福州大学

企业项目实训
中期报告

题　　目：基于机器学习的入侵检测系统

院　　系：计算机与大数据学院

专　　业：人工智能

小组组长：郝闰

小组成员：陈黾杰 陈明轩 石劭群 汤盛尧 王家铭

指导教师：傅明建

目录

一、项目简介与项目目标 3
二、项目各成员相关情况 4
（一）组员进度 4
（二）组员贡献度 4
三、项目开发研发进度 4
（一）数据集： 4
1、对数据集进行检查： 5
2、对数据集进行预处理： 5
（二）算法 9
1、Logistic Regression（逻辑回归）: 9
2、XGBoost: 11
3、K-Nearest Neighbors (KNN，K 近邻): 13
4、Long Short-Term Memory (LSTM): 14
5、对训练结果进行评估： 17
（三）前端 18
四、未来工作安排： 19
（一）工作计划： 19
目标： 19
冲刺期限： 19
任务列表： 19
（二）具体安排： 19
机器学习算法整合： 19
前端数据展示界面增强： 19
性能测试： 20
团队协作与沟通： 20
发布版本： 20

一、项目简介与项目目标

入侵检测系统（IDS）作为网络安全领域的核心组成部分，主要承担着监测网络流量和设备的责任，其宗旨在于发现并标识恶意活动、可疑行为或违反安全策略的各类行为。IDS 的核心功能体现在能够有效识别已知或潜在的安全威胁，并且通过向安全管理员发出警报或将警报传递至集中式安全工具，从而协助网络安全人员快速识别并应对网络威胁。这一系统的存在极大提高了网络安全的可感知性和响应速度，使安全团队能够及时发现并采取适当措施来应对潜在的安全威胁。

在异常检测领域的研究中，传统的机器学习方法如决策树（DT）、随机森林（RF）和支持向量机（SVM）在过去的研究中已经取得了显著成果，为该领域奠定了坚实的理论基础。随着深度学习技术的发展，卷积神经网络（CNN）、循环神经网络（RNN）和长短时记忆网络（LSTM）等算法也成为了异常检测领域的重要工具。它们能够更有效地从复杂数据中提取特征和模式，增强了异常检测的准确性和效率。近年来，生成对抗网络（GAN）和无监督学习算法的应用也日益增多，为异常检测带来了新的视角和创新方法。GAN 通过其独特的生成器和判别器结构，在学习数据分布方面显示出了独特的优势，为异常检测提供了更为灵活和高效的方法。

本项目旨在利用机器学习领域中异常检测的最新研究成果，设计并实现一个包含异常流量检测和日志分析功能的的可视化入侵检测系统，以助力保护网络数据产品的安全。

在本次实训中，我们将基于 CSE-CIC-IDS2018，UNSW_NB15 数据集开展实验，并结合深度学习和传统机器学习的方法设计一系列的对比实验，设计一个强大而高效的入侵检测系统。

二、项目各成员相关情况

（一）组员进度

1、算法部分同学：(郝闰)

测试了传统机器学习算法的性能

阅读图神经网络(GNN)和 few-shot 相关论文，希望用来改进效果和性能

2、前端展示同学：（汤盛尧，石劭群，王家铭 ）

学习前端知识

原型图设计

3、测试同学：（陈明轩，陈黾杰）

学习测试相关知识，掌握相关工具

（二）组员贡献度

郝闰：25%

陈黾杰 陈明轩 石劭群 汤盛尧 王家铭：各 15%

三、项目开发研发进度

（一）数据集：

UNSW-NB 15 数据集的原始网络数据包是由堪培拉新南威尔士大学网络靶场实验室的 IXIA PerfectStorm 工具创建的，用于生成真实的现代正常活动和合成的当代攻击行为的混合体。tcpdump 工具用于捕获 100 GB 的原始流量（例如，Pcap 文件）。该数据集有九种类型的攻击，即 Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms。使用 Argus、Bro-IDS 工具并开发了 12 种算法来生成总共 49 个带有类别标签的特征。这些功能在 UNSW-NB15_features.csv 文件中进行了描述。

