Lecture 6

Ensemble Methods
Boosting & Bagging
Naïve Bayes

Ensemble methods are a class of machine learning techniques used in data mining to improve the predictive performance of models by combining the outputs of multiple base models (learners). These base models can be of the same type or different types. The idea behind ensemble methods is to harness the wisdom of the crowd, aggregating the opinions of multiple models to make more accurate and robust predictions. Some commonly used ensemble methods in data mining include:

Bagging (Bootstrap Aggregating)

Bagging is a simple ensemble method that involves creating multiple bootstrapped (randomly sampled with replacement) datasets from the original data and training a separate base model on each dataset. The final prediction is typically determined by taking a majority vote for classification problems or averaging the predictions for regression problems.

Random Forest, a popular algorithm, is an extension of bagging that applies this approach to decision trees, resulting in a more robust and accurate model.

Boosting:

Boosting is a family of ensemble methods that aim to improve the performance of weak learners (models slightly better than random guessing) by iteratively training and combining them.

AdaBoost (Adaptive Boosting) is a well-known boosting algorithm that assigns weights to misclassified instances and focuses on difficult examples in each iteration.

Gradient Boosting techniques, such as *Gradient Boosting Machines (GBM)* and *XGBoost*, build models sequentially, minimizing the errors of the previous models.

Stacking (Stacked Generalization): Stacking combines the predictions of multiple base models by training a meta-model on their outputs. The base models make predictions on the dataset, and their predictions are used as input features for the meta-model, which generates the final prediction. Stacking can be used to capture complementary information from different base models.

Voting and Averaging: Simple ensemble methods include majority voting and averaging of base model predictions. In majority voting, the most frequently predicted class by the base models is selected for classification problems. In averaging, the predictions of the base models are averaged to produce the final prediction for regression problems.

Random Subspace Method (Feature Bagging): Similar to bagging, this technique applies randomization to feature selection. It trains multiple base models, each on a random subset of features. Feature bagging can help reduce overfitting and increase the diversity of the base models.

Random Patches: Random patches combine both bootstrapping and feature bagging. In this approach, random subsets of data and random subsets of features are used to train individual base models. This technique is often used with ensemble methods like Random Forest.

Blending: Blending is similar to stacking but involves partitioning the training data into different subsets. Each subset is used to train a distinct base model. The base models' predictions are then combined, typically with a simple model like linear regression.

Ensemble methods are known for their ability to enhance the predictive accuracy and generalization performance of machine learning models. They are widely used in various domains, including classification, regression, and feature selection. The choice of which ensemble method to use often depends on the characteristics of the problem, the base models available, and the desired level of model diversity and performance.

Bagging and boosting are two popular ensemble methods used in machine learning to improve the performance of models by combining multiple base models. While they share the goal of increasing predictive accuracy, they differ in several key aspects, despite sometimes being confused.

Bagging (Bootstrap Aggregating):
Base Model Diversity: Bagging focuses on reducing the variance of the base models. It aims to create multiple base models with similar predictive abilities. Each base model is trained independently on a bootstrapped (randomly sampled with replacement) subset of the training data. The predictions of individual base models are typically combined through majority voting (classification) or averaging (regression).

Sequential vs. Parallel: Bagging base models are trained independently and in parallel. There is no sequential adjustment of weights or focus on misclassified instances.

Robustness to Overfitting: Bagging helps reduce overfitting by combining the predictions of multiple base models. Since it averages or votes on the outputs, it tends to provide a more stable and less noisy prediction.

Examples: Random Forest is a well-known ensemble method based on bagging. It applies bagging to decision trees, creating multiple decision trees with the goal of reducing variance and improving generalization.

Boosting:
Base Model Diversity: Boosting focuses on improving the accuracy of weak learners (models that perform slightly better than random guessing). It aims to combine base models in a sequential manner, with each new model addressing the mistakes of the previous ones. The weights of data points are adjusted in each iteration to emphasize incorrectly predicted instances, making boosting particularly useful for difficult examples.

