ANLERNEN VON NETZWERK-TRAFFIC

Learning & Soft Computing

Nicola Jäger, Marc Schmidt, Jonathan Ebinger

Agenda

kNN
dTrees
aNN

Datensätze	Konvertierung	Training	Vergleich
Recherche verschiedener Datensätze im Internet	Vereinheitlichung, Reduzierung und Aufteilung in Test-, Train- und Validierungsdaten	Trainieren der Modelle	Vergleich der Datensätze

Arbeitsweise

colab Pro

- Datenzugriff über Google Drive
- Zusammenarbeiten begrenzt möglich
- Pro Ressourcen für High-Performance RAM
- GPU Beschleunigung

kaggle

- Probleme mit Internetzugriff
- Sehr langsame Ausführungszeiten
- Datenzugriff nicht weiter ausprobiert

Jupyter Notebook

- **Docker Hub: *jupyter/scipy-notebook***
- **Daten befinden sich in einem persistenten Datenträger**
- **Zusammenarbeiten besser als bei colab, aber dennoch nicht optimal**
- **Ausführungszeiten fast identisch mit Google colab**
- **24/7 Uptime**
- **Keine Kosten**

DataSpell

- Dateien lokal auf OS
- Schnellere Ausführungszeiten
- Kein Zusammenarbeiten möglich

Python

- Dateien lokal auf OS
- Sehr schnelle Ausführungszeiten
- Kein Zusammenarbeiten möglich

Datensatz: **Aposemat IoT-23**

- Erstellung im Rahmen des Avast AIC-Labors mit finanzieller Unterstützung von Avast
- Im Januar 2020 veröffentlicht, umfasst Aufnahmen aus den Jahren 2018 bis 2019
- Beinhaltet echten Netzwerkverkehr von IoT-Geräten
  - Infizierter / bösartiger IoT-Netzwerk-Traffic
    - Mirai Botnet
    - Okiru Botnet
    - PortScan
    - C&C-HeartBeat
    - DDoS
  - unbedenklicher/normaler IoT-Netzwerk-Traffic

Vorstellung der IDS-Datensätze

- Name der Datensätze: CTU-IoT-Malware-*

Datensatz	**Normal** Traffic Flows	**Malicious** Traffic Flows
Capture-1-1 (Hide and Seek) [1]	469.275	539.465
Capture-7-1 (Linux.Mirai) [2]	75.955	11.378.759
Capture-35-1 (Mirai) [3]	8.262.389	2.185.398

Malicious Flows bestehen teilweise aus weiteren Kategorien, wie:

- C&C
- C&C-HeartBeat
- C&C-FileDownload
- DDoS
- PartOfAHorizontalPortScan

Raw .log files

ts	uid	orig_h	orig_p	resp_h	resp_p	proto	service	duration	orig_bytes
1.525880e+09	CUmrqr4svHuSXJy5z7	192.168.100.103	51524	65.127.233.163	23	tcp	NaN	2.999051	0

resp_bytes	conn_state	local_orig	local_resp	missed_bytes	history	orig_pkts
0	S0	NaN	NaN	0	S	3

orig_ip_bytes	resp_pkts	resp_ip_bytes	label	detailed-label
180	0	0	Malicious	PartOfAHorizontalPortScan

.log → .csv

orig_h	orig_p	resp_h	resp_p	proto
192.168.100.103	51524	65.127.233.163	23	Y

conn_state	missed_bytes	history	orig_pkts
S0	0	X	3

orig_ip_bytes	resp_pkts	resp_ip_bytes	label
180	0	0	Malicious

Encoding von:

- IP-Adresse
- History
- Protocol
- Label

Zusammenführen der Dateien

1. Datensatz 1 - Datei 1
   - Verteilung: 500.000 : 400.000

2. Datensatz merged - Datei 1 + 7 + 37
   - Nach Konvertierung, Verteilung: 1.000.000 : 1.000.000

Split der Daten in Train/Test/Valid

- Aufspalten in Train **60%**, Test **20%** und Valid **20%** Datensätze

- Transformation der Daten durch **Standardisierung**

- Berechnung von **Mittelwert und Standardabweichung,** um Merkmale oder Variablen in einem Datensatz auf vergleichbare Skalen zu bringen

- Zufälliges **Oversampling** der Train Datensätze (Gleichverteilen)

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

def scale(df: pd.DataFrame, oversample = False):
    X = df.drop(['label'], axis=1)
    y = df['label'].values

    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    if oversample:
        ros = RandomOverSampler()
        X, y = ros.fit_resample(X, y)

    data = np.hstack((X, np.reshape(y, (-1, 1))))
    return data, X, y

train, valid, test = split(SOURCEDF)
train, X_train, y_train = scale(train, True)
valid, X_valid, y_valid = scale(valid, False)
test, X_test, y_test = scale(test, False)
```

Sneak-Peek in die Daten

- Daten wurden **standardisiert**

- 60% - _train
- 20% - _test
- 20% - _vaild

Die Datensätze werden zufällig erstellt
-> Genauigkeiten, können sich nach Datensatz unterscheiden

