ARTIN

Artificial Intelligence

Lab work
From Linear Classifiers to SVMs sv

Student:
So Onishi
Taichi Haraguchi

November 29, 2023

1 Loading and splitting data

a) Download the animals10classes dataset from hippocampus, which is a subset of the Caltech 101 dataset. The animal subset is composed of images belonging to one among 10 classes. Run the code below to check images and labels are read correctly and store the name of the classes in the list labelNamesAll

I used the below program which the purpose is to load the label (class name) of the image.

```
IMDIR = './animals10classes'

#Keep4students
def loadImagesAndLabels(IMDIR):

  labelNamesAll = []

  for root, dirnames, filenames in os.walk(IMDIR):
      labelNamesAll.append(dirnames)

  labelNamesAll = labelNamesAll[0]
  return labelNamesAll

labelNamesALL = loadImagesAndLabels(IMDIR)
print(labelNamesALL)
```

Brief code description;

Function definition and purpose: The loadImagesAndLabels function scans all subdirectories in a given directory (IMDIR) and collects their names as labels in a list.

Directory traversal: os.walk(IMDIR) recursively traverses the specified directory and provides a list of each folder (root), subdirectory (dirnames), and file name (filenames).

Collecting labels: The code adds the subdirectory names to the labelNamesAll list. These subdirectory names actually serve as labels (class names) for the images.

Formatting the label list: labelNamesAll[0] retrieves a list of all subdirectory names in the initial root directory (IMDIR).

Function return value: Finally, the function returns a list of subdirectory names contained in IMDIR.

Output result: Finally, print(labelNamesALL) prints the list of labels obtained.

As a result of this code, the names of 10 different animal classes were retrieved as a list. This can be used as labels (target classes) in the image classification task.

```
['ant', 'butterfly', 'dragonfly', 'lobster', 'crocodile',
 'starfish', 'octopus', 'sea\_horse', 'flamingo', 'crayfish']
```

b) Use the BuildDataset function to create a reduced dataset. In this notebook, we will notably deal with binary classification problems. Print the sizes of the data matrix X and the label vector Y. Print as well the retained list of labels and the content of Y.

The following function is called buildDataset which creates a reduced data set.

```
def buildDataset (IMDIR,labelNamesAll,K=2,N=100,imHeight=100,
imWidth=100,seed=50):
    X = np.zeros([K*N,imHeight*imWidth]) #data matrix, one image per row
    Y = -np.ones([K*N,1]) #label indices initiallized to -1
    labelNames = [] #list of retained categories

    random.seed(a=seed) #comment to make each run random

    globalCount = 0

    for i in range(K):
        #Randomly choose a new category
        while True:
            lab = random.randint(0,len(labelNamesAll)-1)
            if lab not in labelNames:
                break

        filedir = os.path.join(IMDIR,labelNamesAll[lab])
        print('The chosen label ',i, ' is ',labelNamesAll[lab])
        print('It will be read in',filedir)

        labelNames.append(labelNamesAll[lab])

        classCount = 0
        for filename in os.listdir(filedir):
            f = os.path.join(filedir, filename)
            if f.endswith(('.jpg')) and (classCount < N):
                image = skimage.io.imread(f, as_gray=True)
                image = skimage.transform.resize
                    (image, [imHeight,imWidth],mode='constant')#,anti_aliasin
                X[globalCount,:] = image.flatten()
                Y[globalCount,:] = i
                globalCount += 1
                classCount += 1

    print("Total number of samples",globalCount)
    X = X[:globalCount,:]
    Y = Y[:globalCount,:]

    return X,Y,labelNames
```

The purpose of this function is to construct a data matrix X and a label vector (class labels)

Y from an image file.

The meanings of the parameters are as follows;

- **K**: the number of classes to consider
- **N**: the maximum number of images to read from each category (the number of images per label is variable).
- **imHeight, imWidth**: define the size of the target image. All read images are resized to imHeight x imWidth.
- **seed**: fixes the random seed to be able to reproduce the results.

Brief description of the function;

Initialize data matrix and label vectors: X is the data matrix with each image as a row (filled with zeros at initialization). Y is a vector of image class labels (filled with -1 at initialization).

Class Selection and Data Loading: The function randomly selects K different classes. For each selected class, the function reads an image from the specified directory, resizes it, and processes it in grayscale. Each image is flattened (converted to a 1D vector) and stored in the corresponding row of the data matrix X.

