National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Sahil Mulani
Student ID: x22234144

School of Computing
National College of Ireland

Supervisor: Paul Stynes

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Sahil Mulani
Student ID:	x22234144
Programme:	Data Analytics
Year:	2024
Module:	MSc Research Project
Supervisor:	Paul Stynes
Submission Due Date:	12/09/2024
Project Title:	Configuration Manual
Word Count:	744
Page Count:	13

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Sahil Mulani
Date:	11th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Sahil Mulani
x22234144

The configuration Manual will provide a step by step guidance of the project in terms of installation, implementation, development and deployment for the research project of "Comparative performance analysis of Machine Learning with Quantum Machine Learning for breast cancer prediction".

1 System Requirements :

1.1 Hardware Requirements :

- Processor : Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz.
- Installed RAM : 16.0 GB (15.9 GB usable)
- System type : 64-bit operating system, x64-based processor
- Operation System : Windows 11 , 23H2 (Version), 22631.3880 (OS build)

1.2 Software Requirements :

- Python : Version Python 3.7 or higher is recommended.
- Jupyter Notebook(Optional but Recommended) : Provides an interactive environment to develop and visualize the plots.
- IDE : Visual Studio Code or Pycharm can be used to write and execute Python scripts
- Dependencies Management (Optional) : Virtualenv or Conda can be used to manage dependencies and maintain an isolated environment for the project.

2 Runtime Installation Pre-requisites

2.1 Jupyter Notebooks Local Installation

Installation of the Anaconda Python distribution is recommended as it is the easiest way to set up JupyterLab. Please refer to `https://www.anaconda.com/products/individual` for step-by-step guidance on installing Anaconda. For information on installing JupyterLab as a standalone application, visit `https://jupyter.org/install`. To access IBM quantum processing backends locally, an IBM quantum account and a personal access API token are required. For more information, please visit `https://quantum-computing.ibm.com/` for further information.

2.2 Jupyter Notebooks using Google Colaboratory

Jupyter Notebooks can be accessed using the Google Colaboratory. Please visit `https://colab.google/` for further information on setting up Jupyter notebooks on Google Colab. Even while executing the code on Jupyter notebooks via Google Colab, an IBM quantum account and a personal access API token are required to access the IBM quantum processing backends. Please visit `https://quantum-computing.ibm.com/` for further information.

3 Data Preparation

In this research, the Breast Cancer UCI Machine Learning dataset [1] was used.

```
# Loading data
df = pd.read_csv('data.csv')
#removing empty column
df=df.dropna(axis=1)
df['diagnosis'].replace({'M': 1, 'B': 0}, inplace=True)
```

Figure 1: Data Preparation

As shown in Figure 1 categorical values in the 'diagnosis' column are being replaced by numerical values.

```
# Split data set into training and testing
X_train, X_test, Y_train, Y_test = train_test_split(X_stat, Y, test_size=0.25, random_state=42)

# Scaling the data (feature scaling)
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Figure 2: Splitting the Data

As shown in FIgure 2 the data set is divided into training and testing sets using the `train_test_split` function. `X_stat`. The `test_size=0.25` parameter indicates that 25% of the data will be used for testing, and `random_state=42` ensures that the split is reproducible. The features in the training and testing sets are scaled using `StandardScaler`. Figure 3 shows a implementation code for feature selection in which a t-test is performed for each feature to determine if there is a statistically significant difference between the means of benign and malignant cases.

[1] `https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic`

```python
# Perform a t-test to determine if the means of these features are significantly
# different for benign and malignant cases
t_test_results = []

for feature in range(X_stat.shape[1]):
    benign_values = X_stat[Y == 0, feature]
    malignant_values = X_stat[Y == 1, feature]

    # Perform a t-test to compare the means of benign and malignant cases
    t_stat, p_value = ttest_ind(benign_values, malignant_values)

    t_test_results.append([t_stat, p_value])

# Convert t_test_results to a NumPy array
t_test_results = np.array(t_test_results)

# Choose a significance level (e.g., 0.05) to determine if a feature is statistically significant
significance_level = 0.05
significant_features = [i for i, (_, p_value) in enumerate(t_test_results) if p_value < significance_level]
```

Figure 3: Feature Selection

4 Implementation Codes for Machine Learning Models

In this research project, we have applied four machine learning models - Logistic Regression, Random Forest, SVM and KNN.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
```