记录总数为 200 万条，540,044 条，存储在四个 CSV 文件中，即

UNSW-NB15_1.csv、UNSW-NB15_2.csv、UNSW-NB15_3.csv 和 UNSW-NB15_4.csv。地面真值表名为 UNSW-NB15_GT.csv，事件文件列表名为 UNSW-NB15_LIST_EVENTS.csv。 该数据集的一个分区被配置为训练集和测试集，分别为 UNSW_NB15_training-set.csv 和 UNSW_NB15_testing-set.csv 。训练集中的记录数量为 175,341 条记录，测试集为不同类型、攻击和正常的 82,332 条记录。

1、对数据集进行检查：

```
# 看一下数据集数量是否符合要求
import os
import pandas as pd
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

最后得到结果训练集 175341 条记录，测试集 82332 条记录，与预期结果相符合。

2、对数据集进行预处理：

数据预处理

1. 舍弃不想管或过度干预的特征

舍弃一些不相关的特征或者对预测过分干预的特征。如攻击的具体类型。

```
list_drop = ['id','attack_cat']
df = pd.concat([train, test], axis=0, ignore_index=True)
df.drop(list_drop,axis=1,inplace=True)
```

2. 夹紧（Clamping）

主要目的是处理数据中的极端值，以减小某些分布的偏斜，提高数据的稳定性和模型的性能

```
import numpy as np
# Clamp extreme Values
df_numeric = df.select_dtypes(include=[np.number])
df_numeric.describe(include='all')
DEBUG =0

for feature in df_numeric.columns:
    if DEBUG == 1:
        print(feature)
        print('max = '+str(df_numeric[feature].max()))
        print('75th = '+str(df_numeric[feature].quantile(0.95)))
        print('median = '+str(df_numeric[feature].median()))
        print(df_numeric[feature].max()>10*df_numeric[feature].median())
        print('-----------------------------------------------------')
    if df_numeric[feature].max()>10*df_numeric[feature].median() and
df_numeric[feature].max()>10 :
        df[feature] = np.where(df[feature]<df[feature].quantile(0.95),
df[feature], df[feature].quantile(0.95))

df_numeric = df.select_dtypes(include=[np.number])
df_numeric.describe(include='all')
```

对几乎所有的数值应用对数函数，因为它们大多数都呈右偏分布。

逐个应用对数函数将会非常繁琐，因此设立了一个简单的规则：如果连续特征中的唯一值数量超过 50，则应用对数函数。寻找超过 50 个唯一值的原因是为了滤除更倾向于分类行为的基于整数的特征。

```
df_numeric = df.select_dtypes(include=[np.number])
df_before = df_numeric.copy()
DEBUG = 0
for feature in df_numeric.columns:
    if DEBUG == 1:
        print(feature)
        print('nunique = '+str(df_numeric[feature].nunique()))
        print(df_numeric[feature].nunique()>50)
        print('-----------------------------------------------------')
    if df_numeric[feature].nunique()>50:
        if df_numeric[feature].min()==0:
            df[feature] = np.log(df[feature]+1)
        else:
            df[feature] = np.log(df[feature])
```

```
df_numeric = df.select_dtypes(include=[np.number])
```

3. 减少分类特征中的标签

一些特征具有非常高的基数（cardinality），而本操作的目标是将基数降低至每个特征的 5 或 6。该逻辑是选择每个特征中出现次数最多的前 5 个标签作为标签，并将其余的标签设置为“-”（很少使用的）标签。当后续进行编码时，维度将不会爆炸并引发维度灾难。

我们使用卡方检验来参与数据特征的选择。卡方检验的基本原理是比较观察值和期望值之间的差异。在特征选择的上下文中，这意味着我们有：

观察值——在给定的特征和目标类别组合中实际观察到的频率。

期望值——如果特征和目标类别相互独立，则预期观察到的频率。

在特征选择中，使用卡方检验可以帮助识别对预测目标类别最有用的特征。一个高卡方分数的特征可能意味着该特征和目标类别之间有较强的关联，因此可能对建立模型非常有用。

我们在进行 select 之后，使用 One-Hot 编码来对特征进行转换。

```
# Feature Selection
from sklearn.feature_selection import SelectKBest, chi2

best_features = SelectKBest(score_func=chi2,k='all')

