01-introduction-exploration

July 15, 2025

1 Disclaimer

Originally the data set was supposed to be used for the pracitcal exercise in the module Data Mining. After some first classification tasks were performed it became clear that this data set is not ideal for classification. Since the goal was to see good results and learn about data mining concepts this data set has not been further worked out or analyzed.

2 Introduction

The aim of this exercise is to learn about different data mining techniques and possibilities, however details of each algorithm are not part of the scope. As part of this exercise an analysis to explore patterns and potential correlations within a data set will be conducted. This constellation of multiple Jupyter Notebooks documents the data mining process for a train data set. The selected data set contains records for various train journeys since november 2023. The data has been created by manually documenting key statistics about every journey in a spreadsheet. The data set includes information about the train line, departure and arrival stations as well as date & time, delay and ticket inspections. The exact attributes will be analyzed later on. This data set has been chosen due to the personal connection based on manually creating the data over an extended period of time.

3 Data exploration

In this step a first idea of the data should be obtained. A good understanding of the attributes, their distributions and semantic meaning allows for better analysis further on. The data set consists of two related spread sheets: one for the train journeys and general statistics about these journeys, the other for annoucements during the train drive.

3.1 Train journeys

3.1.1 Attributes

The data set has a limited amount of attributes which can be explained either by their name or by annotations in the data set. The original names are in german but for language consistency the english translation will be used when talking about the attributes. The most important attributes are:

- **ID** (“Lfd. Nummer”) is an ID for every record
- **train line** (“Lininen Nummer”) is the train number
- **departure station** (“Start Haltestelle”) is the departure station

- **arrival station** (“Ziel Haltestelle”) is the arrival station
- **planned departure** (“Planmäßige Abfahrtszeit”) and date (“Datum”) are the time and date of depature

These attributes are normally always known before the journey is even taken. They will be referenced as **fixed information** of a journey. The following attributes are determined during and after the journey has been taken and referenced as **variable information**:

- **delay** “Verspätung in min” the delay upon arrival in minutes, however delays are only documented if they are 2 min or longer. Arrival counts as the moment the first passengers step out of the vehicle.
- **ticket checked** (“Kontrolliert”) if there was a ticket inspection. For context in german trains there are commonly no gates to force you to buy a ticket but random checks are performed by ticket inspectors in the trian.
- **platform changed** (“Gleis verlegt”) if the platform (departure or arrival) was different from the planned platform
- **crowdedness** (“Fülle des Zuges”) describes how full the train was
- **train model** (“Zugmodel”) is only documented for specific trains which can be classified as old or new
- **cleanliness** (“Sauberkeit”) takes into consideration if there is trash, if floor or seats are dirty and any smell is in the train

Additionally there are three attributes which exist because the delay of one train does not always reflect the true experience of train driving. For example, if train A has a delay of 15 min but due to that the connecting train B is missed, the traveler might have to wait an additional 15 min for the next train. Therefore the total delay would be 30 min. The exact calculation is the following: *When planning the trip, the connection to be taken is determined. If due to train delays a different connection is taken during the whole trip the relative timings become relevant. External factors when traveling to the departure station (e. g. delayed bus, traffic jam, …) do not affect this calculation.*

- **relative planned departure** (“rel. planmäßige Abfahrtszeit”) the planned departure time, different from the actual departure time
- **relative delay** (“rel. Verspätung”) the delay compared to when one would have arrived at the arrival station when using the planned connection
- **alternative connection** (“Alternativer Anschluss”) if the relative delay is due to missing connecting train

The attribute note (“Bemerkung”) is more a personal note and will not be taken into consideration for this analysis.

