Intrusion-Detection-Model

1 Mounting Drive

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

2 Installations

[ ]:
```
!pip install tensorflow --upgrade
```

[ ]:
```
!pip install keras --upgrade
```

[ ]:
```
!pip install yellowbrick==1.0.1
```

3 Imports

[ ]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import os
import numpy as np
import pandas as pd
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import sklearn.cluster as cluster
from mpl_toolkits.mplot3d import axes3d
# from random import sample, shuffle
import joblib
from yellowbrick.cluster import KElbowVisualizer
from scipy.stats import zscore, multivariate_normal
import seaborn as sns
```

```
from sklearn.metrics import confusion_matrix, f1_score, make_scorer, \
                            accuracy_score, precision_score, recall_score, \
                            precision_recall_curve, roc_curve, auc
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.utils import shuffle
from keras import Sequential
from keras.layers import Dense, Dropout, Convolution1D, MaxPooling1D, LSTM
from keras.utils import to_categorical
from keras.regularizers import l2
from keras.optimizers import Adam
from keras.utils import plot_model
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE, SMOTENC, SVMSMOTE
from collections import Counter
```

[ ]:
```
import warnings
warnings.filterwarnings('ignore')
```

4 Constants

[ ]:
```
CSV_DIR = '/content/drive/My Drive/Colab Notebooks/Intrusion Detection/data/
 ↪NSL-KDD/csv'
BATCH_SIZE = 128
FIG_TITLE_SIZE = 15
LEGEND_TEXT_SIZE = 11
MODELS_DIR = '/content/drive/My Drive/Colab Notebooks/Intrusion Detection/
 ↪saved_models/grid_search'
```

5 Device Check for GPU

[ ]:
```
device = torch.device('cuda: 0' if torch.cuda.is_available() else 'cpu')
print(device)
```

6 Dataset & DataLoader

6.1 NSLKDDDataset

[ ]:
```
class NSLKDDDataset(Dataset):

    def __init__(self, features, labels):
        self.features = np.asarray(features)
        self.labels = np.asarray(labels)

    def __len__(self):
        return self.features.shape[0]

    def __getitem__(self, idx):
        return (torch.from_numpy(self.features[idx]), torch.from_numpy(self.
 ↪labels[idx]))
```

6.2 NSLKDDFeatures

[ ]:
```
class NSLKDDFeatures(Dataset):

    def __init__(self, features):
        self.features = np.asarray(features)

    def __len__(self):
        return self.features.shape[0]

    def __getitem__(self, idx):
        return torch.from_numpy(self.features[idx])
```

6.3 NSLKDDSiameseNetDataSet

[ ]:
```
class NSLKDDSiameseNetDataSet(Dataset):

    def __init__(self, partitioned_data, indices, labels, nsl_kdd_features):
        self.partitioned_data = partitioned_data
        self.indices = indices
        self.labels = labels
        self.nsl_kdd_features = nsl_kdd_features

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, idx):
```

```
        idx_tuple = self.indices[idx]
        if idx_tuple[0] == idx_tuple[1]:
            ret_list = self.partitioned_data[idx_tuple[0]][
                np.random.choice(
                    self.partitioned_data[idx_tuple[0]].shape[0], 2,␣
↪replace=False
                )
            ]
        else:
            ret_list = list(map(
                lambda x: self.partitioned_data[x][
                    np.random.choice(self.partitioned_data[x].shape[0], 1)[0]
                ],
                idx_tuple)
            )

        return (
            torch.from_numpy(ret_list[0]),
            torch.from_numpy(ret_list[1]),
            torch.tensor(self.labels[idx], dtype=torch.int64)
        )
```

6.4 Training Data

[ ]:
```
train_features_pd = pd.read_csv(os.path.join(CSV_DIR, 'training',␣
↪'train_features_pca.csv'), header=None)
train_labels_multiclass_pd = pd.read_csv(os.path.join(CSV_DIR, 'training',␣
↪'train_labels_multiclass.csv'), header=None)
train_labels_binary_pd = pd.read_csv(os.path.join(CSV_DIR, 'training',␣
↪'train_labels_binary.csv'), header=None)
```

[ ]:
```
train_features_pd.shape
```

[ ]:
```
train_multiclass_ds = NSLKDDDataset(train_features, train_labels_multiclass)
train_binary_ds = NSLKDDDataset(train_features, train_labels_binary)
```

[ ]:
```
train_features_ds = NSLKDDFeatures(train_features)
```

[ ]:
```
train_multiclass_dl = DataLoader(
    train_multiclass_ds,
    batch_size=BATCH_SIZE,
    shuffle=True
)

train_binary_dl = DataLoader(
```

```
    train_binary_ds,
    batch_size=BATCH_SIZE,
    shuffle=True
)

train_features_dl = DataLoader(
    train_features_ds,
    batch_size=BATCH_SIZE,
    shuffle=True
)
```

6.5 Testing Data

```
[ ]: test_features_pd = pd.read_csv(os.path.join(CSV_DIR, 'testing',␣
     ↪'test_features_pca.csv'), header=None)
     test_labels_multiclass_pd = pd.read_csv(os.path.join(CSV_DIR, 'testing',␣
     ↪'test_labels_multiclass.csv'), header=None)
     test_labels_binary_pd = pd.read_csv(os.path.join(CSV_DIR, 'testing',␣
     ↪'test_labels_binary.csv'), header=None)
```

```
[ ]: test_features_pd.shape
```

```
[ ]: test_multiclass_ds = NSLKDDDataset(test_features, test_labels_multiclass)
     test_binary_ds = NSLKDDDataset(test_features, test_labels_binary)
```