Sequential Process: Boosting is an iterative and sequential process. Base models are trained sequentially, and the subsequent models give more weight to instances that were misclassified by previous models. This leads to a strong focus on challenging data points.

Handling Overfitting: Boosting can be more prone to overfitting compared to bagging, as it aggressively adjusts the model to minimize errors on the training data. Techniques like early stopping and model complexity control are often used to mitigate overfitting.

Examples: AdaBoost (Adaptive Boosting), Gradient Boosting, and XGBoost are well-known boosting algorithms. These algorithms assign weights to instances based on their accuracy and focus on improving the classification or regression performance over iterations.

In summary, bagging aims to reduce the variance of the base models and is characterized by the independence of training instances, while boosting focuses on increasing the accuracy of the ensemble by sequentially correcting the errors made by previous base models. The choice between bagging and boosting depends on the nature of the problem, the quality of base models, and the trade-off between variance reduction and error correction.

AdaBoost (Adaptive Boosting), Gradient Boosting, and **XGBoost** are all popular ensemble algorithms used in machine learning for improving predictive accuracy. They are based on boosting, a technique that sequentially combines the predictions of weak base models to create a strong learner. While they share similarities in boosting, they also have key differences in terms of algorithm design and performance optimization:

AdaBoost (Adaptive Boosting):
Base Model: AdaBoost typically uses decision trees with a depth of one (stumps) as base models. These stumps are often referred to as "weak learners."

Algorithm Overview: AdaBoost assigns weights to data points and focuses on instances that are misclassified by previous base models. It iteratively trains a sequence of base models, where each new model tries to correct the errors of the previous ones.

Weighted Voting: During each iteration, the weight of incorrectly classified instances is increased, and the weight of correctly classified instances is decreased. The final prediction is made by weighted voting, giving more weight to the most accurate base models.

Robustness to Outliers: AdaBoost is sensitive to outliers or noisy data points because it gives them higher importance as it attempts to correct their misclassifications.

Performance Limitation: AdaBoost can suffer from overfitting if the base model complexity increases, or if the data has a lot of noise.

Gradient Boosting:
Base Model: Gradient Boosting typically uses decision trees, but the trees can be deeper and more complex compared to AdaBoost. It's common to use regression trees for regression problems and classification trees for classification problems.

Algorithm Overview: Gradient Boosting is based on the gradient descent optimization technique. It minimizes a loss function by iteratively adding base models that move in the direction of the negative gradient of the loss function.

Sequential Learning: The base models are trained sequentially, and each new model tries to minimize the error made by the previous ones. Gradient Boosting learns a weighted sum of base models.

Flexibility: Gradient Boosting is more flexible in terms of base model choice and can accommodate a variety of loss functions and optimization techniques.

Handling Outliers: Gradient Boosting is relatively robust to outliers and can handle them gracefully.

XGBoost (Extreme Gradient Boosting):
Base Model: XGBoost uses a specialized type of decision tree called "CART" (Classification and Regression Tree) as base models. It employs a parallel and distributed computing framework.

Algorithm Overview: XGBoost is designed to optimize performance and computational efficiency. It uses a regularized objective function and includes a unique feature that penalizes complexity.

Performance Optimizations: XGBoost incorporates multiple performance optimizations, such as parallelization, out-of-core computing, and hardware-aware computation. These optimizations make it extremely fast and memory efficient.

Regularization: XGBoost provides L1 and L2 regularization techniques to prevent overfitting.

Advanced Features: XGBoost includes advanced features like handling missing values, monotonic constraints, and built-in support for multi-class classification.

Comparison:
All three algorithms are boosting methods, but XGBoost is known for its computational efficiency and scalability.

- AdaBoost focuses on simple base models and can be sensitive to noisy data.
- Gradient Boosting is more flexible and can handle more complex base models.
- XGBoost combines the advantages of Gradient Boosting with significant performance enhancements and regularization techniques.

