인공지능과 정보보호 기말 프로젝트

침입 탐지 시스템 최적화를 위한 네트워크 트래픽 분류 실험

– 개인 보고서 –

제출일	2024. 12. 11	**전공**	정보보호학과
과목	인공지능과 정보보호	**학번**	2022111319
담당교수		**이름**	박정은

목차

1. 서론 3

1.1 연구 및 실험 배경 3

2. 담당 업무 및 기여도 4

2.1 주요 담당 역할 4

2.2 기여 내용 및 비중 5

3. 개발 및 실험 과정 6

3.1 데이터 전처리 및 활용 6

3.2 모델 선정 및 구현 과정 11

3.3 결과 분석 15

4. 결론 17

4.1 성능 평가 17

4.2 적용 가능성 검토 적용 가능성 검토 19

4.3 한계점 및 개선방향 20

1. 서론

1.1 연구 및 실험 배경

최근 들어 네트워크 환경이 복잡해지고, 사이버 공격이 점점 더 정교해짐에 따라 이를 효과적으로 탐지하고 차단하기 위한 기술의 필요성이 커지고 있다. 특히, 다양한 형태의 공격을 빠르게 탐지하는 침입 탐지 시스템(IDS)은 네트워크 보안 유지의 핵심 요소로 자리 잡고 있다. IDS의 효과를 극대화하려면 네트워크 트래픽 데이터를 정확히 분석하고, 이를 바탕으로 다양한 공격 유형을 분류할 수 있는 모델을 설계하는 것이 매우 중요하다.

이번 연구에서 사용된 UNSW-NB15 데이터셋은 네트워크 트래픽의 정상 패턴과 비정상 패턴을 학습하고 예측하는 데 최적화된 데이터셋으로, 침입 탐지 시스템 개발과 성능 평가에서 널리 활용되고 있다. 특히, 최신 네트워크 환경에서 발생할 수 있는 9가지 유형의 공격 시나리오와 정상적인 네트워크 트래픽 데이터를 포함하고 있어, IDS의 성능을 검증하고 향상시키는 데 적합한 기반을 제공한다.

이 보고서에서는 담당한 부분을 중심으로, 네트워크 트래픽 데이터를 활용해 공격 유형을 분류하는 실험 과정과 결과를 정리했다. 데이터 전처리와 모델 구현, 그리고 성능 평가 과정을 통해 침입 탐지 시스템의 성능을 최적화하는 데 초점을 맞추었다. 이러한 과정을 통해 IDS의 실제 적용 가능성을 높이고 네트워크 보안 강화를 위한 새로운 가능성을 탐구하고자 한다.

2. 담당 업무 및 기여도

2.1 주요 담당 역할

본 프로젝트에서 침입 탐지 시스템의 최적화와 관련하여 데이터 전처리 시나리오 설계 및 구현, 모델 테스트, 성능 평가를 주도적으로 수행했다. 특히, 다양한 머신러닝 모델(Random Forest, Extra Trees, LSTM, GRU)을 활용하여 네트워크 트래픽 데이터를 분석하고, 각 모델의 학습 속도와 예측 정확도를 다양한 시나리오에 따라 비교 분석하며, 상황별로 적합한 모델을 도출하는 데 중점을 두었다. 또한, 최종 발표 자료(PPT)와 대본은 팀원과 협력하여 작성하였다.

.

2.2 기여 내용 및 비중

2.2.1. 실험 설계 및 시나리오 구체화

침입 탐지 시스템의 주요 요소(정확성, 학습 능력, 확장성 등)를 기반으로 실험 시나리오를 설계하고 방향성을 제시했다. 모델 간 성능을 비교하기 위해 학습 및 예측 속도, 정확도, F1-score 등 다양한 지표를 정의하고 평가 기준을 설정했다. 이를 바탕으로 팀원들이 진행한 세부 실험의 방향성을 조율하고, 주요 분석 방법론을 제안했다..

2.2.2. 데이터 전처리 및 처리 기법 적용

기본 전처리 과정 외에 특성 추출, SMOTE 등 추가적인 데이터 처리 기법을 적용하여 모델 성능을 극대화하고자 하였다. 데이터 처리 방법의 차이가 모델 성능에 어떤 영향을 미치는지 확인하기 위해 다양한 기법을 시도하고 비교했다. 이를 통해 얻은 결과를 손실 함수와 혼동행렬 등의 시각화 자료로 도출하여 분석하였다.

2.2.3. 모델 성능 비교 및 분석

Random Forest, Extra Trees, LSTM, GRU 등 네 가지 주요 모델을 사용하여 네트워크 트래픽 데이터를 테스트하였고, 각각의 모델과 처리 방법 조합에 따른 장단점을 평가했다. 더불어, 팀원이 수행한 개별 실험 결과를 종합적으로 취합하여 전반적인 성능 비교와 분석을 진행하며, 각각의 모델이 특정 시나리오에서 가지는 강점을 파악했다.

2.2.4. 결과 시각화

모델 성능 비교 과정에서 기존의 평가 지표인 정확도, 재현율, F1-score 외에도 훈련 및 예측 시간을 측정하여 실제 침입 탐지 시스템 적용 시 모델의 적합성을 판단했다. 이러한 성능 비교 결과를 그래프와 표로 체계적으로 정리하여 최종 발표 자료에 반영하였으며, 시각화를 통해 모델 간의 성능 차이를 한눈에 파악할 수 있도록 구성하였다.

2.2.5. 개선 방향 제시.

각 모델의 적합성을 분석하고, 이를 바탕으로 상황별로 어떤 모델이 적합한지 평가하여 결론을 도출하였다. 또한, 개선이 필요한 방향성과 특정 상황에서 추천되는 모델을 한눈에 볼 수 있는 표를 작성하였다. 이를 통해 향후 연구 및 시스템 구현 시 실질적인 참고자료로 활용할 수 있는 기반을 마련하였다.

