Hyperparameter

Koko Friansa

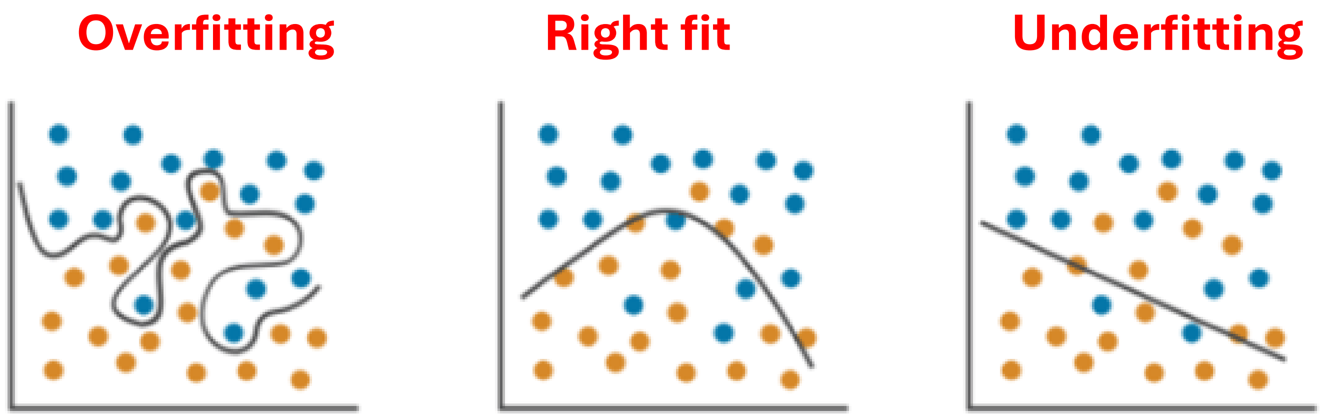
29-11-2024

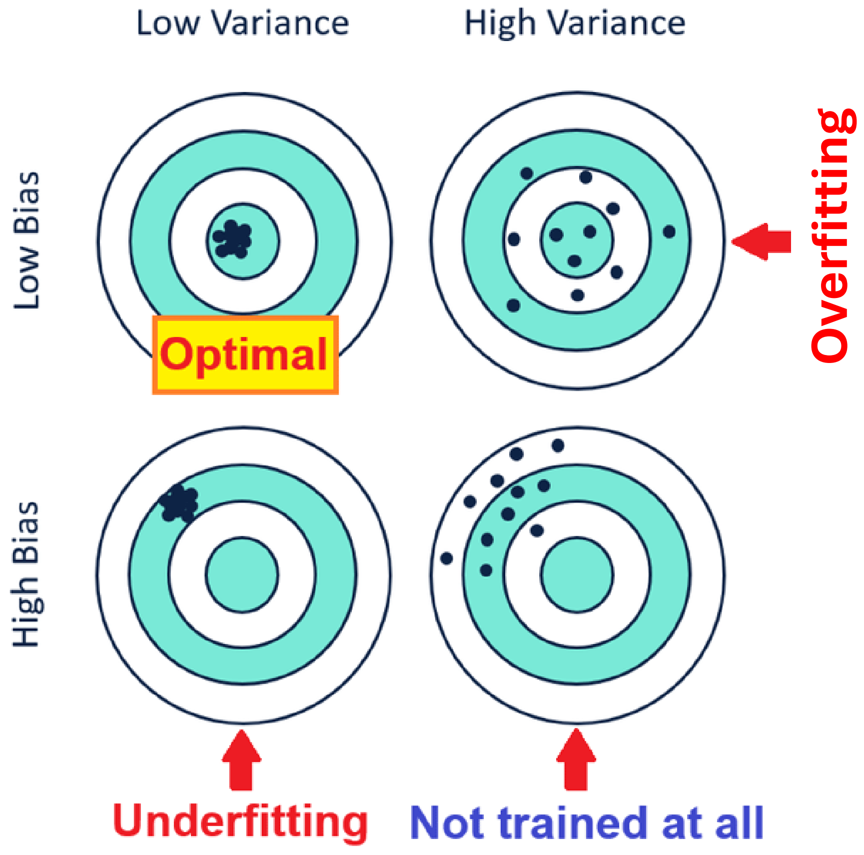
(Sekolah Bisnis Manajemen ITB)

Why Hyperparameter Tuning is Crucial

1. **overfitting** and **underfitting**

- Impact on model **accuracy**
- Computational cost





Introduction

- Every **machine learning** system has hyperparameters
- The most basic task in machine learning is to automatically set these hyperparameters to **optimize performance**

What are Hyperparameters

- **Set before the training** process of a machine learning model begins (Build & Train Models)
- **Control the learning process** to influence how the model learns and the final outcome.

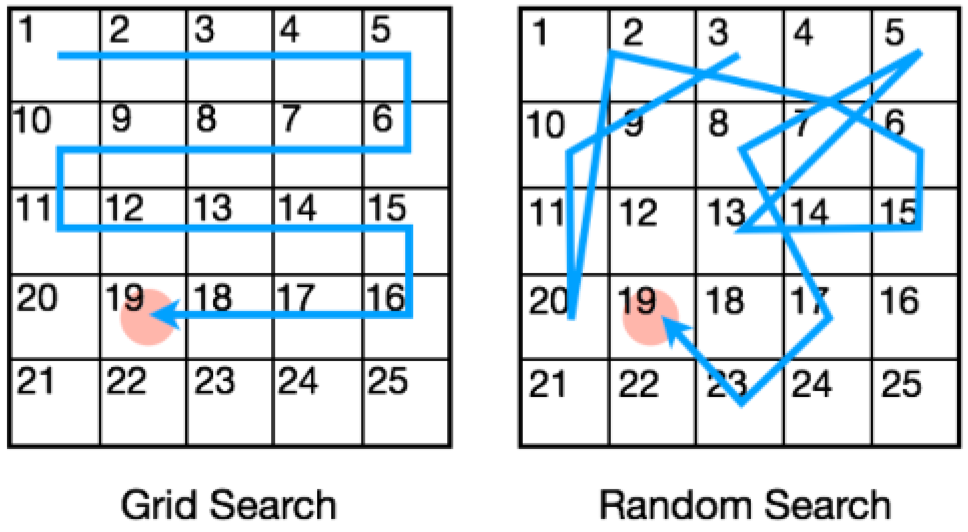
Why Hyperparameter Tuning is Crucial

2. Curse of Dimensionality

- The feature space becomes increasingly sparse for an increasing number of **dimensions**
- Dimensionality reduction (e.g., **Feature selection**)

Hyperparameter Tuning Techniques

- Grid Search
- Random Search



Machine Learning Models

Supervised Learning Algorithm

In supervised learning, the optimal predictive model can be represented as:

$$f^* = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x_i\right), y_i\right)$$

n: Number of samples in the training set.

(x_i, y_i): Individual training data points.

$f(x)$: The predictive model that maps inputs x to predictions.

$\mathcal{L}(f(x), y)$: The loss function measuring the difference between the predicted output $f(x)$ and the true output y.

Supervised Learning Algorithm

- K-nearest neighbor (KNN)
- Perceptron

K-Nearest Neighbors (K-NN)

For a test point x, the predicted label $\hat{y}$ is derived based on the labels y_i of its k-nearest neighbors $N_k(x)$ in the training dataset:

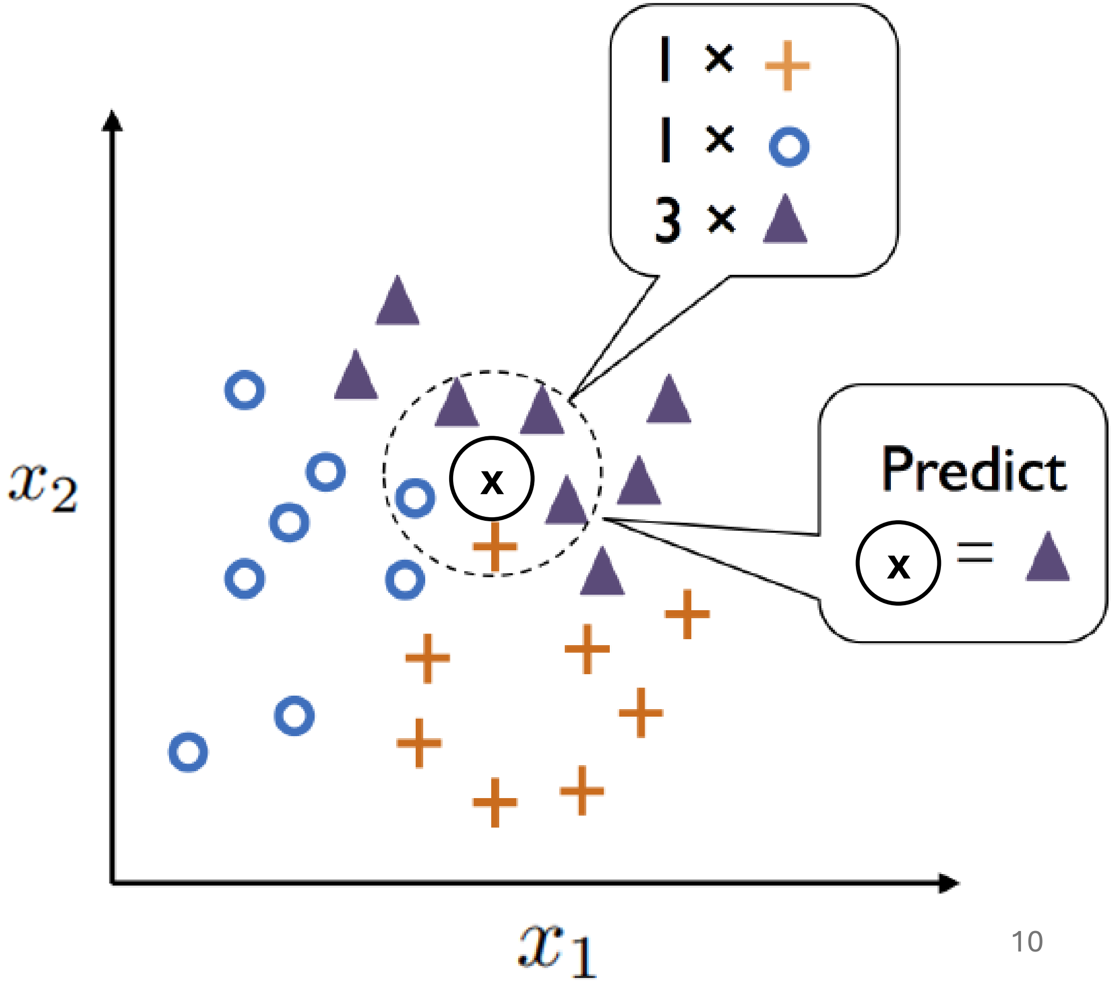
$$\hat{y} = \text{mode}\{y_i : x_i \in N_k(x)\}$$

Where:

$N_k(x)$: The set of k-nearest neighbors to x.

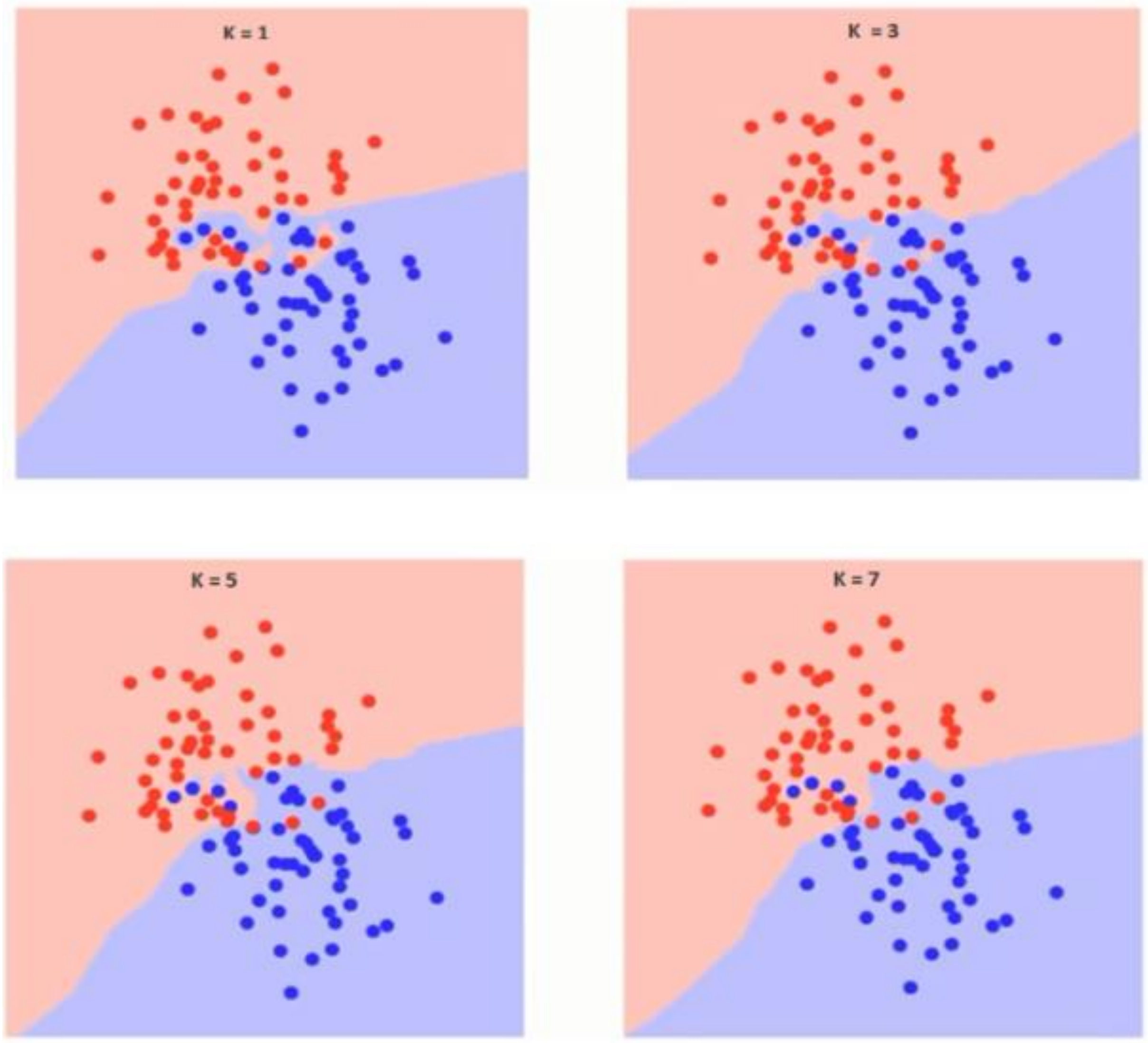
y_i: The label of the i-th neighbor.

mode: The most frequent class label among the neighbors.

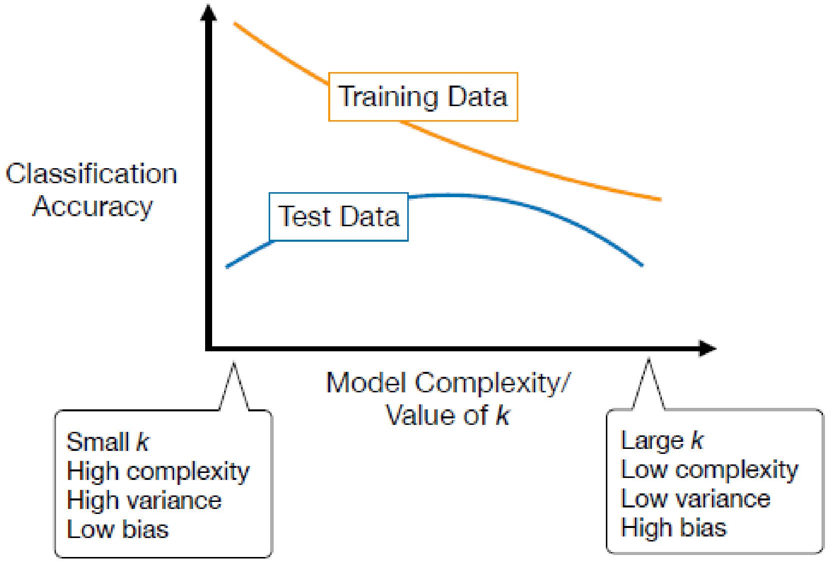


K-nearest neighbor (KNN) is used to classify data points by calculating the distances between different data points.