Label Assignment: For each image, a label (an integer from 0 to K-1) corresponding to the class to which the image belongs is assigned and stored in vector Y.

Final adjustment of the data matrix and label vectors: Function adjusts X and Y based on the number of images actually read. This is important when there are fewer than N images in some classes.

Try actually using the buildDataset function.

```
K=2
imHeight=100
imWidth=100

#Call the buildDataset function
X, Y, labelNames = buildDataset(IMDIR, labelNamesALL)

#Check the built dataset classes
print("Used labels",labelNames)
print("Size of data matrix", X.shape)
print("Class labels", Y.T)
```

The number of classes considered is set to 2 (K=2) and the image size is set to 100 x 100.

The output results from the above program are as follows;

```
The chosen label  0  is  sea_horse
It will be read in ./animals10classes/sea_horse
The chosen label  1  is  crocodile
```

```
It will be read in ./animals10classes/crocodile
Total number of samples 107
Used labels ['sea_horse', 'crocodile']
Size of data matrix (107, 10000)
Class labels [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1.]]
```

The results show that two classes of images are selected, sea_horse and crocodile, there are 107 data samples in total, each sample consists of 10000 features (flattened 100x100 pixel images), and these are stored in the that they are stored in a data matrix X. We can also see that 0 indicates sea_horse and 1 indicates crocodile

c) Split the dataset into a train (70% of samples), a validation (15%) and a test set (15% samples) subsets.

The following code was created and executed to split the data.

```
# shuffle values randomly
X, Y = shuffle(X, Y, random_state=42)

# define rate of train data
train_size = int(0.7 * len(X))
val_size = int(0.15 * len(X))
test_size = len(X) - (train_size + val_size)

# split dataset
# X data include features, Y data include the actual answer
X_train, X_val, X_test = X[: train_size],
    X[train_size: train_size + val_size],
    X[train_size + val_size: train_size + val_size + test_size]

Y_train, Y_val, Y_test = Y[: train_size],
    Y[train_size: train_size + val_size],
    Y[train_size + val_size: train_size + val_size + test_size]

print("size of train dataset",X_train.shape)
print("size of val dataset",X_val.shape)
print("size of test dataset",X_test.shape)
print("train target vector",Y_train.T)
print("val target vector",Y_val.T)
print("test target vector",Y_test.T)
```

The above program involves splitting the dataset into subsets for training, validation, and testing. This process is important for proper training and evaluation of the model.

X, Y = shuffle(X, Y, random_state=42): This line randomly shuffles the data matrix X (features) and the label vector Y (correct labels). This shuffling process can be reproduced by setting random_state

Purpose of the Shuffle;

- **Elimination of bias**: If the data sets are arranged in a particular order (e.g., samples of one class in a row), the model may learn that order. Shuffling eliminates this bias.
- **Improved generalizability**: training in a random order prevents the model from over-fitting a particular pattern in the data and allows for more generalized learning.

The output results from the above program are as follows;

```
size of train dataset (74, 10000)
size of val dataset (16, 10000)
size of test dataset (17, 10000)
train target vector[[1. 0. 0. 1. 1. 1. 0. 0. 1. 0. 1. 0.
0. 1. 0. 1. 0. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 1.
1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 1.
0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 1. 1. 0.
0. 0. 1. 0. 1. 1.]]
val target vector[[1. 1. 0. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 0. 0. 1.]]
test target vector[[0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 0. 1.]]
```

From the results, the sizes of the training, validation, and test sets are displayed as (74, 10000), (16, 10000), and (17, 10000), respectively. This means that each set contains 74, 16, and 17 images, with each image consisting of 10000 features. The "target vector" (Y_train.T, Y_val.T, Y_test.T) for each set indicates the class label of the images in that set.