X = df.iloc[:,4:-2]
y = df.iloc[:,-1]
fit = best_features.fit(X,y)

df_scores=pd.DataFrame(fit.scores_)
df_col=pd.DataFrame(X.columns)

feature_score=pd.concat([df_col,df_scores],axis=1)
feature_score.columns=['feature','score']
feature_score.sort_values(by=['score'],ascending=True,inplace=True)

fig = go.Figure(go.Bar(
           x=feature_score['score'][0:21],
           y=feature_score['feature'][0:21],
           orientation='h'))
```

```
fig.update_layout(title="Top 20 Features",
                  height=1200,
                  showlegend=False,
                 )

fig.show()


#one-hot
X = df.iloc[:,:-1]
y = df.iloc[:,-1]

X.head()
feature_names = list(X.columns)
np.shape(X)

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1,2,3])],
remainder='passthrough')
X = np.array(ct.fit_transform(X))

for label in list(df_cat['state'].value_counts().index)[::-1][1:]:
    feature_names.insert(0, label)

for label in list(df_cat['service'].value_counts().index)[::-1][1:]:
    feature_names.insert(0, label)

for label in list(df_cat['proto'].value_counts().index)[::-1][1:]:
    feature_names.insert(0, label)
```

4. 对数据集进行拆分与特征缩放

```
# 重新把数据集拆分成train&test
X_train = X[:len(train)]
X_test = X[len(train):]
y_train = y[:len(train)]
y_test = y[len(train):]
np.shape(X_test)
df_cat.describe(include='all')

from sklearn.preprocessing import StandardScaler
```

```
sc = StandardScaler()
X_train[:, 18:] = sc.fit_transform(X_train[:, 18:])
X_test[:, 18:] = sc.transform(X_test[:, 18:])
type(y_test)
```

（二）算法

我们用以下几个模型进行分类：

1、Logistic Regression（逻辑回归）：

原理：逻辑回归是一种线性模型，它通过将输入特征的线性组合经过一个逻辑函数，将结果映射到 0 和 1 之间，从而进行二分类。模型通过学习权重和截距来最小化损失函数，以使预测尽可能接近实际的标签。

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score, recall_score, precision_score,
f1_score
from time import time
from tqdm import tqdm

# Assuming X_train, X_test, y_train, y_test are numpy arrays already provided and
preprocessed

# Convert numpy arrays to PyTorch tensors
X_train_tensor = torch.tensor(X_train.astype(np.float32))
X_test_tensor = torch.tensor(X_test.astype(np.float32))
y_train_tensor = torch.tensor(y_train.to_numpy().astype(np.float32))
y_test_tensor = torch.tensor(y_test.to_numpy().astype(np.float32))

# Create TensorDatasets and DataLoaders
train_data = TensorDataset(X_train_tensor, y_train_tensor)
test_data = TensorDataset(X_test_tensor, y_test_tensor)
```

```
train_loader = DataLoader(train_data, batch_size=64, shuffle=False)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)


# Define the logistic regression model
class LogisticRegressionModel(nn.Module):
    def __init__(self, num_features):
        super(LogisticRegressionModel, self).__init__()
        self.linear = nn.Linear(num_features, 1)

    def forward(self, x):
        y_pred = torch.sigmoid(self.linear(x))
        return y_pred


# Initialize the model
model = LogisticRegressionModel(X_train.shape[1])

# Loss and optimizer
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)


# Train the model
def train_model(model, train_loader, criterion, optimizer):
    model.train()
    start_time = time()
    for epoch in tqdm(range(100)):  # Loop over the dataset multiple times
        for i, (inputs, labels) in enumerate(train_loader):
            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels.view(-1, 1))
            loss.backward()
            optimizer.step()
    end_time = time()
    return end_time - start_time


# Function to predict and calculate metrics
def evaluate_model(model, test_loader):
    model.eval()
```

```
    y_pred = []
    y_true = []
    start_time = time()
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            y_pred.extend(outputs.data.numpy().flatten())
            y_true.extend(labels.numpy().flatten())
    end_time = time()
    predict_time = end_time - start_time
    return y_true, y_pred, predict_time


# Train the model and record the time
train_time = train_model(model, train_loader, criterion, optimizer)

# Evaluate the model
y_true, y_pred, predict_time = evaluate_model(model, test_loader)

# Convert probabilities to binary predictions
y_pred_binary = [1 if x >= 0.5 else 0 for x in y_pred]

# Calculate metrics
accuracy = accuracy_score(y_true, y_pred_binary)
recall = recall_score(y_true, y_pred_binary)
precision = precision_score(y_true, y_pred_binary)
f1 = f1_score(y_true, y_pred_binary)
total_time = train_time + predict_time

model_performance =
pd.DataFrame(columns=['Accuracy','Recall','Precision','F1-Score','time to
train','time to predict','total time'])
model_performance.loc['Logistic'] = [accuracy, recall, precision,
f1,train_time,predict_time,total_time]
```

