Project1 Appendix

NYCU Spr2023: AI Capstone

Author: 110652019 林楷傑

Image

```
import os
import torch
import torchvision
import pandas as pd
from skimage import io, transform
import numpy as np
import torch.nn.functional as F
import matplotlib.pyplot as plt

from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision.datasets import CIFAR10
from torchvision import transforms, utils

from einops import rearrange, repeat
from einops.layers.torch import Rearrange

# Ignore warnings
import warnings
warnings.filterwarnings("ignore")

from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, confusion_matrix
import itertools

train_tf = transforms.Compose([
    transforms.RandomRotation(degrees=0.66),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]),
])

test_tf = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]),
])
train_data = CIFAR10(download=True, root="./data", transform=train_tf)
test_data = CIFAR10(root="./data", train=False, transform=test_tf)
train_class_item = dict()
```

```
for item in train_data:
    label = train_data.classes[item[1]]
    if label not in train_class_item:
        train_class_item[label] = 1
    else:
        train_class_item[label] += 1
train_class_item
test_class_item = dict()
for item in test_data:
    label = test_data.classes[item[1]]
    if label not in test_class_item:
        test_class_item[label] = 1
    else:
        test_class_item[label] += 1
test_class_item
BATCH_SIZE = 128
train_dataset_half = torch.utils.data.random_split(train_data, [25000,
25000])[0]
train_dl = DataLoader(train_data, BATCH_SIZE, num_workers=2,
pin_memory=True, shuffle=True)
train_loader_half = DataLoader(dataset=train_dataset_half, batch_size=128,
shuffle=True, num_workers=2, pin_memory=True)
test_dl = DataLoader(test_data, BATCH_SIZE, num_workers=2,
pin_memory=True, shuffle=False)
show_sample = DataLoader(train_data, 16, shuffle=True)

def imshow(img):
    img = img/2 + 0.5
    npimg = img.numpy()
    plt.figure(figsize=(20,20))
    plt.imshow(np.transpose(npimg, (1,2,0)))
    plt.show()

dataiter = iter(show_sample)
images, labels = dataiter.next()
print(images.shape)
imshow(torchvision.utils.make_grid(images))
device = torch.device("cuda") if torch.cuda.is_available() else
torch.device("cpu")
device
def to_device(data,device):
    if isinstance(data,(list,tuple)):
        return [to_device(x,device) for x in data]
    return data.to(device,non_blocking=True)

class ToDeviceLoader:
    def __init__(self,data,device):
        self.data = data
        self.device = device

    def __iter__(self):
        for batch in self.data:
            yield to_device(batch,self.device)
```

```
    def __len__(self):
        return len(self.data)

train_dl = ToDeviceLoader(train_dl, device)
train_dl_half = ToDeviceLoader(train_loader_half, device)
test_dl = ToDeviceLoader(test_dl, device)

def plot_training_metrics(history, model, test_data, y_actual, y_pred,
classes, model_name='model'):
    """
    Input: trained model history, model, test image generator, actual and
predicted labels, class list
    Output: Plots loss vs epochs, accuracy vs epochs, confusion matrix
    """
    fpr, tpr, thresholds = metrics.roc_curve(y_actual, y_pred,
pos_label=9)
    AUC       = metrics.auc(fpr, tpr)*100
    Acc       = accuracy_score(y_actual, y_pred)*100
    results_title =(f"\n Model AUC {AUC:.2f}%, Accuracy {Acc:.2f}% on Test
Data\n")
    print(results_title.format(AUC, Acc))


    # print classification report
    print(classification_report(y_actual, y_pred, target_names=classes))

    # extract data from training history for plotting
    loss_values     = [x.get("train_loss") for x in history]
    val_loss_values = [x["val_loss"] for x in history]
    val_acc_values  = [x["val_acc"] for x in history]


    # get the min loss and max accuracy for plotting
    min_loss = np.min(val_loss_values)

    # create plots
    plt.subplots(figsize=(12,4))

    # plot loss by epochs
    plt.subplot(1,3,1)
    plt.plot(loss_values, 'bo', label = 'Training loss')
    plt.plot(val_loss_values, 'cornflowerblue', label = 'Validation loss')
    plt.title('Validation Loss by Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.axhline(y=min_loss,color='darkslategray', linestyle='--')
    plt.legend()

    # plot accuracy by epochs
    plt.subplot(1,3,2)
    plt.plot(val_acc_values, 'cornflowerblue', label = 'Validation
Accuracy')
    plt.title('Validation Accuracy by Epochs')
    plt.xlabel('Epochs')
```

```
    plt.ylabel('AUC')
    plt.legend()


     # calculate Confusion Matrix
    cm = confusion_matrix(y_true, y_pred)

    # create confusion matrix plot
    plt.subplot(1,3,3)
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.BuPu)
    plt.title(f"Confusion Matrix \nAUC: {AUC:.2f}%")
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    # loop through matrix, plot each
    threshold = cm.max() / 2.
