ASSIGNMENT 3

CS5691 Pattern Recognition and Machine Learning

CS5691 Assignment 3 Code

Team Members:

BE17B007	N Sowmya Manojna
PH17B010	Thakkar Riya Anandbhai
PH17B011	Chaithanya Krishna Moorthy

Indian Institute of Technology, Madras

Contents

All codes excluding the modules are converted to .py files from IPython Notebooks

1 Dataset 1A

1.1 Perceptron

The code written for analyzing Dataset 1A, using Perceptron model is as follows:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import random
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import seaborn as sns

from perceptron import Perceptron

import warnings
warnings.filterwarnings("ignore")


# In[2]:


get_ipython().run_line_magic('matplotlib', 'inline')


# In[3]:


plt.rcParams["font.size"] = 18
plt.rcParams["axes.grid"] = True
plt.rcParams["figure.figsize"] = 12,8
plt.rcParams['font.serif'] = "Cambria"
plt.rcParams['font.family'] = "serif"


# In[4]:


ds1_train = pd.read_csv("train1.csv",header = None)
ds1_test = pd.read_csv("dev1.csv", header=None)
ds1_train.insert(0,"theta",pd.Series(np.ones(len(ds1_train))))
ds1_test.insert(0,"theta",pd.Series(np.ones(len(ds1_test))))
cv, test = train_test_split(ds1_test, test_size = 0.3, random_state = 0)


# In[5]:


fil1 = ds1_train[2] == 0.
fil2 = ds1_train[2] == 1.
ds0_1 = ds1_train.where(fil1 | fil2).dropna()
fil1 = ds1_train[2] == 0.
fil2 = ds1_train[2] == 2.
ds0_2 = ds1_train.where(fil1 | fil2).dropna()
fil1 = ds1_train[2] == 0.
fil2 = ds1_train[2] == 3.
ds0_3 = ds1_train.where(fil1 | fil2).dropna()
```

```
fil1 = ds1_train[2] == 1.
fil2 = ds1_train[2] == 2.
ds1_2 = ds1_train.where(fil1 | fil2).dropna()
fil1 = ds1_train[2] == 1.
fil2 = ds1_train[2] == 3.
ds1_3 = ds1_train.where(fil1 | fil2).dropna()
fil1 = ds1_train[2] == 2.
fil2 = ds1_train[2] == 3.
ds2_3 = ds1_train.where(fil1 | fil2).dropna()

fil1 = cv[2] == 0.
fil2 = cv[2] == 1.
cv0_1 = cv.where(fil1 | fil2).dropna()
fil1 = cv[2] == 0.
fil2 = cv[2] == 2.
cv0_2 = cv.where(fil1 | fil2).dropna()
fil1 = cv[2] == 0.
fil2 = cv[2] == 3.
cv0_3 = cv.where(fil1 | fil2).dropna()
fil1 = cv[2] == 1.
fil2 = cv[2] == 2.
cv1_2 = cv.where(fil1 | fil2).dropna()
fil1 = cv[2] == 1.
fil2 = cv[2] == 3.
cv1_3 = cv.where(fil1 | fil2).dropna()
fil1 = cv[2] == 2.
fil2 = cv[2] == 3.
cv2_3 = cv.where(fil1 | fil2).dropna()

fil1 = test[2] == 0.
fil2 = test[2] == 1.
test0_1 = test.where(fil1 | fil2).dropna()
fil1 = test[2] == 0.
fil2 = test[2] == 2.
test0_2 = test.where(fil1 | fil2).dropna()
fil1 = test[2] == 0.
fil2 = test[2] == 3.
test0_3 = test.where(fil1 | fil2).dropna()
fil1 = test[2] == 1.
fil2 = test[2] == 2.
test1_2 = test.where(fil1 | fil2).dropna()
fil1 = test[2] == 1.
fil2 = test[2] == 3.
test1_3 = test.where(fil1 | fil2).dropna()
fil1 = test[2] == 2.
fil2 = test[2] == 3.
test2_3 = test.where(fil1 | fil2).dropna()


# In[6]:


ds0_1[2] = ds0_1[2].replace([0.,1],[-1,1])
ds0_2[2] = ds0_2[2].replace([0.,2],[-1,1])
ds0_3[2] = ds0_3[2].replace([0.,3],[-1,1])
ds1_2[2] = ds1_2[2].replace([1,2],[-1,1])
ds1_3[2] = ds1_3[2].replace([1,3],[-1,1])
ds2_3[2] = ds2_3[2].replace([2,3],[-1,1])


cv0_1[2] = cv0_1[2].replace([0.,1],[-1,1])
cv0_2[2] = cv0_2[2].replace([0.,2],[-1,1])
cv0_3[2] = cv0_3[2].replace([0.,3],[-1,1])
cv1_2[2] = cv1_2[2].replace([1,2],[-1,1])
cv1_3[2] = cv1_3[2].replace([1,3],[-1,1])
cv2_3[2] = cv2_3[2].replace([2,3],[-1,1])

test0_1[2] = test0_1[2].replace([0.,1],[-1,1])
test0_2[2] = test0_2[2].replace([0.,2],[-1,1])
```

```
test0_3[2] = test0_3[2].replace([0.,3],[-1,1])
test1_2[2] = test1_2[2].replace([1,2],[-1,1])
test1_3[2] = test1_3[2].replace([1,3],[-1,1])
test2_3[2] = test2_3[2].replace([2,3],[-1,1])


# In[7]:


def hyperparameter_testing(train_dat, cv_dat):
    eta_range = [0.001,0.005,0.01,0.05,0.1,1,5,10,100]
    acc_train = []
    acc_cv = []
    for eta in eta_range:
        model = Perceptron(train_dat,learning_rate = eta)
        model.train()
        acc_train.append(model.accuracy(train_dat))
        acc_cv.append(model.accuracy(cv_dat))
    dictionary = {"Hyperparameter": eta_range, "Training Accuracy":acc_train,"CV ...
        Accuracy":acc_cv}
    df = pd.DataFrame(dictionary)
    max_val = np.argmax(np.array(acc_cv))
    print("Maximum accuracy on CV is achieved for the learning rate value: " , ...
        eta_range[max_val])
    return(df)


# In[12]:


tab_01 = hyperparameter_testing(ds0_1,cv0_1)
tab_01.to_csv("acc_02.csv")


# In[8]:


nn0_1 = Perceptron(ds0_1,learning_rate = 0.01)
nn0_1.train()
print(nn0_1.accuracy(test0_1))


# In[23]:


nn0_1.confusionMatrix(ds0_1, name = "training classes 0 and 1",save_fig = True)


# In[24]:


nn0_1.confusionMatrix(test0_1, name = "test classes 0 and 1",save_fig = True)


# In[25]:


nn0_1.plot_decision_region(name = "training classes 0 and 1",savefig = True)


# In[155]:


tab_02 = hyperparameter_testing(ds0_2,cv0_2)
tab_02.to_csv("acc_02.csv")
tab_02


# In[20]:
```

```


nn0_2 = Perceptron(ds0_2)
nn0_2.train()
print(nn0_2.accuracy(test0_2))


# In[26]:


nn0_2.confusionMatrix(ds0_2, name = "training classes 0 and 2",save_fig=True)


# In[27]:


nn0_2.confusionMatrix(test0_2, name = "test classes 0 and 2",save_fig=True)


# In[28]:


nn0_2.plot_decision_region(name = "training classes 0 and 2",savefig = True)


# In[11]:


tab_03 = hyperparameter_testing(ds0_3,cv0_3)
tab_03.to_csv("acc_03.csv")
tab_03


# In[35]:


print(nn0_3.accuracy(test0_3))


# In[29]:


nn0_3 = Perceptron(ds0_3)
nn0_3.train()
nn0_3.plot_decision_region(name = "training classes 0 and 3",savefig = True)


# In[30]:


nn0_3.confusionMatrix(ds0_3, name = "training classes 0 and 3",save_fig=True)
nn0_3.confusionMatrix(test0_3, name = "test classes 0 and 3",save_fig=True)


# In[12]:


tab_13 = hyperparameter_testing(ds1_3,cv1_3)
tab_13.to_csv("acc_13_perc.csv")
tab_13


# In[36]:


print(nn1_3.accuracy(test1_3))


# In[31]:
```

```
nn1_3 = Perceptron(ds1_3)
nn1_3.train()
nn1_3.plot_decision_region(name = "training classes 1 and 3",savefig = True)
nn1_3.confusionMatrix(ds1_3, name = "training classes 1 and 3",save_fig=True)
nn1_3.confusionMatrix(test1_3, name = "test classes 1 and 3",save_fig=True)


# In[13]:


tab_23 = hyperparameter_testing(ds2_3,cv2_3)
tab_23.to_csv("acc_23_perc.csv")
tab_23


# In[37]:


print(nn2_3.accuracy(test2_3))


# In[32]:


nn2_3 = Perceptron(ds2_3)
nn2_3.train()
nn2_3.plot_decision_region(name = "training classes 2 and 3",savefig = True)
nn2_3.confusionMatrix(ds2_3, name = "training classes 2 and 3",save_fig=True)
nn2_3.confusionMatrix(test2_3, name = "test classes 2 and 3",save_fig=True)


# In[14]:


tab_12 = hyperparameter_testing(ds1_2,cv1_2)
tab_12.to_csv("acc_12_perc.csv")
tab_12


# In[38]:


print(nn1_2.accuracy(test1_2))


