In [ ]:
```
# ==========================  Concise Model Explanation with Key Details ==================================
# ========================================================================================================
# 1. Libraries Used and Purposes:
        # 1.1. Numpy and Pandas: Data handling and numerical operations.
        # 1.2. Scikit-learn: Data splitting, feature scaling (StandardScaler), class weights, and evaluation me
        # 1.3. Scipy: Loading matrix files for gene expression data.
        # 1.4. TensorFlow/Keras: Neural network modeling (Sequential API, Dense layers, Dropout, Adam optimizer
        # 1.5. Matplotlib and Seaborn: Visualization of model performance, including training history and confu
        # 1.6. SHAP: Interpreting feature contributions and creating summary plots.

# 2. Data Preparation:
        # - Standardization: Scaled the gene expression data to ensure consistent ranges across features.
        # - Class Weights: Computed class weights to handle the class imbalance problem in PR, SD, and PD predi

# 3. Model Architecture:
        # - Neural Network Structure:
        # - Three fully connected hidden layers with ReLU activation to capture complex relationships between f
        # - Dropout regularization (50%) applied after each hidden layer to prevent overfitting.
        # - Softmax output layer for multi-class classification across PR, SD, and PD.

# 4. Model Training:
        # - Class Weights Application: Applied class weights during training to ensure balanced learning.
        # - Optimizer and Loss Function: Adam optimizer with categorical crossentropy loss for efficient learni

# 5. Evaluation Metrics:
        # - ROC-AUC Score: Evaluated the model's performance using the multi-class ROC-AUC metric.
        # - Confusion Matrix and Classification Report: Provided comprehensive performance insights for each re

# 6. Model Interpretation with SHAP:
        # - SHAP DeepExplainer: Explained feature contributions using a random subset of training data as backg
        # - SHAP Summary Plots:
        # - Beeswarm plots showed individual sample impacts.
        # - Bar plots identified overall feature importance.
```

In [ ]:
```
# ================================================ 1. LIBRARY IMPORTS ========================================
# ========================================================================================================
# Core libraries for data handling and numerical operations
import numpy as np
import pandas as pd

# Machine learning preprocessing
from sklearn.model_selection import train_test_split  # For splitting the data
# For standardizing the features (MinMaxScaler, RobustScaler: KAVEH)
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

# Library for reading matrix files
from scipy.io import mmread  # To read .mtx files commonly used in genomics

# Neural network libraries from Keras
from tensorflow.keras.models import Sequential  # For creating a linear stack of layers in the neural network
from tensorflow.keras.layers import Dense, Dropout  # 'Dense' for fully connected layers, 'Dropout' for regular
from tensorflow.keras.optimizers import Adam  # Optimizer for compiling the neural network

# Additional libraries for reproducibility across multiple runs
import tensorflow as tf
import random
import os

# ============================================ MODEL EVALUATION IMPORTS ======================================
# Import libraries necessary for model evaluation after training
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score  # Evaluation metrics
import seaborn as sns  # For high-level, attractive statistical visualizations
import matplotlib.pyplot as plt  # For creating static, interactive, and animated visualizations in Python
```

In [3]:
```
# ================================================ 2. SET RANDOM SEED ========================================
# ========================================================================================================
# Set a seed value
seed_value = 42  # we can choose any number

# 1. Set `PYTHONHASHSEED` environment variable at a fixed value
os.environ['PYTHONHASHSEED'] = str(seed_value)

# 2. Set `python` built-in pseudo-random generator at a fixed value
random.seed(seed_value)

# 3. Set `numpy` pseudo-random generator at a fixed value
np.random.seed(seed_value)

# 4. Set the `tensorflow` pseudo-random generator at a fixed value
tf.random.set_seed(seed_value)

# ============================================ why need to SET RANDOM SEED ===================================
# Setting a random seed in our code helps to ensure that the results are reproducible by making
# the random number generation predictable.
# In this pipeline, working with Keras and other libraries like NumPy and potentially TensorFlow,
```

