Intrusion Detection System

Machine Learning Projekt SoSe 23

Name: Marc Schmidt, Nicola Jäger, Jonathan Ebinger

Studiengang: IT-Sicherheit

Matrikelnummer: 103629, 105853, 105859

6. August 2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Abstract

In den letzten Jahren haben Angriffe auf Netzwerke und an das Internet angeschlossene Systeme drastisch zugenommen. Aus diesem Grund wird die Analyse von Datenpaketen in Netzwerken zu einem zentralen Element bei der Abwehr von Angriffen aus dem Netz. Zur Paketanalyse werden sogenannte Intrusion Detection Systeme (IDS) eingesetzt. Ein IDS spielt eine entscheidende Rolle beim Schutz von Computernetzwerken, indem es beispielsweise Unregelmäßigkeiten im Datenverkehr oder unberechtigte Zugriffe erkennt und verhindert. Dabei werden auch potenzielle Bedrohungen für interne Benutzer berücksichtigt.

Ziel dieses Projekts ist die Entwicklung eines Vorhersagemodells mithilfe von maschinellem Lernen, das zwischen bösartigem und gutartigem Netzwerkverkehr unterscheiden kann.

Der für das Training verwendete Datensatz ist „Aposemat IoT-23“[1], der durch das Stratosphere Laboratory, AIC Group, (CTU University in Czech Republic) mit finanzieller Unterstützung von der Avast Software[2] erstellt wurde (vgl. Kapitel 3).

Das Forschungsprojekt konnte zeigen, dass Vorhersagemodelle für den verwendeten Datensatz trainiert werden können. Auch bei einer Kombination von mehreren Datensätzen kann beispielsweise das KNN-Model Genauigkeiten von über 90% erzielen.

[1] Sebastian Garcia, Agustin Parmisano and Maria Jose Erquiaga. (2020). IoT-23: A labeled dataset with malicious and benign IoT network traffic (Version 1.0.0) [Data set]. Zenodo. `http://doi.org/10.5281/zenodo.4743746`

[2] Avast ist Hersteller für Sicherheitssoftware und Dienstprogramme, welcher unter anderem die Avast-Antivirensoftware vermarktet

2 Aufbau und Vorgehen

Das Projekt teilt sich in zwei Teile auf:

1. Die erste Aufgabe bestand darin, ein maschinelles Lernmodell auf Basis des oben genannten Datensatzes auszuwählen und zu trainieren. Hierbei wurden verschiedenen Merkmale und Muster analysiert, um effektiv zwischen normalen Verbindungen und Angriffen unterscheiden zu können. Diese Dokumentation befasst sich mit drei verschiedenen Lernmodellen, die jeweils auf zwei verschiedenen Datensätzen trainiert wurden (vgl. Kapitel 5.3).
   - K-Nearest Neighbors Algorithm (vgl. Kapitel 7.1 und 8.1)
   - Decision Trees (vgl. Kapitel 7.2 und 8.2)
   - Artificial Neural Network (vgl. Kapitel 7.3 und 8.3)

   Ziel war es, ein optimales Modell zu entwickeln, das eine möglichst hohe Genauigkeit bei der Erkennung von Angriffen aufweist. Die verwendeten Daten sind in zwei Kategorien unterteilt:
   - schlechte Verbindung:
     Unter einer schlechten Verbindung versteht man Datenverkehr, der sich negativ auf die Leistung, Geschwindigkeit und Verfügbarkeit von Anwendungen auswirken kann. Auch die Schutzziele der Informationssicherheit (Vertraulichkeit, Integrität, Verfügbarkeit) von Daten und Anwendungen können durch diesen Verkehr gefährdet werden. Beispiele hierfür sind Netzwerkangriffe wie DDoS-Attacken oder Verbindungen von Botnetzen wie (Mirai oder Torii). Aber auch anderer Verkehr, der für Phishing, Spam oder die Übertragung von Malware verwendet wird, kann als bösartiger Verkehr angesehen werden.
   - gute Verbindung:
     Gute Verbindungen ist Datenverkehr, der keine negativen Auswirkungen auf Anwendungen und Dienste hat. Dazu gehört z.B. das Laden einer Webseite oder das Versenden von E-Mails ohne böswillige Absicht. Auch jeder andere Verkehr, der nicht darauf abzielt, anderen Schaden zuzufügen oder die Schutzziele zu verletzen, wird als guter Verkehr definiert.

2. Die zweite Aufgabe bestand darin, die verschiedenen Modelle durch Hyperparameteroptimierung und andere Optimierungen zu verbessern, um eine maximale Genauigkeit der Modelle zu erreichen. Außerdem sollte ein Vergleich aller Ergebnisse durchgeführt werden, um das am besten geeignete Modell für die Datensätze zu finden.

3 Datensatzauswahl

Im Rahmen dieses Forschungsprojektes wurden verschiedene Datensätze betrachtet und analysiert. Ein wichtiges Kriterium war, einen Datensatz zu finden, dessen Daten bereits beschriftet sind, um sie später leicht für das Training verwenden zu können. Aus Zeitgründen wurde auf die Auswahl eines unbeschrifteten Datensatzes verzichtet. Die Wahl fiel schließlich auf den *Iot-23* Datensatz [GPE20], der zudem eine Vielzahl unterschiedlicher Dateiformate enthielt, die leicht ausgewertet und während der Verarbeitung in ein Dateframe-Objekt konvertiert werden konnten. Die Daten im Datensatz waren als Textdatei mit Tabulatoren getrennt gespeichert, was das Einlesen erleichterte. Dennoch mussten die Daten aufbereitet werden, da einige Zeilen ganz oder teilweise leer waren. Von besonderem Interesse war, dass der Datensatz in viele verschiedene Datensätze aufgeteilt war, was die Kombination mehrerer Datensätze ermöglichte, um die Modelle noch intensiver zu testen.

Im Folgenden wird der verwendete Datensatz beschrieben:

Das IoT-23-Dataset stellt eine Sammlung von Netzwerkverkehr aus Internet of Things (IoT)-Geräten dar. Es besteht aus 20 Aufnahmen infizierten IoT-Netzwerkverkehrs, der auf Malware-Infektionen zurückzuführen ist, sowie 3 Aufnahmen von normalem IoT-Verkehr. Die Aufnahmen wurden zwischen 2018 und 2019 im Stratosphere Laboratory der AIC Group, FEL, CTU University in der Tschechischen Republik erfasst.

Die Aufnahmen des unbedenklichen IoT-Verkehrs wurden durch Erfassen des Netzwerkverkehrs von drei verschiedenen IoT-Geräten erzeugt: einer Philips HUE-Smarten-LED-Lampe, einem Amazon Echo-Heimassistenten und einem Somfy-Smarten-Türschloss. Sowohl die bösartigen als auch die unbedenklichen Szenarien wurden in einer kontrollierten Netzwerkumgebung mit uneingeschränkter Internetverbindung ausgeführt, ähnlich wie bei jedem anderen realen IoT-Gerät.

Ziel dieses Datensatzes ist es, der Community sowohl bösartigen als auch sauberen Netzwerkverkehr zur Verfügung zu stellen. Die gelabelten Flussdaten bieten Einblicke in das Verhalten von IoT-Geräten im Netzwerk und dienen als Grundlage für die Entwicklung maschineller Lernalgorithmen zur Erkennung von IoT-Malware und zur Analyse von gutartigem IoT-Datenverkehr.

4 Struktur des Repositorys

Alle folgenden Quellcode-Beispiele sind zusätzlich in einem Jupyter-Notebook verfügbar. Das Repository[1] enthält darüber hinaus weitere Exporte einzelner trainierter Modelle, die Abschlusspräsentation sowie verwendete Bilder und Grafiken.

- /images/

  Enthält alle verwendeten Bilder und Grafiken aus dem Jupyter-Notebook.

- /jupyter_notebook/

  Enthält das Jupyter-Notebook, auf dem diese Ausarbeitung basiert.

- /presentation/

  Enthält unsere Abschlusspräsentation, die am 11.07.23 im Zusammenhang mit dem Abschluss des Moduls gehalten wurde.

- /saved_models/

  Enthält alle Trainierte Modelle (ANN, KNN, DTress) auf jeweils beiden Datensätzen im joblib-Format und die Tensorflow-Modelle im SavedModel-Format.

