CSCI-B:559: Efficient AI

Indiana University – Indianapolis

Project Part 1 of 2

Project on Feature Ranking, Efficient Classifier Building based on Information Gain & Efficient Neural Network Building & Feature Contribution Analysis

Name: Kasa Yeshwant

Table of Contents

1.INTRODUCTION

1.1. Project Overview and Objectives

This project explores fundamental concepts and techniques in machine learning (ML) and model optimization by focusing on three key tasks: **feature ranking, decision tree construction, and neural network classification with feature selection**. These tasks aim to develop a structured approach to building, analyzing, and optimizing ML models for improved interpretability, efficiency, and performance. The primary goal of the project is to understand how feature importance influences model predictions, the structure of a decision tree model, and the impact of feature selection on neural network performance. By investigating these aspects, this project provides insights into building efficient, optimized models tailored to data characteristics and problem requirements.

1.2. Summary of Key Tasks

1. **Feature Ranking**:
    - In the first task, feature ranking is performed using **Information Gain**, a metric derived from entropy. Information Gain measures the reduction in uncertainty provided by a feature regarding the target variable. By calculating Information Gain for each feature, the most relevant features are identified and ranked, helping to understand which attributes contribute most to predictions. This ranking process aids in reducing model complexity by focusing on the most informative features, which is crucial for model interpretability and performance.
2. **Decision Tree Based on Information Gain**:
    - The second task builds a **Decision Tree** using Information Gain as the criterion for splitting nodes. A decision tree is a hierarchical model that partitions the data based on feature values to make predictions, with each branch representing a decision rule and each leaf representing an outcome. Starting from the root node, the tree is built by recursively selecting features that provide the highest Information Gain, thus creating a model that makes decisions based on the most informative attributes. This task provides a practical understanding of how decision trees use feature importance to structure data and deliver predictions, allowing for clear, interpretable decision-making.

3. **Neural Network with Backward Feature Selection**:
    - In the third task, a **Neural Network** is trained for classification, followed by **Backward Feature Selection** to analyze feature impact. Backward feature selection involves iteratively removing one feature at a time, retraining the model, and observing the resulting accuracy. This technique reveals how each feature contributes to model performance, aiding in identifying features that can be excluded without significant accuracy loss. Such insights are essential for creating streamlined, efficient models that perform well without unnecessary complexity. This task not only demonstrates neural network training and evaluation but also shows the application of feature selection to optimize model performance.

1.3. Importance of the Project in Understanding Machine Learning and Model Optimization

This project plays a critical role in enhancing understanding of machine learning model construction, interpretability, and optimization. **Feature selection and ranking** are fundamental in ML because they help identify which data attributes are most relevant for making predictions, thereby improving both model accuracy and interpretability. By prioritizing features, we can simplify models, reduce computational overhead, and make models more efficient without compromising on accuracy.

The **decision tree task** exemplifies the concept of interpretability, as it builds a model with a clear structure based on feature importance. Decision trees make predictions in a logical, human-understandable way, allowing users to trace decisions back to specific features. This interpretability is invaluable in applications where transparent decision-making is required, such as in healthcare or finance.

The **neural network with feature selection** task introduces students to the complexities of deep learning while demonstrating the importance of model optimization. Neural networks are powerful but resource-intensive models, and feature selection provides a pathway to make them more efficient. By implementing backward selection, the project highlights how models can be streamlined by removing redundant features, leading to faster inference times and reduced memory usage. This is particularly relevant in edge and mobile applications, where computational resources are limited.

Overall, this project integrates foundational ML concepts with practical model optimization techniques. It enhances understanding of how to balance model complexity with efficiency, teaches methods for improving interpretability, and reinforces the importance of feature selection in optimizing ML models. These skills are essential for building robust, scalable, and efficient models in various real-world applications.

2. DATA PREPARATION

In this project, we work with two distinct datasets: a weather dataset with categorical attributes and the well-known **Iris dataset**. Each dataset has unique characteristics that require different preprocessing steps to prepare the data for feature ranking, decision tree construction, and neural network classification tasks. Proper data preparation is essential for ensuring model accuracy and performance, as it minimizes noise, handles inconsistencies, and standardizes data formats.

2.1. Iris Dataset

The Iris dataset is a well-established dataset in machine learning, containing measurements for three classes of the Iris flower species: Setosa, Versicolor, and Virginica. Each sample in the dataset includes four numerical features related to the flower's morphology, making it an excellent dataset for classification tasks.

- **Features:**
    - **Sepal Length:** Measured in centimeters.
    - **Sepal Width:** Measured in centimeters.
    - **Petal Length:** Measured in centimeters.
    - **Petal Width:** Measured in centimeters.
- **Target Variable:**
    - The target class represents the Iris species, which has three categories: Setosa, Versicolor, and Virginica.

