Lecture B

Machine Learning Methods (Selection)

K-Nearest Neighbors
K-Means
Support Vector Machines
Decision Trees

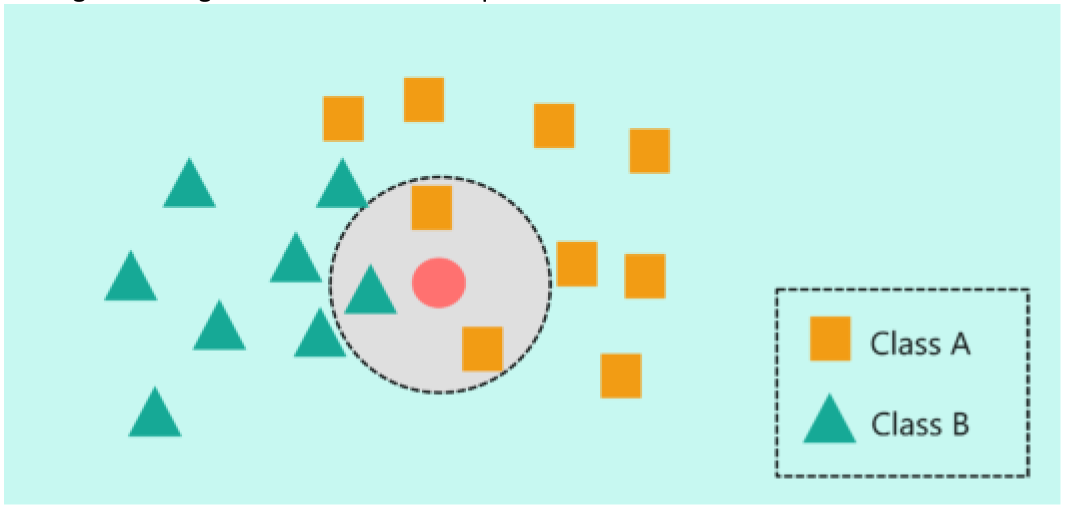
K-Nearest Neighbors (KNN) is a popular supervised machine learning algorithm used for classification and regression tasks. It is a non-parametric algorithm, meaning it does not make any assumptions about the underlying data distribution. KNN is known as an instance-based learning algorithm because it makes predictions based on the similarity of new instances to existing instances in the training dataset.

To implement KNN, we start with the *Training Phase*: During the training phase, KNN stores the entire training dataset, which consists of labeled instances. Each instance is represented by a set of features or attributes, and it is associated with a class label (in the case of classification) or a target value (in the case of regression).

Choosing the Value of K: The algorithm requires the selection of a parameter called "K," which determines the number of nearest neighbors to consider for making predictions. The value of K is typically chosen based on the characteristics of the dataset and is often determined through experimentation or using cross-validation techniques.

Prediction Phase (Classification): Given a new, unlabeled instance, the KNN algorithm searches for the K nearest neighbors in the training dataset. The distance metric used, such as Euclidean distance or Manhattan distance, measures the similarity between instances based on their feature values. The K nearest neighbors are the instances with the smallest distances to the new instance.

Voting (Classification): Once the K nearest neighbors are identified, the algorithm assigns the class label to the new instance by majority voting. In other words, the class label that occurs most frequently among the K neighbors is chosen as the predicted class label for the new instance.



Prediction Phase (Regression): In regression tasks, instead of voting, KNN takes an average (or weighted average) of the target values of the K nearest neighbors and assigns it as the predicted target value for the new instance. The averaging process provides a continuous prediction rather than a discrete class label.

Model Evaluation: After making predictions for all instances in the test dataset, the performance of the KNN algorithm is evaluated using appropriate evaluation metrics such as accuracy, precision, recall, F1-score (for classification), or mean squared error (for regression).

Some important considerations when working with KNN:

- KNN is a lazy learning algorithm because it does not require a training phase involving model parameter estimation. It makes predictions directly based on the training dataset.
- The choice of the distance metric and the value of K significantly impact the algorithm's performance. Different distance metrics may be more appropriate for different types of data.
- The computational complexity of the KNN algorithm can be high, particularly when dealing with large datasets. Efficient data structures like KD-trees or ball trees can be used to speed up the nearest neighbor search.
- KNN can be sensitive to irrelevant and noisy features. Feature selection and dimensionality reduction techniques may be employed to improve the algorithm's performance.
- The KNN algorithm does not provide insights into the underlying relationships or decision boundaries in the data. It is a simple yet effective algorithm for making predictions based on local similarities.