```
pd.DataFrame(X_train)
"""


        0           1           2           3           4
0 0.092973   -0.070335    1.792180    1.711901    1.178841
1 0.092973   -0.470168   -1.720678   -0.819821   -0.740578
2 0.092973   -1.093452    0.839101   -0.702064   -0.740578
3 0.092973    1.147476    1.240883   -0.333229   -0.740578
4 0.092973   -0.070335   -1.031813    1.306920    1.178841

"""
```

```
pd.Series(y_train).head()
"""
0     0
1     1
2     1
3     1
4     0
Name: label, dtype: int64
"""
```

Datensatz 1

kNN - K-Optimierung

Die idealen Parameter wurden im Vorfeld durch ein Ausprobieren ermittelt, später durch GridSearchCV

```
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))

    print(f"finished {k}")

df = pd.DataFrame(knn_r_acc, columns=['K','Train Score', 'Test Score','Vaild Score'])
print(df)
```

Laufzeit: 15 min

kNN - K-Optimierung

k = 2

- Genauigkeit von 99%
- **Achtung: Werte sind gerundet**

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

```
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
'''
```

Laufzeit: 2 min

Nicola

https://10015.io/tools/code-to-image-converter

python, therme: AtomOne Light, Settings Line Number on, Rest default

no background

compact

dTrees - Hyperparameter Tuning

```python
def dtree_grid_search(X,y,nfolds):
    #create a dictionary of all values we want to test
    param_grid = { 'ccp_alpha': np.arange(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_
```

Laufzeit: ca. 5 min (auf 48 Kernen)

```
0.9935989770664833
{'ccp_alpha': 0.0, 'criterion': 'gini', 'max_depth': 4, 'random_state': 1}
```

dTrees - Hyperparameter Tuning

```python
# Create Decision Tree classifer object
dtree = DecisionTreeClassifier(max_depth=4, random_state=1, criterion= "gini")

# Train Decision Tree Classifer
dtree = dtree.fit(X_train,y_train)

dtree.get_params()
```

Laufzeit: 3 sek

```python
{'ccp_alpha': 0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': 4,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'random_state': 1,
 'splitter': 'best'}
```

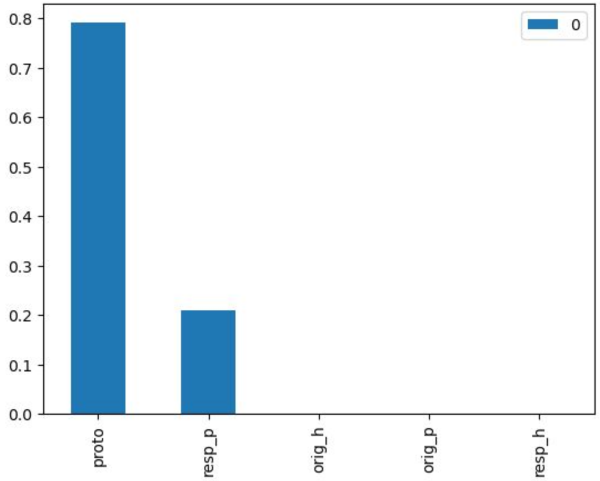
dTrees - Classification report

```python
# Predict the response for validation dataset
y_pred = dtree.predict(X_valid)