```
[ ]: test_features_ds = NSLKDDFeatures(test_features)
```

```
[ ]: test_multiclass_dl = DataLoader(
         test_multiclass_ds,
         batch_size=BATCH_SIZE,
         shuffle=True
     )

     test_binary_dl = DataLoader(
         test_binary_ds,
         batch_size=BATCH_SIZE,
         shuffle=True
     )

     test_features_dl = DataLoader(
         test_features_ds,
         batch_size=BATCH_SIZE,
         shuffle=True
     )
```

6.6 NumPy Arrays

[ ]:
```
train_features_np = np.asarray(train_features_pd)
test_features_np = np.asarray(test_features_pd)

train_labels_binary_np = np.asarray(train_labels_binary_pd)
test_labels_binary_np = np.asarray(test_labels_binary_pd)

train_labels_multiclass_np = np.asarray(train_labels_multiclass_pd)
test_labels_multiclass_np = np.asarray(test_labels_multiclass_pd)
```

7 Data Plots

7.1 Plot Pie Chart

[ ]:
```
def plot_pie_chart(labels_one_hot, title, figure_size=(8,8), have_title=True,␣
 ↪log_scaling=False):
    labels = np.argmax(labels_one_hot, axis=1)
    names = ['Normal', 'DoS', 'Probe', 'R2L', 'U2R']
    _, sizes = np.unique(labels, return_counts=True)

    names_sizes_dict = dict(zip(names, sizes))
    sorted_names_sizes_dict = {k: v for k, v in sorted(
        names_sizes_dict.items(),
        key=lambda item: item[1],
        reverse=True
    )}


    names = list(sorted_names_sizes_dict.keys())
    sizes = list(sorted_names_sizes_dict.values())

    if log_scaling:
        sizes = np.log(sizes)

    fig = plt.figure(figsize=figure_size)
    ax = fig.add_subplot()

    theme = plt.get_cmap('gray')
    ax.set_prop_cycle('color', [theme(1. * i / len(sizes))
                                for i in range(len(sizes))])

    patches, texts, autotexts = ax.pie(
        sizes,
        autopct='%1.1f%%',
```

```
        startangle=90
    )

    autotexts[0].set_color('w')
    autotexts[1].set_color('w')

    # for text in autotexts:
    #     text.set_fontsize(11)

    ax.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

    if have_title:
        plt.title(
            title,
            fontsize=LEGEND_TEXT_SIZE
        )

    plt.legend(
        names,
        fontsize=11
    )

    plt.savefig(title + '.png')
```

[ ]:
```
plot_pie_chart(
    train_labels_multiclass_np,
    title='Class Distributions: Training Data (log scaled)',
    figure_size=(10,10),
    have_title=False,
    log_scaling=True
)
```

[ ]:
```
plot_pie_chart(
    test_labels_multiclass_np,
    title='Class Distributions: Testing Data (log scaled)',
    figure_size=(10,10),
    have_title=False,
    log_scaling=True
)
```

8 Model

8.1 Autoencoder

[ ]:
```
class Autoencoder(nn.Module):

    def __init__(self, in_features, out_features):
        super(Autoencoder, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(in_features=in_features, out_features=out_features),
            nn.Sigmoid()
        )

        self.decoder = nn.Sequential(
            nn.Linear(in_features=out_features, out_features=in_features),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)

        return x
```

8.2 AnomalyDetector

[ ]:
```
class AnomalyDetector(nn.Module):

    def __init__(self):
        super(AnomalyDetector, self).__init__()

        self.activation = nn.LeakyReLU(inplace=True)

        self.model = nn.Sequential(
            # nn.BatchNorm1d(num_features=67),
            nn.Linear(in_features=67, out_features=100),
            self.activation,

            nn.BatchNorm1d(num_features=100),
            nn.Linear(in_features=100, out_features=120),
            self.activation,

            nn.BatchNorm1d(num_features=120),
            nn.Linear(in_features=120, out_features=60),
```

```
            self.activation,

            nn.BatchNorm1d(num_features=60),
            nn.Linear(in_features=60, out_features=30),
            self.activation,

            nn.BatchNorm1d(num_features=30),
            nn.Linear(in_features=30, out_features=10),
            self.activation,

            nn.BatchNorm1d(num_features=10),
            nn.Linear(in_features=10, out_features=2),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.model(x)

        return x
```

8.3 SignatureGenerator

[ ]:
```
class SignatureGenerator(nn.Module):

    def __init__(self):
        super(SignatureGenerator, self).__init__()

        self.activation = nn.LeakyReLU(inplace=True)

        self.model = nn.Sequential(
            nn.Linear(in_features=67, out_features=100),
            self.activation,
            nn.BatchNorm1d(num_features=100),

            nn.Linear(in_features=100, out_features=80),
            self.activation,
            nn.BatchNorm1d(num_features=80),

            nn.Linear(in_features=80, out_features=60),
            self.activation,
            nn.BatchNorm1d(num_features=60),

            nn.Linear(in_features=60, out_features=40),
            self.activation,
            nn.BatchNorm1d(num_features=40),
```

```
            nn.Linear(in_features=40, out_features=20),
            self.activation,
            nn.BatchNorm1d(num_features=20),

            nn.Linear(in_features=20, out_features=10),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.model(x)

        return x
```

8.4 L1Difference

[ ]:
```
class L1Difference(nn.Module):

    def __init__(self):
        super(L1Difference, self).__init__()

    def forward(self, x1, x2):
        return abs(x1 - x2)
```

8.5 SiameseNet

[ ]:
```
class SiameseNet(nn.Module):

    def __init__(self):
        super(SiameseNet, self).__init__()

        self.signature_gen = SignatureGenerator()
        self.l1_diff = L1Difference()
        self.fc = nn.Linear(in_features=10, out_features=2)

    def forward(self, x1, x2):
        x1_sig = self.signature_gen(x1)
        x2_sig = self.signature_gen(x2)

        l1_diff = self.l1_diff(x1_sig, x2_sig)

        out = torch.softmax(self.fc(l1_diff), dim=1)

        return out
```

8.6 MultiClassClassifier

[ ]:
```
class MultiClassClassifier(nn.Module):