2.2. Data Preprocessing Steps

1. **Handling Missing Values:**
    - The Iris dataset does not contain missing values, but as a best practice, we verify this before proceeding. If missing values were present, appropriate methods like mean imputation (for numerical data) or median imputation could be applied.
2. **Encoding Categorical Data:**
    - Label Encoding is applied to convert each category into a numerical value (e.g., Setosa = 0, Versicolor = 1, Virginica = 2). This encoding is suitable for multiclass classification and is directly compatible with both decision tree models and neural networks.
3. **Scaling Features:**
    - Standardization is applied to the four features in the Iris dataset to ensure they all have a mean of zero and a standard deviation of one. Standardized data improves

model convergence during neural network training and stabilizes gradient updates, resulting in more effective and consistent learning.

4. **Splitting Data:**
    - The dataset is split into training and testing sets to evaluate model performance on unseen samples.
    - Given the size of the Iris dataset (150 samples), an 80-20 split is applied, with 80% of the samples used for training and 20% for testing. This approach ensures sufficient data for the model to learn while maintaining a separate test set for unbiased performance evaluation.

2.3. Importance of Data Preparation

Data preparation is a foundational step in building accurate and reliable machine learning models. Through data cleaning, encoding, scaling, and splitting, the datasets are transformed into structured, standardized forms that models can effectively interpret and learn from. For the weather dataset, encoding categorical features ensures compatibility with algorithms, while splitting the data enables fair evaluation of model generalization. In the Iris dataset, encoding, standardization, and data splitting prepare it for complex neural network training while maintaining numerical consistency across features. Overall, data preparation minimizes noise, handles inconsistencies, and creates a solid foundation for model training, ultimately enhancing model performance and interpretability.

3.TASK-1:FEATURE RANKING

Feature ranking is a critical step in many machine learning tasks, as it helps to identify the most informative features for predicting the target variable. In this task, we use **Information Gain** to rank features based on their contribution to reducing uncertainty about the target variable. Information Gain is calculated using **Entropy**, a measure of uncertainty or disorder within a dataset. By calculating Information Gain for each feature, we can determine which features are most valuable for predictive modeling.

3.1. Entropy

Entropy is a measure of uncertainty or randomness within a set of data. In the context of machine learning, it represents the degree of impurity in a set of labels. If all instances in a dataset belong to the same class, the entropy is zero because there is no uncertainty. However, if instances are evenly split among different classes, the entropy is high due to the high level of unpredictability in the class label.

- **Formula for Entropy**:

  $H = - \sum (p(i) * \log_2 (p(i)))$

  Where:

  - P_i is the probability of each outcome in the target variable
  - H(Y) is high when the outcomes are evenly split (high uncertainty) & low when one outcome is more common (low uncertainty)

  So, first we calculate the overall entropy of the target variable (Class) to understand it's initial uncertainty.

3.2. Information Gain

Information Gain quantifies the reduction in entropy achieved by splitting a dataset based on a particular feature. It represents how much "information" or certainty we gain about the target variable when we know the value of that feature. A high Information Gain means that splitting the data on that feature provides significant insight into the target variable.

- **Formula for Information Gain**:

  Information Gain=Entropy(Parent)−Weighted Average Entropy(Children)

  Where:

  - H(Y) = Entropy of the parent/ entropy og the target variable before considering the feature.
  - H(Y|X) is the conditional entropy of Y given the feature X, which is the remaining uncertainty in Y after knowing X.

  The feature with the highest information gain is the most informative, as it gives the greatest reduction in uncertainty.

3.2.2.Steps to calculate Information Gain

To calculate the information gain for each feature in the dataset:

1. **Calculate Overall Entropy H(Y):** Compute the entropy of the target variable Y using it's distribution of Yes & No outcomes.
2. **Calculate Conditional Entropy H(Y|X) for Each Feature:** For each feature, split the dataset by it's unique values. Calculate entropy for the target variable in each subset, then compute the weighted average.
3. **Calculate Information Gain IG(Y|X):** Subtract the conditional entropy from the overall entropy: IG(Y|X) = H(Y) – H(Y|X).

4. **Rank Features by Information Gain:** Repeat these calculations for each feature & rank them by their information gain. Higher values indicate more informative features.

By calculating the Information Gain for each feature, we identify the most informative attributes in the dataset. In this example, the feature with the highest Information Gain will contribute most to reducing uncertainty in predicting the target variable, making it an essential factor in feature ranking. Feature ranking using Information Gain is particularly valuable in decision tree algorithms, where it helps create a clear, interpretable model by selecting the best feature at each decision node. This methodology provides insights into data, reduces dimensionality, and optimizes model performance by focusing on features that add the most predictive power.

