UNIVERSITY OF WATERLOO

FACULTY OF ENGINEERING

ECE 457B - Assignment 2

Prepared by:

Jianxiang (Jack) Xu [20658861]

11 March 2021

Table of Contents

1 Problem 1: Nonlinear Classifier (SVM) [Code 6]

1.1 (a): Parameter tuning and Confusion matrices (15 marks)

For this dataset, SVM is chosen to be studied. In order to tune the hyper parameters to find the best model for the dataset, a various combinations of C-parameters and kernels are evaluated for 5-fold cross validation. The 5-fold validation helps to ensure the integrity and stability of the model by comparing its worst, average, and best performance.

```
def print_latex_header(SVC_PARAMS, folder):
        LINE = "\n\
\\begin{{figure}}[H]\n\
\\centering\n\
\\subfloat[1:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(1:5)]}}}} \, \n\
\\subfloat[2:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(2:5)]}}}} \, \n\
\\subfloat[3:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(3:5)]}}}} \, \n\
\\subfloat[4:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(4:5)]}}}} \, \n\
\\subfloat[5:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(5:5)]}}}} \, \n\
\\caption{{Confusion Matrices for C:{c} K:{k} 5-fold}}\n\
```

Code 1: SVM Hyper-parameters Setting

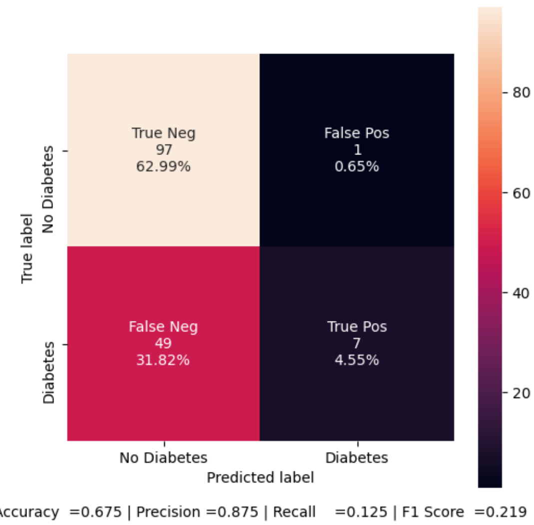
The detailed implementation can be seen:

```
        Y_pos = Y[Y==1]
        Y_neg = Y[Y==0]

        n_pos = len(Y_pos)
        n_neg = len(Y_neg)
        d_n = (n_pos - n_neg)

        X_major = X_pos
        n_repeat = np.ceil(d_n / (n_pos))
        if (n_neg > n_pos):
            X_major = X_neg
            d_n = (n_neg - n_pos)
            n_repeat = np.ceil(d_n / (n_neg))

        X_extra = np.repeat(X_major, d_n, axis=0)
        np.random.shuffle(X_extra)
        X = np.concatenate((X, X_extra[0:d_n]))
        if (n_neg > n_pos):
            Y = np.concatenate((Y, np.ones(d_n)))
        else:
            Y = np.concatenate((Y, np.zeros(d_n)))

        ic(np.shape(X))
        ic(np.shape(Y))
        diag_if_balance(data_=Y, tag_="Balanced")

    # ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    ### TRAIN & VALIDATE ###
    if ENABLE_TRAINING:
        print("=== Processing ===")
        ### SVC ###
        # Processing Automation:
        dict_of_status_log = {}
        for c in SVC_PARAMS['C']:
            status_log_k = {}
            for k in SVC_PARAMS['kernel']:
                print("=== [m:{}-C:{}-K:{}] ===".format(mode,c,k))
                # K-Fold : 5x => choose the best kernel score
                kf = KFold(n_splits=N_FOLD, shuffle=True) # randomize
                indices = kf.split(X)
                # Perform training and validation
                for trial, indices_pair in enumerate(indices):
```

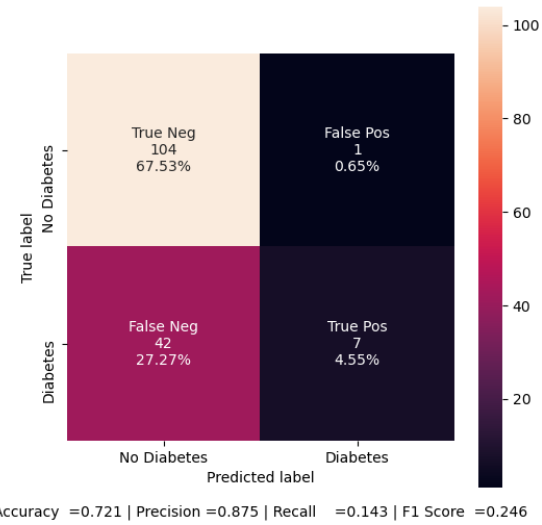
```
                # partition training and test data:
                train_index, test_index = indices_pair
                X_train, X_test = X[train_index], X[test_index]
                y_train, y_test = Y[train_index], Y[test_index]

                # declare SVC model:
                svc_ = SVC(
                    C           = c, # Regularization term
                    kernel      = k,
                )

                # FITTING ...
                svc_classifier_ = svc_.fit(X_train,y_train)

                # Report:
                ic(svc_classifier_)

                ### Test Estimators ###
                y_predict = svc_classifier_.predict(X_test) # round to evaluate

                ### REPORT GEN. ###
                conf_mat = confusion_matrix(y_test, y_predict)

                # print:
                ic(conf_mat)

                # Gen Confusion Matrix Plot
                labels = ["True Neg","False Pos","False Neg","True Pos"]
```

Code 2: SVM 10-Fold Hyper Tuning

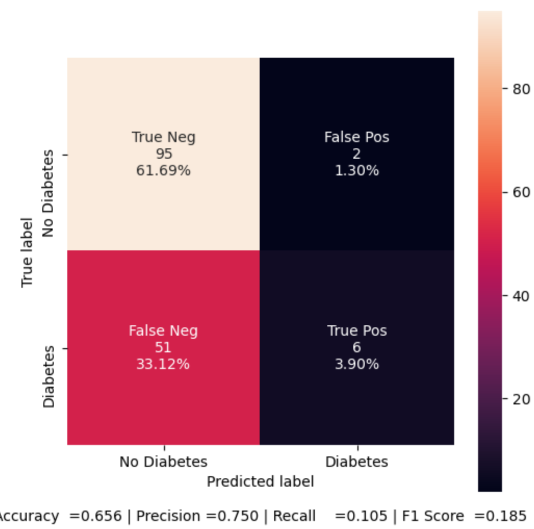
As a result, we may get a total of 90 confusion matrices for 16 sets of hyper-parameters and 5 trials each. They are as stated in **??** - **??** below:



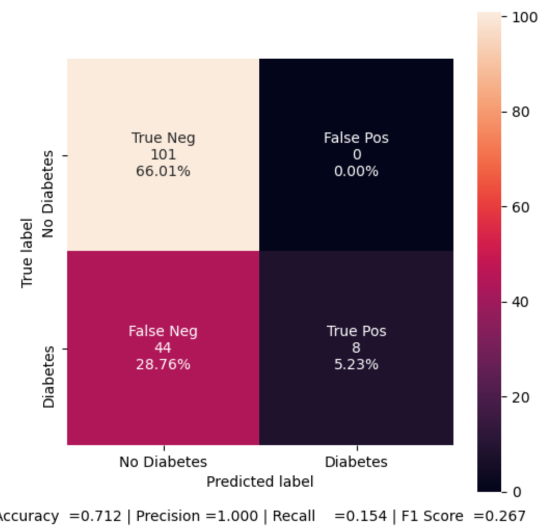
(a) 1:5-Fold



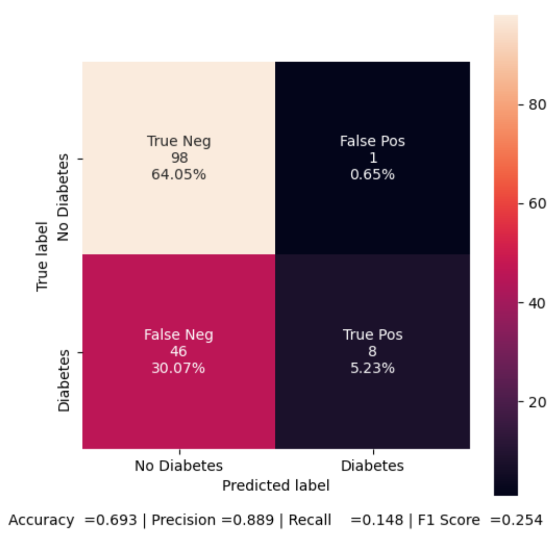
(b) 2:5-Fold



(c) 3:5-Fold

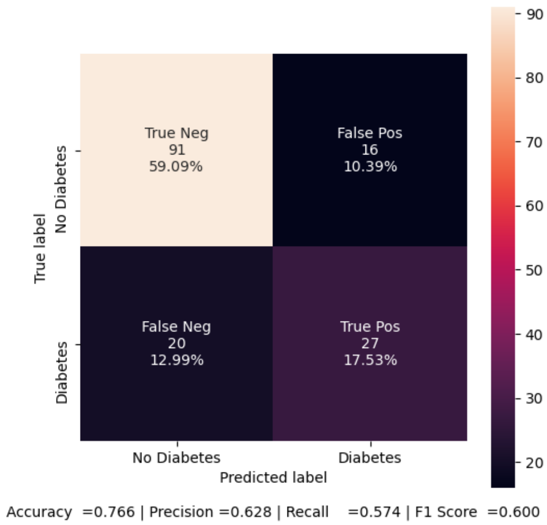


(d) 4:5-Fold

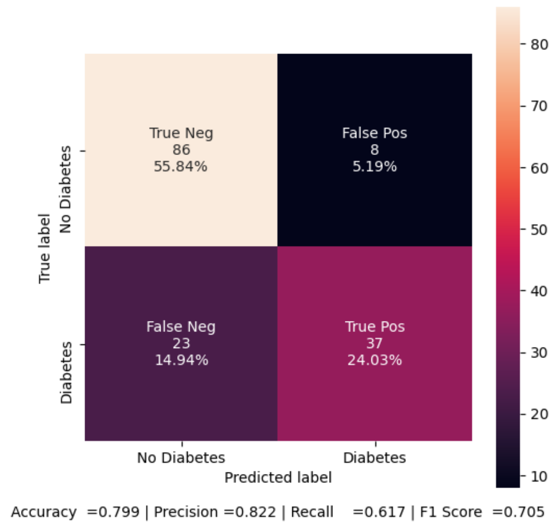


(e) 5:5-Fold

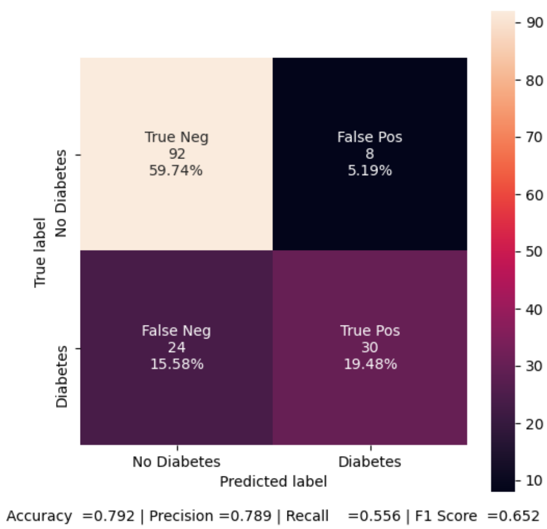
Figure 1-1. Confusion Matrices for C:0.1 K:linear 5-fold



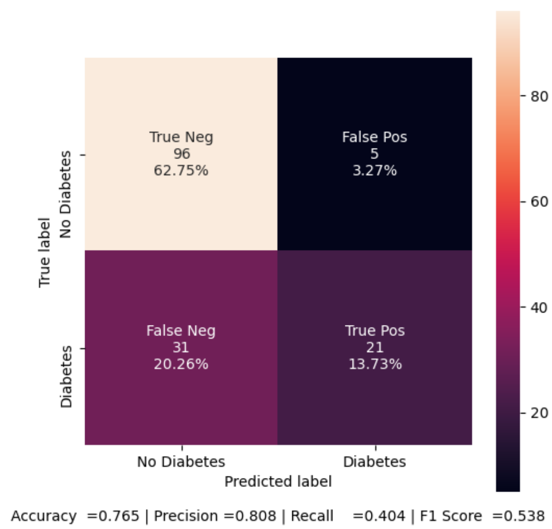
(a) 1:5-Fold



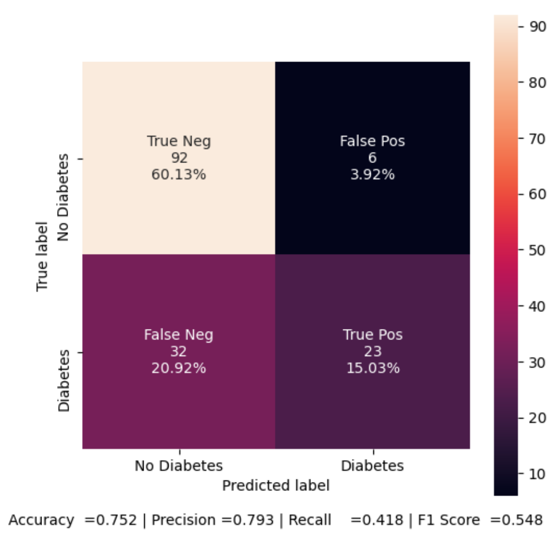
(b) 2:5-Fold



(c) 3:5-Fold

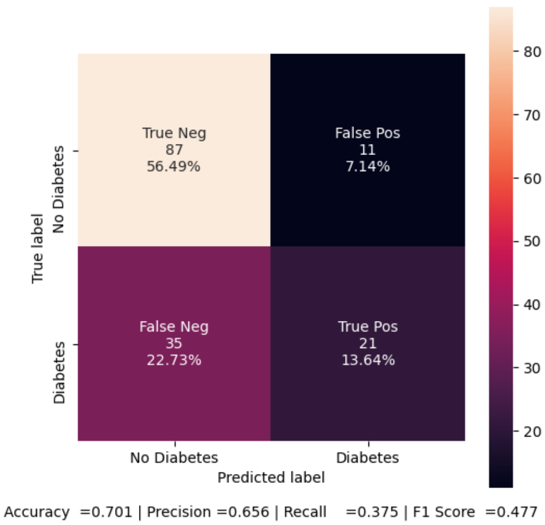


(d) 4:5-Fold

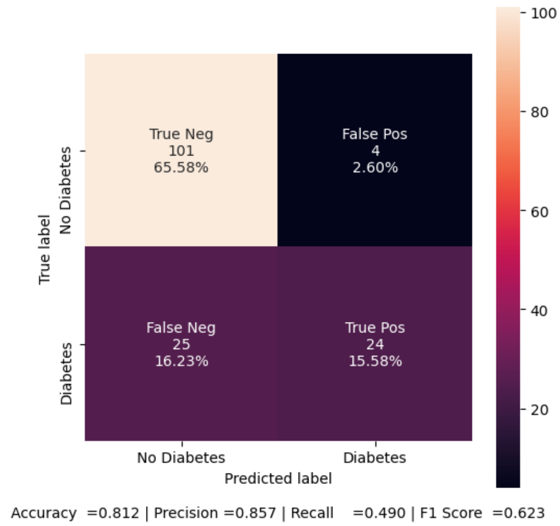


(e) 5:5-Fold

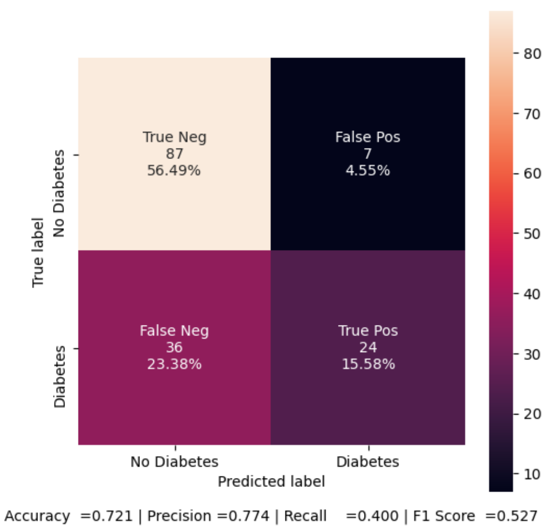
Figure 1-2. Confusion Matrices for C:0.1 K:poly 5-fold



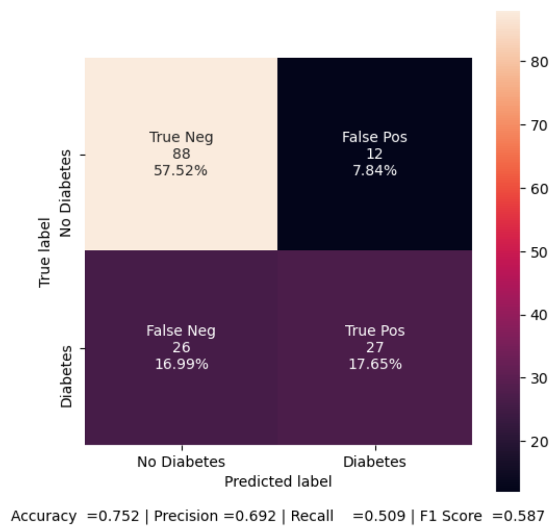
(a) 1:5-Fold



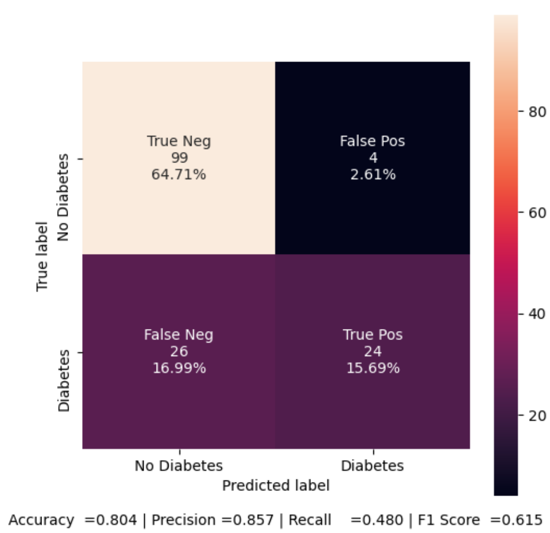
(b) 2:5-Fold



(c) 3:5-Fold

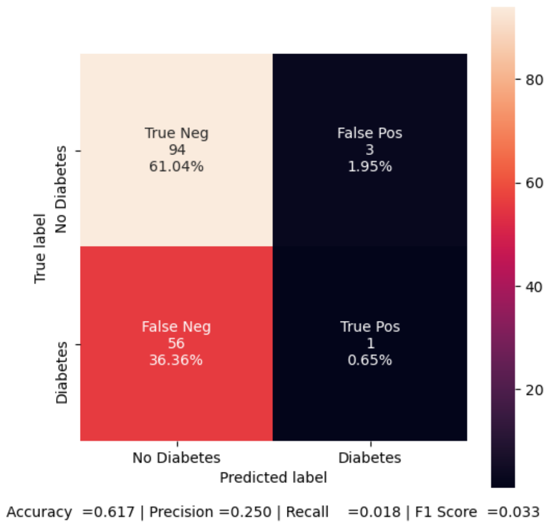


(d) 4:5-Fold

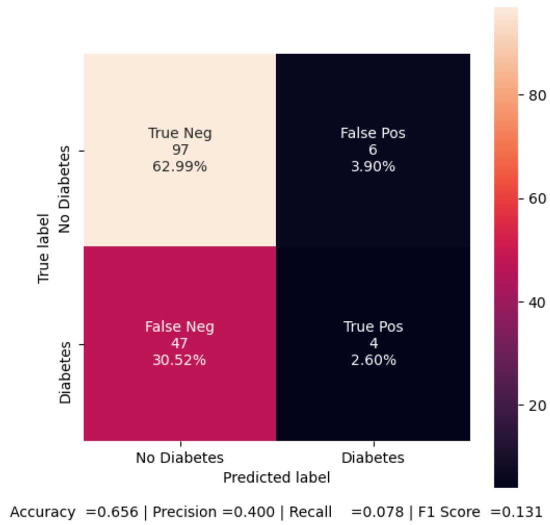


(e) 5:5-Fold

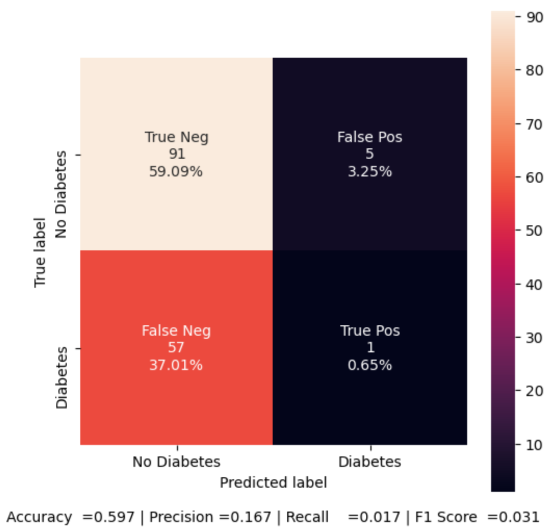
Figure 1-3. Confusion Matrices for C:0.1 K:rbf 5-fold



