```
#%% Model class
class LinearRegression(torch.nn.Module
    def __init__(self, input_size, output_si
        super(LinearRegression, self).__init_
        self.linear = torch.nn.Linear(input_s

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

#%% Model instance
```

Deep Learn
Linear Reg
Classifica
Perceptron
Computer V
Recommenda
Autoencode

PyTorch

The Practical Guide

Bert Gollnick

Rheinwerk Computing

Contents

Chapter 3
Classification Models

"Science is the systematic classification of experience."
—George Henry Lewes

Science is about learning from experience, observations, and experiments. We record these systematically to help us learn from them. We can find, analyze, and classify patterns in the data, and in this respect, there is a great deal of similarity between science and machine learning.

Sorting or classifying things is one of the most fundamental and widely used tasks in machine learning. We encounter classification when assessing whether an email is spam or not, diagnosing diseases, and recognizing objects in images, to name just a few examples.

In this chapter, we'll first look at different classification types in Section 3.1. There are various methods for checking the results of a classification algorithm, and we can display the predictions with the actual class assignments by using a confusion matrix, which we'll explore in Section 3.2. A confusion matrix is useful for evaluating the results of a model, but the ROC curve is more suitable for comparing several models, and we'll discuss it in detail in Section 3.3.

We'll then begin the practical implementation of classification models. We'll get to know an implementation of binary classification that deals with the prediction of two states in Section 3.4. The implementation of multiclass classification, in which the model learns to distinguish among three or more different categories, is somewhat broader, and we'll explore it in Section 3.5.

We'll start with an introduction to the different classification types.

3.1 Classification Types

The simplest way to perform classification is to divide the data into two different classes. This is referred to as *binary classification*. Imagine that you're dealing with the classification of pictures. The possible classes into which you can divide the images "tree" and "house," so the model only recognizes these two categories. Examples of binary classification are shown in Figure 3.1.

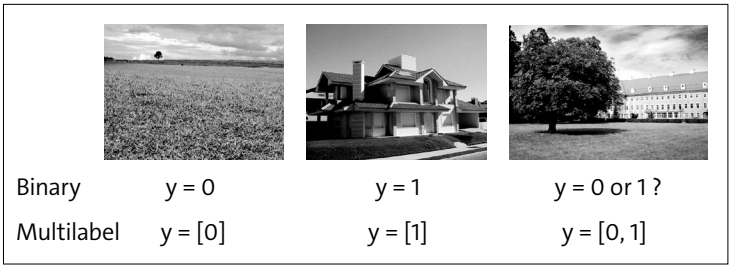


Figure 3.1 Binary Classification

As models don't work with terms such as "tree" and "house" but do work with numerical target values, we must code these categories. In the example, "tree" is always coded with class 0 and "house" is always coded with class 1.

This works very well for images in which only one of the two categories is displayed, but it becomes problematic as soon as several categories are displayed in the image. This can be seen in Figure 3.1 in the image on the right, which shows a tree and a house. The creator of the dataset has the problem of having to choose one of the two categories and the model, which can only predict one of the two classes.

The solution here could be multilabel classification. In this case, we can assign each image to several classes. For example, we can code the image shown on the right as [0, 1], which means "both the tree class tree and the house class are in the image."

This concept is, of course, not limited to two different categories. We can define any number of categories. This can be seen clearly in Figure 3.2, in which another class, "Street," is coded. We're now dealing with three different classes: tree = 0, house = 1, and street = 2. Of course, the principle isn't limited to three classes—we can easily extend it to *N* classes.

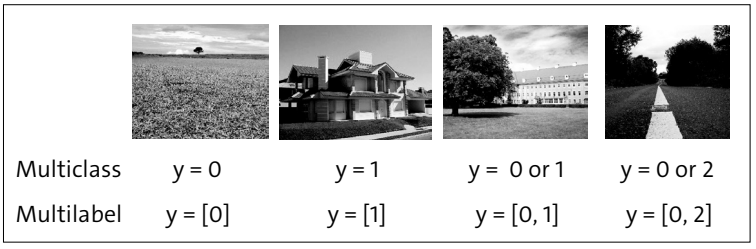


Figure 3.2 Multiclass and Multilabel Classification

If we assign each image to exactly one class, we'll be dealing with multiclass classification again. If, on the other hand, we can assign each image to one or more classes, then that's called *multilabel classification*.

3.2 Confusion Matrix

Before we train an initial model, we need to know how to distinguish a good classifier from a bad one. This is where the *confusion matrix* comes into play. We use it to compare predictions with real values, and it basically has four different fields. As an example, let's take a binary classification model that has been trained to predict tsunamis. We can classify the predictions as follows:

- **True positive (TP)**
  The model didn't predict a tsunami and was right.
- **True negative (TN)**
  The model correctly predicted that a certain event was not a tsunami, and it actually represents a situation in which no tsunami occurred.
- **False positive (FP)**
  The model predicted that a tsunami would occur, and it didn't. This is also referred to as a *false alarm*.
- **False negative (FN)**
  The model predicted that there would be no tsunamis, overlooking a situation in which such an extreme event actually occurred.

We can assign all predictions to one of the four classes, and all values will be aggregated and displayed in a matrix. This matrix, shown in Figure 3.3, is the confusion matrix.

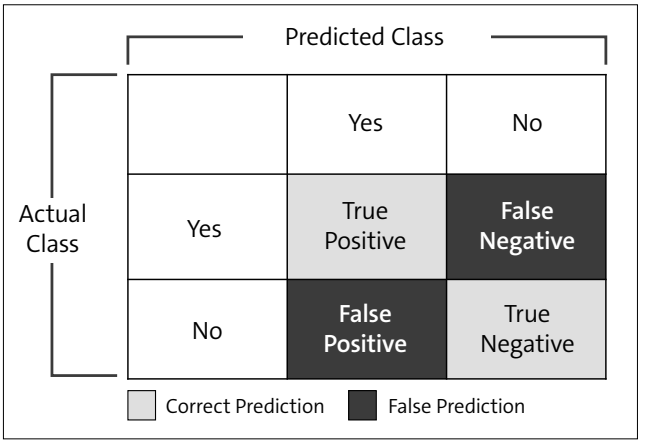


Figure 3.3 Confusion Matrix

In this matrix, the values of the actual and predicted classes are divided into the four fields, and we can then use these four values to derive further measures, such as accuracy, precision, and sensitivity.

In the case of *accuracy*, for example, all correct predictions (*TP* + *TN*) are set in relation to all predictions. The formula reads accordingly:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Precision is calculated as the ratio of true positive predictions to the total number of positive predictions. A high precision value (close to 1) means that the model produces only a few FPs when it makes a positive classification. The positive predictions of the model are therefore very reliable:

$$Precision = \frac{TP}{TP + FP}$$

Recall is calculated as the ratio of true positive predictions to the total number of actual positive cases. A high recall value (close to 1) means that the model makes few FN predictions, which means that it only overlooks a few of the cases that are actually positive:

$$Recall = \frac{TP}{TP + FN}$$

We've neglected one detail up to this point: the threshold value. For example, the model provides us with a probability that a tsunami will or won't occur, but only when we apply a threshold value can we derive a clear prediction class. If we code the classes as 0 (no tsunami will occur) and 1 (a tsunami will occur), we can derive classes from the prediction probabilities.

Let's assume that the prediction value is 0.65. The class assignment will depend on the threshold value, as follows:

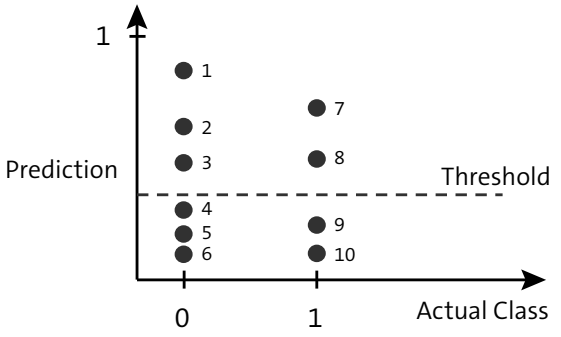
- If the threshold value is 0.5, then every value above 0.5 is classified as class 1 and every value below 0.5 is classified as 0. In our example, class 1 would be predicted.
- This would change if the threshold value were 0.8. In that case, the predicted value of 0.65 would be below the threshold value and class 0 would be predicted.

Figure 3.4 shows vertically the predictions of 10 data points for the two classes (0 or 1). A threshold value is shown with a dashed line Above the threshold value, the points are assigned to class 1, and below the threshold value, the points are assigned to class 0. The actual classes (0 and 1) are plotted on the horizontal axis.