2、XGBoost:

原理：XGBoost 是一种梯度提升算法，通过迭代地训练弱学习器，将它们组合成一个强大的模型。在每一轮迭代中，模型根据前一轮的误差来调整弱学习器的权重，以逐步改进模型的性能。XGBoost 通过最小化损失函数，如平方损失或对数损失，来优化模型。

```python
import xgboost as xgb
from sklearn.metrics import accuracy_score, recall_score, precision_score,
f1_score
import pandas as pd
from time import time

# 假设 X_train, X_test, y_train, y_test 已经是预处理好的 numpy 数组

# 创建 XGBoost DMatrix
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置 XGBoost 参数
params = {
    'objective': 'binary:logistic',
    'max_depth': 8,
    'min_child_weight': 1,
    'eta': 0.3,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'n_estimators': 100,
    'scale_pos_weight': 1,
    'verbosity': 0
}
num_round = 500  # 训练轮数

# 训练模型
start_train = time()
bst = xgb.train(params, dtrain, num_round)
train_time = time() - start_train

# 预测
start_predict = time()
y_pred_prob = bst.predict(dtest)
predict_time = time() - start_predict

# 将概率转换为二元预测
y_pred = [1 if x > 0.5 else 0 for x in y_pred_prob]

# 计算指标
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
```

```
f1 = f1_score(y_test, y_pred)

# 总时间
total_time = train_time + predict_time

# # 将指标存储到 DataFrame
model_performance.loc['XGBOOST'] = [accuracy, recall, precision,
f1,train_time,predict_time,total_time]
```

3、K-Nearest Neighbors (KNN，K 近邻):

原理：KNN 是一种基于实例的学习算法，它根据新样本与训练集中样本的距离，将新样本分配给最近的 K 个邻居，并根据它们的类别投票来决定新样本的类别。KNN 假设相似的样本具有相似的类别，因此通过测量距离来进行分类。

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, recall_score, precision_score,
f1_score
from time import time

# Assuming X_train, X_test, y_train, y_test are numpy arrays already provided and
preprocessed

# 创建 KNN 模型实例
knn = KNeighborsClassifier(n_neighbors=5)  # 可以调整 n_neighbors 的值

# Function to train and evaluate the model
def train_and_evaluate_knn(knn, X_train, y_train, X_test, y_test):
    start_train_time = time()
    knn.fit(X_train, y_train)  # 训练 KNN 模型
    end_train_time = time()
    train_time = end_train_time - start_train_time

    start_predict_time = time()
    y_pred = knn.predict(X_test)  # 使用模型进行预测
    end_predict_time = time()
    predict_time = end_predict_time - start_predict_time

    # 计算性能指标
```

```
    accuracy = accuracy_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    total_time = train_time + predict_time
    model_performance.loc['KNN'] = [accuracy, recall, precision,
f1,train_time,predict_time,total_time]

    # 创建性能指标的 DataFrame
    performance = pd.DataFrame({
        'Accuracy': [accuracy],
        'Recall': [recall],
        'Precision': [precision],
        'F1-Score': [f1],
        'time to train': [train_time],
        'time to predict': [predict_time],
        'total time': [total_time]
    })