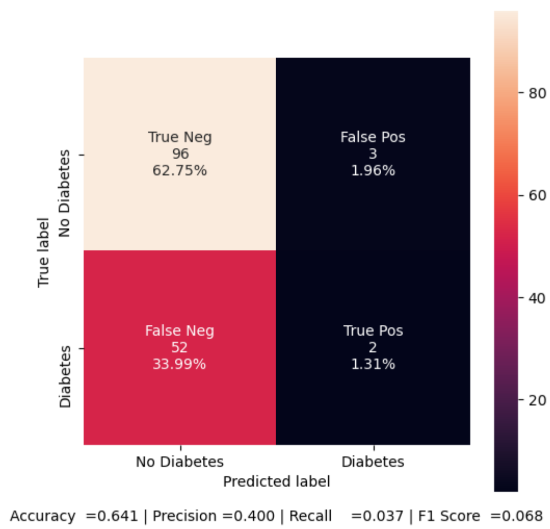
(a) 1:5-Fold



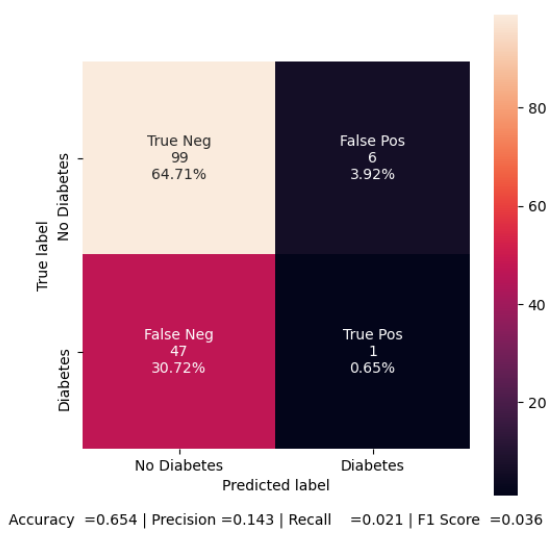
(b) 2:5-Fold



(c) 3:5-Fold

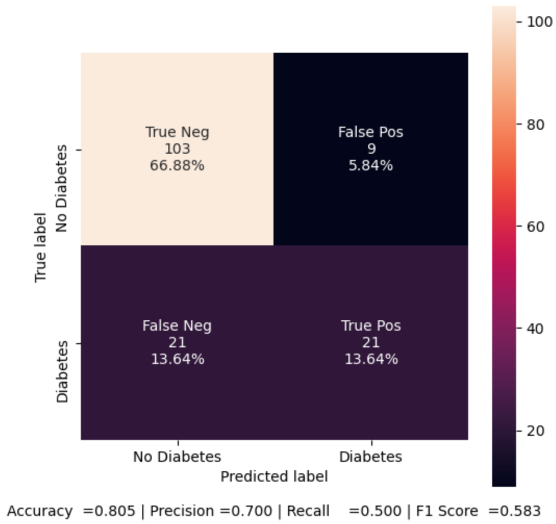


(d) 4:5-Fold

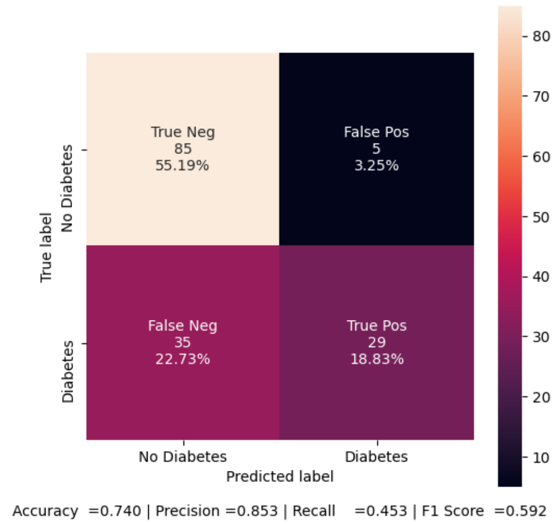


(e) 5:5-Fold

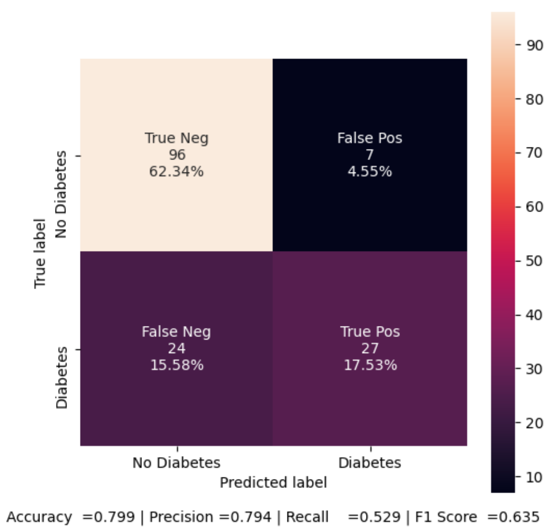
Figure 1-4. Confusion Matrices for C:0.1 K:sigmoid 5-fold



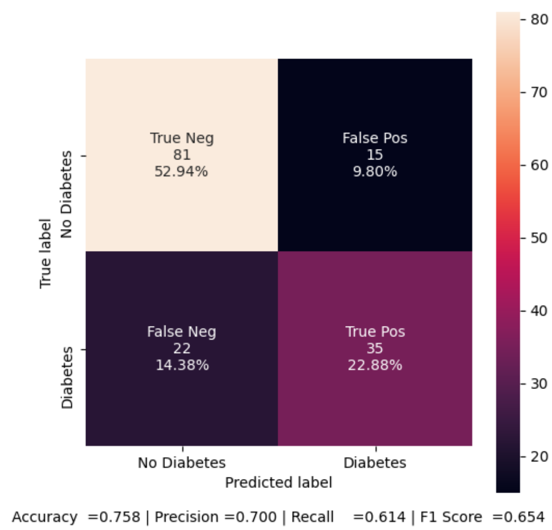
(a) 1:5-Fold



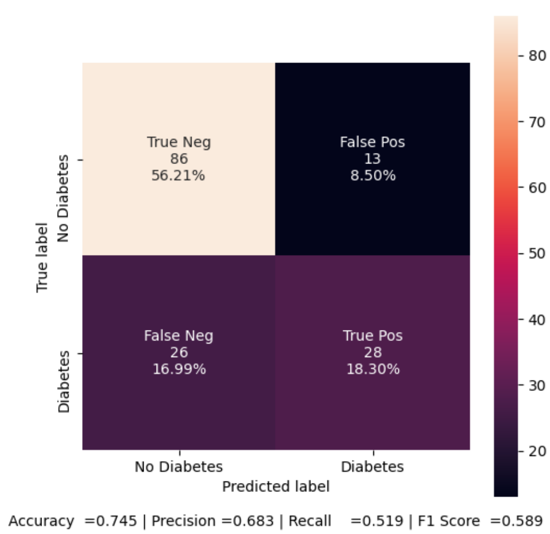
(b) 2:5-Fold



(c) 3:5-Fold

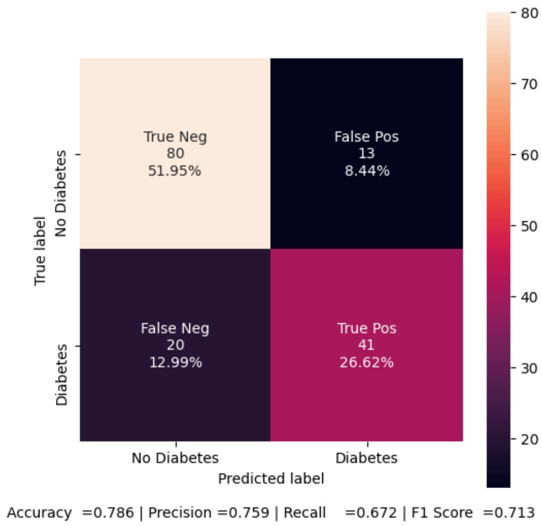


(d) 4:5-Fold

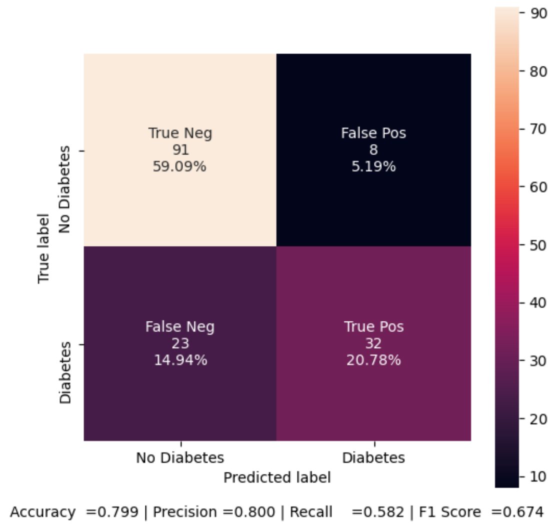


(e) 5:5-Fold

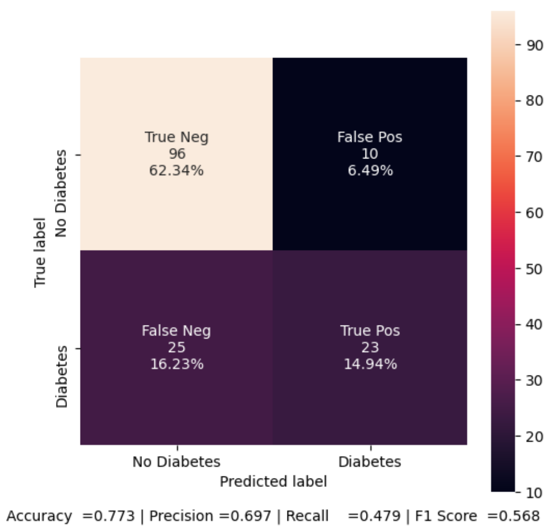
Figure 1-5. Confusion Matrices for C:1 K:linear 5-fold



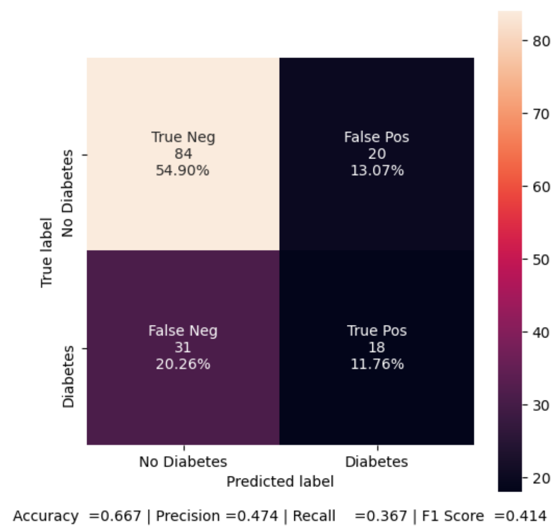
(a) 1:5-Fold



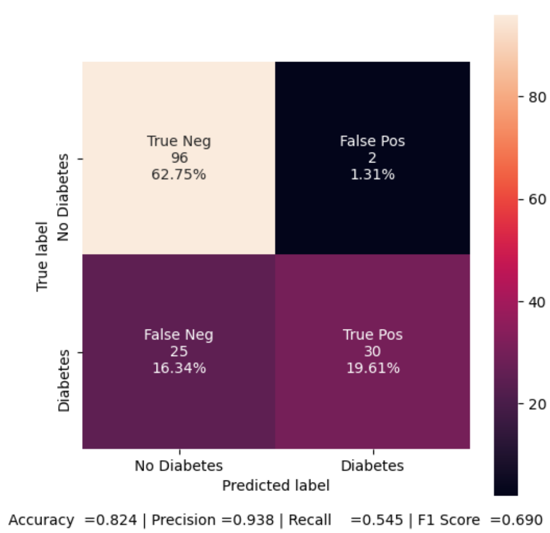
(b) 2:5-Fold



(c) 3:5-Fold

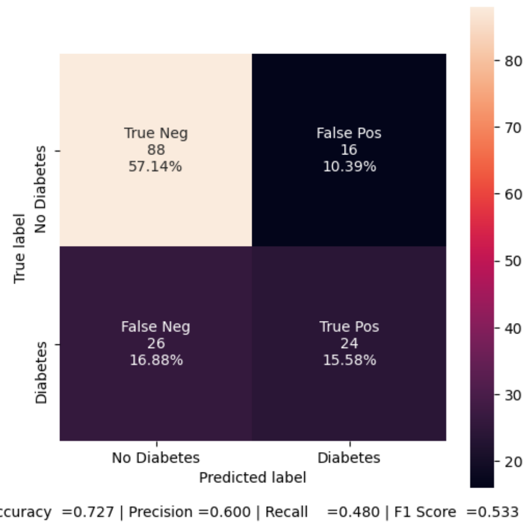


(d) 4:5-Fold

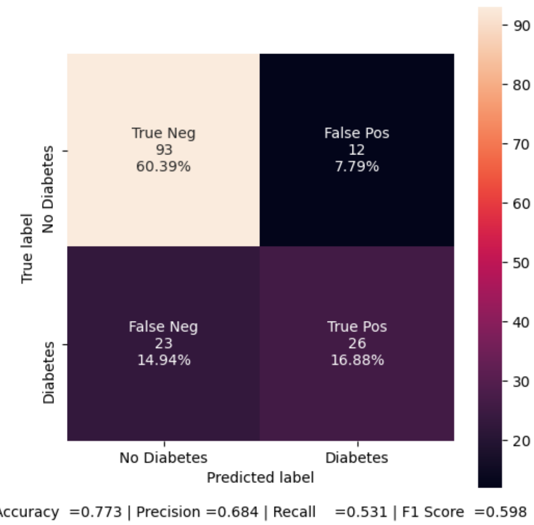


(e) 5:5-Fold

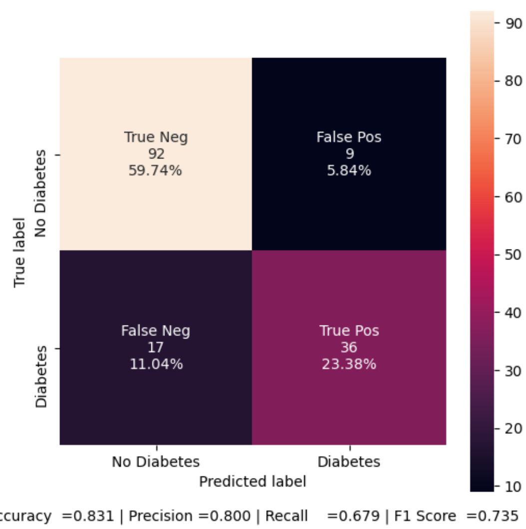
Figure 1-6. Confusion Matrices for C:1 K:poly 5-fold



