term.

There are two approaches to calculating the margin, or the maximum distance between classes, which are hard-margin classification and soft-margin classification. If we use a hard-margin SVMs, the data points will be perfectly separated outside of the support vectors, or "off the street" to continue with Professor Hinton’s analogy. This is represented with the formula,

(wxj + b) yj ≥ a,

and then the margin is maximized, which is represented as: max ɣ= a / ||w||, where a is the margin projected onto w.

Soft-margin classification is more flexible, allowing for some misclassification through the use of slack variables (`ξ`). The hyperparameter, C, adjusts the margin; a larger C value narrows the margin for minimal misclassification while a smaller C value widens it, allowing for more misclassified data3.

### **Nonlinear SVMs**

Much of the data in real-world scenarios are not linearly separable, and that’s where nonlinear SVMs come into play. In order to make the data linearly separable, preprocessing methods are applied to the training data to transform it into a higher-dimensional feature space. That said, higher dimensional spaces can create more complexity by increasing the risk of overfitting the data and by becoming computationally taxing. The “kernel trick” helps to reduce some of that complexity, making the computation more efficient, and it does this by replacing dot product calculations with an equivalent kernel function4.

There are a number of different kernel types that can be applied to classify data. Some popular kernel functions include:

* Polynomial kernel
* Radial basis function kernel (also known as a Gaussian or RBF kernel)
* Sigmoid kernel

### **Support vector regression (SVR)**

Support vector regression (SVR) is an extension of SVMs, which is applied to regression problems (i.e. the outcome is continuous). Similar to linear SVMs, SVR finds a hyperplane with the maximum margin between data points, and it is typically used for time series prediction.

SVR differs from [linear regression](https://www.ibm.com/topics/linear-regression) in that you need to specify the relationship that you’re looking to understand between the independent and dependent variables. An understanding of the relationships between variables and their directions is valuable when using linear regression. This is unnecessary for SVRs as they determine these relationships on their own.

### **The latest AI News + Insights**

Discover expertly curated insights and news on AI, cloud and more in the weekly Think Newsletter.

[Subscribe today](https://www.ibm.com/account/reg/signup?formid=news-urx-52954)

## **How SVMs work**

In this section, we will discuss the process of building a SVM classifier, how it compares to other supervised learning algorithms and its applications within industry today.

### **Building a SVM classifier**

#### **Split your data**

As with other machine learning models, start by splitting your data into a training set and testing set. As an aside, this assumes that you’ve already conducted an [exploratory data analysis](https://www.ibm.com/topics/exploratory-data-analysis) on your data. While this is technically not necessary to build a SVM classifier, it is good practice before using any machine learning model as this will give you an understanding of any missing data or outliers.

#### **Generate and evaluate the model**

Import an SVM module from the library of your choosing, like [scikit-learn](https://scikit-learn.org/stable/modules/svm.html) (link resides outside ibm.com). Train your training samples on the classifier and predict the response. You can evaluate performance by comparing accuracy of the test set to the predicted values. You may want to use other evaluation metrics, like f1-score, precision, or recall.

#### **Hyperparameter tuning**

Hyperparameters can be tuned to improve the performance of an SVM model. Optimal hyperparameters can be found using grid search and cross-validation methods, which will iterate through different kernel, regularization (C), and gamma values to find the best combination.

### **SVMs vs. other supervised learning classifiers**

Different machine learning classifiers can be used for the same use case. It's important to test out and evaluate different models to understand which ones perform the best. That said, it can be helpful to understand the strengths and weaknesses of each to assess its application for your use case.

#### **SVMs vs naive bayes**

Both Naive Bayes and SVM classifies are commonly used for text classification tasks. SVMs tend to perform better than Naive Bayes when the data is not linearly separable. That said, SVMs have to tune for different hyperparameters and can be more computationally expensive.

#### **SVMs vs logistic regression**

SVMs typically perform better with high-dimensional and unstructured datasets, such as image and text data, compared to logistic regression. SVMs are also less sensitive to overfitting and easier to interpret. That said, they can be more computationally expensive.

#### **SVMs vs decision trees**

SVMs perform better with high-dimensional data and are less prone to overfitting compared to decision trees. That said, decision trees are typically faster to train, particularly with smaller datasets, and they are generally easier to interpret.

#### **SVM vs. neural networks**

Similar to other model comparisons, SVMs are more computationally expensive to train and less prone to overfitting, but neural networks are considered more flexible and scalable.

### **Applications of SVMs**

While SVMs can be applied for a number of tasks, these are some of the most popular applications of SVMs across industries.

### **Text classification**

SVMs are commonly used in natural language processing (NLP) for tasks such as sentiment analysis, spam detection, and topic modeling. They lend themselves to these data as they perform well with high-dimensional data.

### **Image** **classification**

SVMs are applied in image classification tasks such as object detection and image retrieval. It can also be useful in security domains, classifying an image as one that has been tampered with.

### **Bioinformatics**

SVMs are also used for protein classification, gene expression analysis, and disease diagnosis. SVMs are often applied in [cancer research](https://pmc.ncbi.nlm.nih.gov/articles/PMC5822181/#R4) (link resides outside ibm.com) because they can detect subtle trends in complex datasets.

### **Geographic** **information** **system (GIS)**

SVMs can analyze layered geophysical structures underground, filtering out the 'noise' from electromagnetic data. They have also helped to predict the seismic liquefaction potential of soil, which is relevant to field of civil engineering.

[SVM IBM](https://www.ibm.com/think/topics/support-vector-machine)

[SVM geekforgeeeks](https://www.geeksforgeeks.org/machine-learning/support-vector-machine-algorithm/)

## **Evaluation Metrics for Classification**

When building machine learning models, it’s important to understand how well they perform. Evaluation metrics help us to measure the effectiveness of our models. Whether we are solving a classification problem, predicting continuous values or clustering data, selecting the right evaluation metric allows us to assess how well the model meets our goals. In this article, we will see commonly used evaluation metrics and discuss how to choose the right metric for our model.

**Classification Metrics**

Classification problems aim to predict discrete categories. To evaluate the performance of classification models, we use the following metrics:

**1. Accuracy**

[**Accuracy**](https://www.geeksforgeeks.org/physics/accuracy-and-precision/)is a fundamental metric used for evaluating the performance of a classification model. It tells us the proportion of correct predictions made by the model out of all predictions.

*Accuracy=NumberofCorrectPredictionsTotalNumberofPredictions*Accuracy*=*TotalNumberofPredictionsNumberofCorrectPredictions*​*

While accuracy provides a quick snapshot, it can be misleading in cases of imbalanced datasets. For example, in a dataset with 90% class A and 10% class B, a model predicting only class A will still achieve 90% accuracy but it will fail to identify any class B instances.

Accuracy is good but it gives a False Positive sense of achieving high accuracy. The problem arises due to the possibility of misclassification of minor class samples being very high.

**2. Precision**

It measures how many of the positive predictions made by the model are actually correct. It's useful when the cost of false positives is high such as in medical diagnoses where predicting a disease when it’s not present can have serious consequences.

*Precision=TPTP+FP*Precision*=*TP*+*FPTP*​*

Where:

* TP = True Positives
* FP = False Positives

[Precision](https://www.geeksforgeeks.org/physics/accuracy-and-precision/) helps ensure that when the model predicts a positive outcome, it’s likely to be correct.

**3. Recall**

[Recall](https://www.geeksforgeeks.org/machine-learning/precision-recall-curve-ml/)or Sensitivity measures how many of the actual positive cases were correctly identified by the model. It is important when missing a positive case (false negative) is more costly than false positives.

*Recall=TPTP+FN*Recall*=*TP*+*FNTP*​*

Where:

* FN = False Negatives

In scenarios where catching all positive cases is important (like disease detection), recall is a key metric.

**4. F1 Score**

The[F1 Score](https://www.geeksforgeeks.org/machine-learning/f1-score-in-machine-learning/) is the harmonic mean of **precision**and **recall**. It is useful when we need a balance between precision and recall as it combines both into a single number. A high F1 score means the model performs well on both metrics. Its range is [0,1].

Lower recall and higher precision gives us great accuracy but then it misses a large number of instances. More the F1 score better will be performance. It can be expressed mathematically in this way:

*F1 Score=2×Precision×RecallPrecision+RecallF1 Score=2×Precision+RecallPrecision×Recall​*

**5. Logarithmic Loss (Log Loss)**

[Log loss](https://www.geeksforgeeks.org/machine-learning/ml-log-loss-and-mean-squared-error/) measures the uncertainty of the model’s predictions. It is calculated by penalizing the model for assigning low probabilities to the correct classes. This metric is used in multi-class classification and is helpful when we want to assess a model’s confidence in its predictions. If there are N  samples belonging to the M class, then we calculate the Log loss in this way:

*Logarithmic Loss=−1N∑i=1N∑j=1Myij⋅log⁡(pij)Logarithmic Loss=−N1​∑i=1N​∑j=1M​yij​⋅log(pij​)*

Where:

* yij*yij*​=Actual class (0 or 1) for sample i*i* and class j*j*
* pij*pij*​ =Predicted probability for sample i*i* and class j*j*

The goal is to minimize Log Loss, as a lower Log Loss shows higher prediction accuracy.

**6. Area Under Curve (AUC) and ROC Curve**

It is useful for binary classification tasks. The [AUC](https://www.geeksforgeeks.org/machine-learning/auc-roc-curve/)value represents the probability that the model will rank a randomly chosen positive example higher than a randomly chosen negative example. AUC ranges from 0 to 1 with higher values showing better model performance.

**1. True Positive Rate(TPR)**

Also known as **sensitivity**or **recall**, the True Positive Rate measures how many actual positive instances were correctly identified by the model. It answers the question: "Out of all the actual positive cases, how many did the model correctly identify?"

Formula:

*TPR=TPTP+FN*TPR*=*TP*+*FNTP*​*    

Where:

* TP = True Positives (correctly predicted positive cases)
* FN = False Negatives (actual positive cases incorrectly predicted as negative)

**2. True Negative Rate(TNR)**

Also called **specificity**, the True Negative Rate measures how many actual negative instances were correctly identified by the model. It answers the question: "Out of all the actual negative cases, how many did the model correctly identify as negative?"

Formula:

*TNR=TNTN+FP*TNR*=*TN*+*FPTN*​* 

Where:

* TN = True Negatives (correctly predicted negative cases)
* FP = False Positives (actual negative cases incorrectly predicted as positive)

**3. False Positive Rate(FPR)**

It measures how many actual negative instances were incorrectly classified as positive. It’s a key metric when the cost of false positives is high such as in fraud detection.

Formula:

*FPR=FPFP+TN*FPR*=*FP*+*TNFP*​*

Where:

* FP = False Positives (incorrectly predicted positive cases)
* TN = True Negatives (correctly predicted negative cases)

**4. False Negative Rate(FNR)**

It measures how many actual positive instances were incorrectly classified as negative. It answers: "Out of all the actual positive cases, how many were misclassified as negative?"