```
# we should set seeds for all these libraries to ensure consistency.
```

In [4]:
```
# ================================================== 3. Loading data  ======================================
# ==========================================================================================================

# Load gene names
gene_names_path = '/mnt/data/10.DL_scRNAseq_OMID_SURAJ/3.Study_Files/Subset_Seurat_Object/\
ICB_Only_NoICB_ICBNE_Excluded/Matrix_Files/ICB_Only_NoICB_ICBNE_Excluded_genes.tsv'
gene_names = pd.read_csv(gene_names_path, header=None, sep='\t')
gene_names_list = gene_names[0].tolist()  # Convert gene names to a list

# Load the expression matrix
expression_matrix_path = '/mnt/data/10.DL_scRNAseq_OMID_SURAJ/3.Study_Files/\
Subset_Seurat_Object/ICB_Only_NoICB_ICBNE_Excluded/Matrix_Files/ICB_Only_NoICB_ICBNE_Excluded_expression_matrix
expression_matrix = mmread(expression_matrix_path).toarray()

# Transpose the expression matrix so that genes become columns and cells become rows
X = pd.DataFrame(expression_matrix.T, columns=gene_names_list)

# Load your metadata
metadata_path = '/mnt/data/10.DL_scRNAseq_OMID_SURAJ/3.Study_Files/Subset_Seurat_Object/\
ICB_Only_NoICB_ICBNE_Excluded/ICB_Only_NoICB_ICBNE_Excluded_Cells_Metadata.csv'
metadata = pd.read_csv(metadata_path)

# One-hot encode the labels
y = pd.get_dummies(metadata['ICB_Response'])

# Ensure the gene names match the number of columns in the transposed expression matrix
if X.shape[1] != len(gene_names_list):
    raise ValueError("The number of gene names does not match the number of features in the transposed expressi


# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Print the number of genes and features to confirm correct loading and transposition
print(f"Number of genes in gene file: {len(gene_names_list)}")
print(f"Number of features (genes) in the matrix: {X.shape[1]}")



# Now your data is ready for model training
```

```
Number of genes in gene file: 29898
Number of features (genes) in the matrix: 29898
```

In [6]:
```
# ======================  4. Class Weights Calculation for my imbalance data (PR, SD, PD) ==================
# ==========================================================================================================
from sklearn.utils.class_weight import compute_class_weight

# Calculate class weights
labels = metadata['ICB_Response'].values
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(labels), y=labels)
class_weights_dict = {i: weight for i, weight in enumerate(class_weights)}
```

In [7]:
```
# ================================================== 4. BUILD MODEL ========================================
# ==========================================================================================================
# Define the model architecture


model = Sequential()