The choice between AdaBoost, Gradient Boosting, and XGBoost depends on the specific problem, the quality of the data, the need for performance optimization, and the trade-offs between computational resources and predictive accuracy. XGBoost is often a popular choice for many machine learning tasks due to its balance of performance and efficiency.

To implement boosting algorithms in R, you can use various packages that provide implementations of popular boosting algorithms like AdaBoost, Gradient Boosting, and XGBoost. Here's a general overview of how to implement these boosting algorithms in R:

AdaBoost (Adaptive Boosting): You can implement AdaBoost in R using the "adabag" package, which provides tools for applying AdaBoost to classification problems. Here's a basic example:

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

# Load your dataset (e.g., data and labels)
```

```
data <- read.csv("your_data.csv")

# Create a formula for your classification task (e.g., Class ~
Feature1 + Feature2)
formula <- Class ~ Feature1 + Feature2

# Train an AdaBoost model
adaboost_model <- boosting(formula, data = data, mfinal = 100)  #
Adjust 'mfinal' as needed

# Make predictions
predictions <- predict(adaboost_model, newdata = data)

# Evaluate the model (e.g., calculate accuracy, confusion matrix,
etc.)
```

Gradient Boosting: For Gradient Boosting, you can use the "gbm" (Generalized Boosted Regression Models) package, which is primarily designed for regression but can also be used for classification problems. Here's an example:

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

# Load your dataset (e.g., data and labels)
data <- read.csv("your_data.csv")

# Create a formula for your classification task (e.g., Class ~
Feature1 + Feature2)
formula <- Class ~ Feature1 + Feature2

# Train a Gradient Boosting model
gbm_model <- gbm(formula, data = data, distribution = "bernoulli",
n.trees = 100)
# Adjust 'n.trees' as needed

# Make predictions
predictions <- predict(gbm_model, newdata = data, n.trees = 100)
# Adjust 'n.trees' as needed

# Evaluate the model (e.g., calculate accuracy, confusion matrix,
etc.)
```

XGBoost (Extreme Gradient Boosting): To use XGBoost in R, you should install and use the "xgboost" package. XGBoost is a popular boosting algorithm known for its speed and efficiency. Here's an example:

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

# Load your dataset (e.g., data matrix and labels)
data <- read.csv("your_data.csv")
```

```
# Prepare the data matrix and labels
X <- as.matrix(data[, c("Feature1", "Feature2")])
y <- data$Class