In the table at the bottom of Figure 3.4, the actual class, the class predicted by the model, and the assigned group resulting from the two values are entered for each data point.

Class 1 was predicted for data point 1, although the actual class is 0. There was therefore an incorrect assignment—specifically, an FP.

In the next step, the points of all groups are counted and entered into the confusion matrix. The result is shown in Figure 3.5.



Point Number	1	2	3	4	5	6	7	8	9	10
Actual Class	0	0	0	0	0	0	1	1	1	1
Predicted Class	1	1	1	0	0	0	1	1	0	0
Assigned Group	FP	FP	FP	TN	TN	TN	TP	TP	FN	FN

Figure 3.4 Model Predictions and Classes

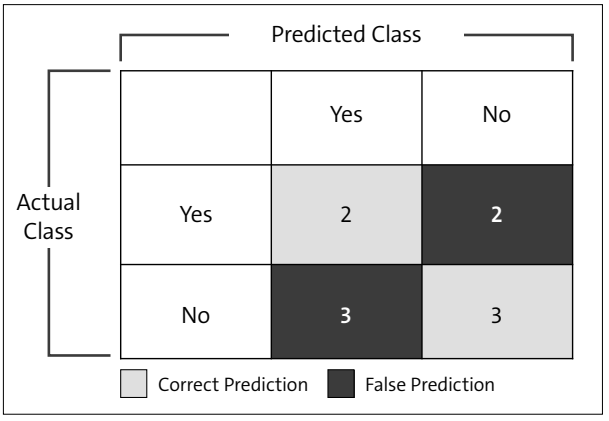


Figure 3.5 Confusion Matrix for Our Example

The confusion matrix gives us a snapshot of the model, but with the ROC curve, we can generalize the concept.

3.3 Receiver Operator Characteristic Curve

The receiver operating characteristic curve (ROC curve) is based on a concept that was developed during the Second World War in an effort to detect enemy vessels such as

submarines. The concept was later adopted in many other scientific disciplines, such as psychology, medicine, and the prediction of extreme weather events. Ultimately, the concept was also adopted in machine learning to assess the quality of models.

In the previous section, we learned that the threshold value has a decisive influence on the confusion matrix. Each change in the threshold value leads to a different confusion matrix. In this section, we'll see that the ROC curve shows us how well our model works across all possible threshold values. You only need to perform the following steps:

1. Determine the sensitivity, which is also known as the *true positive rate* (TPR), and the specificity, which is also known as the *true negative rate* (TNR), for various threshold values between 0 and 1. However, you should use the *false positive rate*, which we abbreviate as 1 – specificity (we'll use this abbreviation again later).
2. Plot both values on a diagram, with FPR on the x-axis and TPR on the y-axis.

One result is illustrated in Figure 3.6.

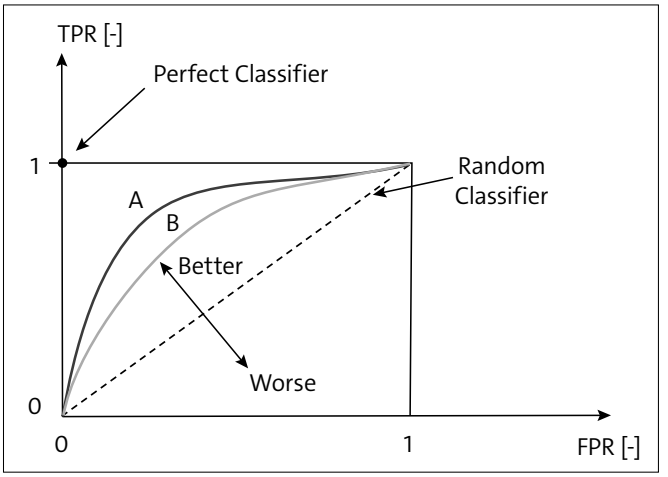


Figure 3.6 ROC Curve

The perfect classification model has an FPR of 0 and a TPR of 1, so it occupies the top left corner. The opposite is a model that has learned nothing and makes predictions completely at random (i.e., it's no better than guessing). Such a model would be exactly on the diagonal. Practical models are located somewhere in this area of tension, with better models being shifted to the top left. In our example, model A shows a consistently better performance than model B.

The concept of the ROC curve can be condensed into a single numerical value: the area under curve (AUC), which is the area under the ROC curve. It summarizes the performance of the model in a single scalar value between 0 and 1. The AUC is the probability that the model will rate a randomly selected positive case higher than a randomly

selected negative case. At a value of 1, the model always makes the right decision, whereas at a value of < 0.5, the model performs worse than a model based purely on chance.

3.4 Coding: Binary Classification

In our first example, we'll train a model that predicts whether network traffic is normal behavior or a hacker attack (meaning it's a model that performs intrusion detection). We'll use the "Intrusion Detection Logs" dataset from Kaggle for this purpose. We'll have to prepare the data first, and then, we'll train the model and evaluate the model results.

3.4.1 Data Preparation

The entire script for this example can be found in *040_Classification\data_prep_binary.py*. Listing 3.1 shows us the required packages, and we'll load the data directly from Kaggle using `kagglehub`. The data is available as a CSV file, so we'll use `pandas` to load and modify the data. We'll also separate the data into training and test data by using functions from the `sklearn` package. Finally, we'll use the `matplotlib` and `seaborn` packages for visualization.

```
#%% packages
import kagglehub
import os
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
```

Listing 3.1 Binary Classification: Import Packages

Let's start by importing the data as shown in Listing 3.2. To do this, we can use the `kagglehub` package directly by passing the ID of the dataset to the `dataset_download` function. This ensures that the data is downloaded to the local hard disk—namely, to a folder whose path is stored in the path variable. This path contains the *Network_logs.csv* file, which we load with `pd.read_csv`. The file has 8,846 data points and 10 features.

```
#%% data import
path = kagglehub.dataset_download("developerghost/intrusion-detection-logs-normal-bot-scan")

print("Path to dataset files:", path)
```

```
file_path = os.path.join(path, "Network_logs.csv")
df = pd.read_csv(file_path)
```

```
(8846, 10)
```

Listing 3.2 Binary Classification: Import Data

First, let's look at the dataset. Figure 3.7 shows the first four lines of it.

	Source_IP	Destination_IP	Port	Request_Type	Protocol	Payload_Size	User_Agent	Status	Intrusion	Scan_Type
0	192.168.142.55	42.156.67.167	80	FTP	UDP	2369	curl/7.68.0	Success	0	Normal
1	53.39.165.18	94.60.242.119	135	SMTP	UDP	1536	Wget/1.20.3	Failure	1	BotAttack
2	192.168.127.91	7.10.192.3	21	SMTP	TCP	1183	Wget/1.20.3	Success	0	Normal
3	192.168.30.40	130.169.82.211	25	HTTPS	TCP	666	Mozilla/5.0	Success	0	Normal

Figure 3.7 Binary Classification: Dataset

Now, let's take a closer look at the features we're dealing with:

```
df.columns
```

```
Index(['Source_IP', 'Destination_IP', 'Port', 'Request_Type', 'Protocol',
       'Payload_Size', 'User_Agent', 'Status', 'Intrusion', 'Scan_Type'],
      dtype='object')
```

The IP address from which a packet is sent (`Source_IP`) and the IP address to which the packet is sent (`Destination_IP`) are recorded, and the data record also contains information on which port and which format (`Request_Type`) was used. This refers to the network protocols (e.g. FTP, SMTP, HTTP), which define the rules according to which computers exchange data via the internet.

The port is a virtual address within a computer that enables the operating system to assign the incoming request to a specific application or service.

The size of the parcel is recorded via the `Payload_Size`, and the `Status` feature tells us whether the packet was sent successfully. Finally, we have the `Intrusion` target value, which is binary coded. A value of 0 means that no attack has taken place, whereas a value of 1 represents an attack.

We can safely disregard some of the features because they are not suitable for drawing conclusions about an attack. These include `Source_IP` and `Destination_IP`. An attacker would certainly not launch a reproducible attack from the same source IP and would very probably not always attack only certain destination IP addresses. We can therefore delete these two features with `df.drop`. The `Scan_Type` feature breaks down our target variable more precisely, and we'll come back to that in Section 3.5, when we will no longer only differentiate between two categories (normal network traffic and attacks) but will also want to predict further classes.