# Input layer: Implicitly defined by the input_dim parameter of the first Dense layer.

# First Dense layer with more units to capture more complex relationships
model.add(Dense(64, activation='relu', input_dim=X_train_scaled.shape[1]))

# First Dropout layer remains the same
model.add(Dropout(0.5))

# Second Dense layer with 64 units, the same as the first to maintain capacity
model.add(Dense(64, activation='relu'))

# Second Dropout layer remains the same
model.add(Dropout(0.5))

# Adding a third Dense layer to add depth to the model
model.add(Dense(32, activation='relu'))

# Third Dropout layer remains the same
model.add(Dropout(0.5))

# Output layer with 3 units for the three classes (PR/SD/PD)
```

```
model.add(Dense(3, activation='softmax'))

# Compile the model with the Adam optimizer and categorical crossentropy loss
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
```

```
2024-05-05 09:57:43.318280: E external/local_xla/xla/stream_executor/cuda/cuda_driver.cc:274] failed call to cu
Init: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
```

In [8]:
```
# ================================================ 5. TRAIN MODEL ============================================
# ============================================================================================================
# Train the model
# "class_weight=class_weights_dict" was added here and this is the difference of this version compared
history = model.fit(X_train_scaled, y_train, validation_split=0.1, epochs=50, batch_size=8, class_weight=class_
```

```
Epoch 1/50
1099/1099 [==============================] - 8s 7ms/step - loss: 3.9951 - accuracy: 0.3843 - val_loss: 0.9862 -
val_accuracy: 0.3756
Epoch 2/50
1099/1099 [==============================] - 7s 7ms/step - loss: 1.6091 - accuracy: 0.4101 - val_loss: 0.8674 -
val_accuracy: 0.6090
Epoch 3/50
1099/1099 [==============================] - 8s 7ms/step - loss: 1.2117 - accuracy: 0.4827 - val_loss: 0.7175 -
val_accuracy: 0.8076
Epoch 4/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.8958 - accuracy: 0.7108 - val_loss: 0.5931 -
val_accuracy: 0.8383
Epoch 5/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.7649 - accuracy: 0.7726 - val_loss: 0.4901 -
val_accuracy: 0.9089
Epoch 6/50
1099/1099 [==============================] - 8s 8ms/step - loss: 0.6425 - accuracy: 0.8215 - val_loss: 0.4067 -
val_accuracy: 0.8956
Epoch 7/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.5667 - accuracy: 0.8587 - val_loss: 0.3155 -
val_accuracy: 0.9263
Epoch 8/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.6074 - accuracy: 0.8787 - val_loss: 0.4215 -
val_accuracy: 0.9222