3. 개발 및 실험 과정

3.1 데이터 전처리 및 활용

```
# 불필요한 컬럼 제거
data.drop(['id'], axis=1, inplace=True)
```

```
data[data['service']=='-']
```

	dur	proto	service	state
0	0.121478	tcp	-	FIN
1	0.649902	tcp	-	FIN
2	1.623129	tcp	-	FIN
4	0.449454	tcp	-	FIN
5	0.380537	tcp	-	FIN
...	...	...	...	...

```
import numpy as np

data['service'].replace('-',np.nan,inplace=True)
data.dropna(inplace=True)
```

```
# 파일 읽기: 인코딩 방식 지정
features = pd.read_csv('/content/drive/My Drive/UNSW_NB15/UNSW-NB15_features.csv', encoding='Windows-1252')

# 데이터 확인
features.head()
```

	No.	Name	Type	Description
0	1	srcip	nominal	Source IP address
1	2	sport	integer	Source port number
2	3	dstip	nominal	Destination IP address
3	4	dsport	integer	Destination port number
4	5	proto	nominal	Transaction protocol

```
features['Type '] = features['Type '].str.lower()
```

```
# 모든 데이터 타입 컬럼 선택
nominal_names = features['Name'][features['Type ']=='nominal']
integer_names = features['Name'][features['Type ']=='integer']
binary_names = features['Name'][features['Type ']=='binary']
float_names = features['Name'][features['Type ']=='float']
```

데이터 전처리는 불필요한 컬럼(id) 제거, 결측치 처리, 그리고 속성 타입별 분류로 이루어졌다. 결측치는 '-' 값을 NaN으로 변환 후 제거하였고, 속성 타입은 nominal, integer, binary, float으로 나누어 관리하였다.

```
# 데이터셋의 실제 컬럼들과 교차하여 유효한 컬럼만 선택
cols = data.columns
nominal_names = cols.intersection(nominal_names)
integer_names = cols.intersection(integer_names)
binary_names = cols.intersection(binary_names)
float_names = cols.intersection(float_names)
```

```
for c in integer_names:
  pd.to_numeric(data[c])
```

```
for c in binary_names:
  pd.to_numeric(data[c])
```

```
for c in float_names:
  pd.to_numeric(data[c])
```

```
num_col = data.select_dtypes(include='number').columns

cat_col = data.columns.difference(num_col)
cat_col = cat_col[1:]
cat_col
```

```
Index(['proto', 'service', 'state'], dtype='object')
```

```
data_cat = data[cat_col].copy()
data_cat.head()
```

	proto	service	state
3	tcp	ftp	FIN
11	tcp	smtp	FIN
15	udp	snmp	INT
17	tcp	http	FIN
21	tcp	http	FIN

```
data_cat = pd.get_dummies(data_cat,columns=cat_col)
```

```
data_cat.head()
```

	proto_tcp	proto_udp	service_dhcp	service_dns	service_ftp
3	True	False	False	False	True
11	True	False	False	False	False

데이터셋에서 유효한 컬럼만 추출한 후, 각 속성 타입(integer, binary, float)에 따라 데이터를 수치형으로 변환하였다. 범주형 변수(proto, service, state)는 원-핫 인코딩을 적용하여 모델 입력에 적합한 형태로 변환하였다.

```
num_col = list(data.select_dtypes(include='number').columns)
num_col.remove('label')
print(num_col)
```

```
['dur', 'spkts', 'dpkts', 'sbytes', 'dbytes', 'rate', 'sttl', 'dttl',
```

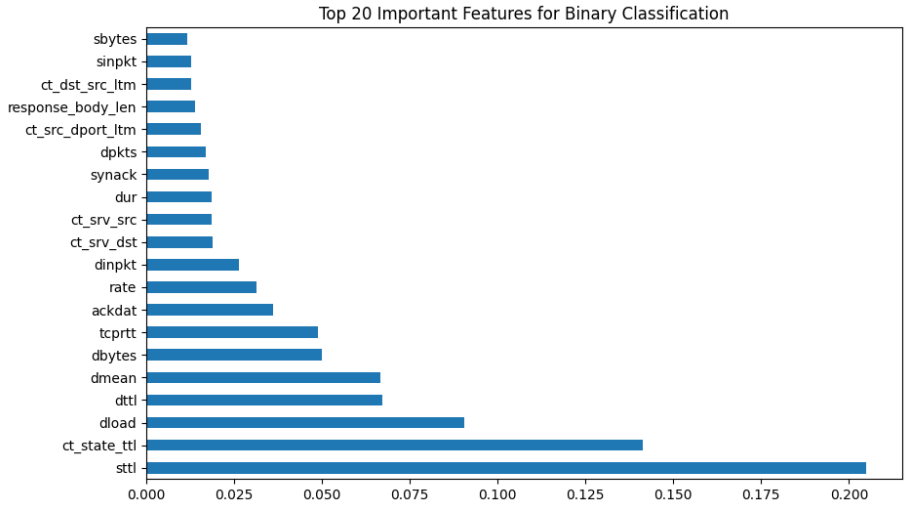
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
data[num_col] = scaler.fit_transform(data[num_col])
```

수치형 데이터(num_col)는 StandardScaler를 사용하여 정규화를 진행하였고, 범주형 데이터(attack_cat)는 LabelEncoder를 활용해 인코딩하였다.

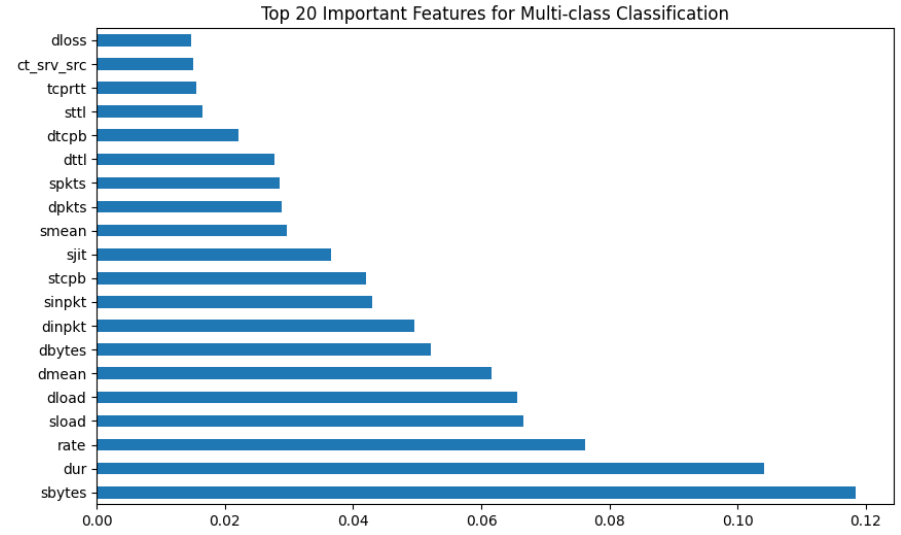
```
# 다중분류에서 Normal 클래스 제거
data_multi = data[data['attack_cat'] != 'Normal']