Hyperparameter: *K*



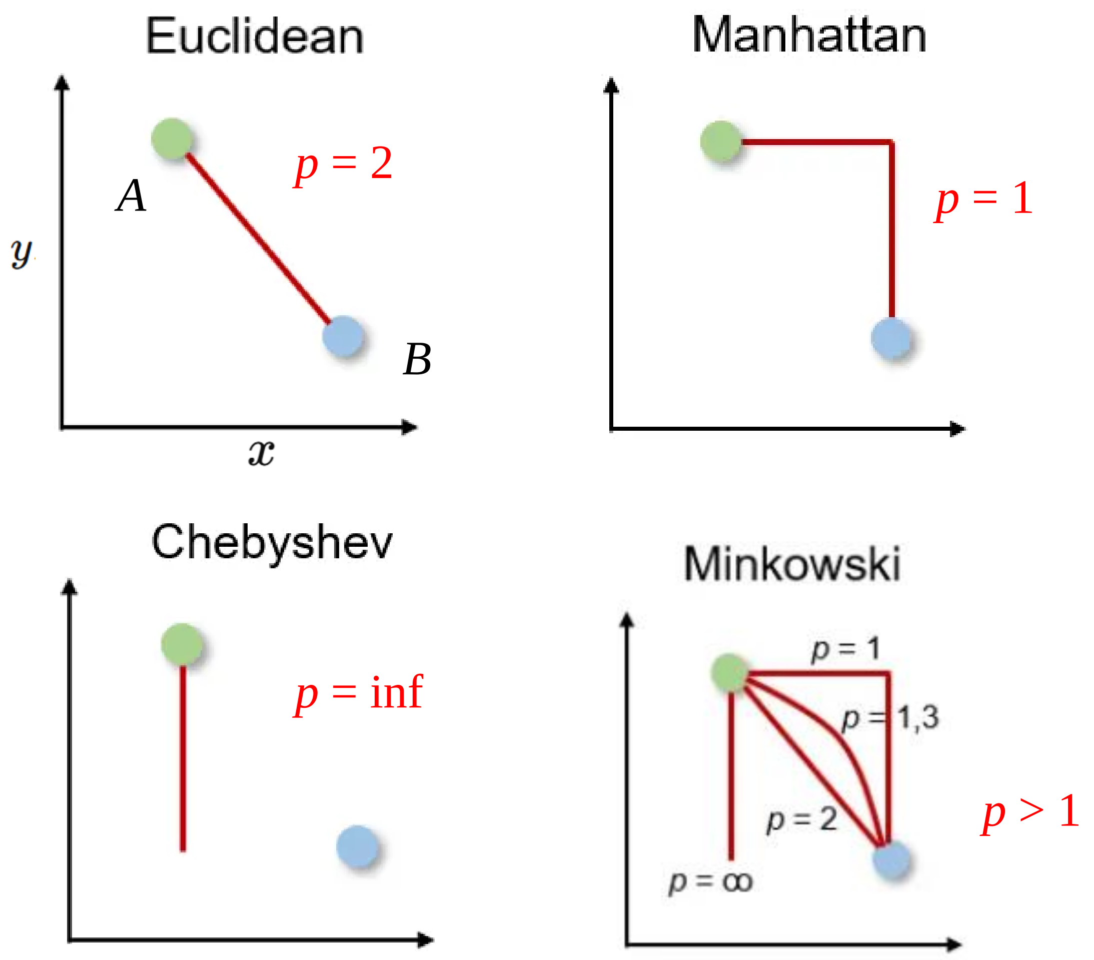
- The value of **K**, the number of nearest neighbors to retrieve
- Choosing the value of **K**:
  - If **K** is too small, sensitive to noise points
  - If **K** is too large, neighborhood may include points from other classes



Calculating the Distances

Distance = $$D_p(A,B) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$

- Euclidean distance (p = 2) $= \sum_{i=1}^{n} |x_i - y_i|$
-
- Manhattan distance (p = 1) $= \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$
- Chebyshev distance (p = ∞) $= \max(|x_i - y_i|)$
- Minkowsky distance $= \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$



Exercise

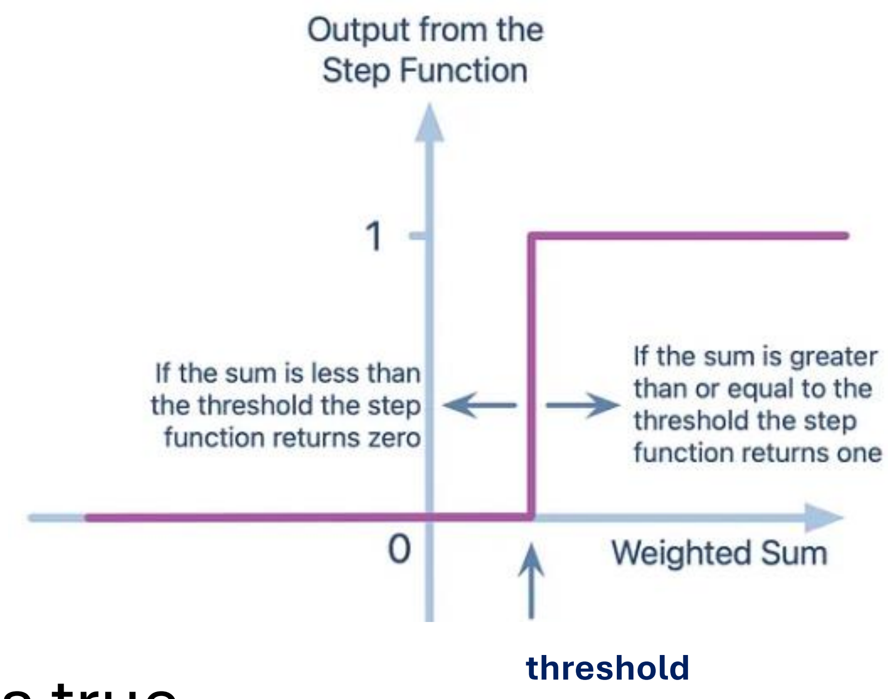
- Build model based on K-NN from this dataset, with k = 2, 3, and 4

	Credit Score (x1)	Income Level (x2)	Loan Approved?
1	86	68	Yes
2	82	50	Yes
3	95	82	Yes
4	88	52	No
5	94	78	No
6	96	53	No
7	93	78	No
8	89	79	Yes
9	88	80	Yes
10	89	82	Yes

Perceptron Model

Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w}^T\mathbf{x} + b \geq 0 \longrightarrow \text{Class 1} \\ \mathbf{0}, & \mathbf{w}^T\mathbf{x} + b < 0 \longrightarrow \text{Class 2} \end{cases}$$



This is called a **step function**, which reads:

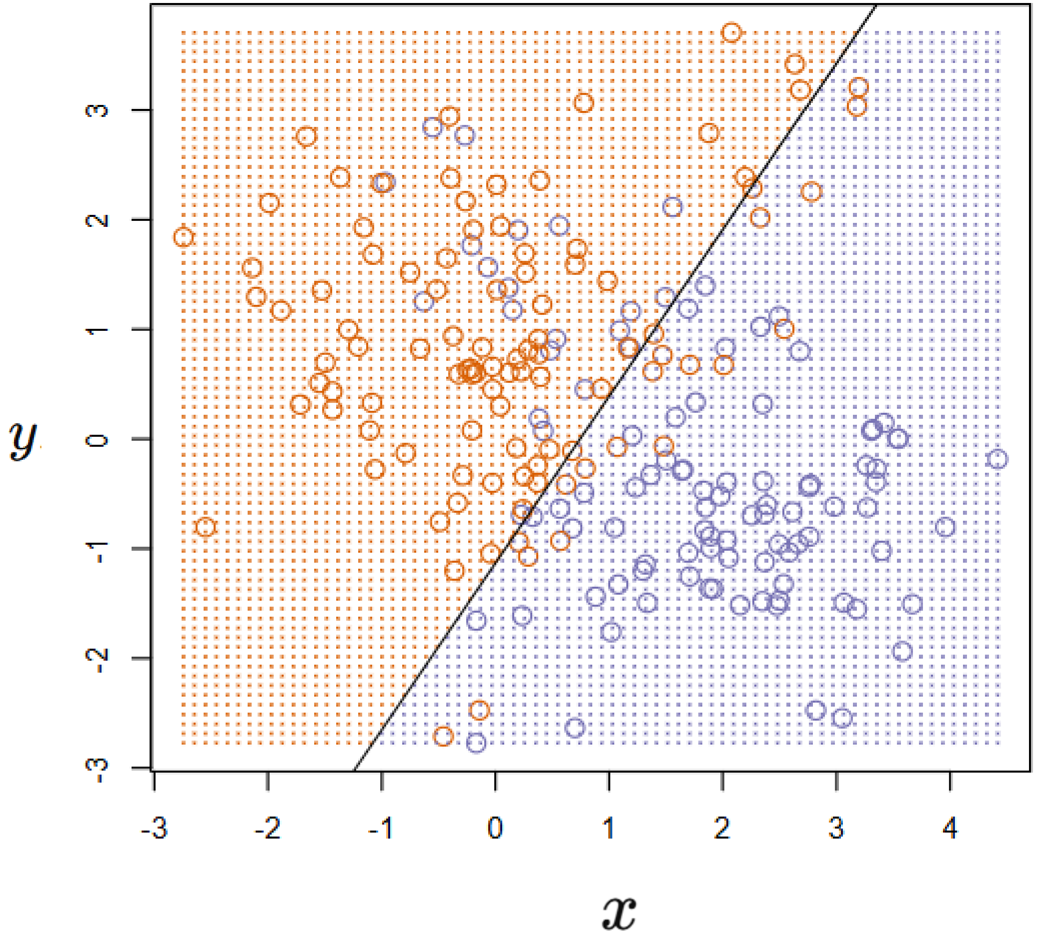
- The output is **1** if $\mathbf{w}^T\mathbf{x} + b \geq 0$ is true,
- and the output is **0** if instead $\mathbf{w}^T\mathbf{x} + b < 0$ is true.
- $\mathbf{w}$ = weights
- b = bias