3.3.Code:

```
import numpy as np
import pandas as pd

# Dataset
data = {
    'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rain', 'Rain', 'Rain',
'Overcast', 'Sunny', 'Sunny', 'Rain', 'Sunny', 'Overcast', 'Overcast',
'Rain'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool',
'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal',
'Normal', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],
    'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong',
'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Strong'],
    'Class': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes',
'Yes', 'Yes', 'Yes', 'Yes', 'No']
}
df = pd.DataFrame(data)

# Function to calculate entropy
def entropy(labels):
    values, counts = np.unique(labels, return_counts=True)
    probabilities = counts / len(labels)
    return -np.sum(probabilities * np.log2(probabilities))

# Function to calculate information gain
def information_gain(df, feature, target='Class'):
    total_entropy = entropy(df[target])
    values, counts = np.unique(df[feature], return_counts=True)
```

```
    weighted_entropy = sum((counts[i] / np.sum(counts)) *
entropy(df[df[feature] == values[i]][target]) for i in range(len(values)))
    return total_entropy - weighted_entropy

# Calculating information gain for each feature
features = ['Outlook', 'Temperature', 'Humidity', 'Wind']
info_gains = {feature: information_gain(df, feature) for feature in
features}
sorted_info_gains = sorted(info_gains.items(), key=lambda x: x[1],
reverse=True)

print("Information Gain for each feature:")
for feature, gain in sorted_info_gains:
    print(f"{feature}: {gain:.4f}")
```

3.4.Results:

```
Information Gain for each feature:
Outlook: 0.2467
Humidity: 0.1518
Wind: 0.0481
Temperature: 0.0292
```

The results of the Information Gain calculations for each feature are as follows:

- **Outlook** has the highest Information Gain (0.2467), indicating it is the most informative feature for predicting the target variable. This feature significantly reduces uncertainty when used to split the data, making it the best initial choice for a decision tree split.
- **Humidity** has a moderate Information Gain (0.1518). While it contributes valuable information, it is less effective than Outlook. Humidity might be a suitable secondary feature when building a decision tree.
- **Wind** and **Temperature** have relatively low Information Gains (0.0481 and 0.0292, respectively), indicating they contribute the least to reducing uncertainty in the target classification. These features are less informative and may not be prioritized in the tree structure.

Insights:

- **Feature Selection**: The calculated Information Gain values indicate which features are most valuable for distinguishing between classes. Outlook, being the highest-ranked feature, will likely form the root node of a decision tree, followed by Humidity.

- **Model Efficiency**: By identifying and prioritizing features with higher Information Gain, we can potentially reduce model complexity and focus on features that offer the most predictive power.
- **Data Interpretability**: Understanding which features contribute most to the model aids interpretability, as we know which variables play the biggest role in decisions. For example, in this dataset, the weather outlook has the largest impact on deciding whether to play or not.

4.TASK-2:DECISION TREE BASED ON INFORMATION GAIN

4.1.Methodology

A **decision tree** is a supervised learning model that represents data decisions in a tree-like structure. Each node in the tree represents a feature (or attribute) decision, each branch represents a possible outcome of the feature, and each leaf node represents a class label or the decision outcome. The decision tree algorithm is especially effective for classification tasks because it provides clear, interpretable decisions based on feature values, following a series of splits that aim to maximize class purity.

4.2. Decision Tree Algorithm and Node Selection Using Information Gain

The goal of the decision tree algorithm is to split the dataset at each node to maximize the homogeneity (purity) of the classes in each subset. **Information Gain** is used as the criterion for selecting the best feature to split at each node. It measures how much "information" or "certainty" about the target class increases when a feature is used to partition the data.

- **Information Gain (IG)**: The feature with the highest Information Gain is selected as the splitting criterion at each node. Information Gain is calculated by finding the difference between the entropy of the parent node and the weighted entropy of the child nodes. This process maximizes purity at each split, ensuring that the data at each branch becomes progressively more homogeneous, making the tree structure more interpretable.

The decision tree algorithm proceeds by recursively selecting features that yield the highest Information Gain until the data is either fully separated by class (pure) or further splits no longer add value.

4.3. Recursive Node Splitting, Branching, and Leaf Node Creation

The decision tree construction involves three main steps: **recursive node splitting**, **recursive branching**, and **leaf node creation**.

1. **Recursive Node Splitting**:
    - At the root node, the feature with the highest Information Gain is selected to split the data.

- For each unique value of the selected feature, a subset of data is created, and the process is recursively repeated for each subset.
- For example, in our decision tree, **Outlook** is chosen as the root node since it has the highest Information Gain. The dataset is then split into three subsets based on the values of Outlook: **Sunny**, **Overcast**, and **Rain**.

