Machine Learning Assignment 3

Name	ID	Task
Malak Alaa Mohamed	23012115	Data Preprocessing, SVM Implementation
Maram Mohamed Mahmoud	23012197	Neural Networks implementation and optimizing + classification report
Ganna Mohamed Abdul Fattah	23012138	Model comparison and analysis

Data Preprocessing:

Loading the dataset: We used the Iris dataset, which is available in scikit-learn. It contains 150 samples of Iris flowers, each with 4 features and a target variable that classifies the species into three classes

```
# Load Iris dataset
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['target'] = iris.target
```

Data exploration:

1. View Basic Info About the Dataset
2. Check for missing values (no missing values)

```
# Show basic info
print(df.info())
```

✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
None
```

```
print(df.head())
```

✓ 0.0s

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2   
1                4.9               3.0                1.4               0.2   
2                4.7               3.2                1.3               0.2   
3                4.6               3.1                1.5               0.2   
4                5.0               3.6                1.4               0.2   

   target
0       0
1       0
2       0
3       0
4       0
```

```
# Check for missing values
print(df.isnull().sum())
```

✓ 0.0s

```
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
target               0
dtype: int64
```

Splitting the data: We split the data into **70% training** and **30% testing**. We used **stratify=y** to make sure each class is equally represented in both sets

```
X = df.drop('target', axis=1) # Features
y = df['target']  # Target variable

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
```

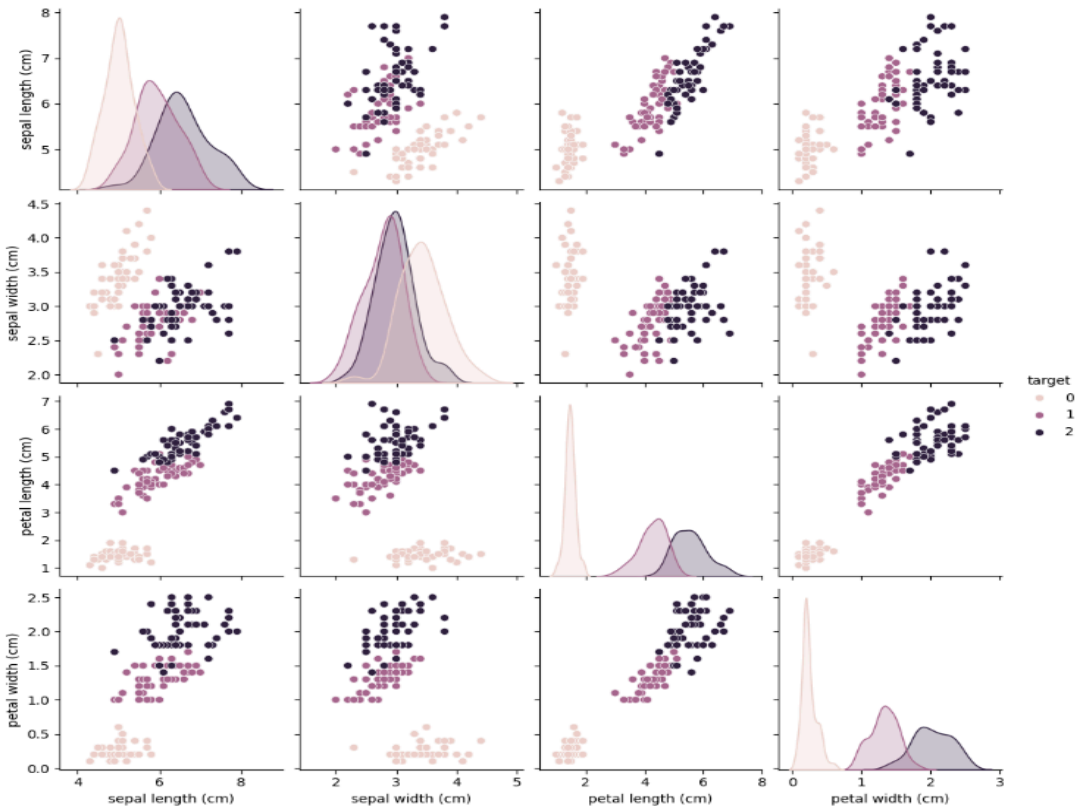
Normalization: We used **StandardScaler** to normalize the data.
why?

- Features like "sepal length" and "petal width" have different scales.
- Without normalization, the SVM might give more importance to larger values.
- **StandardScaler** makes all features have a **mean = 0** and **standard deviation = 1**, so they're on the same scale.