Binary Classification

- Linear prediction function : $y = w^T x + b$

Where:

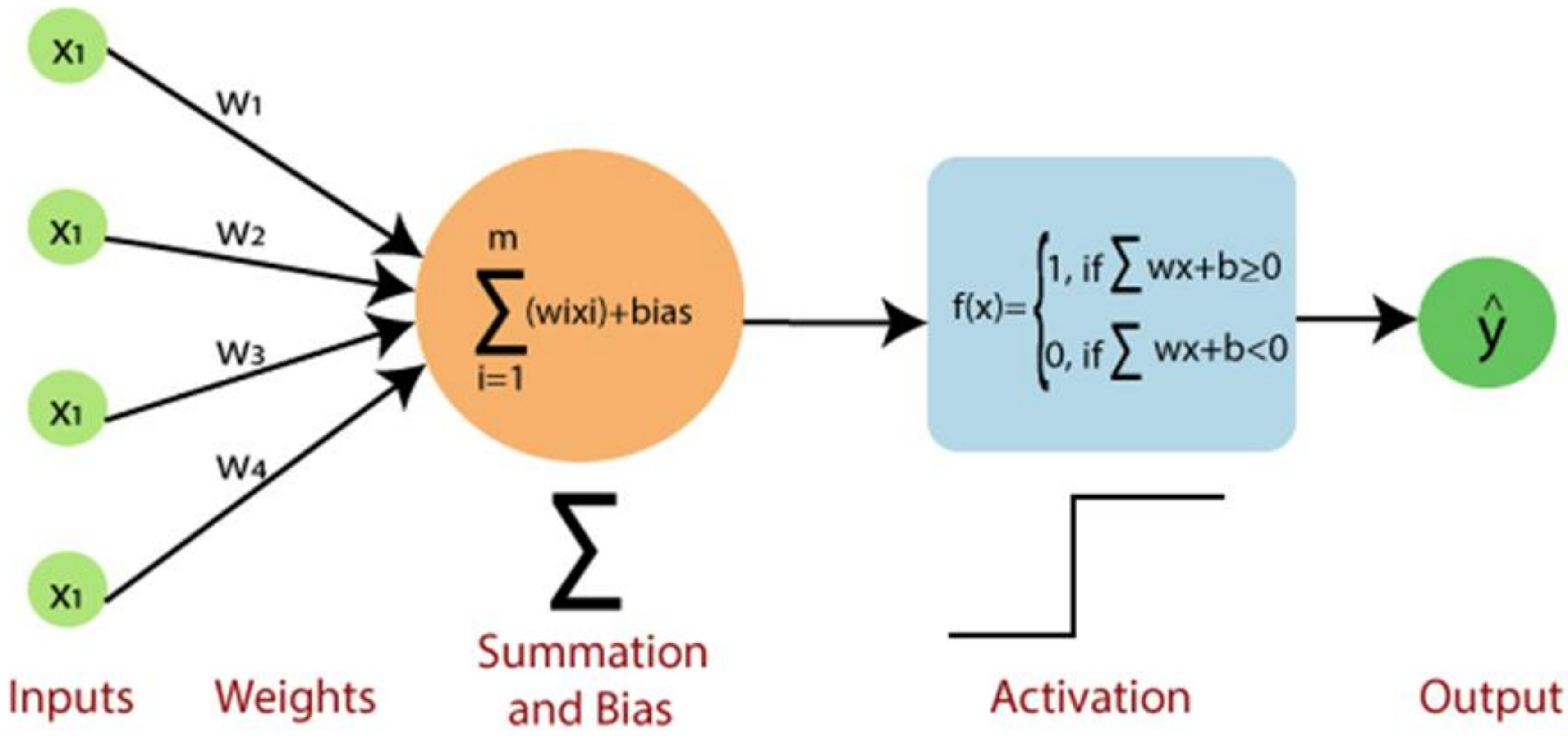
- y: Continuous target variable.
- x: Feature vector.
- w: Weight vector (parameters).
- b: Bias term (intercept).



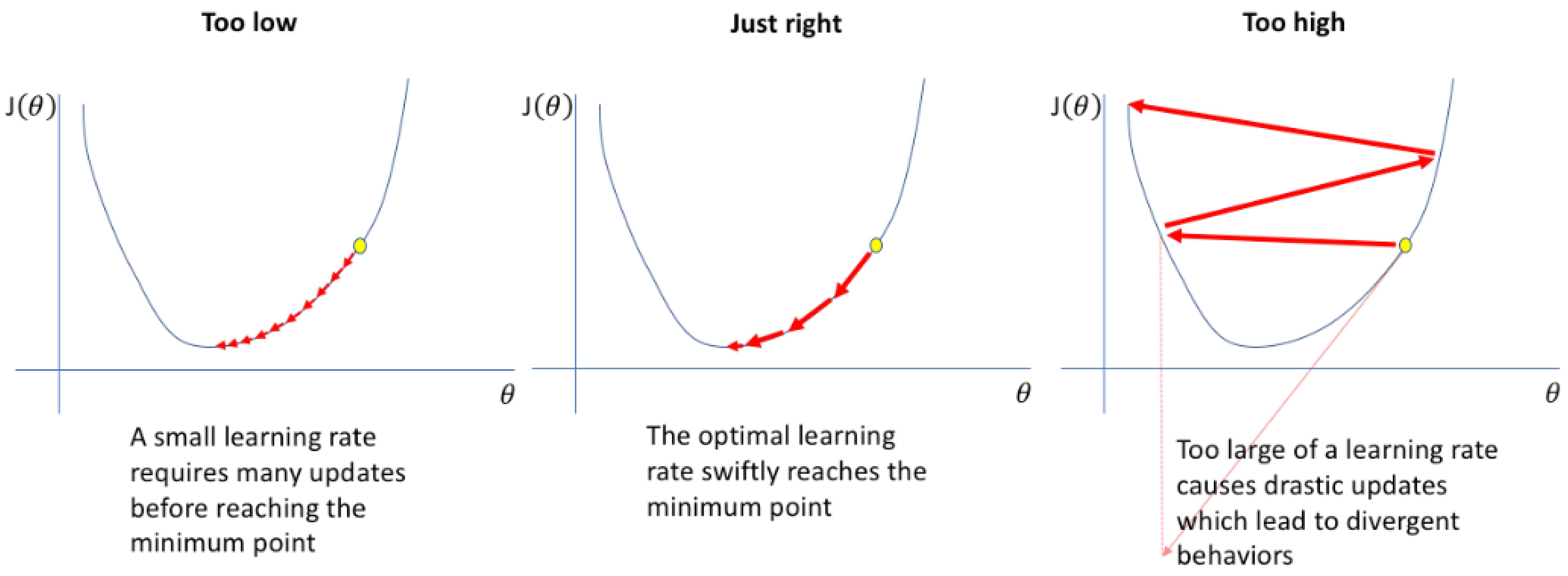
$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w}^T\mathbf{x} + b \geq 0 \\ \mathbf{0}, & \mathbf{w}^T\mathbf{x} + b < 0 \end{cases}$$

Perceptron Model

Perceptron is an algorithm for binary classification that uses a linear prediction function:



Learning Rate

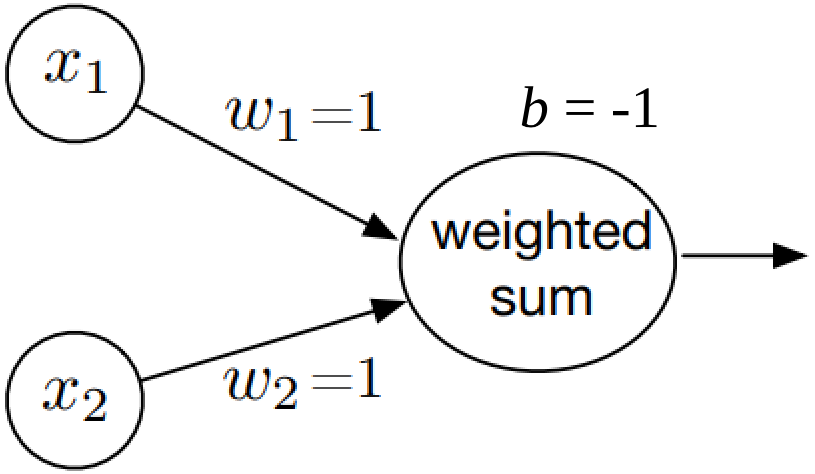


If your learning rate is set too low, training will progress very slowly as you are making very tiny updates to the weights in your network. However, if your learning rate is set too high, it can cause undesirable divergent behavior in your loss function

Example

A perceptron model from the dataset.