# Create classification report
classification_report(y_valid, y_pred, target_names=['benign', 'malicious'])
```

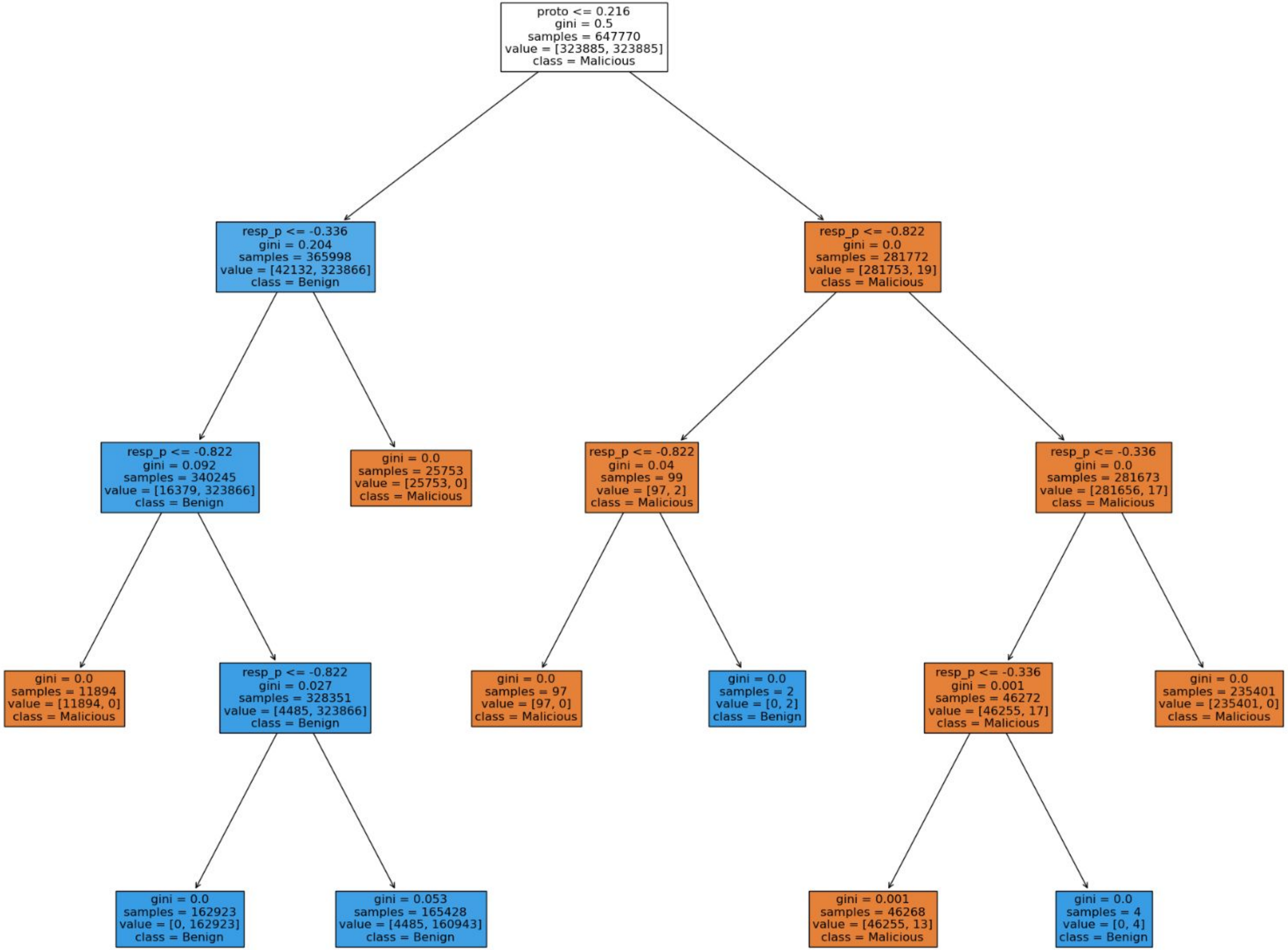
```
              precision    recall  f1-score   support

      benign       1.00      0.99      0.99     94141
   malicious       0.99      1.00      0.99    107608

    accuracy                           0.99    201749
   macro avg       0.99      0.99      0.99    201749
weighted avg       0.99      0.99      0.99    201749
```

dTrees - Feature importances

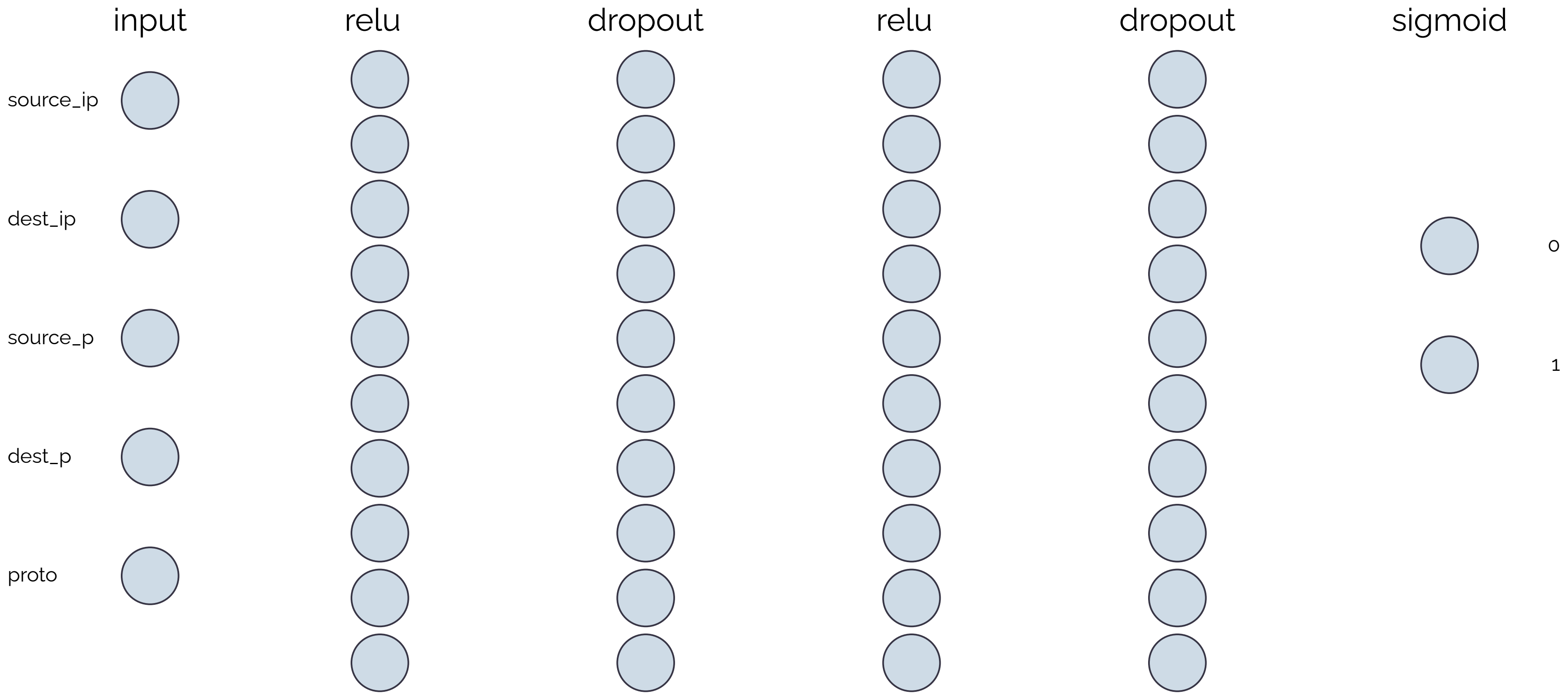
```
df_ds1_drop = df_ds1.drop(['label'], axis=1)
feature_names = df_ds1_drop.columns
feature_importance = pd.DataFrame(dtree.feature_importances_, index = feature_names).sort_values(0, ascending=False)