Formula:

*FNR=FNFN+TP*FNR*=*FN*+*TPFN*​*

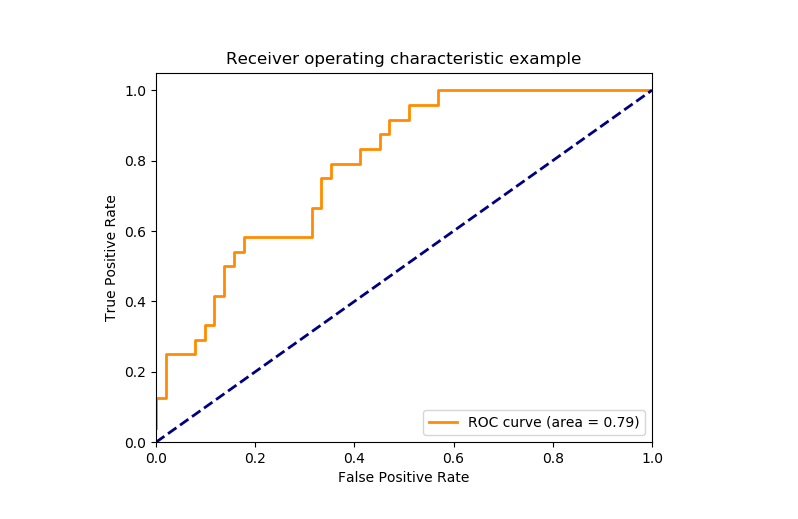
Where:

* FN = False Negatives (incorrectly predicted negative cases)
* TP = True Positives (correctly predicted positive cases)

**ROC Curve**

It is a graphical representation of the True Positive Rate (TPR) vs the False Positive Rate (FPR) at different classification thresholds. The curve helps us visualize the trade-offs between sensitivity (TPR) and specificity (1 - FPR) across various thresholds. Area Under Curve (AUC) quantifies the overall ability of the model to distinguish between positive and negative classes.

* AUC = 1: Perfect model (always correctly classifies positives and negatives).
* AUC = 0.5: Model performs no better than random guessing.
* AUC < 0.5: Model performs worse than random guessing (showing that the model is inverted).

ROC Curve for Evaluation of Classification Models

**7. Confusion Matrix**

[Confusion matrix](https://www.geeksforgeeks.org/machine-learning/confusion-matrix-machine-learning/) creates a N X N matrix, where N is the number of classes or categories that are to be predicted. Here we have N = 2, so we get a 2 X 2 matrix. Suppose there is a problem with our practice which is a [binary classification](https://www.geeksforgeeks.org/deep-learning/binary-cross-entropy-log-loss-for-binary-classification/). Samples of that classification belong to either Yes or No. So, we build our classifier which will predict the class for the new input sample. After that, we tested our model with 165 samples and we get the following result.

| **n=165** | **Predicted No** | **Predited Yes** |
| --- | --- | --- |
| **Actual No** | 50 | 10 |
| **Actual Yes** | 5 | 100 |

There are 4 terms we should keep in mind:

1. **True Positives:** It is the case where we predicted Yes and the real output was also Yes.
2. **True Negatives:** It is the case where we predicted No and the real output was also No.
3. **False Positives:** It is the case where we predicted Yes but it was actually No.
4. **False Negatives:** It is the case where we predicted No but it was actually Yes.

[geekforgeeks evclas](https://www.geeksforgeeks.org/machine-learning/metrics-for-machine-learning-model/)

[analytics evmclas](https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/)

## **Saving ML models**

When working on real-world machine learning (ML) use cases, [finding the best algorithm/model](https://neptune.ai/blog/ml-model-evaluation-and-selection) is not the end of your responsibilities. It is crucial to save, store, and package these models for their future use and deployment to production.

These practices are needed for a number of reasons:

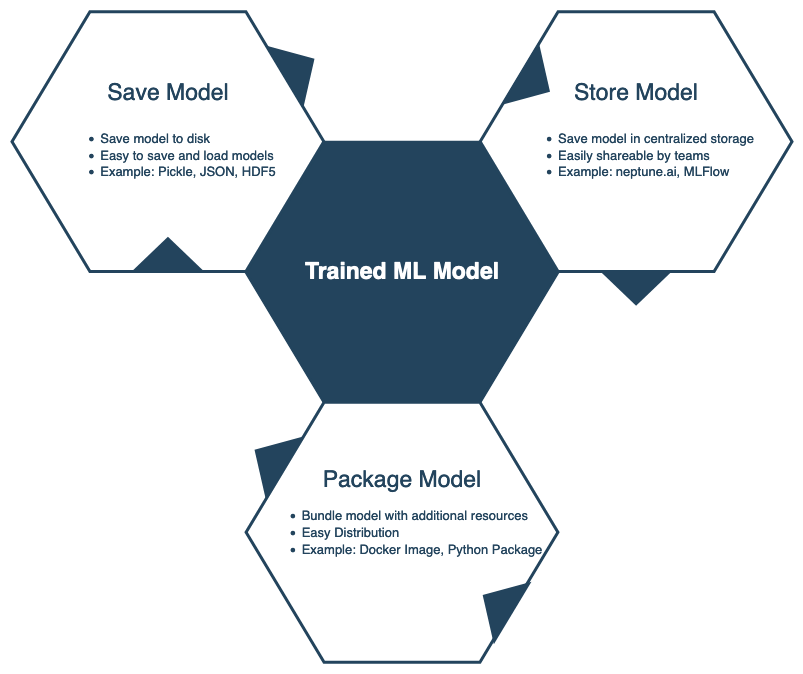
* **Backup:** A trained model can be saved as a backup in case the original data is damaged or destroyed.
* **Reusability & reproducibility:** Building ML models is time-consuming by nature. To save cost and time, it becomes essential that your model gets you the same results every time you run it. Saving and storing your model the right way takes care of this.
* **Deployment:** When [deploying a trained model](https://neptune.ai/blog/model-deployment-strategies) in a real-world setting, it becomes necessary to package it for easy deployment. This makes it possible for other systems and applications to use the same model without much hassle.

To reiterate, while saving and storing ML models allow ease of sharing, reusability, and reproducibility; packaging the models enables quick and painless deployment. These 3 operations work in harmony to simplify the whole model management process.

In this article, you will learn about different methods of saving, storing, and packaging a trained machine-learning model, along with the pros and cons of each method. But before that, you must understand the distinction between these three terms.

**Save vs package vs store ML models**

Although all these terms look similar, they are not the same.

Saving vs Storing vs Packaging ML Models | Source: Author

**Saving** a model refers to the process of saving the model’s parameters, weights, etc., to a file. Usually, all ML and DL models provide some kind of method (eg. model.save()) for saving the models. But you must be aware that save is a single action and gives only a model binary file, so you still need code to make your ML application production-ready.

**Packaging,** on the other hand, refers to the process of bundling or containerizing the necessary components of a model, such as the model file, dependencies, configuration files, etc., into a single deployable package. The goal of a package is to make it easier to distribute and deploy the ML model in a production environment.

Once packaged, a model can be deployed across different environments, which allows the model to be used in various production settings such as web applications, mobile applications, etc. Docker is one of the tools which allows you to do this.

**Storing** the ML model refers to the process of saving the trained model files in a centralized storage that can be accessed anytime when needed. When storing a model, you normally choose some sort of storage from where you can fetch your model and use it anytime. The model registry is a category of tools that solve this issue for you.

Now let’s see how we can save our model.

**How to save a trained model in Python?**

In this section, you will see different ways of saving machine learning (ML) as well as deep learning (DL) models. To begin with, let’s create a simple classification model using the most famous[Iris-dataset](https://archive.ics.uci.edu/ml/datasets/iris).

***Note:****The focus of this article is not to show you how you can create the best ML model but to explain how effectively you can save trained models.*

You first need to load the required dependencies and the iris dataset as follows:

*# load dependencies*

**import** pandas **as** pd

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.preprocessing **import** StandardScaler

**from** sklearn.neighbors **import** KNeighborsClassifier

**from** sklearn.metrics **import** classification\_report, confusion\_matrix

*# load the dataset*

url = "iris.data"

*# column names to use*

names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

*# read the dataset from the URL*

dataset = pd.read\_csv(url, names=names)

*# check the first few rows of iris-classification data*

dataset.head()

Next, you need to split the data into training and testing sets and apply the required preprocessing stages, such as feature standardization.

*# separate the independent and dependent features*

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, 4].values

*# Split dataset into random training and testing subsets*

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,

y, test\_size=0.20)

*# feature standardization*

scaler = StandardScaler()

scaler.fit(X\_train)

X\_train = scaler.transform(X\_train)

X\_test = scaler.transform(X\_test)

Finally, you need to train a classification model (feel free to choose any) on training data and check its performance on testing data.

*# training a KNN classifier*

model = KNeighborsClassifier(n\_neighbors=5)

model.fit(X\_train, y\_train)

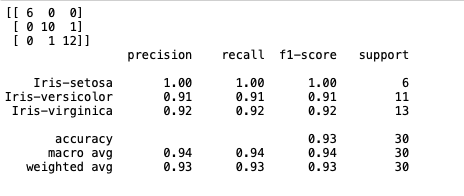
*# make predictions on the testing data*

y\_predict = model.predict(X\_test)

*# check results*

print(confusion\_matrix(y\_test, y\_predict))

print(classification\_report(y\_test, y\_predict))

Iris classification results | Source: Author

Now you have an ML model that you want to save for future use. The first way to save an ML model is by using the[pickle](https://docs.python.org/3/library/pickle.html) file.

**Saving trained model with pickle**

The[pickle](https://docs.python.org/3/library/pickle.html) module can be used to serialize and deserialize the Python objects. **Pickling** is the process of converting a Python object hierarchy into a byte stream, while **Unpickling** is the process of converting a byte stream (from a binary file or other object that appears to be made of bytes) back to an object hierarchy.

For saving the ML models used as a pickle file, you need to use the **Pickle**module that already comes with the default[Python](https://www.python.org/downloads/) installation.

To save your iris classifier model you simply need to decide on a filename and dump your model to a pickle file like this:

**import** pickle

*# save the iris classification model as a pickle file*

model\_pkl\_file = "iris\_classifier\_model.pkl"

**with** open(model\_pkl\_file, 'wb') **as** file:

pickle.dump(model, file)

As you can see the file is opened in **wb (write binary)** mode for saving the model as bytes. Also, the **dump()**method stores the model in the given pickle file.

You can also load this model using the **load()**method of the pickle module. Now you need to open the file in **rb (read binary)** mode to load the saved model.

*# load model from pickle file*

**with** open(model\_pkl\_file, 'rb') **as** file:

model = pickle.load(file)

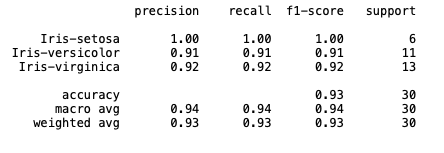
*# evaluate model*

y\_predict = model.predict(X\_test)

*# check results*

print(classification\_report(y\_test, y\_predict))

Once loaded you can use this model to make predictions.