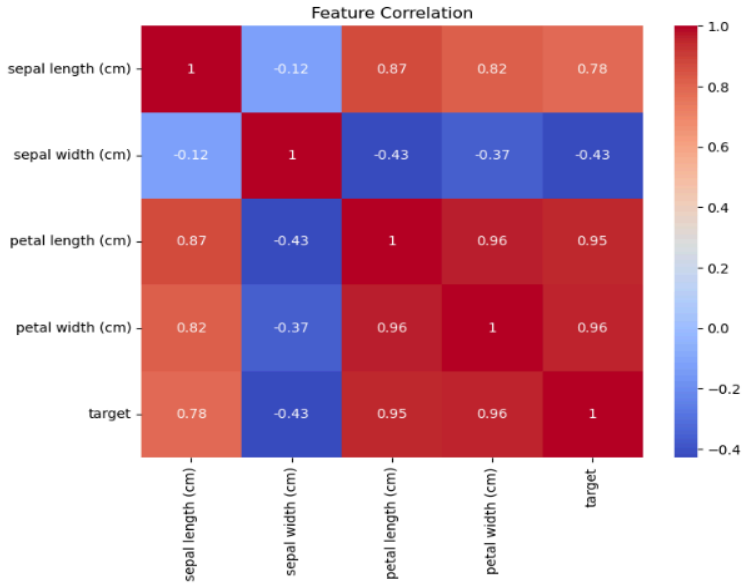
```
# normalizing the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Visualization:

1. **histograms:** to see how the data is distributed
   pair plot to compare features and how they differ between classes



2. **Correlation Heatmap**: To visualize how strongly features are related to each otherTo check for **correlated features**, which might affect the model



SVM Implementation:

How SVM Works:SVM is a supervised learning algorithm used for classification. It tries to find the best boundary (called a hyperplane) that separates the different classes in your data

Why SVM Is Good:

- Works well with **small to medium datasets**
- Handles **non-linear boundaries** using kernels
- Often **accurate and robust**

Step 1: Import SVM and Metrics

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

Step 2: Try Different Kernels

We tested 3 SVM kernels:

- **linear:** A straight decision boundary
- **poly:** Polynomial curves
- **rbf:** Gaussian (smooth curved) boundaries

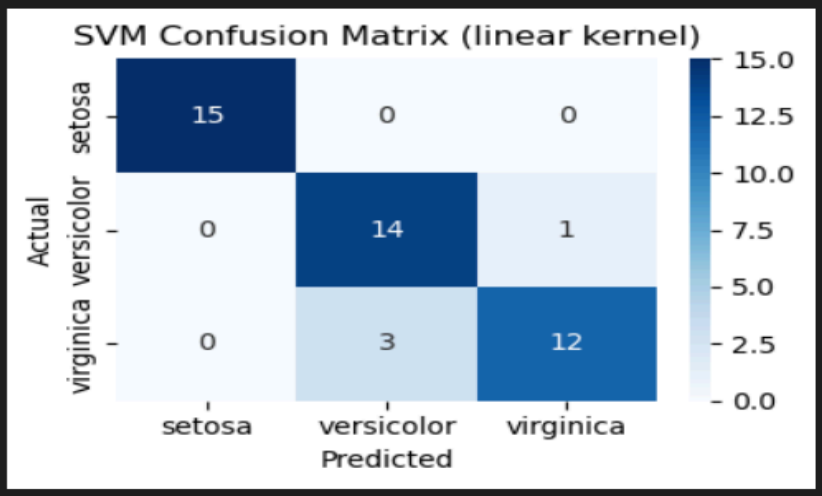
```
# Dictionary to hold results
results = {}

# Initialize SVM with different kernels
kernels = ['linear', 'poly', 'rbf']

for kernel in kernels:
    print(f"\n--- Kernel: {kernel} ---")

