1. Differences between Hyperparameter and Parameter of a Machine Learning Model

Parameter: In machine learning, parameters are internal components of the model that are learned from the data during the training process. These include things like the weights of a linear model or the centroids in a clustering algorithm.

Hyperparameter: These are configurations set before the training process begins and are not learned from the data. Examples include the learning rate in gradient descent or the number of clusters in a clustering algorithm.

Two models to illustrate this:

Linear Regression:

Parameters: The coefficients (weights) for each feature, learned during the training process. Hyperparameters: Regularization strength (for Ridge or Lasso regression).

K-Means Clustering:

Parameters: The centroids of the clusters, which are updated during training. Hyperparameters: The number of clusters *k* k, which must be set prior to training and determines how many groups the algorithm will try to find.

2. 2. Elastic Net as LASSO and Ridge Regularizer

In [2]:
```
import numpy as np
from sklearn.linear_model import ElasticNet, Lasso, Ridge
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt

# Create sample data for regression
X, y = make_regression(n_samples=100, n_features=2, noise=0.1, random_state=42)

# ElasticNet with alpha=1 (Lasso behavior)
elasticnet_lasso = ElasticNet(alpha=1, l1_ratio=1).fit(X, y)  # l1_ratio=1 means Lasso
lasso = Lasso(alpha=1).fit(X, y)

# ElasticNet with alpha=0 (Ridge behavior)
elasticnet_ridge = ElasticNet(alpha=0.0001, l1_ratio=0).fit(X, y)  # l1_ratio=0 means Ridge
ridge = Ridge(alpha=1).fit(X, y)

# Print coefficients
print("ElasticNet (Lasso behavior) coefficients:", elasticnet_lasso.coef_)
print("Lasso coefficients:", lasso.coef_)
print("ElasticNet (Ridge behavior) coefficients:", elasticnet_ridge.coef_)
print("Ridge coefficients:", ridge.coef_)
```

```
ElasticNet (Lasso behavior) coefficients: [86.37920757 73.10200127]
Lasso coefficients: [86.37920757 73.10200127]
ElasticNet (Ridge behavior) coefficients: [87.70815134 74.07008897]
Ridge coefficients: [86.55443761 73.36678334]
```

The coefficients for ElasticNet (Lasso) and Lasso are identical. This is clear evidence that when set l1_ratio=1, ElasticNet behaves exactly like Lasso. Both models have the same coefficient values for both features, proving that ElasticNet can act as a Lasso regularizer.

Similarly, ElasticNet (Ridge) and Ridge show comparable, though not exactly identical, coefficients. The reason for the small discrepancy is due to slight differences in how ElasticNet combines penalties compared to Ridge, even when l1_ratio=0. However, the values are close, which proves that ElasticNet behaves much like Ridge when configured this way.

This makes Elastic Net a flexible model that can balance between LASSO's feature selection and Ridge's shrinkage, making it suitable for situations where you have a large number of correlated features.

3. Feature Importance Analysis for Predicting "Target" Using Two Different Approaches

In [9]:
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.cluster import KMeans
from sklearn.linear_model import ElasticNet
import numpy as np

# Load the dataset
df = pd.read_csv('Dataset4.csv')

# Drop non-numeric columns ('proto', 'service', 'target') before filling missing values
df_numeric = df.drop(columns=['target', 'proto', 'service'])

# Fill missing values with the column mean
df_filled = df_numeric.fillna(df_numeric.mean())

# Standardize the data for K-Means
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df_filled)

# Perform K-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X_scaled)

# Analyze centroids and calculate feature variance across clusters (for K-Means)
centroids = kmeans.cluster_centers_
feature_variance = np.var(centroids, axis=0)
print("Feature variance from K-Means: \n", feature_variance)

# Encode the 'target' column for ElasticNet
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(df['target'])  # Keep target column for encoding

# Set print options to limit decimals to 3 for better readability
np.set_printoptions(precision=3, suppress=True)

# Perform ElasticNet regression (Ensure proper scaling and random_state for reproducibility)
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
elastic_net.fit(X_scaled, y_encoded)

# Get feature importance from ElasticNet coefficients
elastic_net_feature_importance = np.abs(elastic_net.coef_)
print("\nFeature importance from ElasticNet:\n", elastic_net_feature_importance)
```

```
Feature variance from K-Means:
 [    1.063     0.008    10.013   348.628     0.029   455.444     0.014
     0.24      0.239     0.24      1.293    69.141     2.629     3.097
     0.026     2.348     2.888     1.227     2.796     0.689     0.208
     0.128     0.062     0.004     0.        0.014     0.014     0.948
     0.447    14.542     0.354     0.759     0.007     1.206     0.004
     0.827     1.248     0.094     0.752     0.161     0.252     0.811
     0.001     0.776    10.042     0.342     0.897     0.003     0.267
     0.011     0.183     0.203     0.001     0.774    10.033     0.535
     0.866     0.223     0.689     0.175     0.107     0.023     0.007
     0.002     7.372     0.002     0.005     0.149  5338.171 12154.678
826.106 11466.067 18879.835     0.762     0.757     0.019     0.769
     0.119     2.347     2.187     0.004]