    for r, c in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(c, r, format(cm[r, c], 'd'),
                 horizontalalignment="center",
                 color="white" if cm[r, c] > threshold else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.savefig(f"{model_name}.pdf")

    plt.show()

def accuracy(predicted, actual):
    _, predictions = torch.max(predicted, dim=1)
    return
torch.tensor(torch.sum(predictions==actual).item()/len(predictions))

@torch.no_grad()
def get_all_preds(model, loader):
    all_preds = torch.tensor([])
    for batch in loader:
        images, labels = batch

        preds = model(images)
        all_preds = torch.cat((all_preds, preds) ,dim=0)

    return all_preds

class CNN(nn.Module):
    def __init__(self, output_dim):
        super(CNN, self).__init__()
        self.mlp = nn.Sequential()
        self.mlp.add_module("conv2d1", nn.Conv2d(in_channels=3,
out_channels=32, kernel_size=5, padding='same'))
        self.mlp.add_module("relu", nn.ReLU())
        self.mlp.add_module("conv2d2", nn.Conv2d(in_channels=32,
```

```
out_channels=64, kernel_size=5, padding='same'))
        self.mlp.add_module("relu", nn.ReLU())
        self.mlp.add_module("maxpool2d", nn.MaxPool2d(kernel_size = 2,
stride = 2))
        self.mlp.add_module("conv2d3", nn.Conv2d(in_channels=64,
out_channels=64, kernel_size=5, padding='same'))
        self.mlp.add_module("relu", nn.ReLU())
        self.mlp.add_module("maxpool2d", nn.MaxPool2d(kernel_size = 2,
stride = 2))
        self.mlp.add_module("conv2d4", nn.Conv2d(in_channels=64,
out_channels=64, kernel_size=5, padding='same'))
        self.mlp.add_module("relu", nn.ReLU())
        self.mlp.add_module("conv2d5", nn.Conv2d(in_channels=64,
out_channels=32, kernel_size=5, padding='same'))
        self.mlp.add_module("relu", nn.ReLU())
        self.mlp.add_module("maxpool2d", nn.MaxPool2d(kernel_size = 2,
stride = 2))
        self.mlp.add_module("flatten", nn.Flatten())
        self.mlp.add_module("linear1", nn.Linear(8192, 1024))
        self.mlp.add_module("relu", nn.ReLU())
        self.mlp.add_module("linear2", nn.Linear(1024, 128))
        self.mlp.add_module("relu", nn.ReLU())
        self.mlp.add_module("linear3", nn.Linear(128, output_dim))

    def forward(self, x):
        out = self.mlp(x)
        return out

    def train_loss(self, batch):
        images, labels = batch
        out = self.forward(images)
        loss = F.cross_entropy(out, labels)
        return loss

    def val_loss(self, batch):
        images, labels = batch
        out = self.forward(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return {"val_loss":loss.detach(), "val_acc":acc}

    def val_epoch_end(self, outputs):
        batch_losses = [loss["val_loss"] for loss in outputs]
        loss = torch.stack(batch_losses).mean()
        batch_acc = [acc["val_acc"] for acc in outputs]
        acc = torch.stack(batch_acc).mean()
        return {"val_loss":loss.item(), "val_acc":acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {}, val_loss:
{:.4f}, val_acc: {:.4f}".format(
            epoch, result['lrs'][-1], result['train_loss'],
result['val_loss'], result['val_acc']))
```

```
def pair(t):
    return t if isinstance(t, tuple) else (t, t)

class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm =  nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)

class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )
    def forward(self, x):
        return self.net(x)

class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64, dropout = 0.):
        super().__init__()
        inner_dim = dim_head * heads
        project_out = not (heads == 1 and dim_head == dim)
        self.heads = heads
        self.scale = dim_head**-0.5
        self.attend = nn.Softmax(dim = -1)
        self.dropout = nn.Dropout(dropout)