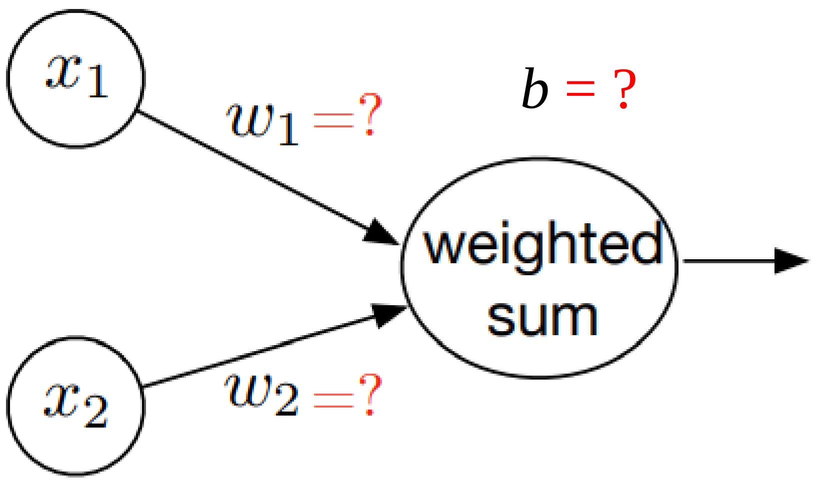
x_1	x_2	Out
0	0	0
0	1	1
1	0	1
1	1	1



Exercise

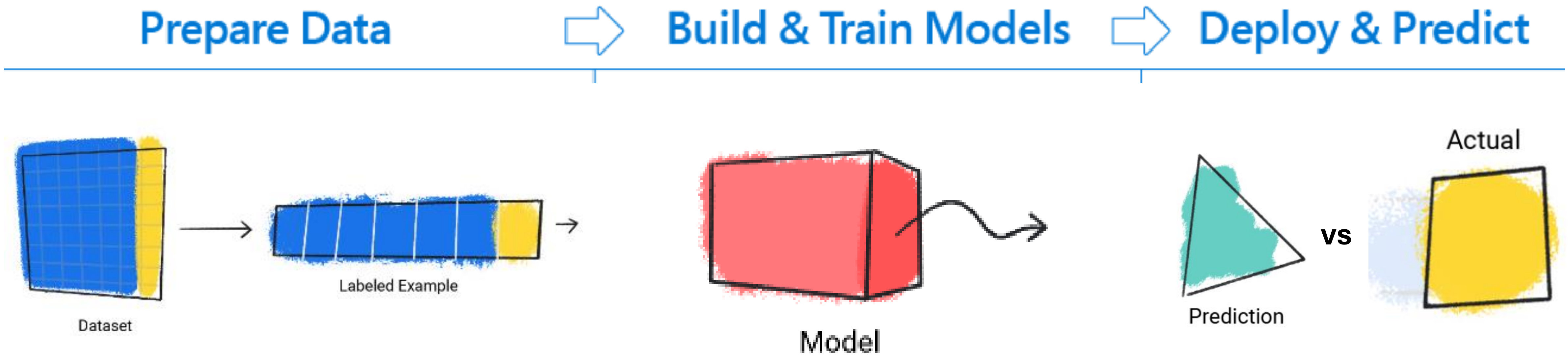
Build a perceptron model from the dataset.

x_1	x_2	Out
0	0	0
0	1	0
1	0	0
1	1	1



Build and Train Model

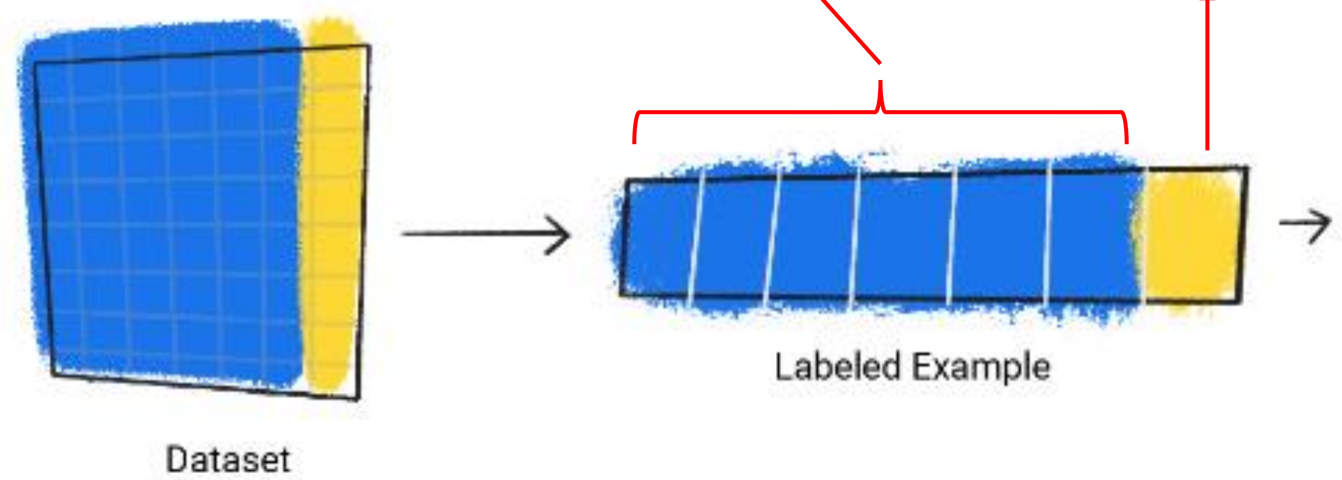
Steps in Implementing the Machine Learning Algorithm



Prepare Data

Predictors/
Features/
Inputs

Target/
Label/
Output



Predictors/
Features/Inputs

Target/
Label/Output

	Age	Amount	Class
1	67	1169	Good
2	22	5951	Bad
3	49	2096	Good
4	45	7882	Good
5	53	4870	Bad
6	35	9055	Good
7	53	2835	Good
8	35	6948	Good
9	61	3059	Good
10	28	5234	Bad
11	25	1295	Bad
12	24	4308	Bad
13	22	1567	Good
14	60	1199	Bad

Choose Class as a target of prediction

Prepare Data (R Script)

```r
# Load and preprocess the data
library(caret)
data("GermanCredit")
View(GermanCredit[, c("Age", "Amount", "Class")])
GermanCredit$Class <- factor(GermanCredit$Class, levels = c("Good", "Bad"))

# Data Preparation----
# Select only numeric predictors and the target
GermanCredit_subset <- GermanCredit[, c("Age", "Amount", "Class")]
GermanCredit_subset <- GermanCredit_subset[complete.cases(GermanCredit_subset), ]

# Normalize features
normalize <- function(x) (x - min(x)) / (max(x) - min(x))
GermanCredit_subset$Age <- normalize(GermanCredit_subset$Age)
GermanCredit_subset$Amount <- normalize(GermanCredit_subset$Amount)

# Split data into training and testing sets
set.seed(123)
train_index <- createDataPartition(GermanCredit_subset$Class, p = 0.8, list = FALSE)
GermanCredit_Train <- GermanCredit_subset[train_index, ]
GermanCredit_Test <- GermanCredit_subset[-train_index, ]

# Convert target variable to binary (Perceptron requires numeric output)
GermanCredit_Train$Class <- ifelse(GermanCredit_Train$Class == "Good", 1, 0)
GermanCredit_Test$Class <- ifelse(GermanCredit_Test$Class == "Good", 1, 0)
```

Choose the predictors and target

Normalization

Percentage of data training and testing

Build and Test KNN Model

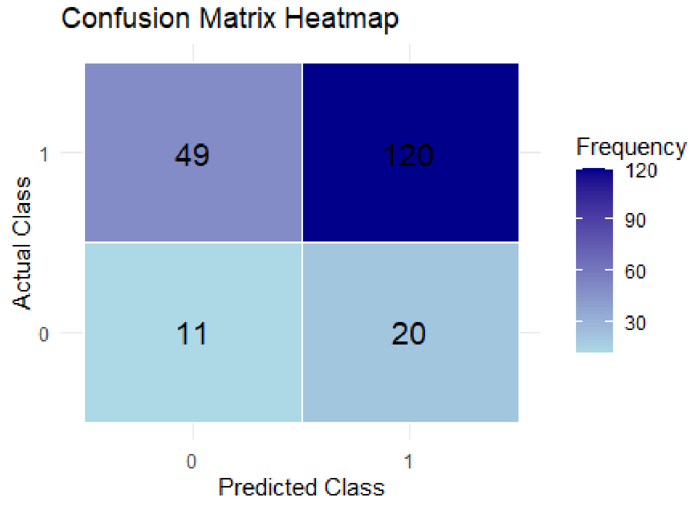
```r
# Train-Test----
# Load necessary library
library(class)

# Extract features and target variable
train_features <- GermanCredit_Train[, c("Age", "Amount")]
train_labels <- GermanCredit_Train$Class
test_features <- GermanCredit_Test[, c("Age", "Amount")]
test_labels <- GermanCredit_Test$Class


# Train and predict using KNN
knn_predictions <- knn(
  train = train_features,
  test = test_features,
  cl = train_labels,
  k = 5,
  p = 2  # 1:Manhattan, 2: Euclidan, inf:Chebyshev distance)
)

# Evaluate performance
confusion_matrix <- table(Predicted = knn_predictions, Actual = test_labels)
print(confusion_matrix)

# Calculate accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
cat("Accuracy:", accuracy, "\n")
```