Figure 4: Importing the Machine Learning Models

Figure 4 shows the libraries imported from sklearn to implement all the machine learning models.

```python
def train_models(X_train_selected, Y_train):
    # Initialize a list to store trained models
    models = []
    # Logistic Regression
    log = LogisticRegression(random_state=42)
    log.fit(X_train_selected, Y_train)
    models.append(log)
    # Random Forest Classifier
    forest = RandomForestClassifier(n_estimators=10, criterion='entropy', random_state=42)
    forest.fit(X_train_selected, Y_train)
    models.append(forest)
    # SVM Linear
    svc_model = SVC(kernel='linear', random_state=42)
    svc_model.fit(X_train_selected, Y_train)
    models.append(svc_model)
    # KNeighbors Classifier
    knn = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2)
    knn.fit(X_train_selected, Y_train)
    models.append(knn)
    return models
```

Figure 5: Training Machine Learning Models

As shown in Figure 5, the `train_models()` which consists of implementation code for training all the machine learning models.

```python
from sklearn.metrics import confusion_matrix
for i in range(len(model)):
  print('model:',i)
  cm=confusion_matrix(Y_test,model[i].predict(X_test_selected))
  tp=cm[0][0]
  tn=cm[1][1]
  fn=cm[1][0]
  fp=cm[0][1]

  print(cm)
  print('Testing Accuracy:',(tp+tn)/(tp+tn+fn+fp))
  print()
```

Figure 6: Evaluating the Models

Figure 6 shows the implementation code for evluating the performance of the machine learning models.

5 Implementation Code for the Deep Learning Models

In this research project, we have applied four deep learning models - artificial neural network, convolutional neural network, recurrent neural network, and multiple layer perceptron.

5.1 Implementation Code of ANN

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.callbacks import EarlyStopping

ANN = tf.keras.Sequential([
  layers.Dense(24,activation='relu'),
  layers.Dense(8,activation='relu'),
  layers.Dense(16,activation='relu'),
  layers.Dense(64,activation='relu'),
  layers.Dense(16,activation='relu'),
  layers.Dense(8,activation='relu'),
  layers.Dense(1,activation='sigmoid')
])

ANN.compile(loss = 'binary_crossentropy',optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005),metrics=['accuracy'])

early=EarlyStopping(monitor='val_loss',mode='min',patience=20)

ANN.fit(X_train, Y_train, epochs=100, validation_data=(X_test,Y_test),callbacks=[early])
```

Figure 7: Implementation Code of ANN

Figure 7 shows the implementation code for the artificial neural network.

5.2 Implementation Code of CNN

```
from keras._tf_keras.keras.layers import Conv2D, MaxPooling2D
from keras._tf_keras.keras.layers import Conv1D


epochs = 100
model = Sequential()
model.add(Conv1D(filters=32, kernel_size=2, activation='relu', input_shape = (30,1)))

model.add(BatchNormalization())
model.add(Dropout(0.2))

model.add(Conv1D(filters=64, kernel_size=2, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

Figure 8: Implementation code of CNN

Figure 7 shows the implementation code for the convolutional neural network.

```
from tensorflow.keras.optimizers import Adam

model.compile(optimizer=Adam(learning_rate=0.00005),
              loss='binary_crossentropy',
              metrics=['accuracy'])

CNN = model.fit(X_train, Y_train, epochs=epochs, validation_data=(X_test, Y_test), verbose=1)
```

Figure 9: Compilation of CNN

Figure 9 shows the compilation and evaluation code for the convolutional neural network.

5.3 Implementation Code of RNN

```
#Reshape the data for RNN input (time steps, features)
time_steps=X_train.shape[1]
X_train=X_train.reshape(-1, time_steps, 1)
X_test=X_test.reshape(-1, time_steps, 1)

# Convert labels to float32 and reshape them
y_train=y_train.astype(np.float32).reshape(-1, 1)
y_test=y_test.astype(np.float32).reshape(-1, 1)
```

Figure 10: Data Preparation for RNN

Figure 10 shows the implementation code of the data preparation for recurrent neural network and figure 11 shows model implementation and evaluation code.

```
# Build the RNN model
model=Sequential()
model.add(SimpleRNN(64,activation='relu', input_shape=(time_steps, 1)))
model.add(Dense(1,activation='sigmoid'))
#Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.1)
# Evaluate the model on the test set
loss, accuracy=model.evaluate(X_test,y_test)
print(f'Test accuracy: {accuracy}')
```

Figure 11: Implementation Code of RNN

5.4 Implementation Code for MLP

```
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Build the custom MLP model
model = Sequential()
model.add(Dense(8, input_dim=X_train.shape[1], activation='selu'))
model.add(Dense(16, activation='selu'))
model.add(Dense(32, activation='selu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Figure 12: Implementation of Multilayer Perceptron

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
MLP = model.fit(X_train, Y_train, epochs=100, batch_size=10, validation_split=0.2, verbose=1)


# Evaluate the model
loss, accuracy = model.evaluate(X_test, Y_test, verbose=0)
print(f'Test Accuracy: {accuracy*100:.2f}%')
```

Figure 13: Evaluation of MLP

Figure 12 shows the implementation code for recurrent neural network and figure 13 shows the model evaluation code.