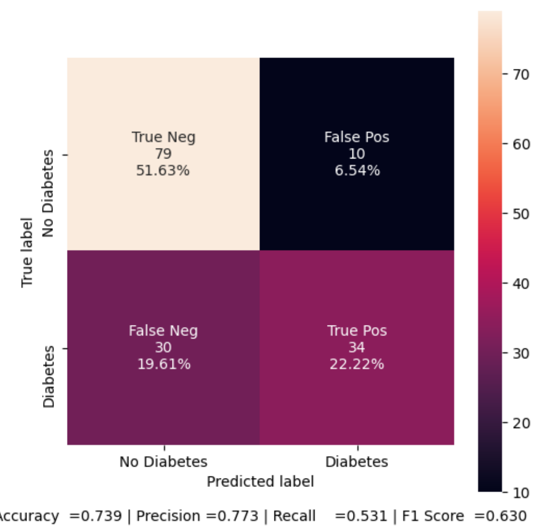
(a) 1:5-Fold



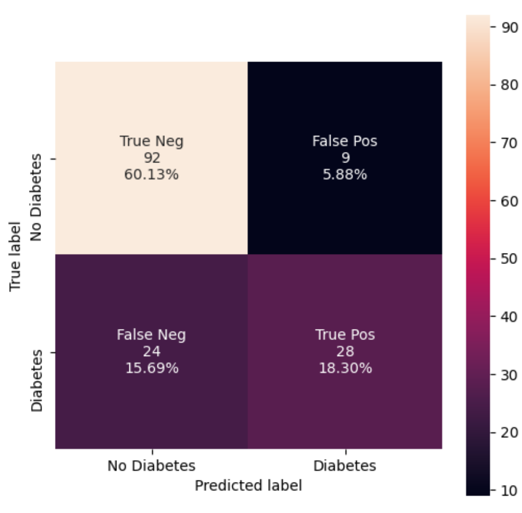
(b) 2:5-Fold



(c) 3:5-Fold

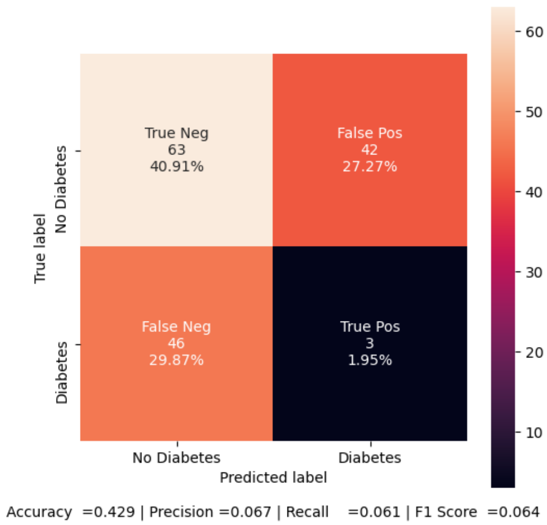


(d) 4:5-Fold

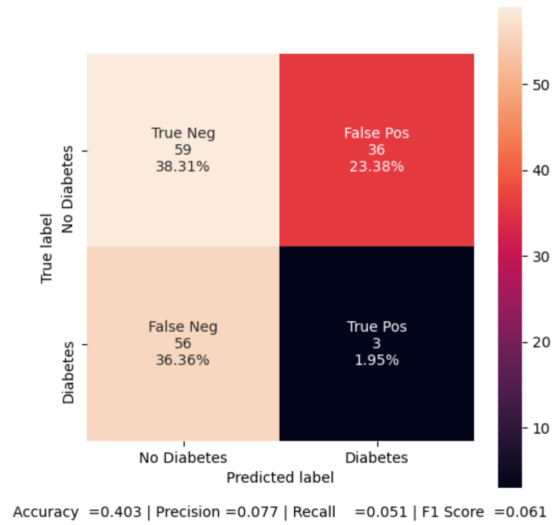


(e) 5:5-Fold

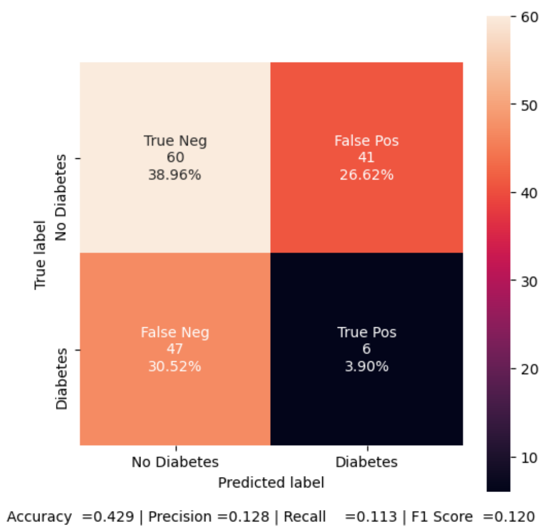
Figure 1-7. Confusion Matrices for C:1 K:rbf 5-fold



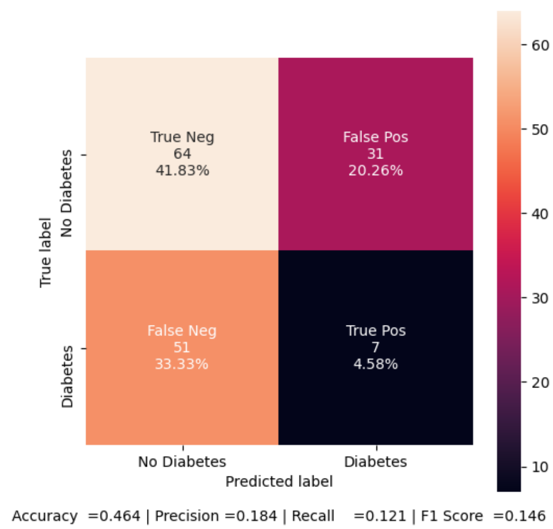
(a) 1:5-Fold



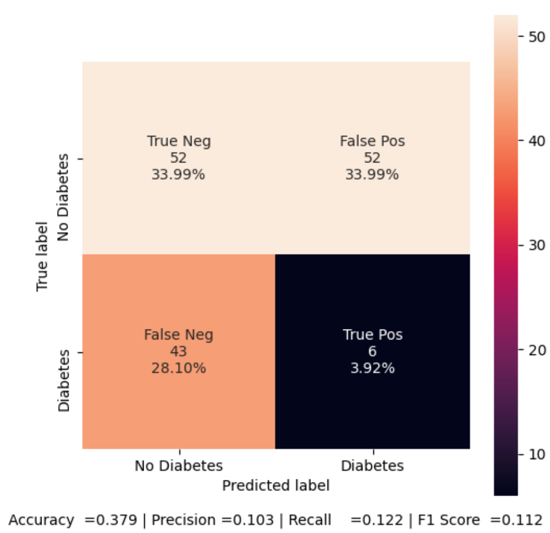
(b) 2:5-Fold



(c) 3:5-Fold

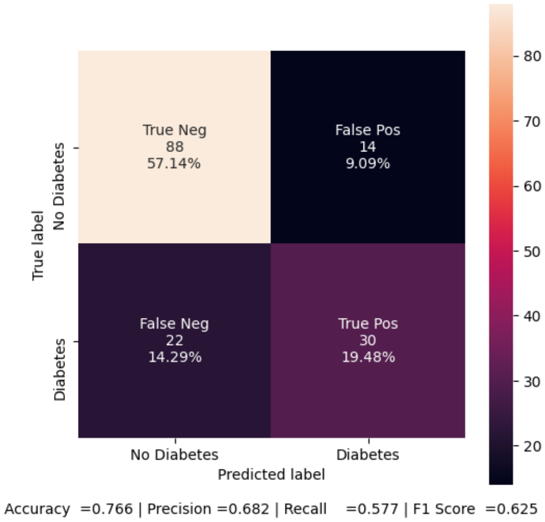


(d) 4:5-Fold

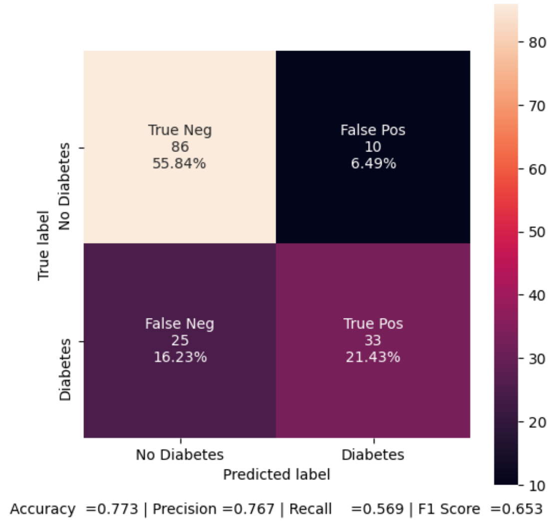


(e) 5:5-Fold

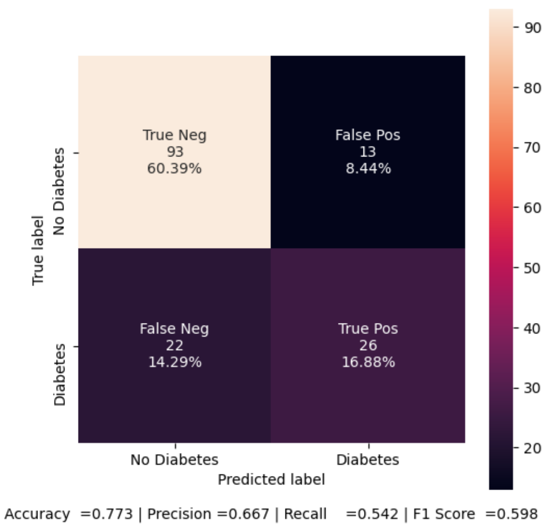
Figure 1-8. Confusion Matrices for C:1 K:sigmoid 5-fold



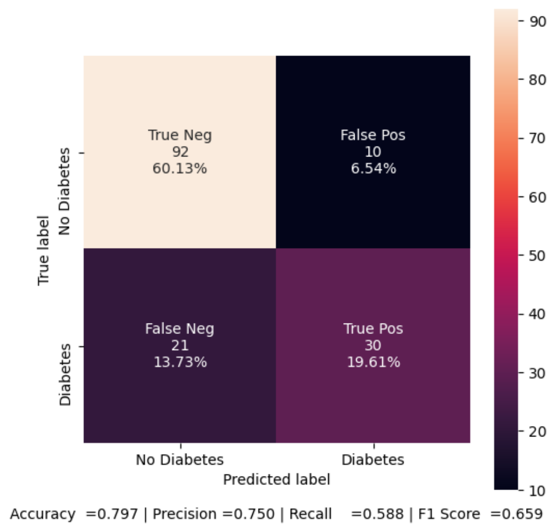
(a) 1:5-Fold



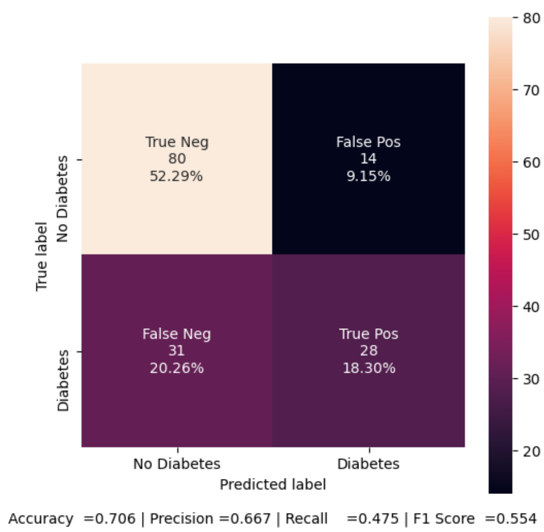
(b) 2:5-Fold



(c) 3:5-Fold

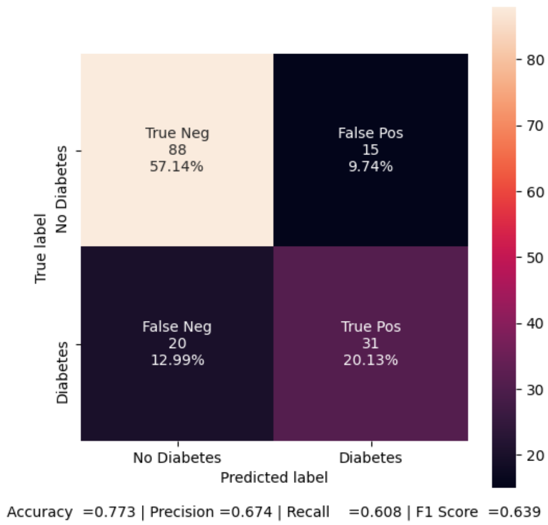


(d) 4:5-Fold

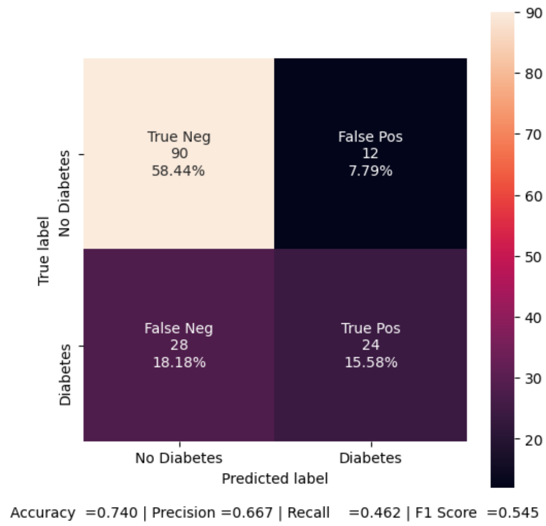


(e) 5:5-Fold

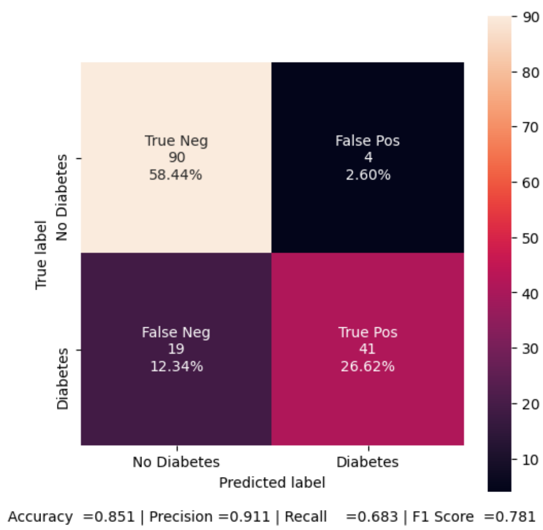
Figure 1-9. Confusion Matrices for C:5 K:linear 5-fold



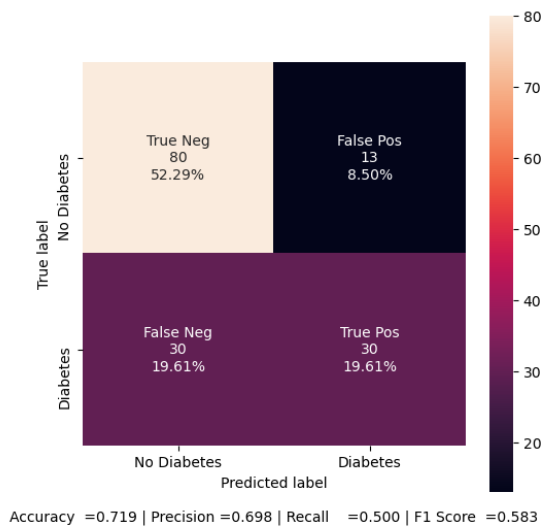
(a) 1:5-Fold



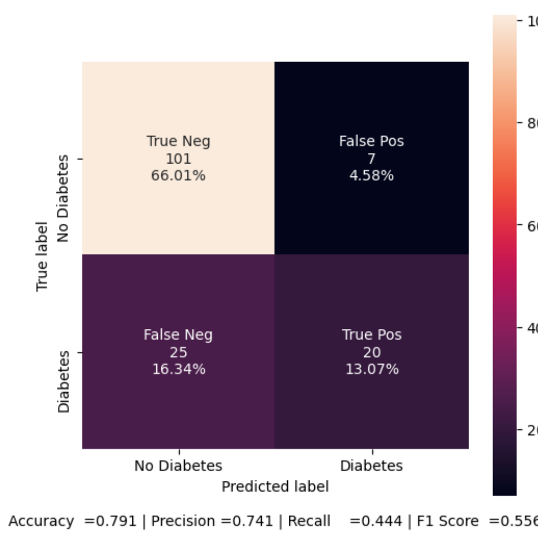
(b) 2:5-Fold



(c) 3:5-Fold

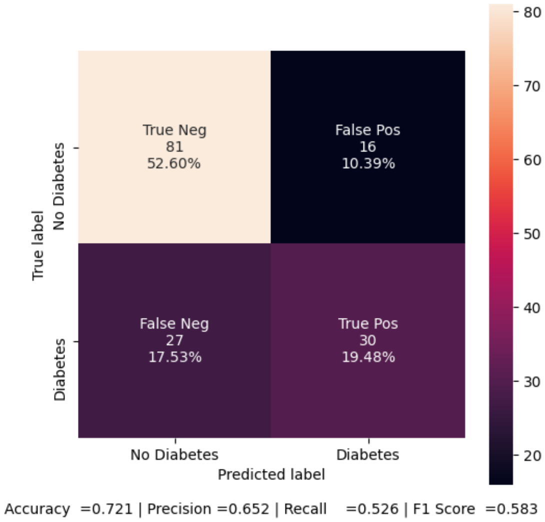


(d) 4:5-Fold

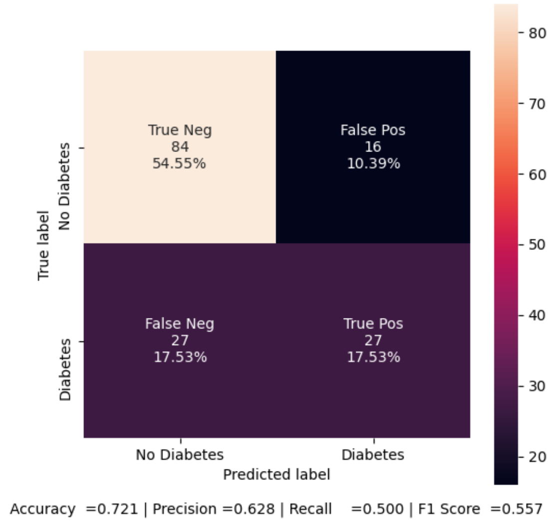


(e) 5:5-Fold

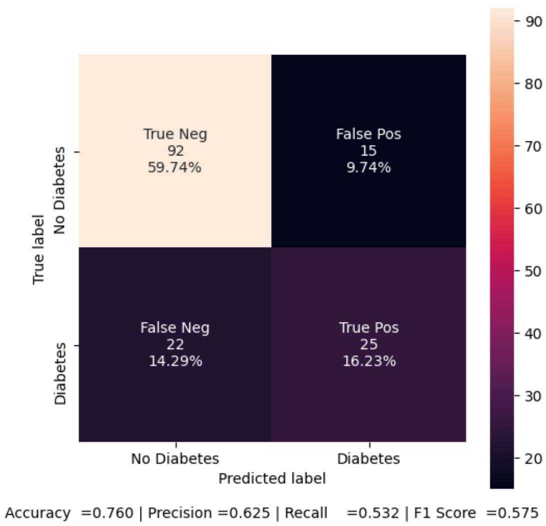
Figure 1-10. Confusion Matrices for C:5 K:poly 5-fold



