In [1]:
```python
# Initialisation
import numpy as np

# import Qiskit
from qiskit import IBMQ, BasicAer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, execute
from qiskit.tools.jupyter import *
provider = IBMQ.load_account()

# Import basic plot tools.
from qiskit.visualization import plot_histogram
```

```
/var/folders/t0/7r1s1xhd3r588zrbxd8d1pk80000gn/T/ipykernel_58872/4258817026.py:9: DeprecationWarning: The qiskit.IBMQ entrypoint and the qiskit-ibmq-provider package (accessible from 'qiskit.providers.ibmq`) are deprecated and will be removed in a future release. Instead you should use the qiskit-ibm-provider package which is accessible from 'qiskit_ibm_provider'. You can install it with 'pip install qiskit_ibm_provider'. Just replace 'qiskit.IBMQ' with 'qiskit_ibm_provider.IBMProvider'
  provider = IBMQ.load_account()
```

In [2]:
```python
from sklearn.datasets import load_breast_cancer
breast_cancer_data = load_breast_cancer()
```

In [3]:
```python
print(breast_cancer_data.DESCR)
```

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

:Number of Instances: 569

:Number of Attributes: 30 numeric, predictive attributes and the class

:Attribute Information:
    - radius (mean of distances from center to points on the perimeter)
    - texture (standard deviation of gray-scale values)
    - perimeter
    - area
    - smoothness (local variation in radius lengths)
    - compactness (perimeter^2 / area - 1.0)
    - concavity (severity of concave portions of the contour)
    - concave points (number of concave portions of the contour)
    - symmetry
    - fractal dimension ("coastline approximation" - 1)

    The mean, standard error, and "worst" or largest (mean of the three
    worst/largest values) of these features were computed for each image,
    resulting in 30 features.  For instance, field 0 is Mean Radius, field
    10 is Radius SE, field 20 is Worst Radius.

    - class:
            - WDBC-Malignant
            - WDBC-Benign

:Summary Statistics:

===================================== ====== ======
                                        Min    Max
===================================== ====== ======
radius (mean):                        6.981  28.11
texture (mean):                       9.71   39.28
perimeter (mean):                     43.79  188.5
area (mean):                          143.5  2501.0
smoothness (mean):                    0.053  0.163
compactness (mean):                   0.019  0.345
concavity (mean):                     0.0    0.427
concave points (mean):                0.0    0.201
symmetry (mean):                      0.106  0.304
fractal dimension (mean):             0.05   0.097
radius (standard error):              0.112  2.873
texture (standard error):             0.36   4.885
perimeter (standard error):           0.757  21.98
area (standard error):                6.802  542.2
smoothness (standard error):          0.002  0.031
compactness (standard error):         0.002  0.135
concavity (standard error):           0.0    0.396
concave points (standard error):      0.0    0.053
symmetry (standard error):            0.008  0.079
fractal dimension (standard error):   0.001  0.03
radius (worst):                       7.93   36.04
texture (worst):                      12.02  49.54
perimeter (worst):                    50.41  251.2
area (worst):                         185.2  4254.0
smoothness (worst):                   0.071  0.223
compactness (worst):                  0.027  1.058
concavity (worst):                    0.0    1.252
concave points (worst):               0.0    0.291
symmetry (worst):                     0.156  0.664
fractal dimension (worst):            0.055  0.208
===================================== ====== ======

:Missing Attribute Values: None

:Class Distribution: 212 - Malignant, 357 - Benign

:Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

:Donor: Nick Street

:Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

|details-start|
**References**
|details-split|

- W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction
  for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on
  Electronic Imaging: Science and Technology, volume 1905, pages 861-870,
  San Jose, CA, 1993.
- O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and
  prognosis via linear programming. Operations Research, 43(4), pages 570-577,
  July-August 1995.
- W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning techniques
  to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77 (1994)
  163-171.