[1] https://github.com/tech-nickel/LSC-IDS

5 Normalisierung

Dieses Kapitel soll einen Überblick über den Weg der Dateien von den reinen unformatierten „.log“ Dateien des ausgewählten Datensatzes zu den verwendeten Datensätzen, mit denen die Modelle trainiert wurden, geben.

5.1 Normalisierung

In Tabelle 5.1 ist eine Beispielzeile aus einer Datei eines Netzwerk-Captures dargestellt.

Spalte	Beispielwert
ts	1525879831.015811
uid	CUmrqr4svHuSXJy5z7
id.orig_h	192.168.100.103
id.orig_p	51524
id.resp_h	65.127.233.163
id.resp_p	23
proto	tcp
service	-
duration	2.999051
orig_bytes	0
resp_bytes	0
conn_state	S0
local_orig	-
local_resp	-
missed_bytes	0
history	S
orig_pkts	3
orig_ip_bytes	180
resp_pkts	0
resp_ip_bytes	0
tunnel_parents	-
label	Malicious
detailed-label	PartOfAHorizontalPortScan

Tab. 5.1: Eine Beispielzeile aus einer der .log Dateien

Um die Daten aus den „.log“ Dateien zu konvertieren, wurden mehrere Funktionen angewendet, um die Daten in das richtige Format zu bringen. Die unten aufgeführten Spalten wurden auf eine kleinere Auswahl reduziert. Diese Auswahl ist auch in **??** sichtbar.
Um die Spalten, die nicht numerisch waren, effektiv von den Modellen verarbeiten zu lassen, wurden die Funktionen aus Code 1 angewendet.

Schließlich wurde mit der Funktion Code 2 eine Konvertierung in ein Dataframe vorgenommen. Außerdem wurden die Labels der einzelnen Flows zu einem Integer konvertiert.
Um nun die Datensätze besser transportieren zu können und nicht immer wieder aus den Originaldaten neu berechnen zu müssen, wurden die Dataframes als CSV-Dateien exportiert. Die Dateien konnten dann für mehrere Trainings gleichzeitig verwendet werden.

Originale Spalten:

- timestamp
- UID
- orig_h
- orig_p
- resp_h
- resp_p
- proto
- service
- duration
- orig_bytes
- resp_bytes
- conn_state
- local_orig
- local_resp
- missed_bytes
- history
- orig_pkts
- orig_ip_bytes
- resp_pkts
- resp_ip_bytes
- label
- detailed-label

Genutzte Spalten für die Modelle:

- orig_h
- orig_p
- resp_h
- resp_p
- proto
- label

```
protos = list(df_raw_log.proto.unique().flatten())
conn_states = list(df_raw_log.conn_state.unique().flatten())
histories = list(df_raw_log.history.unique().flatten())

def convert_ipv4(addr):
    """Konvertieren einer IPv4 Adresse zu einem INT"""
    return struct.unpack("!I", socket.inet_aton(addr))[0]

def con_proto(proto):
    """Konvertieren der Ports ICMP, TCP und UDP zu einem INT"""
    return protos.index(proto)

def convert_connstate(state):
    return conn_states.index(state)

def convert_histories(history):
    return histories.index(history)
```

Code 1: Konvertierungsfunktionen

```
def norm_df(df):
    """
    Normierung des DataFrames, Löschen der Spalten, welche nicht gebraucht werden
    ↪ und konvertieren der Object-Spalten zu dem Typen Int
    """
    df = df.replace('-', np.nan)
    df['orig_h'] = df['orig_h'].apply(convert_ipv4)
    df['resp_h'] = df['resp_h'].apply(convert_ipv4)
    df['proto'] = df['proto'].apply(con_proto)
    df['conn_state'] = df['conn_state'].apply(convert_connstate)
    df['history'] = df['history'].apply(convert_histories)
    df = df.drop(['uid', 'ts', 'tunnel_parents', 'detailed-label', 'service',
    ↪  'duration', 'orig_bytes', 'resp_bytes', 'local_orig', 'local_resp'],
    ↪  axis=1)
    df['label'] = (df['label'] == "Malicious").astype(int)
    df = df.dropna()
    df = df.reset_index(drop=True)
    return df
```

Code 2: Normierung der Daten

5.2 Zusammenführen der Daten

Nachdem eine Auswahl aller verfügbaren Datensätze in CSV-Dateien konvertiert wurde, sollen diese Dateien nun zusammengeführt werden, um das Training variabler zu gestalten und das Potenzial der Modelle zu testen. Um diesen Datensatz zu erhalten, wird die Funktion aus Code 3 verwendet. Diese geht nach folgendem Schema vor:

1. Sie nimmt eine Liste von Zahlen, die die IDs zu den einzelnen CSV-Dateien darstellen.
2. Die einzelnen Dateien werden, in *Malicious* und *Benign* unterteilt.
3. Enthält einer dieser Blöcke mehr als 500'000 Zeilen enthalten, werden die Daten zufällig aufgeteilt und auf 500'000 reduziert.
4. Nachdem alle Datensätze auf eine Größe von jeweils maximal 1'000'000 Zeilen reduziert wurden, wird der gesamte zusammengeführte Datensatz auf die gleiche Anzahl von *Malicious* und *Benign* reduziert. Die Kategorie mit den meisten Daten wird zufällig auf die Anzahl der anderen reduziert.

```
1  def merge(datasets: list):
2    df_return = pd.DataFrame()
3    for d in datasets:
4      df = pd.read_csv(f"work/csv/convert/capture{d}_1.csv")
5      l_dfs = [df[df['label'] == 0], df[df['label'] == 1]]
6      for label in [0,1]:
7        if len(l_dfs[label]) > 500_000:
8          percent = 500_000 / len(l_dfs[label])
9          l_dfs[label] = np.array_split(l_dfs[label].sample(frac=1), 1/percent)[0]
10     df = pd.concat(l_dfs, ignore_index=1)
11     df_return = pd.DataFrame(pd.concat([df_return,df],ignore_index = True))
12   ones = len(df_return[df_return['label'] == 1])
13   zeros = len(df_return[df_return['label'] == 0])
14   thebigger = 1 if ones > zeros else 0
15   perc = zeros / ones
16   perc =  1 - perc if perc > 1 else perc
17   l_dfs = [df_return[df_return['label'] == 0], df_return[df_return['label'] ==
     ↪  1]]
18   rows_to_cut = int(len(l_dfs[thebigger]) * perc)
19   l_dfs[thebigger] = l_dfs[thebigger].iloc[:rows_to_cut]
20   df_return = pd.concat(l_dfs)
21   return df_return
```

Code 3: Zusammenfassung mehrerer Datensätze zu einem

5.3 Verwendete Datensätze

Die folgenden Kapitel befassen sich mit dem Training der Modelle. In der Tabelle 5.2 sind die verwendeten Datensätze aufgelistet. *Datensatz 1* entspricht dem Datensatz *Capture-1-1 (Hide and Seek)*. *Datensatz 2 (merged)* entspricht allen in der Tabelle aufgeführten Datensätzen zusammen (siehe Kapitel 5.2).

Datensatz	Normal Traffic Flows	Malicious Traffic Flows
Capture-1-1 (Hide and Seek)	469.275	539.465
Capture-7-1 (Linux.Mirai)	75.955	11.378.759
Capture-35-1 (Mirai)_h	8.262.389	2.185.398

Tab. 5.2: Verwendete Datensätze

5.4 Sneak Peek der Daten

Die Tabelle 5.3 zeigt jeweils eine Zeile der Daten vor und nach der Konvertierung dar.

Spalte	vorher	nachher
id.orig_h	192.168.100.103	3232261223
id.orig_p	51524	51524
id.resp_h	65.127.233.163	1098901923
id.resp_p	23	23
proto	tcp	1
label	Malicious	1

Tab. 5.3: Beispielzeile der Daten vor und nach der Konvertierung

6 Generierung von Test-, Validierungs- und Trainingsdaten

6.1 Splitting und Oversampling

Um Test-, Validierungs- sowie Trainingsdaten zu generieren haben wir zwei eigene Funktionen implementiert, die einen Datensatz für das spätere Training und Testing aufgeteilt haben.
Die Funktion „split“, zu sehen in Code 4, dient dazu, einen gegebenen DataFrame (df) in drei Teilmengen aufzuteilen, um ein Modell zu trainieren, zu validieren und zu testen. Die Aufteilung erfolgt im Verhältnis 60-20-20 für Training, Validierung und Test. Wir verwenden die Split-Funktion für jedes unserer Modelle, um die Daten in einem ersten Schritt aufzuteilen, um damit weiter arbeiten zu können.