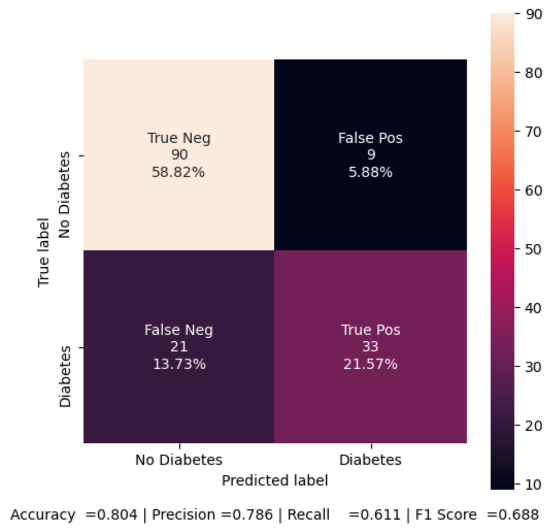
(a) 1:5-Fold



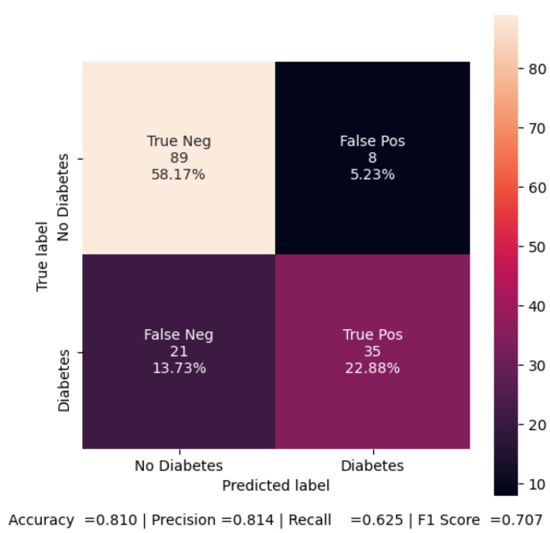
(b) 2:5-Fold



(c) 3:5-Fold

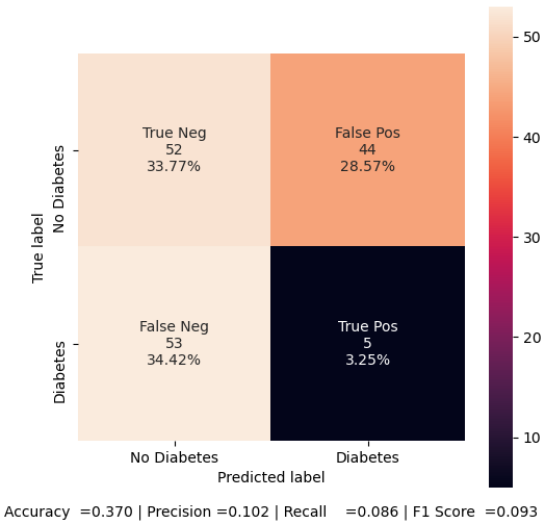


(d) 4:5-Fold

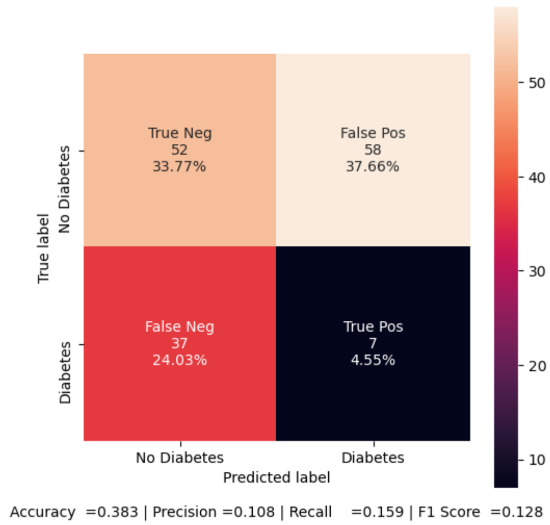


(e) 5:5-Fold

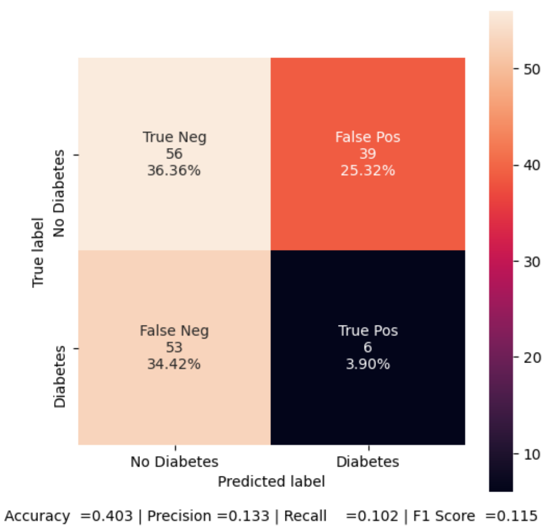
Figure 1-11. Confusion Matrices for C:5 K:rbf 5-fold



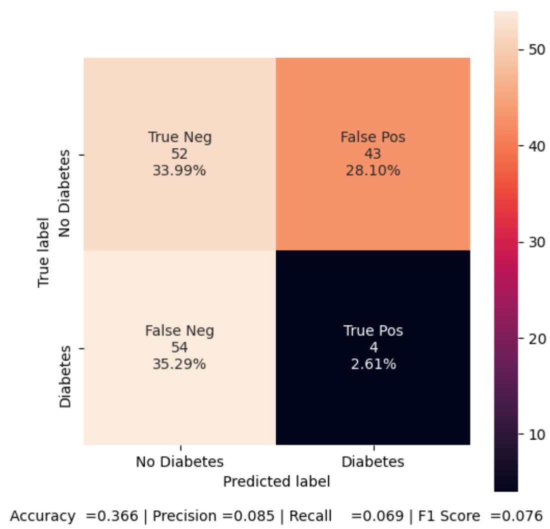
(a) 1:5-Fold



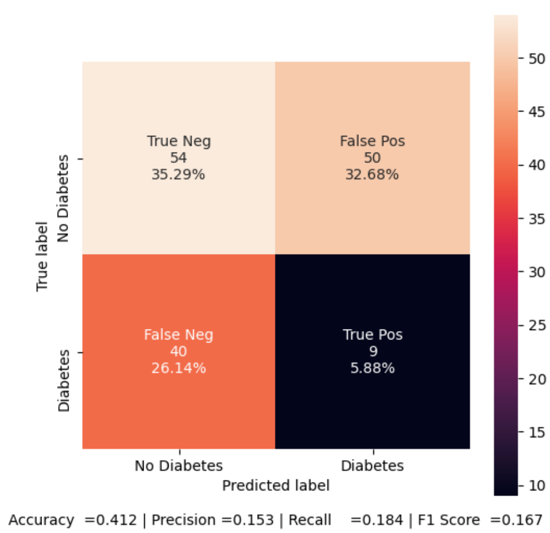
(b) 2:5-Fold



(c) 3:5-Fold

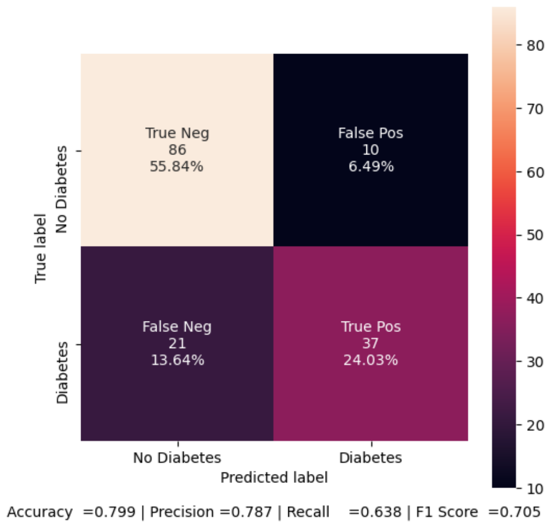


(d) 4:5-Fold

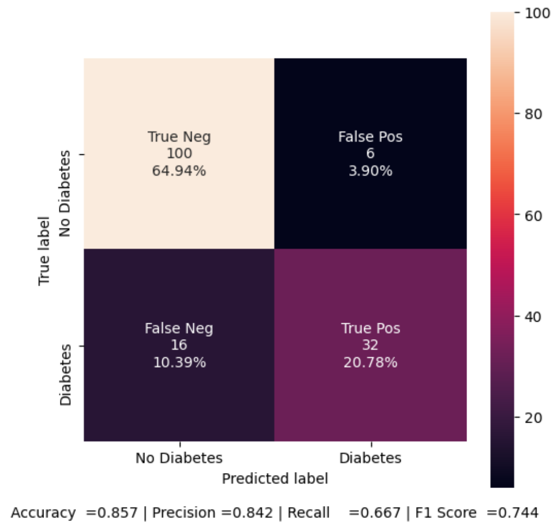


(e) 5:5-Fold

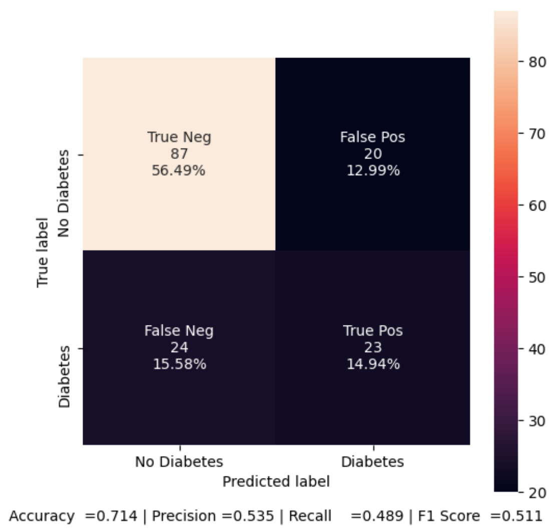
Figure 1-12. Confusion Matrices for C:5 K:sigmoid 5-fold



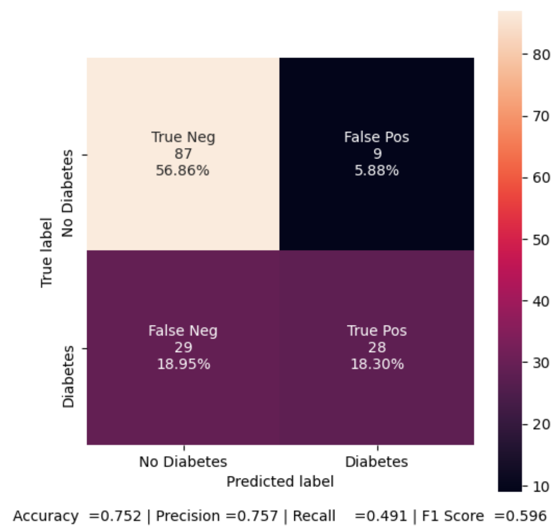
(a) 1:5-Fold



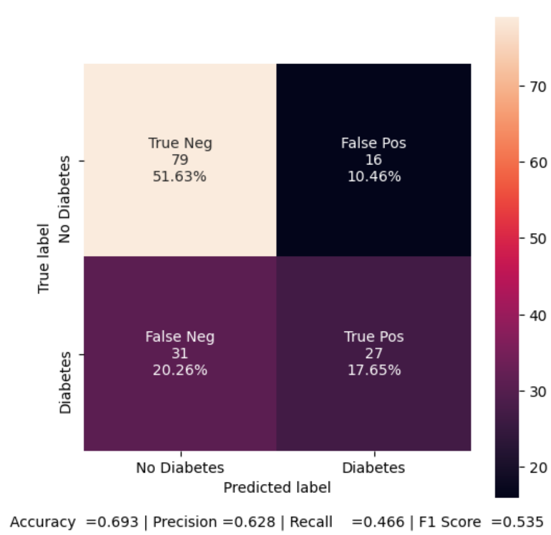
(b) 2:5-Fold



(c) 3:5-Fold

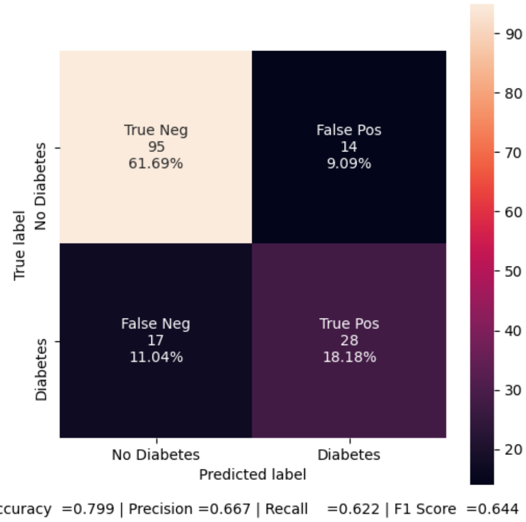


(d) 4:5-Fold

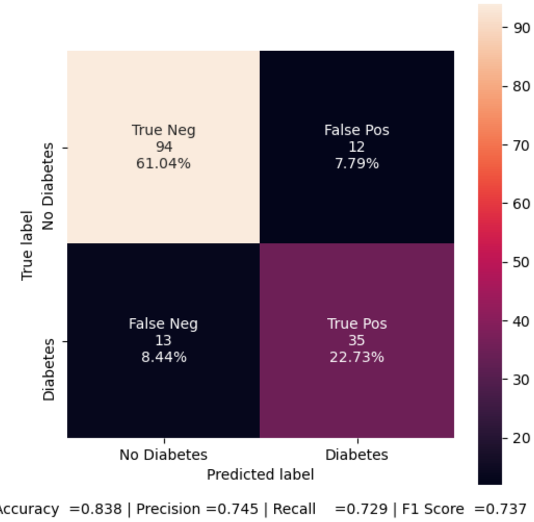


(e) 5:5-Fold

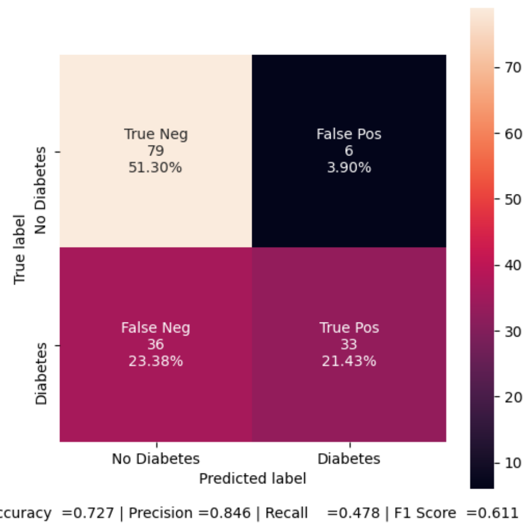
Figure 1-13. Confusion Matrices for C:10 K:linear 5-fold



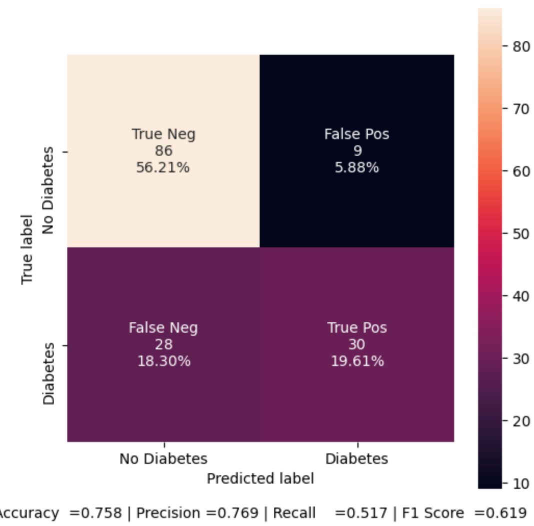
(a) 1:5-Fold



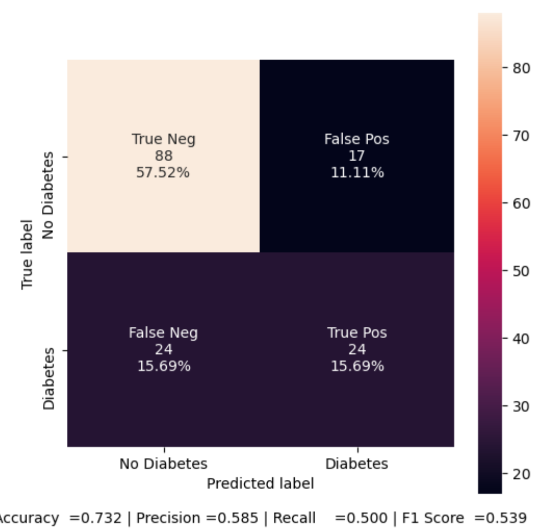
(b) 2:5-Fold



(c) 3:5-Fold

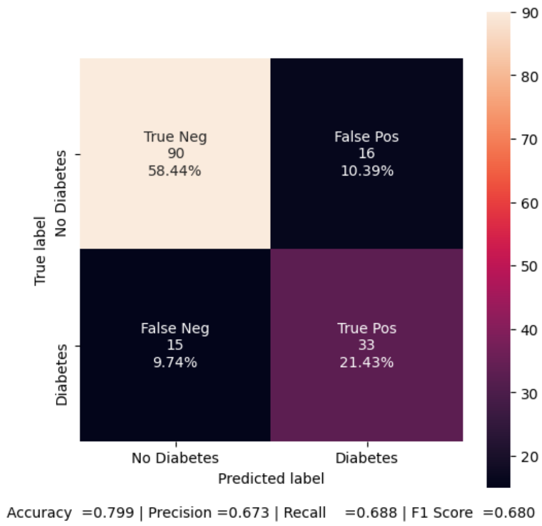


(d) 4:5-Fold

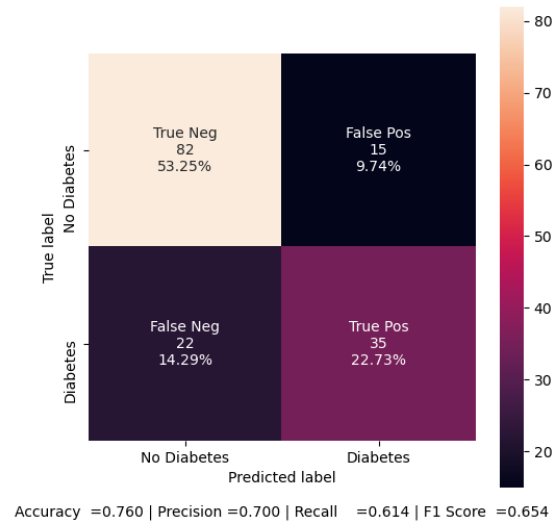


(e) 5:5-Fold

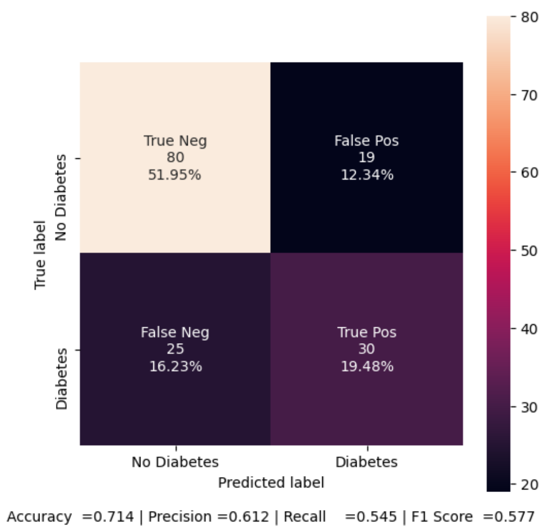
Figure 1-14. Confusion Matrices for C:10 K:poly 5-fold