|details-end|
```

Breast Cancer Wisconsin (Diagnostic) Dataset Summary:

- 569 instances, 30 numeric predictive attributes plus class.
- Attributes include various measurements like radius, texture, perimeter, area, and more, computed as mean, standard error, and "worst."
- Two classes: Malignant (212 instances) and Benign (357 instances), showing an imbalanced distribution.
- Features describe cell nuclei characteristics from FNA breast mass images.
- Features span a wide range of values, indicating scaling may be necessary for machine learning.

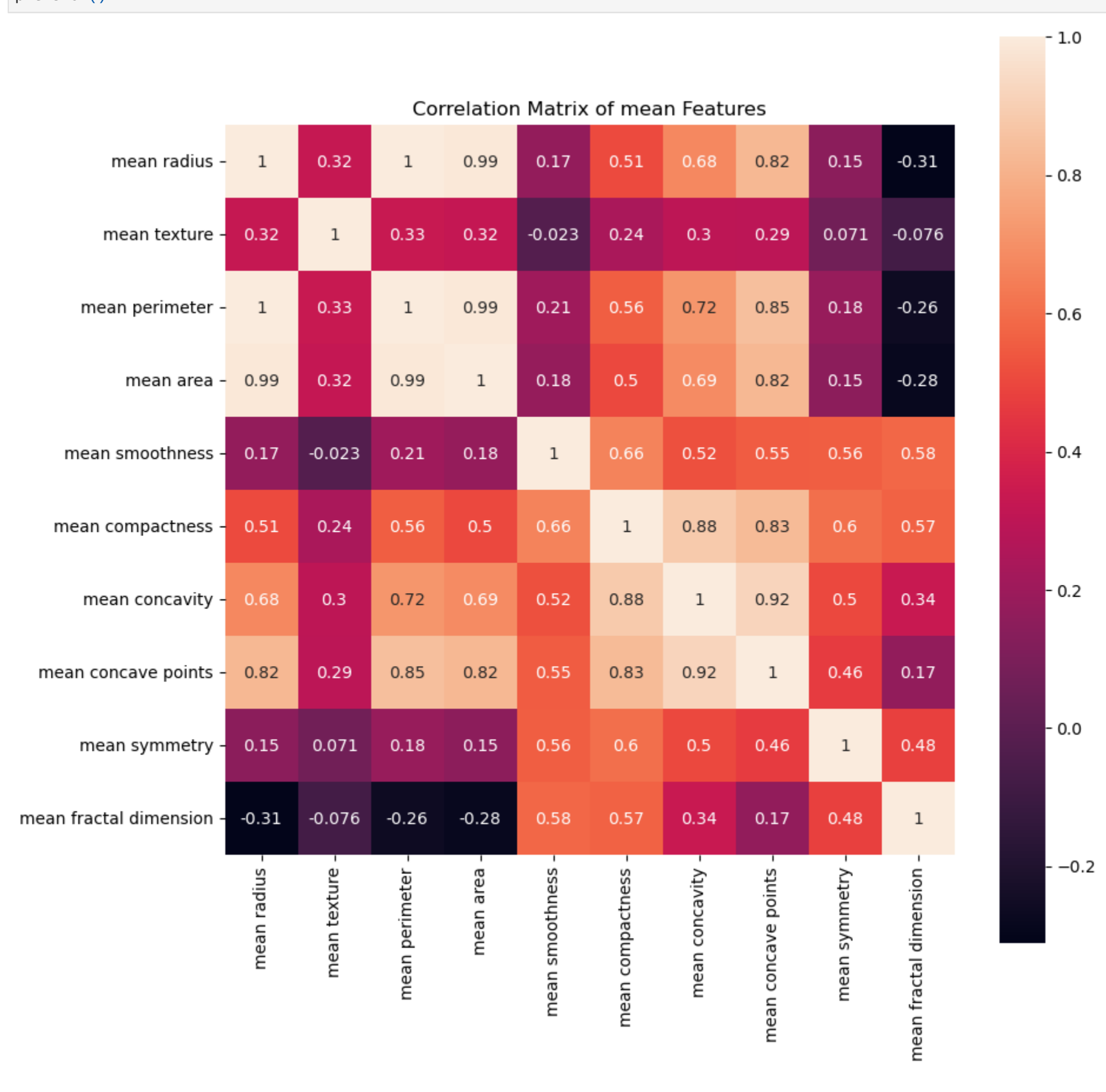
In [4]:
```python
features = breast_cancer_data.data
labels = breast_cancer_data.target
```

Normalize the features; transform all the features onto 0-1.

In [5]:
```python
from sklearn.preprocessing import MinMaxScaler
features = MinMaxScaler().fit_transform(features)
```

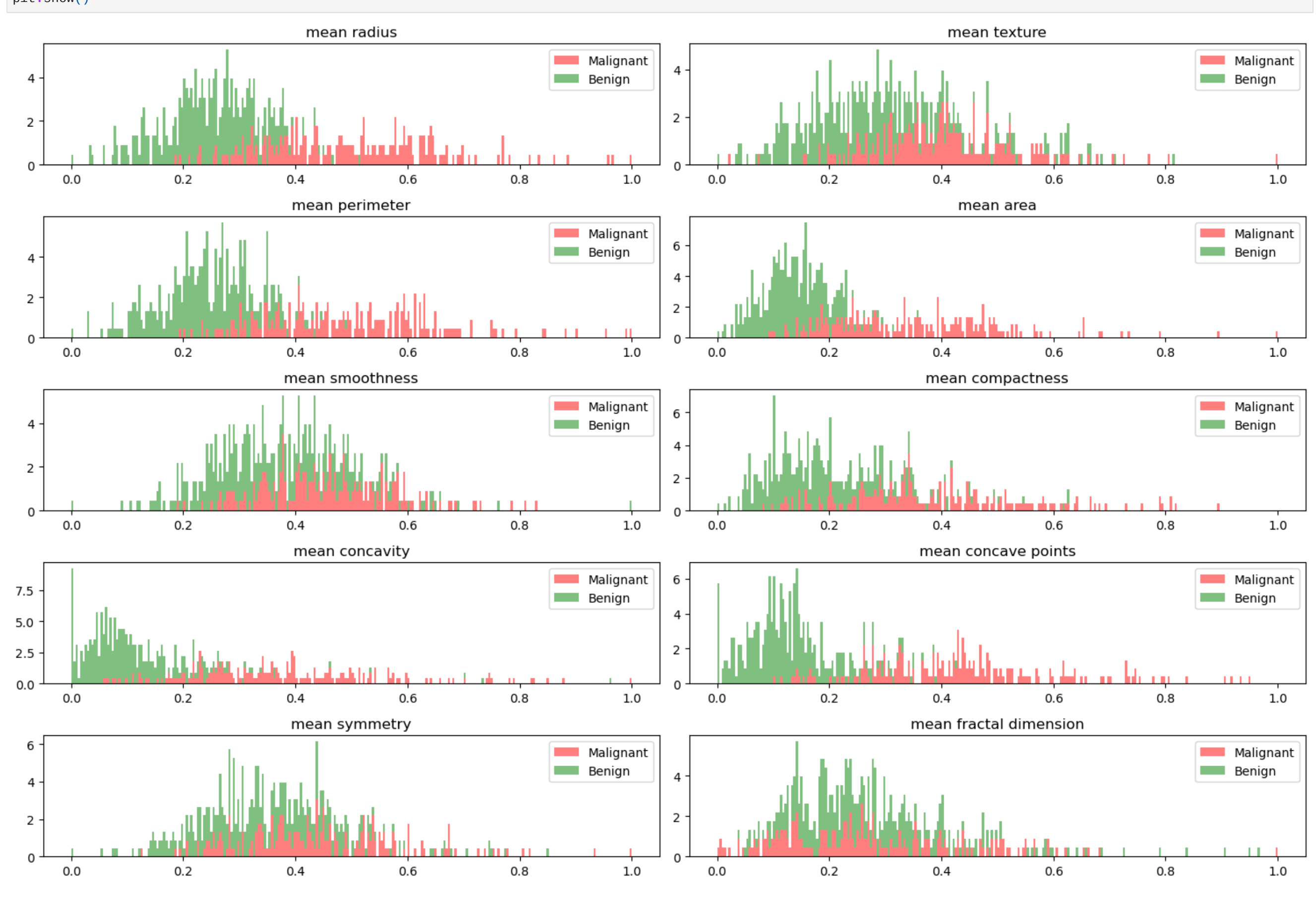
In [6]:
```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.DataFrame(features, columns=breast_cancer_data.feature_names)
df['class'] = [breast_cancer_data.target_names[label] for label in labels]
```

In [9]:
```python
feature_names = breast_cancer_data.feature_names
feature_means = feature_names[:10]
plt.figure(figsize = (10,10))
heat = sns.heatmap(df[feature_means].corr(), vmax=1, square=True, annot=True)
plt.title('Correlation Matrix of mean Features')
plt.savefig('mean_features_corr.png')
plt.show()
```



There is some obvious covariance between some feature, particularly mean radius, mean perimeter, and mean area, which make sense being geometric values.

In [13]:
```python
feature_names = breast_cancer_data.feature_names
feature_means = feature_names[:10]