In our current script, we can delete the feature, as follows:

```
#%% drop features that are not useful for the analysis
df = df.drop(columns=["Source_IP", "Destination_IP", "Scan_Type"])
```

All data must be available as numerical information if we want to train a model, so let's look at the data types with which the features are coded, as follows:

```
df.dtypes
```

```
Port             int64
Request_Type    object
Protocol        object
Payload_Size     int64
User_Agent      object
Status          object
Intrusion        int64
dtype: object
```

Only some of the features are available as integers; many of them have the `Object` data type. Therefore, we must convert these features into numerical data by using one-hot encoding. We can use the `pd.get_dummies` method for this, and it processes the dataset and converts the features as follows:

```
#%% treat categorical variables
df_cat = pd.get_dummies(df, drop_first=True, dtype=int)
```

The `df_cat.shape` property shows that 7 more columns have been created, so we now have 17 features. The first return value corresponds to the number of rows, and the second value corresponds to the number of columns:

```
df_cat.shape
```

```
(8846, 17)
```

Now, you should familiarize yourself with the target variable because you should generally not assume that the different categories are evenly distributed. For the visualization in Figure 3.8, we can use `seaborn` with the `countplot` function, as follows:

```
#%% visualize the distribution of the target variable
sns.countplot(x="Intrusion", data=df_cat)
plt.title("Target variable imbalance")
```

Figure 3.8 shows that there is significantly more "normal" behavior (indicated by class 0) in the network traffic than attacks (indicated by class 1). This is reassuring, but we'll have to take this into account later if we want to check the quality of the model.

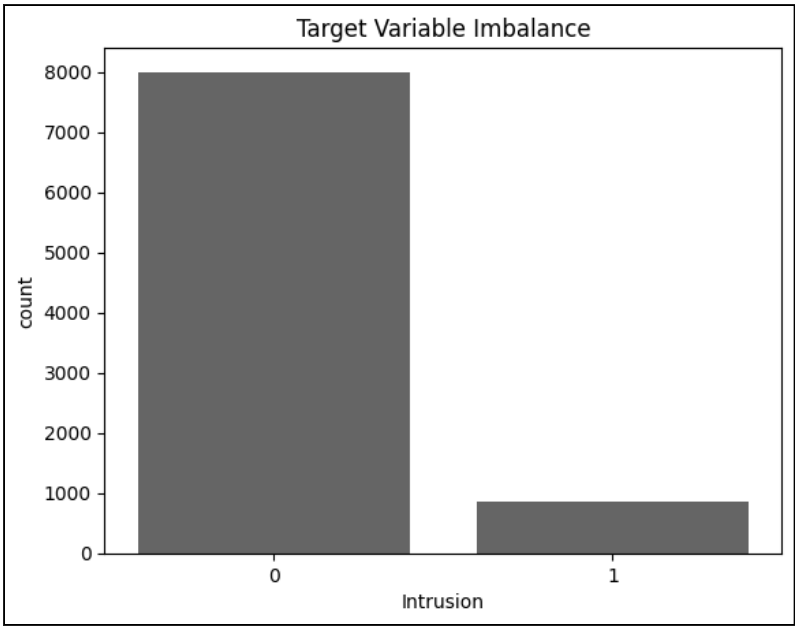


Figure 3.8 Binary Classification: Imbalance of Target Variables

Now, we can look at the correlations (i.e., the relationships) within the features to recognize which features are related to each other. To do this, we first determine the correlation matrix:

```
# Create correlation matrix
corr_matrix = df_cat.corr()
```

The code in Listing 3.3 generates a correlation matrix that is visualized as a heat map.

```
# Create heat map
plt.figure(figsize=(12, 8))
mask = np.triu(np.ones_like(corr_matrix), k=1).T
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0, fmt='.2f',
mask=mask)
plt.title('Correlation matrix of variables')
plt.tight_layout()
```

Listing 3.3 Binary Classification: Correlation Matrix Visualization

Figure 3.9 shows the correlation matrix for all variables. All features are shown as rows and columns, and for each combination of features, the correlation value shows how strong the linear relationship between the two variables is. The larger the value, the stronger the correlation.

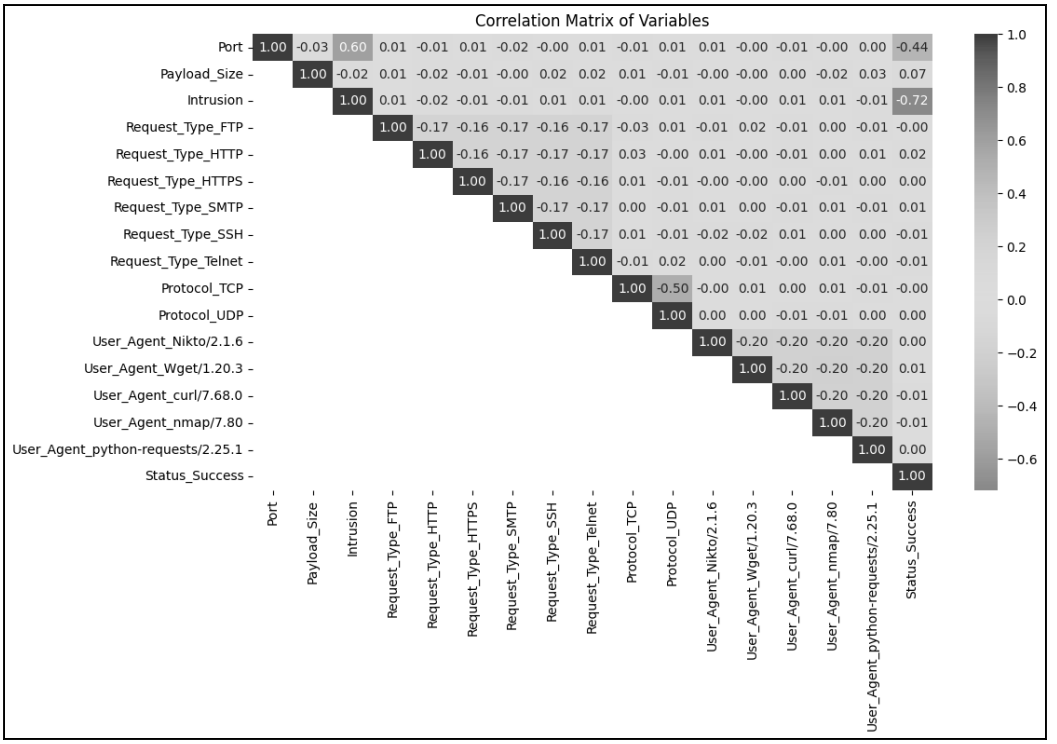


Figure 3.9 Binary Classification: Correlation Matrix

Above all, we're interested in the connection with the `Intrusion` target variable, so we can concentrate on line 3. All correlations between the independent variables and the target variable can be seen there.

The strongest correlation is with the `Status_Success` variable, which has a strong negative value that we can interpret to mean that unsuccessful parcel deliveries are more likely to indicate an attack.

As usual, we'll separate the independent (`X`) and dependent (`y`) features from each other, and we'll also use the `drop` method again. We'll remove the `Intrusion` column from the independent features and store it in `y` as a dependent feature to act as a target variable, as follows:

```
#%% separate independent and dependent variables
X = df_cat.drop(columns=["Intrusion"])
y = df_cat["Intrusion"]
print(f"X shape: {X.shape}, y shape: {y.shape}")
```

```
X shape: (8846, 16), y shape: (8846,)
```

At this point, we can split the data into training and test data by using `train_test_split`. In this case, we use 90% of the data for training and 10% for testing. The `stratify` parameter ensures that the ratio of classes in the target variable `y` in the generated training and test datasets remains the same as in the original overall dataset:

```
#%% split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, ran-
dom_state=42, stratify=y)
print(f"X_train shape: {X_train.shape}, X_test shape: {X_test.shape}")
```

```
X_train shape: (7961, 16), X_test shape: (885, 16)
```

At this point, we have reached the end of the data preparation and can start creating the model.

3.4.2 Modeling

Now, we come to the core of our classification: creating and training the model. The final script can be found in *040_Classification\model_binary.py*. We start by loading the required packages using the code in Listing 3.4, and we load the `DataLoader` and a class called `TensorDataset` (a simplified form of the previously used `Dataset` class) from the `torch` package.