Im Folgenden werden die Funktionsschritte erläutert:

Zufällige Reihenfolge
: Zuerst wird der DataFrame (df) mit der Methode „sample(frac=1)“ zufällig gemischt, um sicherzustellen, dass die Daten nicht in einer bestimmten Reihenfolge vorliegen. Dadurch soll eine mögliche Struktur in den Daten aufgebrochen und das Modell robuster trainiert werden.

Aufteilung
: Wir verwenden die Funktion „np.split“, um den nun vorliegenden zufällig gemischten DataFrame in die drei Teilmengen aufzuteilen. Der erste Teil enthält 60% der Daten und wird für das Training verwendet. Der zweite Teil enthält die nächsten 20% und wird für die Validierung des Trainingsprozesses eingesetzt. Der letzte Teil enthält die verbleibenden 20% der Daten und dient ausschließlich zur abschließenden Bewertung der Leistung des Modells, nachdem es erfolgreich trainiert und validiert wurde.

```
1 def split(df):
2     train, valid, test = np.split(df.sample(frac=1), [int(0.6*len(df)),
      ↪  int(0.8*len(df))])
3     return train, valid, test
```

Code 4: Split-Funktion

Im zweiten Schritt werden sowohl die Train- als auch die Validierungsdaten durch die Scaling-Funktion in Code 5 skaliert und optional ein Oversampling durchgeführt.
Die Funktion verwendet die StandardScaler-Klasse aus der scikit-learn-Bibliothek, um die Merkmale (Features) des DataFrames zu normalisieren, sodass sie einen Nullmittelwert und eine Einheitsvarianz haben. Dadurch werden die Merkmale auf eine vergleichbare Skala gebracht und es wird verhindert, dass einige Merkmale aufgrund ihrer Skala dominieren und andere vernachlässigt werden.

Im Folgenden werden die einzelnen Schritte erläutert:

Trennung von Merkmalen und Labels
Zunächst werden die Labels von den Merkmalen (Features) im DataFrame entfernt. Die Features werden in der Variable „X“ gespeichert, während die Labels in der Variable „y“ gespeichert werden.

Skalierung der Merkmale
Im Scaling-Schritt wird, wie bereits erwähnt, der StandardScaler verwendet, um die Merkmale im DataFrame zu normieren.

Oversampling (optional)
Wenn der Parameter „oversample“ auf „True“ gesetzt ist, führt die Funktion zusätzlich eine „Überabtastung“ durch. Dies ist nützlich, wenn der ursprüngliche Datensatz eine unausgewogene Verteilung der Klassen aufweist, was wiederum zu Problemen bei der Modellierung führen kann. Die Funktion verwendet die RandomOverSampler-Klasse aus der imbalanced-learn-Bibliothek, um die Datensätze der unterrepräsentierten Klassen zu duplizieren, so dass die Klassen gleichmäßiger verteilt sind.

Zusammenführung von Merkmalen und Labels
Nach der Skalierung und optionalen Überabtastung werden die Merkmale (X) und das Label (y) wieder zu einem DataFrame „data“ zusammengeführt. Dabei werden die skalierten Features horizontal mit den Label-Daten verbunden.

```
1  X = pd.DataFrame()
2  def scale(df: pd.DataFrame, oversample = False):
3      X = df.drop(['label'], axis=1)
4      y = df['label']
5
6      scaler = StandardScaler()
7      X = scaler.fit_transform(X)
8
9      if oversample:
10         ros = RandomOverSampler()
11         X, y = ros.fit_resample(X, y)
12
13     data = np.hstack((X, np.reshape(y, (-1, 1))))
14     return data, X, y
```

Code 5: Scale-Funktion

6.2 Erstellung der Datensätze

Mit den Aufrufen, die im Code 6 zu sehen sind, wurden nun die Eingangs in den Code 4 und Code 5 gezeigten Funktionen verwendet, um die benötigten Datensätze zu erzeugen.

```
1 train, valid, test = split(DataFrame)
2 train, X_train, y_train = scale(train, True)
3 valid, X_valid, y_valid = scale(valid, False)
4 test, X_test, y_test = scale(test, False)
```

Code 6: Erstellung der Datensätze mit split und scale

Nach der Ausführung dieses Codes stehen die skalierten und aufbereiteten Datensätze „train“, „valid“ und „test“ zur Verfügung, die jeweils Merkmale und Labels enthalten. Die Arrays *X_train*, *X_valid* und *X_test* enthalten die skalierten Merkmale der entsprechenden Datensätze, während *y_train*, *y_valid* und *y_test* die entsprechenden Labels enthalten. Diese vorverarbeiteten Datensätze können im folgenden Kapiteln für das Training, die Validierung und die Testphase eines maschinellen Lernmodells verwendet werden.

7 Training Datensatz 1

7.1 K-Nearest Neighbors (KNN)

KNN steht für k-Nearest Neighbors und ist ein algorithmisches Modell des maschinellen Lernens. Es handelt sich um eine nicht-parametrische Methode zur Klassifikation und Regression von Daten. Bei KNN basiert die Vorhersage auf der Ähnlichkeit zu anderen Datenpunkten. Der Algorithmus kann zur Lösung von Klassifikations- und Regressionsproblemen verwendet werden. Jedoch ist die Wahl des optimalen k-Werts und der geeigneten Distanzmetrik (und weitere Parameter) entscheidend für die Genauigkeit der Klassifikation. [Guo+03]

7.1.1 Konzept hinter KNN

Um einen neuen Datenpunkt zu klassifizieren, werden die k nächsten Nachbarn in einem Trainingsdatensatz nach ihrem Abstand (abhängig von der Distanzmetrik) zum neuen Punkt ausgewählt. Die Klassifikation erfolgt dann, indem die Labels der k nächsten Nachbarn aufeinander abgestimmt werden. [Guo+03]

7.1.2 K-Optimierung

Da der Datensatz in verschiedene Unterdatensätze aufgeteilt wurde, können unterschiedliche Genauigkeiten ermittelt werden. Durch das Ausprobieren verschiedener Hyperparameter (unter anderem auch durch die Sklearn-Funktion „GridSearchCV“ , vgl. Kapitel 8.1) konnten folgende Hyperparameter ermittelt werden.

- Bestimmung der Abstands-Metriken (metric) = euclidean
- Bestimmung der Abstands-Gewichten (weights) = distance
- Alogrithmus (algorithm) = ball_tree

Um zu verdeutlichen, welche K-Werte am besten geeignet sind, wurde folgender Code (vgl. Code 7) zu Hilfe genommen. Ziel ist es, mit Hilfe einer Schleife über zehn K-Werte zu iterieren und damit ein KNN-Modell zu trainieren. In jeder Iteration wird die Anzahl der Nachbarn durch die Laufvariable bestimmt und immer auf dem Trainingsdatensatz trainiert.Zusätzlich werden für den Train-, Test- und Vaild-Datensatz die Genauigkeiten bestimmt.

```
k_range=range(1,10)
knn_r_acc = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k, algorithm='ball_tree',
    ↪  weights='distance', metric='euclidean', n_jobs=-1)
    knn.fit(X_train,y_train)

    train_score = knn.score(X_train,y_train)
    test_score = knn.score(X_test,y_test)
    vaild_score = knn.score(X_valid,y_valid)

    knn_r_acc.append((k, train_score,test_score, vaild_score))
df = pd.DataFrame(knn_r_acc, columns=['K','Train Score', 'Test Score','Vaild
↪  Score'])
print(df)
```

Code 7: KNN K-Optimierung

Ergebnis der KNN K-Optimierung:

```
'''
   K  Train Score  Test Score  Vaild Score
0  1          1.0    0.998721     0.998533
1  2          1.0    0.998721     0.998533
2  3          1.0    0.998374     0.998171
3  4          1.0    0.998141     0.997898
4  5          1.0    0.997834     0.997398
5  6          1.0    0.997611     0.997120
6  7          1.0    0.997279     0.996798
7  8          1.0    0.996976     0.996506
8  9          1.0    0.996778     0.996312
'''
```

Code 8: KNN K-Optimierung Ergebnis

Code 8 zeigt, dass die Genauigkeit mit den gleichen Daten, mit denen trainiert wurde, bei 100% liegt. Der Testdatensatz ist etwas besser als der Validierungsdatensatz, was auf die Zufallsfunktion in der Verteilung der Daten zurückzuführen ist. Da es in den meisten Fällen sinnvoll ist, k auf einen Wert größer als 1 zu setzen, um eine bessere Stabilität und Robustheit der Vorhersagen zu gewährleisten, wird K=2 als idealer Wert angenommen.