    def __init__(self):
        super(MultiClassClassifier, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(in_features=67, out_features=120),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(num_features=120),

            nn.Linear(in_features=120, out_features=100),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(num_features=100),

            nn.Linear(in_features=100, out_features=80),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(num_features=80),

            nn.Linear(in_features=80, out_features=40),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(num_features=40),

            nn.Linear(in_features=40, out_features=20),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(num_features=20),

            nn.Linear(in_features=20, out_features=10),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(num_features=10),

            nn.Linear(in_features=10, out_features=5),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        return self.model(x)
```

9 Training Utilities

9.1 Network Loss

[ ]:
```
def get_network_loss(net, dataloader):
    agg_loss = 0.0

    criterion = nn.CrossEntropyLoss()

    for i, data in enumerate(dataloader, 0):
        x, y = data
        y = torch.argmax(y, dim=1)

        x = x.to(device)
        y = y.to(device)

        outputs = net(x)
        loss = criterion(outputs, y)
        agg_loss += loss.item()

    return agg_loss / len(dataloader)
```

9.2 Train Network

[ ]:
```
def train_network(net, trainloader, testloader, n_epochs, print_every=500):
    net.double()
    net.to(device)

    optimizer = optim.Adam(net.parameters())
    criterion = nn.CrossEntropyLoss()

    train_losses = []
    test_losses = []
    for epoch in range(n_epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader):
            x, y = data
            y = torch.argmax(y, dim=1)

            x = x.to(device)
            y = y.to(device)

            optimizer.zero_grad()
            outputs = net(x)
            loss = criterion(outputs, y)
```

```
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if i % print_every == print_every-1:
                print('[%d, %5d] loss: %.6f' %
                  (epoch + 1, i + 1, running_loss / print_every))
                running_loss = 0.0

        net.eval()
        train_loss = get_network_loss(net, trainloader)
        test_loss = get_network_loss(net, testloader)
        net.train()

        train_losses.append(train_loss)
        test_losses.append(test_loss)

        print('Training Loss: %.6f' % (train_loss))
        print('Testing Loss: %.6f' % (test_loss))

    return train_losses, test_losses
```

9.3 Encoder Loss

```
[ ]: def get_encoder_loss(autoencoder, dataloader, criterion, autoencoder1=None,␣
      ↪autoencoder2=None):
         agg_loss = 0.0

         for i, data in enumerate(dataloader, 0):
             x = data
             x = x.to(device)

             if autoencoder1:
                 x = autoencoder1.encoder(x)
             if autoencoder2:
                 x = autoencoder2.encoder(x)

             outputs = autoencoder(x)
             loss = criterion(outputs, x)
             agg_loss += loss.item()

         return agg_loss / len(dataloader)
```

9.4 Train Encoder

[ ]:
```
def train_encoder(autoencoder, trainloader, testloader, n_epochs,␣
↪print_every=500, autoencoder1=None, autoencoder2=None):
    autoencoder.double()
    autoencoder.to(device)

    optimizer = optim.Adam(autoencoder.parameters())
    criterion = nn.MSELoss()

    train_losses = []
    test_losses = []
    for epoch in range(n_epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            x = data
            x = x.to(device)

            with torch.no_grad():
                if autoencoder1:
                    x = autoencoder1.encoder(x)
                if autoencoder2:
                    x = autoencoder2.encoder(x)

            optimizer.zero_grad()
            outputs = autoencoder(x)
            loss = criterion(outputs, x)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if i % print_every == print_every-1:
                print('[%d, %5d] loss: %.6f' %
                  (epoch + 1, i + 1, running_loss / print_every))
                running_loss = 0.0

        autoencoder.eval()
        train_loss = get_encoder_loss(autoencoder, trainloader, criterion,␣
↪autoencoder1=autoencoder1, autoencoder2=autoencoder2)
        test_loss = get_encoder_loss(autoencoder, testloader, criterion,␣
↪autoencoder1=autoencoder1, autoencoder2=autoencoder2)
        autoencoder.train()

        train_losses.append(train_loss)
        test_losses.append(test_loss)

        print('Training Loss: %.6f' % (train_loss))
```

```
        print('Testing Loss: %.6f' % (test_loss))

    return train_losses, test_losses
```

9.5 SiameseNet Loss

```
[ ]: def get_siamese_net_loss(siamese_net, dataloader, val_batches):
         agg_loss = 0.0
         criterion = nn.CrossEntropyLoss()

         for i, data in enumerate(dataloader, 0):
             x1, x2, y = data

             x1 = x1.to(device)
             x2 = x2.to(device)
             y = y.to(device)

             outputs = siamese_net(x1.double(), x2.double())
             loss = criterion(outputs, y)
             agg_loss += loss.item()

             if i == val_batches-1:
                 break

         return agg_loss / val_batches
```

9.6 Train SiameseNet

```
[ ]: def train_siamese_net(siamese_net, trainloader, testloader, n_epochs,␣
      ↪print_every=500, val_batches=1):
         siamese_net.double()
         siamese_net.to(device)

         optimizer = optim.Adam(siamese_net.parameters())
         criterion = nn.CrossEntropyLoss()

         train_losses = []
         test_losses = []
         for epoch in range(n_epochs):
             running_loss = 0.0
             for i, data in enumerate(trainloader):
                 x1, x2, y = data

                 x1 = x1.to(device)
```

```
            x2 = x2.to(device)
            y = y.to(device)

            optimizer.zero_grad()
            outputs = siamese_net(x1.double(), x2.double())
            outputs = torch.squeeze(outputs)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if i % print_every == print_every-1:
                print('[%d, %5d] loss: %.6f' %
                  (epoch + 1, i + 1, running_loss / print_every))
                running_loss = 0.0

        siamese_net.eval()
        train_loss = get_siamese_net_loss(siamese_net, trainloader, val_batches)
        test_loss = get_siamese_net_loss(siamese_net, testloader, val_batches)
        siamese_net.train()

