In [1]:
```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *

# qiskit-ibmq-provider has been deprecated.
# Please see the Migration Guides in https://ibm.biz/provider_migration_guide fo
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Sessior

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

# Invoke a primitive. For more details see https://qiskit.org/documentation/part
# result = Sampler("ibmq_qasm_simulator").run(circuits).result()
```

In [2]:
```
# load libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from math import pi, sqrt, acos, asin
from collections import Counter

# qiskit
import qiskit
from qiskit.extensions import Initialize
from qiskit.visualization import plot_histogram
from qiskit import QuantumCircuit, QuantumRegister, execute, Aer

# scikit-learn
from sklearn import datasets
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
```

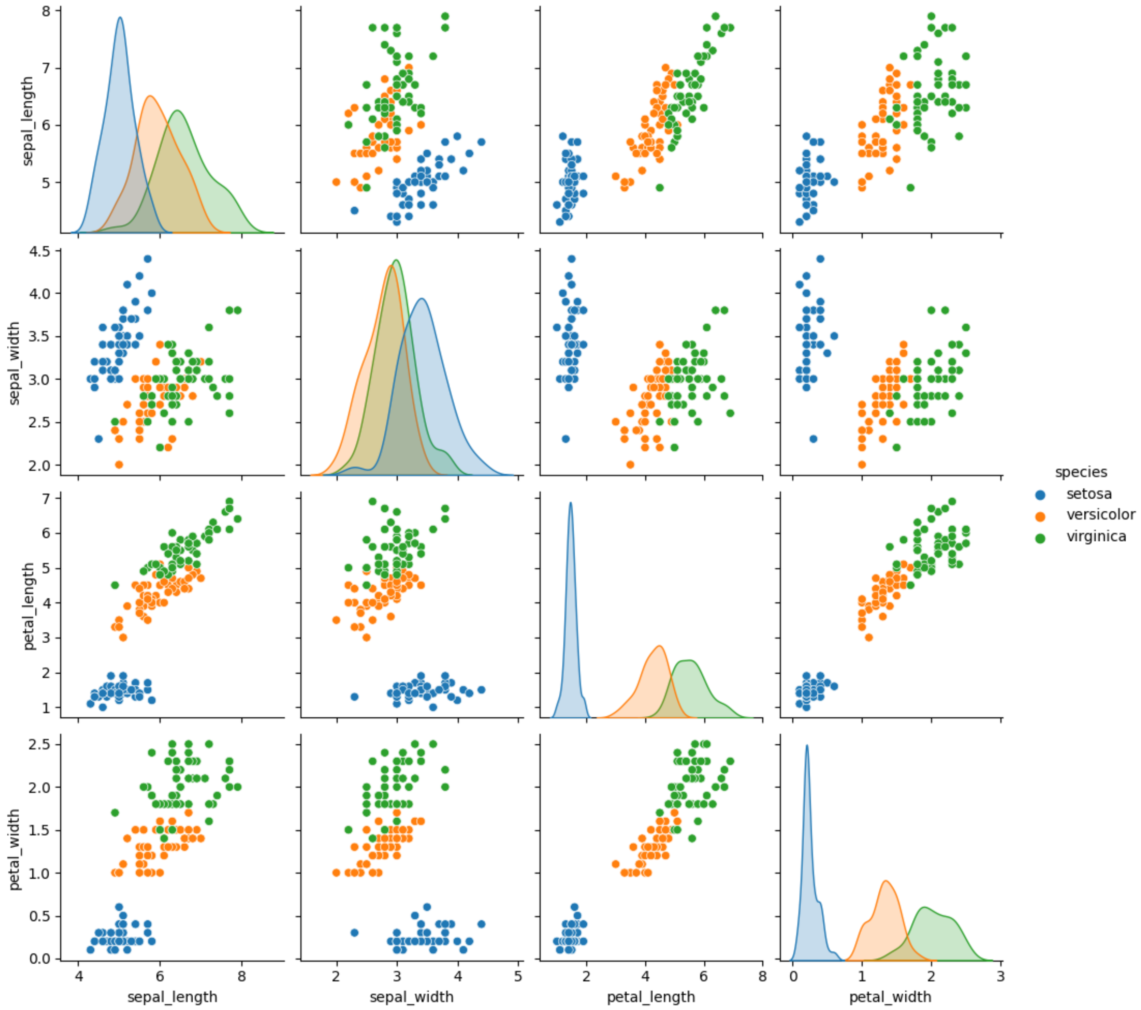
Iris Classical Classification

In [4]:
```
# Load the dataset
iris = load_iris()

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test

# Create the k-NN classifier
knn = KNeighborsClassifier(n_neighbors=3)

# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Predict the classes of the testing set
y_pred = knn.predict(X_test)
```

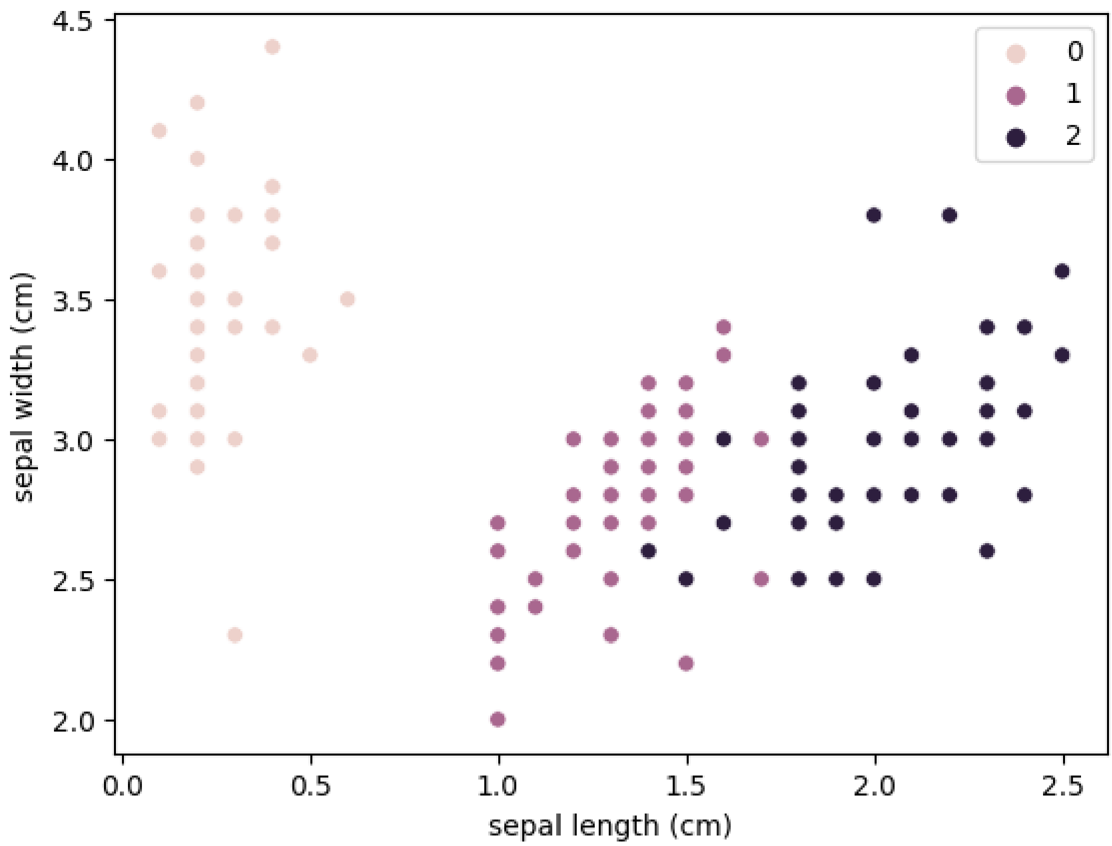
```python
# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy
print("Accuracy:", round(100*accuracy, 2))

# Plot the features
sns.pairplot(data=sns.load_dataset('iris'), hue='species')
```

```
Accuracy: 100.0
```