        self.to_qkv = nn.Linear(dim, inner_dim*3, bias = False)
        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim),
            nn.Dropout(dropout)
        ) if project_out else nn.Identity()

    def forward(self, x):
        qkv = self.to_qkv(x).chunk(3, dim=-1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h =
self.heads), qkv)
        dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale
        attn = self.attend(dots)
        attn = self.dropout(attn)
        out = torch.matmul(attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)')
        return self.to_out(out)

class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout =
0.):
        super().__init__()
```

```
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                PreNorm(dim, Attention(dim, heads=heads,
dim_head=dim_head, dropout=dropout)),
                PreNorm(dim, FeedForward(dim, mlp_dim, dropout=dropout))
            ]))
    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x
        return x
class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth,
heads, mlp_dim, pool='cls', channels=3, dim_head=64, dropout=0.,
emb_dropout=0.):
        super().__init__()
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)
        assert image_height % patch_height == 0 and image_width %
patch_width == 0, 'Image dimensions must be divisible by the patch size.'
        num_patches = (image_height // patch_height) * (image_width //
patch_width)
        patch_dim = channels * patch_height * patch_width
        assert pool in {'cls', 'mean'}, 'pool type must be wither cls (cls
token) or mean (mean pooling)'
        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 =
patch_height, p2 = patch_width),
            nn.LayerNorm(patch_dim),
            nn.Linear(patch_dim, dim),
            nn.LayerNorm(dim),
        )
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches+1,
dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)
        self.transformer = Transformer(dim, depth, heads, dim_head,
mlp_dim, dropout)
        self.pool = pool
        self.to_latent = nn.Identity()
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape
        cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d', b = b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n+1)]
        x = self.dropout(x)
        x = self.transformer(x)
```

```
        x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]
        x = self.to_latent(x)
        return self.mlp_head(x)

    def train_loss(self, batch):
        images, labels = batch
        out = self.forward(images)
        loss = F.cross_entropy(out, labels)
        return loss

    def val_loss(self, batch):
        images, labels = batch
        out = self.forward(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return {"val_loss":loss.detach(), "val_acc":acc}

    def val_epoch_end(self, outputs):
        batch_losses = [loss["val_loss"] for loss in outputs]
        loss = torch.stack(batch_losses).mean()
        batch_acc = [acc["val_acc"] for acc in outputs]
        acc = torch.stack(batch_acc).mean()
        return {"val_loss":loss.item(), "val_acc":acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {}, val_loss:
{:.4f}, val_acc: {:.4f}".format(
            epoch, result['lrs'][-1], result['train_loss'],
result['val_loss'], result['val_acc']))

def evaluate(model, test_dl):
    model.eval()
    outputs = [model.val_loss(batch) for batch in test_dl]
    return model.val_epoch_end(outputs)

def fit(epochs, model, model_name, train_dl, test_dl, optimizer,
scheduler, lr, weight_decay, gamma, step_size):
    torch.cuda.empty_cache()
    history = []
    optimizer = optimizer(model.parameters(), lr=lr,
weight_decay=weight_decay)
    scheduler = scheduler(optimizer,step_size=step_size, gamma=gamma)
    print(model_name, 'fit in...')