Iris classification result | Source: Author

**Pros of the Python pickle approach**

* 1Pickling comes as the standard module in Python which makes it easy to use for saving and restoring ML models.
* 2Pickle files can handle most Python objects including custom objects, making it a versatile way to save models.
* 3For small models, pickle approach is quite fast and efficient.
* 4When an ML model is unpickled, it is restored to its previous state, including any variables or configurations. This makes Python pickle files one of the best alternatives for saving ML models.

**Cons of the Python Pickle Approach**

* 1If you unpickle untrusted data, pickling could pose a security threat. Unpickling an object can execute malicious code, so it’s crucial to only unpickle information from reliable sources.
* 2Pickled objects’ use may be constrained in some circumstances since they cannot be transferred between different Python versions or operating systems.
* 3For models with a big memory footprint, pickling can result in the creation of huge files, which can be problematic.
* 4Pickling can make it difficult to track changes to a model over time, especially if the model is updated frequently and it is not feasible to create multiple pickle files for different versions of models that you try.

Pickle is most suited for small-size models and also has some security issues, these reasons are enough to look for another alternative for saving the ML models. Next, let’s discuss **Joblib**to save and load ML models.

***Note:****In the upcoming sections you will see the same iris classifier model to be saved using different techniques.*

**Saving trained model with Joblib**

[Joblib](https://joblib.readthedocs.io/en/latest/) is a set of tools (typically part of the[Scipy](https://scipy.org/) ecosystem) that provide lightweight pipelining in Python. It majorly focuses on disk-caching, memoization, and parallel computing and is used for saving and loading Python objects. Joblib has been specifically optimized for[NumPy](https://numpy.org/) arrays to make it fast and reliable for ML models that have a lot of parameters.

To save large models with Joblib, you need to use the Python **Joblib** module that comes preinstalled with Python.

**import** joblib

*# save model with joblib*

filename = 'joblib\_model.sav'

joblib.dump(model, filename)

To save the model, you need to define a filename with a *‘.sav’* or *‘.pkl’* extension and call the **dump()**method from Joblib.

Similar to pickle, Joblib provides the **load()** method to load the saved ML model.

*# load model with joblib*

loaded\_model = joblib.load(filename)

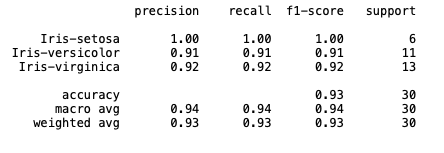
*# evaluate model*

y\_predict = model.predict(X\_test)

*# check results*

print(classification\_report(y\_test, y\_predict))

After loading the model with Joblib you are free to use it on the data to make predictions.

Iris classification results | Source: Author

**Pros of saving ML models with Joblib**

* 1Fast and effective performance is a key component of Joblib, especially for models with substantial memory requirements.
* 2The serialization and deserialization process can be parallelized via Joblib, which can enhance performance on multi-core machines.
* 3For models that demand a lot of memory, Joblib employs a memory-mapped file format to reduce memory utilization.
* 4Joblib offers various security features, such as a whitelist of secure functions that can be utilized during deserialization, to assist safeguard against untrusted data.

**Cons of Saving ML Models with Joblib**

* 1Joblib is optimized for numpy arrays, and may not work as well with other object types.
* 2Joblib offers less flexibility than Pickle because there are fewer options available for configuring the serialization process.
* 3Compared to Pickle, Joblib is less well known, which can make it more difficult to locate help and documentation around it.

Although Joblib solves the major issues faced by pickle, it has some issues on its own. Next, you will see how you can manually save and restore the models using JSON.

**Saving trained model with JSON**

When you want to have full control over the save and restore procedure of your ML model,[JSON](https://docs.python.org/3/library/json.html) comes into play. Unlike the other two methods, this method does not directly dump the ML model to a file; instead, you need to explicitly define the different parameters of your model to save them.

To use this method, you need to use the Python **json** module that again comes along with the default Python installation. Using the JSON method requires additional effort to write all parameters that an ML model contains. To save the model using JSON, let’s create a function like this:

**import** json

*# create json save function*

**def** **save\_json**(model, filepath, X\_train, y\_train):

saved\_model = {}

saved\_model["algorithm"] = model.get\_params()['algorithm'],

saved\_model["max\_iter"] = model.get\_params()['leaf\_size'],

saved\_model["solver"] = model.get\_params()['metric'],

saved\_model["metric\_params"] = model.get\_params()['metric\_params'],

saved\_model["n\_jobs"] = model.get\_params()['n\_jobs'],

saved\_model["n\_neighbors"] = model.get\_params()['n\_neighbors'],

saved\_model["p"] = model.get\_params()['p'],

saved\_model["weights"] = model.get\_params()['weights'],

saved\_model["X\_train"] = X\_train.tolist() **if** X\_train **is** **not** **None** **else** "None",

saved\_model["y\_train"] = y\_train.tolist() **if** y\_train **is** **not** **None** **else** "None"

json\_txt = json.dumps(saved\_model, indent=4)

**with** open(filepath, "w") **as** file:

file.write(json\_txt)

*# save the iris-classification model in a json file*

file\_path = 'json\_model.json'

save\_json(model, file\_path, X\_train, y\_train)

You see how you need to define each model parameter and the data to store it in JSON. Different models have different methods to check out the parameter details. For example, the **get\_params()**for[KNeighboursClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html) gives the list of all the hyperparameters in the model. You need to save all these hyperparameters and data values in a dictionary which is then dumped into a file with the *‘.json’* extension.

To read this JSON file you just need to open it and access the parameters as follows:

*# create json load function*

**def** **load\_json**(filepath):

**with** open(filepath, "r") **as** file:

saved\_model = json.load(file)

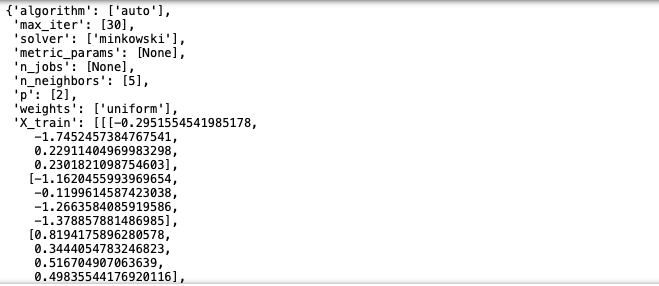
**return** saved\_model

*# load model configurations*

saved\_model = load\_json('json\_model.json')

saved\_model

In the above code, a function **load\_json()**is created that opens the JSON file in read mode and returns all the parameters and data as a dictionary.

JSON Loaded Model | Source: Author

Unfortunately, you can not use the saved model directly with JSON, you need to read these parameters and data to retrain the model all by yourself.

**Pros of saving ML models with JSON**

* 1Models that need to be exchanged between various systems can be done so using JSON, which is a portable format that can be read by a wide variety of programming languages and platforms.
* 2JSON is a text-based format that is easy to read and understand, making it a good choice for models that need to be inspected or edited by humans.
* 3In comparison to Pickle or Joblib, JSON is a lightweight format that creates smaller files, which can be crucial for models that must be transferred over the internet.
* 4Unlike pickle, which executes code during deserialization, JSON is a secure format that minimizes security threats.

**Cons of Saving ML Models with JSON**

* 1Because JSON only supports a small number of data types, it could not be compatible with sophisticated machine learning models that employ unique data types.
* 2In particular, for large models, JSON serialization and deserialization can be slower than other formats.
* 3Compared to alternative formats, JSON offers less flexibility and may take more effort to tailor the serialization procedure.
* 4JSON is a lossy format that may not preserve all of the information in the original model, which can be a problem for models that require exact replication.

To ensure security and JSON/pickle benefits, you can save your model to a dedicated database. Next, you will see how you can save an ML model in a database.

**Saving deep learning model with TensorFlow Keras**

[TensorFlow](https://www.tensorflow.org/) is a popular framework for training DL-based models, and[Ker](https://keras.io/)as is a wrapper for TensorFlow. A neural network design with numerous layers and a set of labeled data are used to train deep learning models. These models have two major components, Weights and Network architecture, that you need to save to restore them for future use. Typically there are two ways to save deep learning models:

1. Save the model architecture in a JSON or YAML file and weights in an[HDF5](https://docs.h5py.org/en/stable/quick.html) file.
2. Save both model and architecture both in HDF5,[protobuf](https://protobuf.dev/getting-started/pythontutorial/), or[tflite](https://pypi.org/project/tflite/) file.

You can refer to any one way to do this, but the widely used method is to save the model weights and architecture together in an HDF5 file.

To save a deep learning model in TensorFlow Keras, you can use the **save()** method of the Keras **Model** object. This method saves the entire model, including the model architecture,[optimizer](https://keras.io/api/optimizers/), and weights, in a format that can be loaded later to make predictions.

Here’s an example code snippet that shows how to save a TensorFlow Keras-based DL model:

*# import tensorflow dependencies*

**from** tensorflow.keras.models **import** Sequential, model\_from\_json

**from** tensorflow.keras.layers **import** Dense

*# define model architecture*

model = Sequential()

model.add(Dense(12, input\_dim=4, activation='relu'))

model.add(Dense(8, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

*# Compile model*

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

*# Fit the model*

model.fit(X\_train, y\_train, epochs=150, batch\_size=10, verbose=0)

*# save model and its architecture*

model.save('model.h5')

This is it, you just need to define the model architecture, train the models with appropriate settings, and finally save it using the **save()**method.

Loading the saved models with Keras is as easy as reading a file in Python. You just need to call the **load\_model()**method by providing the model file path and your model will be loaded.

*# define dependency*

**from** tensorflow.keras.models **import** load\_model

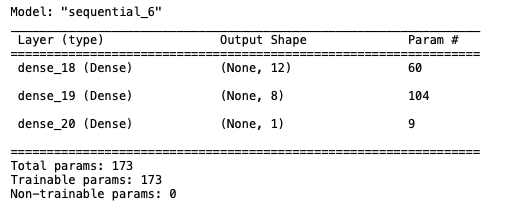
*# load model*

model = load\_model('model.h5')

*# check model info*

model.summary()

Your model is now loaded for use.

Tensorflow loaded model | Source: Author

**Pros of saving models with TensorFlow Keras**

* 1Saving and loading models in TensorFlow Keras is very straightforward using the save() and load\_model() functions. This makes it easy to save and share models with others or to deploy them to production.
* 2The whole model architecture, optimizer, and weights are saved in one file when you save a Keras model. With no need to bother about loading the architecture and weights separately, it is simple to load the model and generate predictions.
* 3TensorFlow Keras supports several file formats for saving models, including the HDF5 format (.h5), the TensorFlow SavedModel format (.pb), and the TensorFlow Lite format (.tflite). This gives you flexibility in choosing the format that best suits your needs.

**Cons of Saving Models with TensorFlow Keras**

* 1When you save a Keras model, the resulting file can be quite large, especially if you have a large number of layers or parameters. This can make it challenging to share or deploy the model, especially in situations where bandwidth or storage space is limited.
* 2Models saved with one version of TensorFlow Keras could not work with another. If you try to load a model that was saved with a different version of Keras or TensorFlow, this may result in problems.
* 3Although it’s simple to save a Keras model, you’re only able to use the features that Keras offers for storing models. A different framework or strategy may be required if you require more flexibility in the way models are saved or loaded.

