Quantum-CNN synergy: Redefining early predication of Pancreatic Cancer

Problem Statement :

Develop a Quantum Convolutional Neural Network (QCNN) model to accurately diagnose pancreatic cancer using medical imaging and biomarker data.

Data

- Source: https://www.kaggle.com/datasets/johnjdavisiv/urinary-biomarkers-for- pancreatic-cancer
- We have two data files: one contains the information about the biomarker values, and the other contains the medical imaging data used for classification.
    - biomarker_data.csv (ID, Creatinine, Plasma_CA19_9, LYVE1, REG1B, TFF1, Age, Sex, Diagnosis)
    - medical_images (ID, Image Data, Diagnosis Class)
- Both these data files share a common column called ID.

Example Data Point:

- biomarker_data.csv
    - ID: 101
    - Creatinine: 0.9
    - Plasma_CA19_9: 450
    - LYVE1: 3.2
    - REG1B: 120
    - TFF1: 550
    - Age: 65
    - Sex: F
    - Diagnosis: Malignant (Class 1)
- medical_images
    - ID: 101
    - Image Data: [MRI Scan Image]
    - Diagnosis Class: Malignant

Type of Machine Learning Problem

Binary classification problem distinguishing malignant from benign cases.

Performance Metrics:

- Binary cross-entropy loss
- Accuracy, Precision, Recall, F1-score
- Confusion matrix analysis

Machine Learing Objectives and Constraints

Objective: Predict the probability of a patient having pancreatic cancer based on biomarker and imaging data.

Constraints:

- Ensure high interpretability and accuracy.
- Class probabilities are required for decision-making.
- Minimize false negatives to enhance early detection.
- Optimize quantum circuit efficiency to handle medical imaging data.

Exploratory Data Analysis

Importing the necessary Libraries

```
import pennylane as qml
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Reading Data

This dataset contains information from 590 patients:

Healthy controls (183) Patients with non-cancerous pancreatic conditions,like chronic pancreatitis

(208)

Patients with pancreatic cancer (199)

Our goal is to try to predict which patients have pancreatic cancer based on the features. Pancreatic cancer often shows no symptoms until it is too late for effective treatment, so an early test could be valuable.

```
# Load dataset
df = pd.read_csv('/content/data.csv.csv')

# Preprocessing
df['diagnosis'] = df['diagnosis'] == 3  # Convert to binary
classification
df['sex'] = df['sex'].map({'M': 1, 'F': 0})

# Select relevant features
features = ['creatinine', 'plasma_CA19_9', 'age', 'sex', 'LYVE1',
'REG1B', 'TFF1']
target = 'diagnosis'
```

Data Preprocesing

```
df = df[features + [target]].copy()
# Drop rows with NaN values
df = df.dropna()


print(df['diagnosis'].value_counts())
print(df.describe())
df.head(20)
```

```
diagnosis
False    200
True     150
Name: count, dtype: int64
       creatinine  plasma CA19 9          age          sex
LYVE1  \
count  350.000000     350.000000  350.000000  350.000000  350.000000