The following table shows the types of attributes:

Attribute Name	Attribute Type	Possible values	Annotation
ID	nominal	natrual numbers (0 excluded)	
train line	nominal	any	
departure station	nominal	any	

Attribute Name	Attribute Type	Possible values	Annotation
arrival station	nominal	any	
planned departure	interval	time	
date	interval	calendar date	
delay	ratio	{null, natrual numbers >1 or 0; X; N}	[1]
ticket checked	nominal	true/false	
platform changed	nominal	true/false	
crowdedness	ordinal	{nearly empty, light crowd, moderate, heavy crowd, packed} ({“(Fast) Leer”, “Wenig voll”, “Normal voll”, “Sehr voll”, “Zu voll”})	[2]
train model	nominal	{old, new, other}, ({“Alt”, “Neu”, “Sonstiges”})	[3]
cleanliness	ordinal	{dirty, alright, very clean}, ({“Dreckig”, “Ok”, “Sehr sauber”})	
relative planned departure	interval	{time; X}	[4]
relative delay	ratio	natrual numbers (0 included)	[5]
alternative connection	nominal	ture/false	

[1] *null* = 0 (a delay of 0 or 1 is documented by not entering a delay), X = train canceled and no relative delay on other connection; N = decided not to take the train due to overcrowding, apparent delays, etc. [2] *nearly empty* = free choice of seats; *light crowd* = people in the train, lots of places including 4-seaters available; *moderate* = 2-seaters are available, maybe a bit of searching; *heavy crowd* = only sitting next to someone is possible; *packed* = even if all seats would be used people would have to stand [3] *other* = a new and old train compartment are connected or temporary model [4] X = when a departure is from a station not initially planned and therefore time can not be determined [5] due to the calculation this delay can also be 0 or 1

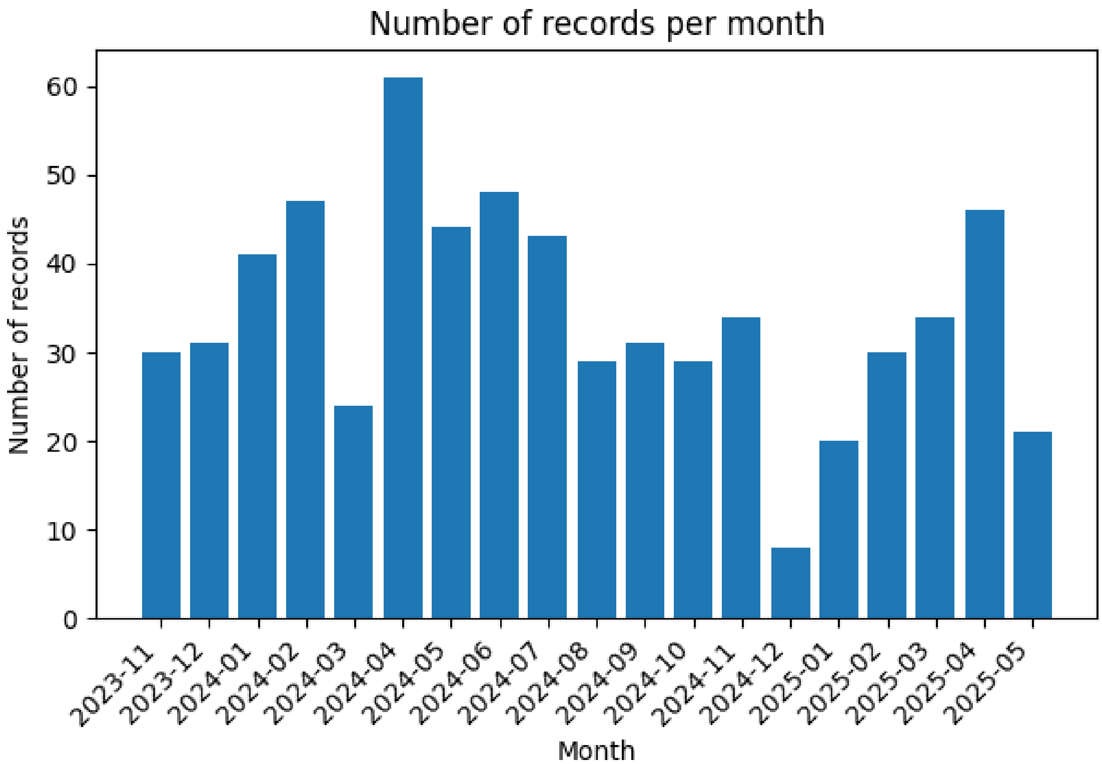
3.1.2 Sample data

The data set does not appear to have a lot of missing values however the last entries are all completely empty except the ID. As an example this is the record with ID 16: `16,RS30,Oldenburg,Bremen,07:05,24.11.2023,4,FALSE,FALSE,Sehr voll,Neu,,,,FALSE,,,,` Without further analysis, it is apparent that a lot of entries contain “Rastede”, “Oldenburg” and “Bremen” as departure or arrival stations. A record which has a relative delay is quite rare. Most records have a normal or no delay at all.