    for epoch in range(epochs):
        model.train()
        train_loss = []
        lrs = []
        scheduler.step()
        for batch in train_dl:
            loss = model.train_loss(batch)
            train_loss.append(loss)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
        result = evaluate(model, test_dl)
        lrs.append(scheduler.get_last_lr()[0])
        result['train_loss'] = torch.stack(train_loss).mean().item()
        result['lrs'] = lrs
        model.epoch_end(epoch, result)
        history.append(result)
    return history

def predict(model, test_dl, device):
    model.eval()
    y_true = torch.tensor([] ,dtype=torch.long, device=device)
    y_out = torch.tensor([], device=device)
    with torch.no_grad():
        for data in test_dl:
                inputs = [i.to(device) for i in data[:-1]]
                labels = data[-1].to(device)
                outputs = model(*inputs)
                y_true = torch.cat((y_true, labels), 0)
                y_out = torch.cat((y_out, outputs), 0)
    y_true = y_true.cpu().numpy()
    _, y_pred = torch.max(y_out, 1)
    y_pred = y_pred.cpu().numpy()
    y_pred_prob = F.softmax(y_out, dim=1).cpu().numpy
    return y_true, y_pred, y_pred_prob

epochs = 20
lr = 3e-4
weight_decay = 1e-5
gamma = 0.5
step_size = 10
optimizer = torch.optim.Adam
scheduler = torch.optim.lr_scheduler.StepLR
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck')
num_classes = len(classes)

model_CNN = CNN(num_classes)
model_CNN = model_CNN.to(device)
model_ViT = ViT(dim=128, image_size=32, patch_size=2, depth=6, heads=16,
mlp_dim=256, num_classes=num_classes, channels=3, dropout=0.1,
emb_dropout=0.1)
model_ViT = model_ViT.to(device)
criterion = nn.CrossEntropyLoss()
total_step = len(train_dl)

%%time
cnn_history = fit(epochs, model_CNN, 'CNN', train_dl, test_dl, optimizer,
scheduler, lr, weight_decay, gamma, step_size)

%%time
vit_history = fit(epochs, model_ViT, 'ViT', train_dl, test_dl, optimizer,
scheduler, lr, weight_decay, gamma, step_size)

%%time
```

```
cnn_history = fit(epochs, model_CNN, 'CNN', train_dl_half, test_dl,
optimizer, scheduler, lr, weight_decay, gamma, step_size)

%%time
vit_history = fit(epochs, model_ViT, 'ViT', train_dl_half, test_dl,
optimizer, scheduler, lr, weight_decay, gamma, step_size)

y_true, y_pred, y_pred_prob = predict(model_CNN, test_dl, device)
plot_training_metrics(cnn_history, model_CNN, test_data, y_true, y_pred,
classes)

y_true_v, y_pred_v, y_pred_prob_v = predict(model_ViT, test_dl, device)
plot_training_metrics(vit_history, model_ViT, test_data, y_true_v,
y_pred_v, classes)

epochs = 20
lr = 1e-3
weight_decay = 1e-6
gamma = 0.5
step_size = 5
model_CNN2 = CNN(num_classes)
model_CNN2 = model_CNN2.to(device)
model_ViT2 = ViT(dim=128, image_size=32, patch_size=2, depth=4, heads=16,
mlp_dim=256, num_classes=num_classes, channels=3, dropout=0.1,
emb_dropout=0.1)
model_ViT2 = model_ViT2.to(device)

%%time
cnn2_history = fit(epochs, model_CNN2, 'CNN2', train_dl, test_dl,
optimizer, scheduler, lr, weight_decay, gamma, step_size)

%%time
vit2_history = fit(epochs, model_ViT2, 'ViT2', train_dl, test_dl,
optimizer, scheduler, lr, weight_decay, gamma, step_size)

y_true, y_pred, y_pred_prob = predict(model_CNN, test_dl, device)
plot_training_metrics(cnn_history, model_CNN, test_data, y_true, y_pred,
classes)
y_true_v, y_pred_v, y_pred_prob_v = predict(model_ViT, test_dl, device)
plot_training_metrics(vit_history, model_ViT, test_data, y_true_v,
y_pred_v, classes)
y_true, y_pred, y_pred_prob = predict(model_CNN2, test_dl, device)
plot_training_metrics(cnn2_history, model_CNN2, test_data, y_true_v,
y_pred_v, classes)
y_true, y_pred, y_pred_prob = predict(model_ViT2, test_dl, device)
plot_training_metrics(vit2_history, model_ViT2, test_data, y_true_v,
y_pred_v, classes)
```

Non-image

```
import yfinance as yf