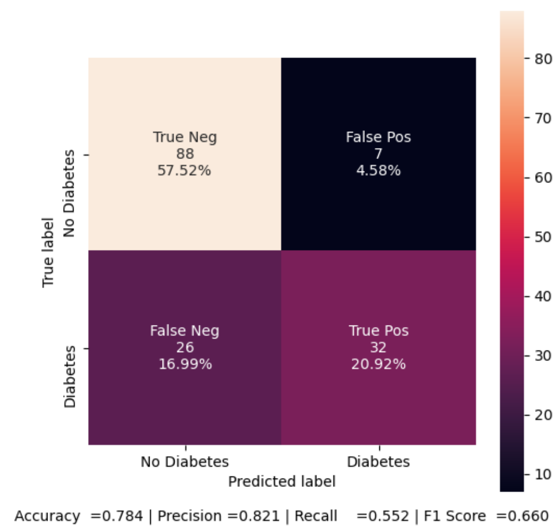
(a) 1:5-Fold



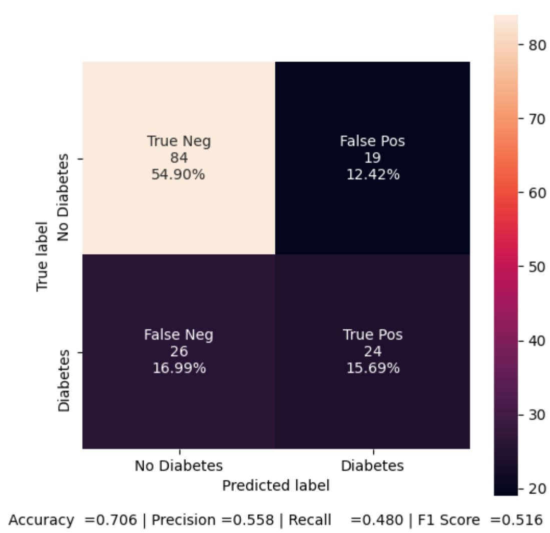
(b) 2:5-Fold



(c) 3:5-Fold

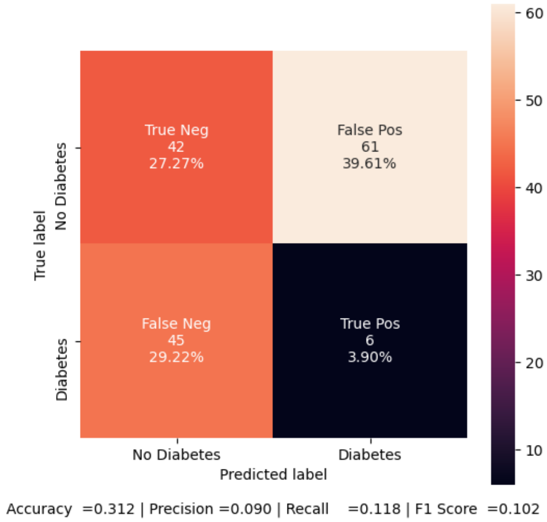


(d) 4:5-Fold

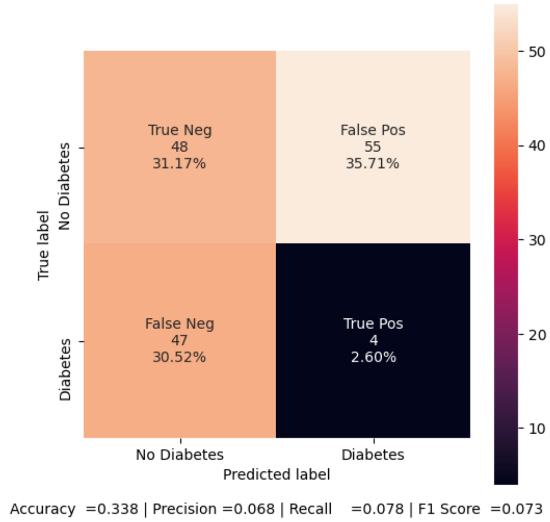


(e) 5:5-Fold

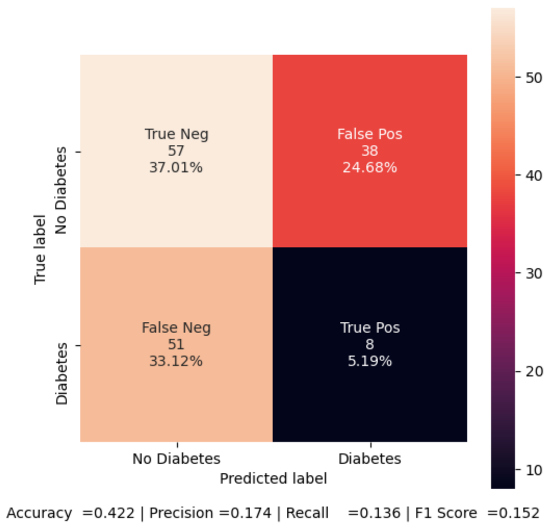
Figure 1-15. Confusion Matrices for C:10 K:rbf 5-fold



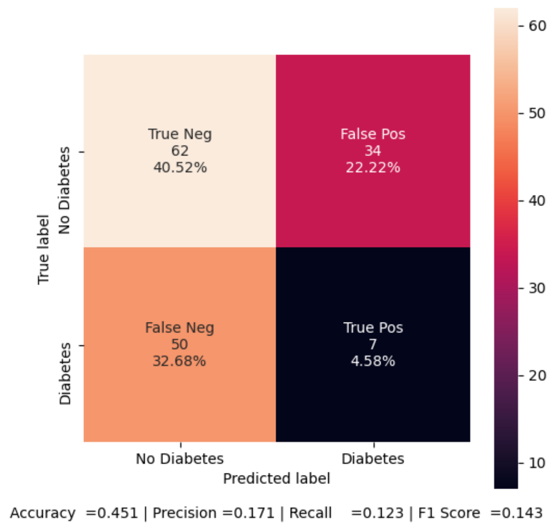
(a) 1:5-Fold



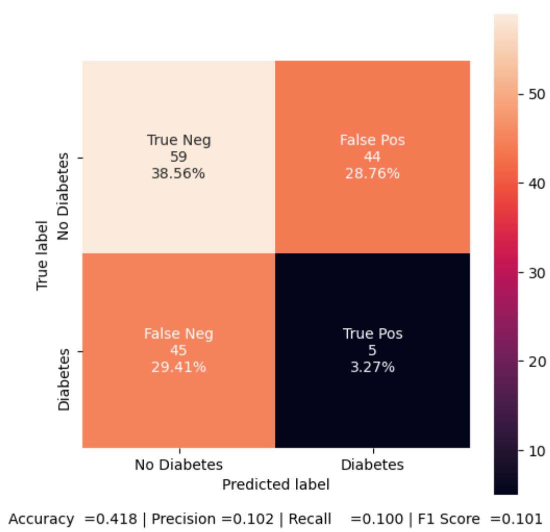
(b) 2:5-Fold



(c) 3:5-Fold



(d) 4:5-Fold



(e) 5:5-Fold

Figure 1-16. Confusion Matrices for C:10 K:sigmoid 5-fold

1.2 (b): Various performance measure (15 marks)

The performance measure is calculated along with confusion matrices (full implementation can be found in Code 5):

```
        #Accuracy is sum of diagonal divided by total observations
        status["Accuracy" ]  = np.trace(cf) / float(np.sum(cf))

        #if it is a binary confusion matrix, show some more stats
        if len(cf)==2:
            #Metrics for Binary Confusion Matrices
            status["Precision"] = cf[1,1] / sum(cf[:,1])
            status["Recall"    ] = cf[1,1] / sum(cf[1,:])
            status["F1 Score" ] = 2 * status["Precision"] * status["Recall"] / (status["Precision"] +
    status["Recall"])
```

Code 3: SVM Performance Measure

The average, best, and worst performance measures out of 5-fold cross-validation in terms of Accuracy, Precision, Recall, and F1 Measure are summarized in Figure 1-17, Figure 1-18, and Figure 1-19 respectively.

The recall measure is the most important in this problem, since we want to eliminate percent of false negative to ensure it is not missing any people who are indeed having diabetes. In another word, we need to maximizing percent of true positive over sum of true positive and false positive, which is the recall performance measure. In addition, the worst performance measure of the recall out of 5-Fold cross-validation shall be also considered when choosing the model, since we would always ensure the worst possible performance of the model.

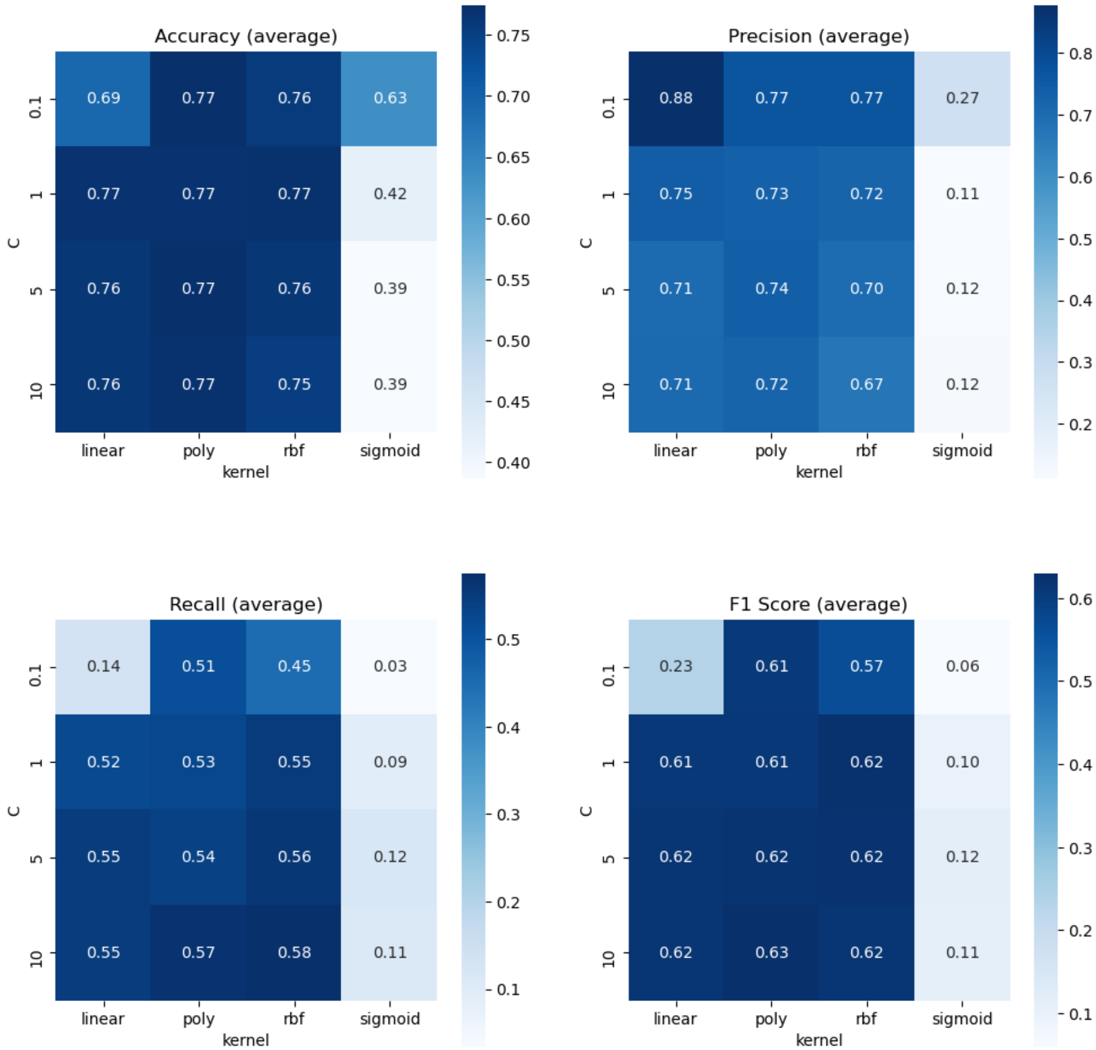


Figure 1-17. Average performance grid matrices for all 16 combinations

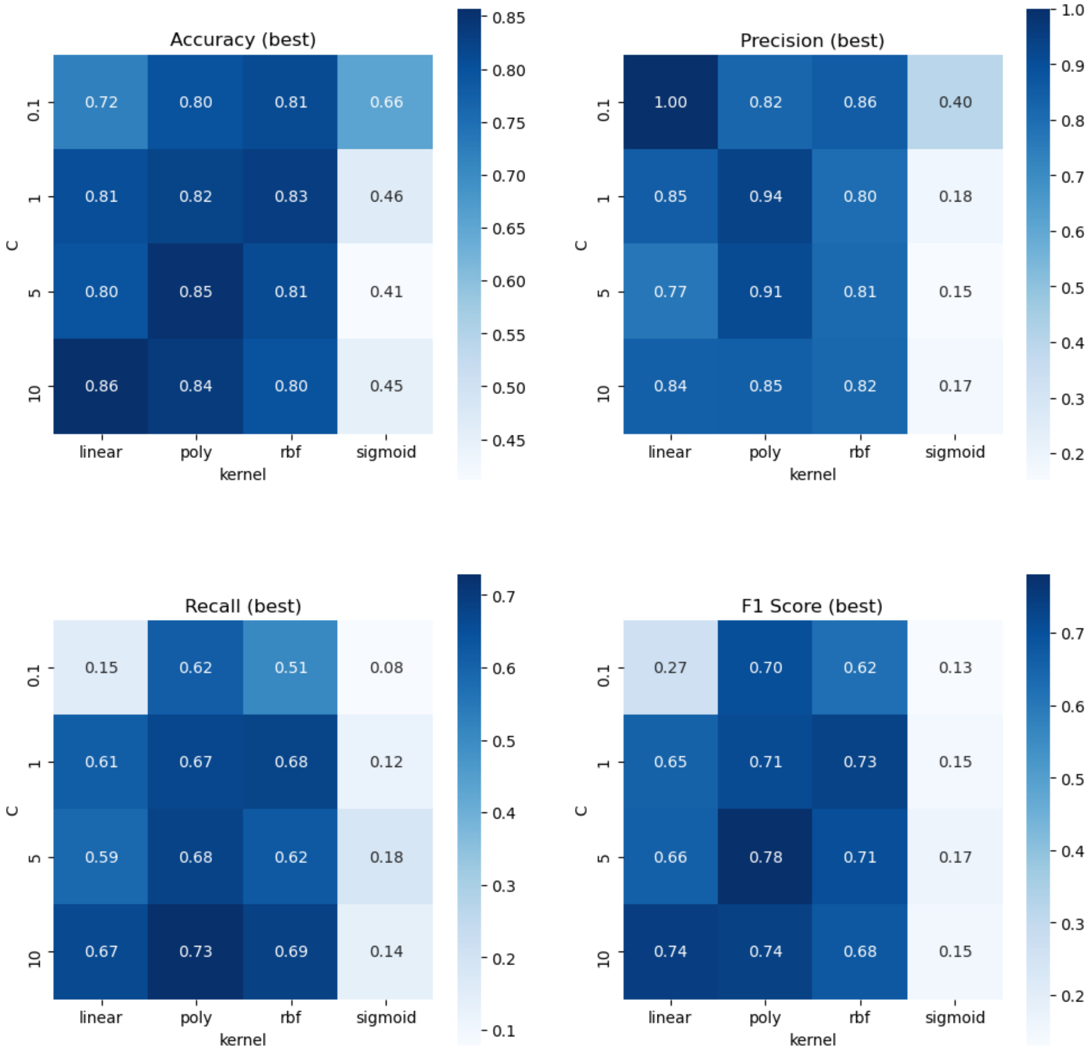


Figure 1-18. Best performance grid matrices for all 16 combinations

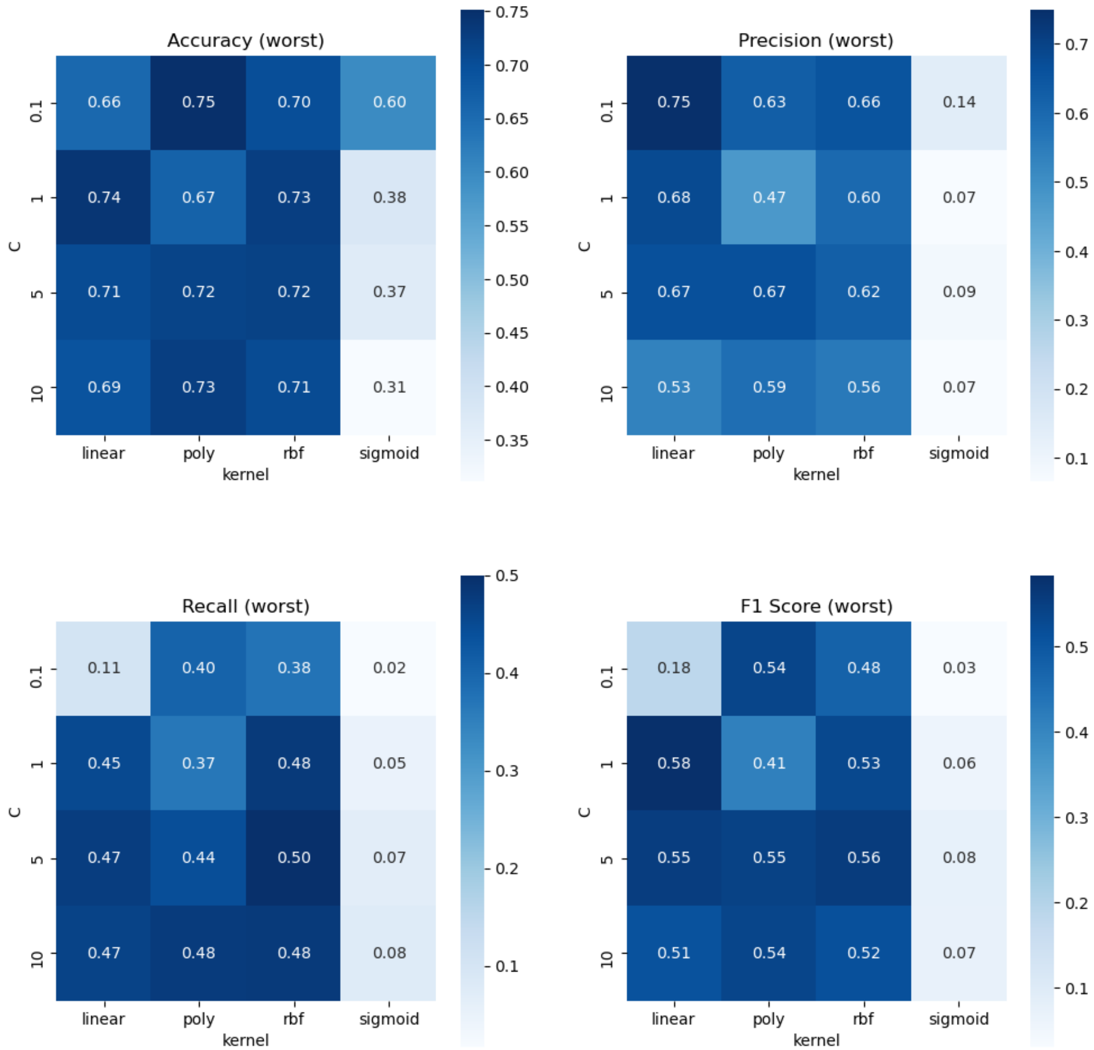


Figure 1-19. Worst performance grid matrices for all 16 combinations

2 Problem 2: Kohonen Self Organizng Map: Unsupervised Learning [Code 7]

2.1 Implementation (Matrix Formulation and Optimization)

In order to optimize the performance and utilize the gpu performance, the update function is modelled and implemented in matrix forms, and any constant is being pre-computed in each iteration, as shown in Code 4.

```python
            # Alpha(k) & s(k):
            alpha_k = self.alpha_0 * np.exp(- k / T)
            s_k = sigma_0 * np.exp(- k / T)
            s_k_2_2_division = 1 / (2 * (s_k ** 2)) # pre-optimization
            # w_ij:
            for x in self.training_data:
                # calculate performance index
                diff = np.linalg.norm(x - self.w, axis = 2)
                # find index of winning node
                ind = np.unravel_index(np.argmin(diff, axis=None), diff.shape) # y,x
                # Update weights for neighbourhood
                xx = np.arange(0, N, 1)
                yy = np.arange(0, N, 1)

                ### matrix form (optimization):
                Mj = np.meshgrid(xx, yy)
                Dx = (Mj[0] - ind[1]) ** 2
                Dy = (Mj[1] - ind[0]) ** 2
                Dij2 = Dx+Dy
                Nij = np.exp(- Dij2 * s_k_2_2_division )
                dxw = np.subtract(x, self.w)
                Nw = np.stack([Nij, Nij, Nij], axis=2) # depth stacking
                self.w = self.w + alpha_k * np.multiply(Nw, dxw)
```