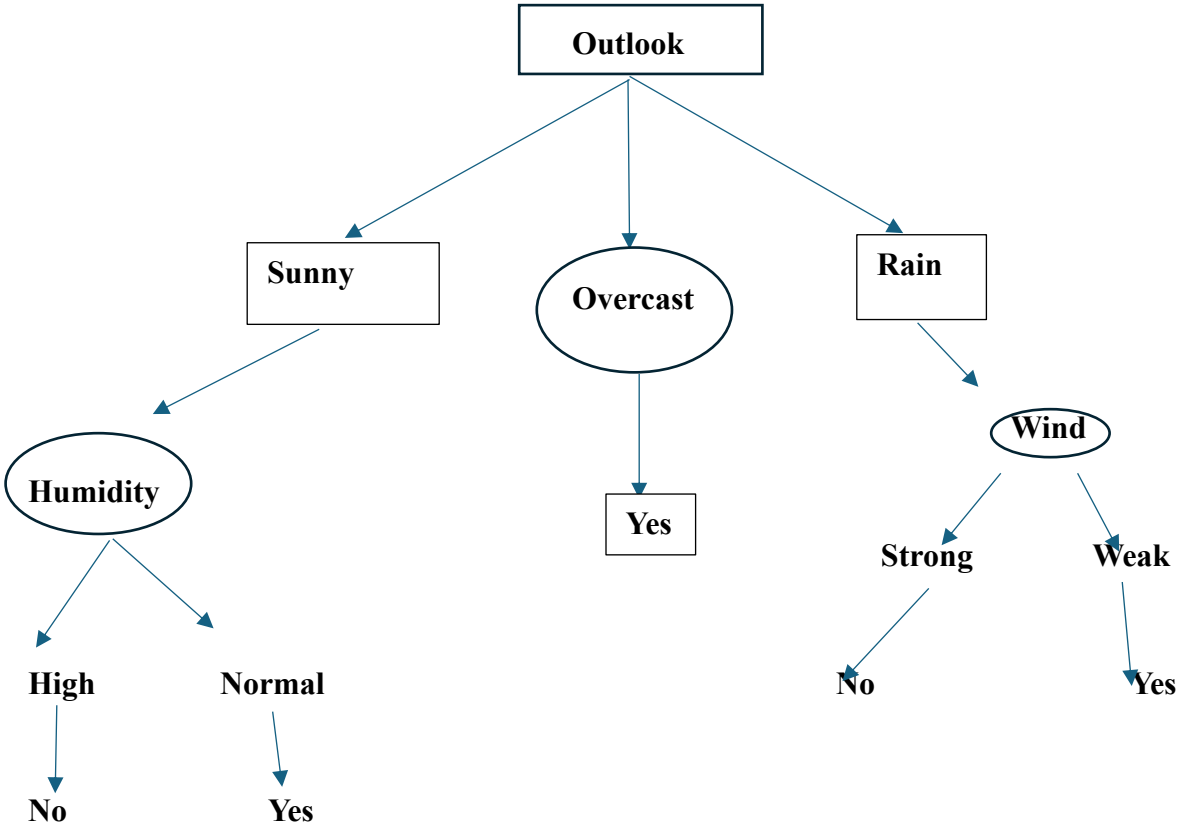
2. **Recursive Branching**:
   - Each subset created by the split at the root node is then further evaluated. For each subset, the Information Gain for remaining features is calculated, and the feature with the highest gain is chosen for the next split.
   - This process continues recursively, where each subset becomes a branch that is further split based on the feature with the highest Information Gain in that subset.
   - In our example, the **Rain** branch is further split using **Wind** (the feature with the highest Information Gain within this subset), resulting in two branches: **Strong** and **Weak**.
3. **Leaf Node Creation**:
   - A node becomes a **leaf node** when it reaches a point where it cannot be split further or when all samples within the node belong to the same class (pure subset).
   - Leaf nodes represent the final decision in the tree, associating a class label with each possible path from the root to a leaf.
   - In the **Overcast** branch, for example, all instances belong to the "Yes" class, creating a pure subset that becomes a leaf node with a class distribution of {'Yes': 4}.