import talib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import pyplot
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import ParameterGrid
from sklearn import metrics
%matplotlib inline

ticker= "2454.TW"
stock_data = yf.download(ticker, start="2018-01-04", end="2022-12-15")
stock_data['Adj Close'].plot()
plt.ylabel("Adjusted Close Prices")
plt.show()

stock_data['Adj Close'].pct_change().plot.hist(bins=50)
plt.xlabel("Adjusted close 1 day percent change")
plt.show()

feature_names = []
for n in [14, 30, 50, 200]:
    stock_data['ma'  + str(n)] = talib.SMA(stock_data['Adj Close'].values,
timeperiod = n)
    stock_data['rsi' + str(n)] = talib.RSI(stock_data['Adj Close'].values,
timeperiod = n)

    feature_names = feature_names + ['ma' + str(n), 'rsi' + str(n)]
stock_data['Volume_1d_change'] = stock_data['Volume'].pct_change()
stock_data['5d_future_close'] = stock_data['Adj Close'].shift(-5)
stock_data['5d_close_future_pct'] =
stock_data['5d_future_close'].pct_change(5)
import seaborn as sns
# Computing the  Correlation Matrix using only the features that have
'mean' in their name.
cols = [col for col in stock_data.columns]
corr = stock_data[cols].corr()

#the matplotlib figure
f, ax = plt.subplots(figsize=(12, 7))
sns.heatmap(corr, vmax=.3, center=0, square=True, linewidths=.5, cbar_kws=
{"shrink": .5})

plt.title('Correlation Heatmap', fontsize = 20)
plt.xticks(fontsize = 10)
plt.yticks(fontsize = 10)
plt.savefig("Corr_Heatmap.png")
stock_data.dropna(inplace = True)
X = stock_data[feature_names]
Y = stock_data['5d_close_future_pct']
```

```
train_size = int(0.90 * Y.shape[0])
X_train = X[:train_size]
y_train = Y[:train_size]
X_test = X[train_size:]
y_test = Y[train_size:]
from sklearn.model_selection import KFold, cross_val_score
kf = KFold(n_splits=10, shuffle=True, random_state=42)
rf_model = RandomForestRegressor(n_estimators = 50, max_depth = 4,
max_features = 4, random_state = 1)
scores = cross_val_score(rf_model, X_train, y_train, cv=kf)
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_test)
y_pred_series = pd.Series(y_pred, index=y_test.index)
y_pred_series.plot()
plt.ylabel("Predicted 5 Day Close Price Change Percent")
plt.show()
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
rf_model = RandomForestRegressor(n_estimators = 20, max_depth = 8,
max_features = 4, random_state = 1)
scores = cross_val_score(rf_model, X_train, y_train, cv=kf)
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_test)
y_pred_series = pd.Series(y_pred, index=y_test.index)
y_pred_series.plot()
plt.ylabel("Predicted 5 Day Close Price Change Percent")
plt.show()
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
rf_model = RandomForestRegressor(n_estimators = 100, max_depth = 4,
max_features = 4, random_state = 1)
scores = cross_val_score(rf_model, X_train, y_train, cv=kf)
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_test)
y_pred_series = pd.Series(y_pred, index=y_test.index)
y_pred_series.plot()
plt.ylabel("Predicted 5 Day Close Price Change Percent")
plt.show()
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
lr_model = LinearRegression()
scores = cross_val_score(rf_model, X_train, y_train, cv=kf)
```

```
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())
lr_model.fit(X_train, y_train)
y_pred = lr_model.predict(X_test)
y_pred_series = pd.Series(y_pred, index=y_test.index)
y_pred_series.plot()
plt.ylabel("Predicted 5 Day Close Price Change Percent")
plt.show()
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
y_test.plot()
plt.ylabel("Real 5 Day Close Price Change Percent")
```

Self-made

```
import pandas as pd
import os
import librosa
import numpy as np

folder_path = 'Music_Data/is_house_music'
# Define the number of MFCC coefficients to extract
n_mfcc = 13
# Define the hop length (in samples)
hop_length = 512
# Get all files in the directory
files = os.listdir(folder_path)