Epoch 9/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.4083 - accuracy: 0.9009 - val_loss: 0.2987 -
val_accuracy: 0.9324
Epoch 10/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.2751 - accuracy: 0.9255 - val_loss: 0.3163 -
val_accuracy: 0.9099
Epoch 11/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.2701 - accuracy: 0.9372 - val_loss: 0.2327 -
val_accuracy: 0.9406
Epoch 12/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.3124 - accuracy: 0.9438 - val_loss: 0.4011 -
val_accuracy: 0.9110
Epoch 13/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.1862 - accuracy: 0.9589 - val_loss: 0.3557 -
val_accuracy: 0.9110
Epoch 14/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.2679 - accuracy: 0.9451 - val_loss: 0.2660 -
val_accuracy: 0.9253
Epoch 15/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.1441 - accuracy: 0.9679 - val_loss: 0.2886 -
val_accuracy: 0.9345
Epoch 16/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.2024 - accuracy: 0.9545 - val_loss: 0.3546 -
val_accuracy: 0.9191
Epoch 17/50
1099/1099 [==============================] - 8s 8ms/step - loss: 0.1325 - accuracy: 0.9685 - val_loss: 0.3594 -
val_accuracy: 0.9304
Epoch 18/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1552 - accuracy: 0.9725 - val_loss: 0.2321 -
val_accuracy: 0.9447
Epoch 19/50
1099/1099 [==============================] - 8s 8ms/step - loss: 0.1678 - accuracy: 0.9746 - val_loss: 0.3420 -
val_accuracy: 0.9099
Epoch 20/50
1099/1099 [==============================] - 8s 7ms/step - loss: 0.1430 - accuracy: 0.9755 - val_loss: 0.2968 -
val_accuracy: 0.9406
Epoch 21/50
1099/1099 [==============================] - 8s 8ms/step - loss: 0.2152 - accuracy: 0.9741 - val_loss: 0.2510 -
val_accuracy: 0.9376
Epoch 22/50
1099/1099 [==============================] - 8s 8ms/step - loss: 0.1362 - accuracy: 0.9744 - val_loss: 0.3637 -
val_accuracy: 0.9253
Epoch 23/50
1099/1099 [==============================] - 8s 8ms/step - loss: 0.2581 - accuracy: 0.9730 - val_loss: 0.5120 -
val_accuracy: 0.9048
Epoch 24/50
1099/1099 [==============================] - 8s 8ms/step - loss: 0.1151 - accuracy: 0.9811 - val_loss: 0.2823 -
val_accuracy: 0.9447
```

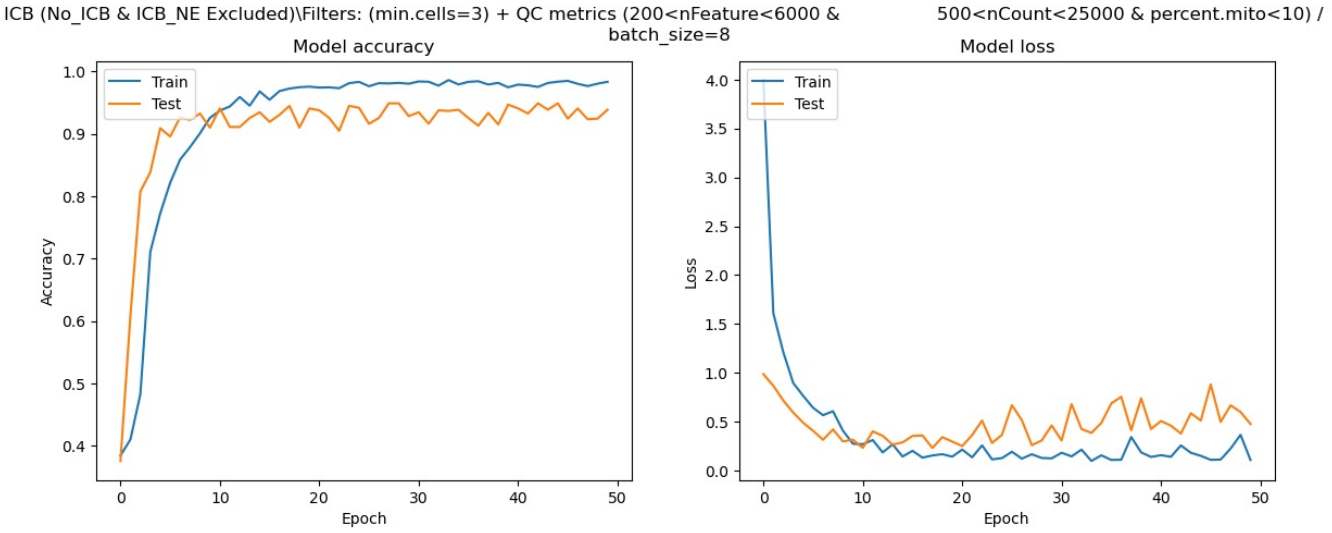
```
Epoch 25/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1279 - accuracy: 0.9832 - val_loss: 0.3655 -
val_accuracy: 0.9417
Epoch 26/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1928 - accuracy: 0.9762 - val_loss: 0.6692 -
val_accuracy: 0.9161
Epoch 27/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1223 - accuracy: 0.9811 - val_loss: 0.5156 -
val_accuracy: 0.9253
Epoch 28/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1674 - accuracy: 0.9807 - val_loss: 0.2600 -
val_accuracy: 0.9488
