Institute of Engineering & Management

Quantum Computing Project Report

Level (Hard)

Submitted by-

Name-Probuddha Dutta -12022002018054 , Class roll-39(CSBS)

Sambit Chowdhury -12022002018075, Class roll-56(CSBS)

Souvik Mandal-12022002018076, Class roll-57(CSBS)

Date:27/11/2024

1. Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset.

Theoretical

Quantum KNN uses quantum states to encode data and measure distances between points in the quantum state space. It employs quantum measurement to compare the test point with training samples, selecting the closest k neighbors for classification.

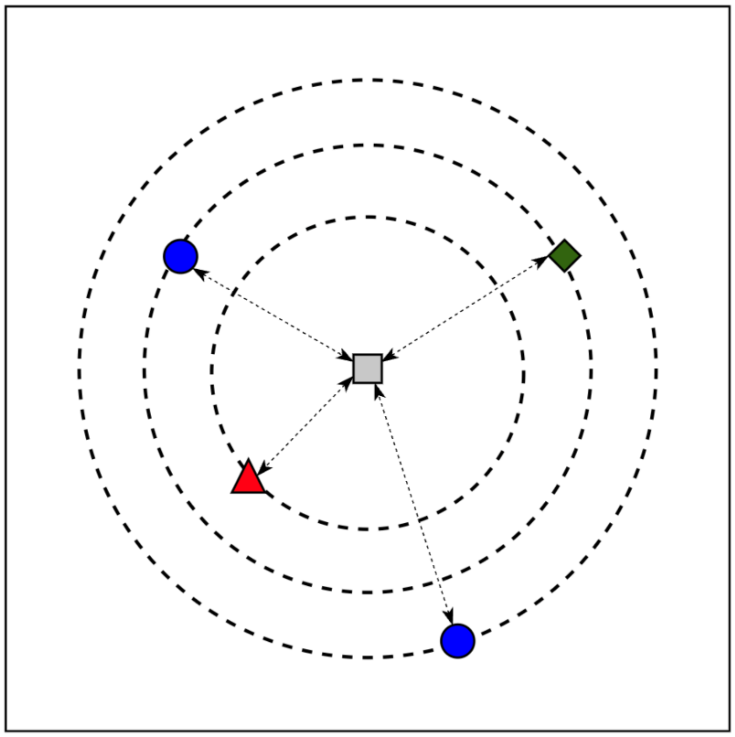
Description

The k nearest neighbor (KNN) classification is a widely-used supervised machine learning algorithm for classifying data points based on the proximity to their neighbors. This non-parametric method classifies samples by a majority vote of their neighbors, with the sample being assigned to the class most common among its k nearest neighbors.

Qiskit, developed by IBM Quantum, is an open-source quantum computing software development kit (SDK) that facilitates gate-based quantum circuit design and execution. As one of the leading quantum computing frameworks, Qiskit enables users to implement quantum algorithms and provides specialized tools for various domains, such as finance and material science simulations.

In this project, we have harnessed the power of Qiskit to implement a quantum version of the KNN algorithm, often referred to as q-KNN. Our approach involves creating a quantum circuit tailored for executing the q-KNN algorithm. We begin by encoding a classical dataset into quantum states, taking advantage of quantum phenomena such as superposition and entanglement. The quantum circuit is then simulated, leveraging Qiskit's powerful simulation backend, to determine the k nearest neighbors based on the quantum states' similarity.

This quantum version of the KNN algorithm aims to explore the potential speed-up and efficiency gains that quantum computing may offer over its classical counterpart. By exploiting quantum computing principles such as superposition and interference, we seek to perform the algorithm's operations in a fundamentally novel way that could only be achieved within a quantum computing framework.



Unknown sample (gray square) classified as 'red' based on its nearest neighbor

Theoretical Background and Project Workflow

The K-Nearest Neighbors (KNN) algorithm is a cornerstone of machine learning, which classifies entities based on the closest training examples in the feature space. Quantum K-Nearest Neighbors (q-KNN) brings this principle into the quantum domain, harnessing the computational advantages of quantum states and entanglement.

Data Encoding:

In q-KNN, classical vectors x⟶ are encoded into quantum states using amplitude encoding. This process translates a classical vector into the amplitudes of a quantum state, leveraging the state's ability to exist in multiple states simultaneously (superposition). The encoding is represented as: |ϕ(x⟶)⟩=∑i=1Nxi|i⟩, where |i⟩ denotes the computational basis states, and N is the dimensionality of the vector space.

State Preparation:

The quantum state |ψ⟩ of the training dataset is a superposed state of all training data points, enabling simultaneous
computation: |ψ⟩=1Ntrain ∑i=1Ntrain |ϕ(x(i))⟩⊗|i⟩ with |ϕ(x(i))⟩ representing the

amplitude-encoded quantum state of the i-th training sample, and |i⟩ being the index qubit in superposition representing the label or the class of the training example.

Distance Evaluation:

The SWAP test is used to measure the overlap between quantum states, which correlates with the distance between data points in the classical space. It involves an ancillary qubit and controlled-SWAP operations to create an interference pattern from which the probability of the ancilla being in state |0⟩ is indicative of the similarity between states.

Interference and Measurement:

Quantum interference is leveraged to measure the probability amplitude, which collapses due to the measurement postulate of quantum mechanics. The probability P(|0⟩) is computed and used to calculate the distance D between the test and training samples: P(|0⟩)=12+12|⟨ϕtest ∣ϕtrain ⟩|2, D=1−|⟨ϕtest ∣ϕtrain ⟩|2

Classical Post-Processing:

After quantum measurement, classical algorithms sort the computed distances and determine the k closest samples. The majority label among these nearest neighbors is assigned to the test sample.

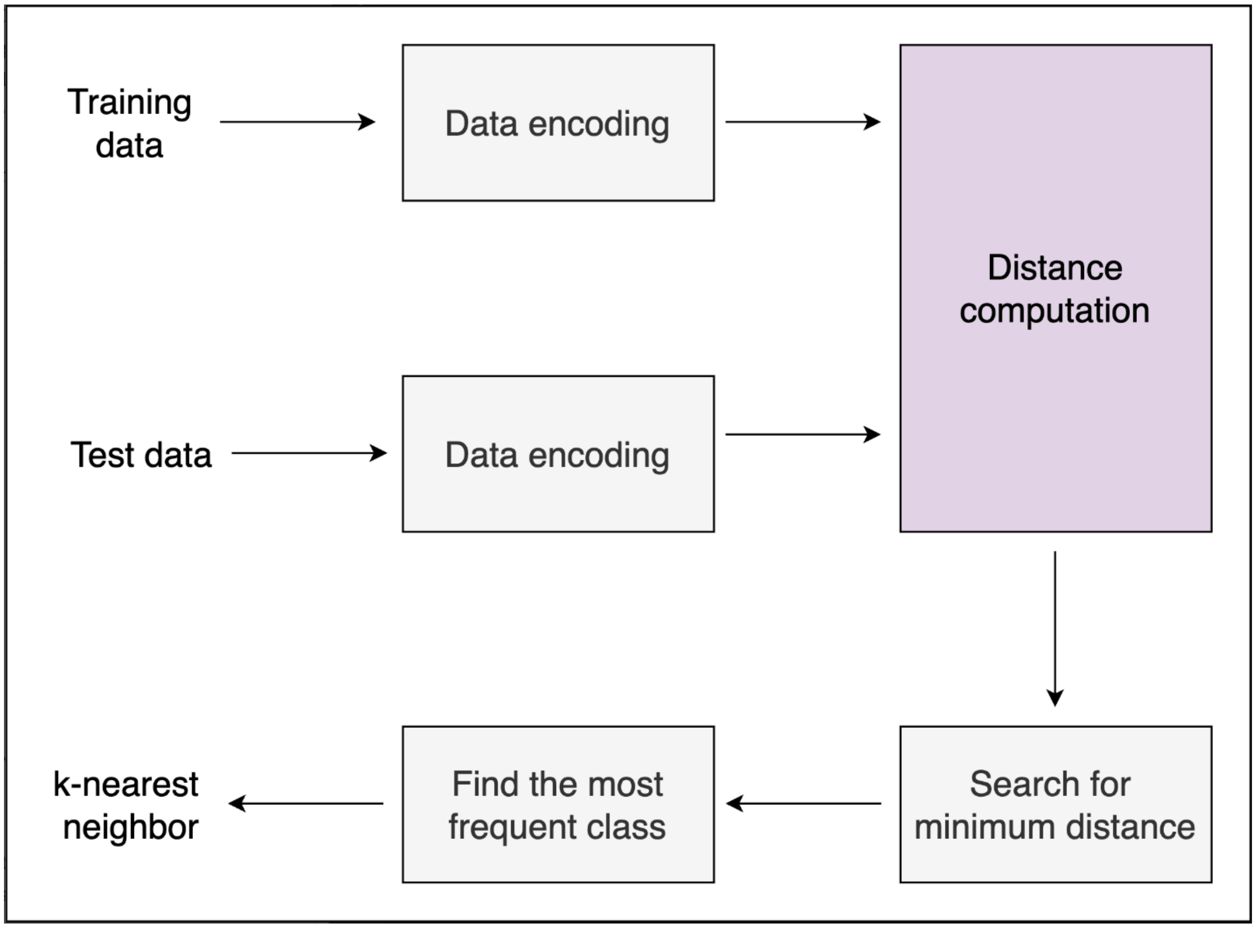
Circuit Execution and Simulation:

The q-KNN algorithm is implemented on a quantum circuit and executed on a quantum simulator. The circuit consists of initialization routines, entanglement operations, SWAP tests, and measurements that collectively simulate the quantum classification process.