# Filter only mp3 files
mp3_files = [file for file in files if file.endswith(".mp3")]
# Define the CSV file path to save the MFCC features
csv_path = "data.csv"
# Create an empty list to store the MFCC features and their corresponding
labels
features = []
labels = []
print(len(mp3_files),'samples')
idx = 0
for mp3_file in mp3_files:
    mp3_file = folder_path+'/'+mp3_file
    #duration = math.floor(librosa.get_duration(filename=mp3_file))
    y, sr =
librosa.load(mp3_file,sr=librosa.get_samplerate(mp3_file),mono=False,dtype
=np.float32)
    # Apply preprocessing techniques
    y = librosa.effects.trim(y)[0]  # Trim silent sections
    y = librosa.util.normalize(y)  # Normalize the audio data
    mfcc = librosa.feature.mfcc(y=y, sr=sr,n_mfcc=n_mfcc,
hop_length=hop_length)  # Extract MFCC features
    mfcc_mean = np.mean(mfcc, axis=2)
```

```
    mfcc_mean = np.mean(mfcc_mean, axis=0)
    features.append(mfcc_mean)
    labels.append(1)

folder_path = 'Music_Data/not_house_music'
files = os.listdir(folder_path)
mp3_files = [file for file in files if file.endswith(".mp3")]
print(len(mp3_files),'samples')
for mp3_file in mp3_files:
    mp3_file = folder_path+'/'+mp3_file
    #duration = math.floor(librosa.get_duration(filename=mp3_file))
    y, sr =
librosa.load(mp3_file,sr=librosa.get_samplerate(mp3_file),mono=False,dtype
=np.float32)
    # Apply preprocessing techniques
    y = librosa.effects.trim(y)[0]  # Trim silent sections
    y = librosa.util.normalize(y)  # Normalize the audio data
    mfcc = librosa.feature.mfcc(y=y, sr=sr,n_mfcc=n_mfcc,
hop_length=hop_length)  # Extract MFCC features
    mfcc_mean = np.mean(mfcc, axis=2)
    mfcc_mean = np.mean(mfcc_mean, axis=0)
    features.append(mfcc_mean)
    labels.append(0)
# Convert the features and labels lists to numpy arrays
features = np.array(features)
labels = np.array(labels)

# Create a pandas DataFrame from the features and labels arrays
df = pd.DataFrame(features, columns=[f"mfcc{i}" for i in range(1,
n_mfcc+1)])
df["label"] = labels

# Save the DataFrame to a CSV file
csv_path = "data.csv"
df.to_csv(csv_path, index=True)
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
music_data = pd.read_csv('data.csv')
print("Dataset has", music_data.shape)
music_data.head()
# Computing the  Correlation Matrix using only the features that have
'mean' in their name.
cols = [col for col in music_data.columns if 'mfcc' in col]
cols.append('label')
corr = music_data[cols].corr()

#the matplotlib figure
f, ax = plt.subplots(figsize=(12, 7))
```

```
sns.heatmap(corr, vmax=.3, center=0, square=True, linewidths=.5, cbar_kws=
{"shrink": .5})

plt.title('Correlation Heatmap', fontsize = 20)
plt.xticks(fontsize = 10)
plt.yticks(fontsize = 10)
plt.savefig("Corr_Heatmap.png")
from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, confusion_matrix
import itertools
def plot_metrics(y_actual, y_pred, classes, model_name='model'):
    """
    Input: trained model history, model, test image generator, actual and
predicted labels, class list
    Output: Plots loss vs epochs, accuracy vs epochs, confusion matrix
    """
    fpr, tpr, thresholds = metrics.roc_curve(y_actual, y_pred)
    AUC = metrics.auc(fpr, tpr)*100
    Acc = accuracy_score(y_actual, y_pred)*100
    results_title =(f"\n Model AUC {AUC:.2f}%, Accuracy {Acc:.2f}% on Test
Data\n")
    print(results_title.format(AUC, Acc))


    # print classification report
    print(classification_report(y_actual, y_pred, target_names=classes))

     # calculate Confusion Matrix
    cm = confusion_matrix(y_actual, y_pred)

    # create confusion matrix plot
    plt.subplot(1,1,1)
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.BuPu)
    plt.title(f"Confusion Matrix \nAUC: {AUC:.2f}%")
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    # loop through matrix, plot each
    threshold = cm.max() / 2.