Epoch 29/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1291 - accuracy: 0.9816 - val_loss: 0.3084 -
val_accuracy: 0.9488
Epoch 30/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1254 - accuracy: 0.9802 - val_loss: 0.4621 -
val_accuracy: 0.9284
Epoch 31/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1826 - accuracy: 0.9838 - val_loss: 0.3083 -
val_accuracy: 0.9345
Epoch 32/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1455 - accuracy: 0.9834 - val_loss: 0.6808 -
val_accuracy: 0.9161
Epoch 33/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.2146 - accuracy: 0.9771 - val_loss: 0.4257 -
val_accuracy: 0.9376
Epoch 34/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.0987 - accuracy: 0.9861 - val_loss: 0.3856 -
val_accuracy: 0.9365
Epoch 35/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1565 - accuracy: 0.9791 - val_loss: 0.4886 -
val_accuracy: 0.9386
Epoch 36/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1091 - accuracy: 0.9833 - val_loss: 0.6874 -
val_accuracy: 0.9253
Epoch 37/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1115 - accuracy: 0.9842 - val_loss: 0.7551 -
val_accuracy: 0.9130
Epoch 38/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.3438 - accuracy: 0.9790 - val_loss: 0.4127 -
val_accuracy: 0.9335
Epoch 39/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1867 - accuracy: 0.9816 - val_loss: 0.7393 -
val_accuracy: 0.9150
Epoch 40/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1398 - accuracy: 0.9745 - val_loss: 0.4258 -
val_accuracy: 0.9468
Epoch 41/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1578 - accuracy: 0.9788 - val_loss: 0.5069 -
val_accuracy: 0.9406
Epoch 42/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1416 - accuracy: 0.9777 - val_loss: 0.4596 -
val_accuracy: 0.9324
Epoch 43/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.2579 - accuracy: 0.9751 - val_loss: 0.3786 -
val_accuracy: 0.9488
Epoch 44/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1824 - accuracy: 0.9815 - val_loss: 0.5863 -
val_accuracy: 0.9386
Epoch 45/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1513 - accuracy: 0.9835 - val_loss: 0.5127 -
val_accuracy: 0.9488
Epoch 46/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1101 - accuracy: 0.9848 - val_loss: 0.8816 -
val_accuracy: 0.9243
Epoch 47/50
1099/1099 [==============================] - 9s 8ms/step - loss: 0.1133 - accuracy: 0.9801 - val_loss: 0.4980 -
val_accuracy: 0.9406
Epoch 48/50
1099/1099 [==============================] - 9s 9ms/step - loss: 0.2251 - accuracy: 0.9765 - val_loss: 0.6674 -
val_accuracy: 0.9232
Epoch 49/50
1099/1099 [==============================] - 9s 9ms/step - loss: 0.3660 - accuracy: 0.9803 - val_loss: 0.5997 -
val_accuracy: 0.9243
Epoch 50/50
