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

1 Dataset 1A **2**
1.1 Perceptron 2
1.2 MLFFNN 2
1.3 Linear SVM 6

2 Dataset 1B **6**
2.1 MLFFNN 6
2.2 Non-Linear SVM 13

3 Dataset 2A **13**
3.1 MLFFNN 13
3.2 Gaussian-kernel SVM 14

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

1 Dataset 1A

1.1 Perceptron

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[ ]:
```

The helper functions used are as follows:

```
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

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')
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[ ]:
```

The helper functions used are as follows:

```
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

3 Dataset 2A

3.1 MLFFNN

The helper function used is as follows:

```
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)
```

3.2 Gaussian-kernel SVM