7.1.3 Training

Durch die KNN K-Optimierung (vgl. Unterabschnitt 7.1.2) konnte gezeigt werden, dass zwei der beste Paramter für k ist. Nachfolgend ist ein Klassifikationsreport mit dem KNN-Model, trainiert auf den Trainingsdatensatz und validiert mit dem Testdatensatz, dargestellt.

```
knn = KNeighborsClassifier(n_neighbors=2, algorithm='ball_tree',
↪  weights='distance', metric='euclidean', n_jobs=-1)
knn.fit(X_train,y_train)
y_pred = knn.predict(X_test)

print(classification_report(y_test, y_pred))
'''
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     94155
           1       1.00      1.00      1.00    107594

    accuracy                           1.00    201749
   macro avg       1.00      1.00      1.00    201749
weighted avg       1.00      1.00      1.00    201749
'''
```

Code 9: KNN Endergebnis

Code 9 zeigt eine gerundete Genauigkeit von 100%. Recall und Precision liegen ebenfalls bei einem gerundeten Wert von 100%.

Ein Export dieses Modells ist im Repository (vgl. Kapitel 4) unter *`/saved_models/knn_datensatz1.joblib`* beigefügt.

7.2 Decision Trees (dTrees)

Entscheidungsbäume (engl. Decicision Trees) sind überwachte Lernmethoden ohne feste Parameter, welche zur Klassifikation und Regression verwendet werden [Scia]. Sie sind hierarchisch strukturiert und stellen einen baumartigen Graph dar. Diese Struktur besteht aus Knoten, die einen Test auf ein Attribut darstellen, und Verzweigungen, die das Ergebnis des Tests darstellen, sowie Blattknoten, die eine Klassenbezeichnung darstellen.

Die Entscheidungen werden durch den gewählten Pfad vom Wurzelknoten aus zum Blattknoten gebildet. Angefangen vom Wurzelknoten werden die Attribute und ihre zugehörigen Werte identifiziert und anhand der Klassifikationsregeln entschieden, welche Verzweigungen gebildet werden. Sobald der Baum einmal erstellt ist, kann er neue unbekannte Daten vorhersagen, indem er, ausgehend vom Wurzelknoten, alle internen Knoten auf dem Pfad in Abhängigkeit von den Testbedingungen der Attribute an jedem Knoten besucht.[Sar+16]

7.2.1 Feature Importances

Das Hauptproblem bei der Konstruktion eines Entscheidungsbaumes ist die Wahl des Features, welches für die Teilung des Baumes verwendet werden soll. Dieser ist ausschlaggebend, wie aussagekräftig das Modell ist. Daher wurde zu Beginn die Feature Importances der einzelnen Attribute der Daten betrachtet. Dies führte mit den verwendeten Daten aus Datensatz 1 zu dem Ergebnis, dass für die Klassifizierung das Protokoll des jeweiligen Pakets, das aussagekräftigste Attribut mit knapp über 80% ist. Zu sehen ist die Verteilung der Attribute und ihrer dazugehörigen Aussagekraft in Prozent in Abbildung 7.1.

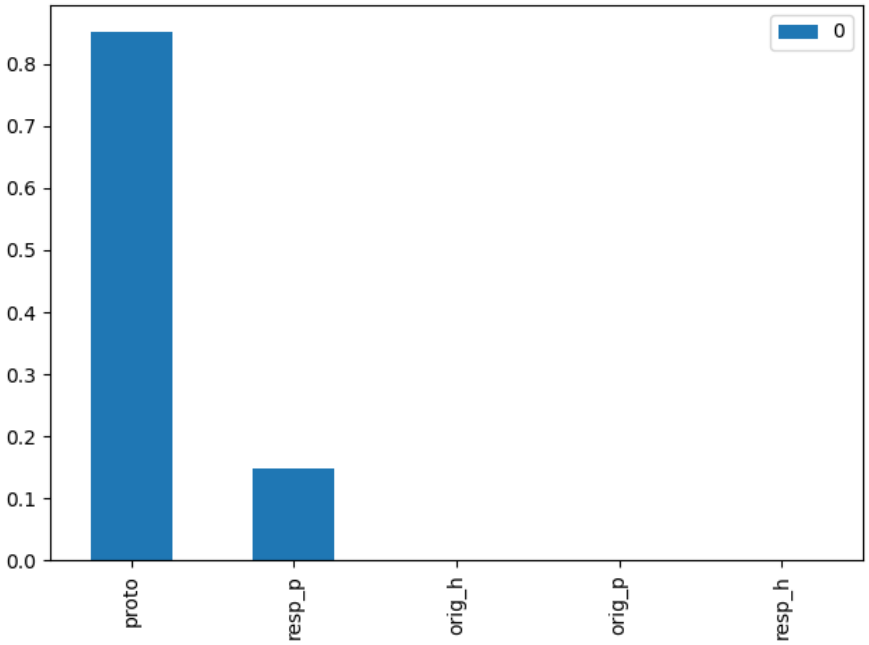

Abb. 7.1: Feature Importances für Datensatz 1

7.2.2 Training

Das Training wurde mit Trainingsdaten, die wir aus unserer implementierten Splitting-Funktion erhalten haben, durchgeführt. Wie Code 10 darstellt, war das Training mit einer Genauigkeit von 98% sehr erfolgreich.

```
              precision    recall  f1-score   support

      benign       1.00      0.95      0.97     94010
   malicious       0.96      1.00      0.98    107739

    accuracy                           0.98    201749
   macro avg       0.98      0.97      0.98    201749
weighted avg       0.98      0.98      0.98    201749
```

Code 10: Classification Report Decision Tree Datensatz 1

Der Graph in Abbildung 7.2 zeigt deutlich, wie der Gini-Wert mit jeder tieferen Ebene abnimmt und am Ende teilweise sogar bei 0 liegt. Der Gini-Wert gibt an, wie gut ein Entscheidungsbaum einen Datensatz nach bestimmten Merkmalen aufteilt. Er liegt zwischen 0 und 1, wobei 0 bedeutet, dass alle Datenpunkte in einer Gruppe gleich sind (hohe Reinheit) und 1 bedeutet, dass die Daten gleichmäßig auf alle möglichen Ausgaben verteilt sind (geringe Reinheit). Dies kann so interpretiert werden, dass der resultierende Graph effektiv und präzise Entscheidungen ermöglicht.

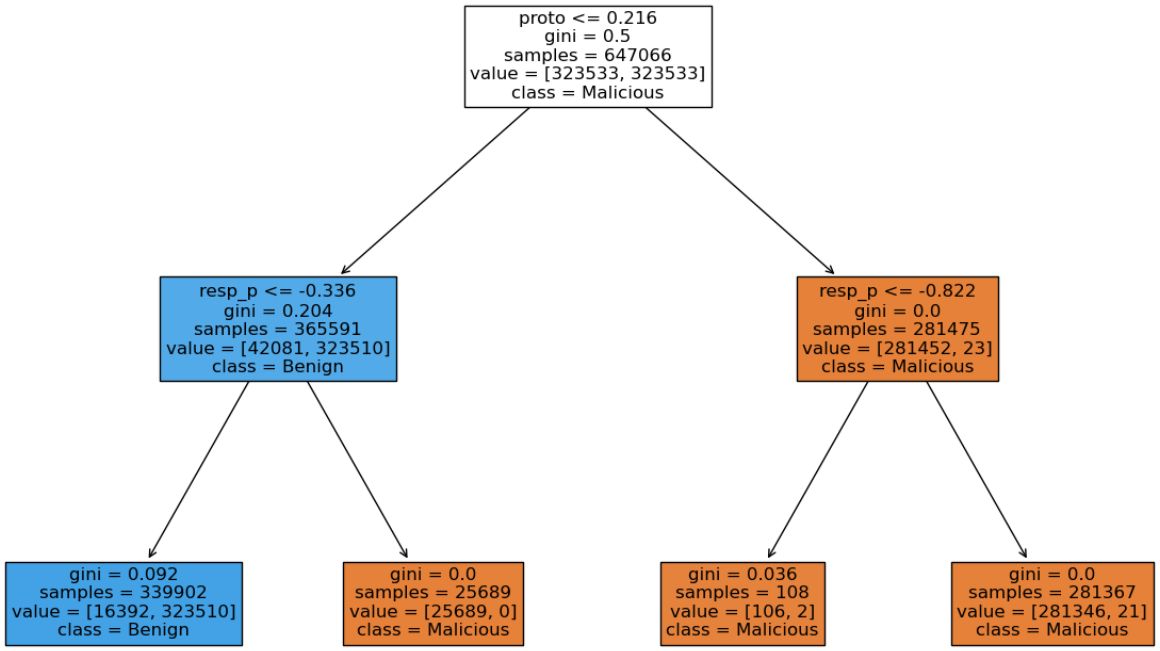

Abb. 7.2: Decision Tree für Datensatz 1

Ein Export dieses Modells ist im Repository (vgl. Kapitel 4) unter *`/saved_models/dtree_datensatz1.joblib`* beigefügt.