feature_importance.head(10).plot(kind='bar')
```



dTrees


proto <= 0.216
gini = 0.5
samples = 647770
value = [323885, 323885]
class = Malicious
resp_p <= -0.336
gini = 0.204
samples = 365998
value = [42132, 323866]
class = Benign
resp_p <= -0.822
gini = 0.0
samples = 281772
value = [281753, 19]
class = Malicious
resp_p <= -0.822
gini = 0.092
samples = 340245
value = [16379, 323866]
class = Benign
gini = 0.0
samples = 25753
value = [25753, 0]
class = Malicious
resp_p <= -0.822
gini = 0.04
samples = 99
value = [97, 2]
class = Malicious
resp_p <= -0.336
gini = 0.0
samples = 281673
value = [281656, 17]
class = Malicious
gini = 0.0
samples = 11894
value = [11894, 0]
class = Malicious
resp_p <= -0.822
gini = 0.027
samples = 328351
value = [4485, 323866]
class = Benign
gini = 0.0
samples = 97
value = [97, 0]
class = Malicious
gini = 0.0
samples = 2
value = [0, 2]
class = Benign
resp_p <= -0.336
gini = 0.001
samples = 46272
value = [46255, 17]
class = Malicious
gini = 0.0
samples = 235401
value = [235401, 0]
class = Malicious
gini = 0.0
samples = 162923
value = [0, 162923]
class = Benign
gini = 0.053
samples = 165428
value = [4485, 160943]
class = Benign
gini = 0.001
samples = 46268
value = [46255, 13]
class = Malicious
gini = 0.0
samples = 4
value = [0, 4]
class = Benign

aNN


input
relu
dropout
relu
dropout
sigmoid
source_ip
dest_ip
source_p
dest_p
proto
0
1

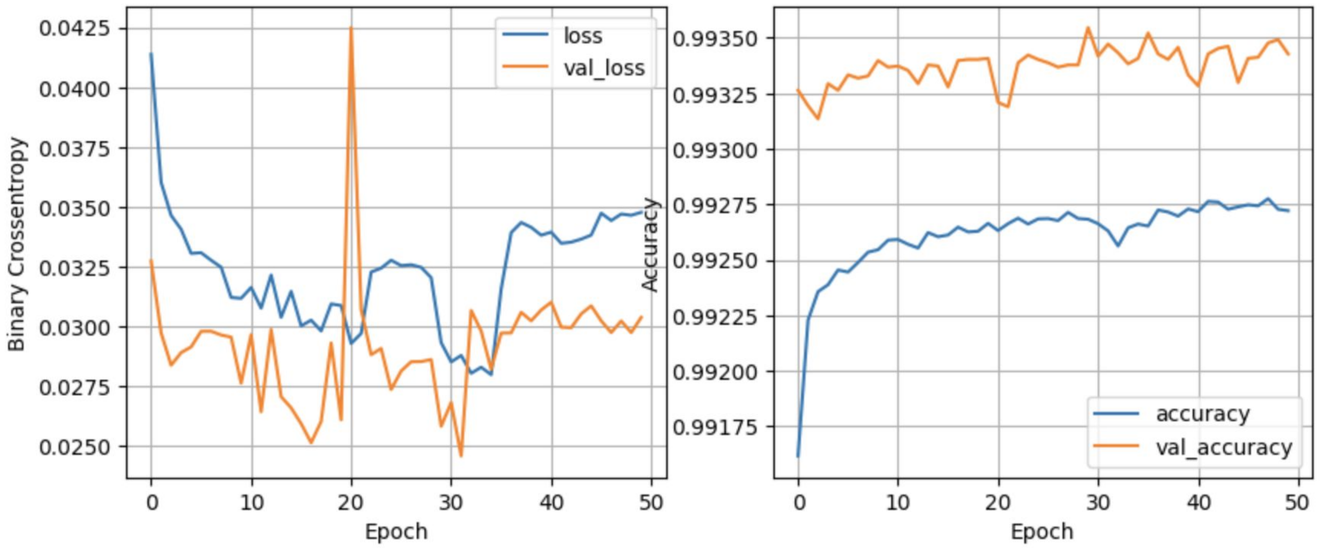
aNN