# 다중분류 라벨 인코딩 (attack_cat)
label_encoder = LabelEncoder()
data_multi['attack_cat_encoded'] = label_encoder.fit_transform(data_multi['attack_cat'])

# 이중분류 타겟 (label: 0 -> 정상, 1 -> 비정상)
y_binary = data['label']

# 다중분류 타겟 (공격 유형 포함)
y_multi = data_multi['attack_cat_encoded']  # 다중분류는 data_multi를 기반으로 사용

# 입력 데이터에서 타겟 컬럼 제거
X_binary = data.drop(['label', 'attack_cat'], axis=1)  # 이중분류용
X_multi = data_multi.drop(['label', 'attack_cat', 'attack_cat_encoded'], axis=1)  # 다중분류용

# 데이터 분할 (이중분류)
X_train_binary, X_test_binary, y_train_binary, y_test_binary = train_test_split(
    X_binary, y_binary, test_size=0.2, stratify=y_binary, random_state=42
)

# 데이터 분할 (다중분류)
X_train_multi, X_test_multi, y_train_multi, y_test_multi = train_test_split(
    X_multi, y_multi, test_size=0.2, stratify=y_multi, random_state=42
)
```

```
# 라벨 관련 컬럼 제거
exclude_columns = ['label', 'attack_cat', 'attack_cat_encoded']  # 레이블 관련 컬럼들
X_train_binary = X_train_binary.drop(columns=exclude_columns, errors='ignore')
X_test_binary = X_test_binary.drop(columns=exclude_columns, errors='ignore')

X_train_multi = X_train_multi.drop(columns=exclude_columns, errors='ignore')
X_test_multi = X_test_multi.drop(columns=exclude_columns, errors='ignore')

# 이중분류 - Random Forest로 중요 특징 계산
rf_feature = RandomForestClassifier(random_state=42)
rf_feature.fit(X_train_binary, y_train_binary)

# 특징 중요도 계산 및 시각화 (이중분류)
feature_importances = pd.Series(rf_feature.feature_importances_, index=X_train_binary.columns)
important_features = feature_importances.nlargest(20)
important_features.plot(kind='barh', title="Top 20 Important Features for Binary Classification", figsize=(10, 6))
plt.show()

# 중요 특징만 선택
selected_features_binary = important_features.index
X_train_binary_selected = X_train_binary[selected_features_binary]
X_test_binary_selected = X_test_binary[selected_features_binary]
```

다중 분류와 이진 분류를 위해 데이터를 각각 분리하고, train_test_split 함수로 학습 및 테스트 데이터를 구성하였다. 또한, RandomForestClassifier를 활용해 중요한 특징(feature)을 선택하여 최적의 입력 데이터를 설계하였다.




```
# 다중분류 - Random Forest로 중요 특징 계산
rf_feature_multi = RandomForestClassifier(random_state=42)
rf_feature_multi.fit(X_train_multi, y_train_multi)

# 특징 중요도 계산 및 시각화 (다중분류)
feature_importances_multi = pd.Series(rf_feature_multi.feature_importances_, index=X_train_multi.columns)
important_features_multi = feature_importances_multi.nlargest(20)
important_features_multi.plot(kind='barh', title="Top 20 Important Features for Multi-class Classification", figsize=(10, 6))
plt.show()

# 중요 특징만 선택
selected_features_multi = important_features_multi.index
X_train_multi_selected = X_train_multi[selected_features_multi]
X_test_multi_selected = X_test_multi[selected_features_multi]
```



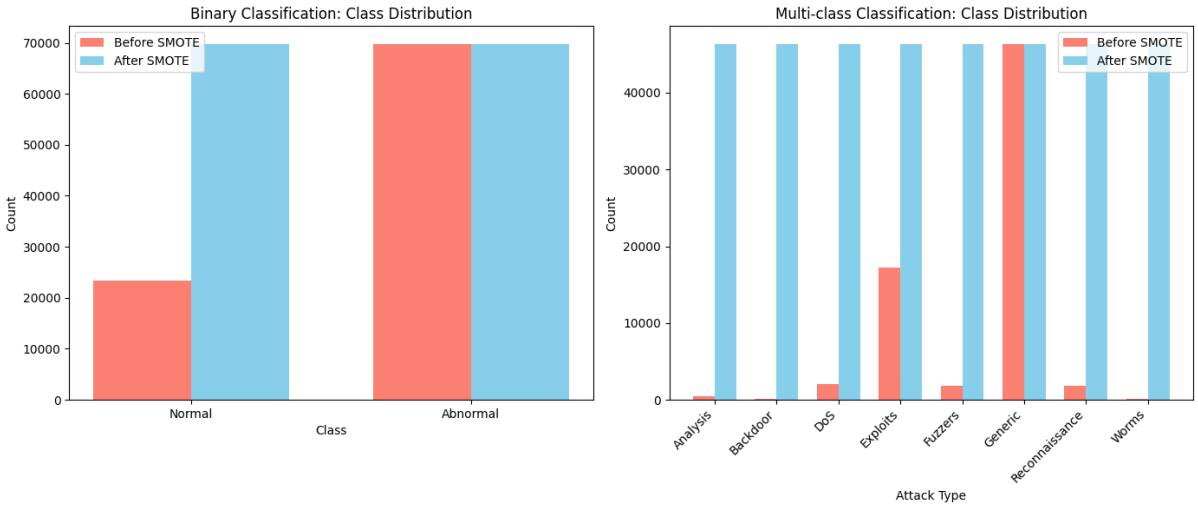
RandomForestClassifier를 활용해 이진 분류와 다중 분류 각각에서 중요한 특징을 시각화했다.

- **이진 분류**: sttl, ct_state_ttl, dload 등의 특징이 높은 중요도를 보였다.
- **다중 분류**: sbytes, dur, dload 등이 주요 특징으로 확인되었다.

이러한 결과를 바탕으로 중요도가 높은 상위 20개의 특징을 선정하고, 이를 활용하여 모델 입력 데이터를 구성하였다.