Hyperparameter
k, p (distance)

```
> print(confusion_matrix)
         Actual
Predicted   0   1
        0  11  20
        1  49 120
> # Calculate accuracy
> accuracy <- sum(diag(confusion_matrix))
> cat("Accuracy:", accuracy, "\n")
Accuracy: 0.655
```

KNN Model Evaluation

```r
# Visualize heatmap----
# Load necessary libraries
library(ggplot2)
library(reshape2)

# Generate the confusion matrix (example using caret's confusionMatrix)
conf_matrix <- confusionMatrix(factor(knn_predictions), factor(GermanCredit_Test$Class))

# Convert confusion matrix to a data frame
conf_matrix_df <- as.data.frame(conf_matrix$table)

# Rename columns for clarity
colnames(conf_matrix_df) <- c("Actual", "Predicted", "Freq")

# Create a heatmap using ggplot2
ggplot(data = conf_matrix_df, aes(x = Predicted, y = Actual, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), color = "black", size = 5) + # Add text for frequency
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix Heatmap",
       x = "Predicted Class",
       y = "Actual Class",
       fill = "Frequency") +
  theme_minimal()
```



Exercise 1

- What will happen if we change the number of K? Will it change the accuracy?

- Change the number of K into:

(a) 3

(b) 7

(c) 13

(d) 19

```
# Train and predict using KNN
knn_predictions <- knn(
  train = train_features,
  test = test_features,
  cl = train_labels,
  k = 5,
  p = 2  # 1:Manhattan, 2: Euclidan, inf:Chebyshev distance)
)
```

Exercise 2

- What will happen if we change the type of distance? Will it change the accuracy?

- Change the type of distance, with change the number of p:

(a) p = 1, (Manhattan)

(b) p = 0.1 (Minkowski)

(c) p = 0.5 (Minkowski)

(d) p = 1.5 (Minkowski)

(e) p = 2.5 (Minkowski)

(f) p = inf (Chebyshev)

```
# Train and predict using KNN
knn_predictions <- knn(
  train = train_features,
  test = test_features,
  cl = train_labels,
  k = 5,
  p = 2  # 1:Manhattan, 2: Euclidan, inf:Chebyshev distance)
)
```

Build Perceptron Model

```r
# Train-Test----
X <- as.matrix(GermanCredit_Train[, c("Age", "Amount")])
y <- GermanCredit_Train$Class
X_test <- as.matrix(GermanCredit_Test[, c("Age", "Amount")])
y_test <- GermanCredit_Test$Class

# Perceptron Functions
act_func <- function(x) {
  ifelse(x >= 0, 1, 0)
}

train_perceptron <- function(X, y, lr = 0.1, epochs = 500) {
  weights <- runif(ncol(X))
  bias <- runif(1)

  for (epoch in 1:epochs) {
    for (i in 1:nrow(X)) {
      linear_output <- sum(X[i, ] * weights) + bias
      prediction <- act_func(linear_output)
      error <- y[i] - prediction
      weights <- weights + lr * error * X[i, ]
      bias <- bias + lr * error
    }
    if (epoch %% 50 == 0) {
      cat("Epoch:", epoch, "Weights:", weights, "Bias:", bias, "\n")
    }
  }
  list(weights = weights, bias = bias)
}
```

- Converts the selected predictors ("Age" and "Amount") into a matrix format.
- A perceptron model requires predictors in a numerical matrix form for matrix operations during training and predictions.

Step function

initializing the weights and bias for the perceptron model.

Test the Perceptron Model

```r
predict_perceptron <- function(model, X) {
  linear_output <- X %*% model$weights + model$bias
  act_func(linear_output)
}

# Test and Evaluate
set.seed(123)
model <- train_perceptron(X, y, lr = 0.1, epochs = 500)
perc_predictions <- predict_perceptron(model, X_test)

library(caret)
conf_matrix <- confusionMatrix(
  factor(perc_predictions, levels = c(0, 1)),
  factor(y_test, levels = c(0, 1))
)
print(conf_matrix)
```

prediction function for a perceptron model

$$\text{linear\_output} = X \cdot \mathbf{w} + b$$

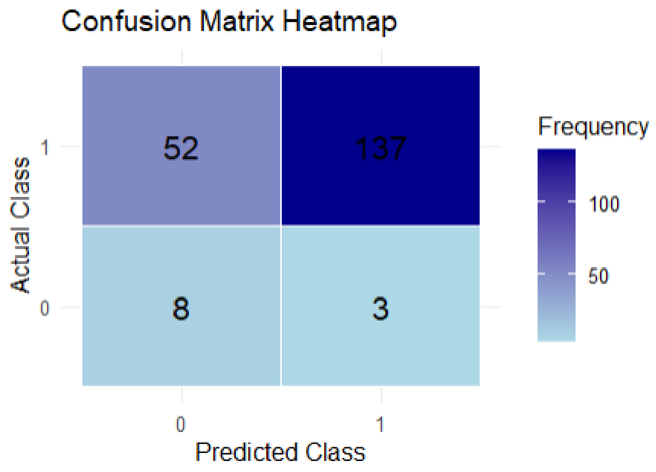
```
> model <- train_perceptron(X, y, lr = 0.1, epochs = 500)
Epoch: 50 Weights: 0.09114895 -0.06369222 Bias: 0.008976922
Epoch: 100 Weights: 0.1286489 -0.07241898 Bias: 0.008976922
Epoch: 150 Weights: 0.09114895 -0.0506241 Bias: 0.008976922
Epoch: 200 Weights: 0.07507752 -0.04848368 Bias: 0.008976922
Epoch: 250 Weights: 0.1090061 -0.05932885 Bias: 0.008976922
Epoch: 300 Weights: 0.1000775 -0.05983506 Bias: 0.008976922
Epoch: 350 Weights: 0.1036489 -0.05431069 Bias: 0.008976922
Epoch: 400 Weights: 0.07507752 -0.04848919 Bias: 0.008976922
Epoch: 450 Weights: 0.1090061 -0.05933435 Bias: 0.008976922
Epoch: 500 Weights: 0.1000775 -0.05984057 Bias: 0.008976922
```

- Optimizes the weights and bias using the training data
- The best model is:
  - W_1 = 0.1000775,
  - W_2 = -0.05984057, and
  - Bias (b) = 0.0089

```
> print(conf_matrix)
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0   8   3
         1  52 137

               Accuracy : 0.725
```

Perceptron Model Evaluation

```r
# Visualize heatmap----
library(reshape2)

# Generate the confusion matrix (example using caret's confusionMatrix)
conf_matrix <- confusionMatrix(factor(perc_predictions), factor(GermanCredit_Test$Class))

# Convert confusion matrix to a data frame
conf_matrix_df <- as.data.frame(conf_matrix$table)

# Rename columns for clarity
colnames(conf_matrix_df) <- c("Actual", "Predicted", "Freq")

# Create a heatmap using ggplot2
ggplot(data = conf_matrix_df, aes(x = Predicted, y = Actual, fill = Freq)) +
  geom_tile(color = "white") +
  geom_text(aes(label = Freq), color = "black", size = 5) + # Add text for frequency
  scale_fill_gradient(low = "lightblue", high = "darkblue") +
  labs(title = "Confusion Matrix Heatmap",
       x = "Predicted Class",
       y = "Actual Class",
       fill = "Frequency") +
  theme_minimal()
```



Exercise

- What will happen if we change the number of learning rate (lr)? and epoch?
- Will it change the accuracy?

```
# Test and Evaluate
set.seed(123)
model <- train_perceptron(X, y, lr = 0.1, epochs = 500)
perc_predictions <- predict_perceptron(model, X_test)
```

Hyperparameter Tuning

Hyperparameter General Formula

The goal of grid and random search is to find:

$$\mathbf{h}^* = \arg\max_{\mathbf{h} \in G} P(D_{\text{train}}, D_{\text{val}}, \mathbf{h})$$

P as the performance metric to be optimized (e.g., accuracy, RMSE)
D as the dataset used (split into training and validation subsets)
h is a combination of hyperparameter values
h* is the one that maximizes (or minimizes, for loss functions)
G is a Cartesian product of all hyperparameter grids, representing all possible combinations

Example for k-NN

For a k-NN model:

$$\mathbf{k}^* = \arg\max_{k \in G} P(D_{\text{train}}, D_{\text{val}}, k)$$

$G =$ {1,3,5,7,9,...}

For each k the k-NN model is trained and evaluated, and the k with the highest validation accuracy is selected.