        train_losses.append(train_loss)
        test_losses.append(test_loss)

        print('Training Loss: %.6f' % (train_loss))
        print('Testing Loss: %.6f' % (test_loss))

    return train_losses, test_losses
```

9.7 Partition Dataset

[ ]:
```
def partition_dataset(data, labels):
    unique_labels = np.unique(labels)
    final_data = [None] * len(unique_labels)
    n_samples = len(labels)
    for i in range(n_samples):
        curr_class = labels[i]
        if final_data[curr_class]:
            final_data[curr_class].append(data[i])
        else:
            final_data[curr_class] = [data[i]]

    return list(map(lambda x: np.asarray(x), final_data))
```

9.8 Plotting Losses

[ ]:
```
def plot_losses(train_losses, test_losses, title, xlabel, ylabel, start_idx=1,␣
↪interval_length=1, figure_size=(14,14)):
    train_losses_aggregated = []
    test_losses_aggregated = []
    n_epochs = len(train_losses)

    start_idx -= 1

    i = start_idx
    while i < n_epochs:
        curr_train_sum = sum(train_losses[i : min(n_epochs, i +␣
↪interval_length)])
        curr_test_sum = sum(test_losses[i : min(n_epochs, i + interval_length)])

        train_losses_aggregated.append(curr_train_sum / interval_length)
        test_losses_aggregated.append(curr_test_sum / interval_length)

        i += interval_length

    n = len(train_losses_aggregated)
    x = list(map(lambda x: (x * interval_length) + start_idx, np.arange(1, n+1)))

    plt.figure(figsize=figure_size)
    plt.plot(x, train_losses_aggregated, marker='o', linestyle='dashed')
    plt.plot(x, test_losses_aggregated, marker='o', linestyle='dashed')
    plt.legend(['Training Loss', 'Testing Loss'])
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.grid()
    plt.xticks(x)

    plt.savefig(title + '.png')
    plt.show()
```

9.9 Saving and Loading Models

[ ]:
```
def save_state_dict(model, filename):
    torch.save(model.state_dict(), filename)
```

[ ]:
```
def load_state_dict(model, state_dict_path):
    state_dict = torch.load(state_dict_path)
    model.load_state_dict(state_dict)
```

```
    model.double()
    model.to(device)
```

10 Training

10.1 Autoencoders

10.1.1 Label Encoding

Autoencoder 1

[ ]:
```
autoencoder_label_1 = Autoencoder(in_features=40, out_features=20)
print(autoencoder_label_1)
```

[ ]:
```
train_losses_encoder_label_1, test_losses_encoder_label_1 = train_encoder(
    autoencoder_label_1,
    trainloader=train_label_loader,
    testloader=test_label_loader,
    n_epochs=30
)
```

[ ]:
```
plot_losses(
    train_losses_encoder_label_1,
    test_losses_encoder_label_1,
    # start_idx=11,
    title='autoencoder_label_1',
    xlabel='Epochs',
    ylabel='Losses',
    interval_length=1,
    figure_size=(14,8)
)
```

[ ]:
```
save_state_dict(autoencoder_label_1, 'autoencoder_label_1.pth')
```

Autoencoder 2

[ ]:
```
autoencoder_label_2 = Autoencoder(in_features=20, out_features=10)
print(autoencoder_label_2)
```

[ ]:
```
train_losses_encoder_label_2, test_losses_encoder_label_2 = train_encoder(
    autoencoder_label_2,
    autoencoder1=autoencoder_label_1,
    trainloader=train_label_loader,
    testloader=test_label_loader,
    n_epochs=30
)
```

[ ]:
```
plot_losses(
    train_losses_encoder_label_2,
    test_losses_encoder_label_2,
    # start_idx=11,
    title='autoencoder_label_2',
    xlabel='Epochs',
    ylabel='Losses',
    interval_length=1,
    figure_size=(14,8)
)
```

[ ]:
```
save_state_dict(autoencoder_label_2, 'autoencoder_label_2.pth')
```

10.2 Networks

[ ]:
```
LOAD_DIR = '/content/drive/My Drive/Colab Notebooks/Intrusion Detection/
 ↪saved_models'
```

10.2.1 Loading Autoencoder 1

[ ]:
```
autoencoder_label_1 = Autoencoder(in_features=40, out_features=20)
load_state_dict(autoencoder_label_1, os.path.join(LOAD_DIR, 'autoencoder_label_1.
 ↪pth'))
```

10.2.2 Loading Autoencoder 2

[ ]:
```
autoencoder_label_2 = Autoencoder(in_features=20, out_features=10)
load_state_dict(autoencoder_label_2, os.path.join(LOAD_DIR, 'autoencoder_label_2.
 ↪pth'))
```

10.2.3 AnomalyDetector

[ ]:
```
anomaly_detector = AnomalyDetector()

print(anomaly_detector)
```

[ ]:
```
train_losses_anomaly_detector, test_losses_anomaly_detector = train_network(
    anomaly_detector,
    trainloader=train_binary_dl,
    testloader=test_binary_dl,
    n_epochs=50,
```