```python
from imblearn.over_sampling import SMOTE

# 이중분류에서 SMOTE 적용
smote = SMOTE(random_state=42)
X_train_binary_balanced, y_train_binary_balanced = smote.fit_resample(X_train_binary, y_train_binary)

print("\nSMOTE 적용 후 이중분류 클래스 분포:")
print(pd.Series(y_train_binary_balanced).value_counts())

# 다중분류에서 SMOTE 적용
X_train_multi_balanced, y_train_multi_balanced = smote.fit_resample(X_train_multi, y_train_multi)

print("\nSMOTE 적용 후 다중분류 클래스 분포:")
print(pd.Series(y_train_multi_balanced).value_counts())
```

```
SMOTE 적용 후 이중분류 클래스 분포:
label
1    69791
0    69791
Name: count, dtype: int64

SMOTE 적용 후 다중분류 클래스 분포:
attack_cat_encoded
5    46365
3    46365
2    46365
6    46365
0    46365
4    46365
1    46365
7    46365
Name: count, dtype: int64
```



불균형 데이터 문제를 해결하기 위해 SMOTE(Synthetic Minority Over-sampling Technique)를 적용하였다.

- 이진 분류: Normal과 Abnormal 클래스 간의 데이터 비율을 균형 있게 조정하였다.
- 다중 분류: 모든 클래스에 대해 동일한 데이터 수를 확보하여 균형을 맞췄다.

SMOTE 적용 전후의 분포를 시각화한 결과, 클래스 간 데이터 비율이 개선되었음을 확인할 수 있었다.

3.2 모델 선정 및 구현 과정

특성 중요도 분석, 데이터 불균형 대응, 그리고 시간적 흐름과 특성 간 상호작용을 모두 고려하기 위해 머신러닝과 딥러닝 모델을 선정했다. 그중 내가 맡은 모델은 Random Forest, Extra Trees, 그리고 LSTM과 GRU이다.

```
# 이진 분류의 클래스 라벨을 문자열로 변환하기 위해 LabelEncoder 사용
label_encoder = LabelEncoder()
label_encoder.fit(['Normal', 'Attack'])  # 이진 분류 클래스에 대한 라벨 설정

# 모델 훈련 및 성능 평가
start_train = time.time()
rf_binary = RandomForestClassifier(random_state=42)
rf_binary.fit(X_train_binary_balanced, y_train_binary_balanced)
end_train = time.time()

start_predict = time.time()
y_pred_binary_rf = rf_binary.predict(X_test_binary_selected)
end_predict = time.time()

# 성능 지표 출력
print("Random Forest - Binary Classification Performance:")
print("Accuracy:", accuracy_score(y_test_binary, y_pred_binary_rf))
print("Classification Report:")
print(classification_report(y_test_binary, y_pred_binary_rf, target_names=label_encoder.classes_))

# 혼동 행렬 계산
conf_matrix = confusion_matrix(y_test_binary, y_pred_binary_rf)

# 혼동 행렬을 퍼센트로 변환
conf_matrix_percent = conf_matrix.astype('float') / conf_matrix.sum(axis=1)[:, np.newaxis] * 100

# 혼동 행렬 퍼센트 시각화
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_percent, annot=True, fmt=".2f", cmap="Blues",
            xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_,
            cbar_kws={'label': 'Percentage (%)'})
plt.title("Random Forest - Binary Classification Confusion Matrix (Percentage)")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()

# 성능 데이터 저장
performance_data.append({
    "Model": "Random Forest",
    "Type": "Binary",
    "Accuracy": accuracy_score(y_test_binary, y_pred_binary_rf),
    "Recall": recall_score(y_test_binary, y_pred_binary_rf),
    "Precision": precision_score(y_test_binary, y_pred_binary_rf),
    "F1-Score": f1_score(y_test_binary, y_pred_binary_rf),
    "Time to Train": end_train - start_train,
    "Time to Predict": end_predict - start_predict,
    "Total Time": (end_train - start_train) + (end_predict - start_predict)
})

# 모델 저장
joblib.dump(rf_binary, '/content/drive/My Drive/UNSW_NB15/rf_binary_model_v4.pkl')
print("Model saved as rf_binary_model.pkl")
```

Random Forest를 통해 학습 데이터를 모델에 훈련시키고, 테스트 데이터를 예측했다. 성능은 정확도, 정밀도, 재현율, F1 점수로 평가했으며, 혼동 행렬(Confusion Matrix)을 시각화했다. 학습 시간 및 예측 시간을 측정해 성능 데이터로 저장했다.