In addition, we import packages for evaluating the model result (`sklearn`) and for visualization (`seaborn` and `matplotlib`). Finally, we need the split training and test data, which is provided via the previously created `data_prep_binary` script.

```
from torch.utils.data import DataLoader, TensorDataset
import torch
import torch.nn as nn
import torch.optim as optim
import seaborn as sns
import numpy as np
from sklearn.metrics import confusion_matrix, accuracy_score, roc_curve, auc
from sklearn.dummy import DummyClassifier
import matplotlib.pyplot as plt
import torch
from data_prep_binary import X_train, X_test, y_train, y_test
```

Listing 3.4 Binary Classification: Import Packages for Modeling

The model and the training are controlled via the hyperparameters listed in Listing 3.5, in which the model is parameterized via the number of nodes in the `HIDDEN_SIZE` hidden layer and the number of `OUTPUT_SIZE` output nodes. The actual model training is parameterized via the `BATCH_SIZE`, the `LEARNING_RATE`, and the number of `EPOCHS`. The `DEVICE`

variable is created to be able to execute the code flexibly on the GPU (if available) or (if not) on the CPU.

```
#%% Hyperparameters
BATCH_SIZE = 32
LEARNING_RATE = 0.0005
EPOCHS = 40
HIDDEN_SIZE = 4
OUTPUT_SIZE = 1
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Listing 3.5 Binary Classification: Hyperparameter Definition

Now, let's move on to creating the `train_dataset` and `test_dataset` datasets. For this, we use an instance of the `TensorDataset` class, which is very easy to use. The independent and dependent variables are passed to it, and if no further preprocessing steps are required, we can use this simple class instead of the dataset class:

```
# %% create a custom dataset class
train_dataset = TensorDataset(torch.FloatTensor(X_train.values), torch.Float-
Tensor(y_train.values))
test_dataset = TensorDataset(torch.FloatTensor(X_test.values), torch.FloatTen-
sor(y_test.values))
```

To create the `DataLoader`, we call the class of the same name and pass the `BATCH_SIZE` and the `shuffle` parameter for random sampling, as follows:

```
# %% create a data loader
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

Now, let's move on to creating the model. We need to start by gaining an understanding of the activation of the output layer and the loss model used. Figure 3.10 shows what options we have here in the context of binary classification. We already know that the output layer will have a node that will give us the probability of class membership, but how should we activate this output layer? We have the following two options, and both are directly linked to the loss model used:

- We can activate the output layer with *sigmoid* and use *BCELoss* as the loss function.
- We can dispense with sigmoid activation if we use the *BCEWithLogitsLoss* loss function, which already has sigmoid activation integrated. If we were to use sigmoid in addition to this loss function in the network, we would be activating twice with sigmoid, which could lead to instability and poor learning progress.

In general, we recommend that you use the latter approach (without explicit sigmoid activation) as your standard approach.

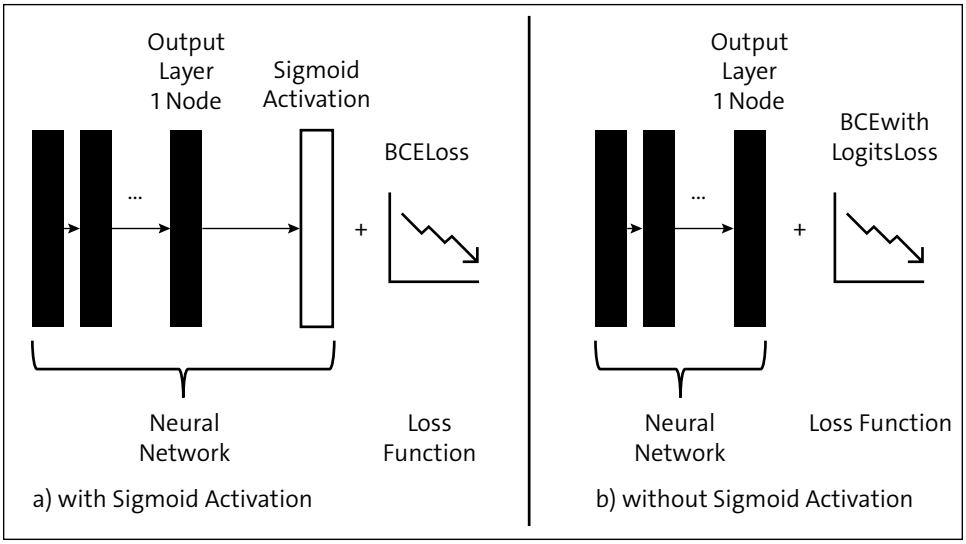


Figure 3.10 Binary Classification: Output Layer and Loss Function

In Listing 3.6, we create the model class. The model consists of its hidden layer with `hidden_size` nodes. The model parameters of the hidden layer are activated with `ReLU`, and `Dropout` is used for regularization. The output layer isn't explicitly activated, as the optimizer takes care of this and activates the data internally with `Sigmoid`.

```
class BinaryClassificationModel(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, dropout_rate=0.2):
        super(BinaryClassificationModel, self).__init__()
        self.lin1 = nn.Linear(input_size, hidden_size)
        self.lin2 = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        x = self.lin1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.lin2(x)
        return x
```

Listing 3.6 Binary Classification: Model Class Definition

We create the model instance in Listing 3.7. To be able to apply the code flexibly to other data, we determine the `input_size` based on the training data. We then create the model by passing the corresponding `input_size`, `hidden_size`, and `output_size` parameters.

```
input_size = X_train.shape[1]

model = BinaryClassificationModel(input_size=input_size, output_size=OUTPUT_
SIZE, hidden_size=HIDDEN_SIZE).to(DEVICE)
```

Listing 3.7 Binary Classification: Model Instance Creation

We use the now familiar `Adam` as the optimizer (see Listing 3.8), but the loss function is more exciting. As previously explained, we use `BCEWithLogitsLoss`, which already includes sigmoid activation.

```
# %% optimizer and loss function with weight decay for regularization
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=1e-4)
loss_fn = nn.BCEWithLogitsLoss()
```

Listing 3.8 Binary Classification: Optimizer and Loss Function

We perform the actual model training as shown in Listing 3.9. We follow the usual procedure: we iterate over the number of epochs and perform a forward pass, the loss calculation, a backward pass, and the update of the parameters in each epoch.

```
#%% training loop
train_losses = []
for epoch in range(EPOCHS):
    train_loss = 0
    for X_train_batch, y_train_batch in train_loader:
        # move data to device
        X_train_batch, y_train_batch = X_train_batch.to(DEVICE), y_train_
batch.to(DEVICE)
        # forward pass
        y_train_batch_pred = model(X_train_batch)
        # calculate loss
        loss = loss_fn(y_train_batch_pred, y_train_batch.reshape(-1, 1))
        # backward pass
        loss.backward()
        # update weights
        optimizer.step()
        # reset gradients
        optimizer.zero_grad()
        # update train loss
        train_loss += loss.item()
    # append train loss
    train_losses.append(train_loss)
    print(f"Epoch {epoch+1}/{EPOCHS}, Loss: {train_losses[-1]:.4f}")
```

Listing 3.9 Binary Classification: Model Training

Now, let's look at whether our training was successful. To do this, we use `seaborn` as an `sns` alias (as in Listing 3.10) and visualize the training loss and the epochs.

```
# %% visualize training loss
plt.figure()
sns.lineplot(x=list(range(EPOCHS)), y=train_losses)
plt.xticks(range(EPOCHS))
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Over Time')
```

Listing 3.10 Binary Classification: Loss Visualization

Figure 3.11 shows the progression of training losses over the epochs. The training is exemplary because the losses initially decrease sharply and then hardly any improvements are achieved from around 20 epochs onward.