# Train an XGBoost model
xgb_model <- xgboost(data = X, label = y, nrounds = 100)
# Adjust 'nrounds' as needed

# Make predictions
predictions <- predict(xgb_model, newdata = X)

# Evaluate the model (e.g., calculate accuracy, confusion matrix,
etc.)
```

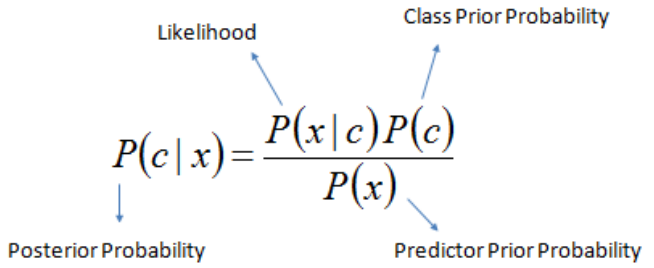
Please note that the above examples provide a basic outline of how to implement boosting algorithms in R. You should replace "your_data.csv" with the actual path to your dataset and adapt the code to your specific dataset, features, and classification problem. Additionally, you can explore further options and tuning parameters provided by these packages to optimize your boosting models. See the package documentation for details.

Naïve Bayes is a probabilistic machine learning algorithm used for classification and is based on Bayes' theorem. It's particularly well-suited for text classification and spam filtering but can be applied to a wide range of classification tasks. The "naïve" in its name arises from the assumption that the features used for classification are conditionally independent, which simplifies the model but may not hold in all real-world scenarios. Here's how the Naïve Bayes algorithm works:

1. *Data Representation*: Naïve Bayes starts with a labeled dataset, where each data point is represented as a feature vector and assigned to one of several classes.

2. *Feature Independence (Naïve Assumption)*: The central assumption in Naïve Bayes is that the features used for classification are conditionally independent. In other words, the presence or absence of one feature does not depend on the presence or absence of any other feature. This is a "naïve" simplifying assumption that may not hold in all situations but works surprisingly well in practice.

3. *Bayesian Probabilistic Framework*: The algorithm applies Bayes' theorem to calculate the conditional probability of each class given the observed features. The formula for this is:



$$P(c \mid \mathrm{X}) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

- $P(c|x)$ is the posterior probability of class (target) given predictor (attribute).

- $P(c)$ is the prior probability of class.
- $P(x|c)$ is the likelihood which is the probability of predictor given class.
- $P(x)$ is the prior probability of predictor.

4. *Training*: During the training phase, the algorithm calculates the following probabilities for each class: $P(Class)$ is the prior probability of each class, which can be estimated by the frequency of each class in the training data. $P(Features \mid Class)$ is the conditional probability of the features given each class. This is typically estimated by counting the occurrences of each feature in each class.

5. *Prediction*: In the prediction phase, the algorithm calculates the conditional probabilities of each class given the observed features for a new data point. It does this for each class.

6. *Classification*: To classify a data point, the algorithm selects the class with the highest conditional probability given the features. This is often referred to as the "maximum a posteriori" or MAP estimation.

Key Considerations:
Naïve Bayes is simple and computationally efficient, making it well-suited for large datasets. It works particularly well for text classification tasks, such as spam email detection and sentiment analysis. While the independence assumption simplifies the model, it may not hold in all cases, which can lead to suboptimal results for certain types of data. Naïve Bayes is sensitive to feature selection, so careful choice of relevant features is crucial for good performance.

There are three common variants of the Naïve Bayes algorithm, depending on the distribution of the features:

- Gaussian Naïve Bayes: Assumes that the features follow a Gaussian (normal) distribution.
- Multinomial Naïve Bayes: Designed for discrete data, often used in text classification.
- Bernoulli Naïve Bayes: Suitable for binary data, such as presence or absence of features in a document.

Naïve Bayes is a versatile and widely used algorithm in various applications, including spam filtering, document categorization, and sentiment analysis.

Let's walk through a small example of using the Naïve Bayes algorithm for a basic text classification task. In this example, we'll classify short text messages as either "spam" or "not spam" (ham). We'll use a simple toy dataset of text messages. This type model can be applied to other types of data in addition to text-based data.

Step 1*: Import Necessary Libraries* : In R, you can use the "tm" (text mining) package for text preprocessing and the "e1071" package for Naïve Bayes classification. First, install and load the required packages:

```
install.packages("tm")
install.packages("e1071")
library(tm) library(e1071)
```

Step 2: *Prepare the Data:* We'll create a small dataset of text messages and their corresponding labels (spam or ham).

```
# Sample data
```

```
text_messages <- c( "Cheap watches for sale!", "Hi there, how are
you?", "Win a free vacation today!", "Meeting at 2 PM.", "URGENT: Call
me ASAP!" )
labels <- c("spam", "ham", "spam", "ham", "spam")
# Create a data frame
data <- data.frame(text = text_messages, label = labels)
```

Step 3: *Preprocess the Text Data:* Text data needs to be preprocessed, which includes converting text to lowercase, removing punctuation, and creating a document-term matrix. The "tm" package is useful for these tasks.

```
# Create a Corpus (collection of text documents)
corpus <- Corpus(VectorSource(data$text))
# Preprocess the text
corpus <- tm_map(corpus, content_transformer(tolower))
corpus <- tm_map(corpus, removePunctuation)
corpus <- tm_map(corpus, removeNumbers)
corpus <- tm_map(corpus, stripWhitespace)
# Create a Document-Term Matrix (DTM)
dtm <- DocumentTermMatrix(corpus)
```

Step 4: *Train the Naïve Bayes Model:* We'll train a Naïve Bayes model using the preprocessed text data and labels.