mean     0.833650     654.002944   59.331429    0.488571    3.199363

std      0.650878    2430.317642   12.988374    0.500585    3.544516

min      0.056550       0.000000   26.000000    0.000000    0.000129

25%      0.361920       8.000000   50.000000    0.000000    0.126749

50%      0.655980      26.500000   61.000000    0.000000    1.886424

75%      1.082933     294.000000   69.000000    1.000000    5.287402

max      4.116840   31000.000000   87.000000    1.000000   23.890323


              REG1B          TFF1
count    350.000000    350.000000
mean     117.811060    672.622797
std      194.361832   1155.171378
min        0.293000      0.005293
25%       11.638096     40.053610
50%       40.581909    327.161512
75%      136.747135    778.645150
max     1215.168000  13344.300000
```

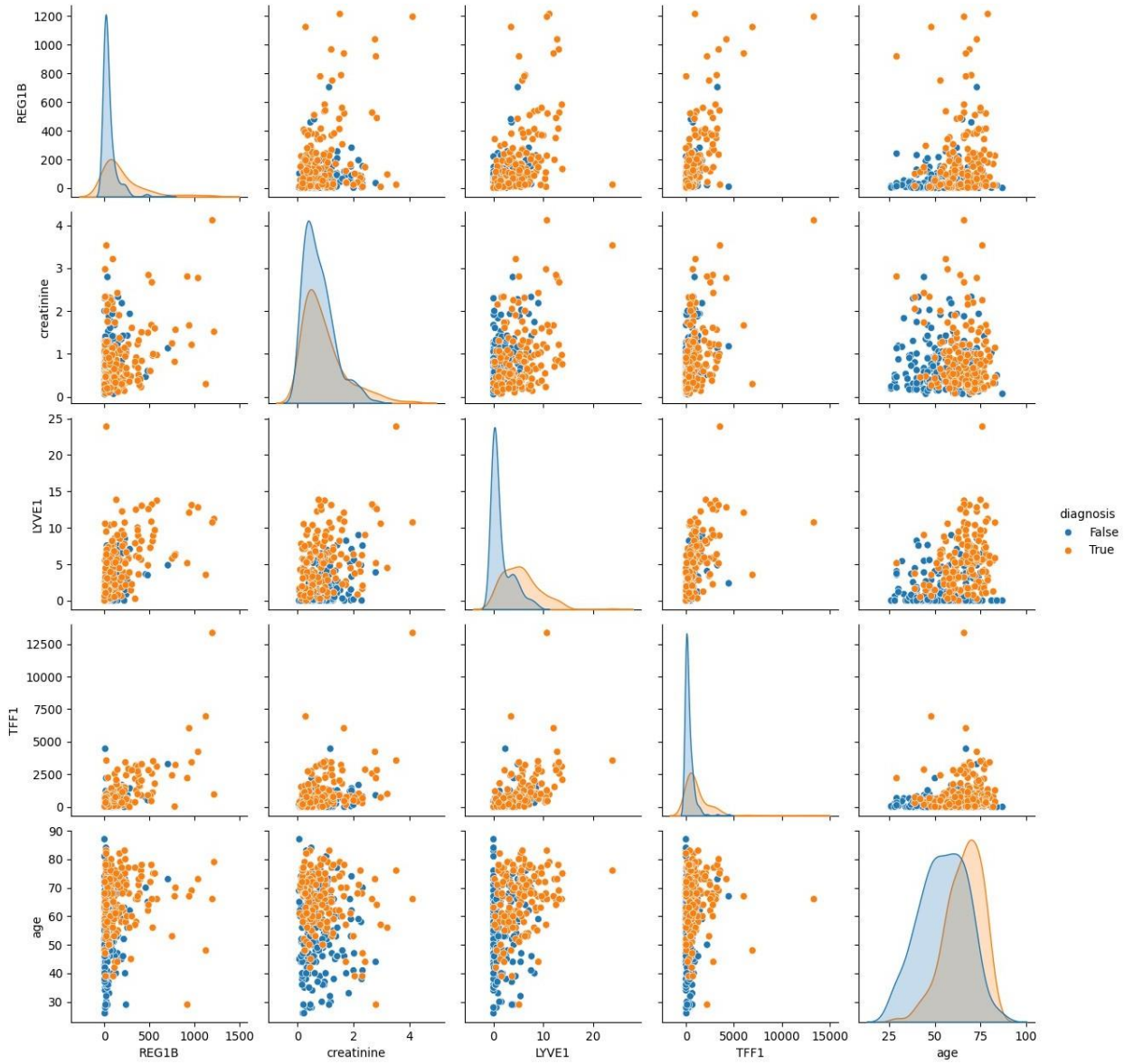

```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 350,\n  \"fields\": [\
n    {\n      \"column\": \"creatinine\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0.6508782912050376,\n
\"min\": 0.05655,\n        \"max\": 4.11684,\n
\"num_unique_values\": 158,\n        \"samples\": [\n
0.10179,\n          1.06314,\n          1.21017\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
```

```
n    },\n    {\n      \"column\": \"plasma_CA19_9\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
2430.3176423108416,\n        \"min\": 0.0,\n        \"max\": 31000.0,\
n        \"num_unique_values\": 266,\n        \"samples\": [\n
318.0,\n          116.0,\n          255.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"age\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 12,\n        \"min\": 26,\n
\"max\": 87,\n        \"num_unique_values\": 58,\n        \"samples\":
[\n          33,\n          56,\n          40\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"sex\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0,\n        \"min\": 0,\n
\"max\": 1,\n\"num_unique_values\": 2,\n                \"samples\":
[\n          1,\n          0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"LYVE1\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 3.5445156217940266,\n        \"min\":
0.00012943,\n        \"max\": 23.890323,\n
\"num_unique_values\": 325,\n        \"samples\": [\n
2.81166,\n          0.00918372\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"REG1B\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 194.36183171103718,\n        \"min\":
0.293,\n        \"max\": 1215.168,\n        \"num_unique_values\":
343,\n        \"samples\": [\n          12.580106,\n
7.141176\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"TFF1\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 1155.171378047606,\n        \"min\": 0.00529308,\n
\"max\": 13344.3,\n        \"num_unique_values\": 332,\n
\"samples\": [\n          0.02558322,\n          2207.57\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"diagnosis\",\n
\"properties\": {\n        \"dtype\": \"boolean\",\n
\"num_unique_values\": 2,\n\"samples\": [\n                  true,\n
false\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n ]\
n}","type":"dataframe","variable_name":"df"}
```

Visualizing The Data