    # Initialize SVM with specified kernel
    svm = SVC(kernel=kernel, gamma='auto')

    # model trainig
    svm.fit(X_train, y_train)

    # Predict on test data
    y_pred = svm.predict(X_test)

    # Evaluatation
    acc = accuracy_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)
    print(f"Accuracy: {acc:.4f}")
    print("Confusion Matrix:")
    print(cm)
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
```

Results:

1. Linear Kernel

- Accuracy: 91.11%
- Performs well overall, especially on Setosa and Versicolor.
- Slight misclassification in Virginica

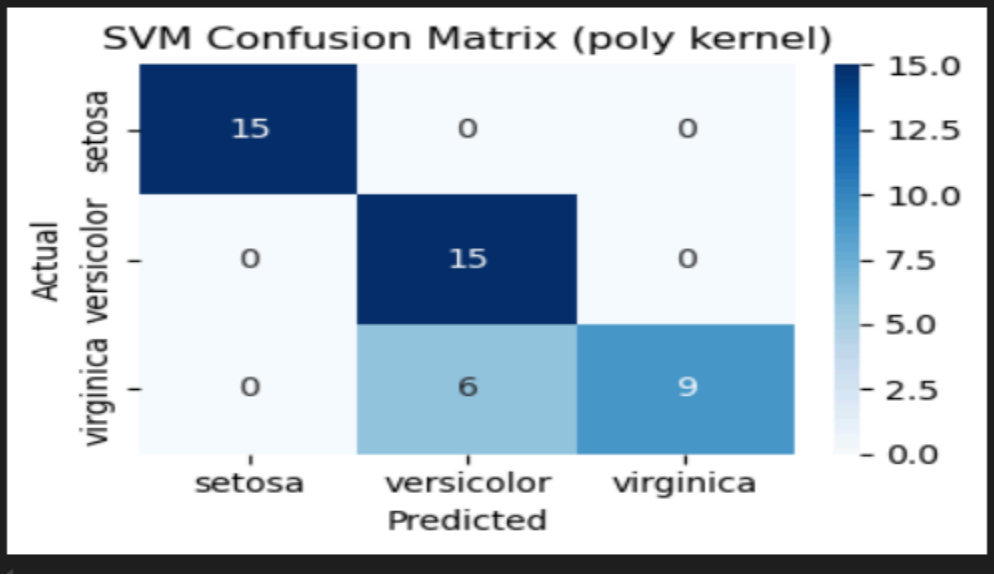
```
--- Kernel: linear ---
Accuracy: 0.9111
Confusion Matrix:
[[15  0  0]
 [ 0 14  1]
 [ 0  3 12]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       0.82      0.93      0.88        15
           2       0.92      0.80      0.86        15

    accuracy                           0.91        45
   macro avg       0.92      0.91      0.91        45
weighted avg       0.92      0.91      0.91        45
```



2. Polynomial Kernel

- Accuracy: 86.67%
- Performs best on Setosa, but struggles with Virginica (some overlap with Versicolor).
- Slightly lower performance than linear and RBF

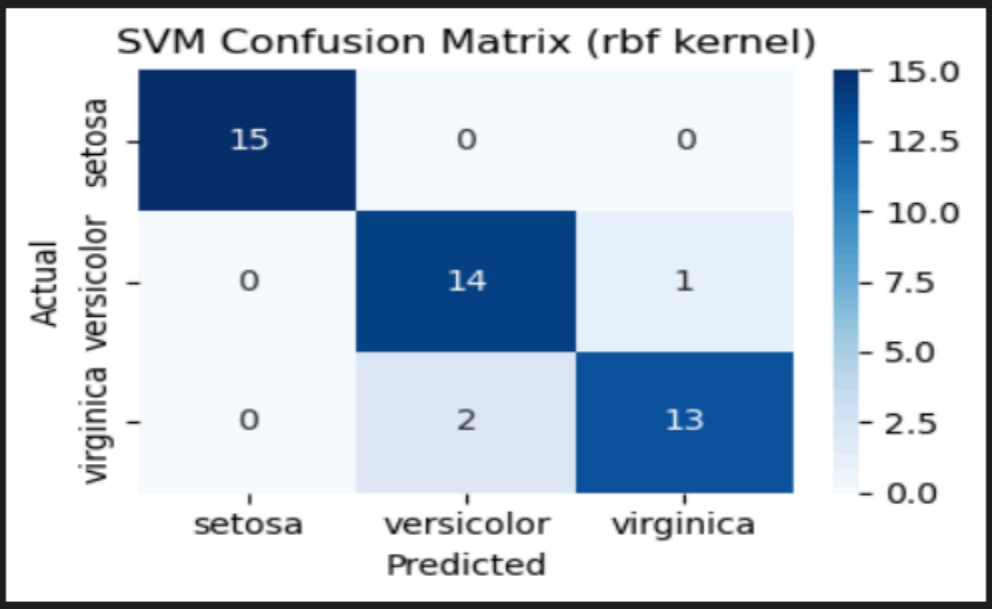
```
--- Kernel: poly ---
Accuracy: 0.8667
Confusion Matrix:
[[15  0  0]
 [ 0 15  0]
 [ 0  6  9]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       0.71      1.00      0.83        15
           2       1.00      0.60      0.75        15

    accuracy                           0.87        45
   macro avg       0.90      0.87      0.86        45
weighted avg       0.90      0.87      0.86        45
```



3. RBF Kernel

- Accuracy: 93.33% Best performance
- Good balance across all three classes.
- Misclassified only a few samples in Versicolor and Virginica.

```
--- Kernel: rbf ---
Accuracy: 0.9333
Confusion Matrix:
[[15  0  0]
 [ 0 14  1]
 [ 0  2 13]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        15
           1       0.88      0.93      0.90        15
           2       0.93      0.87      0.90        15

    accuracy                           0.93        45
   macro avg       0.93      0.93      0.93        45
weighted avg       0.93      0.93      0.93        45
```



Neural Networks implementation using Keras :

Firstly we imported the necessary libraries and to import keras we have to import Tensorflow first and to construct the model we need the following :
Layers , denses (edges also contain the units , neurons , input shape , activation functions and so on ..)