Grid Search in KNN

```r
# Hyperparameter: Grid-Search to find the best k----
grid_search_knn <- function(train_features, train_labels, test_features, test_labels, k_values) {
  results <- data.frame(k = integer(), accuracy = numeric())

  for (k in k_values) {
    # Train and predict using KNN
    knn_predictions <- knn(
      train = train_features,
      test = test_features,
      cl = train_labels,
      k = k
    )
    # Evaluate accuracy
    accuracy <- mean(knn_predictions == test_labels)
    # Store results
    results <- rbind(results, data.frame(k = k, accuracy = accuracy))
  }
  # Return sorted results
  results[order(-results$accuracy), ]
}

# Define range of k values
k_values <- 1:30  # Test k values from 1 to 30

# Perform grid search
grid_results <- grid_search_knn(
  train_features = train_features,
  train_labels = train_labels,
  test_features = test_features,
  test_labels = test_labels,
  k_values = k_values
)
grid_results[order(grid_results$k), ]      #grid_results$k
```

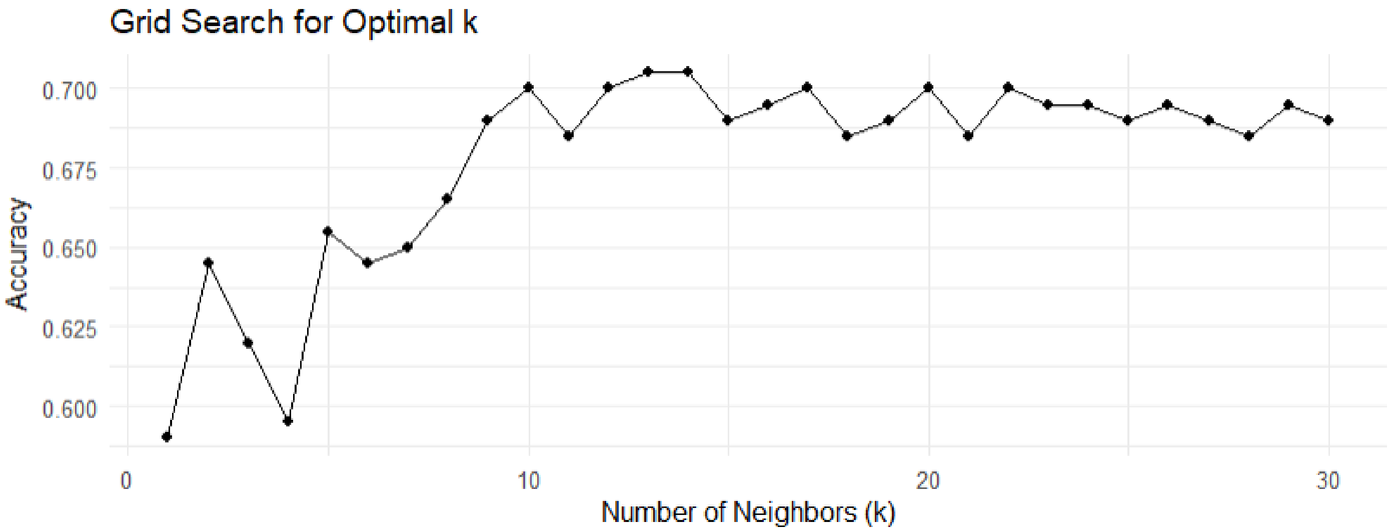
To find the best k

	k	accuracy
1	1	0.590
2	2	0.645
3	3	0.620
4	4	0.595
5	5	0.655
6	6	0.645
7	7	0.650
8	8	0.665
9	9	0.690
10	10	0.700
11	11	0.685
12	12	0.700
13	13	0.705
14	14	0.705
15	15	0.690
16	16	0.695
17	17	0.700
18	18	0.685
19	19	0.690
20	20	0.700
21	21	0.685
22	22	0.700
23	23	0.695
24	24	0.695
25	25	0.690
26	26	0.695
27	27	0.690
28	28	0.685
29	29	0.695
30	30	0.690

k =13 and k =14 is the best k

Grid Search in KNN: Optimal k

```r
# Best k
best_k <- grid_results[1, "k"]
cat("Best k:", best_k, "with accuracy:", grid_results[1, "accuracy"], "\n")

library(ggplot2)
ggplot(grid_results, aes(x = k, y = accuracy)) +
  geom_line() +
  geom_point() +
  labs(title = "Grid Search for Optimal k", x = "Number of Neighbors (k)", y = "Accuracy") +
  theme_minimal()
```



Random Search in KNN

```r
random_search_knn <- function(train_features, train_labels, test_features, test_labels, k_range, n_trials) {
  results <- data.frame(k = integer(), accuracy = numeric())

  for (i in 1:n_trials) {
    # Randomly sample k
    k <- sample(k_range, 1)  # Sample one k from the range

    # Train and predict using KNN
    knn_predictions <- knn(
      train = train_features,
      test = test_features,
      cl = train_labels,
      k = k
    )

    # Evaluate accuracy
    accuracy <- mean(knn_predictions == test_labels)

    # Store results
    results <- rbind(results, data.frame(k = k, accuracy = accuracy))
  }

  # Return sorted results
  results[order(-results$accuracy), ]  # Sort by accuracy in descending order
}

# Parameters for Random Search
k_range <- 1:30  # Range of k-values to search
n_trials <- 15   # Number of random trials
```

n_trials: The number of random k-values to sample and evaluate.

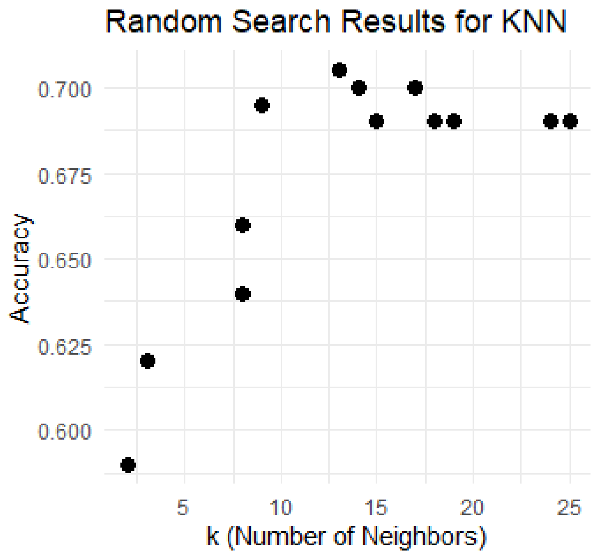
k_range: A range of possible values for k, the number of neighbors in the KNN algorithm.

Random Search in KNN: Optimal k

```
> print(random_results)
    k accuracy
3  13    0.705
2  17    0.700
10 14    0.700
8   9    0.695
13  9    0.695
1  25    0.690
6  15    0.690
7  18    0.690
12 24    0.690
14 19    0.690
9   8    0.660
5   8    0.640
4   3    0.620
15  3    0.620
11  2    0.590
```

```r
# Perform random search
random_results <- random_search_knn(
  train_features = train_features,
  train_labels = train_labels,
  test_features = test_features,
  test_labels = test_labels,
  k_range = k_range,
  n_trials = n_trials
)

# Display results
print(random_results)

random_results[order(-random_results$accuracy), ]        #random_results$k

# Best k
best_k <- random_results[1, "k"]
cat("Best k:", best_k, "with accuracy:", random_results[1, "accuracy"], "\n")

library(ggplot2)
ggplot(random_results, aes(x = k, y = accuracy)) +
  geom_point(size = 3) +
  labs(title = "Random Search Results for KNN",
       x = "k (Number of Neighbors)",
       y = "Accuracy") +
  theme_minimal()
```

k =13 is the best k



Grid Search in Perceptron

```r
# Hyperparameter: Grid-Search to find the best lr and epoch----
# Grid Search Function
grid_search_perceptron <- function(X_train, y_train, X_val, y_val, lr_values, epoch_values) {
  results <- data.frame(lr = numeric(), epochs = integer(), accuracy = numeric())

  for (lr in lr_values) {
    for (epochs in epoch_values) {
      # Train perceptron
      model <- train_perceptron(X_train, y_train, lr = lr, epochs = epochs)

      # Predict on validation set
      predictions <- predict_perceptron(model, X_val)

      # Evaluate performance
      accuracy <- mean(predictions == y_val)  # Accuracy metric

      # Store results
      results <- rbind(results, data.frame(lr = lr, epochs = epochs, accuracy = accuracy))
    }
  }

  # Return sorted results
  results[order(-results$accuracy), ]
}

# Define hyperparameter grid
lr_values <- c(0.01, 0.05, 0.1, 0.5, 1)  # Learning rate values to try
epoch_values <- c(100, 200, 500, 1000)   # Epoch values to try
```

A **grid search function** for hyperparameter tuning of a perceptron model.

lr: The learning rate used.

epochs: The number of epochs used.

lr_values: A vector of candidate learning rates.
epoch_values: A vector of candidate numbers of epochs.