```
label_decoder = LabelEncoder()
label_decoder.fit(data_multi['attack_cat'])

# 모델 훈련 및 성능 평가
start_train = time.time()
rf_multi = RandomForestClassifier(random_state=42)
rf_multi.fit(X_train_multi_balanced, y_train_multi_balanced)
end_train = time.time()

start_predict = time.time()
y_pred_multi_rf = rf_multi.predict(X_test_multi_selected)
end_predict = time.time()

# 성능 지표 출력
print("Random Forest - Multi-class Classification Performance:")
print("Accuracy:", accuracy_score(y_test_multi, y_pred_multi_rf))
print("Classification Report:")
print(classification_report(y_test_multi, y_pred_multi_rf, target_names=label_decoder.classes_))

# 혼동 행렬 계산
conf_matrix = confusion_matrix(y_test_multi, y_pred_multi_rf)

# 혼동 행렬을 퍼센트로 변환
conf_matrix_percent = conf_matrix.astype('float') / conf_matrix.sum(axis=1)[:, np.newaxis] * 100

# 퍼센트를 이용한 혼동 행렬 시각화
plt.figure(figsize=(12, 10))
sns.heatmap(conf_matrix_percent, annot=True, fmt=".2f", cmap="Blues",
            xticklabels=label_decoder.classes_, yticklabels=label_decoder.classes_,
            cbar_kws={'label': 'Percentage (%)'})
plt.title("Random Forest - Multi-class Confusion Matrix (Percentage)")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()

# 성능 데이터 저장
performance_data.append({
    "Model": "Random Forest",
    "Type": "Multi-class",
    "Accuracy": accuracy_score(y_test_multi, y_pred_multi_rf),
    "Recall": None,  # Per-class recall can be extracted from classification_report
    "Precision": None,
    "F1-Score": f1_score(y_test_multi, y_pred_multi_rf, average='weighted'),
    "Time to Train": end_train - start_train,
    "Time to Predict": end_predict - start_predict,
    "Total Time": (end_train - start_train) + (end_predict - start_predict)
})

# 모델 저장
joblib.dump(rf_multi, '/content/drive/My Drive/UNSW_NB15/rf_multi_model_v4.pkl')
print("Model saved as rf_multi_model.pkl")
```

다중 분류에 적용해 동일한 방식으로 모델 학습 및 평가를 수행했다.

```
# 데이터의 실제 shape 가져오기
first_dim = X_train_binary_balanced.shape[0]  # 샘플 수 (첫 번째 차원)
last_dim = 20  # 특징 수 (상위 20개 피처)

# 전체 요소 수 계산
total_elements = X_train_binary_balanced.values.size

# 중간 차원 (step) 계산
step = total_elements // (first_dim * last_dim)

# 계산 결과 출력
print(f"First dimension (samples): {first_dim}")
print(f"Last dimension (features): {last_dim}")
print(f"Calculated step (timesteps): {step}")
```

```
First dimension (samples): 139582
Last dimension (features): 20
Calculated step (timesteps): 1
```

데이터의 시간적 구조를 학습하기 위해 전체 데이터의 크기와 차원을 기반으로 step(timestep)을 계산했다. 이를 통해 각 샘플에 대해 적절한 시간적 차원을 설정했다.

```
# 데이터 3D로 변환
X_train_binary_lstm = X_train_binary_balanced.values.reshape(X_train_binary_balanced.shape[0], 1, X_train_binary_balanced.shape[1])
X_test_binary_lstm = X_test_binary_selected.values.reshape(X_test_binary_selected.shape[0], 1, X_test_binary_selected.shape[1])

# 모델 훈련 및 성능 평가
start_train = time.time()

# LSTM 모델 정의
lstm_binary = Sequential([
    LSTM(64, input_shape=(1, X_train_binary_balanced.shape[1]), return_sequences=True),
    LSTM(32),
    Dense(1, activation='sigmoid')
])

# 모델 컴파일
lstm_binary.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# 훈련 (검증 세트를 사용하여 성능 평가)
history = lstm_binary.fit(
    X_train_binary_lstm, y_train_binary_balanced,
    epochs=50, batch_size=64,
    validation_split=0.2,  # 20%의 데이터를 검증 세트로 사용
    verbose=1
)

end_train = time.time()
```

LSTM 모델의 입력 요구사항에 맞게 데이터를 3차원(samples, timesteps, features)으로 변환했다. 이를 통해 네트워크 트래픽 데이터의 시간적 흐름을 학습할 수 있도록 준비했다. LSTM 계층과 sigmoid 활성화 함수로 구성된 모델을 설계하였으며, 손실 함수는 binary_crossentropy, 최적화 도구는 adam을 사용했다. 검증 데이터를 포함하여 학습(fit)을 진행하며, 성능 평가를 위해 학습 및 검증 손실, 정확도를 기록했다.

```python
# 예측
start_predict = time.time()
lstm_binary_loss, lstm_binary_accuracy = lstm_binary.evaluate(X_test_binary_lstm, y_test_binary)
end_predict = time.time()

print("LSTM - Binary Classification Performance:")
print("Test Accuracy:", lstm_binary_accuracy)
print("Test Loss:", lstm_binary_loss)

# 훈련 중 성능 추적
print("Training History - Loss and Accuracy:")
print("Training Loss:", history.history['loss'][-1])  # 마지막 에포크의 훈련 손실
print("Training Accuracy:", history.history['accuracy'][-1])  # 마지막 에포크의 훈련 정확도
print("Validation Loss:", history.history['val_loss'][-1])  # 마지막 에포크의 검증 손실
print("Validation Accuracy:", history.history['val_accuracy'][-1])  # 마지막 에포크의 검증 정확도

# 성능 데이터 저장
performance_data.append({
    "Model": "LSTM",
    "Type": "Binary",
    "Accuracy": lstm_binary_accuracy,
    "Recall": None,  # F1, Recall 등 추가적으로 계산해야 하는 경우
    "Precision": None,
    "F1-Score": None,
    "Time to Train": end_train - start_train,
    "Time to Predict": end_predict - start_predict,
    "Total Time": (end_train - start_train) + (end_predict - start_predict)
})