# In[33]:


nn1_2 = Perceptron(ds1_2,learning_rate = 0.05)
nn1_2.train()
nn1_2.plot_decision_region(name = "training classes 1 and 2",savefig = True)
nn1_2.confusionMatrix(ds1_2, name = "training classes 1 and 2",save_fig=True)
nn1_2.confusionMatrix(test1_2, name = "test classes 1 and 2",save_fig=True)


# In[ ]:
```

1.1.1 Helper Function

The helper function used is as follows:

1.1.1.1 Perceptron

```
import numpy as np
import pandas as pd
```

```python
import random
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

import warnings
warnings.filterwarnings("ignore")

plt.rcParams["font.size"] = 18
plt.rcParams["axes.grid"] = True
plt.rcParams["figure.figsize"] = 12,8
plt.rcParams['font.serif'] = "Cambria"
plt.rcParams['font.family'] = "serif"


class Perceptron():
    def __init__(self,data,epochs = 25,learning_rate = 0.01,verbose = False):
        self.d = data.shape[1]-1
        self.x = np.array(data.iloc[:,:self.d])
        self.y = np.array(data.iloc[:,self.d])
        self.epoch = epochs
        self.len_data = data.shape[0]
        self.eta = learning_rate
        self.verbose = verbose
    def one_epoch(self,w):
        flag = 0
        lst = list(range(self.len_data))
        random.shuffle(lst)
        for j in range(self.len_data):
            m = lst.pop()
            if w@self.x[m]<0:
                s = -1
            else:
                s = 1
            △ = (self.y[m] - s)
            if △==0:
                flag+=1
            w += (self.eta)*(△/2)*self.x[m]
        return(w,flag)
    def train(self):
        w = np.ones(self.d)
        for i in range(self.epoch):
            result = self.one_epoch(w)
            w = result[0]
            if (self.verbose==True):
                print("No. of correctly classified data points in "+str(i)+"th epoch : ", result[1])
            if result[1] == self.len_data:
                break
        if (self.verbose==True):
            print("Convergence reached in "+str(i)+" epochs")
        self.W = w
        #return(self.W)
    def Y(self,m,c,xRange):
        return(m*xRange+c)

    def plot_decision_region(self, name, savefig = False):
        x1Max = max(np.array(self.x[:,1]))+0.5
        x1Min = min(np.array(self.x[:,1]))-0.5
        x2Max = max(np.array(self.x[:,2]))+0.5
        x2Min = min(np.array(self.x[:,2]))-0.5
        color_list = ["palevioletred","royalblue"]
        xx, yy = np.meshgrid(np.arange(x1Min, x1Max, .02), np.arange(x2Min, x2Max, .02))
        z = self.predict(np.c_[np.ones(xx.ravel().shape),xx.ravel(),yy.ravel()])
        z = z.reshape(xx.shape)
        plt.contourf(xx, yy, z, colors= color_list, alpha=0.1)
        plt.contour(xx, yy, z, colors=color_list, alpha=.1)
        plt.scatter(self.x[:,1],self.x[:,2], c=[color_list[j] for j in self.y==1.])
```

```
        #plt.plot(np.linspace(x1Min,x1Max),self.Y(-self.W[1]/self.W[2],-self.W[0]/...
            self.W[2],np.linspace(x1Min,x1Max)),label = "Decision Region")
        plt.xlabel("X1")
        plt.ylabel("X2")
        plt.title("Decision region plot for the " + name + " data points")
        #plt.legend()
        if savefig == True:
            plt.savefig(name+"dec_reg_perceptron.png")
        plt.show()

    def predict(self,mat):
        return(np.sign(mat @ self.W))
    def accuracy(self,test_data):
        prediction = self.predict(test_data.iloc[:,:self.d])
        compare = prediction == test_data.iloc[:,self.d]
        return(np.sum(compare)/len(test_data))

    def confusionMatrix(self, dat, name, save_fig = False):
        prediction = self.predict(dat.iloc[:,:self.d])
        conf_mat = confusion_matrix(dat.iloc[:,self.d],prediction)
        plt.figure()
        sns.heatmap(conf_mat, annot=True)
        plt.title("1A - Confusion Matrix for " + name + " data (Perceptron)")
        plt.xlabel("Predicted Class")
        plt.ylabel("Actual Class")
        if (save_fig == True):
            plt.savefig("perceptron_" + name + "_confmat.png")
        plt.show()
```

1.2 MLFFNN

The code written for analyzing Dataset 1A, using an MLFFNN model is as follows:

```
#!/usr/bin/env python
# coding: utf-8

# # Assignment 3 - 1A (MLFFNN)
#
# Team members:
# - N Sowmya Manojna (BE17B007)
# - Thakkar Riya Anandbhai (PH17B010)
# - Chaithanya Krishna Moorthy (PH17B011)

# ## Importing Essential Libraries

# In[1]:


import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import matplotlib.pyplot as plt
plt.rcParams["font.size"] = 18
plt.rcParams["axes.grid"] = True
plt.rcParams["figure.figsize"] = 12,8
plt.rcParams['font.serif'] = "Cambria"
plt.rcParams['font.family'] = "serif"

get_ipython().run_line_magic('load_ext', 'autoreload')
```

```
get_ipython().run_line_magic('autoreload', '2')

import warnings
warnings.filterwarnings("ignore")

from gridsearch import GridSearch1A


# ## Reading the data, Splitting it

# In[2]:


# Get the data
column_names = ["x1", "x2", "y"]
df = pd.read_csv("../datasets/1A/train.csv", names=column_names)
df_test = pd.read_csv("../datasets/1A/dev.csv", names=column_names)
display(df.head())

# Split dev into test and validation
df_val, df_test = train_test_split(df_test, test_size=0.3, random_state=42)
display(df_val.head())
display(df_test.head())


# In[3]:


X_train = df.drop("y", axis=1).to_numpy()
y_train = df["y"].to_numpy().astype("int")

X_val = df_val.drop("y", axis=1).to_numpy()
y_val = df_val["y"].to_numpy().astype("int")

X_test = df_test.drop("y", axis=1).to_numpy()
y_test = df_test["y"].to_numpy().astype("int")


# ## Training the Model

# In[4]:


parameters = {"hidden_layer_sizes":[5,8,10,15], "activation":["logistic", "tanh", "...
    relu"],                "solver":["lbfgs", "sgd", "adam"], "batch_size":[100, ...
    200],                "alpha":[0, 0.0001], "learning_rate":["constant", "adaptive"...
    , "invscaling"],               }

mlp = MLPClassifier(random_state=1)

clf = GridSearch1A(mlp, parameters)
clf.fit(X_train, y_train, X_val, y_val)
result_df = pd.DataFrame(clf.cv_results_)
result_df.to_csv("../parameter_search/1A_MLFFNN_train_val.csv")
result_df.head()


# In[5]:


print("Best Parameters Choosen:")
for i in clf.best_params_:
    print("   - ", i, ": ", clf.best_params_[i], sep="")

best_mlp = MLPClassifier(random_state=1, **clf.best_params_)
best_mlp.fit(X_train, y_train)


# ## Best Model Predictions
```

```

# In[6]:


y_pred = best_mlp.predict(X_train)
print("Accuracy:", 100*np.sum(y_pred==y_train)/y_train.size)
conf_mat = confusion_matrix(y_train, y_pred)
plt.figure()
sns.heatmap(conf_mat, annot=True)
plt.title("1A - Train Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1A_MLFFNN_train_confmat.png")
plt.show()

y_val_pred = best_mlp.predict(X_val)
print("Validation Accuracy:", 100*np.sum(y_val_pred==y_val)/y_val.size)
val_conf_mat = confusion_matrix(y_val, y_val_pred)
plt.figure()
sns.heatmap(val_conf_mat, annot=True)
plt.title("1A - Validation Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1A_MLFFNN_val_confmat.png")
plt.show()

y_test_pred = best_mlp.predict(X_test)
print("Validation Accuracy:", 100*np.sum(y_test_pred==y_test)/y_test.size)
test_conf_mat = confusion_matrix(y_test, y_test_pred)
plt.figure()
sns.heatmap(test_conf_mat, annot=True)
plt.title("1A - Test Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1A_MLFFNN_test_confmat.png")
plt.show()


# ## Visualising the decision boundaries

# In[7]:


h = 0.02
x_min, x_max = X_train[:,0].min() - .5, X_train[:,0].max() + .5
y_min, y_max = X_train[:,1].min() - .5, X_train[:,1].max() + .5

xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z_pro = np.argmax(best_mlp.predict_proba(np.c_[xx.ravel(), yy.ravel()]), axis=1)
Z_pro = Z_pro.reshape(xx.shape)

color_list = ["springgreen", "gold", "palevioletred", "royalblue"]
plt.title("1A - Decision Region Plot (MLFFNN)")
plt.contourf(xx, yy, Z_pro, np.unique(Z_pro).size-1, colors=color_list, alpha=0.1)
plt.contour(xx, yy, Z_pro, np.unique(Z_pro).size-1, colors=color_list, alpha=1)
plt.scatter(X_train[:,0], X_train[:,1], c=[color_list[i] for i in y_train])
plt.xlabel("X1")
plt.ylabel("X2")
plt.savefig("images/1A_MLFFNN_Decision_Plot.png")
plt.show()