2 Support Vector Machines

a) Linear SVM Create an SVC model with a linear kernel and default values svmLin=SVC(kernel='linear')

A program that uses SVM to perform classification tasks is as follows;

```
# Create, train and test an svm model
svmLin = SVC(kernel='linear')

# np.ravel returns 1 dimension
svmLin.fit(X_train, np.ravel(Y_train))

# do prediction on test dataset
Y_pred = svmLin.predict(X_test)
```

```
# compare predicted result and test result
print("Predicted Labels: ", Y_pred)
print("ACtual Labels: ", np.ravel(Y_test))

# calculate error of prediction
error = sum(Y_pred != Y_test.ravel())
print("Number of errors: ", error)

print('Accuracy: ', svmLin.score(X_test, Y_test))
print()

# Create, train and test a "probabilistic" SVM model
svmLin = SVC(kernel='linear', probability=True)

svmLin.fit(X_train, np.ravel(Y_train))

# Print predicting the membership probability
# of each class on the test dataset
print("Membership Probability: \n",svmLin.predict_proba(X_test))
```

The output results from above program are as follows;

```
Predicted Labels:  [0. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 1. 1. 1. 1. 0. 1.]
ACtual Labels:  [0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 0. 1.]
Number of errors:  8
Accuracy:  0.5294117647058824

Membership Probability:
 [[0.57894987 0.42105013]
 [0.59297944 0.40702056]
 [0.59606721 0.40393279]
 [0.58508439 0.41491561]
 [0.60052426 0.39947574]
 [0.5888919  0.4111081 ]
 [0.59148856 0.40851144]
 [0.59707477 0.40292523]
 [0.58378302 0.41621698]
 [0.58843988 0.41156012]
 [0.584526   0.415474  ]
 [0.60023224 0.39976776]
 [0.60235285 0.39764715]
 [0.5931351  0.4068649 ]
 [0.59681056 0.40318944]
 [0.57542759 0.42457241]
 [0.59344595 0.40655405]]
```

From the results, the number of errors (the number of cases where the predictions differ from the actual labels) is 8, which means that the model made wrong predictions on some of the samplings. The accuracy is about 53%, which indicates that the model makes about half of the predictions on the test data set correctly.

The predict_proba function indicates the probability that each sample in the test set belongs to each class. The results of the predict_proba function in this case can be said to be close to equal. Therefore, it is possible that the model does not completely identify the data. This suggests that either the features do not have enough information to distinguish the classes, or the model does not fully capture the complexity of the data. The present results indicate that the model has some degree of uncertainty with respect to the test data.

To eliminate uncertainty, feature normalization may be used. Normalization ensures that all features have the same scale and prevents certain features from influencing the model decisions. Normalization is especially useful in this case because the pixel intensity values of the image data are often on different scales.

b) Support vectors

The program to calculate the number of support vectors and parameters for the model is as follows (Show main part);

```
# get the number of support vector
num_support_vectors = svmLin.n_support_.sum()

# get the number of total parameter
total_parameters = num_support_vectors + 1
```

The output results from above program are as follows;

```
The Number of Support Vector: 70
The Number of Parameters: 71
there are 3 hyperparameters, and sometimes more depending on these 3
```

Support vector is the data point closest to the line dividing the data.

svmLin.n_support_.sum() determines the total number of support vectors across all classes of the SVM model. This is the number of data points that are important for defining the classification decision boundaries of the model.

The "number of parameters" is the number of support vectors plus one. This is because in addition to the number of support vectors, the bias term (often added as a constant term) of the SVM model is also counted as a parameter.

In addition, the first support vector is obtained and it is displayed as an image.

c) Kernel SVMs

Below the program is for training three additional SVC models each using one among the kernel functions (show only main part);

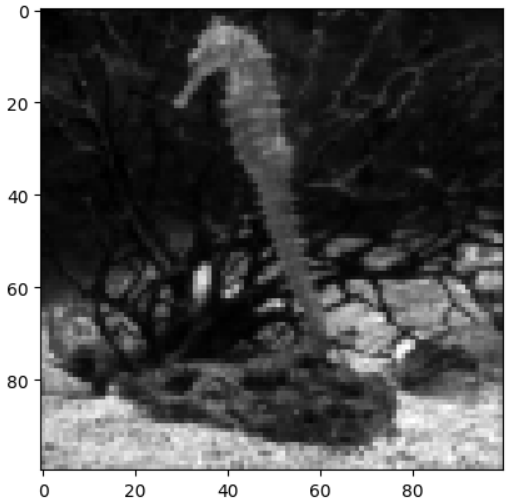

Figure 1: First support vector

```
kernels = ['rbf', 'poly', 'sigmoid']
    for kernel in kernels:
    svm = SVC(kernel=kernel)
    svm.fit(X_train, np.ravel(Y_train))
    Y_pred = svm.predict(X_test)
    error = sum(Y_pred != Y_test.ravel())
```