Overall, KNN is a versatile and widely used algorithm that is suitable for various classification and regression tasks, especially when the underlying data distribution is not well-defined or when the dataset is relatively small.

Implementing K-Nearest Neighbors (KNN) in R is relatively straightforward using the class package, which provides the knn function. Here's a simple example using a hypothetical dataset:

```
# Install and load the 'class' package
install.packages("class")
library(class)

# Create a hypothetical dataset. Skip this step if you have actual data
set.seed(123) #for reproducibility
data <- data.frame(
  Feature1 = rnorm(50),
  Feature2 = rnorm(50),
  Class = c(rep("A", 25), rep("B", 25))
)

# Split the dataset into training and testing sets
train_indices <- sample(1:nrow(data), 30)
train_data <- data[train_indices, ]
test_data <- data[-train_indices, ]
```

```
# Use KNN for classification
k <- 3  #the choice of k is important. An odd number is preferred to avoid ties which produces instability
predicted_classes <- knn(train = train_data[, c("Feature1", "Feature2")],
                  test = test_data[, c("Feature1", "Feature2")],
                  cl = train_data$Class,
                  k = k)

# Compare predicted classes to actual classes
confusion_matrix <- table(predicted_classes, test_data$Class)
print("Confusion Matrix:")
print(confusion_matrix)

# Calculate accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", accuracy))
```

Note: This is a basic example, and in a real-world scenario, you would typically perform more thorough data preprocessing, tune the hyperparameter (like k in KNN), and possibly use cross-validation for better evaluation. If the variables are not a consistent size, normalization is in order so that variables are on a similar scale. Alternative distance metrics are sometimes applied, but such alternatives typically have to be implemented by writing your own code.

K-Means is an unsupervised machine learning algorithm used for clustering, which involves grouping similar instances together based on their features. The goal of K-Means is to partition a dataset into K distinct clusters, where each instance belongs to the cluster with the nearest mean (centroid). It is one of the most widely used clustering algorithms due to its simplicity and efficiency.

How the K-Means algorithm works:

Initialization: Initially, K centroids are randomly selected from the dataset. The number of centroids, K, is a parameter that needs to be defined beforehand.

Assignment: Each instance in the dataset is assigned to the nearest centroid based on a distance metric, typically the Euclidean distance. This step is also known as the "assignment" or "expectation" step.

Update: After assigning instances to clusters, the centroids of each cluster are updated by computing the mean (average) of all instances within that cluster. This step is also referred to as the "update" or "maximization" step.

Iterations: Steps 2 and 3 are repeated iteratively until convergence. Convergence occurs when either the centroids stop moving significantly or a maximum number of iterations is reached. In each iteration, instances are reassigned to the nearest centroid, and the centroids are updated based on the reassigned instances.

Final Result: Once the algorithm converges, the final result is a set of K clusters, each represented by its centroid. Each instance in the dataset is associated with the centroid of the cluster it belongs to.

Choosing the Value of K: The value of K is an important parameter in K-Means and needs to be predefined. Selecting an appropriate value of K can be challenging and often involves using domain knowledge, visual inspection of the data, or techniques such as the elbow method or silhouette coefficient.

K-Means is an iterative algorithm that aims to minimize the within-cluster sum of squares, also known as the distortion or inertia. It tries to make instances within the same cluster as similar as possible while keeping instances from different clusters as dissimilar as possible.

The algorithm is sensitive to the initial random selection of centroids. Different initializations can lead to different results. To mitigate this, K-Means is often run multiple times with different initializations, and the best clustering result is selected based on a chosen criterion (e.g., the lowest distortion).

K-Means is computationally efficient and scales well with large datasets. However, the algorithm's complexity increases with the number of instances and features, making it less suitable for high-dimensional data.

K-Means assumes that clusters are spherical, isotropic, and have equal variances. Thus, it may not perform well on datasets with irregularly shaped or overlapping clusters.

Outliers can significantly affect the clustering results since they can disproportionately influence the centroid calculation. Preprocessing steps, such as outlier detection or data normalization, may be required to handle such situations.