4.4.Code:

```
from collections import Counter

# Function to create a decision tree
class Node:
    def __init__(self, feature=None, value=None, results=None,
children=None):
        self.feature = feature
        self.value = value
        self.results = results  # Class distribution at this node
        self.children = children  # Sub-nodes

def build_tree(df, features, target='Class'):
```

```
    # If all instances have the same class, return a leaf node
    if len(np.unique(df[target])) == 1:
        return Node(results=dict(Counter(df[target])))

    # If no features left to split, return a leaf node
    if len(features) == 0:
        return Node(results=dict(Counter(df[target])))

    # Choose the best feature based on information gain
    info_gains = {feature: information_gain(df, feature) for feature in
features}
    best_feature = max(info_gains, key=info_gains.get)

    # Create root node for the best feature
    tree = Node(feature=best_feature)
    tree.children = {}

    # Split dataset by best feature and create sub-nodes
    for value in np.unique(df[best_feature]):
        sub_data = df[df[best_feature] ==
value].drop(columns=[best_feature])
        tree.children[value] = build_tree(sub_data, [f for f in features
if f != best_feature], target)

    return tree

# Build and print the decision tree
decision_tree = build_tree(df, features)

# Recursive function to print the tree
def print_tree(node, indent=""):
    if node.results:
        print(indent + "Class distribution:", node.results)
    else:
        print(indent + f"{node.feature}")
        for value, child in node.children.items():
            print(indent + f"--> {value}:")
            print_tree(child, indent + "    ")

print("Decision Tree Structure:")
print_tree(decision_tree)
```

4.5. Results:



Insights and Explanation of Decision-Making Process

The decision tree structure produced by Information Gain-based splitting provides several important insights into how the model makes decisions and improves classification accuracy:

Feature Hierarchy and Decision Pathways:

- The root node, **Outlook**, serves as the initial decision point, as it provides the highest Information Gain. This feature effectively divides the dataset based on general weather conditions, a critical factor influencing the target class.
- Each path from the root node to a leaf node represents a unique combination of feature values leading to a specific class prediction, making the model highly interpretable.

2. **Class Separation and Purity in Leaf Nodes**:

- The tree's structure maximizes purity in each branch by progressively filtering data based on high-information features.
- For instance, the **Overcast** branch reaches a leaf node with a pure subset (all instances labeled as "Yes"), indicating high confidence in the classification for that pathway.

3. **Improved Classification Accuracy**:
   - By prioritizing features with high Information Gain, the tree achieves a sequence of decisions that progressively narrows down to accurate class predictions. Each split effectively reduces uncertainty in the target classification, leading to branches with minimal class mixing.
   - For example, in the **Sunny** branch, splitting on **Humidity** helps distinguish between "Yes" and "No" classes, improving classification accuracy by refining the data subset.
4. **Decision Tree Interpretability**:
   - Decision trees are inherently interpretable due to their hierarchical structure, which allows for easy tracing of each decision path. This interpretability is beneficial in applications where understanding the reasoning behind a model's decision is essential.
   - In this example, each branch decision (e.g., "If Outlook is Sunny and Humidity is High, then No") provides clear and logical conditions for classifying instances, making the model transparent and user-friendly.
5. **Data-Driven Decision-Making**:
   - The tree's structure is guided entirely by the data, with each split based on Information Gain calculations. This data-driven approach ensures that the model is tailored to the patterns present in the dataset, resulting in a highly customized and accurate classification model.

5. TASK-3: NEURAL NETWORKS & BACKWARD SEARCH FOR FEATURE SELECTION

In this task, a neural network model is built and trained to classify instances in a dataset. The focus is on understanding the contribution of each feature to the model's performance. To achieve this, we use a **backward search wrapper method** for feature selection, a systematic approach that iteratively removes features to determine their impact on the model's accuracy. This technique enables us to rank features based on their importance and refine the model by retaining only the most impactful features.

5.1.Neural Network Architecture for Classification

A **neural network** is a layered architecture designed to mimic the functioning of the human brain. It consists of multiple interconnected nodes (neurons) that transform input data through a series of computations to produce a desired output. For this classification task, the neural network has three main types of layers: **input layer**, **hidden layers**, and **output layer**. Each layer plays a distinct role in processing the data and enhancing the model's learning capability.

1. **Input Layer**:
    - The input layer receives the raw data (features) and feeds it into the network. Each feature in the dataset corresponds to one neuron in the input layer.
    - For example, if the dataset has four features, the input layer will have four neurons, each one representing an individual feature (e.g., sepal length, sepal width, petal length, and petal width in the Iris dataset).
    - The input layer does not perform any transformations; it simply acts as a conduit for passing data to the hidden layers.
2. **Hidden Layers**:
    - The hidden layers are the computational core of the neural network, where features are combined and transformed to learn complex patterns in the data.
    - In this task, the neural network is constructed with **two hidden layers**, each consisting of 10 neurons. These layers are fully connected, meaning each neuron in one layer is connected to every neuron in the subsequent layer.
    - Each hidden layer uses a **ReLU (Rectified Linear Unit)** activation function, which introduces non-linearity into the model. ReLU activation ensures that the network can learn and represent complex patterns and relationships within the data.
    - The hidden layers extract abstract features by applying weights to the inputs and transforming them through activation functions. These transformations allow the network to learn representations that are essential for distinguishing between classes.
3. **Output Layer**:
    - The output layer produces the final prediction. In this classification task, the output layer has **three neurons**, each corresponding to one of the three classes in the dataset.
    - A **softmax activation function** is applied in the output layer, which converts the output values into probabilities that sum to one. This makes it suitable for multi-

class classification, as each neuron's output can be interpreted as the probability of the sample belonging to a specific class.