By integrating quantum algorithms into classical machine learning workflows, q-KNN represents a novel approach that may provide computational benefits as quantum technology evolves.

Algorithm Workflow

The quantum KNN classifier uses the same approach for classifying data as the classical KNN classifier. The data itself may be either **classical** (encoded through quantum encoding techniques) or **quantum** (obtained from physical quantum-mechanical experiments). However, it provides a significant advantage over its classical counterpart by processing the training samples in parallel. This reduces the number of queries required to run the circuit.



In this algorithm:

1. The classical training and test feature vectors are converted to their quantum statevectors.
2. The encoded data is applied to a quantum circuit to compute the distances. You'll use the SWAP test to compute the distances, which returns the inner product of the training and test vectors.
3. The quantum data is then decoded along with the information about distances.
4. The samples with the minimum distance are extracted.
5. The most frequent class in these samples is assigned to the test sample.

Installation

To set up the necessary environment for running the q-KNN notebook, you need to have Python installed. All required Python libraries can be easily installed through the requirements.txt file included in the repository.

Please follow the steps below to install the dependencies:

ip install -r requirements.txt

Code Overview

The notebook is structured into various sections, each performing a specific part of the q-KNN algorithm:

- **Data Preprocessing**: Loading and visualizing the Iris dataset. Scaling features and splitting the dataset into training and test sets.
- **Data Encoding**: Encoding the test vector and training data into quantum states.
- **Quantum Circuit Creation**: Setting up quantum registers and initializing them with the encoded data.
- **SWAP Test**: Implementing the SWAP test to compare the test vector with the training data quantum states.
- **Simulation and Measurement**: Executing the quantum circuit and measuring the results.
- **Postprocessing**: Decoding the results to compute distances and identifying the nearest neighbors.
- **Model Evaluation**: Running the q-KNN model to classify all test vectors and evaluating the model's accuracy.

Detailed explanations of each code cell are provided within the notebook, guiding you through the entire process.

Features

- Quantum state encoding of classical data.
- Quantum circuit design for the SWAP test.
- Quantum simulation to compute distances in a quantum state space.
- Classification using the q-KNN algorithm.
- Model accuracy computation.

Code Implementation

modules.py

```
from qiskit import QuantumCircuit, QuantumRegister
def swap_test(N):
    `N`: Number of qubits of the quantum registers.
    a = QuantumRegister(N, 'a')
    b = QuantumRegister(N, 'b')
    d = QuantumRegister(1, 'd')
        qc_swap = QuantumCircuit(name = ' SWAP \nTest')
    qc_swap.add_register(a)
    qc_swap.add_register(b)
    qc_swap.add_register(d)

    qc_swap.h(d)
    for i in range(N):
        qc_swap.cswap(d, a[i], b[i]
    qc_swap.h(d)
    return qc_swap
```

```
def data_encoding(data):

    qc = QuantumCircuit(len(data))

    for i, value in enumerate(data):

        qc.ry(value, i)  # Encode data into rotations

    return qc
```

Library Imports

In this cell, we import essential libraries for both classical and quantum computing aspects of our K-Nearest Neighbors (KNN) classifier.

- **Classical Computing Libraries**:
    - numpy for numerical operations.
    - pandas for data manipulation and analysis.
    - seaborn for data visualization.
    - statistics to compute the mode, used in the KNN algorithm.
    - sklearn for dataset handling and preprocessing tools.
- **Quantum Computing Libraries (Qiskit)**:
    - QuantumCircuit, QuantumRegister, ClassicalRegister for building quantum circuits.
    - Aer and execute for running quantum simulations.
    - plot_histogram for visualizing quantum measurements.
    - Statevector for quantum state representation and manipulation.

```
#initiation
# Classical computing modules
import numpy as np
import pandas as pd
import seaborn as sns
from statistics import mode
# Dataset
from sklearn import datasets, preprocessing
from sklearn.model_selection import train_test_split
# Quantum computing modules
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, Aer, execute
from qiskit.visualization import plot_histogram
from qiskit.quantum_info import Statevector
```

Dataset Loading and Initial Exploration

In this cell, we initiate our project by loading the Iris dataset, a classic in the field of machine learning, known for its simplicity and effectiveness in classification tasks.

- **Load the Dataset**:
  - We use datasets.load_iris() from sklearn to load the Iris dataset.
  - feature_labels and class_labels are extracted to understand the dataset's features and classes.
- **Dataset Extraction**:
  - X represents the feature data of the Iris dataset.
  - y holds the target labels (classes) of the dataset.
  - M is set to 4, representing the number of features in the dataset.
- **Initial Data Inspection**:

  - We print feature_labels and class_labels to get an initial understanding of the dataset's structure, which includes features like petal length, petal width, etc., and classes such as Setosa, Versicolor, and Virginica.

This step is crucial as it sets the stage for further data processing and prepares us for the application of both classical and quantum machine learning techniques.

```
2# Load the dataset
iris = datasets.load_iris()
feature_labels = iris.feature_names
class_labels = iris.target_names
# Extract the data
X = iris.data
y = np.array([iris.target])
M = 4

# Print the features and classes
print("Features: ", feature_labels)
print("Classes: ", class_labels)
```

```
Features:  ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Classes:  ['setosa' 'versicolor' 'virginica']
```

Data Visualization with Pair Plot

After loading the Iris dataset, this cell emphasizes data visualization, which is essential for understanding feature relationships and distribution.

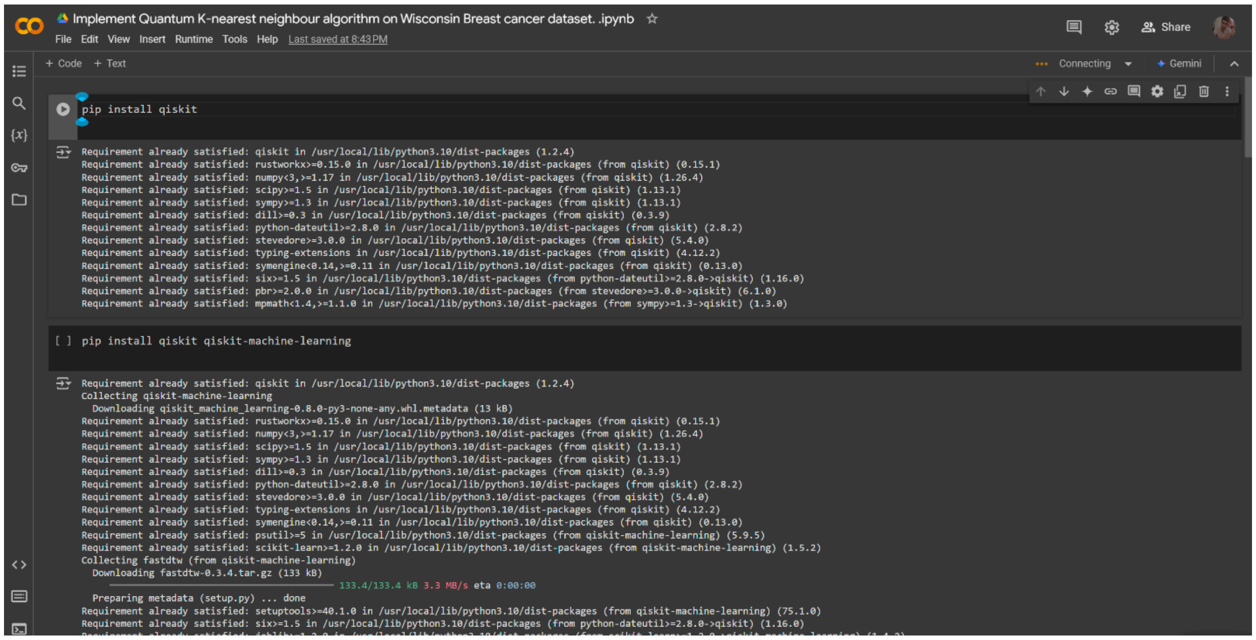
- **DataFrame Creation**:
  - We create a DataFrame df using pandas.
  - The DataFrame combines feature data (X) and target labels (y) for a holistic view.