```
nn_model = tf.keras.Sequential([
        tf.keras.layers.Dense(nodes, activation='relu', input_shape=(5,)),
        # take certain nodes at a specific rate, and dont train them
        tf.keras.layers.Dropout(dropout_prob),
        tf.keras.layers.Dense(nodes, activation='relu'),
        tf.keras.layers.Dropout(dropout_prob),
        # this will set the output to 0 or 1, which helps with the classification
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])
```

```
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]:
                print(f"{nodes} nodes, dropout_prob {dropout_prob}, lr {lr}, batch size {batch_size}")
                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)
                plot_nn(history=history)
                val_loss = model.evaluate(X_test, y_test)[0]
                if val_loss < least_val_loss:
                    least_val_loss = val_loss
                    least_loss_model = model
```

Laufzeit: 12 st

aNN




```
6305/6305 [==============================] - 5s 760us/step - loss: 0.0304 - accuracy: 0.9934
32 nodes, dropout_prob 0.2, lr 0.001, batch size 32
```

Laufzeit: 25 min

aNN - Gridsearch

```
def create_model(neurons=1, activation='relu', dropout_prob=0):
    # create model
    ann = Sequential()
    ann.add(Dense(units = neurons, activation = activation, input_shape=(5,)))
    ann.add(Dense(units = 1))
    ann.add(Dropout(dropout_prob))
    # Compile model
    ann.compile(optimizer = 'adam', loss = 'mean_squared_error')
    return ann
# fix random seed for reproducibility
np.random.seed(0)
# create model
ann = KerasRegressor(build_fn = create_model, epochs = 100, batch_size = 10, verbose = 0)
# define the grid search parameters
parameters = {'neurons': [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60],
              'activation': ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear'],
              'dropout_prob': [0,0.2]}
grid = GridSearchCV(estimator = ann, param_grid = parameters, n_jobs = -1, cv=3)
grid_result = grid.fit(X_train, y_train)
# summarize results
print(f"Best: {grid_result.best_score_} using {grid_result.best_params_}")
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

Abbruch nach 14 st

Datensatz merged

kNN - K-Optimierung (Zusammengefügter Datensatz)

Ursprungsdaten

```python
X = SOURCEDF.drop('label',axis=1)
y = SOURCEDF.label

from sklearn.model_selection import GridSearchCV

grid_params = {
'n_neighbors': [1,2,3,4,5,6,7],
'weights': ['uniform', 'distance'],
'metric': ['euclidean', 'manhattan']
}

gs = GridSearchCV(
    KNeighborsClassifier(),
    grid_params,
    verbose = 1,
    cv = 3,      # Determines the cross-validation splitting strategy
                 # integer, to specify the number of folds
    n_jobs = -1 # use all processors to perform the model
)

gs_results = gs.fit(X, y)
```