    for r, c in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(c, r, format(cm[r, c], 'd'),
                 horizontalalignment="center",
                 color="white" if cm[r, c] > threshold else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.savefig(f"{model_name}.pdf")

    plt.show()
print(music_data.columns)
```

```
music_data = music_data.drop('Unnamed: 0', axis=1)
from sklearn.model_selection import KFold, cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
# Split the data into features (X) and target variable (y)
X = music_data.drop('label', axis=1)
y = music_data['label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# Instantiate a cross-validation object with k=5
kf = KFold(n_splits=10, shuffle=True, random_state=42)

# Instantiate a linear regression model
model = DecisionTreeClassifier(max_depth=5)

# Use cross_val_score to perform cross-validation
scores = cross_val_score(model, X_train, y_train, cv=kf)

# Print the mean and standard deviation of the scores
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())

# Train the model on the training data
model.fit(X_train, y_train)

# Use the model to make predictions on the test data
y_pred = model.predict(X_test)

plot_metrics(y_test, y_pred, classes = ('0','1'), model_name='model')
# Instantiate a linear regression model
model = DecisionTreeClassifier(max_depth=10)

# Use cross_val_score to perform cross-validation
scores = cross_val_score(model, X_train, y_train, cv=kf)

# Print the mean and standard deviation of the scores
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())

# Train the model on the training data
model.fit(X_train, y_train)

# Use the model to make predictions on the test data
y_pred = model.predict(X_test)

plot_metrics(y_test, y_pred, classes = ['0','1'], model_name='model')
#Train another learning algorithm
from sklearn.neighbors import KNeighborsClassifier
# Instantiate a linear regression model
model = KNeighborsClassifier(n_neighbors=1)

# Use cross_val_score to perform cross-validation
scores = cross_val_score(model, X_train, y_train, cv=kf)
```

```
# Print the mean and standard deviation of the scores
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())

# Train the model on the training data
model.fit(X_train, y_train)

# Use the model to make predictions on the test data
y_pred = model.predict(X_test)

plot_metrics(y_test, y_pred, classes = ['0','1'], model_name='model')
# Instantiate a linear regression model
model = KNeighborsClassifier(n_neighbors=10)

# Use cross_val_score to perform cross-validation
scores = cross_val_score(model, X_train, y_train, cv=kf)

# Print the mean and standard deviation of the scores
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())

# Train the model on the training data
model.fit(X_train, y_train)

# Use the model to make predictions on the test data
y_pred = model.predict(X_test)

plot_metrics(y_test, y_pred, classes = ['0','1'], model_name='model')
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
# Instantiate a cross-validation object with k=5
kf = KFold(n_splits=10, shuffle=True, random_state=42)

# Instantiate a linear regression model
model = DecisionTreeClassifier(max_depth=5)

# Use cross_val_score to perform cross-validation
scores = cross_val_score(model, X_train, y_train, cv=kf)

# Print the mean and standard deviation of the scores
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())

# Train the model on the training data
model.fit(X_train, y_train)

# Use the model to make predictions on the test data
y_pred = model.predict(X_test)

plot_metrics(y_test, y_pred, classes = ('0','1'), model_name='model')
#Train another learning algorithm
from sklearn.neighbors import KNeighborsClassifier
# Instantiate a linear regression model
model = KNeighborsClassifier(n_neighbors=1)
```

```
# Use cross_val_score to perform cross-validation
scores = cross_val_score(model, X_train, y_train, cv=kf)

# Print the mean and standard deviation of the scores
print('Training Mean accuracy:', scores.mean())
print('Standard deviation:', scores.std())

# Train the model on the training data
model.fit(X_train, y_train)

# Use the model to make predictions on the test data
y_pred = model.predict(X_test)

plot_metrics(y_test, y_pred, classes = ['0','1'], model_name='model')
```