# Correcting the number of rows in the subplots to match the first quarter of the features
fig, axes = plt.subplots(nrows=5, ncols=2, figsize = (15,10))
axes = axes.flatten()  # Flatten to ensure it's iterable

dataMalignant = df[df['class'] == 'malignant']
dataBenign = df[df['class'] == 'benign']

for idx, ax in enumerate(axes):
    feature = feature_means[idx]
    binwidth = (df[feature].max() - df[feature].min()) / 250
    # Create histograms
    ax.hist([dataMalignant[feature], dataBenign[feature]],
            bins=np.arange(df[feature].min(), df[feature].max() + binwidth, binwidth),
            alpha=0.5, stacked=True, density=True, label=['Malignant', 'Benign'], color=['r', 'g'])
    ax.legend(loc='upper right')
    ax.set_title(feature)

plt.tight_layout()
plt.savefig('mean_features_hist.png')
plt.show()
```



Training a classical model

Training a machine learning model generally involves feeding the model a section of the data, inlcuding labels, this is called 'Supervised Learning'. This will allow the model to gain an understanding of what feature values line up with the classification or label values. In general, 80% of the data set will be used for training. The other 20% will be called the 'test set'. These will be from the same data set, but will have the target label removed. It is then up to the machine learning model to attempt to classify the new data. We will then check what the model predicted for the test data against what the correct labels are to quantify how good the model will be at classifying new data.

Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression tasks. However, it is more commonly used in classification problems.

Key Features:

- **Classification Boundaries**: SVMs are best known for their ability to create non-linear boundaries with the help of kernel functions.
- **Margin Maximization**: The core idea of SVM is to identify the best decision boundary (hyperplane) that separates classes in the feature space. This boundary is chosen to be the one that has the maximum margin, i.e., the maximum distance between data points of both classes.

Applications:

- SVMs are used in applications like face detection, handwriting recognition, image classification, and many areas of biology and physics.

Advantages:

Versatility: Different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

Disadvantages:

- **Sensitivity to Noisy Data**: SVMs are sensitive to the type of kernel used and can overfit if the data is very noisy.

In [14]:
```python
from sklearn.model_selection import train_test_split
from qiskit_algorithms.utils import algorithm_globals

algorithm_globals.random_seed = 4701

train_features, test_features, train_labels, test_labels = train_test_split(
    features, labels, train_size = 0.8, random_state = algorithm_globals.random_seed)
```

In [15]:
```python
from sklearn.svm import SVC

svc = SVC()
_ = svc.fit(train_features, train_labels)  # suppress printing the return value
```

In [16]:
```python
train_score_c4 = svc.score(train_features, train_labels)
test_score_c4 = svc.score(test_features, test_labels)

print(f"Classical SVC on the training dataset: {train_score_c4:.2f}")
print(f"Classical SVC on the test dataset:     {test_score_c4:.2f}")
```