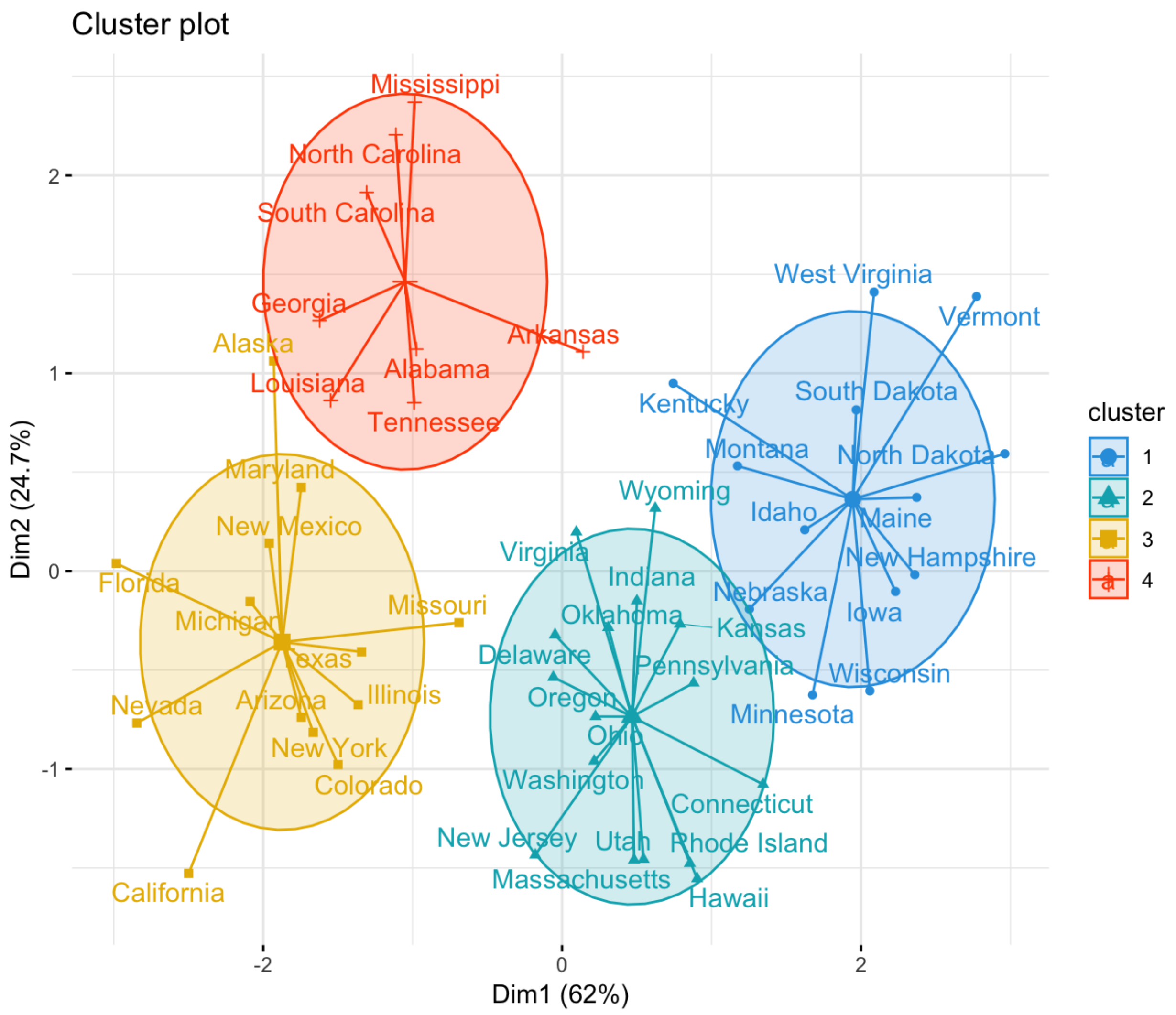
K-Means is widely used in various applications, such as customer segmentation, image compression, document clustering, and anomaly detection. Overall, K-Means is a popular and efficient algorithm for clustering unlabeled data. It provides a simple and interpretable solution for finding clusters based on the similarity of instances' feature values.