6 Implementation Code of Quantum Machine Models

In this research project, we have applied two quantum machine learning models, the variable quantum classifier and the quantum support vector.

As shown in figure 14, the `qiskit_machine_learning` library needs to be imported to install the necessary dependencies required to execute the quantum machine learning models.

```
!pip install qiskit pylatexenc qiskit_machine_learning
```

Figure 14: Importing Qiskit Machine Learning

6.1 Implementation Code of Variational Quantum Classifier

```
from qiskit.circuit.library import ZZFeatureMap

num_features = features_dataset.shape[1]

feature_map = ZZFeatureMap(feature_dimension=num_features, reps=1)
feature_map.decompose().draw(output="mpl")
```

Figure 15: Implementation code of ZZFeatureMap

Figure 15 shows the implementation code for data encoding using the **ZZFeatureMap**.

```
from qiskit.circuit.library import PauliFeatureMap

num_features = features_dataset.shape[1]

feature_map = PauliFeatureMap(feature_dimension=num_features, reps=1)
feature_map.decompose().draw(output="mpl")
```

Figure 16: Implementation code of PauliFeatureMap

Figure 16 shows the implementation code for data encoding using the **PauliFeatureMap**.

```
from qiskit.circuit.library import ZFeatureMap

num_features = features_dataset.shape[1]

feature_map = ZFeatureMap(feature_dimension=num_features, reps=1)
feature_map.decompose().draw(output="mpl")
```

Figure 17: Implementation code of ZFeatureMap

Figure 17 shows the implementation code for data encoding using the **ZFeatureMap**.

```
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(num_qubits=num_features, reps=3)
ansatz.decompose().draw(output="mpl")
```

Figure 18: Implementation Code of EfficentSU2 ansatz

```
from qiskit.circuit.library import RealAmplitudes

ansatz = RealAmplitudes(num_qubits=num_features, reps=3)
ansatz.decompose().draw(output="mpl")
```

Figure 19: Implementation Code of Real Amplitudes ansatz

```python
from qiskit_algorithms.optimizers import COBYLA

optimizer = COBYLA(maxiter=100)

from qiskit.primitives import Sampler

sampler = Sampler()
```

Figure 20: Implementation code of Optimizer and Sampler

```python
from matplotlib import pyplot as plt
from IPython.display import clear_output

objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)


def callback_graph(weights, obj_func_eval):
    clear_output(wait=True)
    objective_func_vals.append(obj_func_eval)
    plt.title("Objective function value against iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective function value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.show()
```

Figure 21: Implementation Code of Callback Graph

```python
import time
import numpy as np


from qiskit_machine_learning.algorithms.classifiers import VQC

vqc = VQC(
    sampler=sampler,
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=optimizer,
    callback=callback_graph,
)

# clear objective value history
objective_func_vals = []

# Convert pandas Series to numpy arrays
train_features = np.array(train_features)
train_labels = np.array(train_labels)

start = time.time()
vqc.fit(train_features, train_labels)
elapsed = time.time() - start

print(f"Training time: {round(elapsed)} seconds")
```

Figure 22: Implementation Code of Varitational Quantum Classifer

6.2 Implementation of Quantum Support vector

```python
from qiskit.circuit.library import ZZFeatureMap

from qiskit_machine_learning.kernels import FidelityQuantumKernel

algorithm_globals.random_seed = 12345

feature_map = ZZFeatureMap(feature_dimension=num_qubits, reps=1)
feature_map.decompose().draw(output="mpl")

qkernel = FidelityQuantumKernel(feature_map=feature_map)
```

Figure 23: Encoding the Data & Generating the Fidelity Quantum Kernel

```python
from qiskit_machine_learning.algorithms import PegasosQSVC

pegasos_qsvc = PegasosQSVC(quantum_kernel=qkernel, C=C, num_steps=tau)

# training
pegasos_qsvc.fit(train_features, train_labels)
pegasos_score_train = pegasos_qsvc.score(train_features, train_labels)
print(f"QSVC Training Accuracy: {pegasos_score_train}")
# testing
pegasos_score = pegasos_qsvc.score(test_features, test_labels)
print(f"QSVC Testing Accuracy: {pegasos_score}")
```

Figure 24: Implementation Code of Quantum Support Vector Classifier

References