There is one more widely used framework named Pytorch for training the DL-based models. Let’s check how you can save Pytorch-based deep learning models with Python.

**Saving deep learning model with Pytorch**

Developed by Facebook, Pytorch is one of the highly used frameworks for developing DL-based solutions. It provides a dynamic computational graph, which allows you to modify your model on-the-fly, making it ideal for research and experimentation. It uses *‘.pt’* and *‘.pth’* file formats to save model architecture and its weights.

To save a deep learning model in PyTorch, you can use the **save()** method of the PyTorch **torch.nn.Module** object. This method saves the entire model, including the model architecture and weights, in a format that can be loaded later to make predictions.

Here’s an example code snippet that shows how to save a PyTorch model:

*# import dependencies*

**import** torch

**import** torch.nn **as** nn

**import** numpy **as** np

*# convert data numpy arrays to tensors*

X\_train = torch.FloatTensor(X\_train)

X\_test = torch.FloatTensor(X\_test)

y\_train = torch.LongTensor(y\_train)

y\_test = torch.LongTensor(y\_test)

*# define model architecture*

**class** **NeuralNetworkClassificationModel**(nn.Module):

**def** **\_\_init\_\_**(self,input\_dim,output\_dim):

super(NeuralNetworkClassificationModel,self).\_\_init\_\_()

self.input\_layer = nn.Linear(input\_dim,128)

self.hidden\_layer1 = nn.Linear(128,64)

self.output\_layer = nn.Linear(64,output\_dim)

self.relu = nn.ReLU()

**def** **forward**(self,x):

out = self.relu(self.input\_layer(x))

out = self.relu(self.hidden\_layer1(out))

out = self.output\_layer(out)

**return** out

*# define input and output dimensions*

input\_dim = 4

output\_dim = 3

model = NeuralNetworkClassificationModel(input\_dim,output\_dim)

*# create our optimizer and loss function object*

learning\_rate = 0.01

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(),lr=learning\_rate)

*# define training steps*

**def** **train\_network**(model,optimizer,criterion,X\_train,y\_train,X\_test,y\_test,num\_epochs,train\_losses,test\_losses):

**for** epoch **in** range(num\_epochs):

*# clear out the gradients from the last step loss.backward()*

optimizer.zero\_grad()

*# forward feed*

output\_train = model(X\_train)

*# calculate the loss*

loss\_train = criterion(output\_train, y\_train)

*# backward propagation: calculate gradients*

loss\_train.backward()

*# update the weights*

optimizer.step()

output\_test = model(X\_test)

loss\_test = criterion(output\_test,y\_test)

train\_losses[epoch] = loss\_train.item()

test\_losses[epoch] = loss\_test.item()

**if** (epoch + 1) % 50 == 0:

print(f"Epoch { epoch+1 }/{ num\_epochs }, Train Loss: { loss\_train.item():.4f }, Test Loss: {loss\_test.item():.4f}")

*# train model*

num\_epochs = 1000

train\_losses = np.zeros(num\_epochs)

test\_losses = np.zeros(num\_epochs)

train\_network(model,optimizer,criterion,X\_train,y\_train,X\_test,y\_test,num\_epochs,train\_losses,test\_losses)

*# save model*

torch.save(model, 'model\_pytorch.pt')

Unlike Tensorflow, Pytorch allows you to have more control over the model training, as seen in the above code. After training the model, you can save the weights and their architecture using **save()**method.

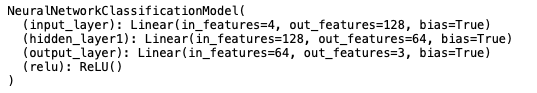
Loading the saved model with Pytorch requires the use of **load()**method.

*# load model*

model = torch.load('model\_pytorch.pt')

*# check model summary*

model.eval()

Pytorch loaded model | Source: Author

**Pros of saving models with Pytorch**

* 1The computational graph used by PyTorch is dynamic, meaning it is built as the program is run. This allows for more flexibility in modifying the model during training or inference.
* 2For dynamic models, such as those with variable-length inputs or outputs, which are frequent in natural language processing (NLP) and computer vision, PyTorch offers improved support.
* 3Given that PyTorch is written in Python and functions well with other Python libraries like NumPy and pandas, manipulating data both before and after training is simple.

**Cons of Saving Models with Pytorch**

* 1Even though PyTorch provides an accessible API, there may be a steep learning curve for newcomers to deep learning or Python programming.
* 2Since PyTorch is essentially a framework for research, it might not have as many tools for production deployment as other deep learning frameworks like TensorFlow or Keras.

This isn’t it, you can use model registry platforms to save DL-based models as well, specially the ones with large size. This makes it easy to deploy and maintain them without requiring extra effort from developers.

You can find the dataset and code used in this article [here](https://github.com/gouravsinghbais/How-to-Save-Trained-Model-in-Python).

**How to package ML models?**

An ML model is typically optimized for performance on the training dataset and the specific environment in which it is trained. But, when it comes to deploying the models in different environments, such as a production environment, there could be various challenges.

These challenges are but not limited to differences in hardware, software, and data inputs. Packaging the model makes it easier to address these problem, as it allows the model to be exported or serialized into a standard format that can be loaded and used in various environments.

There are various options available for packaging right now. By packaging the model in a standard format such as [PMML (Predictive Model Markup Language)](https://access.redhat.com/documentation/en-us/red_hat_process_automation_manager/7.3/html/designing_a_decision_service_using_pmml_models/pmml-con_pmml-models), [ONNX](https://onnx.ai/), [TensorFlow SavedModel format](https://www.tensorflow.org/guide/saved_model), etc. it becomes easier to share and collaborate on a model without being concerned about different libraries and tools used by different teams. Now, let’s check a few examples of packaging an ML model with different frameworks in Python.

**Note:** For this section as well, you will see the same iris-classification example.

**Packaging models with PMML**

Using the PMML library in Python, you can export your machine learning models to PMML format and then deploy that as a web service, a batch processing system, or a data integration platform. This can make it easier to share and collaborate on machine learning models, as well as to deploy them in various production environments.

To package an ML model using PMML you can use different modules like [sklearn2pmml](https://github.com/jpmml/sklearn2pmml), [jpmml-sklearn](https://github.com/jpmml/jpmml-sklearn), [jpmml-tensorflow](https://github.com/jpmml/jpmml-tensorflow), etc.

**Note:** To use PMML, you must have [Java Runtime](https://www.java.com/en/download/manual.jsp) installed on your system.

Here is an example code snippet that allows you to package the trained iris classifier model using PMML.

**from** sklearn2pmml **import** PMMLPipeline, sklearn2pmml

*# package iris classifier model with PMML*

sklearn2pmml(PMMLPipeline([("estimator",

model)]),

"iris\_model.pmml",

with\_repr=**True**)

In the above code, you simply need to create a PMML pipeline object by passing your model object. Then you need to save the PMML object using **sklearn2pmml()**method. That is it, now you can use this **“iris\_model.pmml”**file across different environments.

**Pros of using PMML**

* 1Since PMML is a platform-independent format, PMML models can be integrated with numerous data processing platforms and used in a variety of production situations.
* 2PMML can reduce vendor lock-in as it allows users to export and import models from different machine-learning platforms.
* 3PMML models can be easily deployed in production environments as they can be integrated with various data processing platforms and systems.

**Cons of using PMML**

* 1Some machine learning models and algorithms may not be able to be exported in PMML format as a result of the limited support.
* 2PMML is an XML-based format that can be verbose and inflexible, which may make it difficult to modify or update models after they have been exported in PMML format.
* 3It might be difficult to create PMML models, especially for complicated models with several features and interactions.

**Packaging models with ONNX**

Developed by Microsoft and Facebook, ONNX (Open Neural Network Exchange) is an open format for representing machine learning models. It allows for interoperability between different deep-learning frameworks and tools.

ONNX models can be deployed efficiently on a variety of platforms, including mobile devices, edge devices, and the cloud. It supports a variety of runtimes, including [Caffe2](https://caffe2.ai/), TensorFlow, PyTorch, and [MXNet](https://mxnet.apache.org/versions/1.9.1/), which allows you to deploy your models on different devices and platforms with minimal effort.

To save the model using ONNX, you need to have [onnx](https://github.com/onnx/onnx) and [onnxruntime](https://onnxruntime.ai/docs/get-started/with-python.html) packages downloaded in your system.

Here is an example of how you can convert the existing ML model to ONNX format.

# load dependencies

import onnxmltools

import onnxruntime

# Convert the KNeighborsClassifier model to ONNX format

onnx\_model = onnxmltools.convert\_sklearn(model)

# Save the ONNX model in a file

onnx\_file = "iris\_knn.onnx"

onnxmltools.utils.save\_model(onnx\_model, onnx\_file)

You just need to import the required modules and use the **convert\_sklearn()**method to corvet the sklearn model to the ONNX model. Once the conversion is done, using the **save\_model()** method, you can store the ONNX model in a file with the “.onnx” extension. Although here you see an example of an ML model, ONNX is majorly used for DL models.

You can also load this model using the ONNX Runtime module.

*# Load the ONNX model into ONNX Runtime*

sess = onnxruntime.InferenceSession(onnx\_file)

*# Evaluate the model on some test data*

input\_data = {"X": X\_test[:10].astype('float32')}

output = sess.run(**None**, input\_data)

You need to create a session using **InferenceSession()**method to load the ONNX model from a file and then use **sess.run()**method to make predictions from the model.

**Pros of using ONNX**

* 1With little effort, ONNX models can easily be deployed on a number of platforms, including mobile devices and the cloud. It is simple to deploy models on various hardware and software platforms thanks to ONNX’s support for a wide range of runtimes.
* 2ONNX models are optimized for performance, which means that they can run faster and consume fewer resources than models in other formats.

**Cons of using ONNX**

* 1ONNX is primarily designed for deep learning models and may not be suitable for other types of machine learning models.
* 2ONNX models may not be compatible with all versions of different deep learning frameworks, which may require additional effort to ensure compatibility.

**Packaging models with Tensorflow SavedModel**

Tensorflow’s SavedModel format allows you to easily save and load your deep learning models, and it ensures compatibility with other Tensorflow tools and platforms. Additionally, it provides a streamlined and efficient way to deploy our models in production environments.

SavedModel supports a wide range of deployment scenarios, including serving models with [Tensorflow Serving](https://www.tensorflow.org/tfx/guide/serving), deploying models to mobile devices with [Tensorflow Lite](https://www.tensorflow.org/lite), and exporting models to other ML libraries such as ONNX.

 It provides a simple and streamlined way to save and load Tensorflow models. The API is easy to use and well-documented, and the format is designed to be efficient and scalable.

**Note:**You can use the same TensorFlow model trained in the above section.

To save the model in SavedModel format, you can use the following lines of code:

**import** tensorflow **as** tf

*# using SavedModel format to save the model*

tf.saved\_model.save(model, "my\_model")

You can also load the model with **load()** method.