    - Columns are named using feature_labels and "Species" for easy reference.

- **Pair Plot Visualization**:

    - Utilizing seaborn's pairplot, we visualize pairwise relationships in the dataset.
    - The hue parameter is set to 'Species', allowing us to distinguish data points by their class.
    - This visualization aids in identifying patterns, correlations, and feature separability among different species.

This visual exploration is a key step in preliminary data analysis, offering insights that guide our approach to building the KNN classifier, both in its classical and quantum forms.

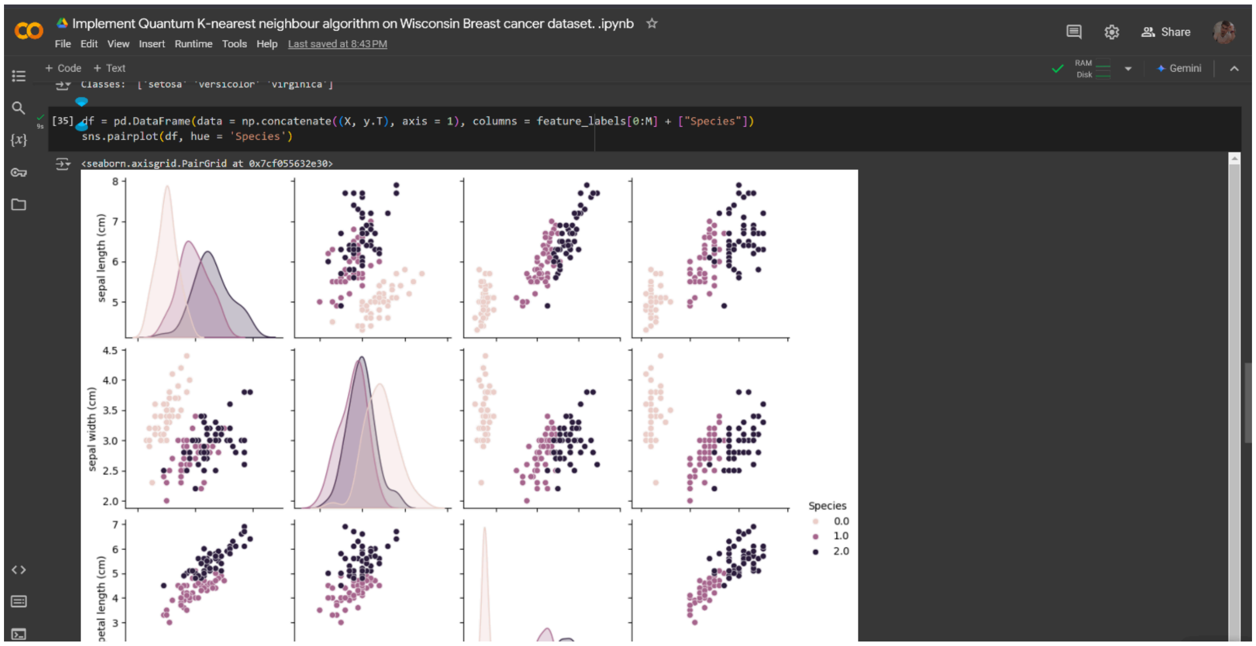

Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb
File Edit View Insert Runtime Tools Help Last saved at 8:43 PM
+ Code + Text
Connecting
Gemini
Share

```
pip install qiskit
```

```
Requirement already satisfied: qiskit in /usr/local/lib/python3.10/dist-packages (1.2.4)
Requirement already satisfied: rustworkx>=0.15.0 in /usr/local/lib/python3.10/dist-packages (from qiskit) (0.15.1)
Requirement already satisfied: numpy<3,>=1.17 in /usr/local/lib/python3.10/dist-packages (from qiskit) (1.26.4)
Requirement already satisfied: scipy>=1.5 in /usr/local/lib/python3.10/dist-packages (from qiskit) (1.13.1)
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.10/dist-packages (from qiskit) (1.13.1)
Requirement already satisfied: dill>=0.3 in /usr/local/lib/python3.10/dist-packages (from qiskit) (0.3.9)
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.10/dist-packages (from qiskit) (2.8.2)
Requirement already satisfied: stevedore>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from qiskit) (5.4.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from qiskit) (4.12.2)
Requirement already satisfied: symengine<0.14,>=0.11 in /usr/local/lib/python3.10/dist-packages (from qiskit) (0.13.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.0->qiskit) (1.16.0)
Requirement already satisfied: pbr>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from stevedore>=3.0.0->qiskit) (6.1.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy>=1.3->qiskit) (1.3.0)
```

```
[ ] pip install qiskit qiskit-machine-learning
```

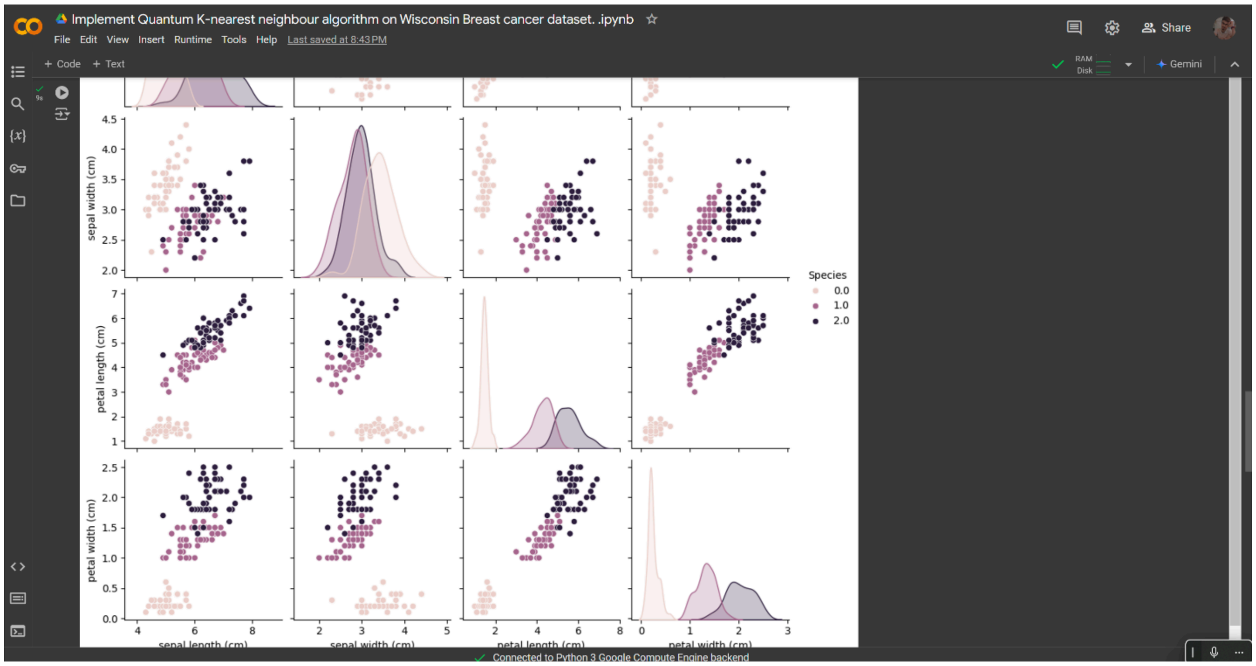
```
Requirement already satisfied: qiskit in /usr/local/lib/python3.10/dist-packages (1.2.4)
Collecting qiskit-machine-learning
  Downloading qiskit_machine_learning-0.8.0-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: rustworkx>=0.15.0 in /usr/local/lib/python3.10/dist-packages (from qiskit) (0.15.1)
Requirement already satisfied: numpy<3,>=1.17 in /usr/local/lib/python3.10/dist-packages (from qiskit) (1.26.4)
Requirement already satisfied: scipy>=1.5 in /usr/local/lib/python3.10/dist-packages (from qiskit) (1.13.1)
Requirement already satisfied: sympy>=1.3 in /usr/local/lib/python3.10/dist-packages (from qiskit) (1.13.1)
Requirement already satisfied: dill>=0.3 in /usr/local/lib/python3.10/dist-packages (from qiskit) (0.3.9)
Requirement already satisfied: python-dateutil>=2.8.0 in /usr/local/lib/python3.10/dist-packages (from qiskit) (2.8.2)
Requirement already satisfied: stevedore>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from qiskit) (5.4.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from qiskit) (4.12.2)
Requirement already satisfied: symengine<0.14,>=0.11 in /usr/local/lib/python3.10/dist-packages (from qiskit) (0.13.0)
Requirement already satisfied: psutil>=5 in /usr/local/lib/python3.10/dist-packages (from qiskit-machine-learning) (5.9.5)
Requirement already satisfied: scikit-learn>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-machine-learning) (1.5.2)
Collecting fastdtw (from qiskit-machine-learning)
  Downloading fastdtw-0.3.4.tar.gz (133 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 133.4/133.4 kB 3.3 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: setuptools>=40.1.0 in /usr/local/lib/python3.10/dist-packages (from qiskit-machine-learning) (75.1.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.0->qiskit) (1.16.0)
```


Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb
File Edit View Insert Runtime Tools Help Last saved at 8:43 PM
Share
+ Code + Text
Gemini
sepal length (cm)
sepal width (cm)
petal length (cm)
Species
0.0
1.0
2.0

```
Classes: ['setosa' 'versicolor' 'virginica']
```

```
[35] df = pd.DataFrame(data = np.concatenate((X, y.T), axis = 1), columns = feature_labels[0:M] + ["Species"])
     sns.pairplot(df, hue = 'Species')
```

```
<seaborn.axisgrid.PairGrid at 0x7cf055632e30>
```


Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb
File Edit View Insert Runtime Tools Help Last saved at 8:43 PM
Share
+ Code + Text
Gemini
sepal width (cm)
petal length (cm)
petal width (cm)
sepal length (cm)
Species
0.0
1.0
2.0
Connected to Python 3 Google Compute Engine backend

Data Normalization

This cell introduces data normalization, a crucial step in preparing our dataset for the K-Nearest Neighbors classifier, especially when it involves quantum computing elements.

- **Min-Max Scaler Initialization**:
  - We initialize a MinMaxScaler from sklearn.preprocessing.
  - The scaler is set to normalize feature values within the range [0, 1].
- **Fitting and Transforming the Dataset**:
  - The scaler is then fitted to our feature dataset X to learn the min and max values.
  - X_normalized is created by transforming X using the fitted scaler.

Normalization ensures that all features contribute equally to the distance calculations in the KNN algorithm, which is essential for its effective functioning. This is particularly important in quantum computing, where feature representation can significantly influence the algorithm's performance.

In [4]:

```
min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))

min_max_scaler.fit(X)

X_normalized = min_max_scaler.transform(X)
```

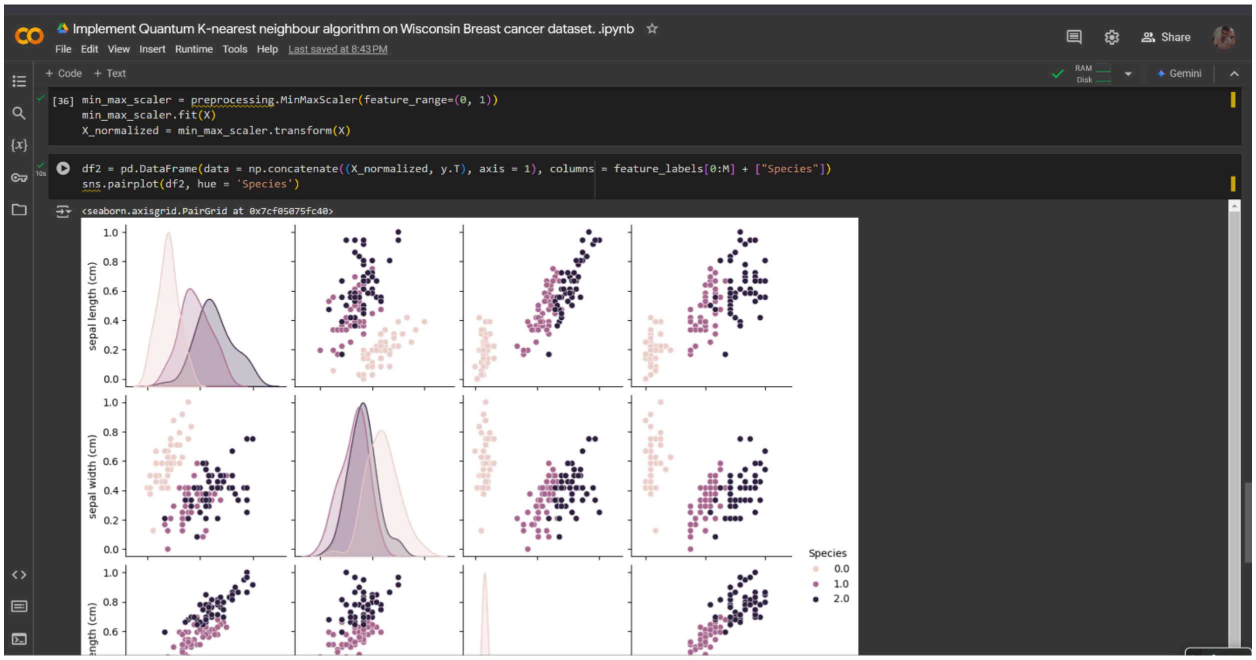
Visualization of Normalized Data

Building upon the earlier visualization steps, this cell aims to visualize the normalized data, providing a comparative perspective with the pre-normalized data.

- **Normalized DataFrame Creation**:
  - A new DataFrame df2 is created using pandas.
  - This DataFrame combines the normalized feature data (X_normalized) with the target labels (y).
  - Column names are set using feature_labels and "Species" for consistency and ease of understanding.
- **Pair Plot of Normalized Data**:
  - We again use seaborn's pairplot for visualization.
  - The hue parameter remains set to 'Species', facilitating class differentiation.

- This visualization allows us to observe the effect of normalization on the dataset, particularly how it impacts the distribution and relationships between features.

Visualizing the normalized data is a key analytical step, offering insights into the data's structure post-normalization and ensuring that our preprocessing steps align with the requirements of a robust KNN classifier.

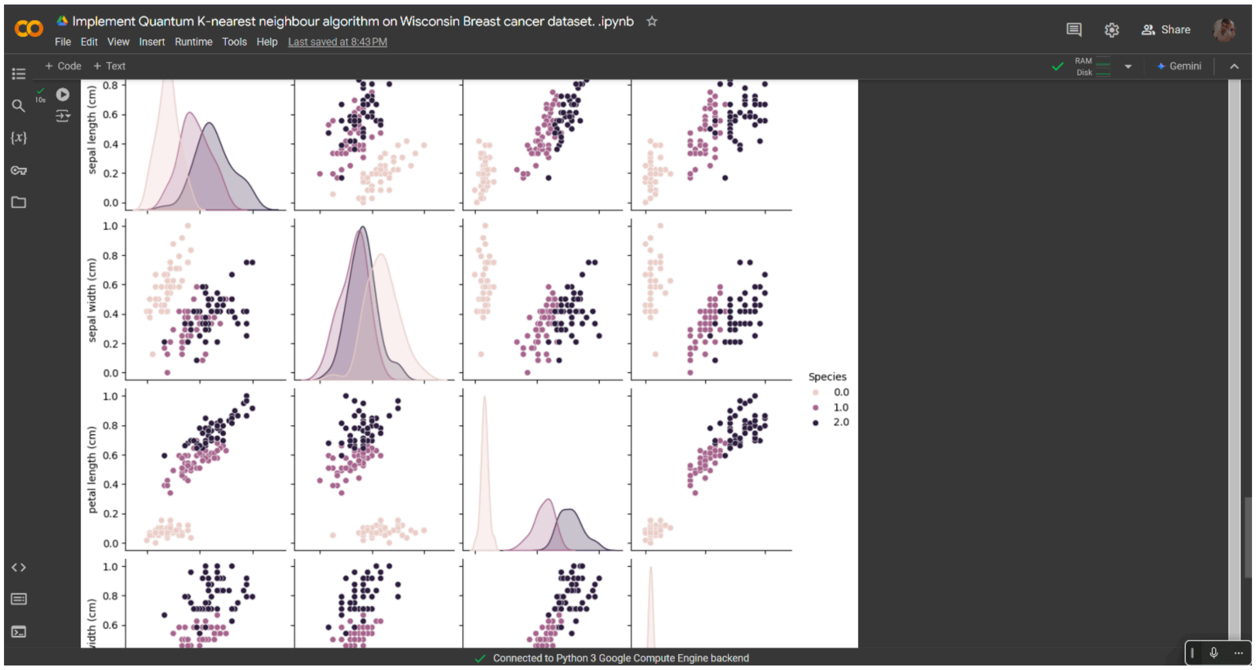


```
[36] min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
     min_max_scaler.fit(X)
     X_normalized = min_max_scaler.transform(X)
```

```
df2 = pd.DataFrame(data = np.concatenate((X_normalized, y.T), axis = 1), columns = feature_labels[0:M] + ["Species"])
sns.pairplot(df2, hue = 'Species')
```

```
<seaborn.axisgrid.PairGrid at 0x7cf05075fc40>
```

Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb
File Edit View Insert Runtime Tools Help Last saved at 8:43 PM
+ Code + Text
Share
Gemini
sepal length (cm)
sepal width (cm)
Species
0.0
1.0
2.0



Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb
File Edit View Insert Runtime Tools Help Last saved at 8:43 PM
+ Code + Text
Share
Gemini
sepal length (cm)
sepal width (cm)
petal length (cm)
Species
0.0
1.0
2.0
Connected to Python 3 Google Compute Engine backend

Splitting the Dataset into Training and Testing Sets

This cell is dedicated to dividing our normalized dataset into distinct training and testing sets, a standard practice in machine learning to evaluate the model's performance.