Out[4]:
```
<seaborn.axisgrid.PairGrid at 0x7f3a845573a0>
```



In [5]:
```python
# Plot the predictions
sns.scatterplot(x=iris.data[:, 3], y=iris.data[:, 1], hue=knn.predict(iris.data)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.show()
```



In [6]:
```
# Generate the classification report
report = classification_report(y_test, y_pred)

# Print the report
print(report)
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Quantum Amplitude Encoding

Consider the Bloch sphere

If the state $|\psi\rangle$ you want to measure is closer to $|0\rangle$ then there's a high chance you end up with 0. Similarly, if the state you want to measure is closer to $|1\rangle$ then there's a high chance you end up with 1 as the measurement with high probability. Finally, if the state $|\psi\rangle$ is in between $|0\rangle$ and $|1\rangle$ then you end up with almost equal chance for 0 and 1.

For instance, the R_y gate is a single-qubit gate that rotates the qubit about the y-axis of the Bloch sphere, which corresponds to changing the phase of the complex amplitude associated with the given qubit.

$$R_y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

The CR_Y gate is a two-qubit gate that applies a controlled-R_y gate, i.e., it performs a rotation about the y-axis of the Bloch sphere of the target qubit, conditioned on the state of the control qubit.

$$CR_Y(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ 0 & 0 & \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

These gates provide a way to manipulate the phases of the complex amplitudes and therefore allow us to create a wide variety of quantum states that can be used for various quantum algorithms.

Mapping Classes to Measurement Counts

For Iris dataset, we've 3 classes:

- We can make use of a single classical register for 3 classes
  - class 0: when count of 0 is significant
  - class 1: when count of 1 is significant
  - class 2: when in superposition / neither 0 nor 1 is dominant

In [3]:
```
def map_counts_to_class(counts, threshold=0.31):
    count_0 = counts.get('0', 0)
    count_1 = counts.get('1', 0)
    total_count = count_0 + count_1

    ratio_0 = count_0 / total_count
    ratio_1 = count_1 / total_count

    if abs(ratio_0 - ratio_1) <= threshold:
        return 1
    elif ratio_0 > ratio_1:
        return 0
    else:
        return 2

counts = {'1': 11, '0': 1013}
class_label = map_counts_to_class(counts)
print("Class label:", class_label)

counts = {'0': 111, '1': 913}
class_label = map_counts_to_class(counts)
print("Class label:", class_label)

counts = {'1': 491, '0': 513}
class_label = map_counts_to_class(counts)
print("Class label:", class_label)

counts = {'1': 400, '0': 624}
class_label = map_counts_to_class(counts)
print("Class label:", class_label)
```

```
Class label: 0
Class label: 2
Class label: 1
Class label: 1
```

In [7]:
```
# Load and normalize Iris dataset
iris = datasets.load_iris()
scaler = MinMaxScaler()
normalized_iris = scaler.fit_transform(iris.data)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(normalized_iris, iris.target

def create_quantum_circuit(features):
    n_qubits = len(features)
    qc = QuantumCircuit(n_qubits+1, 1)

    # Amplitude encoding using Rx gates
    for i, feature in enumerate(features):
        if feature < 0 or feature > 1:
            raise ValueError("All feature values must be between 0 and 1.")
        angle = acos(sqrt(1 - feature))
        qc.ry(2 * angle, i)

        # Create class states using Hadamard and Toffoli gates
        qc.cry(sqrt(angle), i, 4)

    # Measure all qubits
```

```
    qc.measure(4, 0)

    return qc


# Function to predict the class label using the top X most frequent qubit measur
def predict_class(counts, top_x=1, n_classes=3):
    counter = Counter(counts)
    most_common_outcomes = counter.most_common(top_x)
    possible_outcomes = 2 ** len(list(counts.keys())[0])