*# Load the model*

loaded\_model = tf.saved\_model.load("my\_model")

**Pros of using Tensorflow SavedModel**

* 1SavedModel is platform-independent and version-compatible, which makes it easy to share and deploy models across different platforms and versions of TensorFlow.
* 2A variety of deployment scenarios are supported by SavedModel, including exporting models to other ML libraries like ONNX, serving models with TensorFlow Serving, and distributing models to mobile devices using TensorFlow Lite.
* 3SavedModel is optimized for training and inference, with support for distributed training and the ability to use GPUs and TPUs to accelerate training.

**Cons of using Tensorflow SavedModel**

* 1SavedModel files can be large, particularly for complex models, which can make them difficult to store and transfer.
* 2Given that SavedModel is exclusive to TensorFlow, its compatibility with other ML libraries and tools may be constrained.
* 3The saved model is a binary file that can be difficult to inspect, making it harder to understand the details of the model’s architecture and operation.

Now that you have seen multiple ways of packaging ML and DL models, you must also be aware that there are various tools available that provide infrastructure to package, deploy and serve these models. Two of the popular ones are [BentoML](https://www.bentoml.com/) and [MLflow](https://mlflow.org/).

**BentoML**

BentoML is a flexible framework for building and deploying production-ready machine learning services. It allows data scientists to packaging their trained models, their dependencies, and the infrastructure code required to serve the model into a reusable package called a “Bento”.

BentoML supports various machine learning frameworks and deployment platforms and provides a unified API for managing the lifecycle of the model. Once a model is packaged as a Bento, it can be deployed to various serving platforms like [AWS Lambda](https://aws.amazon.com/lambda/), [Kubernetes](https://kubernetes.io/), or [Docker](https://www.docker.com/). BentoML also offers an API server that can be used to serve the model via a REST API. You can know more about it [here](https://github.com/bentoml/BentoML).

**MLflow**

MLflow is an open-source platform for managing the end-to-end machine learning lifecycle. It provides a comprehensive set of tools for tracking experiments, packaging code, and dependencies, and deploying models.

MLflow allows data scientists to easily package their models in a standard format that can be deployed to various platforms like [AWS SageMaker](https://aws.amazon.com/sagemaker/), [Azure ML](https://azure.microsoft.com/en-us/products/machine-learning/), and [Google Cloud AI Platform](https://cloud.google.com/ai-platform/docs/technical-overview). The platform also provides a model registry to manage model versions and track their performance over time. Additionally, MLflow offers a REST API for serving models, which can be easily integrated into web applications or other services.

**How to store ML models?**

Now that we know about saving models let’s see how we can store them to facilitate their quick and easy retrieval.

**Storing ML models in a database**

There is also scope for you to save your ML models in relational databases[PostgreSQL](https://www.postgresql.org/),[MySQL](https://www.mysql.com/),[Oracle SQL](https://www.oracle.com/in/database/sqldeveloper/), etc. or NoSQL databases like[MongoDB](https://www.mongodb.com/),[Cassandra](https://cassandra.apache.org/_/index.html), etc. The choice of database totally depends on factors such as the type and volume of data being stored, the performance and scalability requirements, and the specific needs of the application.

PostgreSQL is a popular choice when working on ML models that provide support for storing and manipulating structured data. Storing ML models in PostgreSQL provides an easy way to keep track of different versions of a model and manage them in a centralized location.

Additionally, it allows for easy sharing of models across a team or organization. However, it’s important to note that storing large models in a database can increase database size and query times, so it’s important to consider the storage capacity and performance of your database when storing models in PostgreSQL.

To save an ML model in a database like PostgreSQL, you need to first Convert the trained model into a serialized format, such as a byte stream (pickle object) or JSON.

**import** pickle

*# serialize the model*

model\_bytes = pickle.dumps(model)

Then open a connection to the database and create a table or collection to store the serialized model. For this, you need to use the **psycopg2**library of Python, which lets you connect to the PostgreSQL database. You can download this library with the help of the Python package installer.

$ pip install psycopg2-**binary**

Then you need to establish a connection to the database to store the ML model like this:

import psycopg2

#  establishing the connection to the Database

conn = psycopg2.connect(

  database="database-name", user=user-name, password='your-password', host='127.0.0.1', port= '5432'

)

To perform any operation on the database, you need to create a[cursor](https://www.doc.ic.ac.uk/project/2012/wmproject2013/chandra/psycopg2-2.5.1/doc/html/cursor.html) object that will help you to execute queries in your Python program.

# create a cursor

cur = conn.cursor()

With the help of this cursor, you can now execute the **CREATE TABLE** query to create a new table.

cur.execute("**CREATE** **TABLE** models (**id** INT PRIMARY **KEY** **NOT** NULL, **name** CHAR(50), **model** BYTEA)")

***Note: Make sure that the model object type is BYTEA.***

Finally, you can store the model and other metadata information using the **INSERT INTO** command.

# Insert the serialized model into the database

cur.execute("INSERT INTO models (id, name, model) VALUES (%s, %s, %s)", (1, 'iris-classifier', model\_bytes))

conn.commit()

# Close the database connection

cur.close()

conn.close()

Once all the operations are done, close the cursor and connection to the database.

Finally, to read the model from the database, you can use the **SELECT** command by filtering the model either on name or id.

import psycopg2

import pickle

# Connect to the database

conn = psycopg2.connect(

database="database-name", user=user-name, password='your-password', host='127.0.0.1', port= '5432'

)

# Retrieve the serialized model from the database

cur = conn.cursor()

cur.execute("SELECT model FROM models WHERE name = %s", ('iris-classifier',))

model\_bytes = cur.fetchone()[0]

# Deserialize the model

model = pickle.loads(model\_bytes)

# Close the database connection

cur.close()

conn.close()

Once the model is loaded from the database, you can use it to make predictions as follows:

# test loaded model

y\_predict = model.predict(X\_test)

# check results

print(classification\_report(y\_test, y\_predict))

This is it, you have the model stored and loaded from the database.

**Pros of storing ML models in a database**

* 1Storing ML models in a database provides a centralized storage location that can be easily accessed by multiple applications and users.
* 2Since most organizations already have databases in place, integrating ML models into the existing infrastructure becomes easier.
* 3Databases are optimized for data retrieval, which means that retrieving the ML models is faster and more efficient.
* 4Databases are designed to provide robust security features such as authentication, authorization, and encryption. This ensures that the stored ML models are secure.

**Cons of storing ML models in a database**

* 1Databases are designed for storing structured data and are not optimized for storing unstructured data such as ML models. As a result, there may be limitations in terms of model size, file formats, and other aspects of ML models that cannot be accommodated by databases.
* 2Storing ML models in a database can be complex and requires expertise in both database management and machine learning.
* 3 If the ML models are large, storing them in a database may lead to scalability issues. Additionally, the retrieval of large models may impact the performance of the database.

While pickle, joblib, and JSON are common ways to save machine learning models, they have limitations when it comes to versioning, sharing, and managing machine learning models. Here ML model registries come to the rescue and resolve all the issues faced by the alternatives.

Next, you will see how saving ML models in the model registry can help you achieve reproducibility and reusability.

**Storing ML models in model registry**

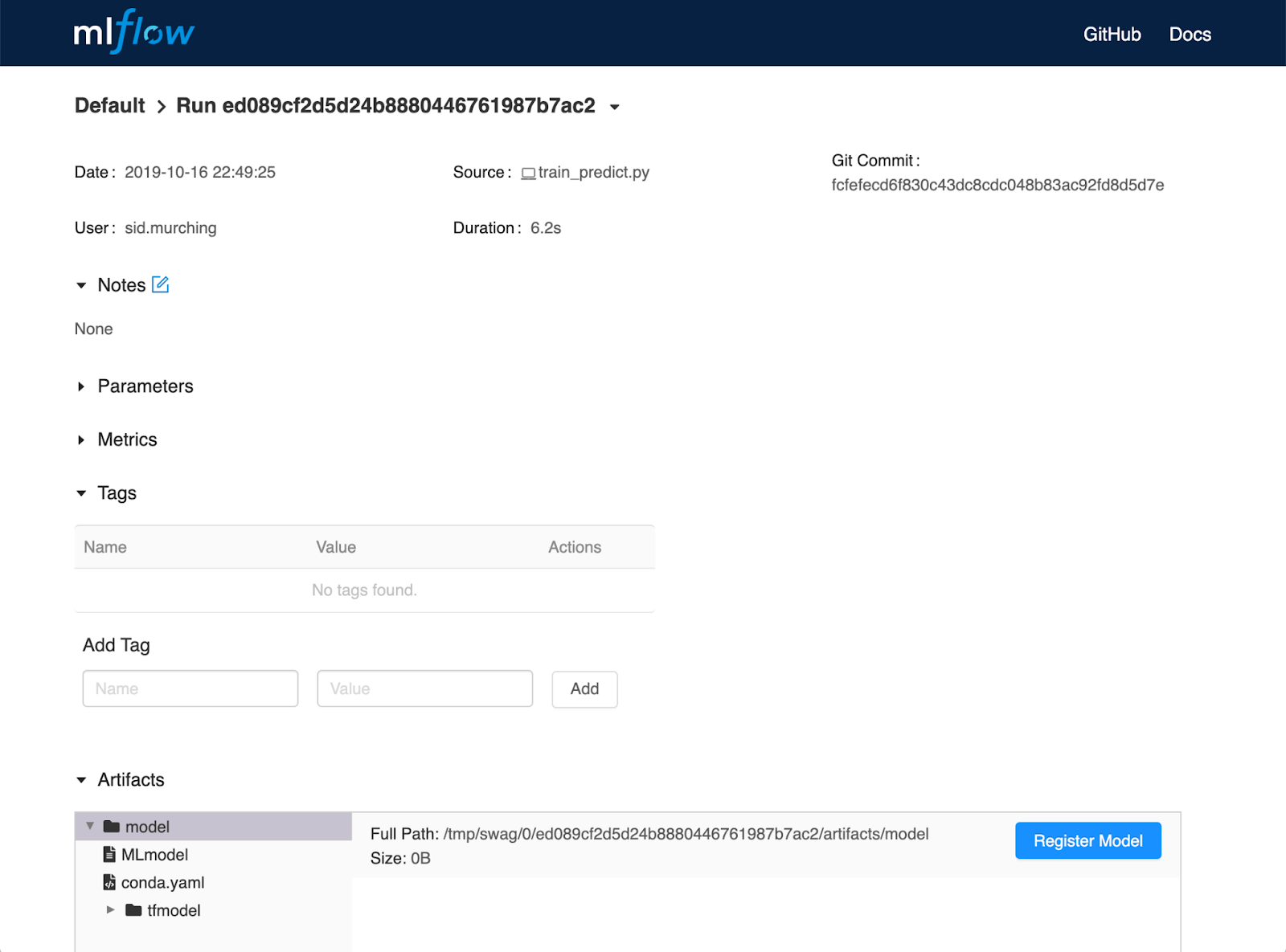
* A[model registry](https://neptune.ai/blog/ml-model-registry) is a central repository that can store, version, and manage machine learning models.
* It typically includes features like[model versioning](https://neptune.ai/blog/version-control-for-ml-models), metadata control, comparing model runs, etc.
* When working on any ML or DL projects, you can save and retrieve the models and their metadata from the model registry anytime you want.
* Above all, model registries enable high collaboration among team members.