1099/1099 [==============================] - 9s 9ms/step - loss: 0.1088 - accuracy: 0.9833 - val_loss: 0.4771 -
val_accuracy: 0.9386
```

In [9]:

```python
# ================================================ 6. EVALUATE MODEL ======================================
# =========================================================================================================
# Evaluate on test set
evaluation = model.evaluate(X_test_scaled, y_test)
print(f'Test Loss: {evaluation[0]}, Test Accuracy: {evaluation[1]}')

# Plot training history
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 5))
```

```
# Add an overall title for the figure
plt.suptitle('ICB (No_ICB & ICB_NE Excluded)\Filters: (min.cells=3) + QC metrics (200<nFeature<6000 & \
             500<nCount<25000 & percent.mito<10) / \n batch_size=8'  )
# # (200<nFeature<6000 & 500<nCount<25000 & percent.mito<10)

# Plot training & validation accuracy values
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')



plt.show()
```

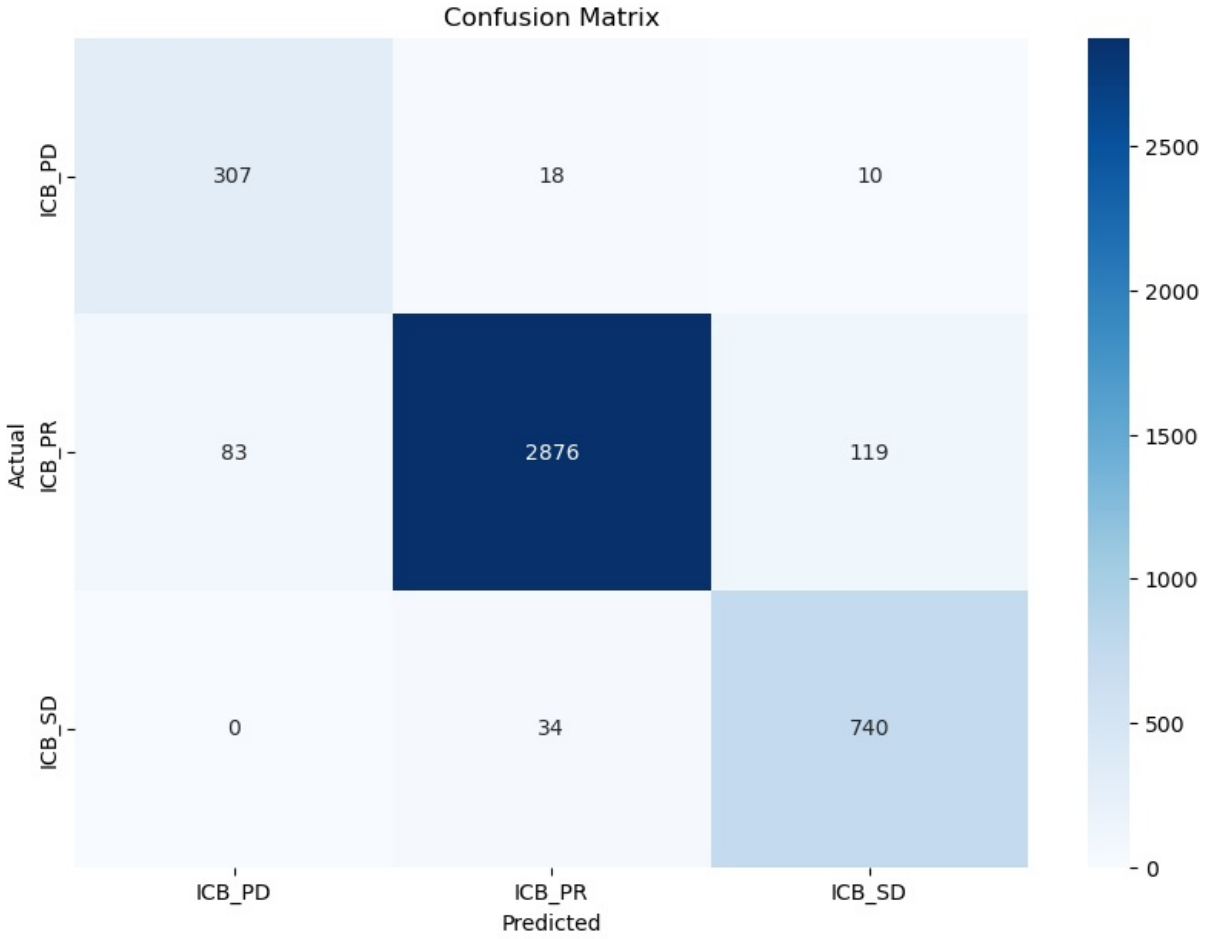
```
131/131 [==============================] - 0s 2ms/step - loss: 0.4941 - accuracy: 0.9369
Test Loss: 0.49412617087364197, Test Accuracy: 0.9369477033615112
```



In [11]:

```
# ======================================== 7. DETAILED EVALUATION OF THE MODEL ============================
# =========================================================================================================
# Import necessary libraries for evaluation
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
import seaborn as sns
import matplotlib.pyplot as plt


# Obtain the model's predictions on the test set
predictions = model.predict(X_test_scaled)
predicted_classes = np.argmax(predictions, axis=1)  # Convert probabilities to class labels
actual_classes = np.argmax(y_test.to_numpy(), axis=1)  # Convert one-hot encoded labels back to class labels

# ROC-AUC Score
# Calculate the ROC-AUC score which provides an aggregate measure of performance across all possible classifica
roc_auc = roc_auc_score(y_test, predictions, multi_class='ovr')
print(f'ROC-AUC Score: {roc_auc}')

# Classification Report
# Generate a classification report which includes precision, recall, f1-score for each class
print(classification_report(actual_classes, predicted_classes, target_names=y_test.columns))

# Confusion Matrix
# Generate and plot the confusion matrix to visualize the performance of the classification algorithm
conf_matrix = confusion_matrix(actual_classes, predicted_classes)
plt.figure(figsize=(10, 7))  # You can adjust the size as needed
sns.heatmap(conf_matrix, annot=True, fmt='g', cmap='Blues', xticklabels=y_test.columns, yticklabels=y_test.colu
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```
131/131 [==============================] - 0s 2ms/step