```
    print_every=100
)
```

[ ]:
```
plot_losses(
    train_losses_network_label,
    test_losses_network_label,
    # start_idx=37,
    title='network_label_double_encoder',
    xlabel='Epochs',
    ylabel='Losses',
    interval_length=1,
    figure_size=(14,8))
```

10.3 Data Distributions

10.3.1 PCA

[ ]:
```
pca = PCA(n_components=3)
pca.fit(train_features_np)
```

[ ]:
```
train_features_3dims = pca.transform(train_features_np)
test_features_3dims = pca.transform(test_features_np)
```

10.3.2 Groups

[ ]:
```
groups_multiclass = [
    'Normal',
    'DoS',
    'Probe',
    'R2L',
    'U2R'
]

groups_binary = [
    'Normal',
    'Malicious'
]

groups_malicious_classes = [
    'DoS',
    'Probe',
    'R2L',
    'U2R'
]
```

10.3.3 Plot Points

[ ]:
```
def plot_points(
        features,
        labels,
        title,
        groups,
        class_idx=None,
        one_hot_labels=True,
        figure_size=(8,8)
    ):

    fig = plt.figure(figsize=figure_size)
    ax = fig.add_subplot(111, projection='3d')

    if one_hot_labels:
        labels_indices = np.asarray(list(map(lambda x: np.argmax(x), labels)))
    else:
        labels_indices = labels

    data = []
    if class_idx:
        i = class_idx
        curr_data_indices = np.where(labels_indices == i)[0]
        curr_data = features[curr_data_indices]
        data.append(curr_data)
    else:
        for i in range(len(groups)):
            curr_data_indices = np.where(labels_indices == i)[0]
            curr_data = features[curr_data_indices]
            data.append(curr_data)

    all_colors = (
        'grey',
        'orange',
        'cyan',
        'green',
        'maroon',
        'chocolate',
        'lightgreen',
        'dodgerblue',
        'darkslategray',
        'lightseagreen',
        'mediumspringgreen',
        'rebeccapurple',
        'hotpink',
        'indigo',
```

```
        'midnightblue',
        'gold',
        'black',
        'crimson',
        'darkkhaki',
        'aqua'
    )

    colors = sample(all_colors, len(groups))

    for features, color, group in zip(data, colors, groups):
        x = list(map(lambda x: x[0], features))
        y = list(map(lambda x: x[1], features))
        z = list(map(lambda x: x[2], features))
        ax.scatter(
            x, y, z,
            c=color,
            label=group,
        )

    plt.legend()
    plt.title(
        title,
        fontsize=15,
        loc='left'
    )

    plt.savefig(title + '.png')
```

10.3.4 Training Data Plots

Binary Class Distribution

[ ]:
```
plot_points(
    train_features_3dims,
    train_labels_binary_np,
    groups=groups_binary,
    title='Binary Class Distribution',
    figure_size=(14,14)
)
```

Multi Class Distribution

[ ]:
```
plot_points(
    train_features_3dims,
    train_labels_multiclass_np,
```

```
    groups=groups_multiclass,
    title='Multi Class Distribution',
    figure_size=(14,14)
)
```

Malicious Data Point Distributions

DoS

[ ]:
```
plot_points(
    train_features_3dims,
    train_labels_multiclass_np,
    class_idx=1,
    groups=['DoS'],
    title='DoS',
    figure_size=(14,14)
)
```

Probe

[ ]:
```
plot_points(
    train_features_3dims,
    train_labels_multiclass_np,
    class_idx=2,
    groups=['Probe'],
    title='Probe',
    figure_size=(14,14)
)
```

R2L

[ ]:
```
plot_points(
    train_features_3dims,
    train_labels_multiclass_np,
    class_idx=3,
    groups=['R2L'],
    title='R2L',
    figure_size=(14,14)
)
```

U2R

[ ]:
```
plot_points(
    train_features_3dims,
    train_labels_multiclass_np,
    class_idx=4,
```

```
    groups=['U2R'],
    title='U2R',
    figure_size=(14,14)
)
```

10.4 Anomaly Detection

10.4.1 Utilities

Plot Confusion Matrix

[ ]:
```
def plot_confusion_matrix(
        cm, classes,
        normalize=False,
        title='Confusion Matrix',
        cmap=plt.cm.Blues,
        print_cm=False,
        save=False
    ):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """

    import matplotlib.pyplot as plt
    import numpy as np
    import itertools

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    if print_cm:
        print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    # plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
```

```
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True')
    plt.xlabel('Predicted')
    # plt.xticks([])
    # plt.yticks([])
    plt.grid(False)

    if save:
        plt.savefig(title + '.png')
```

One Hot to Label Encoding

```
[ ]: def to_label(arr):
         return np.argmax(arr, axis=1)
```

Get Predictions

```
[ ]: def get_predictions(estimator, features, labels):
         return to_label(labels), estimator.predict(features), estimator.
      ↪predict_proba(features)
```

Get Scores

```
[ ]: def get_scores(y_true, y_pred):
         accuracy = accuracy_score(y_true, y_pred)
         precision = precision_score(y_true, y_pred)
         recall = recall_score(y_true, y_pred)
         f1 = f1_score(y_true, y_pred)

         return {
             'accuracy': accuracy,
             'precision': precision,
             'recall': recall,
             'f1': f1
         }
```

Get Adjusted Classes

```
[ ]: def get_adjusted_classes(y_probs, thresh=0.5):
         return [int(y_prob >= thresh) for y_prob in y_probs]
```

Plot ROC Curve

```
[ ]: def plot_roc_curve(fpr, tpr, title, label=None, figure_size=(8,8)):
         """
         The ROC curve, modified from
         Hands-On Machine learning with Scikit-Learn and TensorFlow; p.91
         """
         plt.figure(figsize=figure_size)
         plt.title('ROC Curve')
         plt.plot(fpr, tpr, linewidth=2, label=label)
         plt.plot([0, 1], [0, 1], 'k--')
         plt.axis([-0.005, 1, 0, 1.005])
         plt.xticks(np.arange(0,1, 0.05), rotation=90)
         plt.xlabel("False Positive Rate")
         plt.ylabel("True Positive Rate (Recall)")
         plt.legend(loc='best')

         plt.savefig(title + '.png')
```

Plot Precision Vs Recall

```
[ ]: def plot_precision_recall_curve(p, r, y_scores, y_test, thresholds, title, t=0.
       ↪5, figure_size=(8,8)):
         """
         Plots the precision recall curve and shows the current value for each
         by identifying the classifier's threshold (t).
         """

         # generate new class predictions based on the adjusted_classes
         # function above and view the resulting confusion matrix.
         y_pred_adj = get_adjusted_classes(y_scores, t)