There are various options for the model registry, such as MLflow or Kubeflow. You can also use tools like neptune.ai – even though it’s an experiment tracker, it covers model registry and model versionins capabilities to a great extent. Although all these platforms have unique features on their own, it is rather wise to choose a registry that can provide you with a comprehensive set of features.

**Storing models with MLflow**

MLflow is an open-source platform for managing the end-to-end machine learning lifecycle. It includes a model registry component that allows you to centrally manage models.

You can [register a model with MLflow either in the UI or programmatically](https://mlflow.org/docs/latest/model-registry.html).

Registering a model via UI in MLflow | [Source](https://mlflow.org/docs/latest/model-registry.html#concepts)

Once registered, you can:

* Version your models,
* Transition models through stages (e.g., Staging, Production),
* Add descriptions and tags,
* Compare model versions,
* Fetch registered models from the model registry.

**Storing models with Neptune**

[Neptune](https://neptune.ai/)is an experiment tracker designed with a strong focus on collaboration and scalability. It lets you monitor months-long model training, track massive amounts of data, and compare thousands of metrics in the blink of an eye.

You can [log, store, and organize your model metadata](https://docs.neptune.ai/log_metadata) with Neptune’s flexible Python API. To log the model metadata, use the run object. Depending on your setup, you can separate the model and training metadata by creating multiple runs or log everything together.

Full screen preview

A list of different model versions and associated metadata tracked in neptune.ai

With Neptune, you can:

* Track models and model versions, along with the associated metadata.
* Filter, sort, and compare the versioned data easily.
* Manage model stages using tags.
* Query and download any stored model files and metadata.

**Pros of storing models with model registry**

* 1A centralized location for managing, storing, and version-controlling machine learning models.
* 2Metadata regarding models, such as their version, performance metrics, etc. are frequently included in model registries, making it simpler to follow changes and comprehend the model’s past.
* 3Model registries allow team members to collaborate on models and share their work easily.
* 4Some model registries provide automated deployment options, which can simplify the process of deploying models to production environments.
* 5Model registries often provide security features such as access control, encryption, and authentication, ensuring that models are kept secure and only accessible to authorized users.

**Cons of storing models with model registry**

* 1A paid subscription is necessary for some model registries, which raises the cost of machine learning programs.
* 2Model registries often have a learning curve, and it may take time to get up to speed with their functionality and features.
* 3Using a model registry may require integrating with other tools and systems, which can create additional dependencies.

You have now seen different ways of saving an ML model (model registry being the most optimal one), this is time to check some ways to save the Deep Learning (DL) based models.

**Best practices**

In this section, you will see some of the best practices for saving the ML and DL models.

* **Ensure Library Versions:** Using different library versions for saving and loading the models may create compatibility issues as there could be some structural changes with the library update. You must ensure that library versions while loading the machine learning models should be the same as the library versions used to save the model.
* **Ensure Python Versions:** It is a good practice to use the same Python version across all stages of your ML pipeline development. Sometimes changes in the Python version can create execution issues, for example, TensorflowV1 is supported up till Python 3.7, and if you try to use it with later versions, you will face the errors.
* **Save Both Model Architecture and Weights:** In the case of DL-based models, if you save only model weight but not architecture, then you can not reconstruct the model. Saving the model architecture along with the trained weights ensures that the model can be fully reconstructed and used later on.
* **Document the Model:** The goal, inputs, outputs, and anticipated performance of the model should be documented. This can aid others in understanding the capabilities and constraints of the model.
* **Use Model Registry:** Use a model registry like neptune.ai to keep track of models, their versions, and metadata and to collaborate with team members.
* **Keep the Saved Model Secure:** Keep the saved model secure by encrypting it or storing it in a secure location, especially if it contains sensitive data.

**Conclusions**

In conclusion, saving machine learning models is an important step in the development process, as it allows you to reuse and share your models with others. There are several ways to save machine learning models, each with its own advantages and disadvantages. Some popular methods include using pickle, Joblib, JSON, TensorFlow save, and PyTorch save.

It is important to choose the appropriate file format for your specific use case and to follow best practices for saving and documenting models, such as version control, ensuring language and library versions, and testing the saved model. By following the practices discussed in this article, you can ensure that your machine-learning models are saved correctly, are easy to reuse and deploy, and can be effectively shared with others.

[neptune resource](https://neptune.ai/blog/saving-trained-model-in-python)

[Medium resc](https://medium.com/@rohanmistry231/beyond-pkl-the-complete-guide-to-saving-machine-learning-models-ddefaeb3e53e)

[geek fot geek resc](https://www.geeksforgeeks.org/machine-learning/saving-a-machine-learning-model/)

**Hyperparameter Tuning Techniques**

## **What is hyperparameter tuning?**

Hyperparameter tuning is the practice of identifying and selecting the optimal hyperparameters for use in training a machine learning model. When performed correctly, hyperparameter tuning minimizes the loss function of a machine learning model, which means that the model performance is trained to be as accurate as possible.

Hyperparameter tuning is an experimental practice, with each iteration testing different hyperparameter values until the best ones are identified. This process is critical to the performance of the model as hyperparameters govern its learning process. The amount of neurons in a neural network, a generative AI model’s learning rate and a support vector machine’s kernel size are all examples of hyperparameters.

Good hyperparameter tuning means a stronger performance overall from the machine learning model according to the metrics for its intended task. This is why hyperparameter tuning is also known as hyperparameter optimization.

## **What are hyperparameters?**

Hyperparameters are configuration variables that data scientists set ahead of time to manage the training process of a [machine learning](https://www.ibm.com/think/topics/machine-learning) model. Generative AI and other probabilistic models apply their learnings from training data to predict the most likely outcome for a task. Finding the right combination of hyperparameters is essential to coaxing the best performance from both [supervised learning](https://www.ibm.com/think/topics/supervised-learning) and [unsupervised learning](https://www.ibm.com/think/topics/unsupervised-learning) models.

### **Regularization hyperparameters**

[Regularization](https://www.ibm.com/think/topics/regularization) hyperparameters control the capacity or flexibility of the model, which is how much leeway it has when interpreting data. Apply too light a hand, and the model won’t be able to get specific enough to make good predictions. Go too far, and the model will suffer from [overfitting](https://www.ibm.com/think/topics/overfitting): when it overadapts to its training data and ends up being too niche for real-world use.

### **Hyperparameters vs. model parameters**

The primary difference between hyperparameters and model parameters in [data science](https://www.ibm.com/think/topics/data-science) is that while models learn or estimate parameters from the training datasets they ingest, data scientists define the hyperparameters for the model’s algorithm before the training process begins. Models continue to update parameters as they work, whereas the optimal values of a model’s hyperparameters are identified and set ahead of time.

## **Why is hyperparameter tuning important?**

Hyperparameter tuning is important because it lays the groundwork for a model’s structure, training efficiency and performance. Optimal hyperparameter configurations lead to strong model performance in the real world. [Large language model operations (LLMOps) stress](https://www.ibm.com/think/topics/llmops) the efficiency aspect of good tuning, with an emphasis on minimizing computational power requirements.

### **Bias and variance**

The goal of hyperparameter tuning is to balance the bias-variance tradeoff. Bias is the divergence between a model’s predictions and reality. Models that are undertuned, or underfitted, fail to discern key relationships between datapoints and are unable to draw the required conclusions needed for accurate performance.

Variance is the sensitivity of a model to new data. A reliable model should deliver consistent results when migrating from its training data to other datasets. However, models with high levels of variance are too complex—they are overfitted to their original training datasets and struggle to accommodate new data.

Models with low bias are accurate, while models with low variance are consistent. Good hyperparameter tuning optimizes for both to create the best model for the job while also maximizing computational resource efficiency during training.

## **Hyperparameter examples**

Each machine learning algorithm favors its own respective set of hyperparameters, and it’s not necessary to maximize them in all cases. Sometimes, a more conservative approach when tuning hyperparameters will lead to better performance.

### **Neural network hyperparameters**

[Neural networks](https://www.ibm.com/think/topics/neural-networks) take inspiration from the human brain and are composed of interconnected nodes that send signals to one another. In general, here are some of the most common hyperparameters for neural network model training:

#### **Learning rate**

Learning rate sets the speed at which a model adjusts its parameters in each iteration. These adjustments are known as steps. A high learning rate means that a model will adjust more quickly, but at the risk of unstable performance and data drift. Meanwhile, while a low learning rate is more time-consuming and requires more data, it also makes it more likely that data scientists will pinpoint a model’s minimum loss. [Gradient descent](https://www.ibm.com/think/topics/gradient-descent) optimization is an example of a training metric requiring a set learning rate.

#### **Learning rate decay**

Learning rate decay sets the rate at which the learning rate of a network drops over time, allowing the model to learn more quickly. An algorithm's training progression from its initial activation to ideal performance is known as convergence.

#### **Batch size**

Batch size sets the amount of samples the model will compute before updating its parameters. It has a significant effect on both compute efficiency and accuracy of the training process. On its own, a higher batch size weakens overall performance, but adjusting the learning rate along with batch size can mitigate this loss.

#### **Number of hidden layers**

The number of hidden layers in a neural network determines its depth, which affects its complexity and learning ability. Fewer layers make for a simpler and faster model, but more layers—such as with [deep learning](https://www.ibm.com/think/topics/deep-learning) networks—lead to better classification of input data. Identifying the optimal hyperparameter value here from all the possible combinations is all about a tradeoff between speed with accuracy.

#### **Number of nodes or neurons per layer**

The number of nodes or neurons per layer sets the width of the model. The more nodes or neurons per layer, the greater the breadth of the model and the better able it is to depict complex relationships between data points.

#### **Momentum**

Momentum is the degree to which models update parameters in the same direction as previous iterations, rather than reversing course. Most data scientists begin with a lower hyperparameter value for momentum and then tweak upwards as needed to keep the model on course as it takes in training data.

#### **Epochs**

Epochs is a hyperparameter that sets the amount of times that a model is exposed to its entire training dataset during the training process. Greater exposure can lead to improved performance but runs the risk of overfitting.

#### **Activation function**

Activation function introduces nonlinearity into a model, allowing it to handle more complex datasets. Nonlinear models can generalize and adapt to a greater variety of data.

### **SVM hyperparameters**

Support vector machine (SVM) is a machine learning algorithm specializing in data classification, regression and outlier detection. It has its own essential hyperparameters:

#### **SVM hyperparameter: C**

C is the ratio between the acceptable margin of error and the resulting number of errors when a model acts as a data classifier. A lower C value establishes a smooth decision boundary with a higher error tolerance and more generic performance, but with a risk of incorrect data classification. Meanwhile, a high C value creates a neat decision boundary for more accurate training results but with potential overfitting.

#### **SVM hyperparameter: kernel**

Kernel is a function that establishes the nature of the relationships between data points and separates them into groups accordingly. Depending on the kernel used, data points will show different relationships, which can strongly affect the overall SVM model performance. Linear, polynomial, radial basis function (RBF), and sigmoid are a few of the most commonly used kernels. Linear kernels are simpler and best for easily separable data, while nonlinear kernels are better for more complex datasets.