The output results from above program are as follows;

```
Kernel: rbf
Number of support vectors: 69
Hyperparameters: {'C': 1.0, 'break_ties': False, 'cache_size': 200,
'class_weight': None, 'coef0': 0.0, 'decision_function_shape': 'ovr',
'degree': 3, 'gamma': 'scale', 'kernel': 'rbf', 'max_iter': -1,
'probability': False, 'random_state': None, 'shrinking': True,
'tol': 0.001, 'verbose': False}
Number of errors: 5
Accuracy: 0.7058823529411765

Kernel: poly
Number of support vectors: 65
Hyperparameters: {'C': 1.0, 'break_ties': False, 'cache_size': 200,
'class_weight': None, 'coef0': 0.0, 'decision_function_shape': 'ovr',
'degree': 3, 'gamma': 'scale', 'kernel': 'poly', 'max_iter': -1,
'probability': False, 'random_state': None, 'shrinking': True,
'tol': 0.001, 'verbose': False}
Number of errors: 7
Accuracy: 0.5882352941176471

Kernel: sigmoid
Number of support vectors: 54
```

```
Hyperparameters: {'C': 1.0, 'break_ties': False, 'cache_size': 200,
'class_weight': None, 'coef0': 0.0, 'decision_function_shape': 'ovr',
'degree': 3, 'gamma': 'scale', 'kernel': 'sigmoid', 'max_iter': -1,
'probability': False, 'random_state': None, 'shrinking': True,
'tol': 0.001, 'verbose': False}
Number of errors: 9
Accuracy: 0.47058823529411764
```

From above result, 69 rbf kernels, 65 polynomial kernels, and 54 sigmoid kernels with support vectors.

The rbf kernel shows the highest accuracy (about 70.6%) and the lowest number of errors 5. The polynomial kernel has about 58.8% accuracy and 7 errors. The sigmoid kernel has the lowest accuracy (about 47.1%) and the highest number of errors 9.

The main hyperparameters are the same for all kernels (C: 1.0, gamma: 'scale', etc.). However, there are some hyperparameters that are specific to each kernel (e.g. degree for polynomial kernels). The results of this run show $degree = 3$ for all kernels, but this is a parameter specific to polynomial kernels and is not used for other kernels.

d) Hyperparameter tuning

Below program is which Tune the hyperparameters of both the linear and the non-linear SVMs using the validation set.

```
#Linear
print("Linear")
param_grid_lin = {'C': [0.1,1, 10, 100]}
grid_search_lin = GridSearchCV(SVC(kernel='linear', probability=True),
                               param_grid_lin)
grid_search_lin.fit(X_train, Y_train.ravel())
print('Best Parameters(Linear):', grid_search_lin.best_params_)
print("Validation Score: ",grid_search_lin.best_score_)
print()

#rbf
print("rbf")
param_grid_rbf = {'C': [0.1,1, 10, 100],
              'gamma': [1,0.1,0.01,0.001]}
grid_search_rbf = GridSearchCV(SVC(kernel='rbf', probability=True),
                               param_grid_rbf)
grid_search_rbf.fit(X_train, Y_train.ravel())
print('Best Parameters(rbf):', grid_search_rbf.best_params_)
print("Validation Score: ",grid_search_rbf.best_score_)
print()

#poly
print("poly")
param_grid_poly = {'C': [0.1,1, 10, 100],
```

```
                'gamma': [1,0.1,0.01,0.001],
                'degree': [2, 3, 5]}
grid_search_poly = GridSearchCV(SVC(kernel='poly', probability=True),
                                param_grid_poly)
grid_search_poly.fit(X_train, Y_train.ravel())
print('Best Parameters(poly):', grid_search_poly.best_params_)
print("Validation Score: ",grid_search_poly.best_score_)
print()

#sigmoid
print("sigmoid")
param_grid_sigmoid = {'C': [0.1,1, 10, 100],
                'gamma': [1,0.1,0.01,0.001]}
grid_search_sigmoid = GridSearchCV(SVC(kernel='sigmoid', probability=True),
                                   param_grid_sigmoid)
grid_search_sigmoid.fit(X_train, Y_train.ravel())
print('Best Parameters(sigmoid):', grid_search_sigmoid.best_params_)
print("Validation Score: ",grid_search_sigmoid.best_score_)
```

GridSearchCV function performs cross-validation for each specified hyperparameter combination to find the best combination.

