Detecção de Intrusões com Privacidade Diferencial (DP-IDS)

Autores: Julio Cesar Barbosa Machado, Arthur Linhares Madureira

1. Introdução

Sistemas de detecção de intrusões (IDS) são fundamentais para a segurança cibernética. Contudo, os dados utilizados para treinar tais sistemas frequentemente contêm informações sensíveis, como padrões de tráfego e ações de usuários legítimos. Para mitigar riscos de vazamento, adotamos a abordagem de Privacidade Diferencial (DP), conforme proposto por Abadi et al. (2016), e aplicada a um modelo de rede neural treinado com o algoritmo DP-SGD.

2. Dataset NSL-KDD

Utilizamos o NSL-KDD, uma versão aprimorada do KDD Cup '99, composta por 41 atributos de tráfego de rede. As amostras são rotuladas como 'normal' ou como diferentes tipos de ataques. O NSL-KDD remove redundâncias, oferecendo um benchmark mais realista e balanceado para avaliação de IDS.

3. Privacidade Diferencial em Redes Neurais

Segundo Abadi et al. (2016), Privacidade Diferencial pode ser incorporada ao treinamento de redes neurais por meio de alterações no processo de otimização. O algoritmo DP-SGD consiste em três etapas principais: (i) clipping do gradiente de cada exemplo para limitar sua influência; (ii) adição de ruído Gaussiano; e (iii) cálculo do orçamento de privacidade ε via contabilidade RDP (Rényi Differential Privacy).

4. Dependências

- numpy
- pandas
- matplotlib
- torch
- opacus
- scikit-learn

5. Estrutura de Código

5.1 Importações

```
import os

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import torch

from torch.utils.data import TensorDataset, DataLoader

from opacus import PrivacyEngine

from sklearn.preprocessing import OneHotEncoder, StandardScaler, LabelEncoder

from sklearn.metrics import accuracy_score
```

5.2 Função load_preprocess()

```
def load_preprocess(base_dir):

    feature_file = os.path.join(base_dir, 'Field Names.csv')

    feature_names = pd.read_csv(feature_file, header=None)[0].tolist()

    cols = feature_names + ['attack', 'difficulty']

    train = pd.read_csv(os.path.join(base_dir, 'KDDTrain+.csv'), names=cols,
header=None)

    test  = pd.read_csv(os.path.join(base_dir, 'KDDTest+.csv'),  names=cols,
header=None)

    seen_attacks = set(train['attack'])

    test['attack'] = test['attack'].apply(lambda x: x if x in seen_attacks else
'unknown')

    X_train, y_train = train[feature_names], train['attack']

    X_test,  y_test  = test[feature_names],  test['attack']

    cat_cols = X_train.select_dtypes(include=['object']).columns.tolist()

    num_cols = X_train.select_dtypes(include=[np.number]).columns.tolist()

    enc = OneHotEncoder(sparse_output=False, handle_unknown='ignore')

    X_train_cat = enc.fit_transform(X_train[cat_cols])

    X_test_cat  = enc.transform(X_test[cat_cols])

    scaler = StandardScaler()
```

```
    X_train_num = scaler.fit_transform(X_train[num_cols])
    X_test_num  = scaler.transform(X_test[num_cols])
    X_train_proc = np.hstack([X_train_num, X_train_cat])
    X_test_proc  = np.hstack([X_test_num,  X_test_cat])
    le = LabelEncoder()
    y_train_enc = le.fit_transform(y_train)
    classes = list(le.classes_) + (['unknown'] if 'unknown' not in le.classes_
else [])
    le.classes_ = np.array(classes)
    y_test_enc  = le.transform(y_test)
    return X_train_proc, X_test_proc, y_train_enc, y_test_enc, classes
```

5.3 Função make_mlp()

```
def make_mlp(input_dim, num_classes, hidden_dim=100):
    return torch.nn.Sequential(
        torch.nn.Linear(input_dim, hidden_dim),
        torch.nn.ReLU(),
        torch.nn.Linear(hidden_dim, num_classes)
    )
```

5.4 Função train_baseline()

```
def train_baseline(X_tr, y_tr, X_te, y_te, epochs=10, batch_size=256, lr=0.1):
    model = make_mlp(X_tr.shape[1], len(classes))
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    ds = TensorDataset(torch.tensor(X_tr, dtype=torch.float32),
torch.tensor(y_tr, dtype=torch.long))
    loader = DataLoader(ds, batch_size=batch_size, shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            optimizer.zero_grad()
            loss = torch.nn.functional.cross_entropy(model(xb), yb)
            loss.backward()
```

```
            optimizer.step()

    model.eval()

    with torch.no_grad():

        preds = model(torch.tensor(X_te,
dtype=torch.float32)).argmax(dim=1).numpy()

    return accuracy_score(y_te, preds)
```

5.5 Função train_dp_sgd()

```
def train_dp_sgd(X_tr, y_tr, X_te, y_te, sigma, epochs=10, batch_size=256,

                 lr=0.1, max_grad_norm=1.0, delta=1e-5):

    model = make_mlp(X_tr.shape[1], len(classes))

    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    ds = TensorDataset(torch.tensor(X_tr, dtype=torch.float32),
torch.tensor(y_tr, dtype=torch.long))

    loader = DataLoader(ds, batch_size=batch_size, shuffle=True)

    engine = PrivacyEngine(accountant='rdp')

    model, optimizer, loader = engine.make_private(

        module=model,

        optimizer=optimizer,

        data_loader=loader,

        noise_multiplier=sigma,

        max_grad_norm=max_grad_norm

    )

    for _ in range(epochs):

        model.train()

        for xb, yb in loader:

            optimizer.zero_grad()

            loss = torch.nn.functional.cross_entropy(model(xb), yb)

            loss.backward()

            optimizer.step()

    model.eval()

    with torch.no_grad():
```

```
        preds = model(torch.tensor(X_te,
dtype=torch.float32)).argmax(dim=1).numpy()

    acc = accuracy_score(y_te, preds)

    epsilon = engine.accountant.get_epsilon(delta=delta, alphas=[2, 4, 8, 16,
32])

    return acc, epsilon
```

5.6 Avaliação e Visualização

Executamos quatro variações de σ (0.1, 1.0, 2.0 e 5.0). Para cada valor relatamos a acurácia de teste e o ε correspondente. Um gráfico foi gerado no notebook para ilustrar o trade-off privacidade-utilidade.

6. Resultados

• Acurácia baseline (sem DP): 0.70
• Acurácia com DP-SGD (σ = 1.0): 0.68 (ε ≈ 1.36)
Valores maiores de σ diminuem ε, mantendo a acurácia praticamente constante.

7. Conclusões

Os resultados confirmam o achado seminal de Abadi et al. (2016): é possível treinar redes neurais com garantias formais de privacidade mantendo desempenho competitivo. A queda de ∼2 p.p. na acurácia demonstra que o regime de Privacidade Diferencial é viável para tarefas de IDS em ambientes sensíveis. Como trabalho futuro, recomenda-se explorar arquiteturas mais profundas e ajustar o cálculo de ε via contabilidade RDP com intervalo de ordens mais amplo para obter limites ainda mais justos.