- **Setting the Training Dataset Size**:
  - We define N_train as 100, indicating the number of samples to include in the training set.
- **Train-Test Split**:
  - The train_test_split function from sklearn.model_selection is used.
  - X_normalized (feature data) and y (target labels) are split into training (X_train, y_train) and testing (X_test, y_test) sets.
  - The train_size parameter is set to N_train to specify the size of the training set.

This step is crucial in creating a robust machine learning model, as it provides us with a separate dataset (the testing set) to evaluate the model's performance and generalizability beyond the data it was trained on.

In [6]:

```
N_train = 100 # Training dataset size


X_train, X_test, y_train, y_test = train_test_split(X_normalized, y[0], train_size=N_train)
```

Test Sample Selection and Quantum State Preparation

This cell plays a pivotal role in our Quantum KNN classifier, focusing on selecting a test sample and encoding it as a quantum state vector.

- **Selecting a Test Sample**:
  - We randomly select a test sample index (test_index) from the X_test dataset.
  - phi_test is then assigned the feature values of this selected test sample.
- **Quantum State Encoding**:

  - We encode the test sample features (phi_test) into a quantum state.
  - For each feature i in phi_test, we create a state phi_i representing the amplitude probabilities, ensuring they sum up to 1.
  - The states phi_i are then combined using the Kronecker product to form the complete quantum state phi.
- **Statevector Validation**:
  - We use Qiskit's Statevector to validate phi ensuring it represents a valid quantum state.
  - The is_valid() function checks if the statevector phi is normalized and hence a legitimate state in quantum mechanics.

This step is essential in quantum machine learning, as it translates classical data into a quantum form, enabling us to apply quantum algorithms for tasks like classification.

In [7]:

```
# Select a sample test sample for classification
test_index = int(np.random.rand()*len(X_test))
phi_test = X_test[test_index]

# Create the encoded feature vector
for i in range(M):
    phi_i = [np.sqrt(phi_test[i]), np.sqrt(1- phi_test[i])]
    if i == 0:
        phi = phi_i
    else:
        phi = np.kron(phi_i, phi)

# Validate the statevector
print('Valid statevector: ', Statevector(phi).is_valid())
```

Valid statevector: True

Construction of Quantum State for Training Data

In this cell, we undertake a crucial step in quantum machine learning: constructing a quantum state that encodes our training dataset.

- **Determining the Quantum Register Size**:
  - N is calculated as the ceiling of the logarithm base 2 of the training dataset size (N_train), determining the size of the quantum register needed.
- **Initializing the Quantum State psi**:
  - We initialize psi as a zero vector of size 2^(M + N) to accommodate all training data points and features.
- **Encoding Training Data into Quantum States**:
  - For each training data point i, we perform the following:
    - Encode the index |i> into a quantum state i_vec.
    - Encode the feature vector x (from X_train[i, :]) into a quantum state x_vec using the amplitude encoding scheme.
    - Combine x_vec and i_vec using the Kronecker product to form psi_i.
    - Add psi_i to psi.
  - Normalize psi by dividing it by the square root of N_train.
- **Statevector Validation**:
  - We assert the validity of psi using Qiskit's Statevector class, ensuring it is a square-normalized quantum state.

This step is fundamental in quantum computing as it transforms classical data into a quantum format, enabling the application of quantum algorithms for the classification task.

In [8]:

```
N = int(np.ceil(np.log2(N_train)))
psi = np.zeros(2**(M + N)) # Task 9
```

```
for i in range(N_train):
    # Encode |i>
    i_vec = np.zeros(2**N)
    i_vec[i] = 1

    # Encode |x>
    x = X_train[i, :]
    for j in range(M):
        dummy = [np.sqrt(x[j]), np.sqrt(1- x[j])]
        if j == 0:
            x_vec = dummy
        else:
            x_vec = np.kron(dummy, x_vec)
    psi_i = np.kron(x_vec, i_vec)
    # Task 9
    psi += psi_i
psi /= np.sqrt(N_train)

# Check the validity of the statevector
assert Statevector(psi).is_valid(), "The statevector is not square-normalized."
```

Setting Up Quantum Registers for KNN Classifier

In this cell, we establish the quantum registers required for implementing the Quantum KNN classifier, laying the groundwork for constructing the quantum circuit.

- **Index Register (index_reg)**:
    - A quantum register of size N (calculated previously) is created and named 'i'.
    - This register is used to encode the indices of the training data points.
- **Training Register (train_reg)**:

    - A quantum register of size M (number of features) is created and named 'train'.
    - It serves to hold the quantum state representation of the training data features.
- **Test Register (test_reg)**:
    - Another quantum register of size M, named 'test', is established.
    - This register will encode the quantum state of the test data feature vector.
- **Similarity Register (p)**:
    - A single qubit register is created, named 'similarity'.
    - This qubit will be used to measure the similarity between test and training data points in the quantum circuit.

Setting up these quantum registers is a critical step in preparing for the quantum circuit that will perform the KNN classification. Each register plays a specific role in the process, enabling the encoding of both training and test data, as well as the calculation of their similarities.

In [9]:

```
index_reg = QuantumRegister(N, 'i')
train_reg = QuantumRegister(M, 'train')
test_reg = QuantumRegister(M, 'test')
p = QuantumRegister(1, 'similarity')
```

Creation of the Quantum Circuit for KNN Classifier

This cell marks a significant step in our Quantum KNN project: the creation and initial visualization of the quantum circuit.

- **Quantum Circuit Initialization**:
    - We initialize an empty Quantum Circuit, named qknn.
- **Adding Registers to the Circuit**:

- The previously defined quantum registers index_reg, train_reg, test_reg, and p are added to qknn.
- index_reg is for indexing the training data.
- train_reg and test_reg are for encoding the training and test data features, respectively.
- p is the similarity register used in similarity measurement.

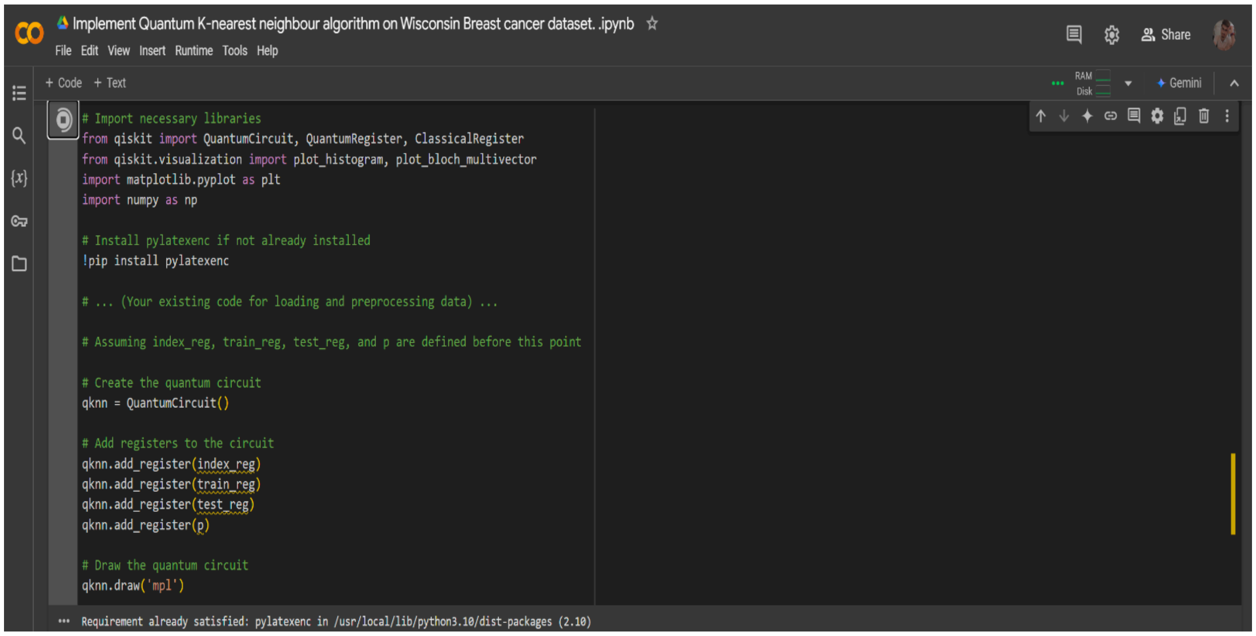
- **Circuit Visualization**:
    - The circuit is visualized using Qiskit's draw method with the 'mpl' (Matplotlib) style.
    - This visualization provides an overview of the quantum circuit's structure and the registers it comprises.