```
Classical SVC on the training dataset: 0.98
Classical SVC on the test dataset:     0.98
```

Training a QML Model

We are training a 'Variational Quanum Classifier', or VQC. It takes a map and an ansatz and constructs a quantum neural network automatically. In the simplest case it is enough to pass the number of qubits and a quantum instance to construct a valid classifier.

In [18]:
```python
from qiskit.circuit.library import ZZFeatureMap

num_features = features.shape[1]

feature_map = ZZFeatureMap(feature_dimension=num_features, reps = 1)
#feature_map.decompose().draw('mpl', fold=50)
```

ZZFeatureMap

The ZZFeatureMap specifically involves the application of Z⊗Z interactions, where "Z" represents the Pauli-Z operator, applied to pairs of qubits. These interactions, combined with single-qubit rotations that encode the input data, create entangled quantum states that reflect the structure of the input data. The name ZZFeatureMap comes from the use of these ZZ interactions, which are key to the map's ability to capture and exploit correlations in the input data in a way that is uniquely quantum.

x[0] ,,, x[3] are placehoders for the features.

In [41]:
```python
from qiskit.circuit.library import RealAmplitudes

ansatz = RealAmplitudes(num_qubits=num_features, reps = 3)
#ansatz.decompose().draw('mpl', fold = 25)
```

The parameters x[0] to x[15] are the trainable weights of the classifier.

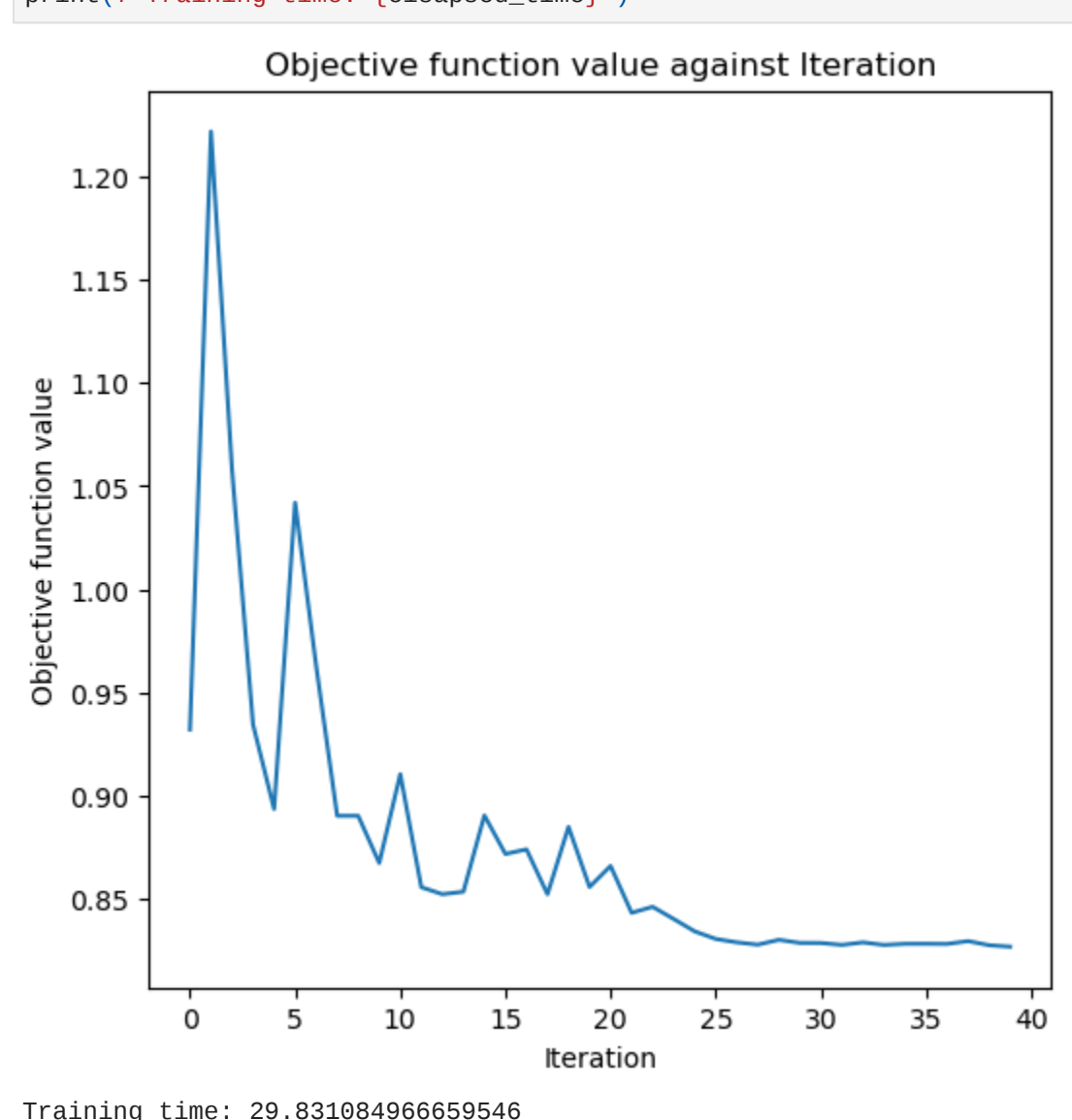
In [20]:
```python
from qiskit_algorithms.optimizers import COBYLA

optimizer = COBYLA(maxiter=100)
```

Using a simulator to train:

In [21]:
```python
from qiskit.primitives import Sampler

sampler = Sampler()
```

In [22]:
```python
from matplotlib import pyplot as plt
from IPython.display import clear_output

objective_func_vals = []