# 모델 저장
lstm_binary.save('/content/drive/My Drive/UNSW_NB15/lstm_binary_model_v4.h5')
print("Model saved as lstm_binary_model_with_validation.h5")
```

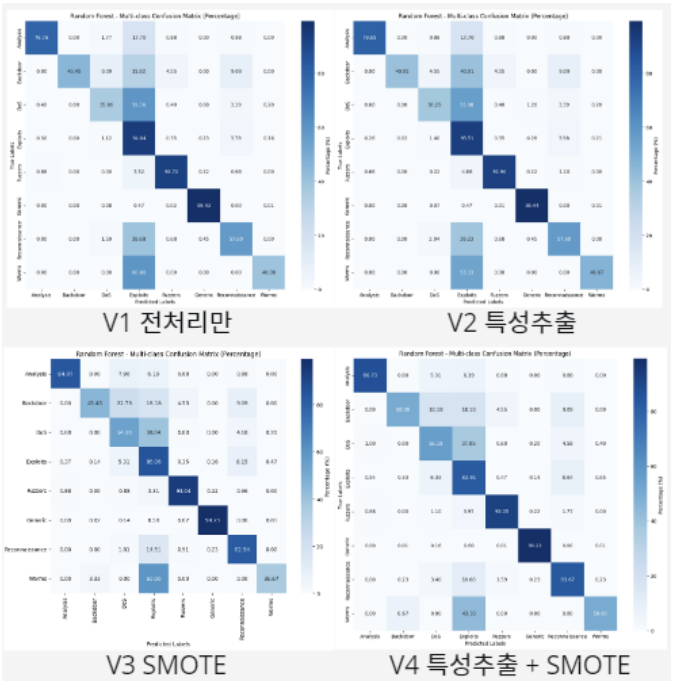
LSTM 모델을 테스트 데이터에 대해 평가하여 정확도와 손실 값을 도출했다. 학습과 검증 과정에서의 손실(loss) 및 정확도(accuracy)를 마지막 에포크 기준으로 추적하며, 이를 통해 모델 성능을 정량적으로 분석하였다. 최종적으로 훈련 시간과 예측 시간을 포함한 성능 데이터를 저장하였다.

```python
# 데이터 3D로 변환
X_train_binary_gru = X_train_binary_balanced.values.reshape(X_train_binary_balanced.shape[0], 1, X_train_binary_balanced.shape[1])
X_test_binary_gru = X_test_binary_selected.values.reshape(X_test_binary_selected.shape[0], 1, X_test_binary_selected.shape[1])

# GRU 모델 정의
start_train = time.time()
gru_binary = Sequential([
    GRU(64, input_shape=(1, X_train_binary_balanced.shape[1]), return_sequences=True),
    GRU(32),
    Dense(1, activation='sigmoid')
])

# 모델 컴파일
gru_binary.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# 훈련
gru_binary.fit(X_train_binary_gru, y_train_binary_balanced, epochs=50, batch_size=64, verbose=1)

end_train = time.time()