    # Map the most frequent outcomes to class labels
    class_labels = []
    for outcome, _ in most_common_outcomes:
        class_label = int(outcome, 2) * n_classes // possible_outcomes
        class_labels.append(class_label)

    # Return the most frequent class label
    return Counter(class_labels).most_common(1)[0][0]


# Test the quantum circuit with the training and testing sets
backend = Aer.get_backend("qasm_simulator")
y_pred_train = []
y_pred_test = []

lcounts = []
for features in X_train:
    qc = create_quantum_circuit(features)

    job = execute(qc, backend, shots=1024)
    counts = job.result().get_counts()
    lcounts.append(counts)

    y_pred_train.append(map_counts_to_class(counts))

for features in X_test:
    qc = create_quantum_circuit(features)

    job = execute(qc, backend, shots=1024)
    counts = job.result().get_counts()
    lcounts.append(counts)

    y_pred_test.append(map_counts_to_class(counts))

# Compute the accuracy of predictions
train_accuracy = np.mean(np.array(y_pred_train) == y_train)
test_accuracy = np.mean(np.array(y_pred_test) == y_test)

print("Train accuracy:", round(100*train_accuracy, 2))
print("Test accuracy:", round(100*test_accuracy, 2))
```

```
Train accuracy: 80.83
Test accuracy: 96.67
```

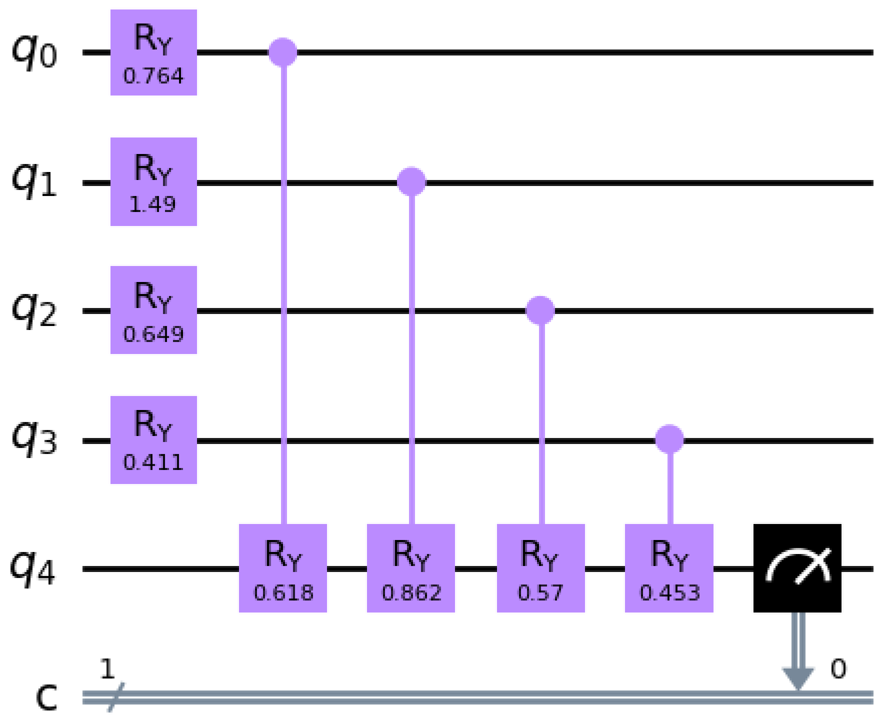
In [8]:

```
# Generate the classification report
report = classification_report(y_test, y_pred_test)

# Print the report
print(report)
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      0.89      0.94         9
           2       0.92      1.00      0.96        11

    accuracy                           0.97        30
   macro avg       0.97      0.96      0.97        30
weighted avg       0.97      0.97      0.97        30
```

In [9]:
```python
qc.draw()
```

Out[9]:



Note: This is not a recommendation rather to show something seemingly simple that can be achieved using quantum gates!