# In[ ]:
```

1.2.1 Helper Function

The helper functions used are as follows:

1.2.1.1 Gridsearch

```python
import numpy as np
import pandas as pd
from time import time
from tqdm import tqdm
from collections import defaultdict
from sklearn.neural_network import MLPClassifier

class GridSearch1A():
    def __init__(self, model, parameters, verbose=0):
        self.model = model
        self.parameters = parameters
        self.verbose = verbose
        params_list = []
        self.params_keys = self.parameters.keys()

        for hls in parameters["hidden_layer_sizes"]:
            for act in parameters["activation"]:
                for s in parameters["solver"]:
                    for bs in parameters["batch_size"]:
                        for a in parameters["alpha"]:
                            for lr in parameters["learning_rate"]:
                                params_list.append({"hidden_layer_sizes":hls, \
                                                    "activation":act, \
                                                    "solver":s, \
                                                    "batch_size":bs, \
                                                    "alpha":a, \
                                                    "learning_rate":lr})
        self.params_list = params_list

    def fit(self, X_train, y_train, X_val, y_val):
        self.cv_results_ = pd.DataFrame(columns=self.params_keys)

        self.params_ = defaultdict(list)
        self.acc_list_ = []
        self.val_acc_list_ = []
        self.t_inv_list_ = []

        for params in tqdm(self.params_list):
            st = time()
            mlp = MLPClassifier(random_state=1, **params)

            mlp.fit(X_train, y_train)
            et = time()

            y_pred = mlp.predict(X_train)
            acc = 100*np.sum(y_pred==y_train)/y_train.size

            y_val_pred = mlp.predict(X_val)
            val_acc = 100*np.sum(y_val_pred==y_val)/y_val.size

            for i in params:
                self.params_[i].append(params[i])

            self.acc_list_.append(acc)
            self.val_acc_list_.append(val_acc)
            self.t_inv_list_.append(1/(et-st))

        for i in params:
            self.cv_results_[i] = self.params_[i]

        self.cv_results_["accuracy"] = self.acc_list_
        self.cv_results_["val_accuracy"] = self.val_acc_list_
        self.cv_results_["sum_accuracy"] = self.cv_results_["accuracy"] + self....
            cv_results_["val_accuracy"]
        self.cv_results_["t_inv"] = self.t_inv_list_
```

```
        self.cv_results_ = self.cv_results_.sort_values(by=["accuracy", "...
            sum_accuracy", "t_inv"], ascending=False, ignore_index=True)

        self.best_params_ = self.cv_results_.iloc[0].to_dict()
        del self.best_params_["accuracy"]
        del self.best_params_["val_accuracy"]
        del self.best_params_["sum_accuracy"]
        del self.best_params_["t_inv"]


class GridSearch1B():
    def __init__(self, model, parameters, verbose=0):
        self.model = model
        self.parameters = parameters
        self.verbose = verbose
        params_list = []
        self.params_keys = self.parameters.keys()

        for hls in parameters["hidden_layer_sizes"]:
            for act in parameters["activation"]:
                for bs in parameters["batch_size"]:
                    for a in parameters["alpha"]:
                        for lr in parameters["learning_rate"]:
                            for es in parameters["early_stopping"]:
                                params_list.append({"hidden_layer_sizes":hls, \
                                                    "early_stopping":es, \
                                                    "learning_rate":lr, \
                                                    "activation":act, \
                                                    "batch_size":bs, \
                                                    "alpha":a})
        self.params_list = params_list

    def fit(self, X_train, y_train, X_val, y_val):
        self.cv_results_ = pd.DataFrame(columns=self.params_keys)

        self.params_ = defaultdict(list)
        self.acc_list_ = []
        self.val_acc_list_ = []
        self.t_inv_list_ = []

        for params in tqdm(self.params_list):
            st = time()
            mlp = MLPClassifier(random_state=1, **params)

            mlp.fit(X_train, y_train)
            et = time()

            y_pred = mlp.predict(X_train)
            acc = 100*np.sum(y_pred==y_train)/y_train.size

            y_val_pred = mlp.predict(X_val)
            val_acc = 100*np.sum(y_val_pred==y_val)/y_val.size

            for i in params:
                self.params_[i].append(params[i])

            self.acc_list_.append(acc)
            self.val_acc_list_.append(val_acc)
            self.t_inv_list_.append(1/(et-st))

        for i in params:
            self.cv_results_[i] = self.params_[i]

        self.cv_results_["accuracy"] = self.acc_list_
        self.cv_results_["val_accuracy"] = self.val_acc_list_
        self.cv_results_["sum_accuracy"] = self.cv_results_["accuracy"] + self....
            cv_results_["val_accuracy"]
        self.cv_results_["t_inv"] = self.t_inv_list_
```

```
        self.cv_results_ = self.cv_results_.sort_values(by=["accuracy", "...
            sum_accuracy", "t_inv"], ascending=False, ignore_index=True)

        self.best_params_ = self.cv_results_.iloc[0].to_dict()
        self.best_params_["early_stopping"] = bool(self.best_params_["...
            early_stopping"])
        del self.best_params_["accuracy"]
        del self.best_params_["val_accuracy"]
        del self.best_params_["sum_accuracy"]
        del self.best_params_["t_inv"]
```

1.3 Linear SVM

The code written for analyzing Dataset 1A, using the Linear SVM model is as follows:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn import svm
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["font.size"]=18
plt.rcParams["axes.grid"]=True
plt.rcParams["figure.figsize"]=12,8
plt.rcParams["font.serif"]="Cambria"
plt.rcParams["font.family"]="serif"


# In[31]:


from statistics import mode


# In[2]:


from sklearn.metrics import classification_report


# In[3]:


from sklearn.model_selection import GridSearchCV


# In[4]:


import seaborn as sns


# In[5]:


color_list=["springgreen","gold","palevioletred","royalblue"]


# In[6]:

```

```
cols=["x1","x2","y"]
train_data=pd.read_csv("train.csv",names=cols)
dev_data=pd.read_csv("dev.csv",names=cols)


# In[7]:


data_cv,data_test=train_test_split(dev_data,test_size=0.3,random_state=42)


# In[8]:


X_train=train_data[["x1","x2"]].to_numpy()
y_train=train_data["y"].to_numpy().astype("int")

X_cv=data_cv[["x1","x2"]].to_numpy()
y_cv=data_cv["y"].to_numpy().astype("int")

X_test=data_test[["x1","x2"]].to_numpy()
y_test=data_test["y"].to_numpy().astype("int")


# In[9]:


train_data.head()


# ## Training the Model

# ## we proceed with C=1:

# ## Linear SVM classifier for every pair of classes:

# In[15]:


def linear_ovo_plot(y1,y2,df,save_name,title,color,conf_title_train,conf_title_test...
    ,conf_train_save_name,conf_test_save_name,df_dev):
    df2=df.loc[df["y"].isin([y1,y2])]
    df2_dev=df_dev.loc[df_dev["y"].isin([y1,y2])]
    df2_cv,df2_test=train_test_split(df2_dev,test_size=0.3,random_state=42)
    predictor=svm.SVC(kernel="linear",C=1,decision_function_shape="ovo").fit(df2....
        iloc[:,:-1],df2.iloc[:,-1])
    h=0.1
    x1_min,x1_max=df2["x1"].min()-1,df2["x1"].max()+1
    x2_min,x2_max=df2["x2"].min()-1,df2["x2"].max()+1
    xx,yy=np.meshgrid(np.arange(x1_min,x1_max,h),np.arange(x2_min,x2_max,h))
    z=predictor.predict(np.c_[xx.ravel(),yy.ravel()])
    z=z.reshape(xx.shape)

    w=predictor.coef_[0]
    a=-w[0]/w[1]


    plt.figure()
    x2=np.linspace(xx.min(),xx.max())
    yx=a*x2-predictor.intercept_[0]/w[1]
    plt.plot(x2,yx,label="Decision Boundary")

    yx=a*x2-(predictor.intercept_[0]-1)/w[1]
    plt.plot(x2,yx,"k--", label="Support Vector")

    yx=a*x2-(predictor.intercept_[0]+1)/w[1]
    plt.plot(x2,yx,"k--",label="Support Vector")
    c1=color_list[y1]
```

```
    c2=color_list[y2]
    colors_list=[c1,c2]

    plt.contourf(xx,yy,z,np.unique(z).size-1,colors=color,alpha=0.25)
    plt.scatter(df2["x1"],df2["x2"],c=[color_list[i] for i in df2["y"].astype(int)...
        ])
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.xlim(xx.min(),xx.max())
    plt.ylim(yy.min(),yy.max())
    plt.legend(loc="upper right")
    plt.savefig(save_name)
    plt.title(title)
    plt.show()

    y_train=df2["y"]
    ytrain_pred=predictor.predict(df2.iloc[:,:-1])

    y_cv=df2_cv["y"]
    y_test=df2_test["y"]
    ytest_pred=predictor.predict(df2_test.iloc[:,:-1])


    conf_mat=confusion_matrix(y_train,ytrain_pred)
    plt.figure()
    sns.heatmap(conf_mat,annot=True)
    plt.title(conf_title_train)
    plt.xlabel("Predicted Class")
    plt.ylabel("Actual Class")
    plt.savefig(conf_train_save_name)
    plt.show()

    conf_mat=confusion_matrix(y_test,ytest_pred)
    plt.figure()
    sns.heatmap(conf_mat,annot=True)
    plt.title(conf_title_test)
    plt.xlabel("Predicted Class")
    plt.ylabel("Actual Class")
    plt.savefig(conf_test_save_name)
    plt.show()



# In[16]:


linear_ovo_plot(1,2,train_data,"images/1A_ovo_12.png","Support vectors and Boundary...
     region between y=1.0 and y=2.0",color=[color_list[1],color_list[2]],...
    conf_title_train="Confusion Matrix on train data",conf_title_test="Confusion ...
    matrix on test data",conf_train_save_name="images/1A_ovo_conf12_train.png",...
    conf_test_save_name="images/1A_ovo_conf12_test.png",df_dev=dev_data)


# In[17]:


linear_ovo_plot(1,3,train_data,"images/1A_ovo_13.png","Support vectors and Boundary...
     region between y=1.0 and y=3.0",color=[color_list[1],color_list[3]],...
    conf_title_train="Confusion Matrix on train data",conf_title_test="Confusion ...
    matrix on test data",conf_train_save_name="images/1A_ovo_conf13_train.png",...
    conf_test_save_name="images/1A_ovo_conf13_test.png",df_dev=dev_data)


# In[18]:


linear_ovo_plot(0,1,train_data,"images/1A_ovo_01.png","Support vectors and Boundary...
     region between y=0.0 and y=1.0",color=[color_list[0],color_list[1]],...
    conf_title_train="Confusion Matrix on train data",conf_title_test="Confusion ...
```

```
    matrix on test data",conf_train_save_name="images/1A_ovo_conf01_train.png",...
    conf_test_save_name="images/1A_ovo_conf01_test.png",df_dev=dev_data)


# In[19]:


linear_ovo_plot(0,2,train_data,"images/1A_ovo_02.png","Support vectors and Boundary...
     region between y=0.0 and y=2.0",color=[color_list[0],color_list[2]],...
    conf_title_train="Confusion Matrix on train data",conf_title_test="Confusion ...
    matrix on test data",conf_train_save_name="images/1A_ovo_conf02_train.png",...
    conf_test_save_name="images/1A_ovo_conf02_test.png",df_dev=dev_data)


# In[20]:


linear_ovo_plot(0,3,train_data,"images/1A_ovo_03.png","Support vectors and Boundary...
     region between y=0.0 and y=3.0",color=[color_list[0],color_list[3]],...
    conf_title_train="Confusion Matrix on train data",conf_title_test="Confusion ...
    matrix on test data",conf_train_save_name="images/1A_ovo_conf03_train.png",...
    conf_test_save_name="images/1A_ovo_conf03_test.png",df_dev=dev_data)


# In[21]:


linear_ovo_plot(2,3,train_data,"images/1A_ovo_23.png","Support vectors and Boundary...
     region between y=2.0 and y=3.0",color=[color_list[2],color_list[3]],...
    conf_title_train="Confusion Matrix on train data",conf_title_test="Confusion ...
    matrix on test data",conf_train_save_name="images/1A_ovo_conf23_train.png",...
    conf_test_save_name="images/1A_ovo_conf23_test.png",df_dev=dev_data)


# # Using one-vs-one models to predict for a test sample:

# In[28]:


def class_model(df,y1,y2,C=1):
    df2=df.loc[df["y"].isin([y1,y2])]
    predictor=svm.SVC(kernel="linear",C=C,decision_function_shape="ovo").fit(df2....
        iloc[:,:-1],df2.iloc[:,-1])
    return(predictor)


# In[29]:


model01=class_model(train_data,0,1)
model02=class_model(train_data,0,2)
model03=class_model(train_data,0,3)
model12=class_model(train_data,1,2)
model13=class_model(train_data,1,3)
model23=class_model(train_data,2,3)


# In[54]:


from collections import Counter


# In[57]:


def ovo_predictor(x):
    c=[]
    c.append(model01.predict(x)[0])
```

```
    c.append(model02.predict(x)[0])
    c.append(model03.predict(x)[0])
    c.append(model12.predict(x)[0])
    c.append(model13.predict(x)[0])
    c.append(model23.predict(x)[0])
    count=Counter(c)
    freq=0
    label=0
    for i in count.keys():
        if count[i]>freq:
            freq=count[i]
            label=i
    return label


# In[58]:


ytrain_pred=[]
ycv_pred=[]
ytest_pred=[]
for i in range(len(X_train)):
    x=X_train[i,:].reshape(1,-1)
    ytrain_pred.append(ovo_predictor(x))
for x in X_cv:
    x=x.reshape(1,-1)
    ycv_pred.append(ovo_predictor(x))
for x in X_test:
    x=x.reshape(1,-1)
    ytest_pred.append(ovo_predictor(x))


# In[59]:


def accuracy(actual,predicted):
    return 100*np.sum(predicted==actual)/actual.size


# In[60]:


accuracy(y_train,ytrain_pred)


# In[67]:


conf_mat=confusion_matrix(y_train,ytrain_pred)
plt.figure()
sns.heatmap(conf_mat,annot=True)
plt.title("1a - Confusion matrix for train data" )
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1a_confmatrix_train.png" )
plt.show()

conf_mat=confusion_matrix(y_test,ytest_pred)
plt.figure()
sns.heatmap(conf_mat,annot=True)
plt.title("1a - Confusion matrix for test data")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1a_confmatrix_test.png")
plt.show()


# In[64]:

```

```

h=0.1
x1_min,x1_max=train_data["x1"].min()-1,train_data["x1"].max()+1
x2_min,x2_max=train_data["x2"].min()-1,train_data["x2"].max()+1
xx,yy=np.meshgrid(np.arange(x1_min,x1_max,h),np.arange(x2_min,x2_max,h))
X=np.c_[xx.ravel(),yy.ravel()]
z=[]
for i in X:
    x=i.reshape(1,-1)
    z.append(ovo_predictor(x))
z=np.array(z)
z=z.reshape(xx.shape)
plt.figure()
plt.contour(xx,yy,z,np.unique(z).size-1,colors=color_list,alpha=1)
plt.contourf(xx,yy,z,np.unique(z).size-1,colors=color_list,alpha=0.25)
plt.scatter(train_data["x1"],train_data["x2"],c=[color_list[i] for i in y_train])
plt.xlabel("X1")
plt.ylabel("X2")
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())
plt.title("1A-Full Decision Region Plot(SVM)")
plt.savefig("images/1A_SVM_full_decision_plot.png")
plt.show()


# In[ ]:
```

2 Dataset 1B

2.1 MLFFNN

The code written for analyzing Dataset 1B, using an MLFFNN model is as follows:

```
#!/usr/bin/env python
# coding: utf-8

# # Assignment 3 - 1B (MLFFNN)
#
# Team members:
# - N Sowmya Manojna (BE17B007)
# - Thakkar Riya Anandbhai (PH17B010)
# - Chaithanya Krishna Moorthy (PH17B011)

# ## Import Essential Libraries

# In[1]:


import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import matplotlib.pyplot as plt
plt.rcParams["font.size"] = 18
plt.rcParams["axes.grid"] = True
plt.rcParams['font.serif'] = "Cambria"
plt.rcParams['font.family'] = "serif"

get_ipython().run_line_magic('load_ext', 'autoreload')
```

```python
get_ipython().run_line_magic('autoreload', '2')

import warnings
warnings.filterwarnings("ignore")

from gridsearch import GridSearch1B


# ## Read the data, Split it

# In[2]:


# Get the data
column_names = ["x1", "x2", "y"]
df = pd.read_csv("../datasets/1B/train.csv", names=column_names)
df_test = pd.read_csv("../datasets/1B/dev.csv", names=column_names)
display(df.head())

# Split dev into test and validation
df_val, df_test = train_test_split(df_test, test_size=0.3, random_state=42)
display(df_val.head())
display(df_test.head())


# In[3]:


X_train = df[["x1", "x2"]].to_numpy()
y_train = df["y"].to_numpy().astype("int")

X_val = df_val[["x1", "x2"]].to_numpy()
y_val = df_val["y"].to_numpy().astype("int")

X_test = df_test[["x1", "x2"]].to_numpy()
y_test = df_test["y"].to_numpy().astype("int")


# ## Training the Model

# In[4]:


parameters = {"hidden_layer_sizes":[(5,5),(6,6),(7,7),(8,8),(9,9),(10,10)], ...
                  "activation":["logistic", "relu"],               "batch_size":[50, ...
    100, 200], "early_stopping":[True, False],               "learning_rate":["...
    constant", "adaptive", "invscaling"],               "alpha":[0.01, 0.001]
             }

mlp = MLPClassifier(random_state=1)

clf = GridSearch1B(mlp, parameters)
clf.fit(X_train, y_train, X_val, y_val)
result_df = pd.DataFrame(clf.cv_results_)
result_df.to_csv("../parameter_search/1B_MLFFNN_train_val.csv")
result_df.head(10)


# In[5]:


print("Best Parameters Choosen:")
for i in clf.best_params_:
    print("   - ", i, ": ", clf.best_params_[i], sep="")

best_mlp = MLPClassifier(random_state=1, **clf.best_params_)
best_mlp.fit(X_train, y_train)
```

```
# ## Best Model Predictions

# In[6]:


y_pred = best_mlp.predict(X_train)
print("Accuracy:", 100*np.sum(y_pred==y_train)/y_train.size)
conf_mat = confusion_matrix(y_train, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(conf_mat, annot=True)
plt.title("1B - Train Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1B_MLFFNN_train_confmat.png")
plt.show()

y_val_pred = best_mlp.predict(X_val)
print("Validation Accuracy:", 100*np.sum(y_val_pred==y_val)/y_val.size)
val_conf_mat = confusion_matrix(y_val, y_val_pred)
plt.figure(figsize=(8,6))
sns.heatmap(val_conf_mat, annot=True)
plt.title("1B - Validation Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1B_MLFFNN_val_confmat.png")
plt.show()

y_test_pred = best_mlp.predict(X_test)
print("Test Accuracy:", 100*np.sum(y_test_pred==y_test)/y_test.size)
test_conf_mat = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8,6))
sns.heatmap(test_conf_mat, annot=True)
plt.title("1B - Test Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1B_MLFFNN_test_confmat.png")
plt.show()


# ## Visualising the decision boundaries

# In[7]:


h = 0.02
x_min, x_max = X_train[:,0].min() - .5, X_train[:,0].max() + .5
y_min, y_max = X_train[:,1].min() - .5, X_train[:,1].max() + .5

xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z_pro = np.argmax(best_mlp.predict_proba(np.c_[xx.ravel(), yy.ravel()]), axis=1)
Z_pro = Z_pro.reshape(xx.shape)

color_list = ["springgreen", "gold", "palevioletred", "royalblue"]
plt.figure(figsize=(12,8))
plt.title("1B - Decision Region Plot (MLFFNN)")
plt.contourf(xx, yy, Z_pro, np.unique(Z_pro).size-1, colors=color_list, alpha=0.1)
plt.contour(xx, yy, Z_pro, np.unique(Z_pro).size-1, colors=color_list, alpha=1)
plt.scatter(X_train[:,0], X_train[:,1], c=[color_list[i] for i in y_train])
plt.xlabel("X1")
plt.ylabel("X2")
plt.savefig("images/1B_MLFFNN_Decision_Plot.png")
plt.show()


# ## Visualising Neuron Responses

# In[8]:

```

```
def get_values(weights, biases, X_train):
    ip = X_train.T
    h1 = weights[0].T @ ip + biases[0].reshape(-1,1)
    a1 = np.maximum(0, h1)
    h2 = weights[1].T @ a1 + biases[1].reshape(-1,1)
    a2 = np.maximum(0, h2)
    h3 = weights[2].T @ a2 + biases[2].reshape(-1,1)
    pred = np.exp(h3)/np.sum(np.exp(h3))

    return a1, a2, pred


# In[9]:


from matplotlib import cm
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import axes3d
grid = np.c_[xx.ravel(), yy.ravel()]

for epochs in [1, 5, 20, 100]:
    mlp = MLPClassifier(random_state=1, max_iter=epochs, **clf.best_params_)
    mlp.fit(X_train, y_train)

    weights = mlp.coefs_
    biases = mlp.intercepts_

    a1, a2, op = get_values(weights, biases, grid)
    a1 = a1.reshape(a1.shape[0], *xx.shape)
    a2 = a2.reshape(a2.shape[0], *xx.shape)
    op = op.reshape(op.shape[0], *xx.shape)


    for i in range(a1.shape[0]):
        fig = plt.figure(figsize=(8,8))
        ax = plt.axes(projection="3d")

        # ax.contour3D(xx, yy, a1[i,:], 500)
        ax.contourf(xx, yy, a1[i,:], 500, cmap=cm.CMRmap)
        ax.set_xlabel("X1")
        ax.set_ylabel("X2")
        ax.set_zlabel("HL1-Neuron "+str(i+1));
        ax.set_title("Epoch: "+ str(epochs) + "; Surface for Layer 1, Neuron "+str(...
            i+1))
        plt.tight_layout()
        plt.savefig("images/1B_MLFFNN_E"+str(epochs)+"_HL1_N"+str(i+1)+".png")
        plt.show()

    for i in range(a2.shape[0]):
        fig = plt.figure(figsize=(8,8))
        ax = plt.axes(projection="3d")

        # ax.contour3D(xx, yy, a2[i,:], 500)
        ax.contourf(xx, yy, a2[i,:], 500, cmap=cm.CMRmap)
        ax.set_xlabel("X1")
        ax.set_ylabel("X2")
        ax.set_zlabel("HL2-Neuron "+str(i+1));
        ax.set_title("Epoch: "+ str(epochs) + "; Surface for Layer 2, Neuron "+str(...
            i+1))
        plt.tight_layout()
        plt.savefig("images/1B_MLFFNN_E"+str(epochs)+"_HL2_N"+str(i+1)+".png")
        plt.show()

    for i in range(op.shape[0]):
        fig = plt.figure(figsize=(8,8))
        ax = plt.axes(projection="3d")

        # ax.contour3D(xx, yy, op[i,:], 500)
        ax.contourf(xx, yy, op[i,:], 500, cmap=cm.CMRmap)
```

```
        ax.set_xlabel("X1")
        ax.set_ylabel("X2")
        ax.set_zlabel("OP-Neuron "+str(i+1));
        ax.set_title("Epoch: "+ str(epochs) + "; Surface for Output Layer, Neuron "...
            +str(i+1))
        plt.tight_layout()
        plt.savefig("images/1B_MLFFNN_E"+str(epochs)+"_OP_N"+str(i+1)+".png")
        plt.show()


mlp = MLPClassifier(random_state=1, max_iter=1000, **clf.best_params_)
mlp.fit(X_train, y_train)

weights = mlp.coefs_
biases = mlp.intercepts_

a1, a2, op = get_values(weights, biases, grid)
a1 = a1.reshape(a1.shape[0], *xx.shape)
a2 = a2.reshape(a2.shape[0], *xx.shape)
op = op.reshape(op.shape[0], *xx.shape)


for i in range(a1.shape[0]):
    fig = plt.figure(figsize=(8,8))
    ax = plt.axes(projection="3d")

    # ax.contour3D(xx, yy, a1[i,:], 500)
    ax.contourf(xx, yy, a1[i,:], 500, cmap=cm.CMRmap)
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.set_zlabel("HL1-Neuron "+str(i+1));
    ax.set_title("Converged; Surface for Layer 1, Neuron "+str(i+1))
    plt.tight_layout()
    plt.savefig("images/1B_MLFFNN_conv_HL1_N"+str(i+1)+".png")
    plt.show()

for i in range(a2.shape[0]):
    fig = plt.figure(figsize=(8,8))
    ax = plt.axes(projection="3d")

    # ax.contour3D(xx, yy, a2[i,:], 500)
    ax.contourf(xx, yy, a2[i,:], 500, cmap=cm.CMRmap)
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.set_zlabel("HL2-Neuron "+str(i+1));
    ax.set_title("Converged; Surface for Layer 2, Neuron "+str(i+1))
    plt.tight_layout()
    plt.savefig("images/1B_MLFFNN_conv_HL2_N"+str(i+1)+".png")
    plt.show()

for i in range(op.shape[0]):
    fig = plt.figure(figsize=(8,8))
    ax = plt.axes(projection="3d")

    # ax.contour3D(xx, yy, op[i,:], 500)
    ax.contourf(xx, yy, op[i,:], 500, cmap=cm.CMRmap)
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.set_zlabel("OP-Neuron "+str(i+1));
    ax.set_title("Converged; Surface for Output Layer, Neuron "+str(i+1))
    plt.tight_layout()
    plt.savefig("images/1B_MLFFNN_conv_OP_N"+str(i+1)+".png")
    plt.show()


# In[ ]:
```

2.1.1 Helper Function

The helper functions used are as follows:

2.1.1.1 Gridsearch

```python
import numpy as np
import pandas as pd
from time import time
from tqdm import tqdm
from collections import defaultdict
from sklearn.neural_network import MLPClassifier

class GridSearch1A():
    def __init__(self, model, parameters, verbose=0):
        self.model = model
        self.parameters = parameters
        self.verbose = verbose
        params_list = []
        self.params_keys = self.parameters.keys()

        for hls in parameters["hidden_layer_sizes"]:
            for act in parameters["activation"]:
                for s in parameters["solver"]:
                    for bs in parameters["batch_size"]:
                        for a in parameters["alpha"]:
                            for lr in parameters["learning_rate"]:
                                params_list.append({"hidden_layer_sizes":hls, \
                                                    "activation":act, \
                                                    "solver":s, \
                                                    "batch_size":bs, \
                                                    "alpha":a, \
                                                    "learning_rate":lr})
        self.params_list = params_list

    def fit(self, X_train, y_train, X_val, y_val):
        self.cv_results_ = pd.DataFrame(columns=self.params_keys)

        self.params_ = defaultdict(list)
        self.acc_list_ = []
        self.val_acc_list_ = []
        self.t_inv_list_ = []

        for params in tqdm(self.params_list):
            st = time()
            mlp = MLPClassifier(random_state=1, **params)

            mlp.fit(X_train, y_train)
            et = time()

            y_pred = mlp.predict(X_train)
            acc = 100*np.sum(y_pred==y_train)/y_train.size

            y_val_pred = mlp.predict(X_val)
            val_acc = 100*np.sum(y_val_pred==y_val)/y_val.size

            for i in params:
                self.params_[i].append(params[i])

            self.acc_list_.append(acc)
            self.val_acc_list_.append(val_acc)
            self.t_inv_list_.append(1/(et-st))

        for i in params:
            self.cv_results_[i] = self.params_[i]

        self.cv_results_["accuracy"] = self.acc_list_
        self.cv_results_["val_accuracy"] = self.val_acc_list_
```

```
        self.cv_results_["sum_accuracy"] = self.cv_results_["accuracy"] + self....
            cv_results_["val_accuracy"]
        self.cv_results_["t_inv"] = self.t_inv_list_
        self.cv_results_ = self.cv_results_.sort_values(by=["accuracy", "...
            sum_accuracy", "t_inv"], ascending=False, ignore_index=True)

        self.best_params_ = self.cv_results_.iloc[0].to_dict()
        del self.best_params_["accuracy"]
        del self.best_params_["val_accuracy"]
        del self.best_params_["sum_accuracy"]
        del self.best_params_["t_inv"]


class GridSearch1B():
    def __init__(self, model, parameters, verbose=0):
        self.model = model
        self.parameters = parameters
        self.verbose = verbose
        params_list = []
        self.params_keys = self.parameters.keys()

        for hls in parameters["hidden_layer_sizes"]:
            for act in parameters["activation"]:
                for bs in parameters["batch_size"]:
                    for a in parameters["alpha"]:
                        for lr in parameters["learning_rate"]:
                            for es in parameters["early_stopping"]:
                                params_list.append({"hidden_layer_sizes":hls, \
                                                    "early_stopping":es, \
                                                    "learning_rate":lr, \
                                                    "activation":act, \
                                                    "batch_size":bs, \
                                                    "alpha":a})
        self.params_list = params_list

    def fit(self, X_train, y_train, X_val, y_val):
        self.cv_results_ = pd.DataFrame(columns=self.params_keys)

        self.params_ = defaultdict(list)
        self.acc_list_ = []
        self.val_acc_list_ = []
        self.t_inv_list_ = []

        for params in tqdm(self.params_list):
            st = time()
            mlp = MLPClassifier(random_state=1, **params)

            mlp.fit(X_train, y_train)
            et = time()

            y_pred = mlp.predict(X_train)
            acc = 100*np.sum(y_pred==y_train)/y_train.size

            y_val_pred = mlp.predict(X_val)
            val_acc = 100*np.sum(y_val_pred==y_val)/y_val.size

            for i in params:
                self.params_[i].append(params[i])

            self.acc_list_.append(acc)
            self.val_acc_list_.append(val_acc)
            self.t_inv_list_.append(1/(et-st))

        for i in params:
            self.cv_results_[i] = self.params_[i]

        self.cv_results_["accuracy"] = self.acc_list_
        self.cv_results_["val_accuracy"] = self.val_acc_list_
```

```
        self.cv_results_["sum_accuracy"] = self.cv_results_["accuracy"] + self....
            cv_results_["val_accuracy"]
        self.cv_results_["t_inv"] = self.t_inv_list_
        self.cv_results_ = self.cv_results_.sort_values(by=["accuracy", "...
            sum_accuracy", "t_inv"], ascending=False, ignore_index=True)

        self.best_params_ = self.cv_results_.iloc[0].to_dict()
        self.best_params_["early_stopping"] = bool(self.best_params_["...
            early_stopping"])
        del self.best_params_["accuracy"]
        del self.best_params_["val_accuracy"]
        del self.best_params_["sum_accuracy"]
        del self.best_params_["t_inv"]
```

2.2 Non-Linear SVM

The code written for analyzing Dataset 1B, using the Non-Linear SVM models is as follows:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn import svm
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["font.size"]=18
plt.rcParams["axes.grid"]=True
plt.rcParams["figure.figsize"]=12,8
plt.rcParams["font.serif"]="Cambria"
plt.rcParams["font.family"]="serif"
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
import seaborn as sns
color_list=["springgreen","gold","palevioletred","cyan"]


# ## Importing and splitting the 1b datasets

# In[2]:


cols=["x1","x2","y"]
train_data=pd.read_csv("train1b.csv",names=cols)
dev_data=pd.read_csv("dev1b.csv",names=cols)
data_cv,data_test=train_test_split(dev_data,test_size=0.3,random_state=42)
X_train=train_data[["x1","x2"]].to_numpy()
y_train=train_data["y"].to_numpy().astype("int")

X_cv=data_cv[["x1","x2"]].to_numpy()
y_cv=data_cv["y"].to_numpy().astype("int")

X_test=data_test[["x1","x2"]].to_numpy()
y_test=data_test["y"].to_numpy().astype("int")


# In[3]:


plt.scatter(train_data["x1"],train_data["x2"],c=[color_list[i] for i in y_train])


```

```
# # Training the polynomial Kernel:

# In[4]:


C_list=[1,10,100,1000]
degree_list=[1,2,3,4,5,6]
gamma_list=[1,0.1,0.01,"auto"]
coef0_list=[10,100]


# In[5]:


param_grid={"C":C_list,"degree":degree_list,"gamma":gamma_list,"coef0":coef0_list,"...
    kernel":["poly"]}


# In[6]:


grid=GridSearchCV(svm.SVC(),param_grid,verbose=7,return_train_score=True,cv=2)


# In[7]:


grid.fit(X_train,y_train)


# In[8]:


results_df=pd.DataFrame(grid.cv_results_)


# In[9]:


results_df=results_df.sort_values(by="rank_test_score")


# In[40]:


results_df.head(10)


# In[48]:


results_df["params"].iloc[9]


# In[12]:


print("Best Parameters Choosen:")
for i in grid.best_params_:
    print("   - ", i, ": ", grid.best_params_[i], sep="")


# In[13]:


best_poly=svm.SVC(C=1000,coef0=100,degree=5,gamma=0.1,kernel="poly")


# In[14]:
```

```python


best_poly.fit(X_train,y_train)
y_cv_polypred=best_poly.predict(X_cv)
y_test_polypred=best_poly.predict(X_test)
y_train_polypred=best_poly.predict(X_train)


# In[15]:


y_poly_trainaccuracy=100*np.sum(y_train_polypred==y_train)/y_train.size
y_poly_cvaccuracy=100*np.sum(y_cv_polypred==y_cv)/y_cv.size


# In[16]:


y_poly_trainaccuracy


# In[17]:


y_poly_cvaccuracy


# In[18]:


y_poly_testaccuracy=100*np.sum(y_test_polypred==y_test)/y_test.size


# In[19]:


y_poly_testaccuracy


# In[20]:


conf_mat=confusion_matrix(y_train,y_train_polypred)
plt.figure()
sns.heatmap(conf_mat,annot=True)
plt.title("1B - Train Confusion Matrix (SVM with Polynomial Kernel)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1B_SVM_poly_train_confmat.png")
plt.show()

print(" Test Accuracy:",y_poly_testaccuracy)
test_conf_mat=confusion_matrix(y_test,y_test_polypred)
plt.figure()
sns.heatmap(test_conf_mat,annot=True)
plt.title("1B - Test Confusion Matrix (SVM with Polynomial Kernel)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1B_SVM_poly_Test_confmat.png")
plt.show()


# # Decision Region Plot for Polynomial Kernel:

# In[21]:


sv=(best_poly.support_vectors_)

```

```

# In[ ]:





# In[49]:


h=0.1
x1_min,x1_max=train_data["x1"].min()-1,train_data["x1"].max()+1
x2_min,x2_max=train_data["x2"].min()-1,train_data["x2"].max()+1
xx,yy=np.meshgrid(np.arange(x1_min,x1_max,h),np.arange(x2_min,x2_max,h))
z=best_poly.predict(np.c_[xx.ravel(),yy.ravel()])
z=z.reshape(xx.shape)
plt.figure()
plt.contour(xx,yy,z,np.unique(z).size-1,colors=color_list,alpha=1)
plt.contourf(xx,yy,z,np.unique(z).size-1,colors=color_list,alpha=0.1)
plt.scatter(train_data["x1"],train_data["x2"],c=[color_list[i] for i in y_train])
plt.scatter(sv[:,0],sv[:,1],marker="x",c="k",label="Support Vectors",alpha=0.5)
plt.xlabel("X1")
plt.ylabel("X2")
plt.legend()
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())
plt.title("1B - Decision Region Plot (Polynomial SVM)")
plt.savefig("images/1B_SVM_poly_decision_plot.png")
plt.show()


# In[ ]:





# # Training the Gaussian Kernel:

# In[23]:


gamma_list=[1,0.01,0.001,0.0001]
C_list=[0.1,1,10,100,1000]

gauss_cv_accuracy={}
gauss_train_accuracy={}
for i in gamma_list:
    gauss_train_accuracy[i]=[]
    gauss_cv_accuracy[i]=[]
    for j in C_list:
        model=svm.SVC(kernel="rbf",decision_function_shape="ovr",C=j,gamma=i)
        model.fit(X_train,y_train)
        ytrain_pred=model.predict(X_train)
        ycv_pred=model.predict(X_cv)
        gauss_train_accuracy[i].append(100*np.sum(ytrain_pred==y_train)/y_train....
            size)
        gauss_cv_accuracy[i].append(100*np.sum(ycv_pred==y_cv)/y_cv.size)


# In[24]:


gauss_cv_accuracy;


# In[25]:


```

```
gauss_accuracy_table=pd.DataFrame()


# In[26]:


gauss_accuracy_table["gamma"...
    ]=[1,1,1,1,1,0.01,0.01,0.01,0.01,0.01,0.001,0.001,0.001,0.001,0.001,0.0001,0.0001,0.0001,0.000


# In[27]:


gauss_accuracy_table["C"...
    ]=[0.1,1,10,100,1000,0.1,1,10,100,1000,0.1,1,10,100,1000,0.1,1,10,100,1000]


# In[28]:


val_ac=[]
for i in (gamma_list):
    for j in range(5):
        val_ac.append(gauss_cv_accuracy[i][j])


# In[29]:


gauss_accuracy_table["validation_accuracy"]=val_ac


# In[30]:


train_ac=[]
for i in (gamma_list):
    for j in range(5):
        train_ac.append(gauss_train_accuracy[i][j])