Implementing k-means clustering in R is straightforward using the kmeans function, which is part of the base R package. Here's a simple example:

```
# Generate a hypothetical dataset. Skip this step if you have a real dataset. We can also implement this
with more than two variables.
set.seed(123) #for reproducibility
data <- data.frame(
  x = rnorm(100, mean = 5),
  y = rnorm(100, mean = 5)
)

# Perform k-means clustering with k=3
k <- 3 #depends on the number of classes you have (or want)
kmeans_result <- kmeans(data, centers = k, nstart = 20)

# Display the cluster centers
print("Cluster Centers:")
print(kmeans_result$centers)

# Assign each data point to a cluster
cluster_assignments <- kmeans_result$cluster
```

```
# Display the cluster assignments
print("Cluster Assignments:")
print(cluster_assignments)

# Visualize the results
plot(data, col = cluster_assignments, pch = 19, main = "K-Means Clustering")
points(kmeans_result$centers, col = 1:k, pch = 3, cex = 2, lwd = 2)
```

Data preprocessing, such as normalization, if often desirable. Like similar algorithms, alternative distance metrics may also be used. The default is Euclidean distance.

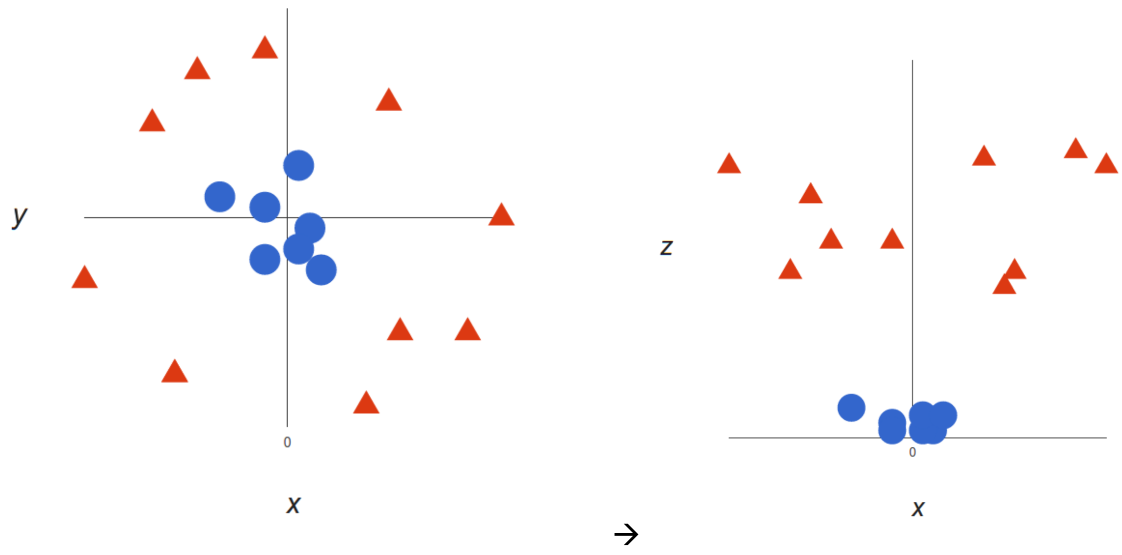


Support Vector Machines (SVMs) are supervised machine learning algorithms used for both classification and regression tasks. SVMs aim to find the optimal hyperplane that maximally separates instances of different classes or predicts the target value with the largest margin. The key idea behind SVMs is to transform the input data into a high-dimensional feature space, where a linear decision boundary can be found.

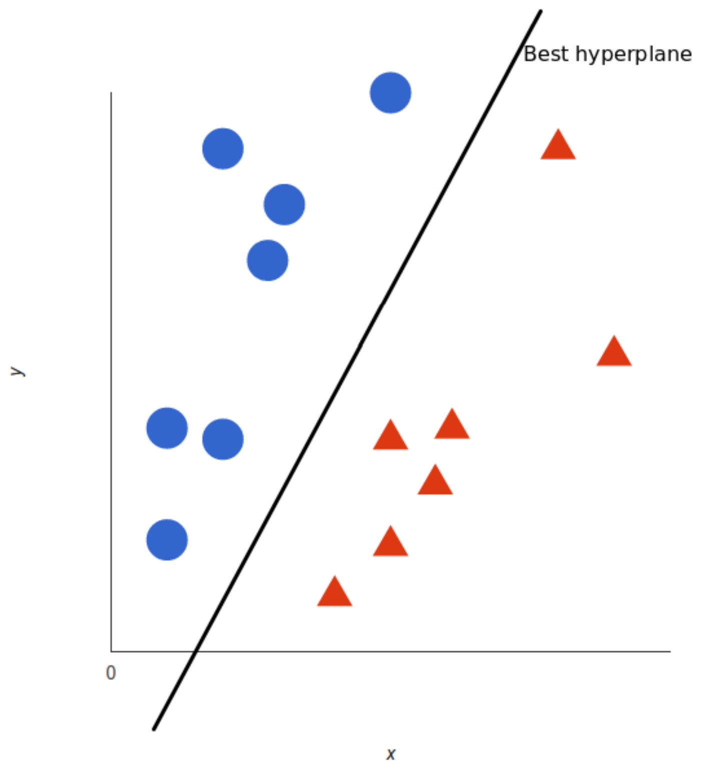
How SVMs are implemented:

Data Preparation: SVMs require labeled training data, where each instance is associated with a class label (in the case of classification) or a target value (in the case of regression). The data is typically represented as a set of feature vectors, where each vector represents the values of the input features.

Feature Mapping (Kernel Trick): SVMs often employ a technique called the "kernel trick" to implicitly map the original feature vectors into a higher-dimensional feature space. This mapping allows for the discovery of more complex decision boundaries that would be difficult to find in the original feature space. Common kernel functions include the linear kernel, polynomial kernel, radial basis function (RBF) kernel, and sigmoid kernel.



Optimization: The SVM algorithm formulates the task as an optimization problem. It aims to find the hyperplane that maximizes the margin between the closest instances of different classes (for classification) or minimizes the regression error (for regression). The optimization problem can be expressed as a quadratic programming (QP) problem or solved using optimization techniques like sequential minimal optimization (SMO).



Parameter Selection: SVMs have two main parameters to tune: the regularization parameter C and the choice of kernel function (if a non-linear kernel is used). The regularization parameter, C, controls the trade-off between achieving a wider margin and allowing misclassifications. A smaller C allows for a larger margin but may lead to more misclassifications, while a larger C imposes a stricter penalty for misclassifications. The kernel function is chosen based on the characteristics of the data and the complexity of the decision boundary.

Prediction: Once the SVM model is trained, it can be used for making predictions on new, unseen instances. For classification, the decision is made based on which side of the hyperplane the instance falls on. Positive values indicate one class, while negative values indicate the other class. For regression, the target value is predicted based on the distance from the hyperplane.

Model Evaluation: The performance of an SVM model is typically evaluated using appropriate evaluation metrics such as accuracy, precision, recall, F1-score (for classification), or mean squared error (for regression). Cross-validation techniques can be employed to estimate the generalization performance of the model.

- SVMs are effective in handling high-dimensional feature spaces, making them suitable for both linear and non-linear problems.
- SVMs are robust to overfitting, thanks to the margin maximization objective and the regularization parameter C.
- SVMs are sensitive to the choice of parameters, especially the regularization parameter C. Improper parameter selection can lead to poor generalization performance.
- SVMs can be memory-intensive, especially when dealing with large datasets. Techniques like kernel approximation or support vector reduction can be employed to mitigate this issue.

- SVMs are binary classifiers by default, but techniques like one-vs-all or one-vs-one can be used to extend them to multi-class classification problems.

Overall, SVMs are powerful algorithms widely used in various machine learning tasks, including classification, regression, and anomaly detection. They offer robust performance, particularly in scenarios with a clear margin between classes or when non-linear decision boundaries are required.

The e1071 package in R provides functions for Support Vector Machines (SVM). Here's a simple example using the built-in Iris dataset:

```
# Install and load the 'e1071' package
install.packages("e1071")
library(e1071)

# Load the Iris dataset
data(iris)

# Create a binary classification problem (setosa vs. non-setosa)
iris_binary <- iris
iris_binary$Species <- factor(ifelse(iris$Species == "setosa", "setosa", "non-setosa"))

# Split the dataset into training and testing sets
set.seed(123)
indices <- sample(1:nrow(iris_binary), 0.7 * nrow(iris_binary))
train_data <- iris_binary[indices, ]
test_data <- iris_binary[-indices, ]

# Train an SVM model
svm_model <- svm(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data = train_data,
kernel = "linear")

# Make predictions on the test set
predictions <- predict(svm_model, test_data)

# Evaluate the performance
confusion_matrix <- table(predictions, test_data$Species)
print("Confusion Matrix:")
print(confusion_matrix)

# Calculate accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", accuracy))
```