Laufzeit: 5 min

Fitting 3 folds for each of 28 candidates, totalling 84 fits

kNN - K-Optimierung (Zusammengefügter Datensatz)

```
gs_results.best_score_
0.8986799842095562

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

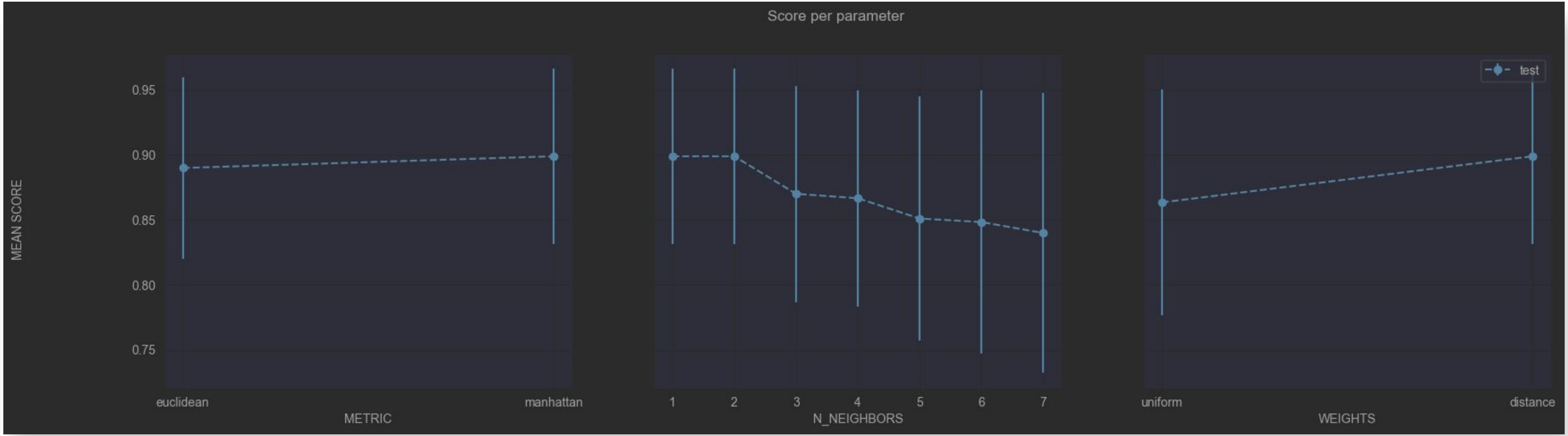
kNN - K-Optimierung (Zusammengefügter Datensatz)

Laufzeit: 2 min

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

knn.score(X_train,y_train)
1.0

knn.score(X_test,y_test)
0.897141417305034

knn.score(X_valid,y_valid)
0.9162863534675615
```

```
y_pred = knn.predict(X_valid)
print(classification_report(y_valid, y_pred))

'''
              precision    recall  f1-score   support

           0       0.93      0.90      0.92    212630
           1       0.91      0.93      0.92    212020

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

kNN - K-Optimierung (Zusammengefügter Datensatz)



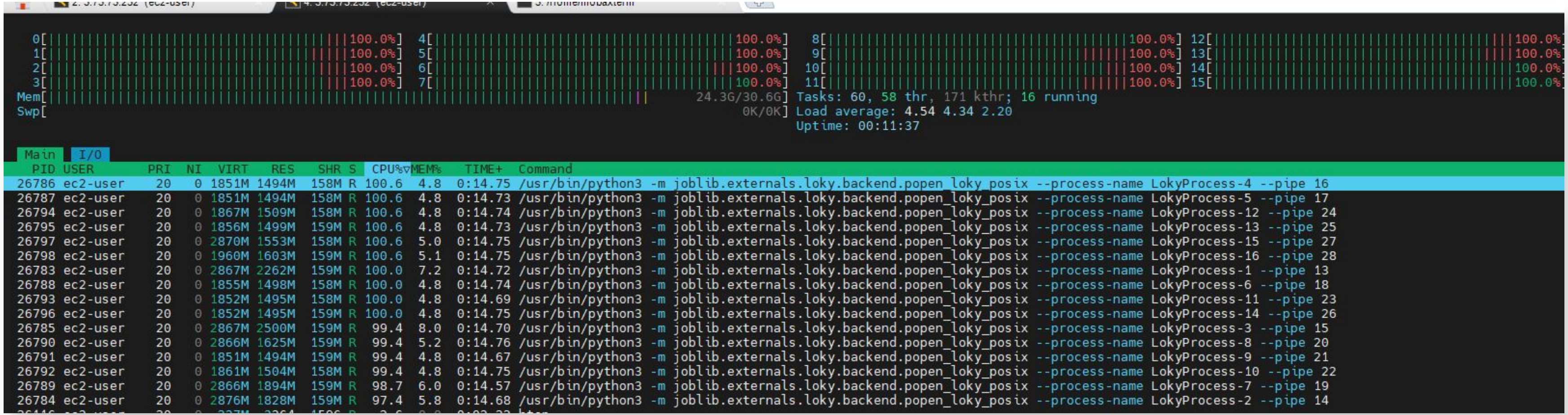
kNN - GridSearchCV mit mehr Parametern

'weights': ['uniform', 'distance'],
'metric': ['euclidean', 'manhattan', **'haversine'**, **'nan_euclidean'**],

Was macht man, wenn man nicht genug CPU/RAM Leistung hat?

-> man nutzt die AWS Cloud

Model	vCPU	Memory (GiB)
c5.large	2	4
c5.xlarge	4	8
c5.2xlarge	8	16
c5.4xlarge	16	32
c5.9xlarge	36	72



kNN - K-Optimierung (Zusammengefügter Datensatz) GridSearchCV

Output