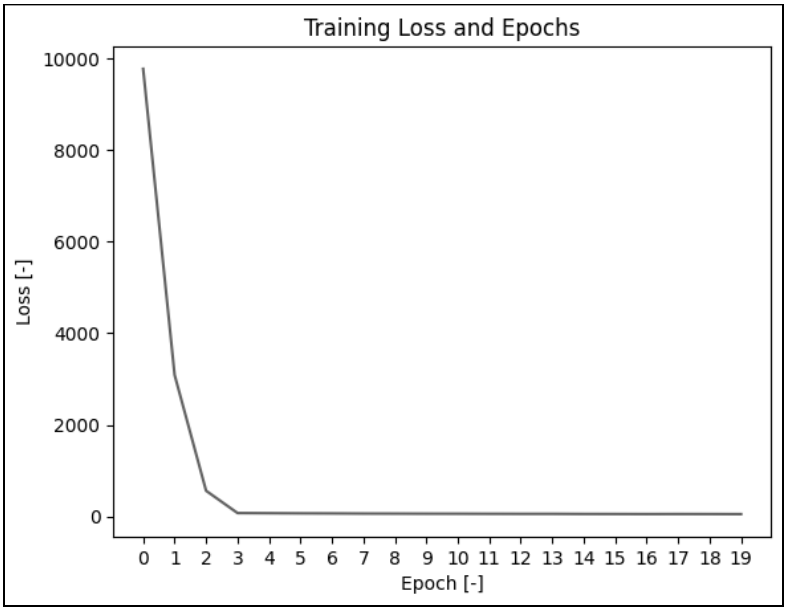


Figure 3.11 Binary Classification: Training Loss and Epochs

3.4.3 Evaluation

Now, let's move on to checking the model by using the test data. Listing 3.11 shows how we let the model predict the `y_test_pred_proba` probabilities to then determine the `y_test_pred_class` classes. We use the `THRESHOLD` limit value for this: whenever the prediction is above the threshold value, it's rounded up to 1, and whenever the prediction is below it, it's rounded down accordingly.

```
y_test_pred_proba, y_test_pred_class, y_test_true = [], [], []
THRESHOLD = 0.5

model.eval()  # Set model to evaluation mode
with torch.no_grad():
    for X_test_batch, y_test_batch in test_loader:
        X_test_batch, y_test_batch = X_test_batch.to(DEVICE),
                                     y_test_batch.to(DEVICE)
        y_test_batch_pred = model(X_test_batch)
        y_test_batch_pred = y_test_batch_pred.cpu().numpy()

        # Store probabilities for ROC curve
        y_test_pred_proba.extend(y_test_batch_pred.flatten().tolist())

        # Store class predictions for confusion matrix
        y_test_batch_pred_class = (y_test_batch_pred > THRESHOLD).astype(int)
        y_test_batch = y_test_batch.cpu().numpy()
        y_test_pred_class.extend(y_test_batch_pred_class.flatten().tolist())
        y_test_true.extend(y_test_batch.flatten().tolist())
```

Listing 3.11 Binary Classification: Model Evaluation

We use these together with the real `y_test_true` target values to display them in a confusion matrix. The code we use for this is shown in Listing 3.12.

```
#%% create confusion matrix
cm = confusion_matrix(y_true=y_test_true, y_pred=y_test_pred_class)
plt.figure()
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion matrix (binary Classification)')
```

Listing 3.12 Binary Classification: Confusion Matrix Visualization

Figure 3.12 shows the confusion matrix of the test data. The predicted classes are shown on the horizontal axis, and the actual classes are shown on the vertical axis.

We can see here that the model is apparently doing a good job. There were no FPs (as indicated in the top right field), but there were some FNs (as indicated in the bottom left field).

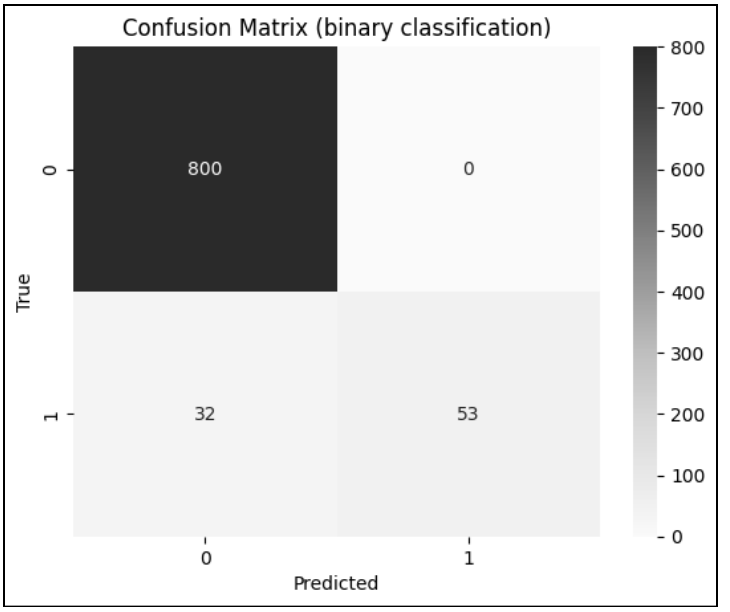


Figure 3.12 Binary Classification: Confusion Matrix

We can also derive the accuracy metric from the confusion matrix. To do this, we can pass the actual and predicted values to the `accuracy_score` function and obtain the corresponding score, as follows:

```
#%% accuracy score
accuracy_score(y_true=y_test_true, y_pred=y_test_pred_class)
```

```
0.9683615819209039
```

The model has an accuracy of 96.8%. That sounds like a very good value, but we must always bear in mind that the target size is very unevenly distributed—there is significantly more normal network traffic (class 0) than malicious network traffic (class 1).

So, to make a definitive statement as to whether the model provides better predictions than pure guessing, we need to train a dummy classifier. This is a class of `scikitlearn` that helps us compare the quality of the model with a model that is only based on pure guessing.

The dummy classifier always predicts the most frequent class and, in our case, delivers an accuracy of 90.2%, as follows:

```
#%% naive classifier and accuracy score
model_naive = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
y_test_pred_naive = model_naive.predict(X_test)
```

```
accuracy_score(y_true=y_test_true, y_pred=y_test_pred_naive)
```

```
0.9016949152542373
```

We can therefore conclude that our model works much better than pure guesswork.

Finally, we visualize the ROC curve based on the code in Listing 3.13. We use the `roc_curve` function, which provides us with the FP rate (`fpr`) and the TP rate (`tpr`). We also determine the area under curve (`auc`) to display it in the graph, and we create the actual diagram with matplotlib.

```
fpr, tpr, thresholds = roc_curve(y_true=y_test_true, y_score=y_test_pred_proba)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'AUC = {roc_auc:.2f}')
plt.plot([0, 1], [0, 1], color='red', lw=1, linestyle='--', label='Random Clas-
sifier')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('FPR [-]')
plt.ylabel('TPR [-]')
plt.title('ROC Kurve (binäry Classification)')
plt.legend(loc="lower right")
plt.grid(True, alpha=0.3)
```

Listing 3.13 Binary Classification: ROC Curve Creation

Figure 3.13 shows the ROC curve, in which the `FPR` is plotted horizontally and the `TPR` is plotted vertically. The dashed red line represents the "random" classifier (dummy classifier), which corresponds exactly to the diagonal.

The ROC curve (the solid blue line) initially runs vertically upward from the lower left-hand corner and ends in the upper right-hand corner. We know that better models show ROC curves that are as close as possible to the upper left-hand corner, and that isn't quite the case here. However, the model still works very well.

Normally, we'd train other models now, and we'd plot their ROC curves in this diagram. I'll leave that for you to do as an exercise.

We have reached the end of the section on binary classification. We've trained a simple model for analyzing network traffic, and we've learned that we must pay particular attention to the activation of the output layer (or its omission) and the loss function used. These two aspects will also play a decisive role in the following section on multiclass classification.

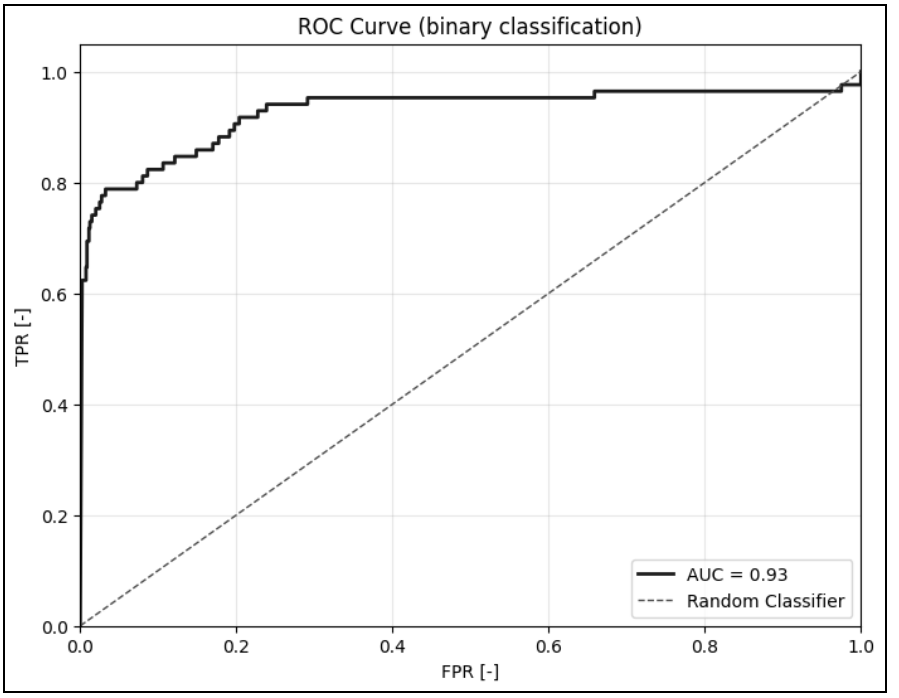


Figure 3.13 Binary Classification: ROC Curve

3.5 Coding: Multiclass Classification

In this example, we'll train a model based on the same dataset as we used in the previous section. What differs is the number of classes of the target variables. Previously, we only had the following two states:

- 0
  Normal network behavior
- 1
  Abnormal network behavior

The dataset offers us the opportunity to break down network behavior in more detail, as follows:

- **Normal**
  This class represents normal behavior.
- **BotAttack**
  This is behavior that suggests an attack.
- **PortScan**
  Port scans are performed by both attackers and security personnel to test security. Therefore, this behavior could be an attack, and it isn't normal behavior.