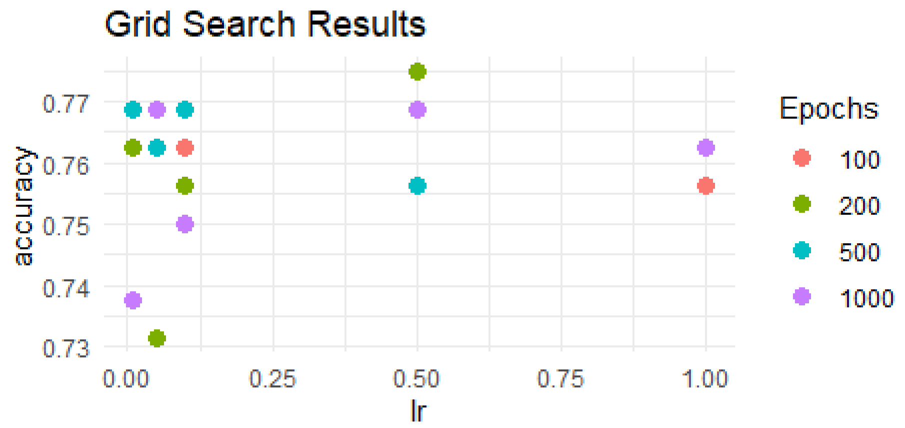
Grid Search in Perceptron: Optimal lr and epoch

```r
# Split data into training and validation sets
set.seed(123)
val_index <- createDataPartition(GermanCredit_Train$Class, p = 0.2, list = FALSE)
X_train <- as.matrix(GermanCredit_Train[-val_index, c("Age", "Amount")])
y_train <- GermanCredit_Train$Class[-val_index]
X_val <- as.matrix(GermanCredit_Train[val_index, c("Age", "Amount")])
y_val <- GermanCredit_Train$Class[val_index]

# Perform grid search
grid_results <- grid_search_perceptron(X_train,
                                       y_train,
                                       X_val,
                                       y_val,
                                       lr_values,
                                       epoch_values)

# Display best parameters
print(grid_results[1, ])  # Best combination

ggplot(grid_results, aes(x = lr, y = accuracy, color = as.factor(epochs))) +
  geom_point(size = 3) +
  labs(title = "Random Search Results", color = "Epochs") +
  theme_minimal()
```

- To find the best lr and epoch

```
Epoch: 550 Weights: 0.2780076 -1.193058 Bias: 0.4646659
Epoch: 600 Weights: 0.3137219 -1.229814 Bias: 0.4646659
Epoch: 650 Weights: 0.5280076 -1.516102 Bias: 0.4646659
Epoch: 700 Weights: 0.2601505 -1.240159 Bias: 0.4646659
Epoch: 750 Weights: 0.331579 -1.328472 Bias: 0.4646659
Epoch: 800 Weights: 0.4744362 -1.413043 Bias: 0.4646659
Epoch: 850 Weights: 0.2958648 -1.279776 Bias: 0.4646659
Epoch: 900 Weights: 0.3137219 -1.201752 Bias: 0.4646659
Epoch: 950 Weights: 0.331579 -1.329187 Bias: 0.4646659
Epoch: 1000 Weights: 0.2958648 -1.282637 Bias: 0.4646659
> # Display best parameters
> print(grid_results[1, ])  # Best combination
    lr epochs accuracy
14 0.5    200    0.775
```

- The best lr = 0.5 and epoch = 200



Random Search in Perceptron

```r
# Hyperparameter: Random-Search to find the best lr and epoch----
random_search_perceptron <- function(X_train, y_train, X_val, y_val, n_trials) {
  results <- data.frame(lr = numeric(), epochs = integer(), accuracy = numeric())

  for (trial in 1:n_trials) {
    # Randomly sample hyperparameters
    lr <- runif(1, min = 0.001, max = 1)  # Random learning rate
    epochs <- sample(100:1000, 1)         # Random number of epochs

    # Train perceptron
    model <- train_perceptron(X_train, y_train, lr = lr, epochs = epochs)

    # Predict on validation set
    predictions <- predict_perceptron(model, X_val)

    # Evaluate performance
    accuracy <- mean(predictions == y_val)  # Accuracy metric

    # Store results
    results <- rbind(results, data.frame(lr = lr,
                                         epochs = epochs,
                                         accuracy = accuracy))
  }

  # Return sorted results
  results[order(-results$accuracy), ]
}
```

A **Random search function** for hyperparameter tuning of a perceptron model.

- Iterates through a number of random trials **(n_trials)**
- Randomly samples a learning rate from a uniform distribution between 0.001 and 1
- Randomly selects an integer for the number of epochs from the range 100 to 1000.

lr: The learning rate used.

epochs: The number of epochs used.

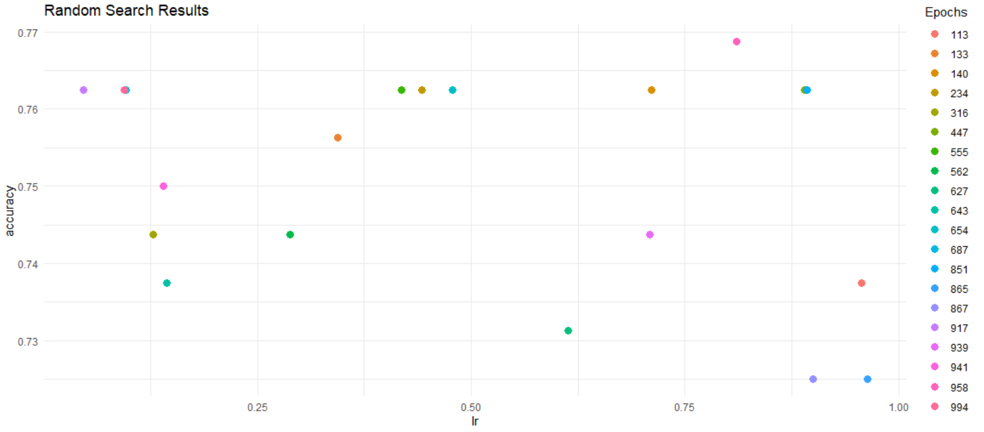
Random Search in Perceptron: Optimal lr and epoch

```r
# Split data into training and validation sets
set.seed(123)
val_index <- createDataPartition(GermanCredit_Train$Class,
                                 p = 0.2,
                                 list = FALSE)
X_train <- as.matrix(GermanCredit_Train[-val_index, c("Age", "Amount")])
y_train <- GermanCredit_Train$Class[-val_index]
X_val <- as.matrix(GermanCredit_Train[val_index, c("Age", "Amount")])
y_val <- GermanCredit_Train$Class[val_index]

# Perform random search
set.seed(123)
n_trials <- 20  # Number of random configurations to test
random_results <- random_search_perceptron(X_train,
                                           y_train,
                                           X_val,
                                           y_val,
                                           n_trials)

# Display best parameters
print(random_results[1, ])  # Best combination

ggplot(random_results, aes(x = lr, y = accuracy, color = as.factor(epochs))) +
  geom_point(size = 3) +
  labs(title = "Random Search Results", color = "Epochs") +
  theme_minimal()
```

- To find the best lr and epoch

```
Epoch: 550 Weights: 0.0322484 -0.116306 Bias: 0.04931937
Epoch: 600 Weights: 0.02887335 -0.1321342 Bias: 0.04931937
Epoch: 650 Weights: 0.0254983 -0.1291391 Bias: 0.04931937
Epoch: 700 Weights: 0.03393593 -0.1234142 Bias: 0.04931937
Epoch: 750 Weights: 0.03056088 -0.1204191 Bias: 0.04931937
Epoch: 800 Weights: 0.0187482 -0.1206582 Bias: 0.04931937
Epoch: 850 Weights: 0.01537315 -0.1176632 Bias: 0.04931937
Epoch: 900 Weights: 0.0187482 -0.1215422 Bias: 0.04931937
Epoch: 950 Weights: 0.03056088 -0.1205075 Bias: 0.04931937
> # Display best parameters
> print(random_results[1, ])  # Best combination
          lr epochs accuracy
14 0.8102543    958  0.76875
```

- The best lr = 0.8102543 and epoch = 958



Thank you