Code 4: KSOM Core Update in Matrix Formulation

The final outcome is outstanding, where it only takes 37.8 seconds to compute, in comparison to the double for loop form, which takes minutes and even hours to compute.

```
[Running] python -u "/Users/jaku/JX-Platform/Github/UW__4B_Individual_Works/ECE_457B/A2/src_code/as2_p2.
    py"
ic| normRGB.shape: (24, 3)
Epoch Number: 1
Epoch Number: 20
Epoch Number: 40
Epoch Number: 100
Epoch Number: 600
Epoch Number: 1
Epoch Number: 20
Epoch Number: 40
Epoch Number: 100
Epoch Number: 600
Epoch Number: 1
Epoch Number: 20
Epoch Number: 40
Epoch Number: 100
Epoch Number: 600

[Done] exited with code=0 in 37.838 seconds
```

2.2 (a): SOM Grids Result (25 marks)

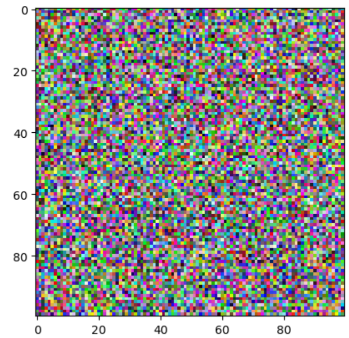


Figure 2-1. Original SOM grid (random colors)

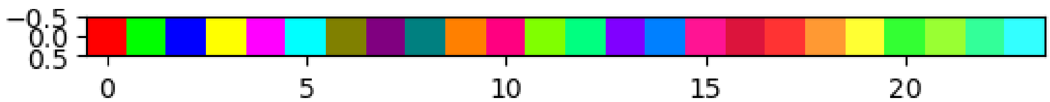
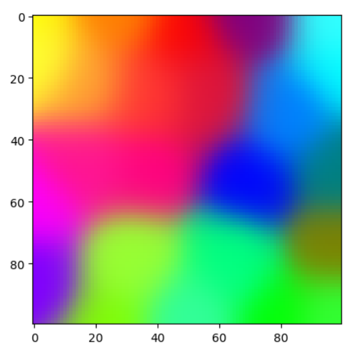
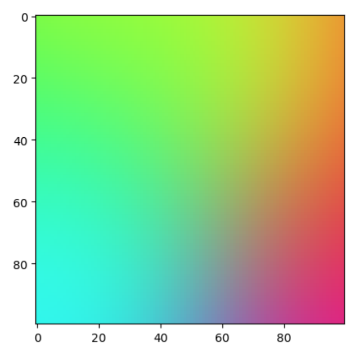
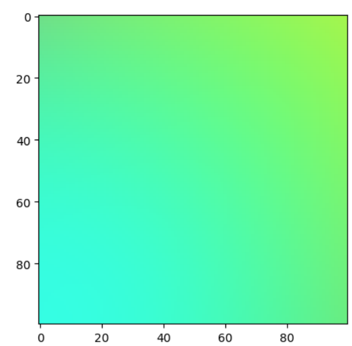
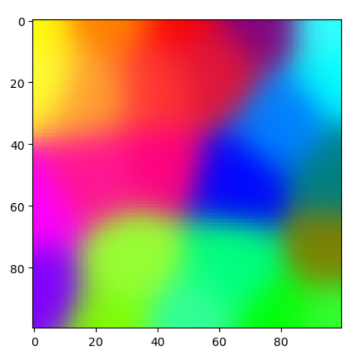
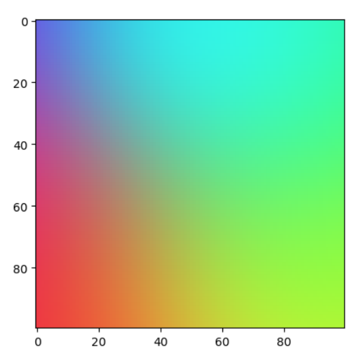
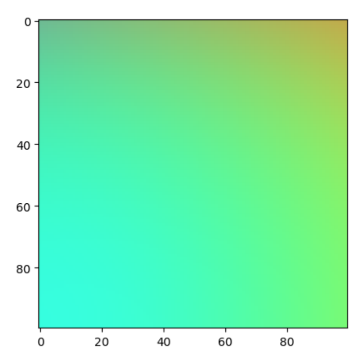


Figure 2-2. 24 randomly selected colors (from HSV wheel to RGB)

The resultant SOM grids are as stated in Table 2-1 below:

	$\sigma_0 = 10$	$\sigma_0 = 40$	$\sigma_0 = 70$
Epochs = 20			
Epochs = 40			

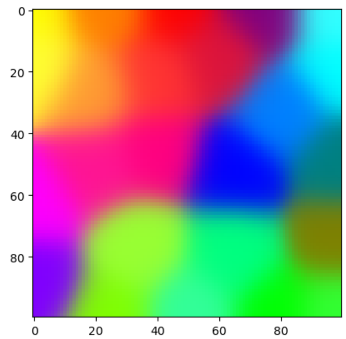
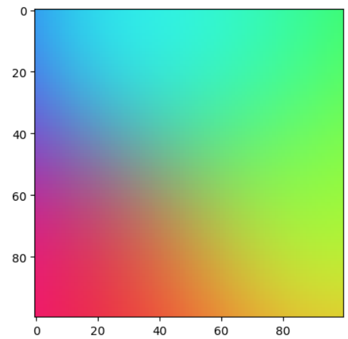
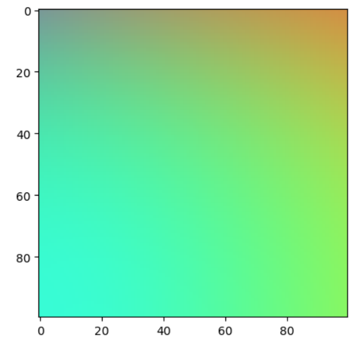
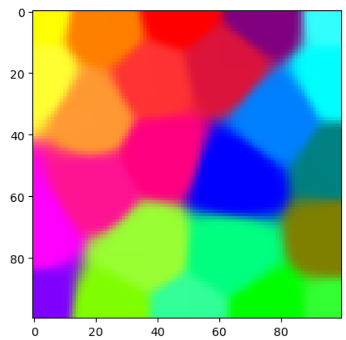
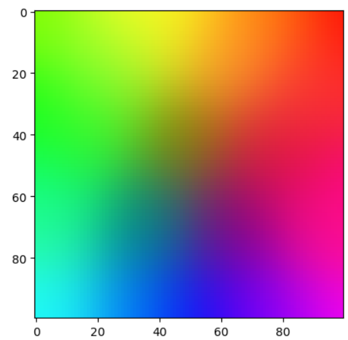
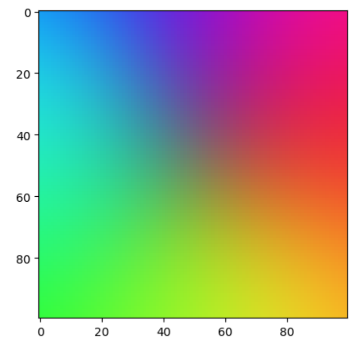
Epochs = 100			
Epochs = 600			

Table 2-1. SOM grid summary table (T: 600 epochs)

2.3 (b): Conclusions (5 marks)

The above implementation utilizes an adaptive neighboring, as the nrighborhood variance decreases overtime, resulting a smaller neighbor size. This is done to help neurons initially adjust their weights to roughly where they want to be then allow them to converge without being dramatically influenced by ”winning” neurons that are far away. In another world, the early stage is the self-organizing or re-ordering phase, and the later stage is fine tuning phase to make the grid coverge.

From the above simulation, we may find the SOM is capable to cluster progressively based on the given training dataset without any supervision. The initial variance of the topological neighborhood size (σ_0) affects the convergence rate of the SOM grid. A smaller initial variance converges slower (or stable equilibrium), but provides a much finer SOM grid with sharp boundaries, as shown by the column for $\sigma_0 = 10$. A larger variance converges much faster, but provides a coarser SOM grid with blended margins, as shown by the column for $\sigma_0 = 70$. We may find the medium variance of $\sigma_0 = 40$ between the two extremes provided the best outcome of the SOM grid, as it quickly reaches equilibrium initially, and provides a finer final SOM grid without loosing too much color resolution.

In short, the magical SOM performance depends on the initial neighbouring size for the adaptive neighborhood function (Gaussian specifically in our case). A large variance encourages a faster stability but uneven result. A smaller variance may cause the map incapable to reach stability quickly enough resulting an incomplete SOM grid, where partial regions of the SOM map is barely utilized.

3 Problem 3: MLP vs Deep Learning Based CNN [Code 8]

3.1 (a): Result and comment (training and testing accuracy) (15 marks)

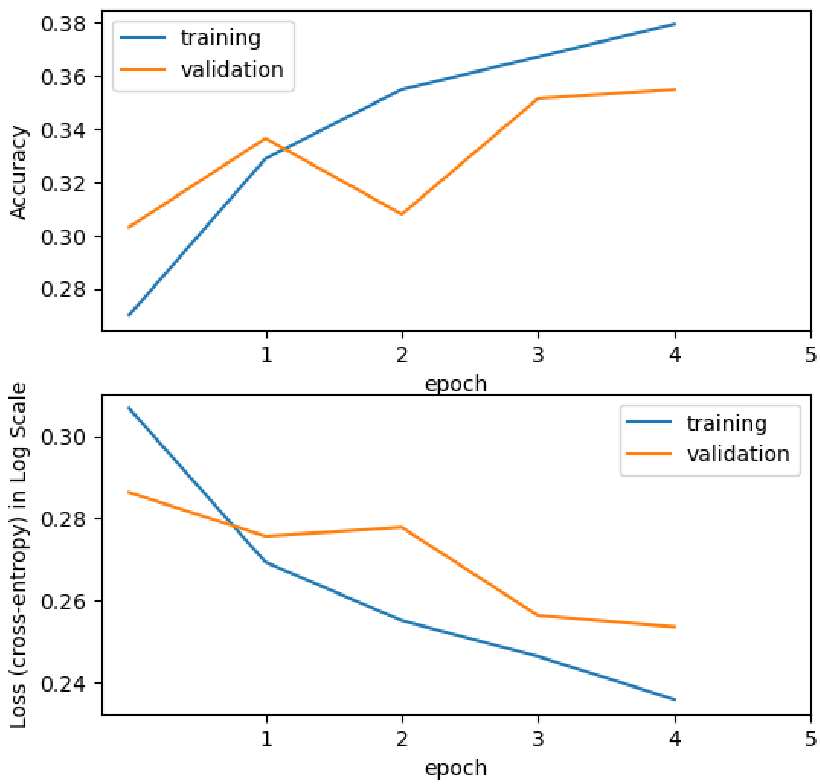


Figure 3-1. MLP Progress

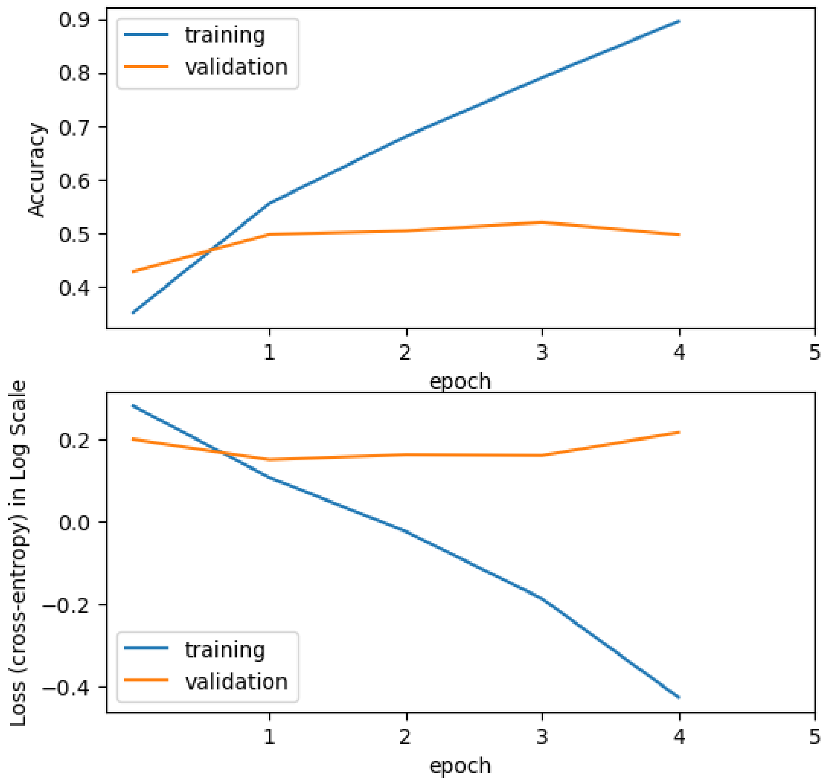


Figure 3-2. CNN-1 Progress

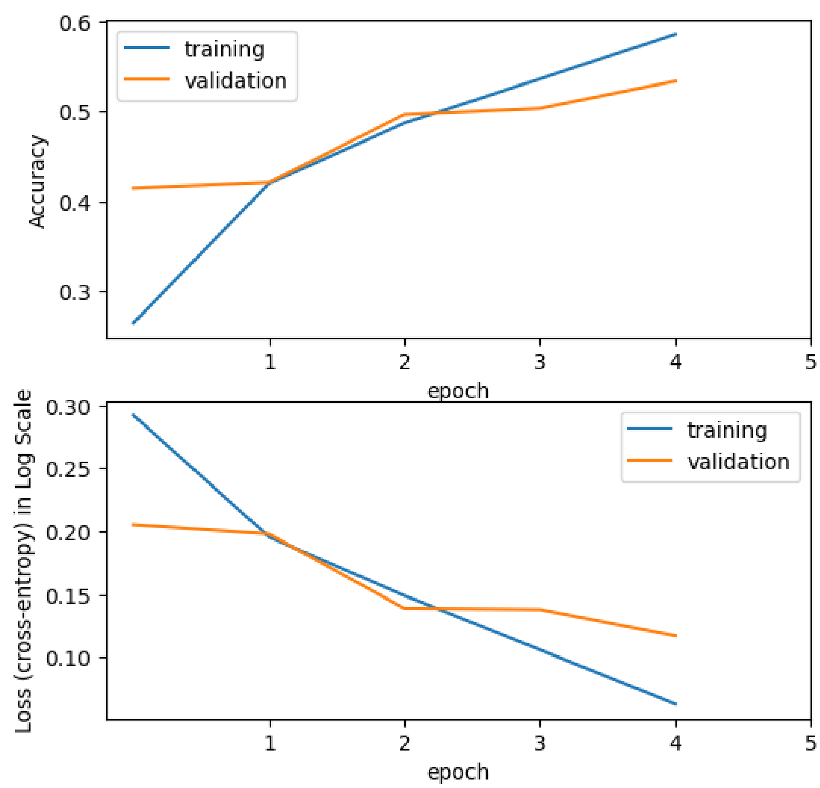
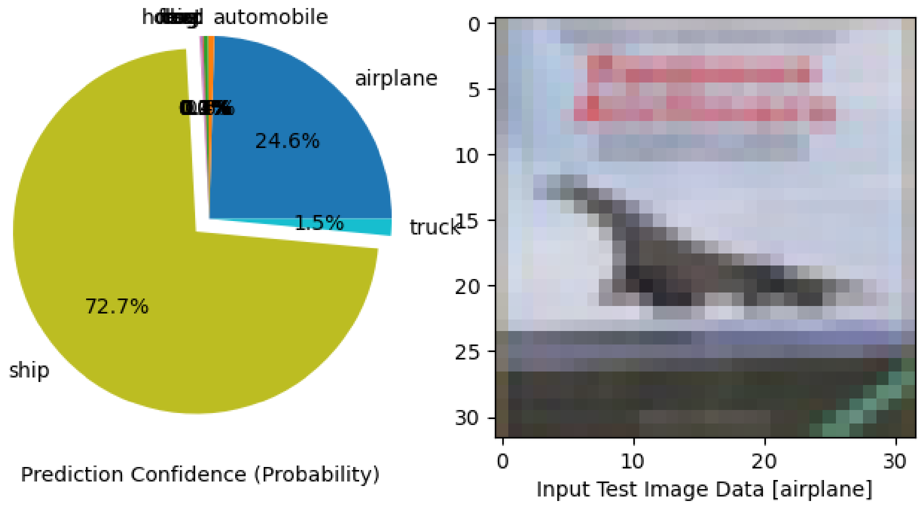
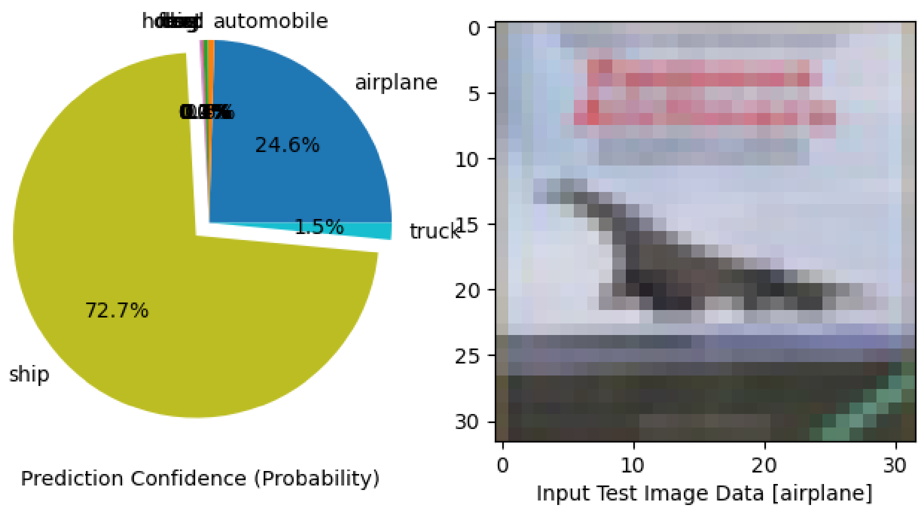