Hyperparameter tuning based on training data using grid_search.fit.

The output results from above program are as follows;

```
Linear
Best Parameters(Linear): {'C': 0.1}
Validation Score:  0.48571428571428565

rbf
Best Parameters(rbf): {'C': 10, 'gamma': 0.001}
Validation Score:  0.6219047619047618

poly
Best Parameters(poly): {'C': 0.1, 'degree': 5, 'gamma': 1}
Validation Score:  0.6495238095238095

sigmoid
Best Parameters(sigmoid): {'C': 1, 'gamma': 0.001}
Validation Score:  0.6076190476190476
```

From above result, 'C': 10, 'gamma': 0.001, 'kernel': 'rbf' was chosen as the best combination. This means that using the rbf kernel, C of 10 and gamma of 0.001 are the most effective settings for this data set.

The best model obtained had an accuracy of about 65% on the validation data set. This shows how well the model generalizes to unknown data.

Validation score of 65% indicates that the model's performance still has room for improvement

3 Performance Measures

a) Fill in the function bellow to computing different evaluation measures and give a performance report

Below program is which compute different evaliation measures;

```
def print_confusion_matrix(y_test, y_pred):

    # True positive
    TP = 0

    # False positive
    FP = 0

    # True negative
    TN = 0

    # False negative
    FN = 0

    for i in range(len(y_pred)):
        if y_test[i]==y_pred[i]==1:
            TP += 1
        if y_pred[i]==1 and y_test[i]!= y_pred[i]:
            FP += 1
        if y_test[i]==y_pred[i]==0:
            TN += 1
        if y_pred[i]==0 and y_test[i]!=y_pred[i]:
            FN += 1

    return TP, FP, TN, FN

def compute_measures(y_test,y_pred, positiveClass=1):

    measures = dict()

    eps = 1e-12

    TP, FP, TN, FN = print_confusion_matrix(y_test, y_pred)

    measures['TP'] = TP
    measures['TN'] = TN
    measures['FP'] = FP
    measures['FN'] = FN
```

```
    # Accuracy
    measures['accuracy'] = (TP+TN)/(TP+TN+FP+FN+eps)

    # Precision or positive predictive rate
    measures['precision'] = TP/(TP+FP+eps)

    # Specificity or true negative rare
    measures['specificity']= TN/(TN+FP+eps)

    # Recall or true positive rate
    measures['recall'] = TP/(TP+FN+eps)

    # F-measure
    measures['f1'] = (2*measures['precision']*measures['recall'])
                        /(measures['precision']+measures['recall']+eps)

    # Negative Predictive Value
    measures['npv'] = TN/(TN+FN+eps)

    # False Predictive Value
    measures['fpv'] = FP/(TP+FP+eps)

    return measures

print("Positive Class is 1")
compute_measures(Y_test, Y_pred)
```

The output results from above program are as follows (when $kernel = sigmoid$);

```
Positive Class is 1

{'TP': 2,
 'TN': 6,
 'FP': 2,
 'FN': 7,
 'accuracy': 0.47058823529409,
 'precision': 0.499999999999875,
 'specificity': 0.7499999999999063,
 'recall': 0.22222222222219754,
 'f1': 0.3076923076918343,
 'npv': 0.46153846153842604,
 'fpv': 0.24999999999996875}
```

From above result, we can see that the overall performance of this model is poor. In particular, accuracy is low at 47.1% and recall is very low at 22.2%, indicating that the

model struggles to properly identify positive classes. precision is 50%, but the f1 score is also low due to the low recall.

These results indicate that the model tends to miss positive classes in particular, suggesting that the model needs to be improved.