# 예측
start_predict = time.time()
gru_binary_loss, gru_binary_accuracy = gru_binary.evaluate(X_test_binary_gru, y_test_binary)
end_predict = time.time()
```

GRU 모델은 LSTM과 유사한 구조로 설계되었으며, 계산 효율성을 높이기 위해 GRU 계층을 활용했다. 데이터를 3차원으로 변환한 후, GRU 계층과 sigmoid 활성화 함수를 사용하여 모델을 정의했다. 손실 함수로 binary_crossentropy, 최적화 도구로 adam을 사용하여 학습을 진행했다.

3.3 결과 분석

데이터 처리와 전처리 방식의 차이가 모델 성능에 미치는 영향을 분석하기 위해 4가지 방식(V1~V4)을 비교하였다.



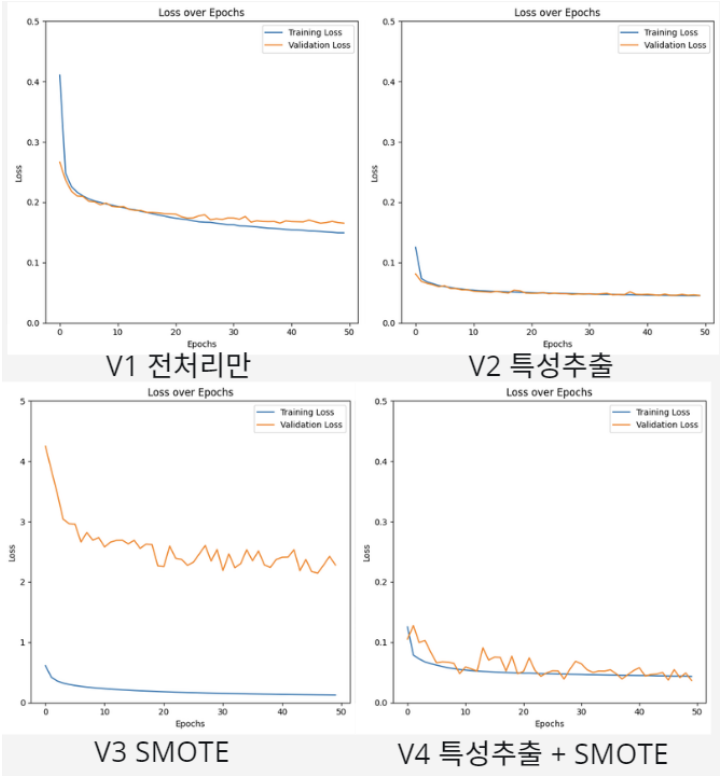
[Random Forest 모델의 버전별 성능 차이]

V1(전처리만): 기본 전처리만 적용한 경우, 소수 클래스 탐지 성능이 낮아 전반적인 분류 성능에 한계가 나타난다.

V2(특성 추출): 주요 특성을 추출한 경우, 성능이 일부 개선되었으나 소수 클래스에서는 여전히 낮은 탐지율을 보인다.

V3(SMOTE): 데이터 불균형 문제를 해결하기 위해 SMOTE를 적용한 결과, Backdoor와 Worms 등의 탐지 성능이 개선된다.

V4(특성 추출 + SMOTE): 두 가지 기법을 결합한 경우, 소수 클래스(Backdoor +4.55%, Reconnaissance +26.07%, Worms +10.00%)의 탐지 성능이 크게 향상되며, 주요 클래스(Generic)의 높은 성능(99.22%)도 유지된다.



[LSTM 모델의 버전별 성능 차이]

V1(전처리만): 학습 손실과 검증 손실이 안정적으로 감소하나, 에포크 후반부에서도 검증 손실이 다소 높은 상태를 유지하여 일부 과적합 가능성이 나타난다.

V2(특성 추출): 학습과 검증 손실 모두 빠르게 감소하며, 검증 손실이 비교적 낮은 수준에서 수렴한다. 특성 추출이 모델 학습 효율에 긍정적 영향을 미친 것으로 판단된다.

V3(SMOTE): 학습 손실은 안정적으로 감소하지만, 검증 손실의 변동이 커져 모델의 일반화 성능에 영향을 미칠 가능성이 나타난다. 또한 학습 손실과 검증 손실의 차이가 크다.

V4(특성 추출 + SMOTE): 훈련 손실과 검증 손실이 모두 낮은 값에서 시작하며, 최종적으로 약 0.05~0.1에서 수렴한다. 검증 손실이 초기 학습 단계에서 변동이 크지만, 에포크가 진행됨에 따라 점차 안정화된다.

이러한 비교를 모든 모델에 대해 진행하였고, 그 결과 모델별, 상황별로 최적의 데이터 처리 방식을 도출할 수 있었다. 이를 통해 특정 데이터 처리 방식이 모델 성능에 미치는 영향을 정량적으로 분석하고, 각 모델이 다양한 상황에서 효과적으로 활용될 수 있는 방안을 제시하였다.

4. 결론

4.1 성능 평가

침입 탐지 시스템에서는 빠른 탐지와 정확한 분류가 핵심적인 요구사항이다. 이에 따라 성능 평가 시 정확도, 재현율, 정밀도, F1-score와 같은 일반적인 지표 외에도 학습 및 예측 시간과 같은 시간 효율성을 추가적으로 분석하였다. 이를 통해 모델별 데이터 처리 방식이 탐지 시스템의 효율성과 성능에 미치는 영향을 전반적으로 비교하고, 실시간 시스템 구현 시 적합한 모델과 데이터 처리 방식을 도출하고자 하였다.

Model Performance Summary

	Model	Type	Accuracy	Recall	Precision	F1-Score	Time to Train	Time to Predict	Total Time
0	Random Forest	Binary	98.6249%	0.991346	0.990324	0.990835	16.77	0.177285	16.94
1	Random Forest	Multi-class	94.7387%	nan	nan	0.944196	17.16	0.295039	17.45
2	Extra Trees	Binary	98.5304%	0.991346	0.989078	0.990211	6.02	0.261523	6.29
3	Extra Trees	Multi-class	94.7043%	nan	nan	0.943358	9.07	0.357153	9.42
4	LSTM	Binary	97.6924%	nan	nan	nan	403.25	1.384077	404.64
5	LSTM	Multi-class	93.6841%	nan	nan	nan	315.71	2.671994	318.38
6	GRU	Binary	97.8901%	nan	nan	nan	437.96	1.411583	439.37
7	GRU	Multi-class	93.5752%	nan	nan	nan	337.58	1.792127	339.37

[전처리]

Model Performance Summary

	Model	Type	Accuracy	Recall	Precision	F1-Score	Time to Train	Time to Predict	Total Time
0	Random Forest	Binary	98.5175%	0.990543	0.989692	0.990118	12.62	0.136334	12.75
1	Random Forest	Multi-class	94.6985%	nan	nan	0.944321	14.02	0.179031	14.19
2	Extra Trees	Binary	98.4014%	0.989970	0.988724	0.989346	6.97	0.231371	7.20
3	Extra Trees	Multi-class	94.5839%	nan	nan	0.943168	5.57	0.341693	5.91
4	LSTM	Binary	97.7655%	nan	nan	nan	348.89	1.343613	350.23
5	LSTM	Multi-class	93.7701%	nan	nan	nan	315.92	1.916590	317.84
6	GRU	Binary	97.9889%	nan	nan	nan	430.08	3.072097	433.16
7	GRU	Multi-class	93.7643%	nan	nan	nan	332.85	2.191885	335.04

[전처리 + 특성추출]

Model Performance Summary

	Model	Type	Accuracy	Recall	Precision	F1-Score	Time to Train	Time to Predict	Total Time
0	Random Forest	Binary	98.6120%	0.989053	0.992409	0.990728	30.89	0.331052	31.22
1	Random Forest	Multi-class	93.6210%	nan	nan	0.939169	302.54	0.334587	302.88
2	Extra Trees	Binary	98.5304%	0.988996	0.991382	0.990188	13.27	0.514757	13.78