         # plot the curve
         plt.figure(figsize=figure_size)
         plt.title(title)
         plt.step(r, p, color='b', alpha=0.2,
                  where='post')
         plt.fill_between(r, p, step='post', alpha=0.2,
                          color='b')
         plt.ylim([0.5, 1.01]);
         plt.xlim([0.5, 1.01]);
         plt.xlabel('Recall');
         plt.ylabel('Precision');

         # plot the current threshold on the line
         close_default_clf = np.argmin(np.abs(thresholds - t))
         plt.plot(r[close_default_clf] - 0.01, p[close_default_clf], '.', c='k',
                 markersize=15)
         plt.text(r[close_default_clf], p[close_default_clf], 'Current Threshold')
```

```
    plt.savefig(title + '.png')
```

Plot Precision and Recall Vs Threshold

```
[ ]: def plot_precision_recall_vs_threshold(precisions, recalls, thresholds, title,␣
      ↪figure_size=(8,8)):
         """
         Modified from:
         Hands-On Machine learning with Scikit-Learn
         and TensorFlow; p.89
         """
         idx = np.argmin(abs(precisions - recalls))
         plt.figure(figsize=figure_size)
         plt.title(title)
         plt.plot(thresholds, precisions[:-1], "k--", label="Precision")
         plt.plot(thresholds, recalls[:-1], "k-", label="Recall")
         plt.plot(thresholds[idx], precisions[idx], 'ro')
         plt.text(thresholds[idx] + 0.01, precisions[idx], f'Threshold =␣
      ↪{thresholds[idx]}')
         plt.ylabel("Score")
         plt.xlabel("Decision Threshold")
         plt.legend(loc='best')

         plt.savefig(title + '.png')

         return thresholds[idx]
```

10.4.2 Clustering Normal Points

```
[ ]: plot_points(
         train_features_3dims,
         train_labels_multiclass_np,
         class_idx=0,
         groups=['Normal'],
         title='Normal',
         figure_size=(14,14)
     )
```

```
[ ]: train_labels = np.asarray(list(map(lambda x: np.argmax(x),␣
      ↪train_labels_binary_np)))
     train_indices_normal = np.where(train_labels == 0)[0]
     train_features_normal = train_features_np[train_indices_normal]
```

```
test_labels = np.asarray(list(map(lambda x: np.argmax(x),␣
 ↪test_labels_binary_np)))
test_indices_normal = np.where(test_labels == 0)[0]
test_features_normal = test_features_np[test_indices_normal]
```

[ ]:
```
train_features_normal.shape
```

[ ]:
```
test_features_normal.shape
```

K-Means

Finding Optimal Clusters using K-Elbow Visualizer

[ ]:
```
model = cluster.KMeans(random_state=42)
visualizer = KElbowVisualizer(
    model,
    k=(2,11),
    metric='calinski_harabasz'
)

visualizer.fit(train_features_normal)
```

Fitting

[ ]:
```
N_CLUSTERS = 4
```

[ ]:
```
kmeans_normal = cluster.KMeans(n_clusters=N_CLUSTERS, random_state=42)
kmeans_normal.fit(train_features_normal)
```

[ ]:
```
kmeans_normal.inertia_
```

[ ]:
```
train_labels_normal = kmeans_normal.predict(train_features_normal)
```

[ ]:
```
plot_points(
    features=pca.transform(train_features_normal),
    labels=train_labels_normal,
    groups=['Cluster ' + str(i+1) for i in range(N_CLUSTERS)],
    one_hot_labels=False,
    title='K-Means Clustering on Normal Data Points - Training',
    figure_size=(14,14)
)
```

[ ]:
```
test_labels_normal = kmeans_normal.predict(test_features_normal)
```

```
[ ]: plot_points(
         features=pca.transform(test_features_normal),
         labels=test_labels_normal,
         groups=['Cluster ' + str(i+1) for i in range(N_CLUSTERS)],
         one_hot_labels=False,
         title='K-Means Clustering on Normal Data Points - Testing',
         figure_size=(14,14)
     )
```

```
[ ]: joblib.dump(kmeans_normal, 'kmeans_normal.joblib')
```

Partitioning Dataset

```
[ ]: partitioned_training_data_normal = partition_dataset(train_features_normal,␣
      ↪train_labels_normal)
```

```
[ ]: for data in partitioned_training_data_normal:
         print(data.shape)
```

```
[ ]: partitioned_testing_data_normal = partition_dataset(test_features_normal,␣
      ↪test_labels_normal)
```

```
[ ]: for data in partitioned_testing_data_normal:
         print(data.shape)
```

SiameseNet Generating Data for NSLKDDSiameseNetDataset Instance

```
[ ]: TRAIN_MULT_FACTOR = 100000
     TEST_MULT_FACTOR = 20000
```

```
[ ]: indices = []
     for i in range(N_CLUSTERS):
         for j in range(i, N_CLUSTERS):
             indices.append((i,j))

     train_indices_final = indices * TRAIN_MULT_FACTOR
     shuffle(train_indices_final)
     test_indices_final = indices * TEST_MULT_FACTOR
     shuffle(test_indices_final)
```

```
[ ]: len(train_indices_final), len(test_indices_final)
```

```
[ ]: train_labels = list(map(lambda x: int(x[0] == x[1]), train_indices_final))
     test_labels = list(map(lambda x: int(x[0] == x[1]), test_indices_final))
```

```
[ ]: len(train_labels), len(test_labels)
```

NSLKDDSiameseNetDataset Instance

```
[ ]: train_data_siamese_net = NSLKDDSiameseNetDataSet(
         partitioned_training_data_normal,
         train_indices_final,
         np.asarray(train_labels),
         train_features
     )
```

```
[ ]: trainloader_siamese_net = DataLoader(
         train_data_siamese_net,
         batch_size=256,
         shuffle=True
     )
```