    return performance

# 使用模型并获取性能指标
train_and_evaluate_knn(knn, X_train, y_train, X_test, y_test)
```

4、Long Short-Term Memory (LSTM):

原理：LSTM 是一种循环神经网络（RNN），设计用于处理和预测序列数据。它具有记忆单元，通过学习来控制存储和遗忘信息，以便更好地捕捉序列中的长距离依赖关系。LSTM 通过门控机制（输入门、遗忘门、输出门）来调节信息的流动，从而在处理序列数据时表现出色。

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import numpy as np
from time import time

# 定义 LSTM 模型
```

```
class LSTMModel(nn.Module):
    def __init__(self):
        super(LSTMModel, self).__init__()
        self.lstm1 = nn.LSTM(56, 20, batch_first=True)
        self.lstm2 = nn.LSTM(20, 20, batch_first=True)
        self.dense = nn.Linear(20, 2)  # 两个输出类别

    def forward(self, x):
        x, _ = self.lstm1(x)
        x, _ = self.lstm2(x)
        x = x[:, -1, :]  # 取最后一个时间步的输出
        x = self.dense(x)
        return x

# 假设 X_train, X_test, y_train, y_test 是已经定义好的 NumPy 数组

# 重塑数据以符合 LSTM 的输入要求：(样本数, 时间步数, 特征数)
X_train_tensor = torch.tensor(X_train.reshape(X_train.shape[0], 1,
56).astype(np.float32))
X_test_tensor = torch.tensor(X_test.reshape(X_test.shape[0], 1,
56).astype(np.float32))
y_train_tensor = torch.tensor(y_train.to_numpy().astype(np.int64))
y_test_tensor = torch.tensor(y_test.to_numpy().astype(np.int64))

# 创建 DataLoader
train_data = TensorDataset(X_train_tensor, y_train_tensor)
test_data = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_data, batch_size=2000, shuffle=False)
test_loader = DataLoader(test_data, batch_size=2000, shuffle=False)

# 初始化模型
model = LSTMModel()

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()  # 适用于二类分类问题
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 训练模型的函数
def train_model(model, train_loader, criterion, optimizer, epochs):
    model.train()
    start_time = time()
    for epoch in range(epochs):
        for inputs, labels in train_loader:
            optimizer.zero_grad()
```

```
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
    end_time = time()
    print(f"训练时间：{end_time - start_time}秒")

# 训练模型&评分
# Function to evaluate the model and calculate metrics
def evaluate_model(model, test_loader):
    model.eval()
    y_pred = []
    y_true = []
    start_time = time()
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            y_pred.extend(predicted.numpy())
            y_true.extend(labels.numpy())
    end_time = time()
    predict_time = end_time - start_time

    # Calculate metrics
    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, precision, recall, f1, predict_time

# Train the model and record the time
start_train = time()
train_model(model, train_loader, criterion, optimizer, epochs=200)
end_train = time()
train_time = end_train - start_train

# Evaluate the model
accuracy, precision, recall, f1, predict_time = evaluate_model(model, test_loader)

# Calculate total time
total_time = train_time + predict_time
```

```
# 创建性能指标的 DataFrame
model_performance.loc['LSTM'] = [accuracy, recall, precision,
f1,train_time,predict_time,total_time]
```

5、对训练结果进行评估：

我们使用以下几种方法对训练结果进行评估：

准确率（Accuracy）：准确率是模型正确性的度量。它计算为正确预测的实例数与总实例数的比率。

召回率（Recall）：召回率衡量模型捕获所有相关实例或真正类的能力。它是正确预测的正实例数与实际正实例总数的比率。

精确率（Precision）：精确率是正预测的准确性度量。它是正确预测的正实例数与总预测的正实例数的比率。

F1 分数：F1 分数是精确率和召回率的调和平均值。它提供了在精确率和召回率之间取得平衡的单一分数，考虑了假正类和假负类。

训练时间（Train Time）：训练时间是模型在训练阶段对给定数据集进行训练所需的时间。

预测时间（Predict Time）：预测时间是模型在新的、未见过的数据上进行预测所需的时间。

总时间（Total Time）：总时间是训练时间和预测时间的总和，代表了使用模型的整体计算成本。

对几种算法的评估结果大致如下：

	Accuracy	Recall	Precision	F1-Score	time to train	time to predict	total time
Logistic	81.02%	99.46%	0.745650	85.23%	156.6	0.5	157.1
XGBOOST	86.79%	97.49%	0.819412	89.04%	31.6	0.6	32.2
KNN	86.01%	96.45%	0.815264	88.36%	0.0	33.9	33.9
LSTM	82.42%	99.08%	0.761624	86.12%	389.5	0.8	390.4