ROC-AUC Score: 0.9874122170047323
              precision    recall  f1-score   support

      ICB_PD       0.79      0.92      0.85       335
      ICB_PR       0.98      0.93      0.96      3078
      ICB_SD       0.85      0.96      0.90       774

    accuracy                           0.94      4187
   macro avg       0.87      0.94      0.90      4187
weighted avg       0.94      0.94      0.94      4187
```



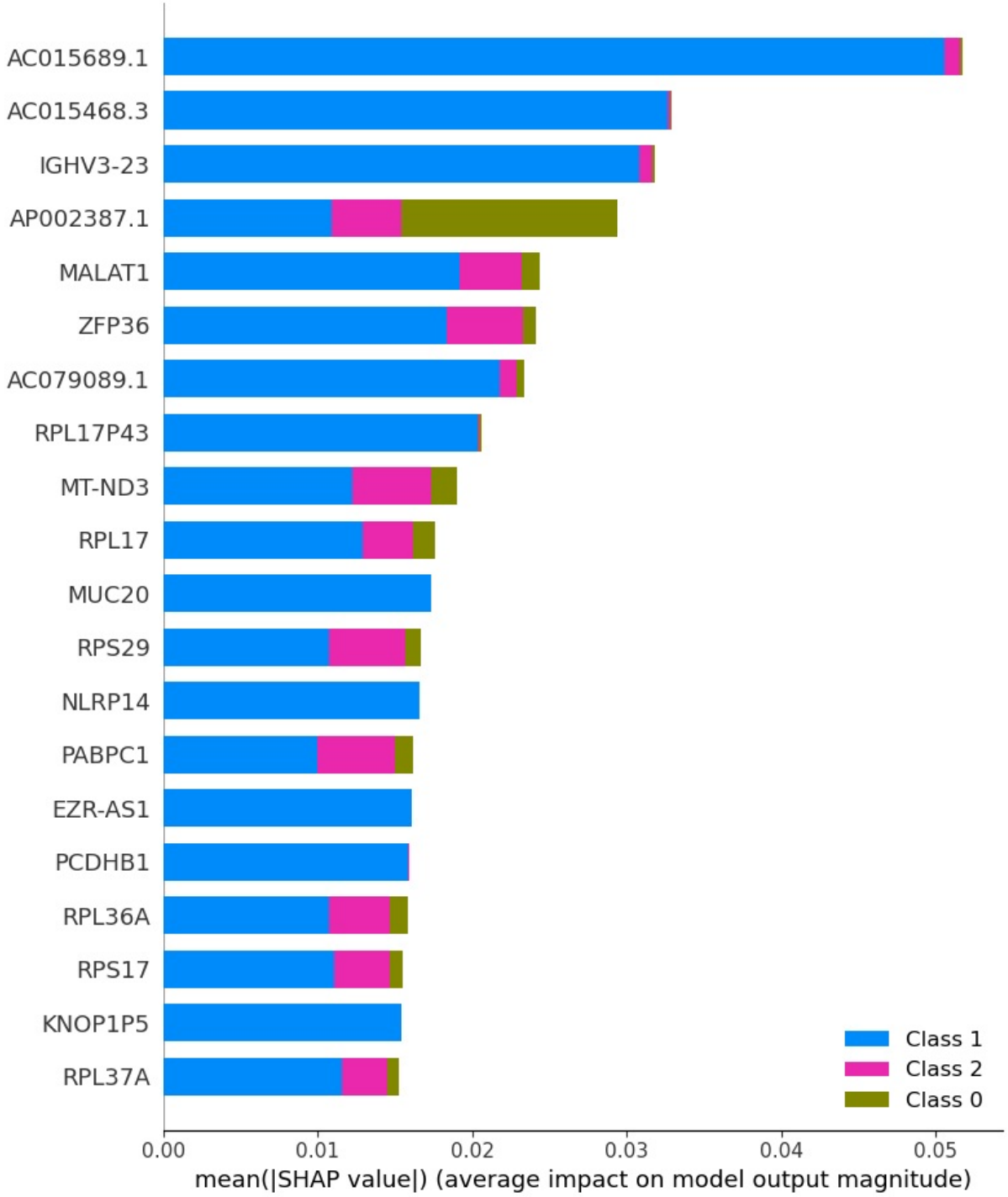
In [12]:
```
# To confirm the order of the classes, & inspect the column names of the one-hot encoded y variable
print(y.columns)
```

```
Index(['ICB_PD', 'ICB_PR', 'ICB_SD'], dtype='object')
```

In [19]:
```
# ========================================== 8. MODEL INTERPRETATION WITH SHAP ================================
# ============================================== explainer: (DeepExplainer) ===================================
# =================================================================================================================

# Import SHAP library for model interpretation
import shap

# Create a SHAP explainer
# The SHAP explainer will explain the output of the neural network
# Sample a larger random subset from the training set as the background for a complex dataset
background = shap.utils.sample(X_train_scaled, 100)  # * we can adjust the number based on the complixity our d

# Create the SHAP explainer with the selected background dataset
explainer = shap.DeepExplainer(model, background)


# Compute SHAP values
# These values represent the impact of each feature on the model's prediction
shap_values = explainer.shap_values(X_test_scaled)



# ================================================     Plot SHAP values   ========================================
# This plot will help understand how each feature influences the model's predictions

# Plot SHAP values for overall feature importance across all classes
# This plot will help understand the overall feature importance across all classes
shap.summary_plot(shap_values, X_test_scaled, plot_type='bar', feature_names=X.columns)


# =================================================================================================================
# * If our dataset is very large or our model is complex, using more background samples
```

```
# can be beneficial for a more accurate interpretation, but it will also demand more computational resources.
# - For a simple dataset: 100-300 samples
# - For complex datasets: 500-1000 or more
# - starting with a smaller number of background samples: 100-500 samples
```

```
Your TensorFlow version is newer than 2.4.0 and so graph support has been removed in eager mode and some static graphs may not be supported. See PR #1483 for discussion.