In this section, we'll learn the implications of predicting multiple classes instead of just two.

Our approach is to import and prepare the data (in Section 3.5.1) before we make use of it in the training of the model (in Section 3.5.2). Finally, we'll evaluate the trained model (in Section 3.5.3).

3.5.1 Data Preparation

The entire script for this example can be found in *040_Classification\model_multiclass.py*. As always, we start loading the packages as in Listing 3.14. To help improve your understanding, we've divided the packages into the areas of data import, data processing, modeling, and visualization. There are no new packages on which we need to go into in more detail.

```
# data import
import kagglehub
# data preparation
import os
import pandas as pd
from sklearn.model_selection import train_test_split
# Modeling
import torch
from torch.utils.data import DataLoader, TensorDataset
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.dummy import DummyClassifier
import numpy as np
# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
```

Listing 3.14 Multiclass Classification: Package Import

In Listing 3.15, the data is loaded directly from Kaggle using the `kagglehub` package and the downloaded CSV file is loaded using `pandas`.

```
#%% data import
path = kagglehub.dataset_download("developerghost/intrusion-detection-logs-nor-
mal-bot-scan")
file_path = os.path.join(path, "Network_logs.csv")
df = pd.read_csv(file_path)
```

Listing 3.15 Multiclass Classification: Data Import

Some features are not required, and we can delete them directly. These include the `Source_IP` and `Destination_IP`, as well as the target variable from the previous `Intrusion` section.

```
#%% drop features that are not useful for the analysis
df = df.drop(columns=["Source_IP", "Destination_IP", "Intrusion"])
```

In the next step, we convert *categorical features* into *numerical features* by using *one-hot encoding*, as follows:

```
#%% treat categorical variables
df_cat = pd.get_dummies(df, columns=[ 'Request_Type', 'Protocol',  'User_
Agent','Status'], drop_first=True, dtype=int)
```

Listing 3.16 shows the separation of independent and dependent variables. As previously mentioned, we use `Scan_Type` as the target variable. We also convert this categorical variable into a numerical format with `pd.factorize` when we create `y`. When creating the independent features `X`, we take into account all features except `Scan_Type`.

```
#%% separate independent and dependent variables
X = df_cat.drop(columns=['Scan_Type']).astype(float)
y = pd.factorize(df_cat["Scan_Type"])[0].astype(float)

print(f"X shape: {X.shape}, y shape: {y.shape}")
```

```
X shape: (8846, 16), y shape: (8846)
```

Listing 3.16 Multiclass Classification: Independent and Dependent Variable Separation

In the next step (in Listing 3.17), we divide the data into training, validation, and test data. If our aim is to split the data into three groups, we must use a two-step procedure because there is no function in `sklearn` that creates the three groups in one step. Therefore, we begin by splitting the data into temporary and test data so that we can create the training and validation data from the temporary data in a second step.

Here, 10% of the data is reserved for testing, and the remaining data is split up in a second step.

```
#%% split data into training, validation, and testing sets
# First, split off test set
X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=0.1, random_
state=42)
# Split remaining data into training and validation
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=
0.2, random_state=42)
```

```
print(f"X_train shape: {X_train.shape}, X_val shape: {X_val.shape}, X_test
shape: {X_test.shape}")
```

```
X_train shape: (6368, 16), X_val shape: (1593, 16), X_test shape: (885, 16)
```

Listing 3.17 Multiclass Classification: Splitting Data into Training, Validation, and Testing

Now, we can pause for a moment and look at the distribution of classes in the training dataset, as shown in Listing 3.18.

```
# Check class distribution
print("Class distribution in training set:")
unique, counts = np.unique(y_train, return_counts=True)
for class_idx, count in zip(unique, counts):
    print(f"Class {class_idx}: {count} samples ({count/len(y_
train)*100:.1f}%)")
```

```
Number of classes: 3
Class distribution in training set:
Class 0.0: 5759 samples (90.4%)
Class 1.0: 346 samples (5.4%)
Class 2.0: 263 samples (4.1%)
```

Listing 3.18 Multiclass Classification: Class Distribution Determination

A predominant share of over 90% of the data records are assigned to class 0, and classes 1 and 2 have nearly the same number of data records at just over 4% and 5%, respectively. The data is therefore very unevenly distributed.

When distributing the classes, it's also advisable to ensure that the distribution among the training, validation, and test data is similar. For this purpose, we use the code in Listing 3.19, which determines the percentages and visualizes them in a bar chart.

```
plt.figure()
# Convert counts to percentages for training data
percentages_train = (counts / len(y_train)) * 100

# Get validation data distribution
unique_val, counts_val = np.unique(y_val, return_counts=True)
percentages_val = (counts_val / len(y_val)) * 100

# Get test data distribution
unique_test, counts_test = np.unique(y_test, return_counts=True)
percentages_test = (counts_test / len(y_test)) * 100
```

```
# Plot distributions
width = 0.25
plt.bar(unique - width, percentages_train, width, label='Training')
plt.bar(unique, percentages_val, width, label='Validation')
plt.bar(unique + width, percentages_test, width, label='Test')

plt.xlabel('Class [-]')
plt.ylabel(Percentage [%]')
plt.title('Class distribution in Training-, Validiation- and Testset')
plt.xticks([0, 1, 2])
plt.legend()
```

Listing 3.19 Multiclass Classification: Class Distribution Visualization

The result of the code can be seen in Figure 3.14.

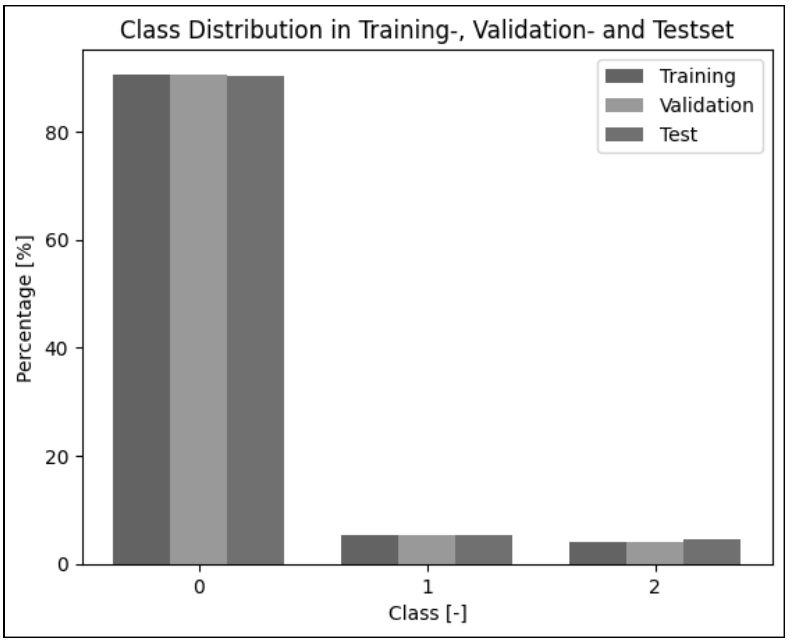


Figure 3.14 Multiclass Classification: Class Distribution in Training, Validation, and Test Datasets

The fact that the classes are very unequal isn't ideal, but it's often unavoidable. However, we need to ensure that the class distribution in the three different datasets is at least similar. In this case, the distributions are very similar, so we can continue with the next step.

Listing 3.20 shows how to create the `dataset` and `DataLoader` instances for the training, validation, and test data. It's worth mentioning here that the data is randomly sampled

for the `train_dataloader`, whereas this isn't necessary for the `val_dataloader` and `test_dataloader`.

Since most classification loss functions in PyTorch require the class labels as 64-bit integers, we must use a `torch.LongTensor` at this point.

```
train_dataset = TensorDataset(torch.FloatTensor(X_train.values), torch.LongTen-
sor(y_train))
val_dataset = TensorDataset(torch.FloatTensor(X_val.values), torch.LongTen-
sor(y_val))
test_dataset = TensorDataset(torch.FloatTensor(X_test.values), torch.LongTen-
sor(y_test))

# %% create a data loader
train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuf-
fle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=BATCH_SIZE, shuffle=
False)
test_loader = DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE, shuffle=
False)
```

Listing 3.20 Multiclass Classification: Dataset and DataLoader

At this point, the `DataLoader` instances are available, and we'll use them in the following modeling and evaluation.