7.3 Neural Network (ANN)

ANN steht für Artificial Neural Network und löst Probleme des maschinellen Lernens mit Hilfe von Algorithmen, die von der Funktionsweise des menschlichen Gehirns inspiriert sind.

7.3.1 Konzept hinter ANN

Ein neuronales Netz besteht aus einer Reihe miteinander verbundener künstlicher Neuronen, die Informationen verarbeiten und weiterleiten. Diese Neuronen sind in Schichten organisiert: eine Eingabeschicht, eine oder mehrere verborgene Schichten und eine Ausgangsschicht. Jedes Neuron berechnet einen gewichteten Eingangswert, wendet eine Aktivierungsfunktion an und gibt das Ergebnis an die nächsten Neuronen weiter. Durch Anpassung der Gewichte und Schwellenwerte während des Trainings lernt das Netz, Muster und Beziehungen in den Eingabedaten zu erkennen.[Woo21]
Das neuronale Netz und die darin enthaltenen Schichten, die wir für unser Training verwendet haben, sind in Abbildung 7.3 zu sehen.

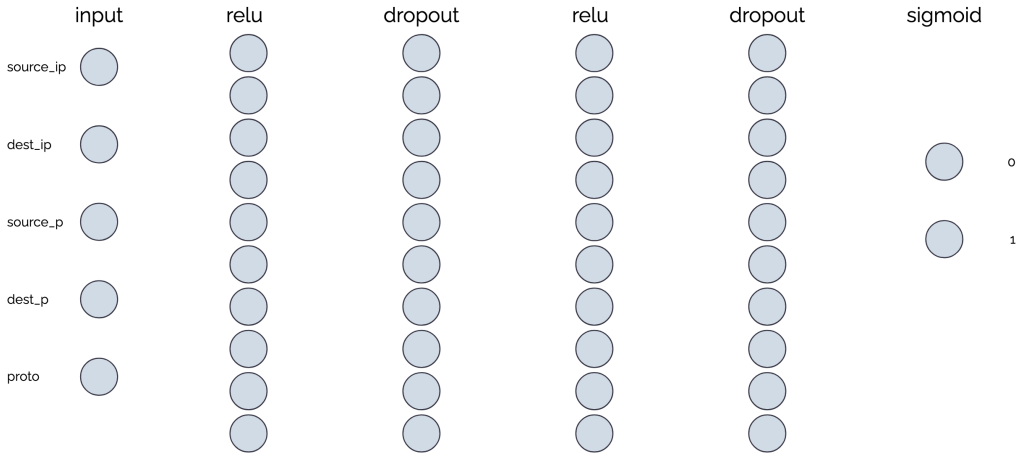

Abb. 7.3: Neuronales Netz

Auf der linken Seite sind die Eingangsparameter zu sehen. Als Hidden-Layers dienen zwei Layer mit *relu*-Funktion und zwei *dropout* Layer. Als Output-Layer dient ein Layer mit *sigmoid*-Funktion, der eine binäre Entscheidung liefert.

7.3.2 Training

Für Datensatz 1 wird nur ein Modell mit vorgegebenen Parametern trainiert, eine detaillierte Analyse der einzelnen Parameter wurde mit dem fusionierten Datensatz in Abschnitt 8.3 durchgeführt.
Die Auswahl der Parameter ist in Tabelle 7.1 zu sehen.

Nodes	Dropout Wahrscheinlichkeit	Lernrate	Batch Größe
64	0	0	64

Tab. 7.1: Parameter für Modell mit Datensatz 1

Die Ergebnisse des Modells sind in Abbildung 7.4 dargestellt. Das Modell wurde mit dem ersten Datensatz trainiert und getestet und ergab einen *Loss* von 69% und eine *Genauigkeit* von 99,86%.

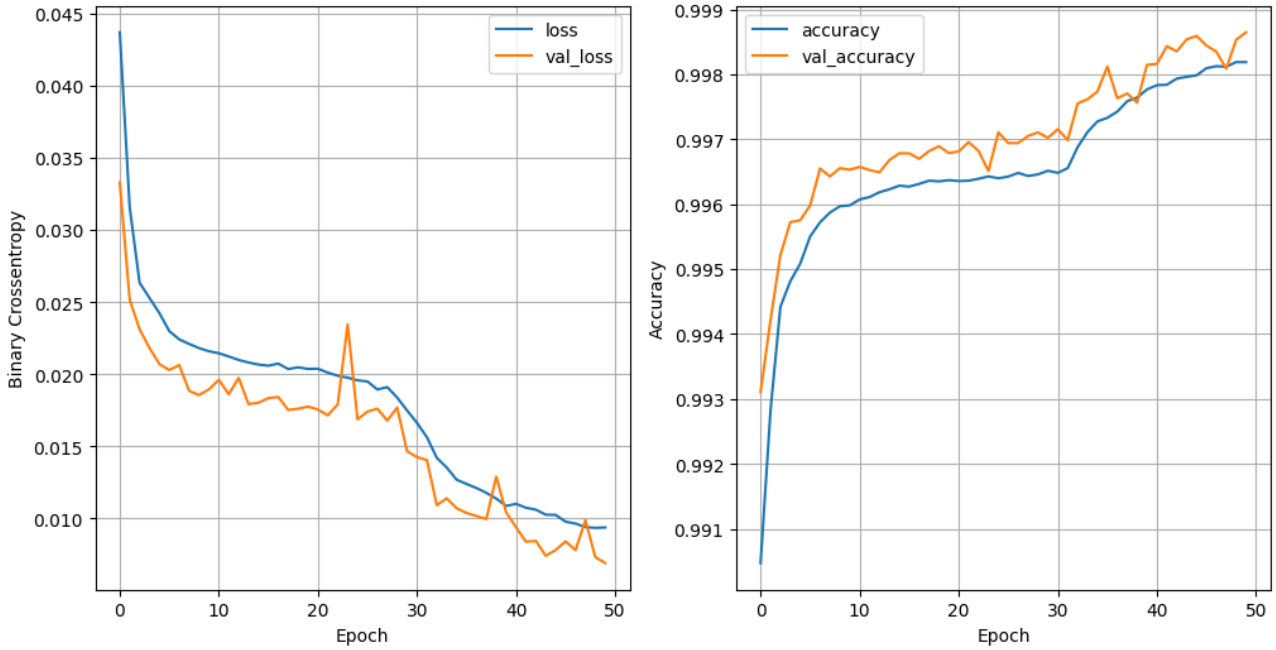

Abb. 7.4: Modell mit Datensatz 1

Ein Export dieses Modells ist im Repository (vgl. Kapitel 4) unter *`/saved_models/dataset1_keras_model/`* beigefügt.