Figure 3-3. CNN-2 Progress

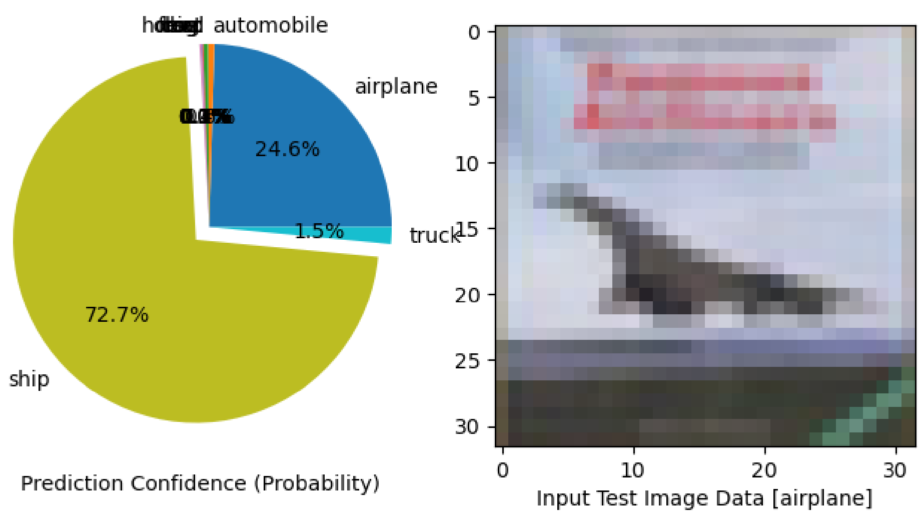
3.2 (b): Plot training and validation curves (15 marks)



(a) MLP

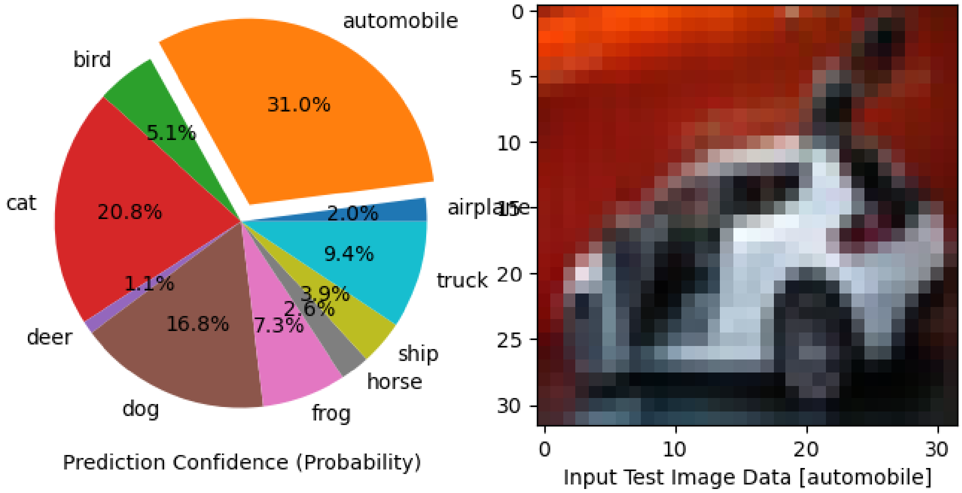


(b) CNN-1

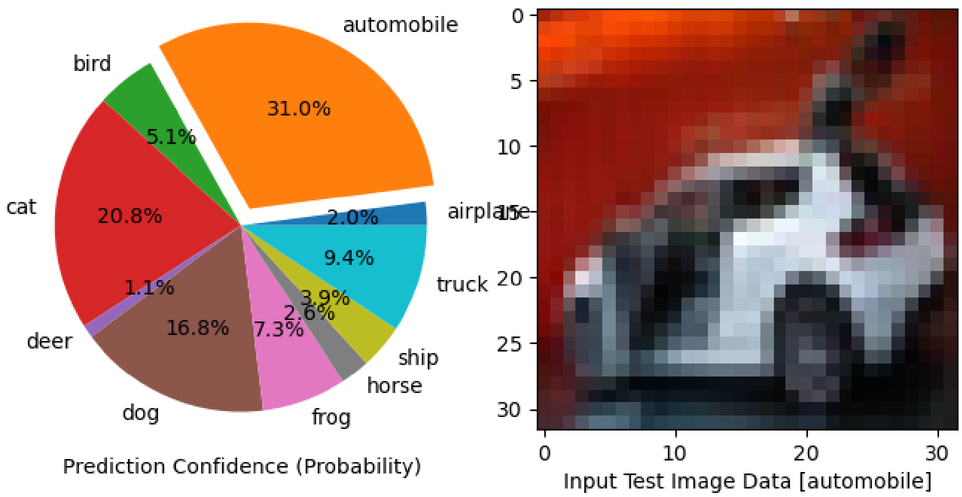


(c) CNN-2

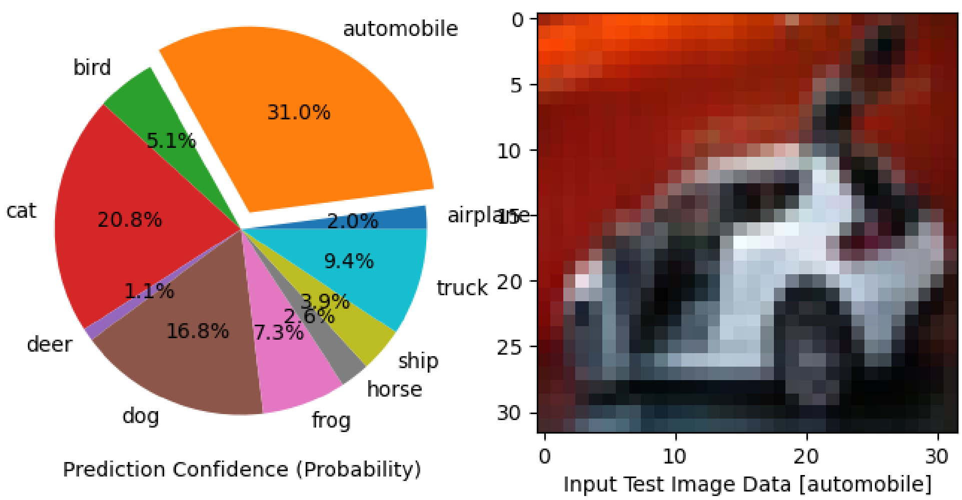
Figure 3-4. Testing Sample



(a) MLP



(b) CNN-1



(c) CNN-2

Figure 3-5. Testing Sample

```
# jaku @ Jacks-MacBook-Pro in ~/JX-Platform/Github/UW__4B_Individual_Works/ECE_457B/A2/src_code on git:
    main x [20:24:23]
$ python as2_p3.py
2021-03-10 20:29:03.468760: I tensorflow/compiler/jit/xla_cpu_device.cc:41] Not creating XLA devices,
    tf_xla_enable_xla_devices not set
2021-03-10 20:29:03.469028: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary
    is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions
    in performance-critical operations: AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
ic| np.shape(y_train): (10000, 10)
ic| np.shape(X_train): (10000, 32, 32, 3)
ic| np.shape(y_test): (10000, 10)
ic| np.shape(X_test): (10000, 32, 32, 3)
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1573376
_________________________________________________________________
dense_1 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,578,506
Trainable params: 1,578,506
Non-trainable params: 0
_________________________________________________________________
ic| model.summary(): None
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 64)        1792
_________________________________________________________________
flatten_1 (Flatten)          (None, 57600)             0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               29491712
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 29,498,634
Trainable params: 29,498,634
Non-trainable params: 0
_________________________________________________________________
ic| model.summary(): None
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 30, 30, 64)        1792
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               1180160
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,486,666
Trainable params: 1,486,666
```

```
Non-trainable params: 0
_________________________________________________________________
ic| model.summary(): None
2021-03-10 20:29:06.892492: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the
    MLIR optimization passes are enabled (registered 2)
Epoch 1/5
313/313 [==============================] - 3s 9ms/step - loss: 2.3115 - accuracy: 0.2018 - val_loss:
    1.9089 - val_accuracy: 0.3081
Epoch 2/5
313/313 [==============================] - 2s 8ms/step - loss: 1.8969 - accuracy: 0.3178 - val_loss:
    1.8433 - val_accuracy: 0.3347
Epoch 3/5
313/313 [==============================] - 3s 8ms/step - loss: 1.8276 - accuracy: 0.3480 - val_loss:
    1.8220 - val_accuracy: 0.3614
Epoch 4/5
313/313 [==============================] - 3s 8ms/step - loss: 1.7912 - accuracy: 0.3663 - val_loss:
    1.9133 - val_accuracy: 0.3177
Epoch 5/5
313/313 [==============================] - 3s 8ms/step - loss: 1.7703 - accuracy: 0.3605 - val_loss:
    1.7566 - val_accuracy: 0.3657
Epoch 1/5
313/313 [==============================] - 68s 217ms/step - loss: 2.2876 - accuracy: 0.2726 - val_loss:
    1.5611 - val_accuracy: 0.4378
Epoch 2/5
313/313 [==============================] - 76s 242ms/step - loss: 1.2784 - accuracy: 0.5525 - val_loss:
    1.4239 - val_accuracy: 0.4875
Epoch 3/5
313/313 [==============================] - 79s 253ms/step - loss: 0.9874 - accuracy: 0.6519 - val_loss:
    1.4682 - val_accuracy: 0.4847
Epoch 4/5
313/313 [==============================] - 75s 239ms/step - loss: 0.7326 - accuracy: 0.7633 - val_loss:
    1.4848 - val_accuracy: 0.5039
Epoch 5/5
313/313 [==============================] - 69s 220ms/step - loss: 0.4928 - accuracy: 0.8481 - val_loss:
    1.6049 - val_accuracy: 0.5052
Epoch 1/5
313/313 [==============================] - 16s 48ms/step - loss: 2.2444 - accuracy: 0.1803 - val_loss:
    1.6663 - val_accuracy: 0.3851
Epoch 2/5
313/313 [==============================] - 16s 50ms/step - loss: 1.6087 - accuracy: 0.4011 - val_loss:
    1.4804 - val_accuracy: 0.4693
Epoch 3/5
313/313 [==============================] - 15s 47ms/step - loss: 1.4373 - accuracy: 0.4679 - val_loss:
    1.4051 - val_accuracy: 0.4922
Epoch 4/5
313/313 [==============================] - 15s 47ms/step - loss: 1.2821 - accuracy: 0.5459 - val_loss:
    1.2692 - val_accuracy: 0.5448
Epoch 5/5
313/313 [==============================] - 15s 48ms/step - loss: 1.1340 - accuracy: 0.5992 - val_loss:
    1.2754 - val_accuracy: 0.5373
ic| h.history['accuracy'][-1]: 0.36899998784065247
ic| h.history['val_accuracy'][-1]: 0.36570000648498535
ic| h.history['loss'][-1]: 1.7567451000213623
ic| h.history['val_loss'][-1]: 1.756562352180481
ic| h.history['accuracy'][-1]: 0.840399980545044
ic| h.history['val_accuracy'][-1]: 0.5052000284194946
ic| h.history['loss'][-1]: 0.5050879120826721
ic| h.history['val_loss'][-1]: 1.6048572063446045
ic| h.history['accuracy'][-1]: 0.5878000259399414
ic| h.history['val_accuracy'][-1]: 0.5372999906539917
ic| h.history['loss'][-1]: 1.1485614776611328
ic| h.history['val_loss'][-1]: 1.2754000425338745
```

Appendix A Handy Custom Library

```python
import os

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def get_files(DIR:str, file_end:str=".png"):
    return [ os.path.join(DIR, f) for f in os.listdir(DIR) if f.endswith(file_end) ]

def create_all_folders(DIR:str):
    path_ = ""
    for folder_name_ in DIR.split("/"):
        path_ = os.path.join(path_, folder_name_)
        create_folder(path_, False)

def clean_folder(DIR:str):
    create_folder(DIR=DIR, clean=True)

def create_folder(DIR:str, clean:bool=False):
    if not os.path.exists(DIR):
        os.mkdir(DIR)
    elif clean:
        filelist = get_files(DIR)
        for f in filelist:
            os.remove(f)

def make_confusion_matrix(
    cf,
    group_names=None,
    categories='auto',
    count=True,
    percent=True,
    cbar=True,
    xyticks=True,
    xyplotlabels=True,
    sum_stats=True,
    figsize=None,
    cmap='Blues',
    title=None
):
    '''
    This function will make a pretty plot of an sklearn Confusion Matrix cm using a Seaborn heatmap visualization.
    Arguments
    ---------
    cf:            confusion matrix to be passed in
    group_names:   List of strings that represent the labels row by row to be shown in each square.
    categories:    List of strings containing the categories to be displayed on the x,y axis. Default is 'auto'
    count:         If True, show the raw number in the confusion matrix. Default is True.
    normalize:     If True, show the proportions for each category. Default is True.
    cbar:          If True, show the color bar. The cbar values are based off the values in the confusion matrix.
                   Default is True.
    xyticks:       If True, show x and y ticks. Default is True.
    xyplotlabels:  If True, show 'True Label' and 'Predicted Label' on the figure. Default is True.
    sum_stats:     If True, display summary statistics below the figure. Default is True.
    figsize:       Tuple representing the figure size. Default will be the matplotlib rcParams value.
    cmap:          Colormap of the values displayed from matplotlib.pyplot.cm. Default is 'Blues'
                   See http://matplotlib.org/examples/color/colormaps_reference.html

    title:         Title for the heatmap. Default is None.

    https://github.com/DTrimarchi10/confusion_matrix
    '''


    # CODE TO GENERATE TEXT INSIDE EACH SQUARE
    blanks = ['' for i in range(cf.size)]

```

```python
    if group_names and len(group_names)==cf.size:
        group_labels = ["{}\n".format(value) for value in group_names]
    else:
        group_labels = blanks

    if count:
        group_counts = ["{0:0.0f}\n".format(value) for value in cf.flatten()]
    else:
        group_counts = blanks

    if percent:
        group_percentages = ["{0:.2%}".format(value) for value in cf.flatten()/np.sum(cf)]
    else:
        group_percentages = blanks

    box_labels = [f"{v1}{v2}{v3}".strip() for v1, v2, v3 in zip(group_labels,group_counts,group_percentages)]
    box_labels = np.asarray(box_labels).reshape(cf.shape[0],cf.shape[1])


    # CODE TO GENERATE SUMMARY STATISTICS & TEXT FOR SUMMARY STATS
    status = {}
    if sum_stats:
        #Accuracy is sum of diagonal divided by total observations
        status["Accuracy" ]  = np.trace(cf) / float(np.sum(cf))

        #if it is a binary confusion matrix, show some more stats
        if len(cf)==2:
            #Metrics for Binary Confusion Matrices
            status["Precision"] = cf[1,1] / sum(cf[:,1])
            status["Recall"   ] = cf[1,1] / sum(cf[1,:])
            status["F1 Score" ] = 2 * status["Precision"] * status["Recall"] / (status["Precision"] + status["Recall"])

    stats_text = "\n\n" + " | ".join(["{:10s}={:.3f}".format(key, status[key]) for key in status])

    # SET FIGURE PARAMETERS ACCORDING TO OTHER ARGUMENTS
    if figsize==None:
        #Get default figure size if not set
        figsize = plt.rcParams.get('figure.figsize')

    if xyticks==False:
        #Do not show categories if xyticks is False
        categories=False


    # MAKE THE HEATMAP VISUALIZATION
    fig = plt.figure(figsize=figsize)
    ax = plt.axes()
    sns.heatmap(cf,annot=box_labels,fmt="",cmap=cmap,cbar=cbar,xticklabels=categories,yticklabels=categories)
    ax.set_aspect(1)
    if xyplotlabels:
        plt.ylabel('True label')
        plt.xlabel('Predicted label' + stats_text)
    else:
        plt.xlabel(stats_text)

    if title:
        plt.title(title)

    return fig, status


def make_comparison_matrix(
    dict_of_status_log,
    report_method="best",
    xlabel="",
    ylabel="",
    cbar=True,
    figsize=None,
    cmap='Blues',
```

```python
    title=None
):
    '''
    This function will make a pretty plot of an sklearn Confusion Matrix cm using a Seaborn heatmap visualization.
    Arguments
    '''

    y_header = list(dict_of_status_log)
    x_header = list(dict_of_status_log[y_header[0]])
    entry_list = list(dict_of_status_log[y_header[0]][x_header[0]])
    sqr = np.ceil(np.sqrt(len(entry_list)))

    fig = plt.figure(figsize=figsize)
    for i, entry in enumerate(entry_list):
        # fetch data
        data = np.zeros((len(y_header), len(x_header)))
        for j,x in enumerate(x_header):
            for k,y in enumerate(y_header):
                if report_method == 'best':
                    data[k, j] = np.max(dict_of_status_log[y][x][entry])
                elif report_method == 'worst':
                    data[k, j] = np.min(dict_of_status_log[y][x][entry])
                elif report_method == 'average':
                    data[k, j] = np.average(dict_of_status_log[y][x][entry])
                else:
                    raise ValueError("Only 'best/worst/average' is implemented!")


        group_labels = ["{:.2f}".format(value) for value in data.flatten()]
        box_labels = np.asarray(group_labels).reshape(data.shape[0], data.shape[1])

        # MAKE THE HEATMAP VISUALIZATION
        ax = plt.subplot(sqr,sqr,i+1)
        sns.heatmap(data,annot=box_labels,fmt="",cmap=cmap,cbar=cbar,xticklabels=x_header,yticklabels=y_header)
        ax.set_aspect(1)
        plt.ylabel(ylabel)
        plt.xlabel(xlabel)
        plt.title("{} ({})".format(entry, report_method))

    return fig

def pie_plot(
        labels,
        sizes,
        title,
        figsize=(6,6),
        startangle=90,
        shadow=False
    ):
    fig = plt.figure(figsize=figsize)
    ax = plt.subplot(1, 1, 1)
    ax.pie(sizes, labels=labels, autopct='%1.1f%%',
            shadow=shadow, startangle=startangle)
    ax.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
    plt.title(title)
    return fig

def imgs_plot(
        dict_of_imgs,
        figsize=(6,6)
    ):
    fig = plt.figure(figsize=figsize)
    sqr = np.ceil(np.sqrt(len(dict_of_imgs)))

    for i,label in enumerate(dict_of_imgs):
        ax = plt.subplot(sqr,sqr,i+1)
        ax.imshow(dict_of_imgs[label])
        plt.xlabel(label)

    plt.tight_layout()
```

```
    return fig


def progress_plot(
    h,
    figsize=(6,6)
):
    # Plot
    fig = plt.figure(figsize=figsize)
    plt.subplot(2, 1, 1)
    plt.plot(h.history['accuracy'], label="training")
    plt.plot(h.history['val_accuracy'], label="validation")
    plt.ylabel("Accuracy")
    plt.xlabel("epoch")
    plt.xticks(list(range(1, 1+len(h.history['accuracy']))))
    plt.legend()

    plt.subplot(2, 1, 2)
    plt.plot(np.log10(h.history['loss']), label="training")
    plt.plot(np.log10(h.history['val_loss']), label="validation")
    plt.xticks(list(range(1, 1+len(h.history['loss']))))
    plt.ylabel("Loss (cross-entropy) in Log Scale")
    plt.xlabel("epoch")
    plt.legend()

    return fig

def output_prediction_result_plot(
    labels,
    dict_input_x,
    dict_prob,
    figsize = (12,6),
    OUT_DIR = "",
    tag     = ""
):
    for test_name in dict_prob:
        fig = plt.figure(figsize=figsize)
        # pie : prediction percentage
        ax = plt.subplot(1, 2, 1)
        predict_label = np.argmax(dict_prob[test_name])
        explode = np.zeros(len(labels))
        explode[predict_label] = 0.1
        ax.pie(dict_prob[test_name], labels=tuple(labels), autopct='%1.1f%%', explode=explode)
        plt.xlabel("Prediction Confidence (Probability)")
        ax.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
        # test img
        ax = plt.subplot(1, 2, 2)
        ax.imshow(dict_input_x[test_name])
        plt.xlabel("Input Test Image Data [{}]".format(test_name))
        fig.savefig("{}/test_sample_prediction_{}[{}].png".format(OUT_DIR, tag, test_name), bbox_inches = 'tight')
        plt.close(fig)

    return fig
```