```
import tensorflow
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

1-Preparing data:

We needed only to convert the target Y , so we used label encoding to convert labels from strings to integers then used one hot encoding to be easy for the model to work with .

```
#encoding from strings to integers
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train)
y_test_encoded  = le.transform(y_test)
```

Then one hot encoding from keras **to_categorical**
Also we got the number of classes using len(set(y)) as the set contains only the unique values = three classes

```
#converting into one hot
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train_encoded, num_classes=num_classes)
y_test = to_categorical(y_test_encoded, num_classes=num_classes)
```

2-Define the Model:

From my point of view this was the most important and interesting part as i learned a lot while modifying the layers and the comments indicates our trials
The process is like the following 1- inputs get into the neuron 2- keras initialize the weights and do the sum in the neuron then applying the activation function we specified so the first layer means input layer + 1’st hidden layer

Then modifying the layers as if the model needs to be more complex or not , we played in the units = number of neurons for each layer also the number of layers , also we used activation function softmax as it produces probabilities for each class and takes the highest probability as the chosen class

```
model = Sequential([
   Dense (64, input_shape=[4] , activation='relu') ,
   Dense (32  , activation='relu') , # we tried first with simple number
of units=  24 , 12 and acc = 69%
   #then we wanted to improve the acc and reducing the loss so we made the
model more complex
   # so we added another hidden layer and units = 64 , 32 the acc = 86%
then we added more layers and units to improve the acc
   #but the acc reduced so our data needs less complexity
   #Dense (16  , activation='relu') ,
   Dense (3,  activation='softmax')
])

#we will try to use early stopping to prevent over fitting
# Add early stopping
from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)
```

Also we needed to use early stopping to address over fitting in the model as the model while training gave me a validation acc = 100%

3-Compile the Model:

To compile the model we used adam optimizer as it worked perfectly with us in optimizing and the loss was categorical_cross_entropy as we worked with a classification project also to evaluate the model while training phase we used metrics = [‘accuracy’]

```
model.compile(optimizer = 'adam' , loss='categorical_crossentropy' ,
metrics = ['accuracy'] )
```

4-Training the model and hyperparameter tuning:

In the hyperparameter tuning we played with the number of layers , epochs and the batch size and in the comments we indicated our trials.

There is also a small note : in the number of epochs we put it in a variable as we used it later to visualize the loss across number of epochs

```
epochs = 50#making it in a variable because i will use it latter in
visualizing
history = model.fit( X_train ,y_train  , validation_split=0.2, batch_size
= 12
                    , epochs= epochs , verbose = 1,
                    callbacks =[early_stop])
#we played also in the number of epochs and the batch size so we tried at
first the number of epochs = 10
#and the batch size  = 64 the acc = 85 % at max
#we reduced the batch size into 12 only and increased the number of epochs
to 50
# and the acc improved to 98% (maybe it overfits so we will tets it to
know and tune)
#after evaluating we found the trainig acc = 98% and the testing is 91 %
abit overfitting
# since we said previously we will use early stopping so we will increase
the number of epochs
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8353 - loss: 0.5372 - val_accuracy: 0.8571 - val_loss: 0.4
Epoch 8/50
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8379 - loss: 0.4524 - val_accuracy: 0.9048 - val_loss: 0.4
Epoch 9/50
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8365 - loss: 0.4353 - val_accuracy: 0.9048 - val_loss: 0.3
Epoch 10/50
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9171 - loss: 0.3413 - val_accuracy: 0.9048 - val_loss: 0.3
Epoch 11/50
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8832 - loss: 0.3359 - val_accuracy: 0.9048 - val_loss: 0.3
Epoch 12/50
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.8798 - loss: 0.2962 - val_accuracy: 0.9048 - val_loss: 0.
Epoch 13/50
...
Epoch 49/50
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9774 - loss: 0.0686 - val_accuracy: 1.0000 - val_loss: 0.0
Epoch 50/50
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9695 - loss: 0.0684 - val_accuracy: 1.0000 - val_loss: 0.0
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

our Notes

when we used early stopping we added get best weights = True and that exactly what happened until we get the best weights before the model over fits

5-Evaluating the model and see whether if it overfits or not:

```
test_loss , test_acc= model.evaluate(X_test , y_test)
print(f'the test accuracy :{test_acc*100:.2f}%')
```

```
[114]  print(f'the test accuracy :{test_acc*100:.2f}%')