```
# Train a Naive Bayes
model nb_model <- naiveBayes(x = dtm, y = data$label)
```

Step 5: *Make Predictions:* Use the trained Naïve Bayes model to make predictions for new text messages. In this example, we'll predict the labels for the same messages used in training.

```
# Prepare new text data for prediction
new_text <- c("Congratulations! You've won a prize!", "Let's have
lunch tomorrow.")
# Create a new Corpus
new_corpus <- Corpus(VectorSource(new_text))
# Preprocess the new text
new_corpus <- tm_map(new_corpus, content_transformer(tolower))
new_corpus <- tm_map(new_corpus, removePunctuation)
new_corpus <- tm_map(new_corpus, removeNumbers)
new_corpus <- tm_map(new_corpus, stripWhitespace)
# Create a Document-Term Matrix for new text
new_dtm <- DocumentTermMatrix(new_corpus)
# Make predictions
predictions <- predict(nb_model, newdata = new_dtm)
```

Step 6: *Evaluate the Model:* For a simple example like this, you can compare the predicted labels to the actual labels to assess the model's performance.

```
# Combine the new text and predictions
results <- data.frame(text = new_text, predicted_label = predictions,
stringsAsFactors = FALSE)
# Print the results
print(results)
```

This example demonstrates a basic text classification task using Naïve Bayes. In practice, you would typically work with larger and more diverse datasets and perform more thorough preprocessing. Additionally, you would evaluate the model's performance using metrics such as accuracy, precision, recall, and F1 score for a more comprehensive assessment.

Multinomial Naïve Bayes, Gaussian Naïve Bayes, Bernoulli Naïve Bayes, and the standard (or regular) Naïve Bayes are variants of the Naïve Bayes algorithm designed for different types of data and feature distributions. They differ in how they handle and model the data. Here's a summary of their key differences, advantages, and disadvantages:

1. **Multinomial Naïve Bayes**:
Data Type: Designed for discrete data, especially text data (e.g., word counts, term frequencies).
Feature Distribution: Assumes that features follow a multinomial distribution, making it suitable for count-based data.

Advantages:

- Works well for text classification tasks, such as spam detection and document categorization.
- Can handle data with multiple discrete categories or classes.

Disadvantages:

- Ignores word order and dependencies between terms in text data.
- May not perform well with continuous or real-valued features.

How it Works:
Data Representation: In text classification, each document is represented as a feature vector, with each feature representing the count or frequency of a term (word) in the document.

Model Assumption: Multinomial Naïve Bayes assumes that the features follow a multinomial distribution.

Probability Estimation: For each class, the algorithm estimates the conditional probability distribution of feature counts/frequencies. It calculates the likelihood of observing each feature in the documents of that class.

Prior Probability: It estimates the prior probability of each class based on the class distribution in the training data.

Classification: To classify a new document, the algorithm calculates the posterior probability of each class given the observed feature counts. It selects the class with the highest posterior probability.

2. **Gaussian Naïve Bayes**:
Data Type: Designed for continuous data with a Gaussian (normal) distribution.
Feature Distribution: Assumes that features follow a Gaussian distribution, making it suitable for real-valued data.

Advantages:

- Effective for continuous and normally distributed features.
- Works well for real-valued data, such as measurements.

Disadvantages:

- Sensitive to outliers in the data.
- May not perform well if features do not follow a Gaussian distribution.

How it Works:
Data Representation: Each data point is represented as a feature vector of real-valued features.

Model Assumption: Gaussian Naïve Bayes assumes that the features follow a Gaussian distribution within each class.

Probability Estimation: For each class, the algorithm estimates the parameters of the Gaussian distribution, including the mean and variance, for each feature.

Prior Probability: It estimates the prior probability of each class based on the class distribution in the training data.

Classification: To classify a new data point, the algorithm calculates the posterior probability for each class based on the Gaussian distribution parameters. It selects the class with the highest posterior probability.