#### **SVM hyperparameter: gamma**

Gamma sets the level of influence support vectors have on the decision boundary. Support vectors are the data points closest to the hyperplane: the border between groups of data. Higher values pull strong influence from nearby vectors, while lower values limit the influence from more distant ones. Setting too high a gamma value can cause overfitting, while too low a value can muddy the decision boundary.

### **XGBoost hyperparameters**

XGBoost stands for “extreme gradient [boosting](https://www.ibm.com/topics/boosting)” and is an ensemble algorithm that blends the predictions of multiple weaker models, known as decision trees, for a more accurate result. Gradient-boosted algorithms tend to outperform [random forest](https://www.ibm.com/think/topics/random-forest) models, another type of ensemble algorithm comprising multiple decision trees.

The most important hyperparameters for [XGBoost](https://www.ibm.com/think/topics/xgboost) are:

#### **learning\_rate**

learning\_rate is similar to the learning rate hyperparameter used by neural networks. This function controls the level of correction made during each round of training. Potential values range from 0 to 1, with 0.3 as the default.

#### **n\_estimators**

n\_estimators sets the number of trees in the model. This hyperparameter is known as num\_boost\_rounds in the original XGBoost, whereas the popular Python API scikit-learn introduced the name n\_estimators.

#### **max\_depth**

max\_depth determines the architecture of the decision tree, setting the maximum amount of nodes from the tree to each leaf—the final classifier. More nodes lead to more nuanced data classification, while smaller trees easily avoid overfitting.

#### **min\_child\_weight**

min\_child\_weight is the minimum weight—the importance of a given class to the overall model training process—needed to spawn a new tree. Lower minimum weights create more trees but with potential overfitting, while larger weights reduce complexity by requiring more data to split trees.

#### **subsample**

subsample sets the percentage of data samples used during each training round, and colsample\_bytree fixes the percentage of features to use in tree construction.

## **How does hyperparameter tuning work?**

Hyperparameter tuning centers around the objective function, which analyzes a group, or tuple, of hyperparameters and calculates the projected loss. Optimal hyperparameter tuning minimizes loss according to the chosen metrics. The results are confirmed via cross-validation, which measures how closely they generalize to other datasets outside the specific training instance.

### **Hyperparameter tuning methods**

Data scientists have a variety of hyperparameter tuning methods at their disposal, each with its respective strengths and weaknesses. Hyperparamter tuning can be performed manually or automated as part of an [AutoML (automated machine learning)](https://www.ibm.com/think/topics/automl) strategy.

#### **Grid search**

Grid search is a comprehensive and exhaustive hyperparameter tuning method. After data scientists establish every possible value for each hyperparameter, a grid search constructs models for every possible configuration of those discrete hyperparameter values. These models are each evaluated for performance and compared against each other, with the best model ultimately selected for training.

In this way, grid search is similar to brute-forcing a PIN by inputting every potential combination of numbers until the correct sequence is discovered. While it does enable data scientists to consider all possible configurations in the hyperparameter space, grid search is inefficient and computationally resource-intensive.

#### **Randomized search**

Random search differs from grid search in that data scientists provide statistical distributions instead of discrete values for each hyperparameter. A randomized search pulls samples from each range and constructs models for each combination. Over the course of several iterations, the models are weighed against one other until the best model is found.

Randomized search is preferable to grid search in situations where the hyperparameter search space contains large distributions—it would simply require too much effort to test each discrete value. Random search algorithms can return results comparable to grid search in considerably less time, though it isn’t guaranteed to discover the most optimal hyperparameter configuration.

#### **Bayesian optimization**

Bayesian optimization is a sequential model-based optimization (SMBO) algorithm in which each iteration of testing improves the sampling method of the next. Both grid and random searches can be performed concurrently, but each test is performed in isolation—data scientists can’t use what they’ve learned to inform subsequent tests.

Based on prior tests, Bayesian optimization probabilistically selects a new set of hyperparameter values that is likely to deliver better results. The probabilistic model is referred to as a surrogate of the original objective function. Because surrogate models are compute-efficient, they’re usually updated and improved each time the objective function is executed.

The better the surrogate gets at predicting optimal hyperparameters, the faster the process becomes, with fewer objective function tests required. This makes Bayesian optimization far more efficient than the other methods, since no time is wasted on unsuitable combinations of hyperparameter values.

The process of statistically determining the relationship between an outcome—in this case, the best model performance—and a set of variables is known as regression analysis. Gaussian processes are one such SMBO popular with data scientists.

#### **Hyperband**

Introduced in 2016, [Hyperband](https://arxiv.org/abs/1603.06560) is designed to improve on random search by truncating the use of training configurations that fail to deliver strong results while allocating more resources to positive configurations.

This “early stopping” is achieved through successive halving, a process that whittles down the pool of configurations by removing the worst-performing half after each round of training. The top 50% of each batch is carried into the next iteration until one optimal hyperparameter configuration remains.

[IBM resc](https://www.ibm.com/think/topics/hyperparameter-tuning)

[medium resc](https://medium.com/pythoneers/hyperparameter-tuning-in-data-science-ad1a03d830b9)

[geekforgeeks resc](https://www.geeksforgeeks.org/machine-learning/hyperparameter-tuning/)

## **1. Dataset Considerations for Each Model**

The algorithm’s nature affects which datasets it works well with.

| **Model** | **Works Best When** | **May Struggle With** |
| --- | --- | --- |
| **SVM** | Low-to-medium dimensional data (or high-dimensional but with proper regularization), clean separation between classes, not too many outliers | Very large datasets (training can be slow), highly imbalanced classes, lots of noise |
| **Decision Tree** | Small-to-medium datasets, nonlinear decision boundaries, mixed feature types (numeric + categorical) | Overfitting on noisy data if not pruned, biased towards features with many categories |
| **Random Forest** | Medium-to-large datasets, high-dimensional data, robust to noise and overfitting, good for mixed data types | Can still overfit on small datasets, less interpretable, slower with extremely large feature spaces |

## **2. Data Preparation Checklist**

### **A. Understand the Problem**

* **Type of classification:** binary, multiclass, or multilabel?
* **Evaluation metric priority:** Accuracy? F1? AUC? (Important for imbalanced data)
* **Data size & feature count:** SVM might need dimensionality reduction for very large datasets.

### **B. Cleaning**

* **Missing values:**
  + SVM: Needs imputation (cannot handle NaN directly).
  + Decision Trees / Random Forest: Can handle some missingness but better to impute for consistency.
* **Duplicate rows:** Remove unless justified.
* **Outliers:**
  + SVM is sensitive → detect and consider removing or scaling effects.
  + Trees are robust to outliers.

### **C. Feature Encoding**

* **Categorical features:**
  + SVM: One-hot encoding (scaling after encoding).
  + Decision Tree / Random Forest: Can handle ordinal or one-hot, but beware of label encoding for non-ordinal categories (splits may be biased).
* **Text features:** Vectorize (TF-IDF, CountVectorizer).

### **D. Scaling & Transformation**

| **Model** | **Scaling Needed?** | **Notes** |
| --- | --- | --- |
| SVM | ✅ **Required** | StandardScaler or MinMaxScaler to avoid bias from feature magnitude |
| Decision Tree | ❌ Not required | Scaling doesn’t affect splits |
| Random Forest | ❌ Not required | Scaling doesn’t affect splits |

Other transformations:

* **Log transform** skewed features for SVM.
* **Polynomial features** can help SVM capture non-linear boundaries (but increases dimensionality).

### **E. Class Imbalance Handling**

If your target classes are imbalanced:

* **Data-level:** Oversampling (SMOTE), undersampling.
* **Model-level:** Use class\_weight='balanced' in SVM and tree-based models.

### **F. Feature Selection / Dimensionality Reduction**

* **SVM:** Use PCA or SelectKBest to speed up and avoid overfitting in high dimensions.
* **Trees / RF:** Naturally handle feature selection, but removing irrelevant features can still help performance.

## **3. Training Considerations per Model**

### **SVM**

* Choose kernel (linear, rbf, poly) based on data patterns.
* Tune C (regularization strength) and gamma (kernel influence).
* Scaling is **critical**.

### **Decision Tree**

* Prune via max\_depth, min\_samples\_split, min\_samples\_leaf to avoid overfitting.
* Handle categorical data carefully.
* Check feature\_importances\_ for insight.

### **Random Forest**

* Tune n\_estimators (more trees → slower but better stability).
* Tune max\_features for diversity in trees.
* Check OOB (out-of-bag) score for quick validation.

## **4. Validation & Testing**

* Always use **train-test split** or cross-validation to avoid overfitting.
* For imbalanced datasets, use **StratifiedKFold** to maintain class ratios.
* Monitor both training and validation metrics — big gaps = overfitting.

## **5. Common Pipelines**

Here’s a mental model for each:

**SVM Pipeline**

Data Cleaning → Encode categorical → Scale (StandardScaler) → Feature selection/PCA → SVM (tuned)

**Decision Tree Pipeline**

Data Cleaning → Encode categorical (ordinal or one-hot) → Tree with pruning parameters

**Random Forest Pipeline**

Data Cleaning → Encode categorical → Optional scaling → Random Forest (tuned n\_estimators, max\_depth, max\_features)

## **6. When to Pick Which**

* **SVM:** Small/medium datasets with clean, scaled, and potentially high-dimensional but not huge data; good when boundary is complex but margin matters.
* **Decision Tree:** Quick baseline, interpretable model, can handle mixed data.
* **Random Forest:** Strong default choice when you want robustness, less tuning hassle, and good generalization.

svm = SVC(kernel="rbf", C=1.0, gamma="scale", probability=True)

## **1.** SVC **Overview**

SVC is scikit-learn’s **Support Vector Classification** implementation.  
It creates a hyperplane (or set of hyperplanes) in a high-dimensional space to separate classes with the largest possible margin.

## **2. Parameter-by-Parameter Breakdown**

### **A.** kernel

* **What it does:**  
  Defines the transformation function used to map your input data into a higher-dimensional space where it might be more separable.
* **Your value:** "rbf" (**Radial Basis Function**, also called Gaussian kernel)
  + Good for non-linear decision boundaries.
  + Works well when the relationship between features and classes is complex and not just linear.
* **Alternatives:**

| **Kernel** | **When to Use** | **Equation Notes** |
| --- | --- | --- |
| "linear" | Data is linearly separable or has many features | Equivalent to LinearSVC but slower for large datasets |
| "poly" | When you want polynomial decision boundaries | Controlled by degree |
| "sigmoid" | Rarely used; can behave like a neural network’s activation | Needs careful tuning |
| Callable | Custom kernel function | Advanced use cases |

### **B.** C

* **What it does:**  
  Regularization parameter — controls the trade-off between **maximizing the margin** and **minimizing classification error**.
* **Your value:** 1.0 (default)
  + **Higher C** → Less regularization → Model tries to classify all points correctly (can overfit).
  + **Lower C** → More regularization → Larger margin but allows more misclassifications (can underfit).
* **Typical tuning range:** 0.001 to 1000 (log scale search).