8 Training Datensatz merged

8.1 K-Nearest Neighbors (KNN)

Die folgenden Unterkapitel befassen sich mit dem Training eines KNN-Modells auf dem zusammengesetzten Datensatz. Dazu wurde die Funktion GridSearchCV aus der Scikit-Learn-Bibliothek verwendet. Diese führt eine systematische Suche über einem vorgegebenen Parametergitterraum durch, um die besten Hyperparameter für ein Lernmodell zu finden. Sie trainiert und evaluiert das Modell für jede mögliche Kombination der angegebenen Hyperparameter und gibt das Modell mit den besten gefundenen Hyperparametern zurück. Zusätzlich verwendet GridSearchCV eine Kreuzvalidierung. Dabei wird der vorliegende Datensatz in mehrere Teile (Folds) zerlegt. Das Modell wird wiederholt auf verschiedenen Teilmengen trainiert und auf den verbleibenden Teilmengen getestet. Die Kreuzvalidierung ermöglicht eine robustere Abschätzung der Modellleistung, da der gesamte Datensatz in Trainings- und Testsets aufgeteilt wird und das Modell wiederholt auf verschiedenen Teilmengen trainiert und getestet wird. Aus diesem Grund wurde die nachfolgende Hyperparameteroptimierung auf den Originaldaten durchgeführt (siehe Code 11 Zeilen 1 und 2).[Scib]

8.1.1 Hyperparameteroptimierung

```
1  X = SOURCEDF.drop('label',axis=1)
2  y = SOURCEDF.label
3
4  from sklearn.model_selection import GridSearchCV
5
6  grid_params = {
7  'n_neighbors': [1,2,3,4,5,6,7],
8  'weights': ['uniform', 'distance'],
9  'metric': ['euclidean', 'manhattan'],
10 }
11
12 gs = GridSearchCV(
13     KNeighborsClassifier(),
14     grid_params,
15     verbose = 10,
16     cv = 3,      # Determines the cross-validation splitting strategy
17                  # integer, to specify the number of folds
18     n_jobs = -1 # use all processors to perform the model
19 )
20
21 gs_results = gs.fit(X, y)
```

Code 11: KNN Hyperparameteroptimierung

Der GridSearchCV-Algorithmus führt eine Hyperparameter-Optimierung durch, bei der 28 verschiedene Kombinationen von Hyperparametern getestet werden. Für jede dieser Kombinationen wird der

Trainingsdatensatz in 3 Teilmengen (Folds) aufgeteilt. Das Modell wird dann insgesamt 84 Mal trainiert und getestet, da es für jede Kombination 3-fach kreuz-validiert wird.
Ergebnis der KNN Hyperparameteroptimierung:

```
gs_results.best_score_
0.8986799842095562

gs_results.best_params_
{'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'distance'}
```

Code 12: KNN Hyperparameteroptimierung Ergebnisse

Abbildung 8.1 zeigt, die Genauigkeit verschiedener Parameter des GridSearches. Im linken Diagramm ist zu erkennen, dass die Metric „manhatten“ eine bessere Genauigkeit erzielt gegenüber „euclidean“ . Rechts wird die Genauigkeit zwischen Gewichten dargestellt, hier führt das Gewicht „distance“ zu besseren Ergebnissen als „uniform“ . In Grafik in der Mitte ist zu entnehmen, dass bei k=2 die besten Genauigkeiten erzielt werden können. Diese Grafik deckt sich mit der Ausgabe von gs_results.best_params.

Weitere Hyperparameteroptimierung

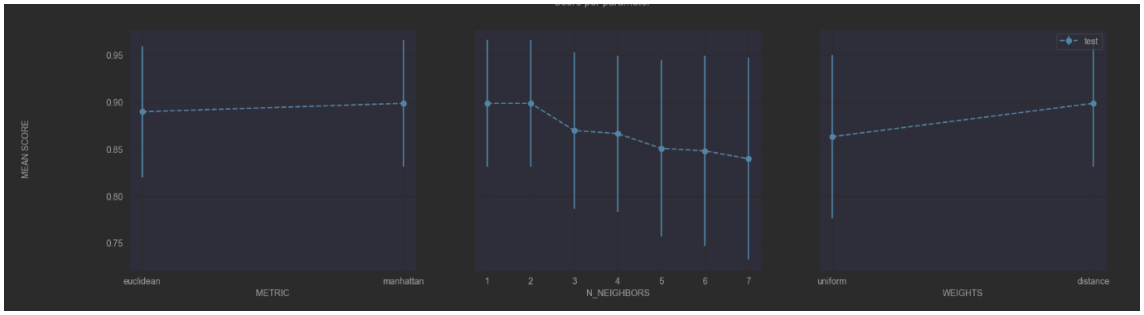

Abb. 8.1: KNN GridSearchCV Verteilung, Quelle: eigene Darstellung

8.1.2 Weitere Parametertests

Nun wurden weitere GridSearchCVs auf anderen Parametersätzen erstellt, um weitere Scores zu erhalten. Beispielsweise wurde auf folgendem Parameterset trainiert (vgl. Code 13). Es konnte jedoch keine weitere Verbesserung der Daten erzielt werden.

```
'metric': ['l1', 'l2']
Ergebnis:
{'metric': 'l1', 'n_neighbors': 2, 'weights': 'distance'}
0.8986799842095562

'metric': ['euclidean', 'manhattan'],
'algorithm': ['auto', 'ball_tree']
Ergebnis:
{'algorithm': 'auto', 'metric': 'manhattan', 'n_neighbors': 2, 'weights':
↪  'distance'}
0.8986799842095562
```

Code 13: KNN Hyperparameteroptimierung Parameterset

8.1.3 Training

Folgende Ergebnisse konnten durch die in Unterabschnitt 8.1.1 errechneten Parameter erreicht werden:

```
knn = KNeighborsClassifier(n_neighbors=2, metric='manhattan',weights='distance',
↪  n_jobs=-1)
knn.fit(X_train,y_train)
y_pred = knn.predict(X_test)

knn.score(X_test,y_test)
0.897141417305034

knn.score(X_valid,y_valid)
0.9162863534675615

print(classification_report(y_test, y_pred))
'''           precision    recall  f1-score   support

           0       0.92      0.92      0.92    212190
           1       0.92      0.92      0.92    212460

    accuracy                           0.92    424650
   macro avg       0.92      0.92      0.92    424650
weighted avg       0.92      0.92      0.92    424650'''
```

Code 14: KNN Endergebnis

Ein Export dieses Modells ist im Repository (vgl. Kapitel 4) unter *`/saved_models/knn_datensatz2.joblib`* beigefügt.

8.2 Decision Trees (dTrees)

8.2.1 Hyperparameteroptimierung

Da die Ergebnisse mit den Default-Parametern auf dem Merged-Datensatz deutlich schlechtere Ergebnisse mit teilweise unter 50% erzielten, wurde auch hier die Funktion GridSearchCV aus der Scikit-Learn-Bibliothek verwendet, um weitestgehend automatisch die besten Parameter für den Datenansatz zu finden. Dabei wurde die in Code 15 gezeigte Funktion verwendet.

```
def dtree_grid_search(X,y,nfolds):
    #create a dictionary of all values we want to test
    param_grid = {'ccp_alpha': np.arrange(0,0.8,0.01, 'max_depth':
    ↪  [1,2,3,4],'random_state': [1,2,3,4,5], 'criterion': ['entropy', 'gini']}
    # decision tree model
    dtree_model=DecisionTreeClassifier()
    #use gridsearch to test all values
    dtree_gscv = GridSearchCV(dtree_model, param_grid, cv=nfolds, n_jobs=30)
    #fit model to data
    dtree_gscv.fit(X, y)
    # print bets score achieved
    print(dtree_gscv.best_score_)
    return dtree_gscv.best_params_
```

Code 15: Hyperparameteroptimierung mit GridSearchCV

Nach einigen zunächst zufälligen und dann systematischen Durchläufen wurde die Arbeit mit folgenden Hyperparametern (vgl. Code 16) fortgesetzt, da mit diesen die meisten guten Ergebnisse erzielt wurden. Mit der bereits in Kapitel 6 eingeführten zufallsbasierten Split-Funktion war es uns leider aufgrund von, unter anderem: Overfitting, nicht möglich, konsistente Ergebnisse auf dem Datensatz zu erzielen. Das beste Ergebnis wurde mit dem Modell erzielt, welches im beiliegenden Export zu finden ist. Der Auszug aus dem Klassifikationsbericht ist in Code 17 zu finden.

```
DecisionTreeClassifier(ccp_alpha=0.1, max_depth=4, random_state=1)
```

Code 16: Decison Tree Features

```
              precision    recall  f1-score   support

      benign       0.99      0.95      0.97    212241
   malicious       0.95      0.99      0.97    212409

    accuracy                           0.97    424650
   macro avg       0.97      0.97      0.97    424650
weighted avg       0.97      0.97      0.97    424650
```

Code 17: Classification Report Decision Tree Datensatz 2

Ein Export dieses Modells ist im Repository (vgl. Kapitel 4) unter *`/saved_models/dtree_datensatz2.joblib`* beigefügt.