# In[31]:


gauss_accuracy_table["Train accuracy"]=train_ac


# In[32]:


gauss_accuracy_table


# In[33]:


best_gauss_model=svm.SVC(kernel="rbf",decision_function_shape="ovr",C=1,gamma=1)
best_gauss_model.fit(X_train,y_train)
ygauss_testpred=best_gauss_model.predict(X_test)


# In[34]:


test_gauss_accuracy=100*np.sum(ygauss_testpred==y_test)/y_test.size


# In[35]:
```

```


test_gauss_accuracy


# # Confusion matrix for train and test data set, best gaussian model

# In[36]:


ytraingauss_pred=best_gauss_model.predict(X_train)
print(" Train Accuracy:",100*np.sum(ytraingauss_pred==y_train)/y_train.size)
conf_mat=confusion_matrix(y_train,ytraingauss_pred)
plt.figure()
sns.heatmap(conf_mat,annot=True)
plt.title("1B - Train Confusion Matrix (SVM with Gaussian Kernel)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1B_SVM_gauss_train_confmat.png")
plt.show()

print(" Test Accuracy:",test_gauss_accuracy)
test_conf_mat=confusion_matrix(y_test,ygauss_testpred)
plt.figure()
sns.heatmap(test_conf_mat,annot=True)
plt.title("1B - Test Confusion Matrix (SVM with Gaussian Kernel)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/1B_SVM_gauss_Test_confmat.png")
plt.show()




# # Decision function plot:

# In[37]:


sv_gauss=best_gauss_model.support_vectors_


# In[38]:


h=0.1
x1_min,x1_max=train_data["x1"].min()-1,train_data["x1"].max()+1
x2_min,x2_max=train_data["x2"].min()-1,train_data["x2"].max()+1
xx,yy=np.meshgrid(np.arange(x1_min,x1_max,h),np.arange(x2_min,x2_max,h))
z=best_gauss_model.predict(np.c_[xx.ravel(),yy.ravel()])
z=z.reshape(xx.shape)
plt.figure()
plt.contour(xx,yy,z,np.unique(z).size-1,colors=color_list,alpha=1)
plt.contourf(xx,yy,z,np.unique(z).size-1,colors=color_list,alpha=0.1)
plt.scatter(train_data["x1"],train_data["x2"],c=[color_list[i] for i in y_train])
plt.scatter(sv_gauss[:,0],sv_gauss[:,1],marker="x",c="k",label="Support Vectors",...
    alpha=1)
plt.legend()
plt.xlabel("X1")
plt.ylabel("X2")
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())
plt.title("1B - Decision Region Plot (Gaussian SVM)")
plt.savefig("images/1B_SVM_gauss_decision_plot.png")
plt.show()


# In[ ]:
```

3 Dataset 2A

3.1 MLFFNN

The code written for analyzing Dataset 2A, using an MLFFNN model is as follows:

```python
#!/usr/bin/env python
# coding: utf-8

# # Assignment 3 - 2 (MLFFNN)
#
# Team members:
# - N Sowmya Manojna (BE17B007)
# - Thakkar Riya Anandbhai (PH17B010)
# - Chaithanya Krishna Moorthy (PH17B011)

# ## Import Essential Libraries

# In[1]:


import numpy as np
import pandas as pd
import seaborn as sns
from ast import literal_eval
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

import matplotlib.pyplot as plt
plt.rcParams["font.size"] = 18
plt.rcParams["axes.grid"] = True
plt.rcParams["figure.figsize"] = 12,8
plt.rcParams['font.serif'] = "Cambria"
plt.rcParams['font.family'] = "serif"

get_ipython().run_line_magic('load_ext', 'autoreload')
get_ipython().run_line_magic('autoreload', '2')

import warnings
warnings.filterwarnings("ignore")

from gridsearch import GridSearch2A


# ## Reading the data, Splitting it

# In[2]:


# Get the data
df = pd.read_csv("../datasets/2A/train_new.csv")
df_test = pd.read_csv("../datasets/2A/dev_new.csv")
display(df.head())

# Split dev into test and validation
df_val, df_test = train_test_split(df_test, test_size=0.3, random_state=42)
display(df_val.head())
display(df_test.head())


# In[3]:

```

```

X_train = df.drop("class", axis=1)
y_train = df["class"].to_numpy().astype("int")

X_val = df_val.drop("class", axis=1)
y_val = df_val["class"].to_numpy().astype("int")

X_test = df_test.drop("class", axis=1)
y_test = df_test["class"].to_numpy().astype("int")


# In[4]:


display(df.describe())
display(df_val.describe())
display(df_test.describe())


# ## Preprocessing Dataset

# In[5]:


scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = pd.DataFrame(scaler.transform(X_train), columns=X_train.columns)
X_val_scaled = pd.DataFrame(scaler.transform(X_val), columns=X_val.columns)
X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns)

display(X_train_scaled.describe())
display(X_val_scaled.describe())
display(X_test_scaled.describe())


# ## Training the Model

# In[6]:


parameters = {
              "pca__n_components":list(range(1,25)),
              "mlp__hidden_layer_sizes":[(10,10), (25,25), (50,50), (75,75)], \
              "mlp__batch_size":[50, 100, "auto"], "mlp__alpha":[0.01, 0.001], \
              "mlp__learning_rate":["constant", "adaptive", "invscaling"], \
             }

model = Pipeline([('pca', PCA()), ('mlp', MLPClassifier(max_iter=500, random_state=1))])

clf = GridSearch2A(model, parameters, verbose=1)
clf.fit(X_train, y_train, X_val, y_val)
result_df = pd.DataFrame(clf.cv_results_)
result_df.to_csv("../parameter_search/2A_MLFFNN_train_val.csv")
display(result_df.head(10))


# In[7]:


clf.cv_results_ = clf.cv_results_.sort_values(by=["val_accuracy", "accuracy", "sum_accuracy", "t_inv"], ascending=False, ignore_index=True)

clf.best_params_ = clf.cv_results_.iloc[0].to_dict()
del clf.best_params_["accuracy"]
del clf.best_params_["val_accuracy"]
del clf.best_params_["sum_accuracy"]
del clf.best_params_["t_inv"]

```

```python

# In[8]:


print("Best Parameters Choosen:")
for i in clf.best_params_:
    print("  - ", i, ": ", clf.best_params_[i], sep="")

pca_params = {}
pca_params["n_components"] = clf.best_params_["n_components"]
mlp_params = clf.best_params_
mlp_params["hidden_layer_sizes"] = literal_eval(mlp_params["hidden_layer_sizes"])
try:
    mlp_params["batch_size"] = int(mlp_params["batch_size"])
except:
    pass

del mlp_params["n_components"]

best_model = Pipeline([('pca', PCA(**pca_params)),                              ('mlp', ...
    MLPClassifier(max_iter=500, random_state=1, **mlp_params))])
best_model.fit(X_train, y_train)


# In[9]:


y_pred = best_model.predict(X_train)
print("Accuracy:", 100*np.sum(y_pred==y_train)/y_train.size)
conf_mat = confusion_matrix(y_train, y_pred)
plt.figure()
sns.heatmap(conf_mat, annot=True)
plt.title("2A - Train Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/2A_MLFFNN_train_confmat.png")
plt.show()

y_val_pred = best_model.predict(X_val)
print("Validation Accuracy:", 100*np.sum(y_val_pred==y_val)/y_val.size)
val_conf_mat = confusion_matrix(y_val, y_val_pred)
plt.figure()
sns.heatmap(val_conf_mat, annot=True)
plt.title("2A - Validation Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/2A_MLFFNN_val_confmat.png")
plt.show()

y_test_pred = best_model.predict(X_test)
print("Test Accuracy:", 100*np.sum(y_test_pred==y_test)/y_test.size)
test_conf_mat = confusion_matrix(y_test, y_test_pred)
plt.figure()
sns.heatmap(test_conf_mat, annot=True)
plt.title("2A - Test Confusion Matrix (MLFFNN)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/2A_MLFFNN_test_confmat.png")
plt.show()


# In[ ]:
```

3.1.1 Helper Function

The helper functions used are as follows:

3.1.1.1 Data Consolidation

```python
import os
import numpy as np
import pandas as pd
from tqdm import tqdm

def get_consolidated_data2A(classes_present):
    df = pd.DataFrame()
    df_test = pd.DataFrame()
    for i in classes_present:
        df_new = pd.read_csv("../datasets/2A/"+i+"/train.csv")
        df_new["image_names"] = classes_present[i]
        df_new = df_new.rename(columns={"image_names":"class"})
        df = df.append(df_new)

        df_new_test = pd.read_csv("../datasets/2A/"+i+"/dev.csv")
        df_new_test["image_names"] = classes_present[i]
        df_new_test = df_new_test.rename(columns={"image_names":"class"})
        df_test = df_test.append(df_new_test)

    df.to_csv("../datasets/2A/train.csv", index=False)
    df_test.to_csv("../datasets/2A/dev.csv", index=False)

if __name__ == "__main__":
    classes_present = {"coast":0, "highway":1, "mountain":2, "opencountry":3, "...
        tallbuilding":4}
    get_consolidated_data2A(classes_present)
```

3.1.1.2 Gridsearch