       2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9199 - loss: 0.1526
       the test accuracy :91.11%
```

The test Accuracy= 91% which is very good for our model

6-Making predictions to get the classification report and the confusion matrix:

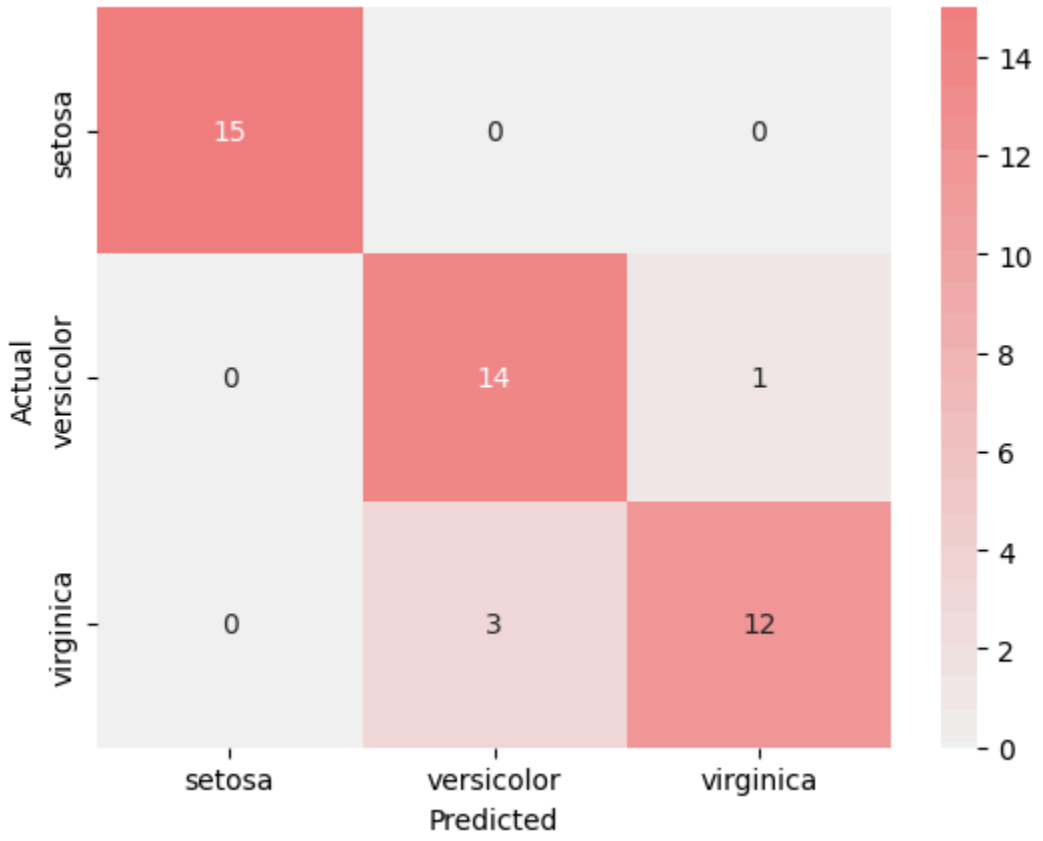
We reversed the one hot encoding back

```
from sklearn.metrics import classification_report, confusion_matrix

# Get true labels (reverse one-hot encoding)
y_true = np.argmax(y_test, axis=1)

# Get predicted labels for test set
y_pred = np.argmax(model.predict(X_test), axis=1)

print("\nClassification Report:")
print(classification_report(y_true, y_pred, target_names=le.classes_))

print("\nConfusion Matrix:")
print(confusion_matrix(y_true, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.82      0.93      0.88        15
   virginica       0.92      0.80      0.86        15

    accuracy                           0.91        45
   macro avg       0.92      0.91      0.91        45
weighted avg       0.92      0.91      0.91        45