```
[ ]: test_data_siamese_net = NSLKDDSiameseNetDataSet(
         partitioned_testing_data_normal,
         test_indices_final,
         np.asarray(test_labels),
         train_features
     )
```

```
[ ]: testloader_siamese_net = DataLoader(
         test_data_siamese_net,
         batch_size=256,
         shuffle=True
     )
```

Training Signatures for Each Cluster

```
[ ]: siamese_net = SiameseNet()
     print(siamese_net)
```

```
[ ]: train_losses_siamese_net, test_losses_siamese_net = train_siamese_net(
         siamese_net,
         trainloader=trainloader_siamese_net,
         testloader=testloader_siamese_net,
         n_epochs=50
     )
```

```
[ ]:
```

Z-Score

```
[ ]: curr_cluster = partitioned_training_data_normal[0]
```

```
[ ]: curr_cluster.shape
```

```
[ ]: z_scores = zscore(curr_cluster)
```

```
[ ]: z_scores.shape
```

```
[ ]: sns.kdeplot(z_scores[:,5])
```

10.4.3 Random Forest Classifier

Grid Search

Recall

```
[ ]: rfc = RandomForestClassifier(
         class_weight='balanced',
         max_depth=58,
         min_samples_leaf=2
     )

     param_grid = {
         'n_estimators': np.arange(100, 120),
     }

     rfc_grid_search_recall = GridSearchCV(
         estimator=rfc,
         param_grid=param_grid,
         scoring=make_scorer(recall_score),
         verbose=2,
         cv=5,
         n_jobs=5
     )
```

```
[ ]: rfc_grid_search_recall.fit(train_features_np, to_label(train_labels_binary_np))
```

```
[ ]: rfc_grid_search_recall.best_score_
```

```
[ ]: rfc_grid_search_recall.best_params_
```

```
[ ]: joblib.dump(rfc_grid_search_recall, 'rfc_grid_search_recall.joblib')
```

Model

```
[ ]: rfc_grid_search_recall = joblib.load(os.path.join(MODELS_DIR,␣
       ↪'rfc_grid_search_recall.joblib'))
```

```
[ ]: rfc = rfc_grid_search_recall.best_estimator_

     y_true, y_pred, y_probs = get_predictions(
         rfc,
         test_features_np,
         test_labels_binary_np
     )
```

```
[ ]: cm = confusion_matrix(
         y_pred=y_pred,
         y_true=y_true
     )
```

```
[ ]: plot_confusion_matrix(
         cm,
         classes=[0,1],
         title='Random Forest Classifier with Optimized Recall',
         save=True,
         normalize=True,
         cmap=plt.cm.Greys
     )
```

```
[ ]: p, r, thresholds = precision_recall_curve(y_true, y_probs[:,1])
```

```
[ ]: best_threshold_rfc = plot_precision_recall_vs_threshold(
         p, r, thresholds,
         title='Precision and Recall Scores as a Function of the Decision Threshold',
     )
```

```
[ ]: plot_precision_recall_curve(
         p, r, y_probs[:,1], y_true, thresholds,
         title='Precision Recall Curve',
         t=best_threshold_rfc,
     )
```

```
[ ]: fpr, tpr, auc_thresholds = roc_curve(y_true, y_probs[:,1])
```

```
[ ]: auc(fpr, tpr)
```

```
[ ]: plot_roc_curve(
         fpr, tpr,
         title='ROC Curve',
         label='Recall Optimized'
     )
```

```
[ ]: y_pred_adjusted = get_adjusted_classes(y_probs[:,1], thresh=best_threshold_rfc)
```

```
[ ]: cm = confusion_matrix(
         y_pred=y_pred_adjusted,
         y_true=y_true
     )
```

```
[ ]: plot_confusion_matrix(
         cm,
         classes=[0,1],
         title='Random Forest Classifier with Optimized Recall',
         save=True,
         normalize=True,
         cmap=plt.cm.Greys
     )
```

10.4.4 AdaBoost Classifier

Grid Search

Recall

```
[ ]: adaboost_clf = AdaBoostClassifier()

     param_grid = {
         'n_estimators': np.arange(50, 70)
     }

     adaboost_clf_grid_search_recall = GridSearchCV(
         estimator=adaboost_clf,
         param_grid=param_grid,
         scoring=make_scorer(recall_score),
         verbose=2,
         cv=5,
         n_jobs=5
     )
```

```
[ ]: adaboost_clf_grid_search_recall.fit(train_features_np,␣
      ↪to_label(train_labels_binary_np))
```

```
[ ]: joblib.dump(adaboost_clf_grid_search_recall, 'adaboost_clf_grid_search_recall.
      ↪joblib')
```

Model

```
[ ]: adaboost_clf_grid_search_recall = joblib.load(os.path.join(MODELS_DIR,␣
      ↪'adaboost_clf_grid_search_recall.joblib'))
```

[ ]:
```
adaboost_clf = adaboost_clf_grid_search_recall.best_estimator_