8.3 Neural Network (ANN)

Code 18 und Code 19 zeigen die Implementierung in Python. Code 19 iteriert über mehrere Parameter, um die beste Kombination aller Parameter zu ermitteln. Ihre Genauigkeit wird dann verglichen und das Modell mit der besten Leistung ausgewählt. Für alle Trainingsläufe wurde die Anzahl der Epochen auf 50 festgelegt.

```
def nn_train(X_train: pd.DataFrame, y_train: pd.DataFrame, X_valid: pd.DataFrame,
↪  y_valid: pd.DataFrame, epochs: int, nodes: int, dropout_prob: float, lr: int,
↪  batch_size: int):
    nn_model = tf.keras.Sequential([
        tf.keras.layers.Dense(nodes, activation='relu', input_shape=(5,)),
        tf.keras.layers.Dropout(dropout_prob),
        tf.keras.layers.Dense(nodes, activation='relu'),
        tf.keras.layers.Dropout(dropout_prob),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])

    nn_model.compile(optimizer=tf.keras.optimizers.Adam(lr),
    ↪  loss='binary_crossentropy', metrics=['accuracy'])
    history = nn_model.fit(X_train, y_train, epochs=epochs,
    ↪  batch_size=batch_size, validation_data=(X_valid, y_valid), verbose=0)
    return nn_model, history
```

Code 18: Training Funktion

```
epochs = 50
for nodes in (16, 32, 64):
    for dropout_prob in [0, 0.2]:
        for lr in [0.1, 0.005, 0.001]:
            for batch_size in [32, 64, 128]:
                model, history = nn_train(X_train=X_train, y_train=y_train,
                ↪  X_valid=X_valid, y_valid=y_valid, epochs=epochs, nodes=nodes,
                ↪  dropout_prob=dropout_prob, lr=lr, batch_size=batch_size)
                val_loss = model.evaluate(X_test, y_test)[0]
```

Code 19: Parameter Search

In Tabelle 8.1 sind die Genauigkeiten aus den verschiedenen Modellen aufgelistet. Das Training aller Modelle dauerte auf einer 8-Kern-CPU ohne Grafikbeschleunigung ca. 29 Stunden.

Nodes	Dropout Wahrscheinlichkeit	Lernrate	Batch Größe	Genauigkeit [%]
(16; 32; 64)	0	0.1	32	(73,90; 75,46; 74,21)
(16; 32; 64)	0	0.1	64	(50,04; 49,96; 51,15)
(16; 32; 64)	0	0.1	128	(85,97; 82,77; 74,27)
(16; 32; 64)	0	0.005	32	(87,57; 87,63; 87,61)
(16; 32; 64)	0	0.005	64	(87,69; 87,68; 87,59)
(16; 32; 64)	0	0.005	128	(87,68; 87,69; 87,75)
(16; 32; 64)	0	0.001	32	(87,63; 87,53; 87,76)
(16; 32; 64)	0	0.001	64	(87,65; 87,78; 87,79)
(16; 32; 64)	0	0.001	128	(87,77; 87,65; 87,78)
(16; 32; 64)	0.2	0.1	32	(73,62; 73,57; 73,88)
(16; 32; 64)	0.2	0.1	64	(81,59; 73,57; 74,94)
(16; 32; 64)	0.2	0.1	128	(82,85; 74,93; 74,27)
(16; 32; 64)	0.2	0.005	32	(87,40; 87,35; 87,22)
(16; 32; 64)	0.2	0.005	64	(87,03; 87,40; 87,59)
(16; 32; 64)	0.2	0.005	128	(87,40; 87,49; 87,39)
(16; 32; 64)	0.2	0.001	32	(87,10; 87,35; 87,38)
(16; 32; 64)	0.2	0.001	64	(87,08; 87,37; 87,59)
(16; 32; 64)	0.2	0.001	128	(87,20; 87,39; 87,52)

Tab. 8.1: Trainierte Modelle mit den parametrisierten Details und den zugehörigen Genauigkeiten in Prozent

Zu erkennen ist, dass die Kombination aus Batch-Größe $= 64 \wedge$ Lernrate $= 0\,1$ eine besonders hohen *loss*-Wert und damit eine besonders geringe Genauigkeit hervorruft. Abbildung 8.2 zeigt die Verläufe der beiden Parameter, über den Trainingszeitraum von 50 Epochen.

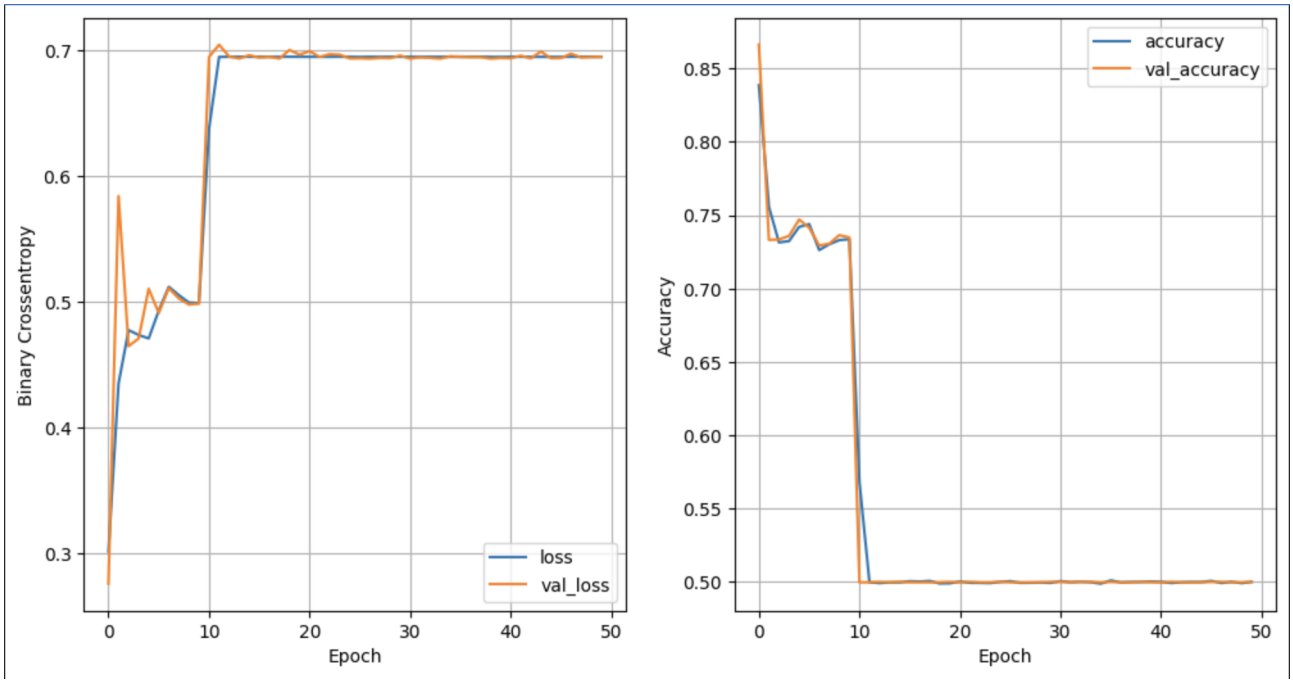

Abb. 8.2: Schlechtestes ANN-Modell

Das beste Modell hatte die Parameter Batch-Größe = 64 ∧ Lernrate = 0 001.