3.1.3 Visualizations (TODO show by week days and months)

[1]:
```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# The first three lines are invalid -> skip
journeys = pd.read_csv('../data/train-drives.csv', skiprows=3, encoding='utf-8')

# Convert date to datetime, coerce errors to NaT
journeys['Datum'] = pd.to_datetime(journeys['Datum'], format='%d.%m.%Y',␣
 ↪errors='coerce')

# Drop rows with invalid or missing dates
journeys = journeys.dropna(subset=['Datum'])

# Group by year and month
months = journeys['Datum'].dt.strftime('%Y-%m')
month_counts = months.value_counts().sort_index()

# Prepare sorted months and counts for plotting
sorted_months = month_counts.index.tolist()
counts = month_counts.values.tolist()
```

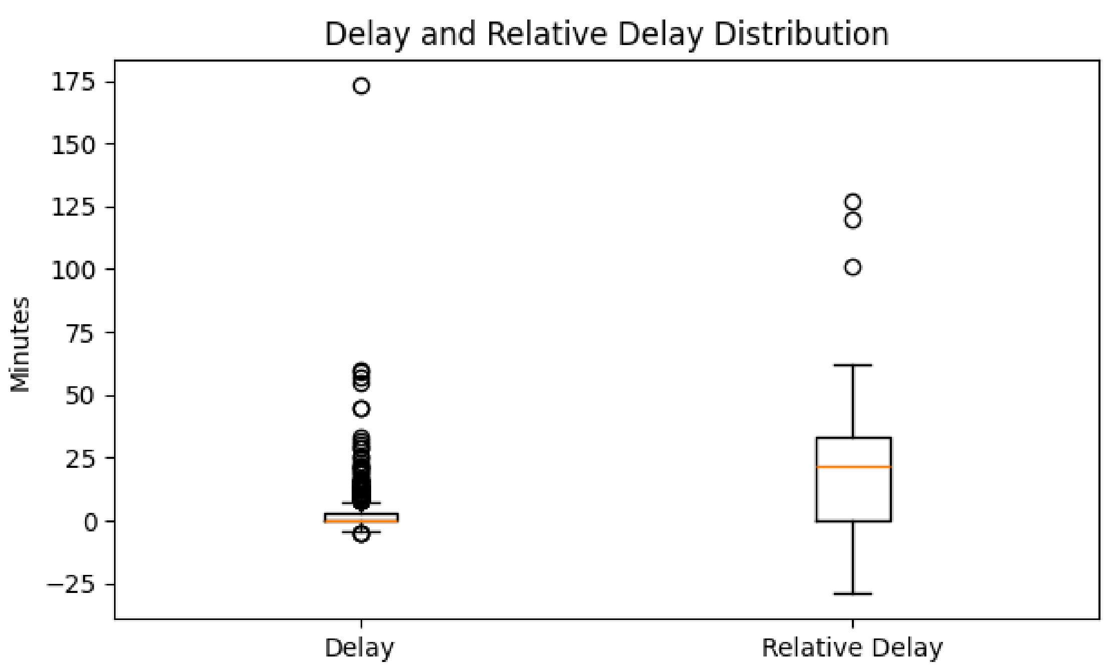
[2]:
```python
# Create a histogram of the number of records per month
plt.figure(figsize=(7, 4))
plt.bar(sorted_months, counts)
plt.xlabel('Month')
plt.ylabel('Number of records')
plt.title('Number of records per month')
plt.xticks(rotation=45, ha='right')
plt.show()
```