3.5.2 Modeling

At the beginning of the modeling, it makes sense for us to define the hyperparameters that influence the training in one place. The following parameters are defined in Listing 3.21 for this purpose:

- **`BATCH_SIZE`**
  This is the size of the batch.
- **`LEARNING_RATE`**
  This is the learning rate.
- **`EPOCHS`**
  This is the number of epochs.
- **`HIDDEN_SIZE`**
  This is the number of hidden nodes.
- **`OUTPUT_SIZE`**
  This is the number of output nodes.
- **`DEVICE`**
  This is the device on which the training will be performed.

```
BATCH_SIZE = 128
LEARNING_RATE = 0.001
EPOCHS = 50
HIDDEN_SIZE = 64
OUTPUT_SIZE = len(df_cat["Scan_Type"].unique())
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Number of classes: {OUTPUT_SIZE}")
```

Listing 3.21 Multiclass Classification: Hyperparameter Definition

Now, we come to a core element: the structure of the neural network. Figure 3.15 shows the structure of the network that we are implementing. There are three blocks, each of which has a linear layer, batch normalization, an activation using `ReLU`, and dropout to prevent overfitting.

Here, we can recognize a pattern in which the number of features and thus the number of parameters decreases with the depth of the network. This is very common and useful in many areas of deep learning because it reduces complexity, and the reduction of parameters in the later layers reduces the overall complexity and computational load of the model.

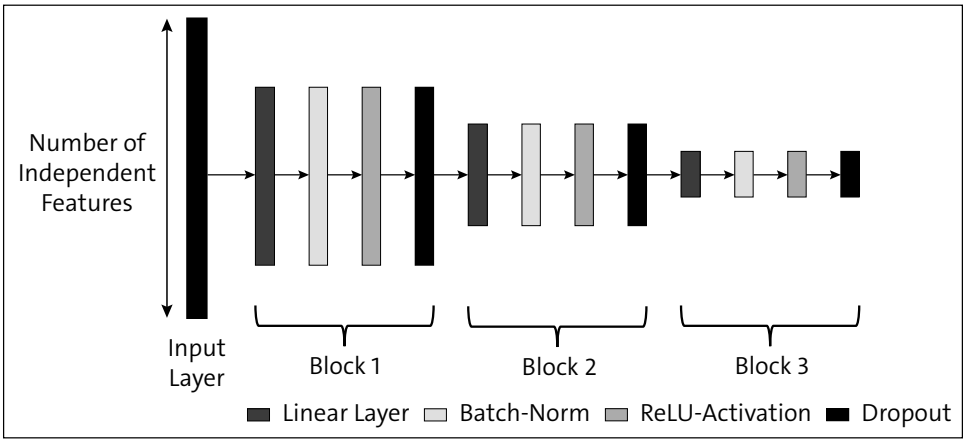


Figure 3.15 Multiclass Classification: Network Structure

The model should be flexibly usable with other datasets. For this reason, the number of `input_size` independent features, the number of nodes in the `hidden_size` hidden layer, and the number of `output_size` output nodes are transferred to the `MulticlassModel` model class in Listing 3.22. In addition, an optional parameter for the dropout rate (`dropout_rate`) is defined.

```
class MulticlassModel(nn.Module):
    def __init__(self, input_size, output_size, hidden_size,
        dropout_rate=0.3):
```

```
        super(MulticlassModel, self).__init__()
        self.linear_in = nn.Linear(input_size, hidden_size)
        self.bn1 = nn.BatchNorm1d(hidden_size)
        self.linear_hidden1 = nn.Linear(hidden_size, hidden_size // 2)
        self.bn2 = nn.BatchNorm1d(hidden_size // 2)
        self.linear_hidden2 = nn.Linear(hidden_size // 2, hidden_size // 4)
        self.bn3 = nn.BatchNorm1d(hidden_size // 4)
        self.linear_out = nn.Linear(hidden_size // 4, output_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        x = self.linear_in(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.linear_hidden1(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.linear_hidden2(x)
        x = self.bn3(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.linear_out(x)
        return x

input_size = X_train.shape[1]

model = MulticlassModel(input_size=input_size, output_size=OUTPUT_SIZE, hidden_
size=HIDDEN_SIZE).to(DEVICE)
```

Listing 3.22 Multiclass Classification: Model Class and Instance

`Adam` is used as the optimizer, and `CrossEntropyLoss` is used as the loss function, as shown in Listing 3.23.

```
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
loss_fn = nn.CrossEntropyLoss()
```

Listing 3.23 Multiclass Classification: Optimizer and Loss Function

The model training is shown in Listing 3.24. The training loop iterates over the previously defined number of `EPOCHS`, and in each epoch, the model is improved based on the training data and validated using the validation data. The `train_loss` and `val_loss` losses are determined so we can visualize them later.

```
#%% training loop
train_losses = []
val_losses = []
for epoch in range(EPOCHS):
    train_loss = 0
    val_loss = 0
    for X_train_batch, y_train_batch in train_loader:
        # move data to device
        X_train_batch, y_train_batch = X_train_batch.to(DEVICE),
        y_train_batch.to(DEVICE)
        # forward pass
        y_train_batch_pred = model(X_train_batch)
        # calculate loss
        loss = loss_fn(y_train_batch_pred, y_train_batch)
        # backward pass
        loss.backward()
        # update weights
        optimizer.step()
        # reset gradients
        optimizer.zero_grad()
        # update train loss
        train_loss += loss.item()
    # normalize and append train loss
    train_losses.append(train_loss / len(train_loader))
    print(f"Epoch {epoch+1}/{EPOCHS}, Loss: {train_losses[-1]:.4f}")
    for X_val_batch, y_val_batch in val_loader:
        # move data to device
        X_val_batch, y_val_batch = X_val_batch.to(DEVICE),
            y_val_batch.to(DEVICE)
        # forward pass
        y_val_batch_pred = model(X_val_batch)
        # calculate loss
        loss = loss_fn(y_val_batch_pred, y_val_batch)
        # update val loss
        val_loss += loss.item()
    # normalize and append val loss
    val_losses.append(val_loss / len(val_loader))
    print(f"Epoch {epoch+1}/{EPOCHS}, Val Loss: {val_losses[-1]:.4f}")
```

```
Epoch 1/50, Loss: 0.9680
Epoch 1/50, Val Loss: 0.7753
...
Epoch 50/50, Loss: 0.1326
Epoch 50/50, Val Loss: 0.1137
```

Listing 3.24 Multiclass Classification: Model Training

The losses will be output on the console, and they will continue to decrease with the number of epochs. In the next section, we'll take a closer look at the evaluation.

3.5.3 Evaluation

We can display the previously determined losses of the training and validation data in a diagram by using the code in Listing 3.25.

```
# %% visualize training and validationloss
plt.figure()
sns.lineplot(x=list(range(EPOCHS)), y=train_losses, label='Training Loss')
sns.lineplot(x=list(range(EPOCHS)), y=val_losses, label='Validation Loss')
plt.xticks(range(0, EPOCHS, 5))
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Over Time')
plt.legend()
```

Listing 3.25 Multiclass Classification: Loss Visualization

Figure 3.16 shows the progress of the model training and how the losses for the training and validation data decrease as the number of epochs increases. The training stops after 50 epochs because that's the previously defined total number of epochs. This number of epochs is also a sensible choice because there are hardly any improvements left to be made after that point.

Now, we want to create predictions for the test data as in Listing 3.26 (i.e., data that the model has never seen before) to check whether the model can really generalize well. To do this, we let the trained model create `y_test_batch_pred` predictions for the `X_test_batch` independent test data.

The predictions are referred to as *logits*, and they are the direct outputs of the last linear layer of a classification model before a normalizing activation function is applied. Their value range is between $-\infty$ and $+\infty$, and we can determine probabilities from these predictions by applying `torch.softmax`. These probabilities are normalized outputs of the logits after the application of softmax, and they have a value range from 0 to 1. This gives you the probabilities for the three `y_test_batch_pred_proba` classes.