```
Fitting 3 folds for each of 40 candidates, totalling 120 fits
[CV 1/3; 1/40] START metric=euclidean, n_neighbors=1, weights=uniform..........
[CV 2/3; 1/40] START metric=euclidean, n_neighbors=1, weights=uniform..........
[CV 2/3; 2/40] START metric=euclidean, n_neighbors=1, weights=distance.........
[CV 3/3; 1/40] START metric=euclidean, n_neighbors=1, weights=uniform..........
[CV 2/3; 3/40] START metric=euclidean, n_neighbors=2, weights=uniform..........
[CV 1/3; 2/40] START metric=euclidean, n_neighbors=1, weights=distance.........
[CV 1/3; 3/40] START metric=euclidean, n_neighbors=2, weights=uniform..........
[CV 3/3; 3/40] START metric=euclidean, n_neighbors=2, weights=uniform..........
[CV 3/3; 2/40] START metric=euclidean, n_neighbors=1, weights=distance.........
[CV 1/3; 4/40] START metric=euclidean, n_neighbors=2, weights=distance.........
[CV 2/3; 4/40] START metric=euclidean, n_neighbors=2, weights=distance.........
[CV 2/3; 2/40] END metric=euclidean, n_neighbors=1, weights=distance;, score=0.908 total time=   9.8s
[CV 3/3; 4/40] START metric=euclidean, n_neighbors=2, weights=distance.........
[CV 2/3; 4/40] END metric=euclidean, n_neighbors=2, weights=distance;, score=0.908 total time=  16.0s
[CV 3/3; 2/40] END metric=euclidean, n_neighbors=1, weights=distance;, score=0.964 total time=  16.0s
[CV 1/3; 5/40] START metric=euclidean, n_neighbors=3, weights=uniform..........
```

Probleme:
- durch OOM-Killer
    - n_jobs = 11,
    - pre_dispatch = 11
- Sehr lange Ausführungszeiten! auch mit mehr Ressourcen
- Lösung: Maximal **zwei** Metrics verwendet, nicht alle gleichzeitig

kNN - K-Optimierung (Zusammengefügter Datensatz) GridSearchCV

Auszug aus den Ergebnissen:

'metric': ['l1', 'l2'] =
'Algorithm': default

{'metric': 'l1', 'n_neighbors': 2, 'weights': 'distance'}
0.8986799842095562

'metric': ['euclidean', 'manhattan'],
'algorithm': ['auto', 'ball_tree']

{'algorithm': 'auto', 'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'distance'}
0.8986799842095562

Dennoch keine weiteren Verbesserungen auf den Split-Daten!

dTrees (Zusammengefügter Datensatz) - Hyperparameter Tuning

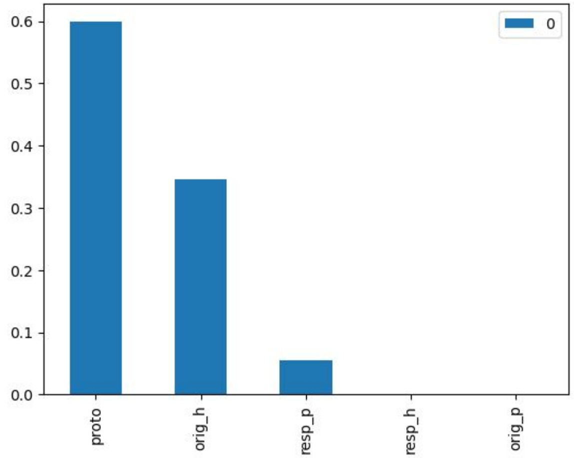
```python
def dtree_grid_search(X,y,nfolds):
    #create a dictionary of all values we want to test
    param_grid = {'max_depth': [1,2,3,4],'random_state': [None,1,2], 'criterion': ['entropy', 'gini']}
    # decision tree model
    dtree_model=DecisionTreeClassifier()
    #use gridsearch to test all 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_
```

Laufzeit: ca. 3 min (auf 48 Kernen)