根据上述数据我们可知：

Logistic Regression 在召回率上表现非常出色，几乎捕获了所有的正实例，但精确率相对较低。训练时间较长，但预测时间和总时间都较短。

XGBoost 在准确率、召回率和精确率上表现良好，并且训练时间相对较短，总时间也较短。

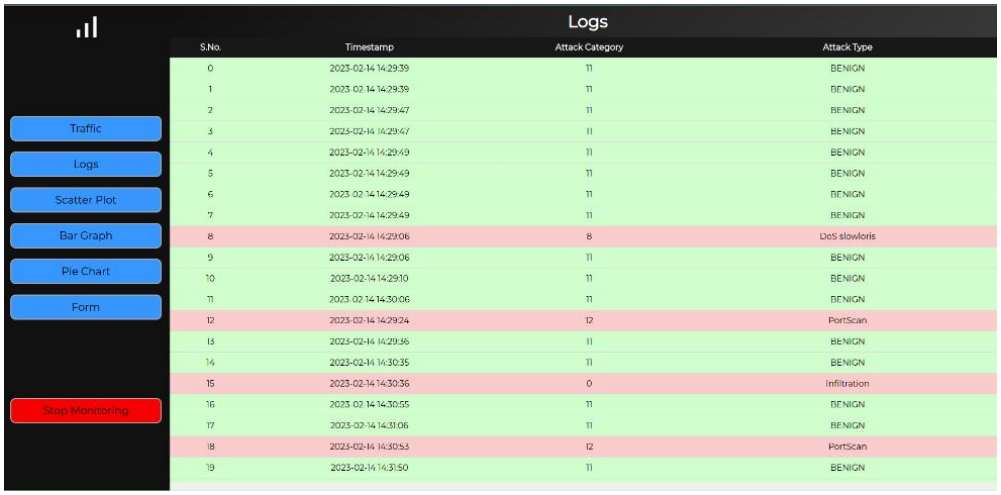
KNN 在召回率上表现不错，但相比之下，精确率和总时间较高。不同于其他算法，KNN 在训练时无需明确的训练过程，但在预测时需要较长的时间。

LSTM 在召回率上表现良好，但相比之下，训练时间较长，而精确率和总时间相对较高。适用于序列数据的 LSTM 可能在入侵检测中提供了一定的优势。

（三）前端

使用 react 框架，完成数据展示页面的一部分

效果图如下图所示：



四、未来工作安排：

（一）工作计划：

目标： 在 7 天内完成入侵检测系统的 Alpha 版本，包括图神经网络的整合、前端数据展示界面的增强以及性能测试。

冲刺期限： 7 天

任务列表：

1、完成图神经网络的整合，模型在验证集上达到预期的性能指标。

2、前端数据展示界面得到用户认可，满足用户需求。

3、完成性能测试，发现并解决系统的性能瓶颈。

4、团队成员之间的协作和沟通良好，项目文档得到及时更新。

（二）具体安排：

机器学习算法整合：

第 1-2 天： 图神经网络（GNN）算法实现，确保其在入侵检测任务上的准确性。

第 3-4 天： 编写机器学习分析后的后端 API 数据请求接口。

前端数据展示界面增强：

第 1-2 天： 分析当前前端展示界面的需求，设计新的数据可视化组件。

第 3-4 天： 实施新的数据可视化组件，确保其与后端的数据接口协调良好。

性能测试：

第 5 天： 制定性能测试计划，明确测试的指标和场景，开发性能测试脚本和工具。

第 6 天： 运行性能测试，并分析测试结果，确定系统的瓶颈和优化点。

第 7 天： 优化系统性能，确保在实际使用场景中的稳定性和高效性。

团队协作与沟通：

每天： 召开站立式会议，分享进展和挑战，及时解决问题。

每天： 更新项目文档，包括算法的文档、前端界面的设计文档以及性能测试报告。

发布版本：

第 7 天： 进行集成测试，确保各个模块的协同工作正常。

第 7 天： 编写发布说明文档，准备发布材料，包括软件包、文档和必要的支持资源