3	Extra Trees	Multi-class	93.2772%	nan	nan	0.935549	76.66	0.453404	77.11
4	LSTM	Binary	97.8213%	nan	nan	nan	645.68	1.885014	647.57
5	LSTM	Multi-class	91.1509%	nan	nan	nan	1755.82	2.685532	1758.51
6	GRU	Binary	97.8729%	nan	nan	nan	755.38	1.539457	756.92
7	GRU	Multi-class	90.4917%	nan	nan	nan	1769.36	1.359433	1770.72

[전처리 + SMOTE]

Model Performance Summary

	Model	Type	Accuracy	Recall	Precision	F1-Score	Time to Train	Time to Predict	Total Time
0	Random Forest	Binary	98.5475%	0.988423	0.992176	0.990296	21.81	0.172932	21.98
1	Random Forest	Multi-class	93.1625%	nan	nan	0.935794	281.66	0.415768	282.07
2	Extra Trees	Binary	98.4702%	0.988308	0.991262	0.989783	11.10	0.265296	11.36
3	Extra Trees	Multi-class	92.8359%	nan	nan	0.932458	63.67	0.563854	64.23
4	LSTM	Binary	97.8987%	nan	nan	nan	636.73	2.294092	639.03
5	LSTM	Multi-class	89.5060%	nan	nan	nan	1833.72	1.749470	1835.47
6	GRU	Binary	97.8041%	nan	nan	nan	770.64	3.137188	773.77
7	GRU	Multi-class	89.7524%	nan	nan	nan	1789.59	1.942874	1791.54

[전처리 + 특성추출 + SMOTE]

분석 결과, 데이터의 복잡성과 시간 효율성 요구사항에 따라 적합한 모델과 처리 방식이 다르게 나타났다. 간단한 데이터셋에서는 Random Forest(V4)가 이진 분류에서, Random Forest(V2)가 다중 분류에서 우수한 성능을 보였다. 반면, 복잡한 데이터셋에서는 시간적 특성을 학습하는 LSTM/GRU(V2)가 이진 분류와 다중 분류 모두에서 높은 성능을 기록하였다. 시간 효율성이 중요한 경우, Extra Trees(V2)는 빠른 학습과 예측 속도를 제공하며 효율적인 선택으로 확인되었다. 최대 성능을 목표로 하는 경우에는 소수 클래스 탐지와 전반적인 성능에서 Random Forest(V4)가 이진 분류에, Random Forest(V2)가 다중 분류에 가장 적합하였다.

이러한 분석을 통해 침입 탐지 시스템에 최적화된 모델과 데이터 처리 방식을 상황별로 제시할 수 있었으며, 개선이 필요한 부분과 적용 가능성을 기반으로 실질적인 발전 방향을 도출하였다.

4.2 적용 가능성 검토 적용 가능성 검토

침입 탐지 시스템의 핵심 요소인 실시간 탐지 능력, 정확도와 오탐률, 적응성과 학습 능력, 확장성, 그리고 다양한 공격 유형 탐지를 기준으로 상황별 최적의 모델을 도출하였다. 이를 기반으로 각 요소에 적합한 모델과 필요한 향후 과제를 다음과 같이 정리하였다.

요소	적합 모델	설명	향후 과제
실시간 탐지 능력	RF, DT	예측 속도가 매우 빠르고 실시간 탐지에 적합	딥러닝 모델(LSTM, GRU)의 학습 및 예측 시간 최적화 필요
정확성과 오탐률	XGBoost, RF	SMOTE 및 하이퍼파라미터 튜닝으로 높은 정확도와 낮은 오탐률	오탐률을 더욱 줄이기 위해 데이터 전처리 및 분류기 튜닝 방식 고도화 필요
적응성과 학습 능력	XGBoost	하이퍼파라미터 튜닝으로 학습 성능 강화	딥러닝 모델의 데이터 업데이트 자동화 및 학습 속도 최적화 필요
확장성	RF, ET	학습 및 예측 시간이 짧고 병렬 처리 지원으로 대규모 네트워크 환경에 적합	딥러닝 모델(CNN, LSTM, GRU)의 학습 시간 개선 및 대규모 데이터 처리 최적화 필요
다양한 공격 유형 탐지	XGBoost, CNN	XGBoost는 이진 및 다중 클래스 분류에서 뛰어난 성능을 보임 CNN은 공간적 패턴 학습에 강점	CNN의 학습 속도 및 효율성을 개선해 복잡한 공격 유형 탐지 성능 강화

이를 통해 침입 탐지 시스템에서 요구되는 다양한 요소를 충족하기 위해 상황별로 적합한 모델과 기술적 과제를 파악하고, 최적의 적용 방안을 제시할 수 있었다.

4.3 한계점 및 개선방향

이번 연구는 다양한 모델과 데이터 처리 방식을 활용하여 침입 탐지 시스템의 성능을 분석하고 최적의 적용 방안을 도출하는 데 초점을 맞췄다. 그러나 몇 가지 한계점이 존재하며, 이를 보완하기 위한 개선 방향은 다음과 같다:

1. **데이터셋의 일반화 한계**
    - **한계점**: 연구에 사용된 UNSW-NB15 데이터셋은 특정 네트워크 환경을 기반으로 수집된 데이터로, 모든 네트워크 환경에서 동일한 성능을 보장하기 어렵다.
    - **개선 방향**: 다양한 네트워크 환경에서 수집된 데이터셋을 추가적으로 활용하여 모델의 일반화 능력을 향상시키고, 실제 환경에서의 적용 가능성을 높인다.
2. **실시간 처리 성능 부족**
    - **한계점**: 딥러닝 모델(LSTM, GRU)은 복잡한 데이터에서 우수한 성능을 보였으나, 학습 및 예측 속도가 느려 실시간 탐지에 적용하기에는 제약이 있다.
    - **개선 방향**: 경량화된 딥러닝 모델 또는 하드웨어 가속 기술(GPU, TPU)을 활용하여 학습 및 예측 속도를 최적화한다.
3. **오탐률 및 소수 클래스 성능**
    - **한계점**: 일부 소수 클래스(Backdoor, Worms)에서는 여전히 낮은 탐지율을 보이며, 오탐률이 완벽히 제거되지 않았다.
    - **개선 방향**: SMOTE와 같은 데이터 불균형 해결 기법의 고도화 및 앙상블 모델의 튜닝을 통해 소수 클래스 탐지율과 오탐률을 개선한다.
4. **다중 클래스 탐지의 복잡성**
    - **한계점**: 다중 클래스 분류에서는 특정 클래스 간 혼동이 발생하여 정확도가 상대적으로 낮아지는 경향이 나타났다.
    - **개선 방향**: 클래스 간 차별성을 높이기 위한 추가적인 데이터 전처리 및 하이퍼파라미터 최적화를 수행한다.

이번 연구는 침입 탐지 시스템의 성능 향상을 위한 기초적인 분석과 방법론을 제시하였으며, 향후 연구에서는 이러한 한계점을 보완하여 실제 네트워크 환경에서도 효과적으로 활용할 수 있는 솔루션을 제안하는 데 중점을 두어야 할 것이다.