Feature importance from ElasticNet:
 [0.    0.21  0.008 0.    0.    0.    0.    0.    0.205 0.
 0.04  0.    0.    0.    0.    0.535 0.    0.206 0.    0.    0.    0.166
 0.    0.07  0.    0.891 0.261 0.013 0.368 0.    0.096 0.    0.    0.
 0.204 0.    0.    0.    0.069 0.    0.014 0.03  0.    0.    0.    0.
 0.171 0.    0.    0.    0.013 0.    0.009 0.    0.    0.027 0.174 0.
 0.    0.067 0.    0.    0.    0.    0.    0.    0.    0.    0.
 0.    0.141 0.    0.    0.042 0.    0.    0.399 0.   ]
```

Similarities:

Identifying important features: Both methods aim to provide insights into the importance of features for a specific task: K-Means does it indirectly by analyzing the spread of feature values in clusters, and ElasticNet does it directly by evaluating how features contribute to the prediction of the target variable. Numerical importance: Both techniques rank features numerically based on their contribution to the clustering process (K-Means) or predictive modeling (ElasticNet).

Differences: Methodology:

K-Means is unsupervised, meaning it doesn't use the target variable for its analysis. The importance of a feature is inferred by how much the feature's values vary across the cluster centroids. ElasticNet is supervised and directly models the relationship between features and the target variable, assigning importance based on how much each feature contributes to the prediction. Interpretation:

In K-Means, feature importance is indicated by the variance across cluster centroids. Features with higher variance across clusters are more likely to be important in distinguishing the clusters. In ElasticNet, feature importance is determined by the magnitude of the coefficients. Larger coefficients indicate that a feature has a strong influence on the target variable. Feature Importance Scaling:

K-Means looks at the data without regard to any specific target, grouping based on inherent structures. ElasticNet focuses on the target variable, optimizing for its prediction, and may discard features that don't contribute significantly to the predictive performance.

4. Three supervised machine learning models to predict 'target'

In [10]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.3, random_state=42)

# Model 1 - Logistic Regression
logistic_model = LogisticRegression(max_iter=1000, random_state=42)
logistic_model.fit(X_train, y_train)
y_pred_logistic = logistic_model.predict(X_test)
accuracy_logistic = accuracy_score(y_test, y_pred_logistic)
print(f"Logistic Regression Accuracy: {accuracy_logistic:.3f}")

# Model 2 - Gradient Boosting Classifier
boosting_model = GradientBoostingClassifier(n_estimators=100, random_state=42)
boosting_model.fit(X_train, y_train)
y_pred_boosting = boosting_model.predict(X_test)
accuracy_boosting = accuracy_score(y_test, y_pred_boosting)
print(f"Gradient Boosting Accuracy: {accuracy_boosting:.3f}")

# Model 3 - Random Forest Classifier
forest_model = RandomForestClassifier(n_estimators=100, random_state=42)
forest_model.fit(X_train, y_train)
y_pred_forest = forest_model.predict(X_test)
accuracy_forest = accuracy_score(y_test, y_pred_forest)
print(f"Random Forest Accuracy: {accuracy_forest:.3f}")
```

```
Logistic Regression Accuracy: 0.987
Gradient Boosting Accuracy: 0.997
Random Forest Accuracy: 0.998
```

a. These results show high accuracy for all three models, with Random Forest and Gradient Boosting performing slightly better than Logistic Regression. Given the high test accuracies, overfitting is unlikely. However, ensuring there isn't a significant discrepancy between training and test accuracies is recommended to confirm this.

b. Logistic Regression: Fast and interpretable, suitable for linear relationships.

Gradient Boosting: Sequential learning to handle complex patterns, reduces bias effectively.

Random Forest: Builds multiple trees to reduce variance and is robust against overfitting, good for complex data sets.

c. Logistic Regression:

Hyperparameter: I set max_iter=1000 to ensure that the optimization algorithm has enough iterations to converge, especially when dealing with complex datasets. Reason: This ensures the logistic regression model has enough iterations to find the optimal coefficients.

Gradient Boosting:

Hyperparameter: n_estimators=100 was used, which means 100 trees were built sequentially. Reason: This number balances performance and computational cost. More trees could lead to better accuracy but at the cost of longer training time.