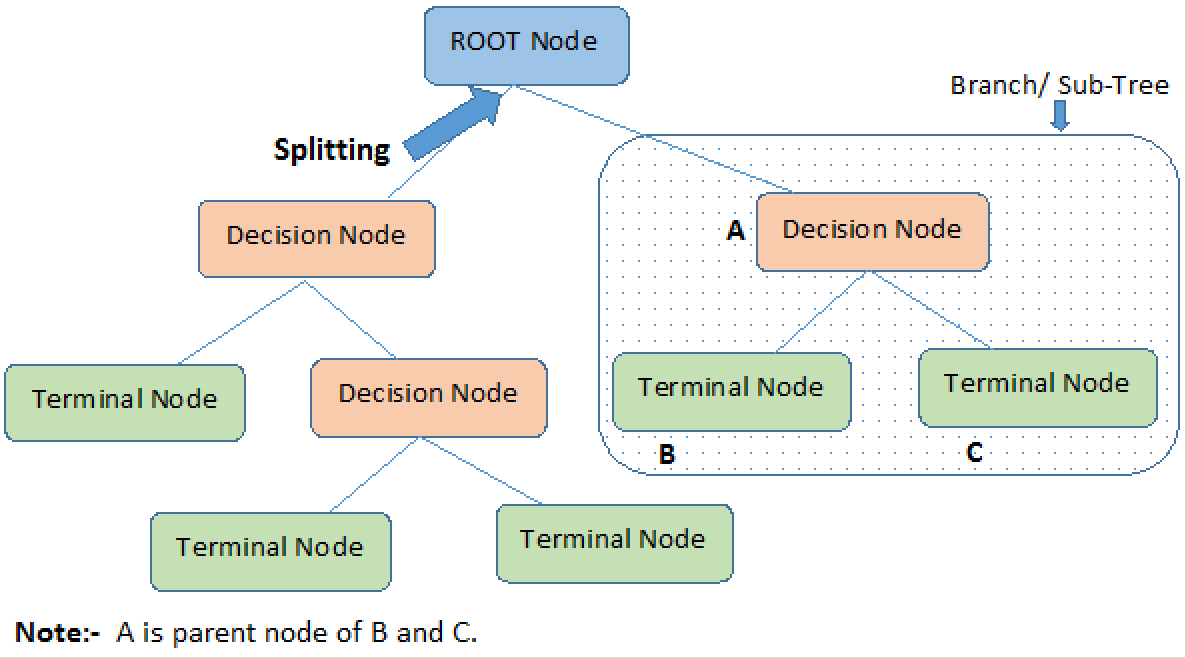
Note: SVMs offer various kernel options (linear, polynomial, radial basis function, etc.), and the choice of kernel depends on the nature of your data. Additionally, parameters like cost (C) can be tuned for better performance.

Decision Trees are versatile supervised machine learning algorithms used for both classification and regression tasks. They create a model that predicts the target variable's value by learning simple decision rules inferred from the input features. Decision Trees are highly interpretable and can handle both numerical and categorical data.

How Decision Trees are implemented:

Data Preparation: Decision Trees require labeled training data, where each instance is associated with a class label (in the case of classification) or a target value (in the case of regression). The data is typically represented as a set of feature vectors, where each vector represents the values of the input features.

Splitting Criteria: Decision Trees employ a splitting criterion to determine how to divide the data at each node of the tree. The most common splitting criteria are Gini impurity and entropy for classification tasks and mean squared error or mean absolute error for regression tasks. These criteria measure the homogeneity or impurity of a set of instances and help decide the best feature and split point for creating child nodes.



Building the Tree: The construction of the Decision Tree starts with the root node, which represents the entire dataset. At each node, the algorithm selects the best feature and split point based on the chosen splitting criterion. This process is recursively applied to create child nodes, splitting the data into subsets based on the selected criteria until a stopping condition is met. Stopping conditions can include reaching a maximum depth, a minimum number of instances per node, or the inability to improve the splitting criterion significantly.

Handling Categorical Features: Decision Trees can handle both numerical and categorical features. For categorical features, the algorithm typically performs a multi-way split, creating separate branches for each category. This process effectively partitions the data based on the feature's different categories.

Pruning (Optional): Pruning is a technique used to prevent overfitting by reducing the complexity of the Decision Tree. It involves removing unnecessary branches or nodes that do not contribute significantly to the model's predictive power. Pruning can be done using approaches such as cost complexity pruning (also known as weakest link pruning) or reduced error pruning.