# objective function characterizes the distance between the predictions and known labeled data.

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

def callback_graph(weights, obj_func_eval):

# Append the value of the objective function to an array so we
# can plot the iteration verses the objective function value

            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()
```

In [28]:
```python
import time
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 = []

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

print(f'Training time: {elsapsed_time}')
```




```
Training time: 29.831084966659546
```

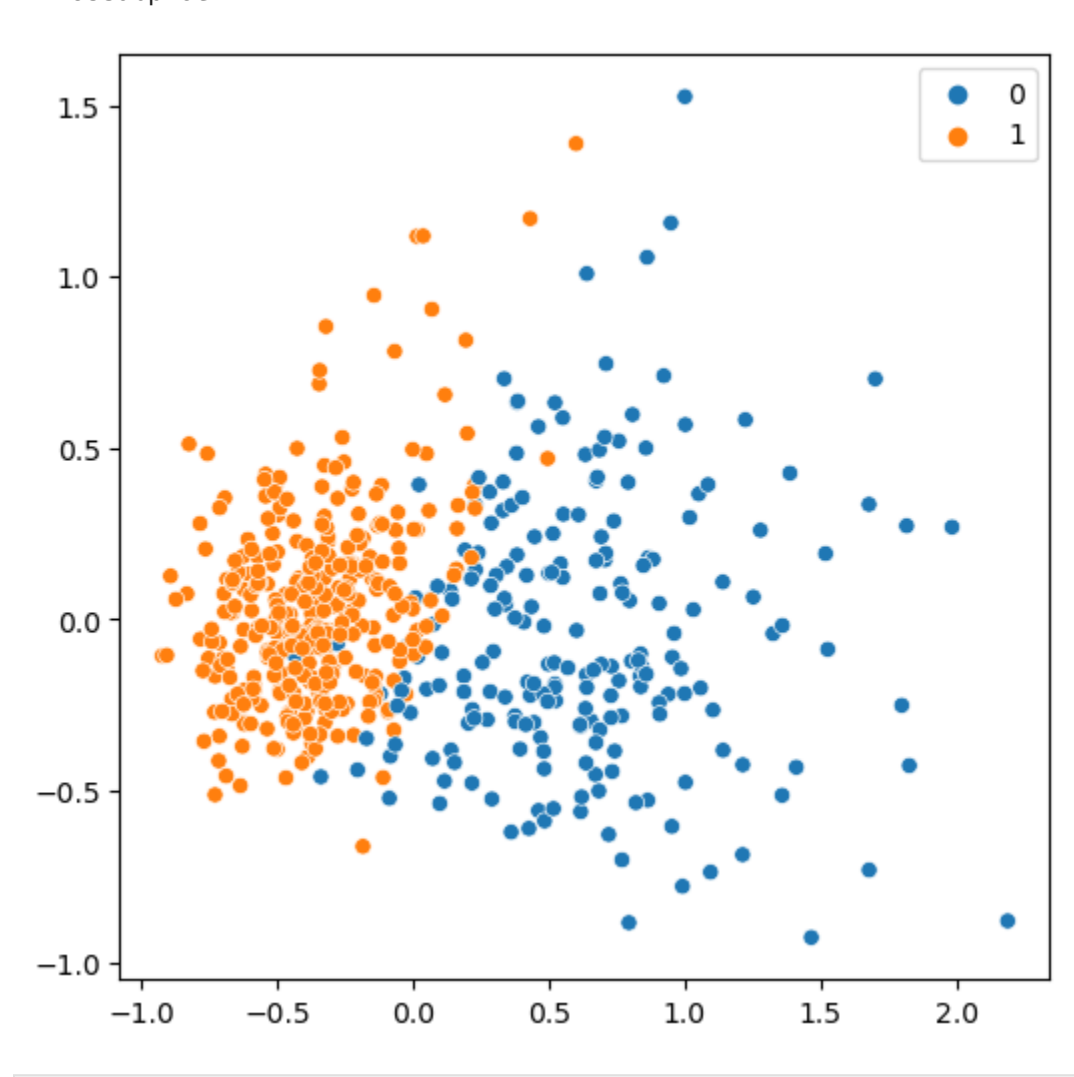
In [29]:
```python
train_score_q4 = vqc.score(train_features, train_labels)
test_score_q4 = vqc.score(test_features, test_labels)

print(f"Quantum VQC on the training dataset: {train_score_q4:.2f}")
print(f"Quantum VQC on the test dataset:     {test_score_q4:.2f}")
```

```
Quantum VQC on the training dataset: 0.69
Quantum VQC on the test dataset:     0.61
```

Feature Reduction

In [30]:
```python
from sklearn.decomposition import PCA

features = PCA(n_components=2).fit_transform(features)