3. ***Bernoulli Naïve Bayes***:
Data Type: Designed for binary data (e.g., presence or absence of features).
Feature Distribution: Assumes that features follow a Bernoulli distribution.

Advantages:

- Useful for binary and presence/absence data, such as text data after binarization.
- Suitable for tasks like sentiment analysis or spam detection.

Disadvantages:

- Ignores feature frequency information, only considering the presence/absence.
- May not perform well with continuous or multi-category data.

How it Works:
Data Representation: Each data point is represented as a binary feature vector, where each feature is either 0 (absent) or 1 (present).

Model Assumption: Bernoulli Naïve Bayes assumes that the features follow a Bernoulli distribution, which models binary data.

Probability Estimation: For each class, the algorithm estimates the probability of each feature being present (1) or absent (0) in documents of that class.

Prior Probability: It estimates the prior probability of each class based on the class distribution in the training data.

Classification: To classify a new data point, the algorithm calculates the posterior probability of each class based on the Bernoulli distribution parameters. It selects the class with the highest posterior probability.

4. **Regular (Standard) Naïve Bayes**:
Data Type: More general and versatile; can be used for both discrete and continuous data.
Feature Distribution: Does not assume a specific feature distribution; it can handle different types of data.

Advantages:

- Versatile and can work with a mix of feature types (e.g., both text and numeric features).
- Doesn't make strict distributional assumptions, making it applicable to a wide range of problems.

Disadvantages:

- May not perform as well as specialized Naïve Bayes variants on specific types of data.
- Assumes feature independence, which may not hold in all cases.

In summary, the choice of which Naïve Bayes variant to use depends on the nature of the data and the problem you're trying to solve. It's essential to consider the data's characteristics and distribution when selecting the appropriate Naïve Bayes algorithm. Specialized variants like Multinomial, Gaussian, and Bernoulli Naïve Bayes are tailored for specific types of data and can offer better performance in those specific contexts. Regular Naïve Bayes is a more general and flexible option when the data may have mixed types or when you want to avoid making strong distributional assumptions.

In all three variants, the "Naïve" part of Naïve Bayes refers to the independence assumption: the algorithms assume that the features are conditionally independent given the class label. While this assumption simplifies the models, it may not always hold in practice. However, Naïve Bayes algorithms are known for their simplicity and often perform well in various classification tasks, especially when the characteristics of the data align with the assumptions of the chosen variant.

When the naïve assumption of independence fails, the models can perform less well than other classification models. Because it's simplicity, however, it may be a good first choice of model, and then compare the results with other models in an attempt to improve the results (performance).

Resources:

1. https://www.r-bloggers.com/2021/04/naive-bayes-classification-in-r/
2. https://www.geeksforgeeks.org/naive-bayes-classifier-in-r-programming/
3. https://uc-r.github.io/naive_bayes
4. https://www.learnbymarketing.com/tutorials/naive-bayes-in-r/
5. https://www.edureka.co/blog/naive-bayes-in-r/
6. https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
7. https://www.analyticsvidhya.com/blog/2017/02/introduction-to-ensembling-along-with-implementation-in-r/
8. https://www.pluralsight.com/guides/ensemble-modeling-with-r
9. https://machinelearningmastery.com/machine-learning-ensembles-with-r/
10. https://daviddalpiaz.github.io/r4sl/ensemble-methods.html
11. https://www.listendata.com/2015/08/ensemble-learning-stacking-blending.html
12. https://www.datacamp.com/tutorial/ensemble-r-machine-learning
13. https://www.tmwr.org/ensembles
14. https://rpubs.com/guptadeepak/ensemble

15. https://www.r-bloggers.com/2021/02/machine-learning-with-r-a-complete-guide-to-gradient-boosting-and-xgboost/
16. https://www.geeksforgeeks.org/boosting-in-r/
17. https://bradleyboehmke.github.io/HOML/gbm.html
18. https://datascienceplus.com/gradient-boosting-in-r/