```
sns.pairplot(data = df, vars=['REG1B', 'creatinine', 'LYVE1', 'TFF1',
'age'], hue='diagnosis', diag_kind='kde')
```

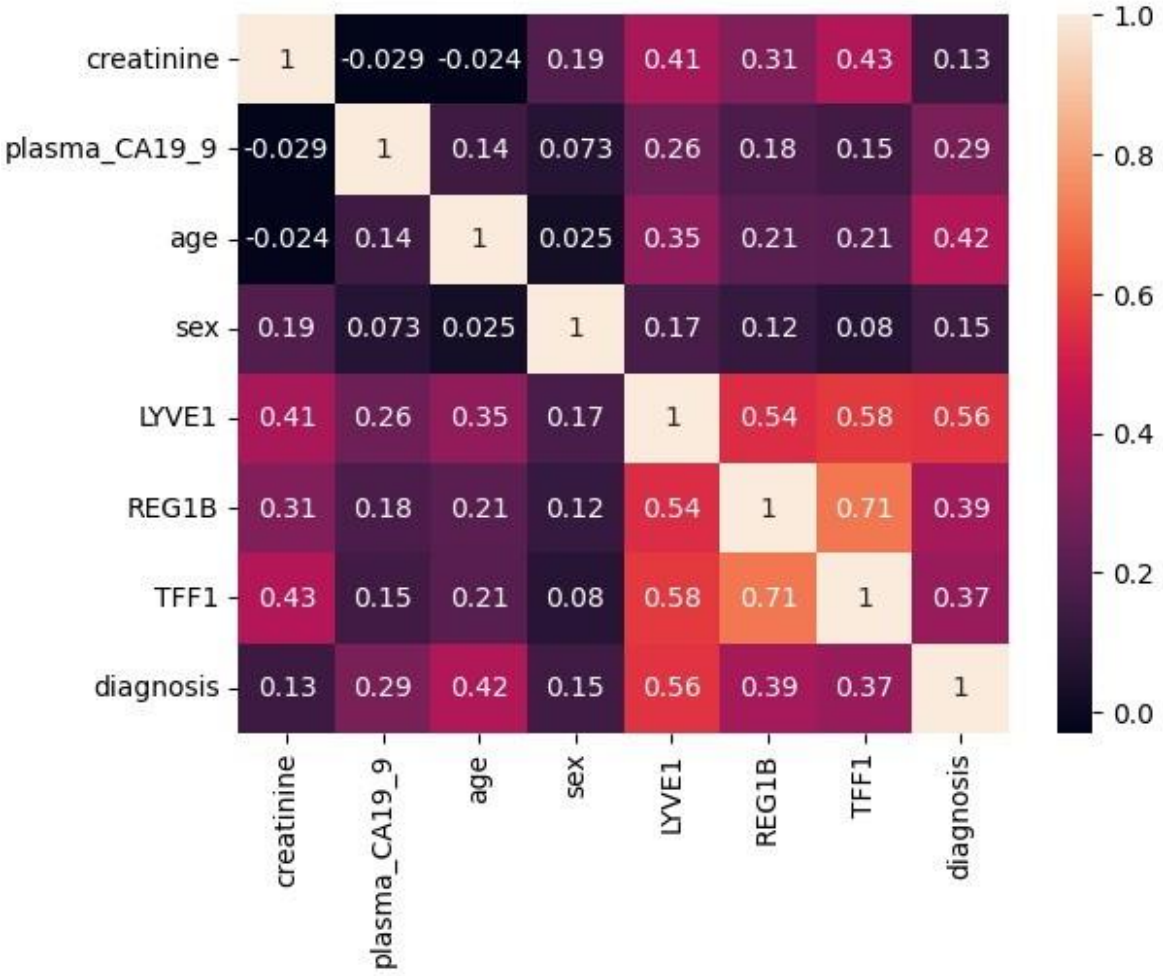
```
<seaborn.axisgrid.PairGrid at 0x7e685297a7d0>
```



It seems that there is a higher frequency of low values for REGB1 in negative diagnoses (FALSE) than in positive diagnoses (TRUE). Similarly, in the case of TFF1, the values appear to be higher in positive diagnoses, and small values seem to be more frequent in negative diagnoses. The distribution of LYVE1 values with respect to the diagnosis looks promising, as there is a higher probability of high values in the case of a positive diagnosis (TRUE), and a higher probability of low values in the case of negative diagnoses (FALSE) and the distributions are more separated.