plt.rcParams["figure.figsize"] = (6, 6)
sns.scatterplot(x=features[:, 0], y=features[:, 1], hue=labels, palette="tab10")
```

Out[30]:
```
<AxesSubplot:>
```



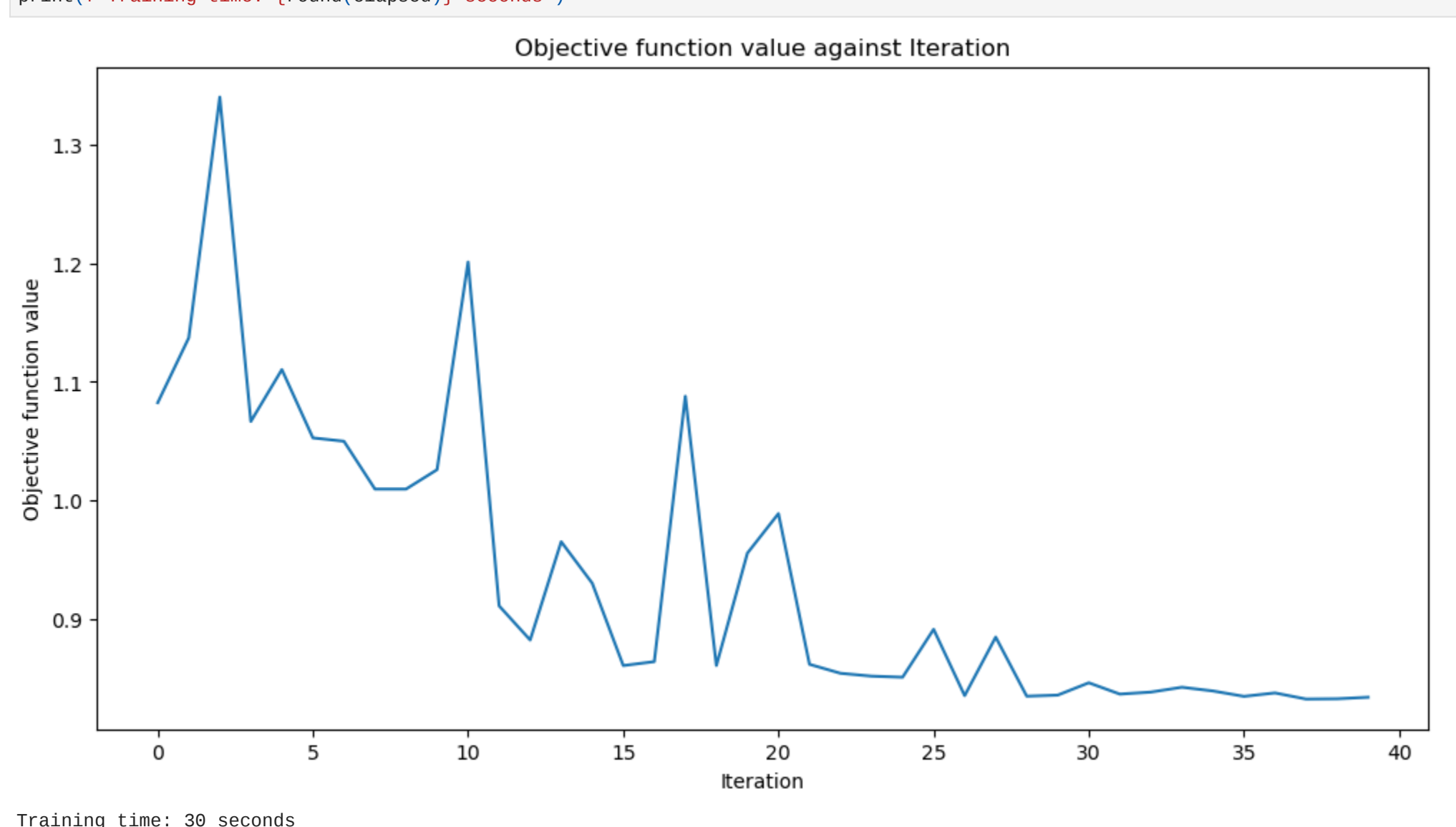
In [31]:
```python
train_features, test_features, train_labels, test_labels = train_test_split(
    features, labels, train_size=0.8, random_state=algorithm_globals.random_seed
)

svc.fit(train_features, train_labels)

train_score_c2 = svc.score(train_features, train_labels)
test_score_c2 = svc.score(test_features, test_labels)

print(f"Classical SVC on the training dataset: {train_score_c2:.2f}")
print(f"Classical SVC on the test dataset:     {test_score_c2:.2f}")
```

```
Classical SVC on the training dataset: 0.95
Classical SVC on the test dataset:     0.95
```

In [32]:
```python
num_features = features.shape[1]

feature_map = ZZFeatureMap(feature_dimension=num_features, reps=1)
ansatz = RealAmplitudes(num_qubits=num_features, reps=3)
```

In [33]:
```python
optimizer = COBYLA(maxiter=40) # reduction of iterations due to fewer qubits
```

In [34]:
```python
vqc = VQC(
    sampler=sampler,
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=optimizer,
    callback=callback_graph,
)