Confusion Matrix:
[[15  0  0]
 [ 0 14  1]
 [ 0  3 12]]
```

Confusion Matrix

```
rose_palette = sns.light_palette("lightcoral", as_cmap=True)
cm = confusion_matrix(y_true , y_pred)
sns.heatmap(cm  ,annot=True , cmap=rose_palette , xticklabels=le.classes_,
yticklabels=le.classes_)
plt.ylabel('Actual') #actual
plt.xlabel('Predicted')#predicted
plt.show()
```



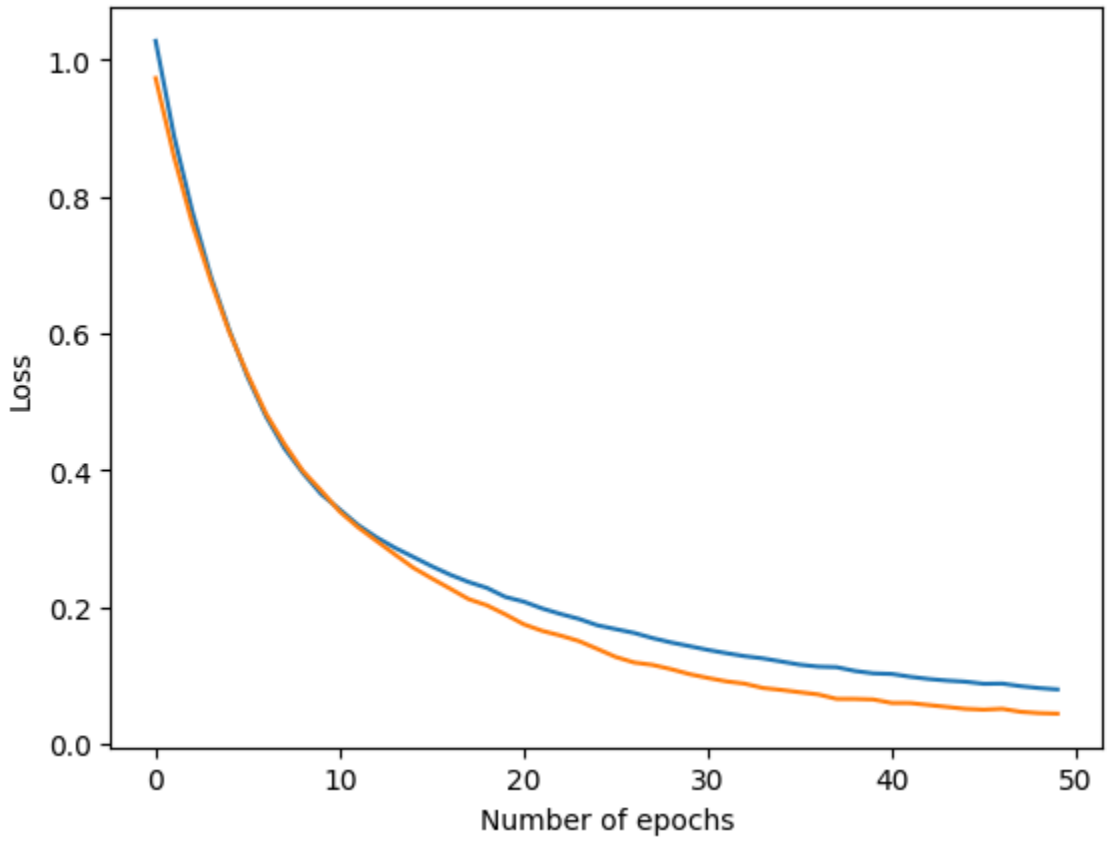
The Training and validation loss graph through epochs :

Firstly we saved the model in variable named history to use it now to get the history of the model we did .history()

```
train_loss= history.history['loss']
val_loss= history.history['val_loss']
```

Also we saved the number of epochs in variable epochs which enabled us to use itin visualizing