- The class with the highest probability is selected as the model's predicted class.

5.2.Backward Search Wrapper Method for Feature Selection

Feature selection is a critical step in optimizing machine learning models. By selecting the most relevant features, we can reduce model complexity, improve interpretability, and often achieve better performance. The **backward search wrapper method** is a systematic approach to feature selection that evaluates the model's performance after removing one feature at a time. This method allows us to rank features based on their impact on classification accuracy, thereby identifying which features contribute most to the model's performance.

5.2.1. Implementation of Backward Search Wrapper Method for Feature Selection

The backward search wrapper method was implemented as follows:

1. **Initial Training with All Features**:
    - The neural network was first trained with all features in the dataset. The resulting accuracy was recorded as a benchmark to evaluate the impact of subsequent feature removals.
2. **Iterative Feature Removal and Retraining**:
    - For each feature in the dataset:
        - The feature was temporarily removed from the dataset.
        - The neural network model was retrained using the remaining features.
        - The accuracy of the model was calculated and recorded.
    - This process was repeated for each feature, with the accuracy values after each removal compared to the initial accuracy.

5.3.Code:

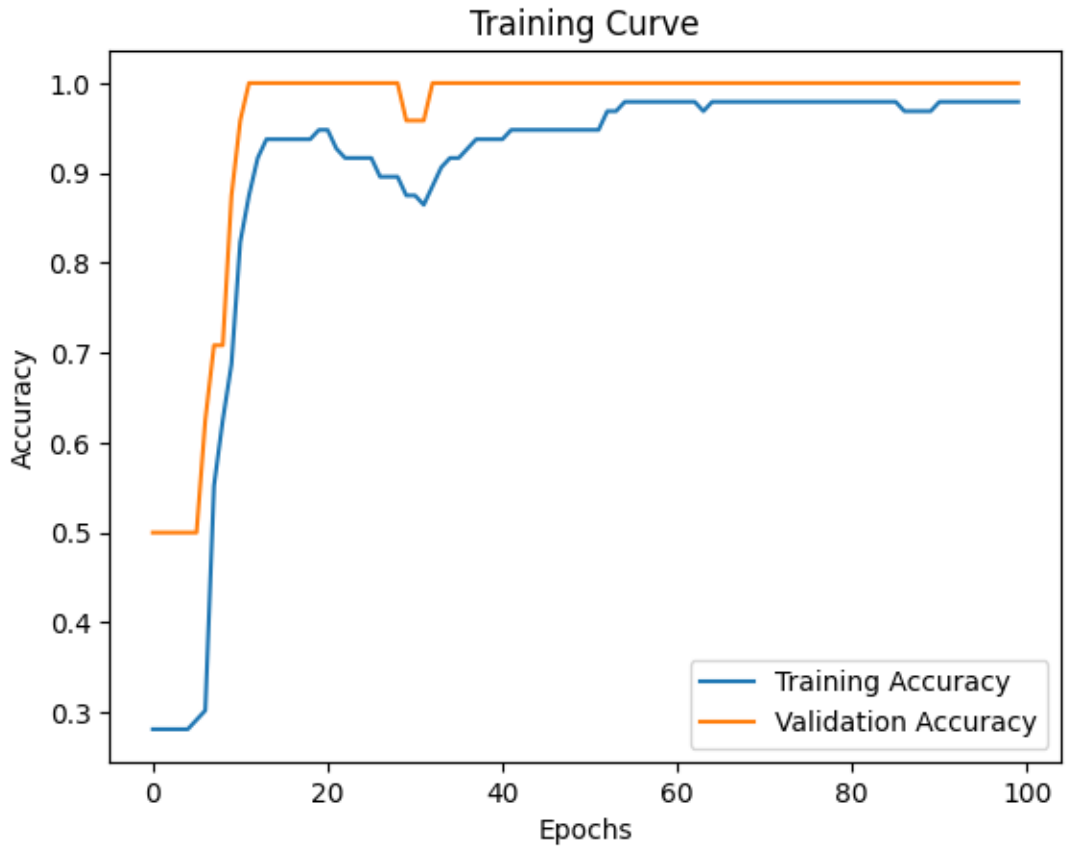
Iris Dataset - Efficient Neural Network Building and Feature Contribution Analysis