# clear objective value history
objective_func_vals = []

# make the objective function plot look nicer.
plt.rcParams["figure.figsize"] = (12, 6)


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

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



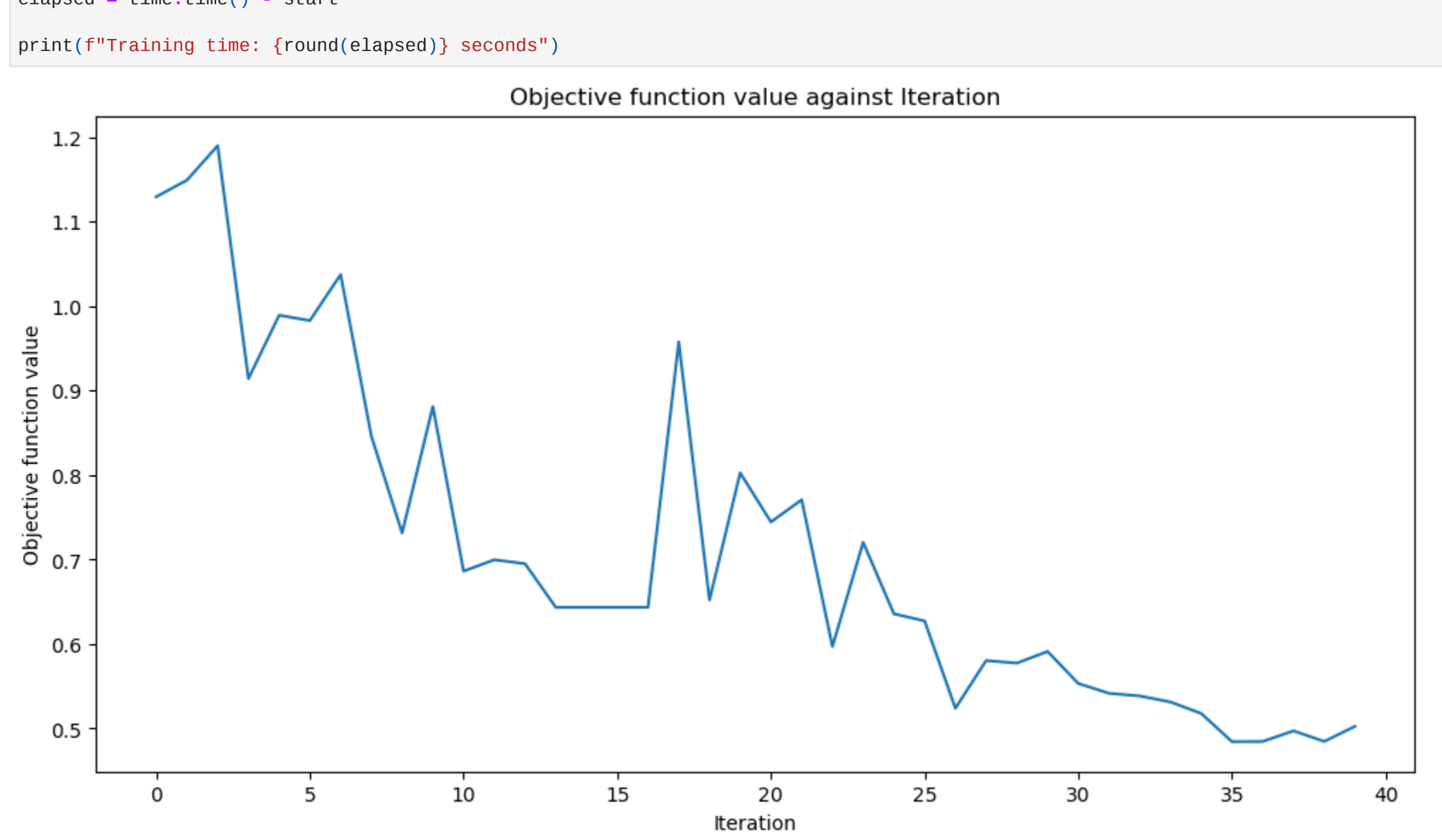

```
Training time: 30 seconds
```

In [35]:
```python
train_score_q2_ra = vqc.score(train_features, train_labels)
test_score_q2_ra = vqc.score(test_features, test_labels)

print(f"Quantum VQC on the training dataset using RealAmplitudes: {train_score_q2_ra:.2f}")
print(f"Quantum VQC on the test dataset using RealAmplitudes:     {test_score_q2_ra:.2f}")
```

```
Quantum VQC on the training dataset using RealAmplitudes: 0.68
Quantum VQC on the test dataset using RealAmplitudes:     0.63
```

Note the objective function is almost flattening, meaning increasing the number of iterations won't be able to increase the score. We will need to try another ansatz.