y_true, y_pred, y_probs = get_predictions(
    adaboost_clf,
    test_features_np,
    test_labels_binary_np
)
```

[ ]:
```
cm = confusion_matrix(
    y_pred=y_pred,
    y_true=y_true
)
```

[ ]:
```
plot_confusion_matrix(
    cm,
    classes=[0,1],
    title='AdaBoost Classifier with Optimized Recall',
    save=True,
    normalize=True,
    cmap=plt.cm.Greys
)
```

[ ]:
```
p, r, thresholds = precision_recall_curve(y_true, y_probs[:,1])
```

[ ]:
```
best_threshold = plot_precision_recall_vs_threshold(
    p, r, thresholds,
    title='Precision and Recall Scores as a Function of the Decision Threshold',
)
```

[ ]:
```
plot_precision_recall_curve(
    p, r, y_probs[:,1], y_true, thresholds,
    title='Precision Recall Curve',
    t=best_threshold,
)
```

[ ]:
```
fpr, tpr, auc_thresholds = roc_curve(y_true, y_probs[:,1])
```

[ ]:
```
auc(fpr, tpr)
```

[ ]:
```
plot_roc_curve(
    fpr, tpr,
    title='ROC Curve',
    label='Recall Optimized'
)
```

[ ]:
```
y_pred_adjusted = get_adjusted_classes(y_probs[:,1], thresh=best_threshold)
```

[ ]:
```
cm = confusion_matrix(
    y_pred=y_pred_adjusted,
    y_true=y_true
)
```

[ ]:
```
plot_confusion_matrix(
    cm,
    classes=[0,1],
    title='AdaBoost Classifier with Optimized Recall',
    save=True,
    normalize=True,
    cmap=plt.cm.Greys
)
```

[ ]:
```

```

10.5 MultiClassClassifier

Utilities

[ ]:
```
def print_counts(labels):
    labels = np.argmax(labels, axis=1)
    for i in range(5):
        print(len(np.where(labels == i)[0]))
```

[ ]:
```
print_counts(test_labels_multiclass_np)
```

Model

[ ]:
```
over_sampler = SVMSMOTE()
```

[ ]:
```
x_train, y_train = over_sampler.fit_resample(
    train_features_np, np.argmax(
        train_labels_multiclass_np,
        axis=1
    )
)
```

[ ]:
```
x_train, y_train = shuffle(x_train, y_train)
```

[ ]:
```
Counter(y_train)
```

[ ]:
```
x_test, y_test = test_features_np, test_labels_multiclass_np
```

[ ]:
```
x_train.shape, x_test.shape, y_train.shape, y_test.shape
```

```
[ ]: lstm_output_size = 70
```

```
[ ]: cnn = Sequential()
     cnn.add(Convolution1D(64, 3,␣
      ↪border_mode="same",activation="relu",input_shape=(67, 1)))
     cnn.add(Convolution1D(64, 3, border_mode="same", activation="relu"))
     cnn.add(MaxPooling1D(pool_length=(2)))
     cnn.add(Convolution1D(128, 3, border_mode="same", activation="relu"))
     cnn.add(Convolution1D(128, 3, border_mode="same", activation="relu"))
     cnn.add(MaxPooling1D(pool_length=(2)))
     cnn.add(LSTM(lstm_output_size))
     cnn.add(Dropout(0.1))
     cnn.add(Dense(5, activation="softmax"))
```

```
[ ]: cnn.summary()
```

```
[ ]: cnn.compile(loss='categorical_crossentropy', optimizer='adam',␣
      ↪metrics=['accuracy'])
```

```
[ ]: history_1 = cnn.fit(
         np.expand_dims(x_train, axis=2), to_categorical(
             y_train,
             num_classes=5
         ),
         batch_size=128,
         epochs=5,
         validation_data=(
             np.expand_dims(x_test, axis=2), y_test
         )
     )
```

```
[ ]: all_losses = []
```

```
[ ]: for loss in history_1.history['loss']:
         all_losses.append(loss)
```

```
[ ]: all_acc = []
```

```
[ ]: for acc in history_1.history['accuracy']:
         all_acc.append(acc)
```

```
[ ]: len(all_losses), len(all_acc)
```

```
[ ]: plt.figure(figsize=(8,8))

     plt.plot(
         np.linspace(1, 10, 10) * 10,
```

```
    all_losses,
    'k--',
    marker='D'
)

plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.savefig('Loss.png')
```

[ ]:
```
plt.figure(figsize=(8,8))

plt.plot(
    np.linspace(1, 10, 10) * 10,
    all_acc,
    'k--',
    marker='D'
)

plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.savefig('Accuracy.png')
```

[ ]:
```
cnn.save('conv_model.h5')
```

[ ]:
```
import pickle as pk

with open('history.pkl', 'wb') as f:
    pk.dump(history, f)
```

[ ]:
```
import pickle as pk

with open('history_1.pkl', 'wb') as f:
    pk.dump(history_1, f)
```

[ ]:
```
plot_model(cnn, show_layer_names=False, show_shapes=True, t)
```

[ ]:
```
y_pred_rfc = rfc.predict(test_features_np)
```

[ ]:
```
normal_samples_indices = np.where(y_pred_rfc == 0)[0]
x_train_normal = test_features_np[normal_samples_indices]
y_train_multiclass_normal = test_labels_multiclass_np[normal_samples_indices]
```

[ ]:
```
y_train_multiclass_normal
```

[ ]:
```
x_train_normal.shape, y_train_multiclass_normal.shape
```

```
[ ]: malicious_samples_indices = np.where(y_pred_rfc == 1)[0]
x_train_malicious = test_features_np[malicious_samples_indices]
y_train_multiclass_malicious = ␣
 ↪test_labels_multiclass_np[malicious_samples_indices]
```

```
[ ]: x_train_malicious.shape, y_train_multiclass_malicious.shape
```

```
[ ]: y_true, y_pred, _ = get_predictions(
    cnn,
    np.expand_dims(x_train_malicious, axis=2),
    y_train_multiclass_malicious
)
```

```
[ ]: y_true_with_zeros = np.concatenate((y_true, np.argmax(y_train_multiclass_normal,␣
 ↪axis=1)))
```

```
[ ]: y_true_with_zeros.shape
```

```
[ ]: pred = np.argmax(y_pred, axis=1)
```

```
[ ]: pred_with_zeros = np.concatenate((pred, np.asarray([0] * 13920)))
```

```
[ ]: pred_with_zeros.shape
```

```
[ ]: pred_with_zeros
```

```
[ ]: cm = confusion_matrix(
    y_pred=pred_with_zeros,
    y_true=y_true_with_zeros
)
```

```
[ ]: plot_confusion_matrix(
    cm,
    classes=['Normal', 'DoS', 'Probe', 'U2R', 'R2L'],
    title='Final Confusion',
    save=True,
    normalize=True,
    cmap=plt.cm.Greys
)
```

```
[ ]:
```