```
plt.plot(train_loss)
plt.plot(val_loss)
plt.xlabel('Number of epochs')
plt.ylabel('Loss')
plt.show()
```



It decreased then stabilized also we tried in the hyperparameter tuning phase increasing the number of epochs and the acc decreased so the graph is realistic and we can rely on it

Performance Comparison: SVM vs. Neural Network

SVM (RBF Kernel) Performance:

- **Accuracy**: 0.9333 (93.33%)

Classification Report:

- **Setosa**: Precision = 1.00, Recall = 1.00, F1-score = 1.00
- **Versicolor**: Precision = 0.88, Recall = 0.93, F1-score = 0.90
- **Virginica**: Precision = 0.93, Recall = 0.87, F1-score = 0.90
- **Macro Avg**: Precision = 0.93, Recall = 0.93, F1-score = 0.93

Neural Network Performance:

- **Accuracy**: 0.9111 (91.11%)

Classification Report:

- **Setosa**: Precision = 1.00, Recall = 1.00, F1-score = 1.00
- **Versicolor**: Precision = 0.82, Recall = 0.93, F1-score = 0.88
- **Virginica**: Precision = 0.92, Recall = 0.80, F1-score = 0.86
- **Macro Avg**: Precision = 0.92, Recall = 0.91, F1-score = 0.91

Comparison:

- **Accuracy**: SVM (RBF) outperforms the NN slightly (93.33% vs. 91.11%).
- **Precision, Recall, F1-score**:
    - Both models perfectly classify Setosa (precision, recall, F1-score = 1.00).
    - For Versicolor, both have the same recall (0.93), but SVM has higher precision (0.88 vs. 0.82) and F1-score (0.90 vs. 0.88).
    - For Virginica, SVM has better recall (0.87 vs. 0.80) and F1-score (0.90 vs. 0.86), with similar precision (0.93 vs. 0.92).
    - Overall (macro avg), SVM slightly outperforms the NN across all metrics.

- **Confusion Matrix**:
    - ★ Both models correctly classify all 15 Setosa samples.
    - ★ Both misclassify 1 Versicolor sample as Virginica.
    - ★ SVM misclassifies 2 Virginica samples as Versicolor, while the NN misclassifies 3, indicating SVM's slight edge in distinguishing Virginica.

Overfitting Assessment

- SVM (RBF Kernel)
  - ★ **Training Accuracy**: 0.9714 (97.14%)