Random Forest:

Hyperparameter: n_estimators=100 specifies the number of trees in the forest. Reason: A reasonable number of trees helps reduce variance and improve model stability without causing excessive training time. The default number of trees works well for a balanced trade-off between accuracy and computation.

d. Random Forest is recommended for its performance and flexibility, making it ideal for your classification needs. For enhanced accuracy with consideration for training time, Gradient Boosting can be an alternative. Accuracy: Random Forest outperformed the other models with an accuracy of 0.998, which is the best performance among the three models. Robustness: Random Forest is less prone to overfitting than Gradient Boosting because of its bagging approach, which reduces variance by averaging multiple decision trees. Scalability: Random Forest is faster to train than Gradient Boosting due to its parallel nature, and it can handle large datasets effectively without requiring significant tuning. Flexibility: It can handle both linear and non-linear relationships in the data, making it suitable for a variety of real-world problems

5. Three ensemble models for predicting "target"

In [12]:
```
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier, BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Prepare the data (splitting into train and test sets)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.3, random_state=42)

# Model 1 - AdaBoost
adaboost_model = AdaBoostClassifier(algorithm='SAMME', n_estimators=100, random_state=42)
adaboost_model.fit(X_train, y_train)
y_pred_adaboost = adaboost_model.predict(X_test)
accuracy_adaboost = accuracy_score(y_test, y_pred_adaboost)
print(f"AdaBoost Accuracy: {accuracy_adaboost:.3f}")

# Model 2 - Boosting
boosting_model = GradientBoostingClassifier(n_estimators=100, random_state=42)
boosting_model.fit(X_train, y_train)
y_pred_boosting = boosting_model.predict(X_test)
accuracy_boosting = accuracy_score(y_test, y_pred_boosting)
print(f"Gradient Boosting Accuracy: {accuracy_boosting:.3f}")

#Model 3 - Bagging
bagging_model = BaggingClassifier(DecisionTreeClassifier(), n_estimators=100, random_state=42)
bagging_model.fit(X_train, y_train)
y_pred_bagging = bagging_model.predict(X_test)
accuracy_bagging = accuracy_score(y_test, y_pred_bagging)
print(f"Bagging Accuracy: {accuracy_bagging:.3f}")
```

```
AdaBoost Accuracy: 0.884
Gradient Boosting Accuracy: 0.997
Bagging Accuracy: 0.998
```

a. Ensemble models are used when you want to improve the predictive power and robustness of machine learning models. The models in Q4 (e.g., Logistic Regression) work well individually but may not capture all the complexities in the data. Ensembles like Bagging, AdaBoost, and Gradient Boosting combine multiple weak learners to: Reduce bias (in the case of Boosting). Reduce variance (in the case of Bagging). Increase accuracy, which is evident from the high performance of Bagging (0.998) and Gradient Boosting (0.997) in Q5.

b. Similarities: All three models (AdaBoost, Gradient Boosting, Bagging) are ensemble methods, meaning they combine multiple weak models to create a stronger model. All of them aim to improve accuracy by either focusing on reducing bias (Boosting) or variance (Bagging).

Differences: AdaBoost: Sequentially increases the weight of misclassified samples, focusing on reducing bias. It achieved an accuracy of 0.884, which is lower than the others, possibly due to its simplicity in adjusting weights. Gradient Boosting: Like AdaBoost, it builds models sequentially but corrects errors in a more refined way (using gradient descent). It performs better, with an accuracy of 0.997, since it captures complex patterns. Bagging: Builds models (decision trees) independently and averages their predictions to reduce variance. It performs the best with an accuracy of 0.998, likely due to its ability to handle noisy features and prevent overfitting.

c. Bagging is preferable when you have large datasets with high variance or noisy data. It's effective at reducing overfitting and performs well out of the box. Gradient Boosting is preferred when accuracy is critical, and you are willing to trade off some computational time for better performance. It is ideal for datasets with complex patterns. AdaBoost is more suitable when you need a fast, simple model that focuses on misclassified instances. However, it may not always outperform more complex ensemble methods like Gradient Boosting.

d. Performance Comparison:

Logistic Regression (Q4): 0.987

Gradient Boosting (Q4 & Q5): 0.997

Random Forest/Bagging (Q4 & Q5): 0.998

AdaBoost (Q5): 0.884

Model Complexity: Logistic Regression: Simple, fast, but may not handle complex relationships well. AdaBoost: Simple, focuses on misclassified instances but struggles with very complex data. Gradient Boosting: More complex, accurate, but slower due to its sequential nature. Bagging (Random Forest): Complex but computationally efficient (parallel trees) and performs the best in this case. Best Model: Based on performance and complexity, Bagging (Random Forest) is the best method with the highest accuracy (0.998) and strong robustness. It is less prone to overfitting, making it a reliable choice for a variety of datasets.

e. Yes, ensemble models can be built using classifiers other than decision trees. For example:

In [18]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.svm import SVC

# Bagging using SVC as the base estimator
bagging_svm = BaggingClassifier(estimator=SVC(), n_estimators=10, random_state=42)
bagging_svm.fit(X_train, y_train)
y_pred_bagging_svm = bagging_svm.predict(X_test)
accuracy_bagging_svm = accuracy_score(y_test, y_pred_bagging_svm)
print(f"Bagging with SVC Accuracy: {accuracy_bagging_svm:.3f}")
```

```
Bagging with SVC Accuracy: 0.991
```

This is a Support Vector Machines (SVM) as the base estimator in Bagging. In this examples, the base learners are SVM, demonstrating that ensemble methods are not limited to decision trees.