Constructing and visualizing the quantum circuit is a crucial stage. It lays out the foundation on which the quantum algorithm for the KNN classifier will be built and operated.

In [10]:

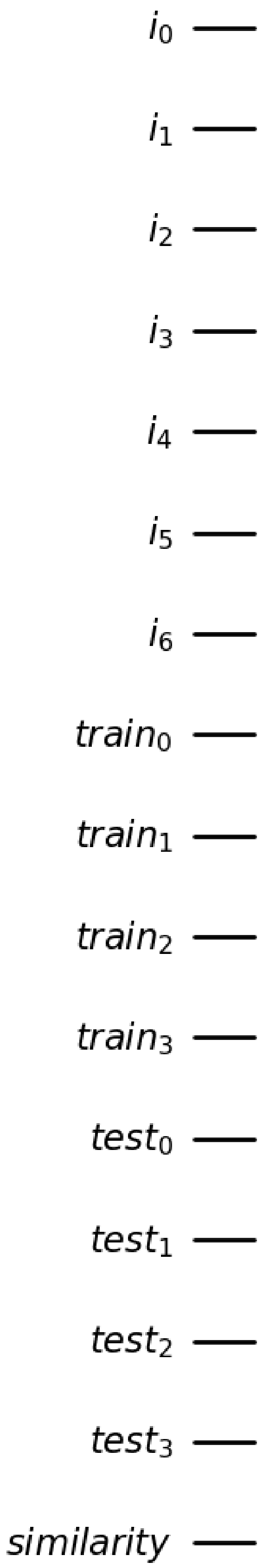
```
# Create the quantum circuit
qknn = QuantumCircuit()

# Add registers to the circuit
qknn.add_register(index_reg)
qknn.add_register(train_reg)
qknn.add_register(test_reg)
qknn.add_register(p)

# Draw the quantum circuit
qknn.draw('mpl')
```

Out[10]:


Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb
File Edit View Insert Runtime Tools Help
Share
+ Code + Text
RAM
Disk
Gemini

```
# Import necessary libraries
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.visualization import plot_histogram, plot_bloch_multivector
import matplotlib.pyplot as plt
import numpy as np

# Install pylatexenc if not already installed
!pip install pylatexenc

# ... (Your existing code for loading and preprocessing data) ...

# Assuming index_reg, train_reg, test_reg, and p are defined before this point

# Create the quantum circuit
qknn = QuantumCircuit()

# Add registers to the circuit
qknn.add_register(index_reg)
qknn.add_register(train_reg)
qknn.add_register(test_reg)
qknn.add_register(p)

# Draw the quantum circuit
qknn.draw('mpl')
```

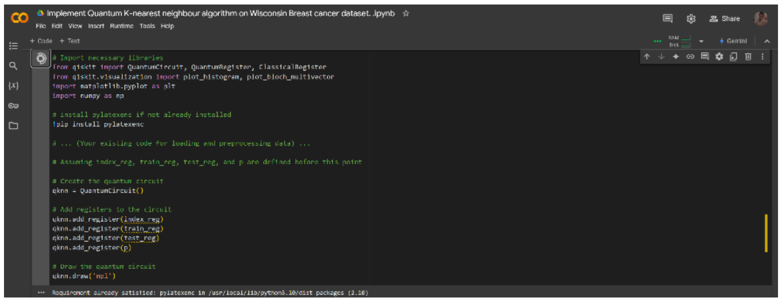
```
Requirement already satisfied: pylatexenc in /usr/local/lib/python3.10/dist-packages (2.10)
```


i_0
i_1
i_2
i_3
i_4
i_5
i_6
$train_0$
$train_1$
$train_2$
$train_3$
$test_0$
$test_1$
$test_2$
$test_3$
$similarity$

Initializing Quantum Registers in the Circuit

This cell is dedicated to initializing the quantum registers in our KNN quantum circuit, a crucial step in preparing the circuit for the classification process.

- **Initializing the Training Register**:
    - The initialize method is used to set the index_reg and train_reg registers with the quantum state psi.
    - psi represents the encoded training dataset, as previously constructed.
    - The state is applied to the first N qubits of index_reg and the first M qubits of train_reg.
- **Initializing the Test Register**:
    - Similarly, the initialize method is used to set the test_reg register with the quantum state phi.
    - phi is the encoded state of the selected test sample.
- **Circuit Visualization (Post-Initialization)**:
    - After initialization, the quantum circuit qknn is visualized again using the 'mpl' style.
    - This visualization helps to confirm that the registers are correctly initialized with the respective quantum states.


Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
Share
Gemini

```
# Import necessary libraries
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.visualization import plot_histogram, plot_bloch_multivector
import matplotlib.pyplot as plt
import numpy as np

# install pylatexenc if not already installed
!pip install pylatexenc

# ... (Your existing code for loading and preprocessing data) ...

# Assuming index_reg, train_reg, test_reg, and p are defined before this point

# Create the quantum circuit
qknn = QuantumCircuit()

# Add registers to the circuit
qknn.add_register(index_reg)
qknn.add_register(train_reg)
qknn.add_register(test_reg)
qknn.add_register(p)

# Draw the quantum circuit
qknn.draw('mpl')
```

```
Requirement already satisfied: pylatexenc in /usr/local/lib/python3.10/dist packages (2.10)
```

Implementing Quantum SWAP Test on the Circuit

This cell introduces a critical component of the Quantum KNN classifier: the Quantum SWAP Test. The SWAP Test is a quantum algorithm used to estimate the similarity between two quantum states.

- **Importing the SWAP Test Module**:
  - The Quantum SWAP Test module is imported from a custom module named modules.
  - This indicates the use of a predefined function or circuit that implements the SWAP Test.
- **Applying the SWAP Test**:
  - The swap_test function/module, which presumably implements the Quantum SWAP Test algorithm, is appended to the qknn circuit.
  - It is applied to the first M qubits of both train_reg and test_reg, and the single qubit in the p register.
  - The train_reg and test_reg hold the quantum states of the training and test data, respectively, while p is used for measuring similarity.
- **Circuit Visualization (Post-SWAP Test)**:
  - The circuit qknn is visualized again with the 'mpl' style to show the inclusion of the SWAP Test.
  - This visualization allows us to verify the correct application of the SWAP Test within the circuit.

Defining Quantum KNN Module and Evaluating Model Accuracy

This final cell encapsulates the entire Quantum KNN process in a function and evaluates the classifier's performance across the test dataset.

- **Quantum KNN Module (q_knn_module)**:
  - The function q_knn_module is defined to implement the Quantum KNN classifier for a given test sample index test_index and the quantum state psi encoding the training data.
  - It follows the steps previously executed individually: encoding the test sample into a quantum state, initializing quantum registers, applying the swap test, executing the circuit, and processing the results.