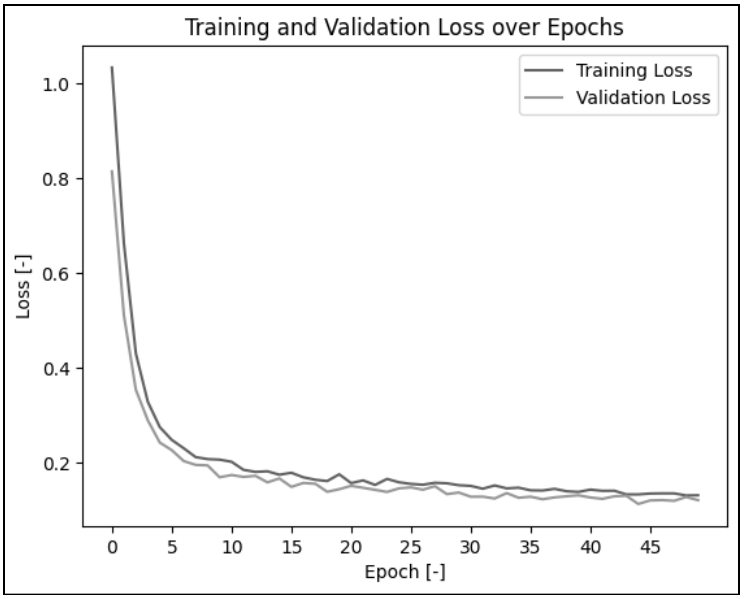


Figure 3.16 Multiclass Classification: Losses

To determine the class with the highest probability, we use `torch.argmax`. This allows us to determine the index of the corresponding element, and we receive a list with the indices of the predicted classes in the `y_test_batch_pred_class` object. The real classes are summarized in the `y_test_true`, and the predicted classes are summarized in the `y_test_pred_class` object. We'll use the lists in subsequent steps, for example, to determine the confusion matrix.

```
#%% create test predictions for multiclass classification
y_test_pred_proba, y_test_pred_class, y_test_true = [], [], []

model.eval()  # Set model to evaluation mode
with torch.no_grad():
    for X_test_batch, y_test_batch in test_loader:
        X_test_batch, y_test_batch = X_test_batch.to(DEVICE),
                                     y_test_batch.to(DEVICE)
        y_test_batch_pred = model(X_test_batch)

        # Apply softmax to get probabilities
        y_test_batch_pred_proba = torch.softmax(y_test_batch_pred, dim=1)
        y_test_batch_pred_proba = y_test_batch_pred_proba.cpu().numpy()

        # Get predicted classes (argmax)
        y_test_batch_pred_class = torch.argmax(y_test_batch_pred,
        dim=1).cpu().numpy()
        y_test_batch = y_test_batch.cpu().numpy()
```

```
    # Store predictions and true labels
    y_test_pred_proba.extend(y_test_batch_pred_proba.tolist())
    y_test_pred_class.extend(y_test_batch_pred_class.tolist())
    y_test_true.extend(y_test_batch.tolist())
```

Listing 3.26 Multiclass Classification: Create Predictions for Test Data

We can use the `confusion_matrix` function to directly determine the confusion matrix by passing the real and predicted values. We can then create a graphic from this table with the code in Listing 3.27.

```
#%% create confusion matrix
cm = confusion_matrix(y_true=y_test_true, y_pred=y_test_pred_class)
plt.figure()
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted class)
plt.ylabel('Actual class')
plt.title('Confusion matrix')
```

Listing 3.27 Multiclass Classification: Confusion Matrix Creation

The confusion matrix of the test data for our multiclass classification model is shown in Figure 3.17. The predicted classes are plotted horizontally, and the true classes are plotted vertically. The correct predictions are shown on the main diagonal, and outside the main diagonal are the incorrect predictions.

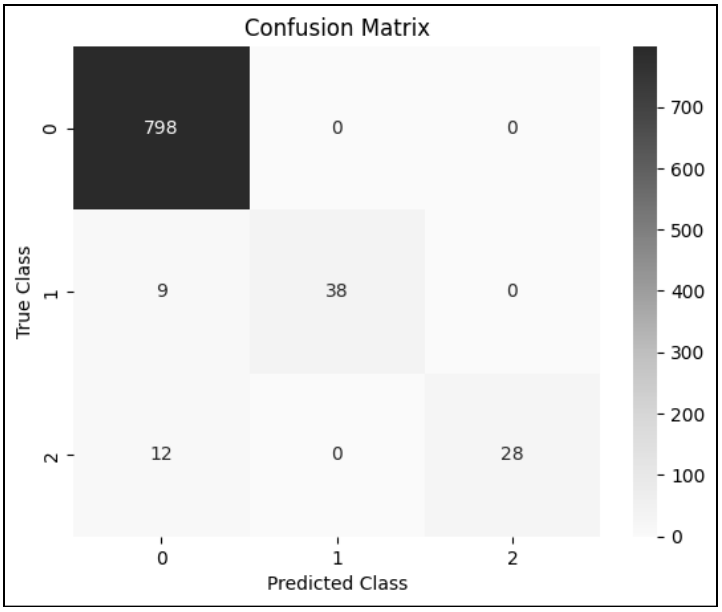


Figure 3.17 Multiclass Classification: Confusion Matrix

Basically, the result looks good because most of the values can be found in the diagonal. However, there are still a few misclassifications. For example, the real class 1 (`BotAttack`) was often incorrectly predicted as class 0 (`Normal`), meaning it was an FN. Often, we'll need to make a decision based on a single metric, so at this point, we can determine the accuracy of the model with the code in Listing 3.28.

```
#%% accuracy score
accuracy_score(y_true=y_test_true, y_pred=y_test_pred_class)
```

```
0.9740112994350283
```

Listing 3.28 Multiclass Classification: Model Accuracy Score

The model delivers the correct prediction in 97.4% of cases, and we must categorize this result because we know that the data is unbalanced (i.e., there are significantly more elements of one class than of the other classes). For this purpose, we'll determine the dummy classifier accuracy with the code in Listing 3.29, which reflects the results of a model based purely on guesswork.

```
#%% naive classifier and accuracy score
model_naive = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
y_test_pred_naive = model_naive.predict(X_test)
accuracy_score(y_true=y_test_true, y_pred=y_test_pred_naive)
```

```
0.9016949152542373
```

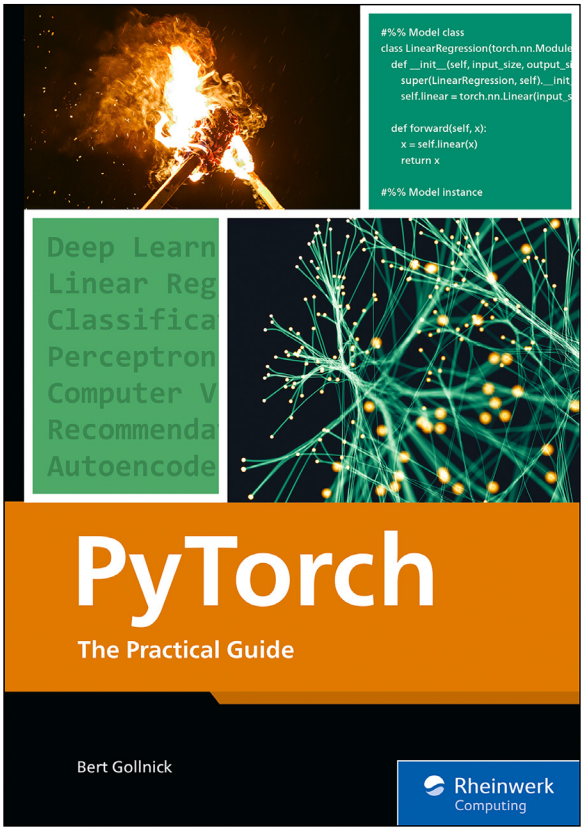
Listing 3.29 Multiclass Classification: Accuracy Score for Dummy Classifier

The dummy classifier has an accuracy of 90.2%, and with such high values, it makes sense to talk about error rather than accuracy. Accordingly, the dummy classifier has an error of just under 10%, whereas the model we trained only has an error of 2.6%.

We have now reached the end of our section on multiclass classifiers. We've learned that the number of output nodes changes relative to the binary classifier and that we must use a different loss model.

3.6 Summary

In this chapter, we've explored the classification models. In Section 3.1, we learned about the different types of classification: binary, multiclass, and multilabel classification. In Section 3.2, we dove into the evaluation of classification models and provided you with insights into how to create and interpret confusion matrices. Building on that, in Section 3.3, we learned about the ROC curve, which is suitable for comparing different models with one another.



Bert Gollnick

PyTorch

The Practical Guide

- Train, tune, and deploy deep learning models with PyTorch
- Implement models for linear regression, classification, computer vision, recommendation systems, and more
- Work with PyTorch Lightning, TensorBoard, LangChain, and FastAPI

rheinwerk-computing.com/6207

The Author

Bert Gollnick is a senior data scientist who specializes in renewable energy. For several years, he has taught courses about data science and machine learning, and more recently, about generative AI and natural language processing.

ISBN 978-1-4932-2786-0 • 416 pages • 02/2026

E-book: $54.99 • **Print book:** $59.95 • **Bundle:** $69.99

Rheinwerk Publishing