  - ★ **Test Accuracy**: 0.9333 (93.33%)
  - ★ **Gap**: 97.14%−93.33%=3.81%

The 3.81% gap is small, suggesting the model isn't overfitting significantly. The SVM with the RBF kernel is capturing the underlying patterns without memorizing the training data excessively.

- Neural Network:

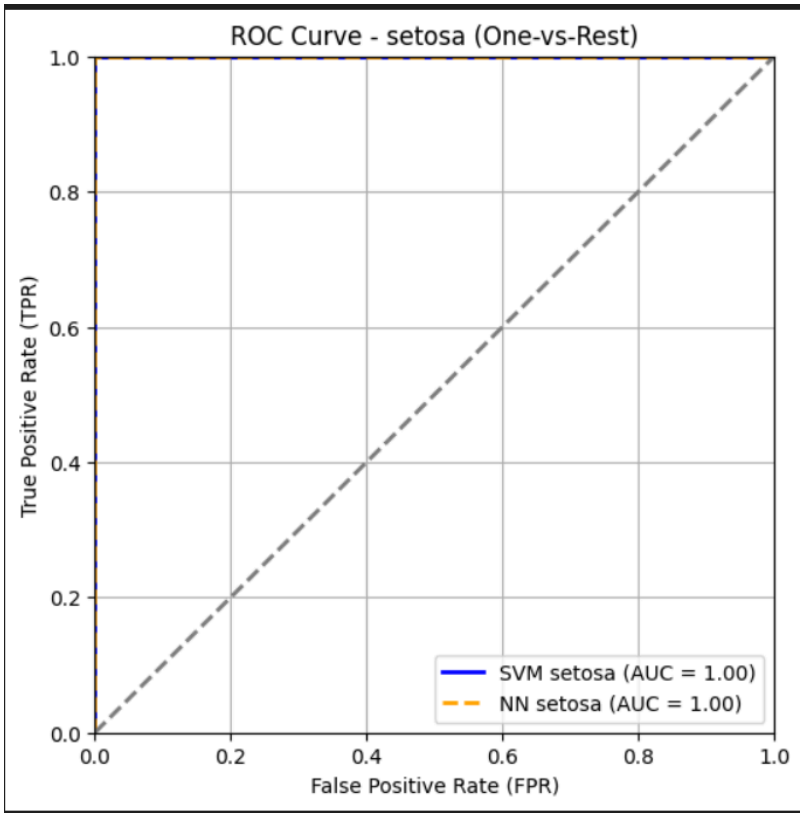
  The training accuracy (98%) is higher than the test accuracy (91.11%), a 7% gap, indicating overfitting.

  Early stopping helped mitigate overfitting by restoring the best weights, but the model still overfits slightly.

  However, to reduce overfitting we could:

- Add dropout layers (e.g., Dropout(0.3)) after the hidden layers to prevent over-reliance on specific neurons.
- Apply L2 regularization (e.g., kernel_regularizer='l2') to penalize large weights.

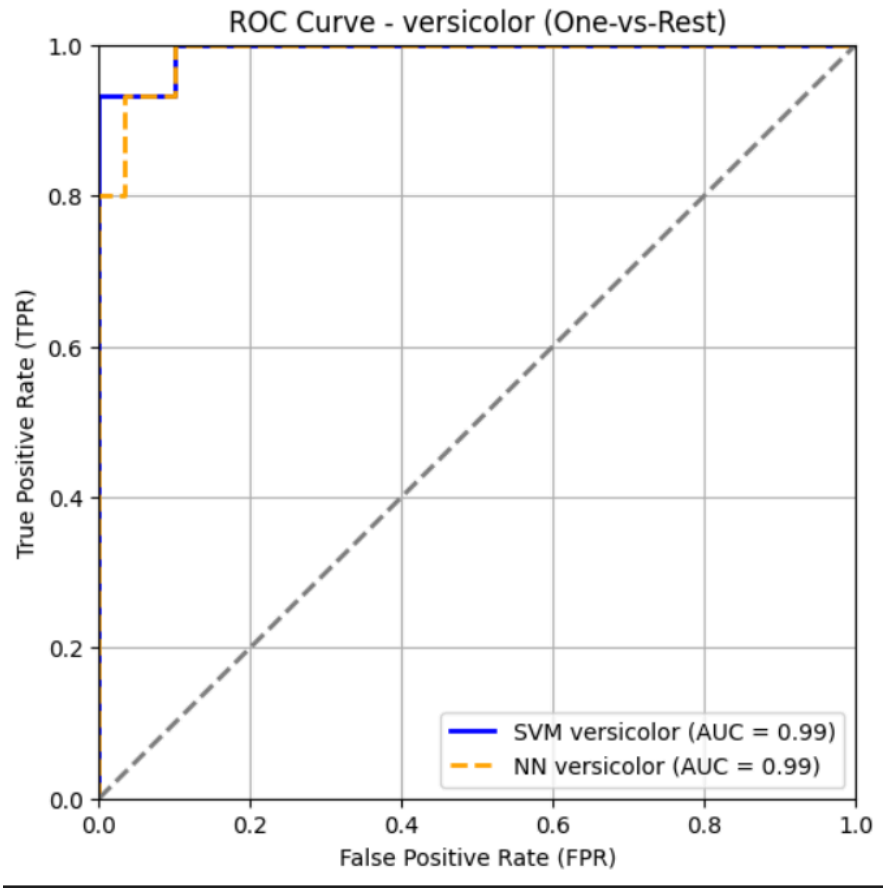
ROC Plot Analysis:



ROC Curve:
The ROC curve for Setosa shows perfect performance (AUC = 1.00) for both SVM and NN, with TPR = 1.0 and FPR = 0.0 at the optimal threshold.

TPR and FPR:

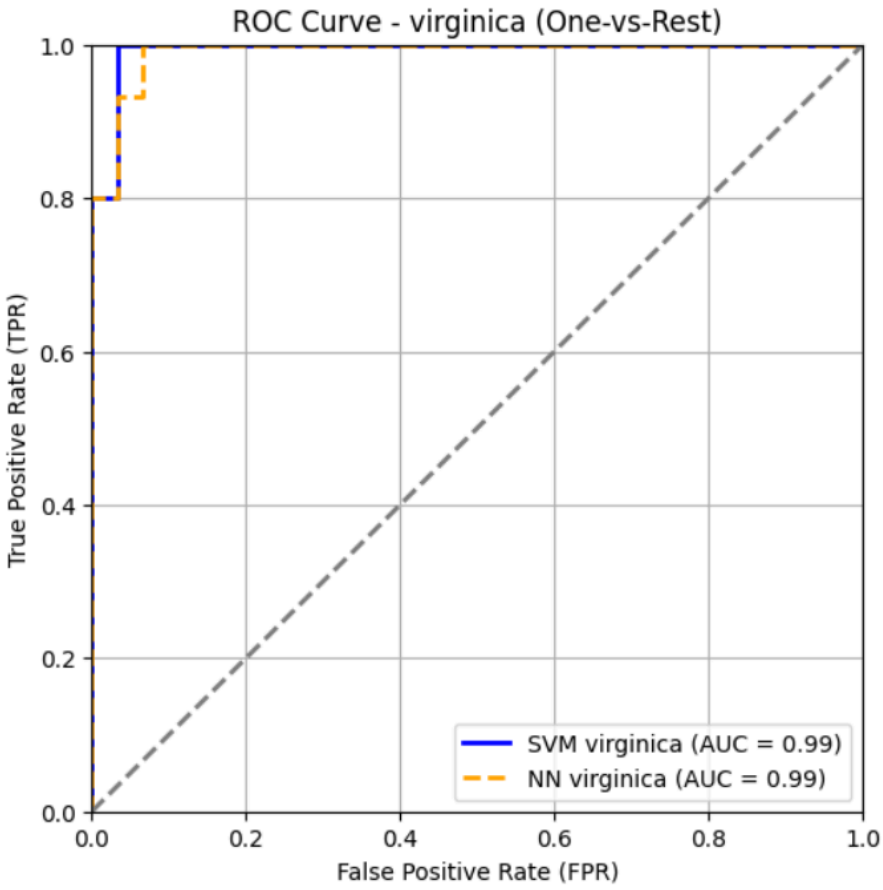
TPR = 1.0 (all Setosa samples correctly classified)
FPR = 0.0 (no non-Setosa samples misclassified as Setosa).



ROC Curve:
The ROC curve for Versicolor shows near-perfect performance (AUC = 0.99) for both SVM and NN, with TPR ≈ 0.93 and FPR ≈ 0.02–0.067 at the optimal threshold.

TPR and FPR:

TPR = 0.933 (14/15 Versicolor samples correctly classified).
FPR ≈ 0.067 (2/30 non-Versicolor samples misclassified as Versicolor).



ROC Curve:
The ROC curve for Virginica shows near-perfect performance (AUC = 0.99) for both SVM and NN, with TPR ≈ 0.87 (SVM) or 0.80 (NN) and FPR ≈ 0.02–0.033 at the optimal threshold.

TPR and FPR:

SVM: TPR = 0.867 (13/15 Virginica samples correctly classified), FPR ≈ 0.033 (1/30 non-Virginica samples misclassified as Virginica).
NN: TPR = 0.80 (12/15 Virginica samples correctly classified), FPR ≈ 0.033 (same false positives).