Blow program is which using in-build sklearn measures;

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred, labels=[0, 1])
print(f"Confusion Matrix:\n{cm}" )

tn, fp, fn, tp = cm.ravel()
(tn, fp, fn, tp)

from sklearn.metrics import classification_report

print(classification_report(Y_test, Y_pred))
```

The output results from above program are as follows;

```
Confusion Matrix:
[[6 2]   # [[True Negative (TN)  False Positive (FP)]
 [7 2]]  #  [False Negative (FN) True Positive (TP)]]

              precision    recall  f1-score   support

         0.0       0.46      0.75      0.57         8
         1.0       0.50      0.22      0.31         9

    accuracy                           0.47        17
   macro avg       0.48      0.49      0.44        17
weighted avg       0.48      0.47      0.43        17
```

From above 2 result, we can compare my result vs the in-build sklearn measures.

The results obtained with both methods are consistent, confirming that the manual calculation method was accurate.

It is clear that the performance of the model is poor for both methods. In particular, the recall for the 1.0 class (positive class) is low, indicating that the model misses many positive cases in identifying this class. Precision is also low, around 50%, suggesting that the model's performance needs to be improved.

b) ROC curves

Below program is which polt the ROC curves of all the trained svm models before and after hyperparameter tuning (show only main part);

```
# Logistic Regression
from sklearn.metrics import roc_curve

# fit a model
model = LogisticRegression()
model.fit(X_train, np.ravel(Y_train))

# plot no skill roc curve
plt.plot([0, 1], [0, 1], linestyle='--', label='No Skill')

# return predicted class (0 or 1)
Y_pred_logreg_probas = model.predict_proba(X_test)

# calculate roc curve for logistic regression
fpr, tpr, _ = roc_curve(Y_test, Y_pred_logreg_probas[:,1])
plt.plot(fpr, tpr, marker='.', label='Logistic')

# calculate roc curve using the SVC parameter
# in 'rbf', 'linear', 'poly', 'sigmoid'
kernels = ['rbf', 'linear', 'poly', 'sigmoid']
for kernel in kernels:
    svm = SVC(kernel=kernel, probability=True)
    svm.fit(X_train, Y_train)
    Y_pred_svc_best_probas = svm.predict_proba(X_test)
    fpr, tpr, _ = roc_curve(Y_test, Y_pred_svc_best_probas[:,1])
    plt.plot(fpr, tpr, marker='v', label=kernel)
```

Below program is which polt the ROC curves using the best SVC parameter (show only main part);

```
# calculate roc curve for logistic regression
fpr_logreg, tpr_logreg, _ = roc_curve(Y_test, Y_pred_logreg_probas[:,1])
plt.plot(fpr_logreg, tpr_logreg, marker='.', label='Logistic')

# SVM with Linear Kernel after tuning
svm_lin = SVC(kernel ='linear' ,probability=True, C=0.1)
svm_lin.fit(X_train, Y_train.ravel())
Y_pred_svm_lin_probas = svm_lin.predict_proba(X_test)
fpr_lin, tpr_lin, _ = roc_curve(Y_test, Y_pred_svm_lin_probas[:, 1])
plt.plot(fpr_lin, tpr_lin, marker='.', label=f'Linear best')

# SVM with RBF after tuning
svm_rbf = SVC(kernel ='rbf' ,probability=True, C=10, gamma=0.001)
svm_rbf.fit(X_train, Y_train.ravel())
Y_pred_svm_rbf_probas = svm_rbf.predict_proba(X_test)
fpr_rbf, tpr_rbf, _ = roc_curve(Y_test, Y_pred_svm_rbf_probas[:, 1])
plt.plot(fpr_rbf, tpr_rbf, marker='.', label=f'rbf best')
```

```
# SVM with Poly after tuning
svm_poly = SVC(kernel ='poly' ,probability=True, C=0.1, degree=5, gamma=1)
svm_poly.fit(X_train, Y_train.ravel())
Y_pred_svm_poly_probas = svm_poly.predict_proba(X_test)
fpr_poly, tpr_poly, _ = roc_curve(Y_test, Y_pred_svm_poly_probas[:, 1])
plt.plot(fpr_poly, tpr_poly, marker='.', label=f'poly best')