In [36]:
```python
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(num_qubits=num_features, reps=3)
optimizer = COBYLA(maxiter=40)

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

# clear objective value history
objective_func_vals = []

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

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



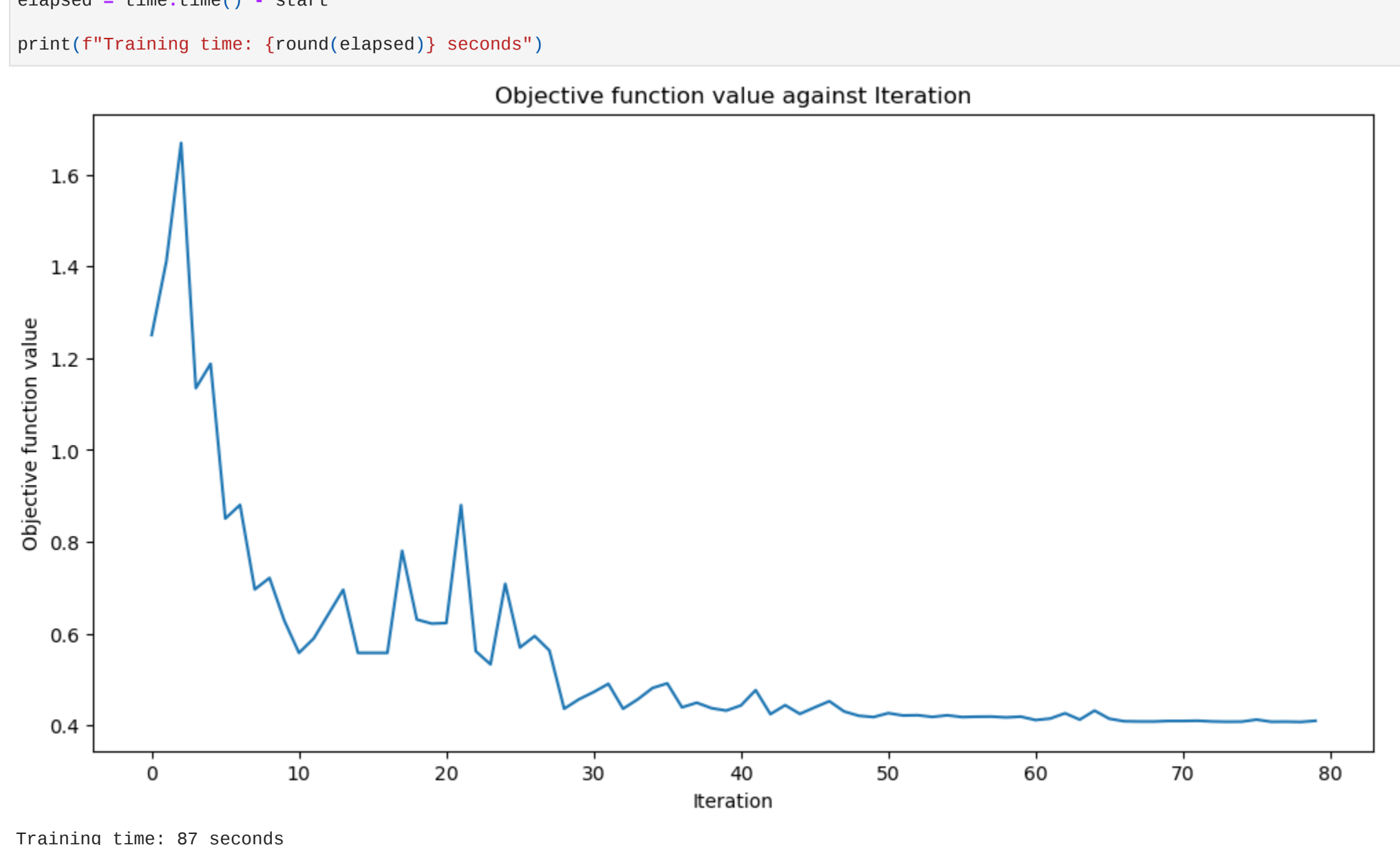

```
Training time: 41 seconds
```

In [37]:
```python
train_score_q2_eff = vqc.score(train_features, train_labels)
test_score_q2_eff = vqc.score(test_features, test_labels)

print(f"Quantum VQC on the training dataset using EfficientSU2: {train_score_q2_eff:.2f}")
print(f"Quantum VQC on the test dataset using EfficientSU2:     {test_score_q2_eff:.2f}")
```

```
Quantum VQC on the training dataset using EfficientSU2: 0.87
Quantum VQC on the test dataset using EfficientSU2:     0.83
```

Better than previous. lets try increase the number of iterations.

In [38]:
```python
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(num_qubits=num_features, reps=3)
optimizer = COBYLA(maxiter=80)

vqc2 = VQC(
    sampler=sampler,
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=optimizer,
    callback=callback_graph,
)

# clear objective value history
objective_func_vals = []

start = time.time()
vqc2.fit(train_features, train_labels)
elapsed = time.time() - start

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




```
Training time: 87 seconds
```

In [39]:
```python
train_score_q3_eff = vqc2.score(train_features, train_labels)
test_score_q3_eff = vqc2.score(test_features, test_labels)

print(f"Quantum VQC on the training dataset using EfficientSU2: {train_score_q3_eff:.2f}")
print(f"Quantum VQC on the test dataset using EfficientSU2:     {test_score_q3_eff:.2f}")
```

```
Quantum VQC on the training dataset using EfficientSU2: 0.90
Quantum VQC on the test dataset using EfficientSU2:     0.86
```

In [40]:
```python
print(f"Model                          | Test Score | Train Score")
print(f"SVC, 4 features                | {train_score_c4:10.2f} | {test_score_c4:10.2f}")
print(f"VQC, 4 features, RealAmplitudes | {train_score_q4:10.2f} | {test_score_q4:10.2f}")
print(f"----------------------------------------------------------")
print(f"SVC, 2 features                | {train_score_c2:10.2f} | {test_score_c2:10.2f}")
print(f"VQC, 2 features, RealAmplitudes | {train_score_q2_ra:10.2f} | {test_score_q2_ra:10.2f}")
print(f"VQC, 2 features, EfficientSU2   | {train_score_q2_eff:10.2f} | {test_score_q2_eff:10.2f}")
print(f"----------------------------------------------------------")
print(f"Quantum VQC on the training dataset using EfficientSU2: {train_score_q3_eff:.2f}")
print(f"Quantum VQC on the test dataset using EfficientSU2:     {test_score_q3_eff:.2f}")
```

```
Model                          | Test Score | Train Score
SVC, 4 features                |       0.98 |       0.98
VQC, 4 features, RealAmplitudes |       0.69 |       0.61
----------------------------------------------------------
SVC, 2 features                |       0.95 |       0.95
VQC, 2 features, RealAmplitudes |       0.68 |       0.63
VQC, 2 features, EfficientSU2   |       0.87 |       0.83
```

Resources:

- IBM
- QML Summer School 2021
- datacamp
- kaggle