### **C.** gamma

* **What it does:**  
  Controls how far the influence of a single training example reaches in the **RBF**, **poly**, and **sigmoid** kernels.
* **Your value:** "scale" (default in sklearn ≥ 0.22)
  + "scale" means:

γ=1nfeatures×Var(X)\gamma = \frac{1}{n\_{\text{features}} \times \text{Var}(X)}

* + "auto" means:

γ=1nfeatures\gamma = \frac{1}{n\_{\text{features}}}

* **Interpretation for RBF:**
  + **Small gamma** → Large radius of influence → smoother decision boundary.
  + **Large gamma** → Small radius of influence → more complex boundary (risk of overfitting).
* **Typical tuning range:** 1e-4 to 10 (log scale search).

### **D.** probability

* **What it does:**  
  Enables probability estimates using **Platt scaling** (extra cross-validation internally).
* **Your value:** True
  + This means you can call .predict\_proba(X) to get probability scores.
  + **Downside:** Increases training time significantly.
  + If you only need class labels, keep probability=False.

## **3. Other Useful Parameters in** SVC

| **Parameter** | **Purpose** | **Notes** |
| --- | --- | --- |
| degree | Degree for "poly" kernel | Ignored if kernel is not "poly" |
| coef0 | Controls influence of higher-order vs lower-order terms in "poly" or "sigmoid" kernels | Needs tuning for those kernels |
| shrinking | Use shrinking heuristic for faster convergence | Default True, set False for large datasets if needed |
| tol | Stopping criterion tolerance | Default 1e-3, lower for more precision |
| max\_iter | Limit iterations | -1 means no limit |
| class\_weight | Handle imbalanced datasets | 'balanced' or dict per class |

## **4. Intuition Summary**

* **C** → "How much do I care about correctly classifying every training point?"
  + Big C = "I care a lot" (less margin, more fit to training data)
  + Small C = "I’m okay with some mistakes" (wider margin, more generalization)
* **gamma** → "How far does one point’s influence spread?"
  + Big gamma = "I care only about nearby points" (complex boundary)
  + Small gamma = "I care about far-away points too" (smoother boundary)
* **kernel** → "What shape is my decision boundary?"
  + Linear, polynomial, circular blobs (RBF), etc.

**Overfitting in Decision Trees** happens when the model learns the training data too well, capturing noise and random fluctuations instead of just the underlying patterns. This results in **high accuracy on the training set** but **poor performance on unseen data** because the model fails to generalize.

**Why it happens:**

* Decision Trees can grow very deep, splitting until each leaf has very few samples or even just one.
* They may create overly complex rules to perfectly classify the training data.

**Prevention techniques:**

1. **Limit tree depth (max\_depth)** – Restrict how deep the tree can grow to avoid overly specific rules.
2. **Minimum samples per split/leaf (min\_samples\_split, min\_samples\_leaf)** – Require a minimum number of samples before making a split or creating a leaf.
3. **Limit number of features (max\_features)** – Use only a subset of features for splitting to reduce complexity.
4. **Pruning** – Remove branches that have little impact on prediction accuracy (pre-pruning stops growth early, post-pruning removes after full growth).
5. **Use ensemble methods** – Techniques like **Random Forest** average predictions from multiple trees, reducing the risk of overfitting.

**Out-of-Bag (OOB) Error** is a built-in validation method used in **Random Forests** to estimate model performance without needing a separate validation set.

**Concept:**  
In Random Forest, each tree is trained on a **bootstrap sample** (random sampling *with replacement*) of the training data. On average, about **63%** of the samples are used to train each tree, and the remaining **37%** are **“out-of-bag” samples**—data points not included in that tree’s training. These OOB samples can be used to test that tree’s performance.

**Calculation:**

1. For each data point, collect predictions from all trees where that point was OOB (not used in training for that tree).
2. Compare these aggregated predictions to the true labels to compute accuracy or error.
3. The average error across all points is the **OOB error**.

**Formula for OOB Error Rate:**

​ OOB Error= Number of incorrect OOB predictions​ / Total number of samples

OOB error provides an **unbiased performance estimate** while using all the data for both training and validation.

You choose **F1-score over Accuracy** when dealing with **imbalanced datasets** or when **false positives and false negatives are more important than overall correct predictions**.

**Why:**

* **Accuracy** can be misleading if one class dominates. For example, if 95% of samples belong to one class, predicting everything as that class gives 95% accuracy but 0% usefulness for detecting the minority class.
* **F1-score** is the harmonic mean of **precision** and **recall**, giving a balanced measure that considers both false positives and false negatives.
* It’s especially useful when **both precision and recall matter** and you want a single metric to optimize.

**When to use:**

* Fraud detection
* Medical diagnosis
* Spam filtering
* Any case with imbalanced classes where missing a minority-class prediction is costly

**Precision** is important in model evaluation because it measures the proportion of correctly predicted positive cases out of all predicted positives. High precision means that when the model predicts a positive, it is usually correct, which is crucial in scenarios where **false positives** carry a high cost. For example, in spam detection, precision ensures that legitimate emails are not wrongly marked as spam, and in medical diagnosis, it reduces the risk of wrongly diagnosing healthy patients as having a disease.

**Recall** is important in model evaluation because it measures the proportion of actual positive cases that the model correctly identifies. High recall ensures that most of the true positives are captured, which is crucial in scenarios where **missing positive cases** has serious consequences. For example, in disease screening, recall ensures that most patients with the disease are detected, and in fraud detection, it helps catch as many fraudulent cases as possible.

Common formats for saving ML models include:

1. **Pickle (.pkl or .pickle)** – Python’s built-in serialization format, used to save almost any Python object, including models.
2. **Joblib (.joblib)** – Optimized for storing large NumPy arrays efficiently, often used for scikit-learn models.
3. **HDF5 (.h5)** – Common for saving Keras/TensorFlow models; stores model architecture, weights, and optimizer state.
4. **ONNX (.onnx)** – Open Neural Network Exchange format for interoperability between different ML frameworks.
5. **SavedModel (TensorFlow)** – Directory-based format containing model architecture, weights, and assets.
6. **PMML** – Predictive Model Markup Language, an XML-based standard for sharing models across platforms.

Yes ✅ — you can absolutely use **SVM**, **DecisionTree**, or almost any scikit-learn estimator with **GridSearchCV** or **RandomizedSearchCV**.

Both search methods are **model-agnostic** — they don’t care which model you use, as long as:

1. The model follows scikit-learn’s **estimator API** (fit/predict methods).
2. You use the **correct parameter names** for that specific model.

### **Examples**

#### 1️⃣ Using **DecisionTreeClassifier** with GridSearchCV

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import GridSearchCV

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

# Load dataset

X, y = load\_iris(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=42)

# Model

dt = DecisionTreeClassifier(random\_state=42)

# Parameter grid

param\_grid = {

'max\_depth': [None, 3, 5, 10],

'criterion': ['gini', 'entropy']

}

# Grid search

grid = GridSearchCV(dt, param\_grid, cv=5)

grid.fit(X\_train, y\_train)

print("Best Params:", grid.best\_params\_)

print("Best Score:", grid.best\_score\_)

#### 2️⃣ Using **SVC** (Support Vector Machine) with RandomizedSearchCV

from sklearn.svm import SVC

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from scipy.stats import uniform

# Load dataset

X, y = load\_iris(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=42)

# Model

svc = SVC()

# Parameter distribution

param\_dist = {

'C': uniform(0.1, 10),

'kernel': ['linear', 'rbf', 'poly'],

'gamma': ['scale', 'auto']

}

# Randomized search

rand\_search = RandomizedSearchCV(svc, param\_dist, n\_iter=10, cv=5, random\_state=42)

rand\_search.fit(X\_train, y\_train)

print("Best Params:", rand\_search.best\_params\_)

print("Best Score:", rand\_search.best\_score\_)

### **Key points when switching models**

* Always check the **model’s documentation** for valid parameter names:
  + [DecisionTreeClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)
  + [SVC](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html)
* The parameters in your grid must exactly match the names in the docs.
* The dataset type matters — SVMs can be slower on large datasets, while trees and forests scale better.

If you want, I can make **one combined script** where you can easily switch between RandomForest, DecisionTree, and SVM with GridSearchCV or RandomizedSearchCV. That way, you can experiment with all three quickly.

## **1️⃣ DecisionTreeClassifier**

[Docs](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

| **Parameter** | **Description** | **Common Values** |
| --- | --- | --- |
| criterion | Function to measure split quality | ['gini', 'entropy', 'log\_loss'] |
| splitter | Strategy to choose split | ['best', 'random'] |
| max\_depth | Maximum depth of tree | [None, 5, 10, 20, 50] |
| min\_samples\_split | Min samples to split a node | [2, 5, 10, 20] |
| min\_samples\_leaf | Min samples at a leaf | [1, 2, 4, 6] |
| max\_features | Max features considered per split | [None, 'sqrt', 'log2'] |
| max\_leaf\_nodes | Max number of leaf nodes | [None, 10, 20, 50] |
| min\_impurity\_decrease | Min impurity decrease to split | [0.0, 0.01, 0.05] |

## **2️⃣ RandomForestClassifier**

[Docs](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html)

| **Parameter** | **Description** | **Common Values** |
| --- | --- | --- |
| n\_estimators | Number of trees | [50, 100, 200, 500] |
| criterion | Split quality metric | ['gini', 'entropy', 'log\_loss'] |
| max\_depth | Max tree depth | [None, 5, 10, 20, 50] |
| min\_samples\_split | Min samples to split | [2, 5, 10, 20] |
| min\_samples\_leaf | Min samples at a leaf | [1, 2, 4, 6] |
| max\_features | Max features per split | ['sqrt', 'log2', None] |
| bootstrap | Whether bootstrap samples are used | [True, False] |
| max\_leaf\_nodes | Max leaf nodes | [None, 10, 20, 50] |
| min\_impurity\_decrease | Min impurity decrease | [0.0, 0.01, 0.05] |

## **3️⃣ SVC (Support Vector Classifier)**

[Docs](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html)

| **Parameter** | **Description** | **Common Values** |
| --- | --- | --- |
| C | Regularization strength | [0.1, 1, 10, 100] |
| kernel | Kernel type | ['linear', 'rbf', 'poly', 'sigmoid'] |
| degree | Degree for polynomial kernel | [2, 3, 4, 5] (only for *poly*) |
| gamma | Kernel coefficient | ['scale', 'auto', 0.1, 0.01, 1] |
| coef0 | Independent term in kernel | [0.0, 0.1, 0.5, 1.0] (for *poly* & *sigmoid*) |
| shrinking | Use shrinking heuristic | [True, False] |
| probability | Enable probability estimates | [True, False] |
| class\_weight | Class balance handling | [None, 'balanced'] |

💡 **Tips for use in GridSearchCV / RandomizedSearchCV**

* Don’t throw all possible values at once — start **small** to avoid huge computation time.
* For large datasets, **RandomizedSearchCV** is better with a **wide but limited range**.
* Always check **model documentation** for your scikit-learn version — parameter options can change.