```
corr = df.dropna().corr()
sns.heatmap(corr, annot=True)

<Axes: >
```



Spliting the data into training and testing data

```
# Train-test split
X = df[features].values
y = df[target].values.astype(np.float32)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)

# Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
y_test = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)
```

```
print(f"X_train shape: {X_train.shape}")  # Expected: (batch_size, 7)
print(f"y_train shape: {y_train.shape}")  # Expected: (batch_size, 1)
```

```
X_train shape: torch.Size([280, 7])
y_train shape: torch.Size([280, 1])
```

Defining the QCNN

```
# Quantum device
n_qubits = len(features)  # One qubit per feature
dev = qml.device("default.qubit", wires=n_qubits)

# Define Quantum Circuit
def quantum_circuit(inputs, weights):
    qml.AngleEmbedding(inputs, wires=range(n_qubits))
    qml.BasicEntanglerLayers(weights, wires=range(n_qubits))
    return qml.expval(qml.PauliZ(0))  # Measure qubit 0

# QNode
weight_shapes = {"weights": (3, n_qubits)}
qnode = qml.QNode(quantum_circuit, dev, interface="torch",
diff_method="best")

# Quantum Layer
qlayer = qml.qnn.TorchLayer(qnode, weight_shapes)

# Define Hybrid Model
class HybridQNN(nn.Module):
    def __init__(self):  # Fix the typo here
        super().__init__()
        self.fc1 = nn.Linear(7, 7)  # Ensure input and output are both
7
        self.qlayer = qlayer
        self.fc2 = nn.Linear(1, 1)  # Ensure the output layer matches
the quantum layer output
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = torch.tanh(self.fc1(x))  # Apply non-linearity
        x = self.qlayer(x).reshape(-1, 1)  # Ensure quantum output is
correctly shaped
        x = self.sigmoid(self.fc2(x))  # Ensure final output is
between 0 and 1
        return x
```

Training The model

```
# Model, Loss, and Optimizer
model = HybridQNN()
```

```
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training Loop
epochs = 100
for epoch in range(epochs):
    optimizer.zero_grad()
    y_pred = model(X_train)

   # print(f"y_pred min: {y_pred.min()}, max: {y_pred.max()}")
    #print(f"y_train min: {y_train.min()}, max: {y_train.max()}")

    loss = criterion(y_pred, y_train)

    loss.backward()
    optimizer.step()

    if epoch % 10 == 0:
        print(f"Epoch {epoch}: Loss = {loss.item():.4f}")
```

```
Epoch 0: Loss = 0.6979
Epoch 10: Loss = 0.6578
Epoch 20: Loss = 0.6321
Epoch 30: Loss = 0.6079
Epoch 40: Loss = 0.5874
Epoch 50: Loss = 0.5681
Epoch 60: Loss = 0.5498
Epoch 70: Loss = 0.5293
Epoch 80: Loss = 0.5044
Epoch 90: Loss = 0.4806
```

Tesing The QCNN model

```
# Evaluation
y_pred_test = model(X_test).detach().numpy()
y_pred_test = (y_pred_test > 0.5).astype(int)

accuracy = np.mean(y_pred_test == y_test.numpy())
print(f"\nTest Accuracy: {accuracy:.4f}")
```

```
Test Accuracy: 0.8714
```

Saving The Model

```
torch.save(model.state_dict(), "pancreas_detection.pth")
```

Saving the operations that are made on the input data

```
import joblib

# Save the fitted scaler after training
joblib.dump(scaler, "scaler.pkl")
print("Scaler saved successfully!")
```

```
Scaler saved successfully!
```

Identifying Importance of Each Feature

```
import numpy as np
import matplotlib.pyplot as plt
import torch

# Make sure model and X_test are tensors
feature_names = features  # Replace with actual names

def compute_feature_importance(model, X_test, perturbation=0.1):
    model.eval()
    base_predictions = model(X_test).detach().numpy().flatten()  #
Remove `.values`
    importances = []

    for i in range(X_test.shape[1]):  # Iterate over features
        X_perturbed = X_test.clone().detach()  # Clone to avoid
modifying the original tensor
        X_perturbed[:, i] += perturbation  # Slightly perturb feature
i
        perturbed_predictions =
model(X_perturbed).detach().numpy().flatten()

        # Compute importance as mean absolute difference in
predictions
        importance = np.mean(np.abs(perturbed_predictions -
base_predictions))
        importances.append(importance)

    return np.array(importances)

# Compute importance scores
importances = compute_feature_importance(model, X_test)

# Sort and plot
indices = np.argsort(importances)[::-1]
plt.figure(figsize=(10,  8))
plt.title("Feature Importances (Approximate)")
plt.barh(range(len(importances)), importances[indices], color="red",
align="center")
plt.yticks(range(len(importances)), np.array(feature_names)[indices])
# Use actual feature names
```

```
plt.gca().invert_yaxis()
plt.xlabel("Importance")
plt.show()
```