# SVM with Sigmoid after tuning
svm_sigmoid = SVC(kernel ='sigmoid' ,probability=True, C=1, gamma=0.001)
svm_sigmoid.fit(X_train, Y_train.ravel())
Y_pred_svm_sigmoid_probas = svm_sigmoid.predict_proba(X_test)
fpr_sigmoid, tpr_sigmoid, _ = roc_curve(Y_test, Y_pred_svm_sigmoid_probas[:, 1])
plt.plot(fpr_sigmoid, tpr_sigmoid, marker='.', label=f'sigmoid best')
```

The output results from above program are as follows;

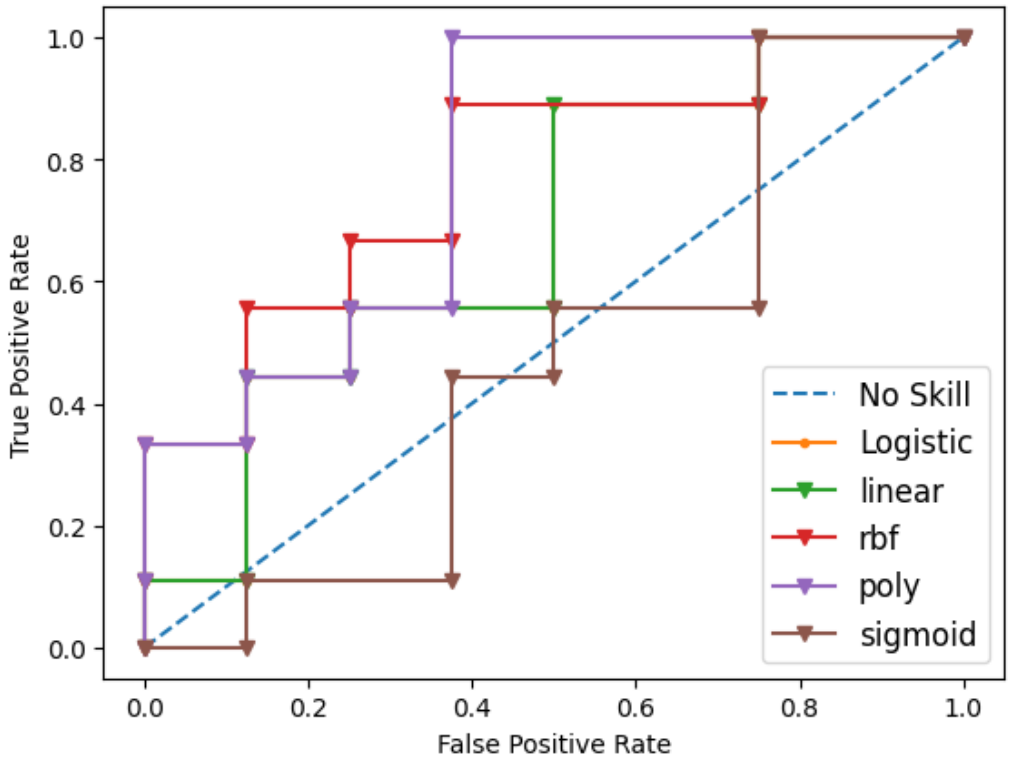

Figure 2: ROC curves using default parameters

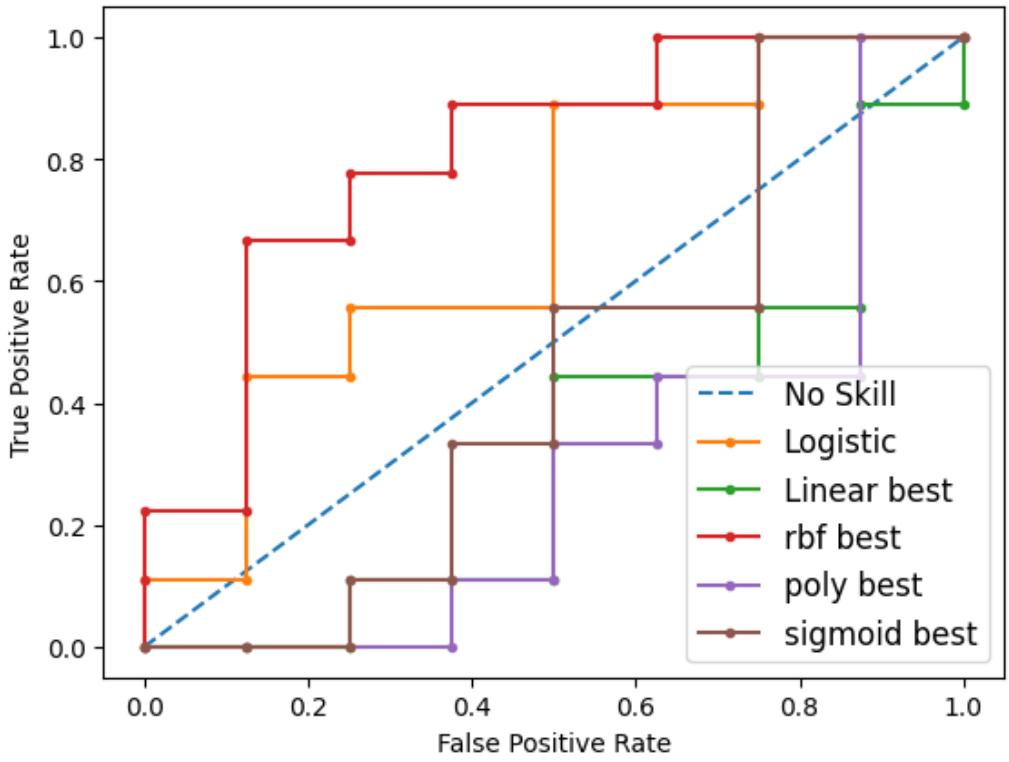

Figure 3: ROC curves using best parameters

Comparing the results of each kernel before and after tuning, rbf after tuning is better than one before tuning. However, others got worse. It indicates the possibility of overfitting. we also conclude that our best model is the one using rbf after tuning. This is because the area under the curve is the largest.

c) Qualitative Results

Applying the result of "b) ROC curves", our best model is rbf. Thus, we coded using it as shown in the following program. Figure 4 shows Qualitative Results. Most predictions are same as ground truth. However, some predictions are different.

Below program is which shows some of the test images and writes on the title the predictions vs the ground truth labels (show only main part);