Code 5: My Custom Library

Appendix B P1 - Code

```
# python
import numpy as np
import matplotlib.pyplot as plt

# sklearn
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split #,KFold,cross_val_score
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix

from icecream import ic
```

```python
from enum import IntEnum, auto

# jx-lib
import jx_lib

# data header
class DATA_HEADER(IntEnum):
    PREGNANCIES              = 0
    GLUCOSE                  = auto()
    BLOODPRESSURE            = auto()
    SKINTHICKNESS            = auto()
    INSULIN                  = auto()
    BMI                      = auto()
    DIABETESPEDIGREEFUNCTION = auto()
    AGE                      = auto()
    OUTCOM                   = auto()

# misc:
def print_latex_header(SVC_PARAMS, folder):
        LINE = "\n\
\\begin{{figure}}[H]\n\
\\centering\n\
\\subfloat[1:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(1:5)]}}}} \, \n\
\\subfloat[2:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(2:5)]}}}} \, \n\
\\subfloat[3:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(3:5)]}}}} \, \n\
\\subfloat[4:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(4:5)]}}}} \, \n\
\\subfloat[5:5-Fold]{{\\includegraphics[height=200px]{{../src_code/{folder}/Confusion_matrix_[m:unmodified-C:{c}-K:{k}-(5:5)]}}}} \, \n\
\\caption{{Confusion Matrices for C:{c} K:{k} 5-fold}}\n\
\\label{{table:confusion:{itr}}}\n\
\\end{{figure}}\n\
        "
        ii = 0
        for c in SVC_PARAMS['C']:
            for k in SVC_PARAMS['kernel']:
                ii += 1
                print(LINE.format(c=c, k=k, folder=folder, itr=ii))

##############################
#####        MAIN        #####
##############################
def main():
    # USER DEFINE: ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    mode = "unmodified"
    ENABLE_TRAINING = True
    SVC_PARAMS = {
        'C' : [0.1,1,5,10],
        'kernel': ['linear','poly','rbf','sigmoid']
    }
    categories = ["No Diabetes", "Diabetes"]
    N_FOLD = 5
    PRINT_LATEX = (ENABLE_TRAINING == False)

    # ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    MODES_AVALIABLE = ["unmodified", "balance"]
    if mode not in MODES_AVALIABLE:
        raise ValueError("Invalid mode selection!!")
    ### Directory generation ###
    OUT_DIR = "output/p1/{}".format(mode)
    jx_lib.create_all_folders(DIR=OUT_DIR)
    if ENABLE_TRAINING:
        # directory cleaning
        jx_lib.clean_folder(DIR=OUT_DIR)
    if PRINT_LATEX:
        print_latex_header(SVC_PARAMS=SVC_PARAMS, folder=OUT_DIR)
    ### IMPORT DATA ###
    data = np.loadtxt(open("diabetes.csv"), delimiter=",")

```

```
    ### PRE-PROCESSING DATA ###
    # Feature-wise Normalization:
    X = data[:, 0:DATA_HEADER.OUTCOM]
    Y = data[:, DATA_HEADER.OUTCOM]
    x_min = np.min(X, axis=0)
    x_max = np.max(X, axis=0)
    X =  (X - x_min)/(x_max - x_min)
    Y = np.int8(Y)

    ic(np.shape(X))
    ic(np.max(X, axis=0))
    ic(np.min(X, axis=0))
    ic(np.shape(Y))
    ic(np.max(Y))
    ic(np.min(Y))
    ### Pre-data diagnosis ###
    print("=== Pre-Data Diagnosis ===")
    def diag_if_balance(data_, tag_):
        n_positive = sum(data_)
        n_total = len(data_)
        n_negative = n_total - n_positive
        ic(n_total)
        ic(n_positive)
        ic(n_negative)
        if (n_positive != n_total/2):
            print("[Issue Found] Data not balanced!")
        fig = jx_lib.pie_plot(
            labels=["+", "-"],
            sizes=[n_positive, n_negative],
            title="Label Distribution ({})".format(tag_)
        )
        img_path = "{}/Pie_{}_{}_data.png".format(OUT_DIR, mode, tag_)
        fig.savefig(img_path, bbox_inches = 'tight')
        plt.close(fig)

    diag_if_balance(data_=Y, tag_="Original")

    ### Pre-processing ###
    print("=== Pre-processing ===")
    if "balance" in mode:
        print(" > Pre-processing: balance data:")
        X_pos = X[Y==1]
        X_neg = X[Y==0]
        Y_pos = Y[Y==1]
        Y_neg = Y[Y==0]

        n_pos = len(Y_pos)
        n_neg = len(Y_neg)
        d_n = (n_pos - n_neg)

        X_major = X_pos
        n_repeat = np.ceil(d_n / (n_pos))
        if (n_neg > n_pos):
            X_major = X_neg
            d_n = (n_neg - n_pos)
            n_repeat = np.ceil(d_n / (n_neg))

        X_extra = np.repeat(X_major, d_n, axis=0)
        np.random.shuffle(X_extra)
        X = np.concatenate((X, X_extra[0:d_n]))
        if (n_neg > n_pos):
            Y = np.concatenate((Y, np.ones(d_n)))
        else:
            Y = np.concatenate((Y, np.zeros(d_n)))

        ic(np.shape(X))
        ic(np.shape(Y))
        diag_if_balance(data_=Y, tag_="Balanced")

    # ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    ### TRAIN & VALIDATE ###
    if ENABLE_TRAINING:
```

```
        print("=== Processing ===")
        ### SVC ###
        # Processing Automation:
        dict_of_status_log = {}
        for c in SVC_PARAMS['C']:
            status_log_k = {}
            for k in SVC_PARAMS['kernel']:
                print("=== [m:{}-C:{}-K:{}] ===".format(mode,c,k))
                # K-Fold : 5x => choose the best kernel score
                kf = KFold(n_splits=N_FOLD, shuffle=True) # randomize
                indices = kf.split(X)
                # Perform training and validation
                for trial, indices_pair in enumerate(indices):
                    # partition training and test data:
                    train_index, test_index = indices_pair
                    X_train, X_test = X[train_index], X[test_index]
                    y_train, y_test = Y[train_index], Y[test_index]

                    # declare SVC model:
                    svc_ = SVC(
                        C           = c, # Regularization term
                        kernel      = k,
                    )

                    # FITTING ...
                    svc_classifier_ = svc_.fit(X_train,y_train)

                    # Report:
                    ic(svc_classifier_)

                    ### Test Estimators ###
                    y_predict = svc_classifier_.predict(X_test) # round to evaluate

                    ### REPORT GEN. ###
                    conf_mat = confusion_matrix(y_test, y_predict)

                    # print:
                    ic(conf_mat)

                    # Gen Confusion Matrix Plot
                    labels = ["True Neg","False Pos","False Neg","True Pos"]
                    fig, status = jx_lib.make_confusion_matrix(
                        cf          = conf_mat,
                        group_names = labels,
                        categories  = categories,
                        figsize     = (6,6),
                        cmap        = "rocket"
                    )

                    # store template
                    ic(trial)
                    if trial == 0:
                        status_log_k[k] = status
                    # log all trials
                    for key, val in status.items():
                        if trial == 0:
                            status_log_k[k][key] = [val]
                        else:
                            status_log_k[k][key].append(val)

                    name = "Confusion_matrix_[m:{}-C:{}-K:{}-({}:{})]"\
                        .format(mode, c, k, trial+1, N_FOLD)
                    fig.savefig("{}/{}.png".format(OUT_DIR, name), bbox_inches = 'tight')
                    plt.close(fig)

            # buffer log
            dict_of_status_log[c] = status_log_k

        ### Gen Comparison Summary ###
        ic(dict_of_status_log)
        for method in ['best', 'worst', 'average']:
            img_path = "{}/Summary_{}_{}.png".format(OUT_DIR, mode, method)
```

```
            ic(img_path)
            fig = jx_lib.make_comparison_matrix(
                dict_of_status_log = dict_of_status_log,
                report_method = method,
                figsize = (12,12),
                xlabel  = "kernel",
                ylabel  = "C",
            )
            fig.savefig(img_path, bbox_inches = 'tight')
            plt.close(fig)

if __name__ == "__main__":
    main()
```

Code 6: SVM Full Implementation

Appendix C P2 - Code

```
# lib
import numpy as np
import matplotlib.pyplot as plt
import copy
from typing import List, Dict, Optional
from icecream import ic
import colorsys

# custom lib
import jx_lib

class SOM:
    def __init__(
        self,
        training_data, # Assumed: normalized
        space: int      = 100, # 100 x 100 grid of neurons
        alpha_0: float  = 0.8,
        verbose: bool   = True,
        path: str       = "output",
    ):
        # Initialize the system
        self.training_data = training_data
        self.space = space
        self.alpha_0 = alpha_0
        self.verbose = verbose
        self.path = path

        # Initialize random weights
        self.w = np.random.random((space,space,3))

        # init output
        jx_lib.create_all_folders(path)
        if verbose:
            self.imshow(data=self.w, name="w_0")
            self.imshow(data=np.reshape(self.training_data,(1,self.training_data.shape[0],3)), name="
    color_bar")

    def imshow(self, data, name:str, save:bool=True):
        fig = plt.figure()
        plt.imshow(data)
        if save:
            fig.savefig("{}/{}.png".format(self.path, name), bbox_inches = 'tight')
        if self.verbose:
            plt.title(name)
            # plt.show(block=False)
            # plt.pause(0.5)
            plt.close(fig)

    def learn(
        self,
        sigma_0: int = 10, # [10,40,70]
        tot_training_epochs: int = 10 #600
    ):
```

```
        self.tot_training_epochs = tot_training_epochs
        # learn:
        epoch = 1
        T = tot_training_epochs
        N = self.space
        while epoch <= tot_training_epochs:
            k = epoch

            # Alpha(k) & s(k):
            alpha_k = self.alpha_0 * np.exp(- k / T)
            s_k = sigma_0 * np.exp(- k / T)
            s_k_2_2_division = 1 / (2 * (s_k ** 2)) # pre-optimization
            # w_ij:
            for x in self.training_data:
                # calculate performance index
                diff = np.linalg.norm(x - self.w, axis = 2)
                # find index of winning node
                ind = np.unravel_index(np.argmin(diff, axis=None), diff.shape) # y,x
                # Update weights for neighbourhood
                xx = np.arange(0, N, 1)
                yy = np.arange(0, N, 1)

                ### matrix form (optimization):
                Mj = np.meshgrid(xx, yy)
                Dx = (Mj[0] - ind[1]) ** 2
                Dy = (Mj[1] - ind[0]) ** 2
                Dij2 = Dx+Dy
                Nij = np.exp(- Dij2 * s_k_2_2_division )
                dxw = np.subtract(x, self.w)
                Nw = np.stack([Nij, Nij, Nij], axis=2) # depth stacking
                self.w = self.w + alpha_k * np.multiply(Nw, dxw)

            plot_ind = [1, 20, 40, 100, 600]
            if epoch in plot_ind:
                print("Epoch Number: {}".format(epoch))
                self.imshow(data=self.w,
                    name="[s={}]_w_{}".format(sigma_0, epoch))

            epoch += 1

def hsv2rgb(h,s,v):
    return tuple(round(i * 255) for i in colorsys.hsv_to_rgb(h,s,v))

def main():
    ### IMPORT DATA ###
    # manual pick:
    inputRGB = np.array([
        [255,0,0],
        [0,255,0],
        [0,0,255],
        [255,255,0],
        [255,0,255],
        [0,255,255],
        [128,128,0],
        [128,0,128],
        [0,128,128],
        [255,128,0],
        [255,0,128],
        [128,255,0],
        [0,255,128],
        [128,0,255],
        [0,128,255],
        [255,20,147],
        [220,20,60],
        [255,51,51],
        [255,153,51],
        [255,255,51],
        [51,255,51],
        [153,255,51],
        [51,255,153],
        [51,255,255]])
    # inputRGB = np.array([
```

```python
    #       [0,0,0],
    #       [255,255,255],
    #       [255,0,0],
    #       [0,255,0],
    #       [0,0,255],
    #       [255,255,0],
    #       [0,255,255],
    #       [255,0,255],
    #       [192,192,192],
    #       [128,128,128],
    #       [128,0,0],
    #       [128,128,0],
    #       [0,128,0],
    #       [128,0,128],
    #       [0,128,128],
    #       [0,0,128],
    #       [188,143,143],
    #       [210,105,30],
    #       [147,112,219],
    #       [127,255,212],
    #       [144,238,144],
    #       [255,160,122],
    #       [178,34,34],
    #       [72,61,139],
    # ])

    # randomly generate 24 Colors
    # inputRGB = []
    # for i in np.random.uniform(size=24):
    #     inputRGB.append(list(hsv2rgb(i,1.0,1.0)))
    # inputRGB = np.array(inputRGB)

    # normalization
    normRGB = inputRGB/255.0
    ic(normRGB.shape)

    # SOM:
    for s in [10, 40, 70]:
        som = SOM(
            training_data = normRGB,
            # DEFAULT:
            space    = 100, # 100 x 100 grid of neurons
            alpha_0  = 0.8,
            verbose  = True,
            path     = "output/p2",
        )
        som.learn(
            sigma_0 = s, # [10,40,70]
            tot_training_epochs= 600
        )



if __name__ == "__main__":
    main()
```

Code 7: KSOM Full Implementation

Appendix D P3 - Code

```python
# python
import numpy as np
import matplotlib.pyplot as plt
from enum import IntEnum, auto

# sklearn
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split #,KFold, cross_val_score
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix
```

```python

# tensorflow
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D

# debug:
from icecream import ic

# jx-lib
import jx_lib

class YLabel(IntEnum):
    airplane   = 0
    automobile = auto()
    bird       = auto()
    cat        = auto()
    deer       = auto()
    dog        = auto()
    frog       = auto()
    horse      = auto()
    ship       = auto()
    truck      = auto()

def main():
    # USER DEFINE: ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    ### MODEL ###
    model_list = {
        "MLP": Sequential([
            Flatten(input_shape=(32, 32, 3)),
            Dense(512, activation='sigmoid'),
            Dense(10, activation='softmax')
        ]),
        "CNN-1": Sequential([
            Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)),
            Flatten(),
            Dense(512, activation='sigmoid'),
            Dense(10, activation='softmax')
        ]),
        "CNN-2": Sequential([
            Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)),
            MaxPooling2D((2, 2)),
            Conv2D(64, (3, 3), activation='relu'),
            MaxPooling2D((2, 2)),
            Flatten(),
            Dense(512, activation='sigmoid'),
            Dropout(0.2),
            Dense(512, activation='sigmoid'),
            Dropout(0.2),
            Dense(10, activation='softmax')
        ]),
    }
    ### CONST ###
    PERCENT_TRAINING_SET = 0.2
    BATCH_SIZE           = 32
    MAX_EPOCHS           = 5
    LEARNING_RATE        = 0.001
    LOSS_METHOD          = 'categorical_crossentropy'
    METRIC               = 'accuracy'
    # INIT: ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    # MODES_AVALIABLE = ["unmodified", "balance"]
    # if mode not in MODES_AVALIABLE:
    #     raise ValueError("Invalid mode selection!!")
    ### Directory generation ###
    OUT_DIR = "output/p3"#{}".format(mode)
    jx_lib.create_all_folders(DIR=OUT_DIR)
    #     # directory cleaning
    #     jx_lib.clean_folder(DIR=OUT_DIR)
    def file_path(file_name, tag=".png"):
```

```
        return "{}/{}.{}".format(OUT_DIR, file_name, tag)
    # DATA: ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    ### IMPORT DATA ###
    (X_train_original, y_train_original), (X_test_original, y_test_original) = cifar10.load_data()
    # sample test images for visual reference:
    sample_imgs = {}
    for label in YLabel:
        index = np.where(y_test_original == label)
        sample_imgs[label.name] = X_test_original[index[0][0]]/255.0 # normalize too
    ### PRE-PROCESSING DATA ###
    # one-hot encoding:
    y_train, y_test = to_categorical(y_train_original), to_categorical(y_test_original)
    # normalization (min-max) , since we know the image data is in [0,255]:
    X_train, X_test = X_train_original/255.0, X_test_original/255.0
    # randomly sample 20% of the training set as the training set:
    n_trainingset = len(y_train)
    downsample_index_test_data = np.random.randint(0,n_trainingset,int(n_trainingset * PERCENT_TRAINING_SET))
    X_train = X_train[downsample_index_test_data]
    y_train = y_train[downsample_index_test_data]
    # dataset:
    ic(np.shape(y_train))
    ic(np.shape(X_train))
    ic(np.shape(y_test))
    ic(np.shape(X_test))
    # output sample dataset images:
    fig = jx_lib.imgs_plot(dict_of_imgs=sample_imgs)
    fig.savefig(file_path("sample_imgs"), bbox_inches = 'tight')
    plt.close(fig)
    # TRAIN: ---- ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    # Print summary:
    for model_name, model in model_list.items():
        ic(model.summary())

    histories = {}
    for model_name, model in model_list.items():
        model.compile(
            optimizer   = Adam(lr=LEARNING_RATE),
            loss        = LOSS_METHOD,
            metrics     = [METRIC]
        )
        histories[model_name] = model.fit(
            X_train, y_train,
            verbose=1,
            batch_size=BATCH_SIZE,
            epochs=MAX_EPOCHS,
            validation_data=(X_test, y_test)
        )

    # SUMMARY: - ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    for h_name, h in histories.items():
        # report
        ic(h.history['accuracy'][-1])
        ic(h.history['val_accuracy'][-1])
        ic(h.history['loss'][-1])
        ic(h.history['val_loss'][-1])
        # plot
        fig = jx_lib.progress_plot(h=h)
        fig.savefig(file_path("progress_{}".format(h_name)), bbox_inches = 'tight')
        plt.close(fig)

    # SAMPLE: - ----- ----- ----- ----- ----- ----- ----- ----- ----- #
    for h_name, h in histories.items():
        labels = []
        dict_input_x = {}
        dict_y_pred = {}
        dict_prob = {}
        for label in YLabel:
            labels.append(label.name)
            prediction = model.predict(sample_imgs[label.name].reshape(1, 32, 32, 3))
            probability = np.squeeze(prediction)
            dict_y_pred[label.name] = prediction
```

```
                dict_prob[label.name] = probability
            # plot sample results
            jx_lib.output_prediction_result_plot(
                labels       = labels,
                dict_input_x = sample_imgs,
                dict_prob    = dict_prob,
                figsize      = (8, 4),
                OUT_DIR      = OUT_DIR,
                tag          = h_name
            )



if __name__ == "__main__":
    main()
```

Code 8: MLP and CNN Full Implementation