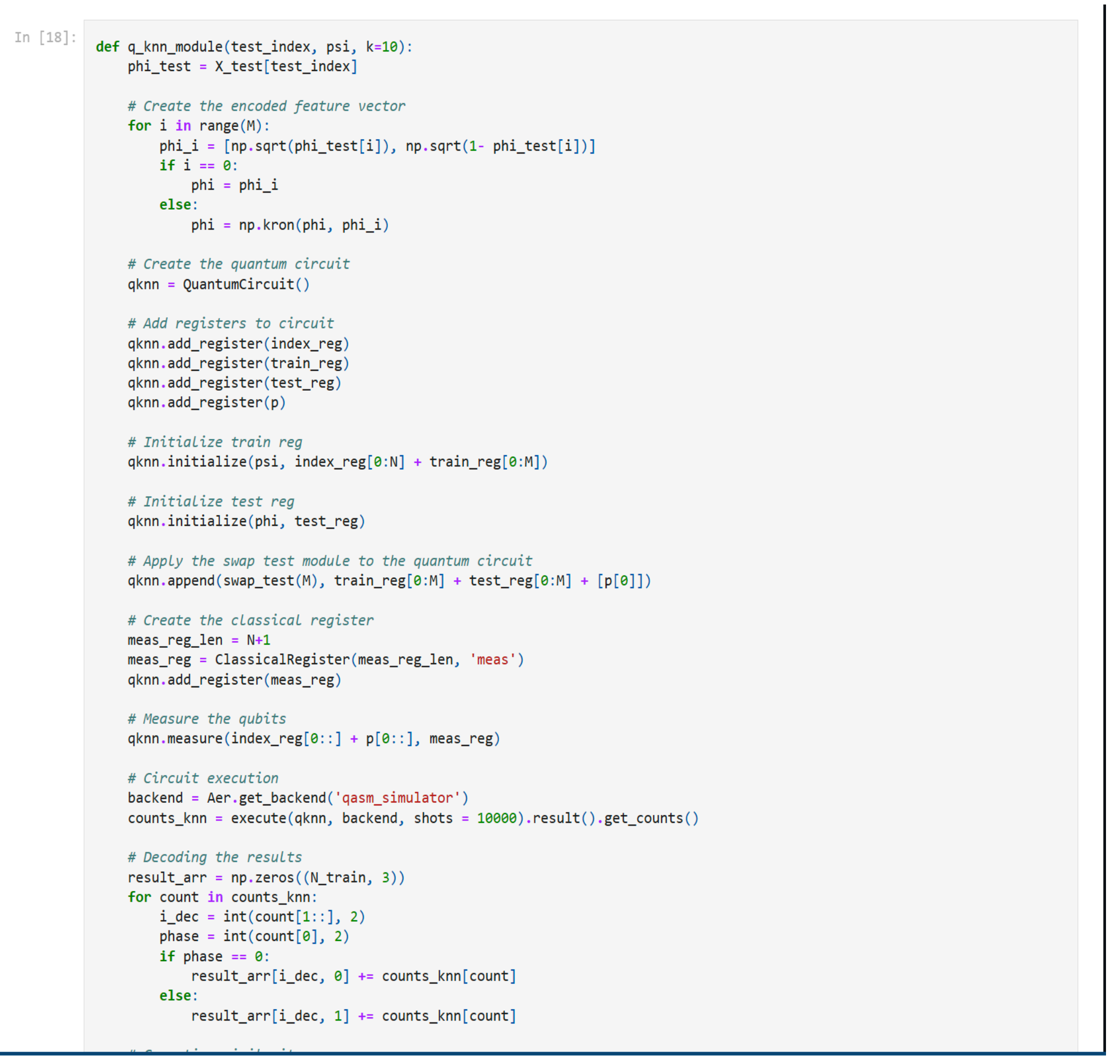
    - The function returns the predicted and expected classes for the given test sample.
- **Applying the Function on Test Data**:
    - We iterate over each test sample in X_test, applying q_knn_module to obtain the predicted (y_pred) and expected (y_exp) classes.
    - These classes are collected in y_pred_arr and y_exp_arr.
- **Calculating Model Accuracy**:
    - The model's accuracy is calculated by comparing the predicted and expected classes for each test sample.
    - We count the number of true (correct) predictions t and false (incorrect) predictions f.
    - The accuracy is then calculated as the percentage of true predictions out of the total and printed.

This comprehensive function and accuracy evaluation provide a holistic view of the Quantum KNN classifier's performance, demonstrating its effectiveness in classifying data using quantum computing principles.

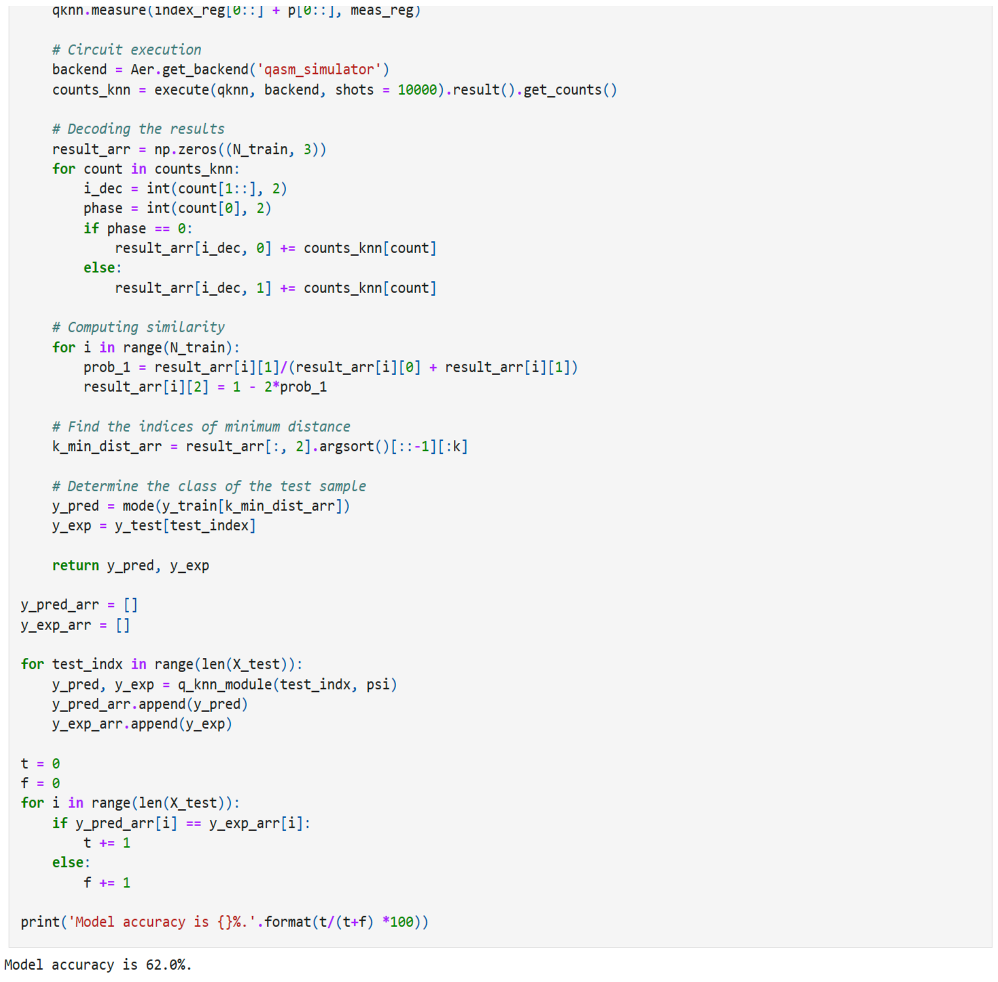
In [18]:

```
def q_knn_module(test_index, psi, k=10):
    phi_test = X_test[test_index]
    for i in range(M):
        phi_i = [np.sqrt(phi_test[i]), np.sqrt(1- phi_test[i])]
        if i == 0:
            phi = phi_i
        else:
            phi = np.kron(phi, phi_i)
    qknn = QuantumCircuit()
    qknn.add_register(index_reg)
    qknn.add_register(train_reg)
    qknn.add_register(test_reg)
    qknn.add_register(p)
    qknn.initialize(psi, index_reg[0:N] + train_reg[0:M])
    qknn.initialize(phi, test_reg)
    qknn.append(swap_test(M), train_reg[0:M] + test_reg[0:M] + [p[0]])

        meas_reg_len = N+1
    meas_reg = ClassicalRegister(meas_reg_len, 'meas')
    qknn.add_register(meas_reg)
    qknn.measure(index_reg[0::] + p[0::], meas_reg)
    backend = Aer.get_backend('qasm_simulator')
    counts_knn = execute(qknn, backend, shots = 10000).result().get_counts()
    result_arr = np.zeros((N_train, 3))
    for count in counts_knn:
        i_dec = int(count[1::], 2)
        phase = int(count[0], 2)
        if phase == 0:
            result_arr[i_dec, 0] += counts_knn[count]
        else:
            result_arr[i_dec, 1] += counts_knn[count]
    for i in range(N_train):
        prob_1 = result_arr[i][1]/(result_arr[i][0] + result_arr[i][1])
        result_arr[i][2] = 1 - 2*prob_1
    k_min_dist_arr = result_arr[:, 2].argsort()[::-1][:k]
    y_pred = mode(y_train[k_min_dist_arr])
    y_exp = y_test[test_index]
```

```
    return y_pred, y_exp
y_pred_arr = []
y_exp_arr = []

for test_indx in range(len(X_test)):
    y_pred, y_exp = q_knn_module(test_indx, psi)
    y_pred_arr.append(y_pred)
    y_exp_arr.append(y_exp)
t = 0
f = 0
for i in range(len(X_test)):
    if y_pred_arr[i] == y_exp_arr[i]:
        t += 1
    else:
        f += 1

print('Model accuracy is {}%.'.format(t/(t+f) *100))
```



```
In [18]:
def q_knn_module(test_index, psi, k=10):
    phi_test = X_test[test_index]

    # Create the encoded feature vector
    for i in range(M):
        phi_i = [np.sqrt(phi_test[i]), np.sqrt(1- phi_test[i])]
        if i == 0:
            phi = phi_i
        else:
            phi = np.kron(phi, phi_i)

    # Create the quantum circuit
    qknn = QuantumCircuit()

    # Add registers to circuit
    qknn.add_register(index_reg)
    qknn.add_register(train_reg)
    qknn.add_register(test_reg)
    qknn.add_register(p)

    # Initialize train reg
    qknn.initialize(psi, index_reg[0:N] + train_reg[0:M])

    # Initialize test reg
    qknn.initialize(phi, test_reg)

    # Apply the swap test module to the quantum circuit
    qknn.append(swap_test(M), train_reg[0:M] + test_reg[0:M] + [p[0]])

    # Create the classical register
    meas_reg_len = N+1
    meas_reg = ClassicalRegister(meas_reg_len, 'meas')
    qknn.add_register(meas_reg)

    # Measure the qubits
    qknn.measure(index_reg[0::] + p[0::], meas_reg)

    # Circuit execution
    backend = Aer.get_backend('qasm_simulator')
    counts_knn = execute(qknn, backend, shots = 10000).result().get_counts()

    # Decoding the results
    result_arr = np.zeros((N_train, 3))
    for count in counts_knn:
        i_dec = int(count[1::], 2)
        phase = int(count[0], 2)
        if phase == 0:
            result_arr[i_dec, 0] += counts_knn[count]
        else:
            result_arr[i_dec, 1] += counts_knn[count]
```