```
Y_bes = Y_pred_svm_rbf_probas

fig=plt.figure()
imCounter = 1
for i in range(len(Y_test)):
    image=np.reshape(X_test[i,:], (imHeight,imWidth))

    plt.subplot(5,7,imCounter)
    plt.imshow(image,cmap='gray')
    plt.axis('off')
    gtLabel = labelNames[Y_test.ravel()[i].astype(int)]
    predLabel = labelNames[Y_best.ravel()[i].astype(int)]
```

```
plt.title('GT: {}. \n Pred: {}'.format(gtLabel, predLabel))

imCounter += 1
```

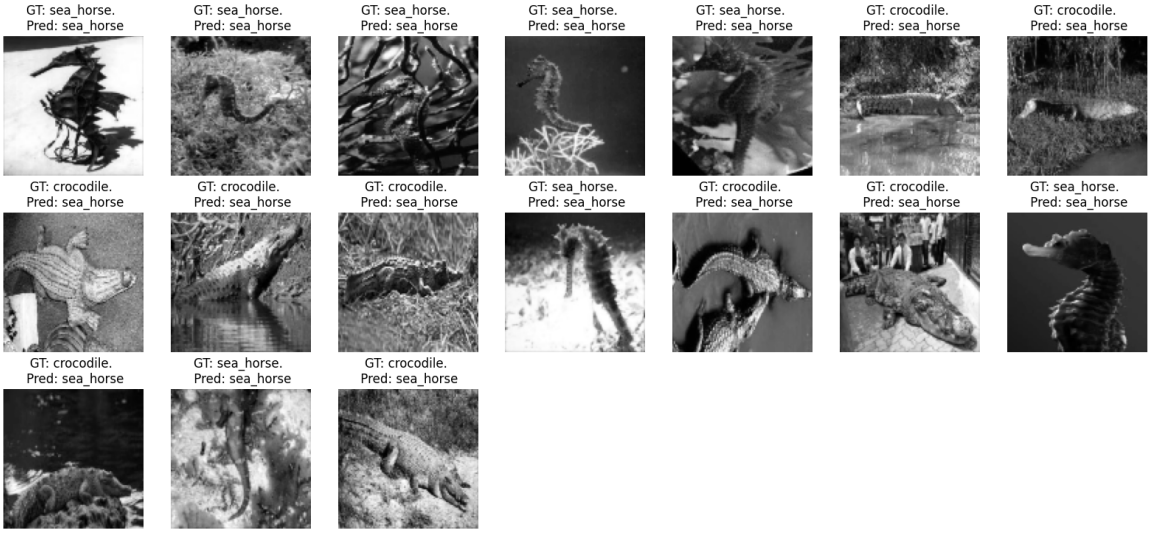

Figure 4: Prediction vs Grand truth

From above result, we can see that images can be displayed that allow comparison of predicted and the ground truth.

Furthermore, since there is a small number of matches between the predicted and ground trueth values from the images, it is considered that there is a need to improve the classification model.