Prediction: Once the Decision Tree is built, it can be used for making predictions on new, unseen instances. For classification, each instance is traversed through the tree, following the decision rules at each node until a leaf node is reached, which represents the predicted class label. For regression, the target value is determined based on the average or majority value of instances in the leaf node.

Model Evaluation: The performance of the Decision Tree model is typically evaluated using appropriate evaluation metrics such as accuracy, precision, recall, F1-score (for classification), or mean squared error (for regression). Cross-validation techniques can be employed to estimate the generalization performance of the model.

Some important considerations of Decision Trees:

- Decision Trees are interpretable and provide insights into the decision-making process, as the rules learned can be easily visualized and understood.
- Decision Trees can handle both categorical and numerical features, making them suitable for a wide range of data types.
- Decision Trees can suffer from overfitting, especially when the tree grows deep or when dealing with noisy or sparse datasets. Pruning and other techniques can help mitigate this issue.
- Decision Trees are prone to instability and can produce different trees when trained on slightly different datasets. Ensemble methods like Random Forests or Gradient Boosted Trees can help improve stability and predictive performance.
- Decision Trees are computationally efficient for both training and prediction. However, their complexity can increase with the number of features or instances, making them less suitable for high-dimensional data.
- Decision Trees can handle missing values by using strategies like surrogate splits or imputation techniques.
- Decision Trees are sensitive to small changes in the data, which can lead to different splits and different results. Randomization techniques like feature bagging or random subspace can introduce variability to address this issue.

Overall, Decision Trees are powerful and widely used algorithms in machine learning due to their simplicity, interpretability, and ability to handle both classification and regression tasks.

The rpart package in R is commonly used for implementing decision trees. Here's a simple example using the built-in Iris dataset:

```
# Install and load the 'rpart' package
install.packages("rpart")
library(rpart)

# Load the Iris dataset
data(iris)
```

```
# Create a decision tree model
tree_model <- rpart(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data = iris,
method = "class")

# Visualize the decision tree
plot(tree_model)
text(tree_model, cex = 0.8)

# Make predictions on new data (e.g., using a subset of the original data for simplicity)
new_data <- data.frame(
  Sepal.Length = c(5.0, 6.5, 7.2),
  Sepal.Width = c(3.0, 3.0, 3.1),
  Petal.Length = c(1.5, 4.6, 6.0),
  Petal.Width = c(0.2, 1.5, 2.5)
)

predictions <- predict(tree_model, newdata = new_data, type = "class")
print("Predicted Species:")
print(predictions)
```

Note: Decision trees can be more complex, and you might want to consider tuning parameters or pruning the tree for better performance and interpretability in a real-world scenario (helps to avoid overfitting).

More examples of all four algorithms are available in the resources below.

Resources:

1. https://www.datacamp.com/tutorial/k-nearest-neighbors-knn-classification-with-r-tutorial
2. https://rpubs.com/pmtam/knn
3. https://www.geeksforgeeks.org/k-nn-classifier-in-r-programming/
4. https://towardsdatascience.com/k-nearest-neighbors-algorithm-with-examples-in-r-simply-explained-knn-1f2c88da405c
5. https://www.analyticsvidhya.com/blog/2015/08/learning-concept-knn-algorithms-programming/
6. https://www.edureka.co/blog/knn-algorithm-in-r/
7. https://www.datanovia.com/en/lessons/k-means-clustering-in-r-algorith-and-practical-examples/
8. https://uc-r.github.io/kmeans_clustering
9. https://www.datacamp.com/tutorial/k-means-clustering-r
10. https://www.geeksforgeeks.org/k-means-clustering-in-r-programming/
11. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/kmeans.html
12. https://www.datacamp.com/tutorial/support-vector-machines-r
13. https://www.geeksforgeeks.org/classifying-data-using-support-vector-machinessvms-in-r/
14. https://uc-r.github.io/svm
15. https://cran.r-project.org/web/packages/e1071/vignettes/svmdoc.pdf
16. https://www.datacamp.com/tutorial/decision-trees-R
17. https://www.r-bloggers.com/2021/04/decision-trees-in-r/

18. https://www.guru99.com/r-decision-trees.html
19. https://community.rstudio.com/t/decision-tree-in-r/5561