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt

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

# Separate features and target
X = df.drop(columns=['species'])
y = LabelEncoder().fit_transform(df['species'])  # Encode species as
integers

# Split the dataset into training and testing sets (80%:20% split)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Building & Training Neural Network

```
# Define the neural network model
def build_model(input_shape):
    model = Sequential([
        Dense(10, activation='relu', input_shape=(input_shape,)),
        Dense(10, activation='relu'),
        Dense(3, activation='softmax')
    ])
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# Build and train the model
model = build_model(X_train.shape[1])
history = model.fit(X_train, y_train, epochs=100, validation_split=0.2,
verbose=0)

# Plot the training curve
plt.plot(history.history['accuracy'], label='Training Accuracy')
```

```
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Training Curve")
plt.show()
```



Evaluate the Model

```
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_accuracy:.4f}")
```

Feature Selection using Backward Search

```
def backward_search(X, y):
    features = list(X.columns)
    results = []

    for i in range(len(features)):
```

```
        # Drop one feature at a time
        X_temp = X.drop(columns=[features[i]])
        X_train_temp, X_test_temp, y_train_temp, y_test_temp =
train_test_split(X_temp, y, test_size=0.2, random_state=42)

        # Build and evaluate the model
        model = build_model(X_train_temp.shape[1])
        model.fit(X_train_temp, y_train_temp, epochs=100,
validation_split=0.2, verbose=0)
        test_loss, test_accuracy = model.evaluate(X_test_temp,
y_test_temp, verbose=0)

        # Store the results
        results.append((features[i], test_accuracy))
        print(f"Removed Feature: {features[i]}, Test Accuracy:
{test_accuracy:.4f}")

    return results

# Perform backward search
backward_search_results = backward_search(X, y)

# Sort and display the feature removal impact on accuracy
sorted_results = sorted(backward_search_results, key=lambda x: x[1],
reverse=True)
print("Feature Contribution Ranking (higher accuracy after removal means
less contribution):")
for feature, accuracy in sorted_results:
    print(f"Feature Removed: {feature}, Accuracy: {accuracy:.4f}")
```

Results:

Feature Contribution Ranking (higher accuracy after removal means less contribution):

Feature Removed: sepal_width, Accuracy: 1.0000

Feature Removed: sepal_length, Accuracy: 0.9667

Feature Removed: petal_length, Accuracy: 0.9000

Feature Removed: petal_width, Accuracy: 0.8667

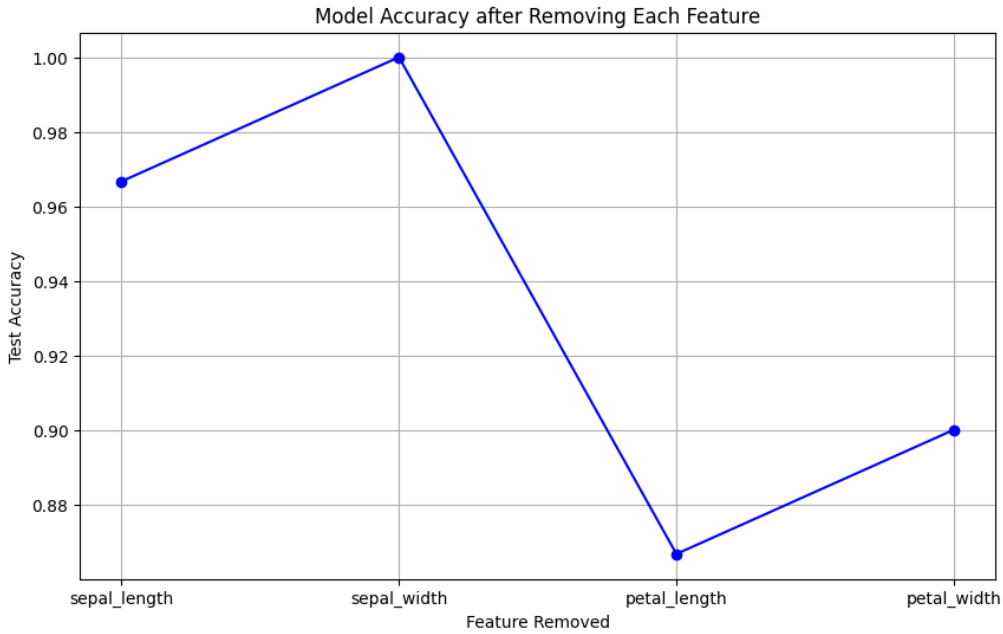
Training loss with Epochs with Backward Search