`tf.keras.backend.set_learning_phase` is deprecated and will be removed after 2020-10-11. To update it, simply pass a True/False value to the `training` argument of the `__call__` method of your layer or model.
```



In [17]:
```
# ========================================== Interpretation of the above plot ============================
# ========================================================================================================
    # - This plot highlights which features the model finds most predictive for each class.
    # - By examining individual feature impacts, researchers can understand which genes are most critical in
    # distinguishing among the response classes.
    # - The relatively smaller green bar segments across features suggest less differentiation for PD as compar

# Strong Biomarker Genes:
    # 1- PR:
        # - The large contributions of AC015689.1 and AC015468.3 to PR suggest they could be considered potenti
        # biomarkers indicative of a partial response.
        # - Genes like AC015689.1 and AC015468.3 stand out because their contributions to PR are highly signifi
        # they contribute less to SD or PD, making them more distinct as primary biomarkers for PR.

    # 2- SD:
        # - MALAT1 and other features (e.g., ZFP36, MT-ND3, RPS29, and PABPC1) do show substantial contribution
        # - If a feature (gene) is contributing positively to more than one class but at different levels,
        # it may not be the most reliable stand-alone biomarker for differentiating those classes.
        # - For instance, MALAT1 has more influence on PR than SD, making it less reliable for specifically ide
        # - MALAT1, ZFP36, and similar genes could serve as supporting features for SD predictions but might re
```

```
        # additional complementary biomarkers to achieve higher specificity.


# ================================= My suggested strategies for Reliable Biomarker Identification ============
# ============================================================================================================
# 1- Combining Features:
        # - Panel Approach:
        # - Use multiple features in combination to create a predictive panel. For example, while MALAT1 and ZF
        # might not differentiate well between PR and SD, combining their contributions with other distinct fea


# 2-  Grouping genes into higher-order biological features:
        # we can group genes into higher-order biological features like pathways, networks, or functional categ
        # this allows models to capture underlying biological processes and relationships better.

        # ============= Gene Grouping Based on:
            # 1- Pathway-Based Transformation:
                # we can aggregate and group genes into pathways (e.g. KEGG, Reactome, and GO)

            # 2- Gene Set Enrichment Analysis (GSEA):
                # we can identify enriched pathways or gene sets

            # 3- Protein-Protein Interaction (PPI) Networks:
                # we can use PPI networks to group genes based on their protein interactions and identify signi

            # 4- Gene Ontology (GO):
                # We can group features into functional categories using GO annotations.


            # ========== Advantages of this approach
            # 1- Reduction of Dimensionality

            # 2- Biological Relevance:
            # The model can focus on features that have known biological functions rather than isolated genes.

            # 3- Predictive Power:
            # This approach may highlight pathways or processes predictive of the clinical outcomes we are stud
```

In [ ]:

```
# ================================= 9. Extract shap values and their genes and save it in a csv file =========
# ============================================================================================================

import pandas as pd

# Convert SHAP values to a 1D array for the DataFrame
shap_values_1d = shap_array[non_zero_indices].flatten()

# Create a DataFrame with gene names and their corresponding SHAP values
shap_summary_df = pd.DataFrame({
    'Gene': non_zero_gene_names,
    'SHAP Value': shap_values_1d
})


# ====== # Save the DataFrame to a CSV file
# Define the desired file path
file_path = "~/Desktop/Python/DL_Sc_RNA_Seq/1.First_try/1.Basic_DNN_scRNAseq_PR_SD_PD/Shap_Value_Gene_names/\
shap_values_with_genes_backgroundSamples100.csv"

# Expand the user's home directory symbol '~' to the full path
full_file_path = os.path.expanduser(file_path)

# Ensure the directory exists, create if it doesn't
os.makedirs(os.path.dirname(full_file_path), exist_ok=True)

# Save the DataFrame to the CSV file at the specified path
shap_summary_df.to_csv(full_file_path, index=False)

print(f"SHAP values with gene names saved to '{full_file_path}'")
# ==========
```