```
0.9858344685226235
{'criterion': 'gini', 'max_depth': 4, 'random_state': 1}
```

dTrees (Zusammengefügter Datensatz)

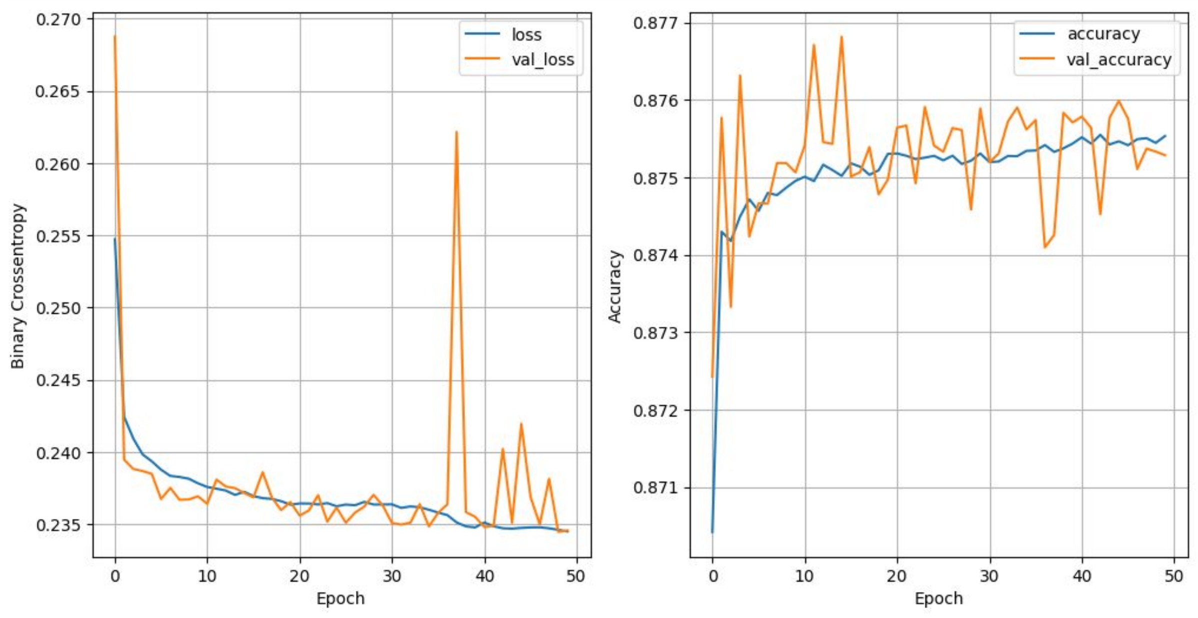
Feature importances



Classification Report

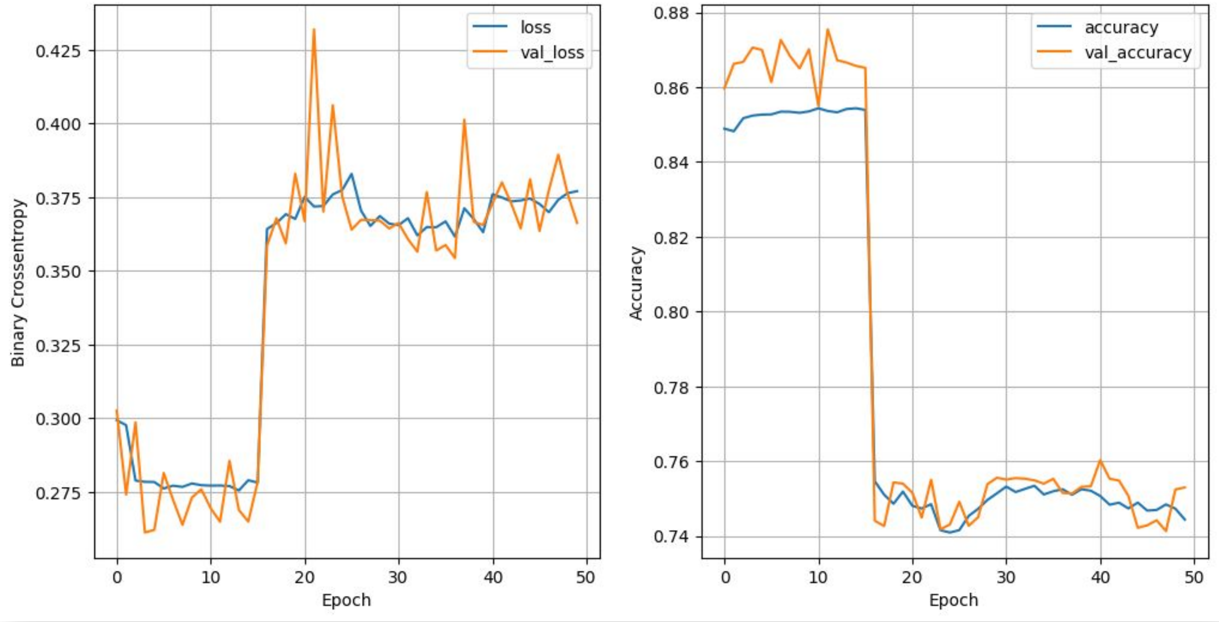
```
              precision    recall  f1-score   support

      benign       0.79      0.95      0.86    212148
   malicious       0.94      0.74      0.83    212502

    accuracy                           0.85    424650
   macro avg       0.86      0.85      0.85    424650
weighted avg       0.86      0.85      0.85    424650
```

aNN (Zusammengefügter Datensatz)



loss: 0.2343 - accuracy: 0.8752
16 nodes, dropout_prob 0, lr 0.005, batch size 64

aNN (Zusammengefügter Datensatz)



loss: 0.3613 - accuracy: 0.7534
16 nodes, dropout_prob 0, lr 0.1, batch size 64

Vergleich der Trainings (beider Datensatz)

Modelle	Capture-1-1 Datensatz	Zusammengefügter Datensatz
kNN	99,85 %	91,62 %
dTrees	99,99 %	84,86 %
aNN	99,87 %	87.5%

Datensatz Quellen

INTRUSION DETECTOR LEARNING
https://www.openml.org/search?type=data&status=active&sort=runs&id=42746

HIKARI-2021: Generating Network Intrusion Detection Dataset Based on Real and Encrypted Synthetic Attack Traffic
https://zenodo.org/record/5199540#.ZFn2FC9BxB0

Kaggle IDS
https://www.kaggle.com/datasets/amankumar255/network-intrusion-detection

Cyber-security Datasets
https://www.kaggle.com/discussions/general/335189

Kyoto University's Honeypots
http://www.takakura.com/Kyoto_data/new_data201704/

IoT Dataset (das was wir nutzen)
https://www.stratosphereips.org/datasets-iot23

Datensatz Quellen

Dataset

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

[1]

https://mcfp.felk.cvut.cz/publicDatasets/IoTDatasets/CTU-IoT-Malware-Capture-1-1/bro/conn.log.labeled

[2}
https://mcfp.felk.cvut.cz/publicDatasets/IoTDatasets/CTU-IoT-Malware-Capture-7-1/bro/conn.log.labeled

[3]
https://mcfp.felk.cvut.cz/publicDatasets/IoTDatasets/CTU-IoT-Malware-Capture-35-1/bro/conn.log.labeled

Vielen Dank für Eure Aufmerksamkeit!