```python
import numpy as np
import pandas as pd
from time import time
from tqdm import tqdm
from collections import defaultdict
from sklearn.neural_network import MLPClassifier

class GridSearch1A():
    def __init__(self, model, parameters, verbose=0):
        self.model = model
        self.parameters = parameters
        self.verbose = verbose
        params_list = []
        self.params_keys = self.parameters.keys()

        for hls in parameters["hidden_layer_sizes"]:
            for act in parameters["activation"]:
                for s in parameters["solver"]:
                    for bs in parameters["batch_size"]:
                        for a in parameters["alpha"]:
                            for lr in parameters["learning_rate"]:
                                params_list.append({"hidden_layer_sizes":hls, \
                                                    "activation":act, \
                                                    "solver":s, \
                                                    "batch_size":bs, \
                                                    "alpha":a, \
                                                    "learning_rate":lr})
        self.params_list = params_list

    def fit(self, X_train, y_train, X_val, y_val):
        self.cv_results_ = pd.DataFrame(columns=self.params_keys)

        self.params_ = defaultdict(list)
        self.acc_list_ = []
        self.val_acc_list_ = []
        self.t_inv_list_ = []

```

```
        for params in tqdm(self.params_list):
            st = time()
            mlp = MLPClassifier(random_state=1, **params)

            mlp.fit(X_train, y_train)
            et = time()

            y_pred = mlp.predict(X_train)
            acc = 100*np.sum(y_pred==y_train)/y_train.size

            y_val_pred = mlp.predict(X_val)
            val_acc = 100*np.sum(y_val_pred==y_val)/y_val.size

            for i in params:
                self.params_[i].append(params[i])

            self.acc_list_.append(acc)
            self.val_acc_list_.append(val_acc)
            self.t_inv_list_.append(1/(et-st))

        for i in params:
            self.cv_results_[i] = self.params_[i]

        self.cv_results_["accuracy"] = self.acc_list_
        self.cv_results_["val_accuracy"] = self.val_acc_list_
        self.cv_results_["sum_accuracy"] = self.cv_results_["accuracy"] + self....
            cv_results_["val_accuracy"]
        self.cv_results_["t_inv"] = self.t_inv_list_
        self.cv_results_ = self.cv_results_.sort_values(by=["accuracy", "...
            sum_accuracy", "t_inv"], ascending=False, ignore_index=True)

        self.best_params_ = self.cv_results_.iloc[0].to_dict()
        del self.best_params_["accuracy"]
        del self.best_params_["val_accuracy"]
        del self.best_params_["sum_accuracy"]
        del self.best_params_["t_inv"]


class GridSearch1B():
    def __init__(self, model, parameters, verbose=0):
        self.model = model
        self.parameters = parameters
        self.verbose = verbose
        params_list = []
        self.params_keys = self.parameters.keys()

        for hls in parameters["hidden_layer_sizes"]:
            for act in parameters["activation"]:
                for bs in parameters["batch_size"]:
                    for a in parameters["alpha"]:
                        for lr in parameters["learning_rate"]:
                            for es in parameters["early_stopping"]:
                                params_list.append({"hidden_layer_sizes":hls, \
                                                    "early_stopping":es, \
                                                    "learning_rate":lr, \
                                                    "activation":act, \
                                                    "batch_size":bs, \
                                                    "alpha":a})
        self.params_list = params_list

    def fit(self, X_train, y_train, X_val, y_val):
        self.cv_results_ = pd.DataFrame(columns=self.params_keys)

        self.params_ = defaultdict(list)
        self.acc_list_ = []
        self.val_acc_list_ = []
        self.t_inv_list_ = []

        for params in tqdm(self.params_list):
```

```
            st = time()
            mlp = MLPClassifier(random_state=1, **params)

            mlp.fit(X_train, y_train)
            et = time()

            y_pred = mlp.predict(X_train)
            acc = 100*np.sum(y_pred==y_train)/y_train.size

            y_val_pred = mlp.predict(X_val)
            val_acc = 100*np.sum(y_val_pred==y_val)/y_val.size

            for i in params:
                self.params_[i].append(params[i])

            self.acc_list_.append(acc)
            self.val_acc_list_.append(val_acc)
            self.t_inv_list_.append(1/(et-st))

        for i in params:
            self.cv_results_[i] = self.params_[i]

        self.cv_results_["accuracy"] = self.acc_list_
        self.cv_results_["val_accuracy"] = self.val_acc_list_
        self.cv_results_["sum_accuracy"] = self.cv_results_["accuracy"] + self....
            cv_results_["val_accuracy"]
        self.cv_results_["t_inv"] = self.t_inv_list_
        self.cv_results_ = self.cv_results_.sort_values(by=["accuracy", "...
            sum_accuracy", "t_inv"], ascending=False, ignore_index=True)

        self.best_params_ = self.cv_results_.iloc[0].to_dict()
        self.best_params_["early_stopping"] = bool(self.best_params_["...
            early_stopping"])
        del self.best_params_["accuracy"]
        del self.best_params_["val_accuracy"]
        del self.best_params_["sum_accuracy"]
        del self.best_params_["t_inv"]
```

3.2 Gaussian-kernel SVM

The code written for analyzing Dataset 1A, using the Gaussian-kernel SVM model is as follows:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn import svm
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["font.size"]=18
plt.rcParams["axes.grid"]=True
plt.rcParams["figure.figsize"]=12,8
plt.rcParams["font.serif"]="Cambria"
plt.rcParams["font.family"]="serif"
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
import seaborn as sns
color_list=["springgreen","gold","palevioletred","cyan"]


# In[2]:
```

```


train_data=pd.read_csv("train_new.csv")
dev_data=pd.read_csv("dev_new.csv")
data_cv,data_test=train_test_split(dev_data,test_size=0.3,random_state=42)
X_train=train_data.drop("class",axis=1).to_numpy()
y_train=train_data["class"].to_numpy().astype("int")

X_cv=data_cv.drop("class",axis=1).to_numpy()
y_cv=data_cv["class"].to_numpy().astype("int")

X_test=data_test.drop("class",axis=1).to_numpy()
y_test=data_test["class"].to_numpy().astype("int")


# In[3]:


train_data.describe()


# In[4]:


dev_data.describe()


# In[5]:


plt.figure(figsize=(30,30))
cor=train_data.corr()
sns.heatmap(cor,annot=True,cmap=plt.cm.Reds)
plt.show()


# In[ ]:


gamma_list=[50,1,0.01,0.001,0.0001,10,100,"auto","scale"]
C_list=[0.01,0.1,1,10,100,1000]

cv_accuracy={}
train_accuracy={}
for i in gamma_list:
    train_accuracy[i]=[]
    cv_accuracy[i]=[]
    for j in C_list:
        model=svm.SVC(kernel="rbf",decision_function_shape="ovr",C=j,gamma=i,...
            probability=True)
        model.fit(X_train,y_train)
        ytrain_pred=model.predict(X_train)
        ycv_pred=model.predict(X_cv)
        train_accuracy[i].append(100*np.sum(ytrain_pred==y_train)/y_train.size)
        cv_accuracy[i].append(100*np.sum(ycv_pred==y_cv)/y_cv.size)


# In[ ]:


cv_accuracy


# In[22]:


C_list=[0.1,0.01,1,10,100,1000]
gamma_list=[0.1,0.01,1,5,10,100,1000,"auto","scale"]
param_grid={"C":C_list,"gamma":gamma_list,"kernel":["rbf"],"tol":[0.1,0.01,1],"...
```

```
    class_weight":["balanced",None],"break_ties":[True,False],"shrinking":[True,...
    False]}
grid=GridSearchCV(svm.SVC(),param_grid,verbose=7,return_train_score=True,cv=2)


# In[23]:


grid.fit(X_train,y_train)


# In[67]:


results_df=pd.DataFrame(grid.cv_results_)


# In[73]:


results_df=results_df.sort_values(by="rank_test_score")
results_df=results_df.reset_index(drop=True)


# In[74]:


results_df.head(10)


# In[28]:


results_df.iloc[0,:]


# In[33]:


params=grid.best_params_


# In[34]:


params


# In[39]:


model=svm.SVC(C=10,break_ties=False,class_weight=None,gamma=1,kernel="rbf",...
    shrinking=True,tol=0.01)


# In[40]:


model.fit(X_train,y_train)


# In[42]:


ytrain_pred=model.predict(X_train)
ytest_pred=model.predict(X_test)
ycv_pred=model.predict(X_cv)


```

```
# In[43]:


y_trainaccuracy=100*np.sum(ytrain_pred==y_train)/y_train.size
y_cvaccuracy=100*np.sum(ycv_pred==y_cv)/y_cv.size
y_testaccuracy=100*np.sum(ytest_pred==y_test)/y_test.size


# In[44]:


y_trainaccuracy


# In[45]:


y_cvaccuracy


# In[46]:


y_testaccuracy


# In[48]:


conf_mat=confusion_matrix(y_train,ytrain_pred)
plt.figure()
sns.heatmap(conf_mat,annot=True)
plt.title("2A - Train Confusion Matrix (SVM with Gaussian Kernel)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/2A_SVM_gauss_train_confmat.png")
plt.show()

print(" Test Accuracy:",y_testaccuracy)
test_conf_mat=confusion_matrix(y_test,ytest_pred)
plt.figure()
sns.heatmap(test_conf_mat,annot=True)
plt.title("2A - Test Confusion Matrix (SVM with Gaussian Kernel)")
plt.xlabel("Predicted Class")
plt.ylabel("Actual Class")
plt.savefig("images/2A_SVM_gauss_Test_confmat.png")
plt.show()


# In[ ]:





# In[32]:


X_train


# In[49]:


results_df.iloc[0,:]


# In[122]:

```

```

results_df["params"][620]


# In[123]:


results_df["mean_train_score"][620]


# In[124]:


results_df["mean_test_score"][620]


# In[71]:


results_df.head()


# In[ ]:
```