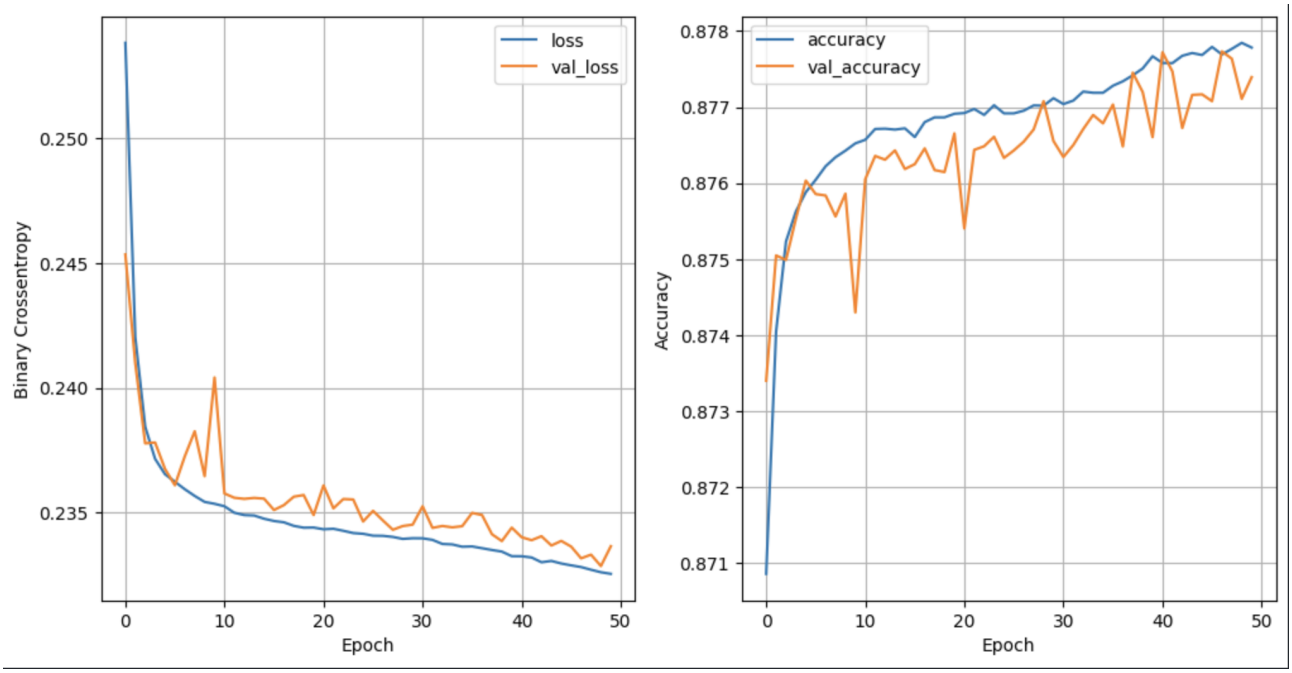

Abb. 8.3: Bestes ANN Modell

Eine Frage, die mit Hilfe der Tabelle und Abbildung 8.4 beantwortet werden kann, ist, ob die Dropout-Layer tatsächlich einen Effekt haben. Die Grafik zeigt, dass sich die Wahrscheinlichkeiten für die gewählten Parameter nicht signifikant unterscheiden, was auf einen geringen Einfluss hindeutet.

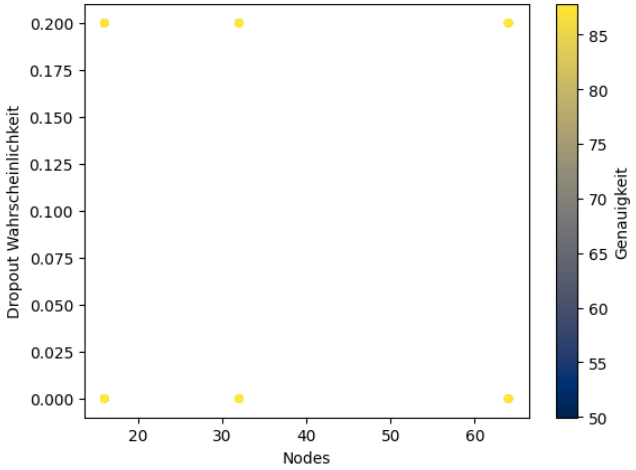

Abb. 8.4: Dropout Probability der verschiedenen Modelle

Ein Export dieses Modells ist im Repository (vgl. Kapitel 4) unter *`/saved_models/dataset_merged_keras_model/`* beigefügt.

9 Ergebnis und Fazit

Abschließend werden die Ergebnisse der Modelle zusammengefasst und miteinander verglichen. Außerdem wird eine Selbstevaluierung unter Einbeziehung der Projektergebnisse durchgeführt.

9.1 Vergleich der Modelle

Die für das Training verwendeten Modelle haben durchweg gute bis sehr gute Ergebnisse erzielt. Da die Auswahl der Modelle unter anderem nach diesem Kriterium erfolgte, ist dies ein zu erwartender Einflussfaktor.
Aus Tabelle 9.1 lassen sich die verschiedenen Genauigkeiten vor und nach dem Zusammenführen der Daten erkennen. Deutlich zu erkennen ist die nahezu konstant bei 100% liegende Genauigkeit bei allen Modellen bei Datensatz (DS) 1. Dies ist vermutlich auf die Homogenität der Daten zurückzuführen, da die Daten im gleichen Umfeld mit gleichen Begebenheiten aufgenommen wurden und zusätzlich nur anteilig verwendet wurden. Aus diesem Grund wurde die Entscheidung getroffen, das Training für die Modelle schwieriger zu gestalten und mehrere Datensätze miteinander zu verbinden. Das Resultat fällt dabei sehr anders aus, die Genauigkeit einiger Modelle sinkt deutlich, was auf eine heterogene Verteilung hindeutet. **Wichtig zu beachten:** Hier wurde bei den ersten Versuchen, mit Standardparametern deutlich geringere Genauigkeiten erreicht, diese konnten mit Hyperparameteroptimierung jedoch um ein Vielfaches gesteigert werden.
Wir sind uns ebenfalls bewusst, dass die trainierten Modelle nur mit unseren speziell aufbereiteten Daten arbeiten können. Eine Analyse aller Parameter des aufgezeichneten Netzwerkverkehrs oder gar eine Live-Auswertung an laufenden Verbindungen, wie sie in „state-of-the-art“ Intrusion Detection Systemen implementiert ist, hätte den Rahmen dieser Arbeit gesprengt.

Modell	DS 1 [%]	DS merged [%]
kNN	100	92
dTree	98	97
ANN	100	88

Tab. 9.1: Genauigkeit der verschiedenen Modelle in Abhängigkeit vom Datensatz (DS)

9.2 Fazit

Zusammenfassend, war das Training der Modelle ein sehr lehrreiches und interessantes Projekt. Es hat uns als Team sehr viel Spaß gemacht. Es konnten verschiedene Ansätze getestet und angewendet werden. Am Anfang waren die Ziele etwas zu hoch gesteckt, z.B. konnten die Versuche nicht wie geplant mit „echten“ Netzwerkdaten aus unseren Heimnetzwerken durchgeführt werden. Auch wurde anfangs sehr viel Zeit in die Normalisierung und Darstellung der Daten investiert, anstatt sich wirklich mit den notwendigen Eingabewerten der Modelle zu beschäftigen und nach diesen Kriterien zu normalisieren. Dies konnte jedoch zum Ende hin aufgeholt und durch die Beschäftigung mit verschiedenen Modellen auf den Daten ausgeglichen werden.
Alles in allem ein gelungenes Projekt, das uns allen die Methoden und Mechanismen des maschinellen Lernens näher gebracht hat.

Literatur

[Scia] *1.10. Decision Trees — scikit-learn.org*. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. [Accessed 14-Jul-2023].

[GPE20] Sebastian Garcia, Agustin Parmisano und Maria Jose Erquiaga. *IoT-23: A labeled dataset with malicious and benign IoT network traffic*. en. 2020. DOI: 10.5281/ZENODO.4743746. URL: https://zenodo.org/record/4743746.

[Guo+03] Gongde Guo u. a. „KNN Model-Based Approach in Classification“. In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. Springer Berlin Heidelberg, 2003, S. 986–996. DOI: 10.1007/978-3-540-39964-3_62. URL: https://doi.org/10.1007/978-3-540-39964-3_62.

[Sar+16] Kajal Saraswat u. a. „Decision Tree Based Algorithm for Intrusion Detection“. In: *International Journal of Advanced Networking and Applications* 07 (Jan. 2016), S. 2828–2834.

[Scib] *sklearn.model_selection.GridSearchCV — scikit-learn.org*. https://scikit-learn.org/stable/modules/tree.html. [Accessed 24-Jul-2023].

[Woo21] Chris Woodford. *Neural networks*. Accessed on 2023-08-03. 2021. URL: https://www.explainthatstuff.com/introduction-to-neural-networks.html (besucht am 03. 08. 2023).