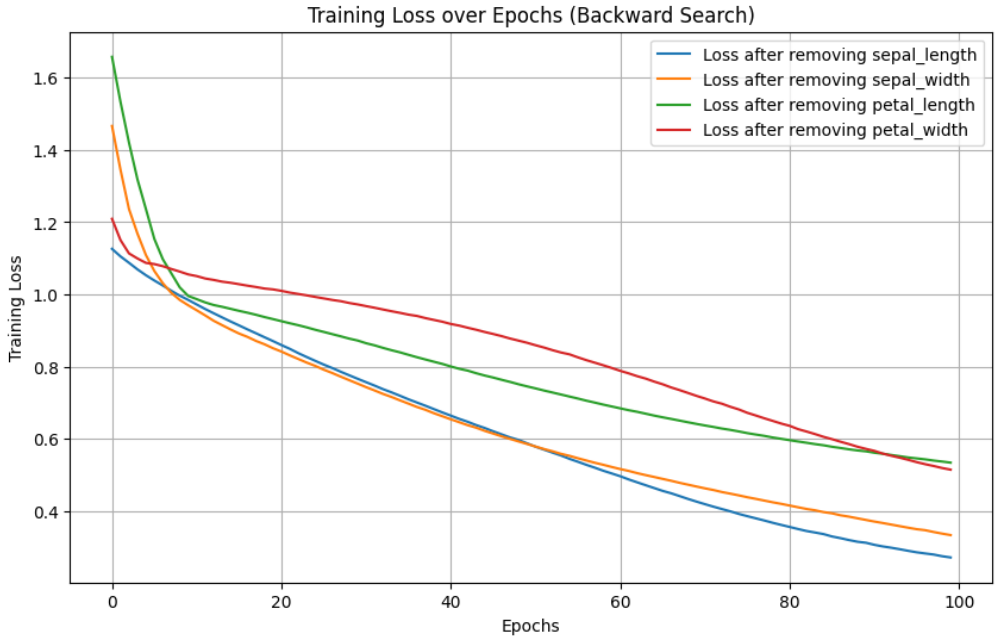
```
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
import matplotlib.pyplot as plt

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

# Separate features and target
X = df.drop(columns=['species'])
y = LabelEncoder().fit_transform(df['species'])  # Encode species as
integers

# Define the neural network model
def build_model(input_shape):
    model = Sequential([
        Input(shape=(input_shape,)),
        Dense(10, activation='relu'),
        Dense(10, activation='relu'),
        Dense(3, activation='softmax')
    ])
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# Perform backward search for feature selection and track accuracy and
training loss
def backward_search_with_plots(X, y):
    features = list(X.columns)
    feature_accuracy = []

    plt.figure(figsize=(10, 6))

    for i in range(len(features)):
        # Drop one feature at a time
        X_temp = X.drop(columns=[features[i]])
        X_train_temp, X_test_temp, y_train_temp, y_test_temp =
train_test_split(X_temp, y, test_size=0.2, random_state=42)

        # Build and train the model
        model = build_model(X_train_temp.shape[1])
        history = model.fit(X_train_temp, y_train_temp, epochs=100,
verbose=0)
```

```
        # Evaluate the model on the test set
        test_loss, test_accuracy = model.evaluate(X_test_temp,
y_test_temp, verbose=0)
        feature_accuracy.append((features[i], test_accuracy))

        # Plot training loss over epochs for each iteration
        plt.plot(history.history['loss'], label=f"Loss after removing
{features[i]}")

    # Display the training loss plot
    plt.title('Training Loss over Epochs (Backward Search)')
    plt.xlabel('Epochs')
    plt.ylabel('Training Loss')
    plt.legend()
    plt.grid()
    plt.show()

    # Plot feature accuracy after each removal
    feature_names = [x[0] for x in feature_accuracy]
    accuracies = [x[1] for x in feature_accuracy]

    plt.figure(figsize=(10, 6))
    plt.plot(feature_names, accuracies, marker='o', linestyle='-',
color='b')
    plt.title('Model Accuracy after Removing Each Feature')
    plt.xlabel('Feature Removed')
    plt.ylabel('Test Accuracy')
    plt.grid()
    plt.show()

    return feature_accuracy

# Perform backward search with plots
backward_search_results = backward_search_with_plots(X, y)
print("Backward Search Results:", backward_search_results)
```

Backward Search Results: [('sepal_length', 0.9666666388511658), ('sepal_width', 1.0), ('petal_length', 0.8666666746139526), ('petal_width', 0.8999999761581421)]


Model Accuracy after Removing Each Feature
Test Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
0.88
sepal_length
sepal_width
petal_length
petal_width
Feature Removed


Training Loss over Epochs (Backward Search)
Loss after removing sepal_length
Loss after removing sepal_width
Loss after removing petal_length
Loss after removing petal_width
Training Loss
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
20
40
60
80
100
Epochs