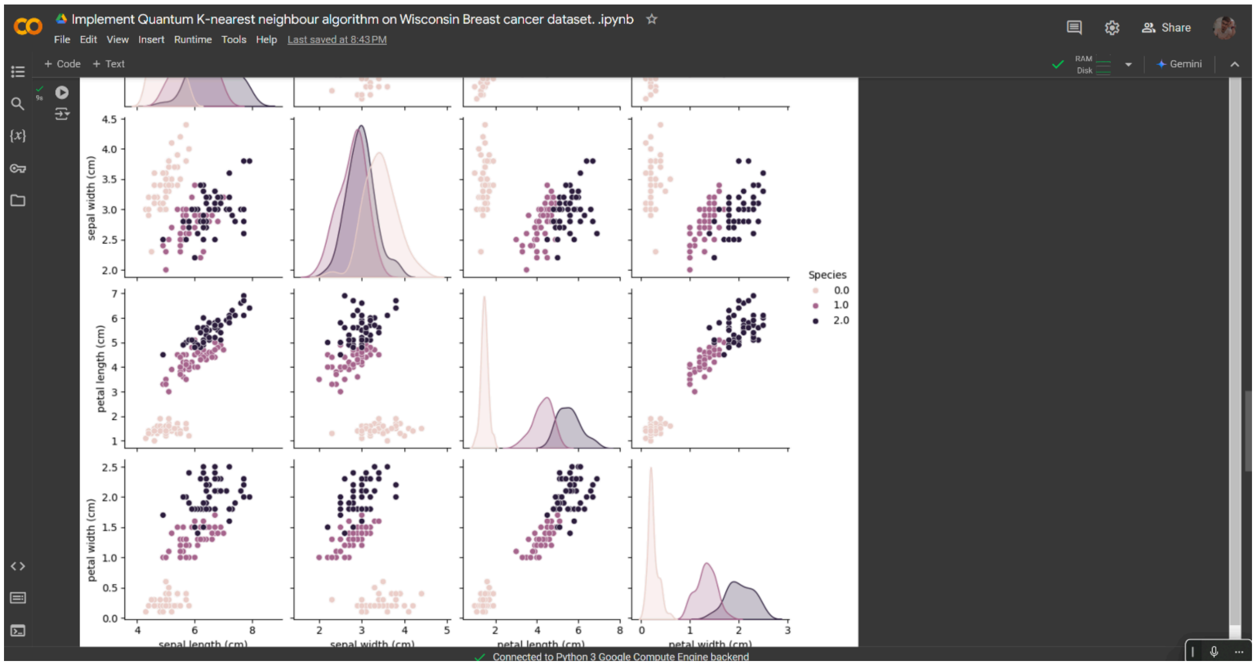
```
    qknn.measure(index_reg[0::] + p[0::], meas_reg)

    # Circuit execution
    backend = Aer.get_backend('qasm_simulator')
    counts_knn = execute(qknn, backend, shots = 10000).result().get_counts()

    # Decoding the results
    result_arr = np.zeros((N_train, 3))
    for count in counts_knn:
        i_dec = int(count[1::], 2)
        phase = int(count[0], 2)
        if phase == 0:
            result_arr[i_dec, 0] += counts_knn[count]
        else:
            result_arr[i_dec, 1] += counts_knn[count]

    # Computing similarity
    for i in range(N_train):
        prob_1 = result_arr[i][1]/(result_arr[i][0] + result_arr[i][1])
        result_arr[i][2] = 1 - 2*prob_1

    # Find the indices of minimum distance
    k_min_dist_arr = result_arr[:, 2].argsort()[::-1][:k]

    # Determine the class of the test sample
    y_pred = mode(y_train[k_min_dist_arr])
    y_exp = y_test[test_index]

    return y_pred, y_exp

y_pred_arr = []
y_exp_arr = []

for test_indx in range(len(X_test)):
    y_pred, y_exp = q_knn_module(test_indx, psi)
    y_pred_arr.append(y_pred)
    y_exp_arr.append(y_exp)

t = 0
f = 0
for i in range(len(X_test)):
    if y_pred_arr[i] == y_exp_arr[i]:
        t += 1
    else:
        f += 1

print('Model accuracy is {}%.'.format(t/(t+f) *100))
```

```
Model accuracy is 62.0%.
```

Results

https://colabhttps://github.com/Probuddhadutta/quantum-computing-.research.google.com/drive/1fexWtUA97e6Xh9Z-ZOf4OiAp9Exm_fZV#scrollTo=yxMgb0LbRwYs

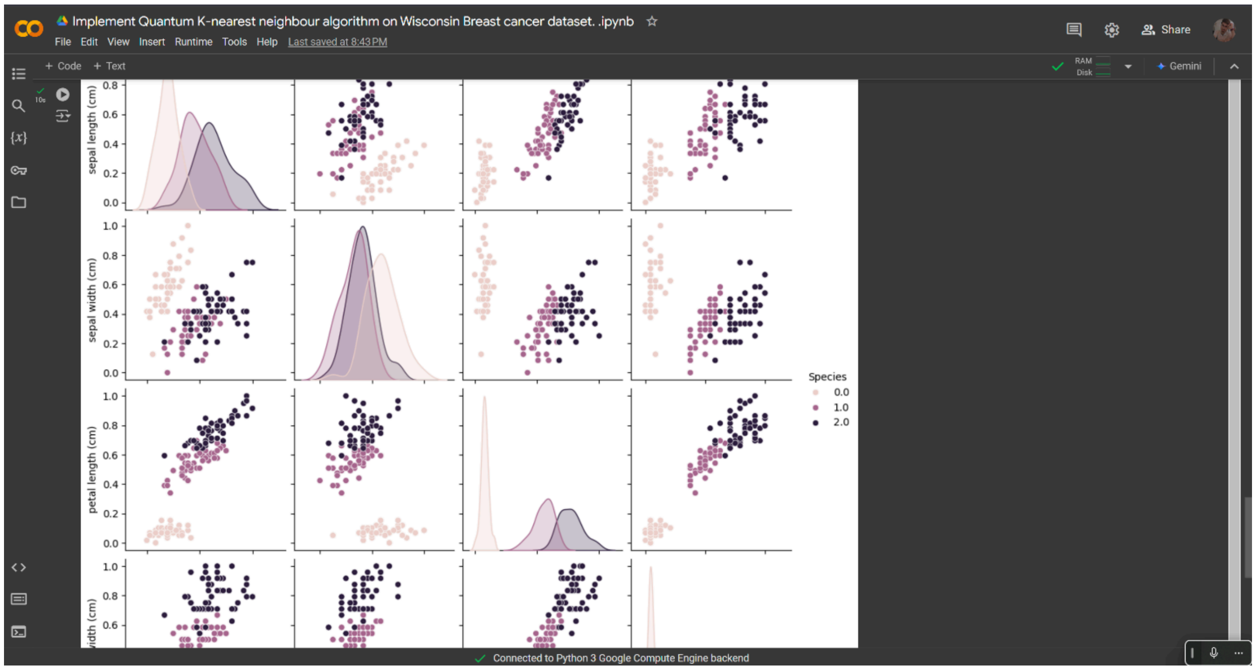
Set 1


Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb

Connected to Python 3 Google Compute Engine backend

Set 2

https://github.com/Probuddhadutta/quantum-computing-


Implement Quantum K-nearest neighbour algorithm on Wisconsin Breast cancer dataset. .ipynb

Connected to Python 3 Google Compute Engine backend

Model accuracy 60.2%