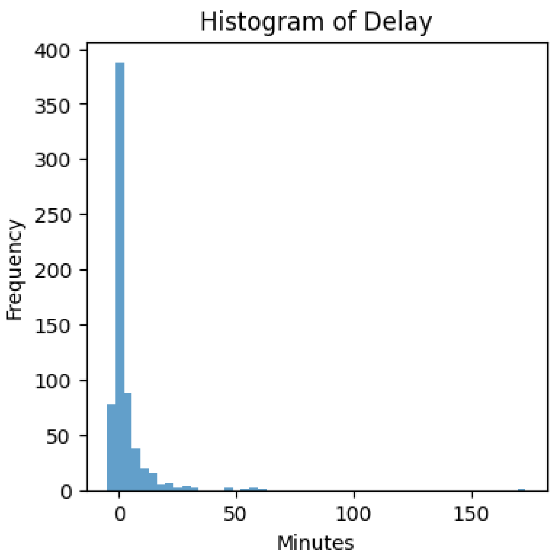


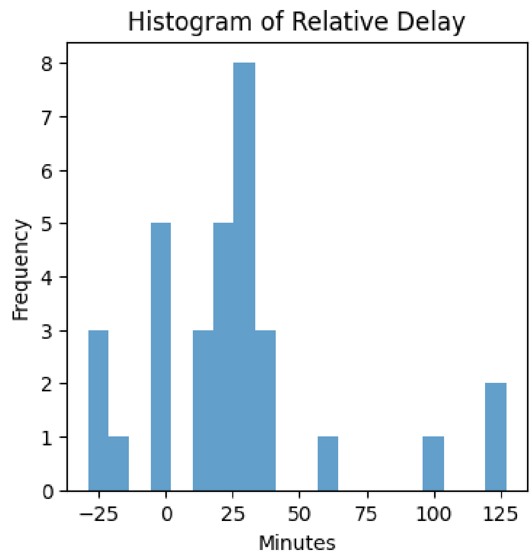
The records are quite regularly distributed with most months having at least 20 records. December 2024 has less than 10 records, which could indicate an anomaly in the data or just fewer journeys taken in that month.

[3]:
```
# Convert columns to numeric, convert empty values to 0
delay = pd.to_numeric(journeys['Verspätung in min'], errors='coerce')
rel_delay = pd.to_numeric(journeys['rel. Verspätung'], errors='coerce')

delay_filled = delay.fillna(0)

# Create box plots
plt.figure(figsize=(7, 4))
plt.boxplot([delay_filled.dropna(), rel_delay.dropna()], tick_labels=['Delay',␣
 ↪'Relative Delay'])
plt.ylabel('Minutes')
plt.title('Delay and Relative Delay Distribution')
plt.show()

# Create historgram
plt.figure(figsize=(9, 4))
plt.subplot(1, 2, 1)
plt.hist(delay_filled, bins=50, alpha=0.7, label='Delay')
```

```
plt.xlabel('Minutes')
plt.ylabel('Frequency')
plt.title('Histogram of Delay')
plt.subplot(1, 2, 2)
plt.hist(rel_delay, bins=20, alpha=0.7, label='Relative Delay')
plt.xlabel('Minutes')
plt.ylabel('Frequency')
plt.title('Histogram of Relative Delay')
plt.show()


# Count occurrences of 'X' and 'N' in delay columns
delay_x_count = journeys['Verspätung in min'].astype(str).str.upper().eq('X').
    ↪sum()
delay_n_count = journeys['Verspätung in min'].astype(str).str.upper().eq('N').
    ↪sum()
rel_delay_x_count = journeys['rel. Verspätung'].astype(str).str.upper().eq('X').
    ↪sum()

print('-- Statistics about Delay --')
print(delay_filled.describe())
print(f"'X' in Delay: {delay_x_count}; 'N' in Dealy: {delay_n_count}\n")
print('-- Statistics about Relative Delay --')
print(rel_delay.describe())
print(f"'X' in Relative Delay: {rel_delay_x_count}")
```



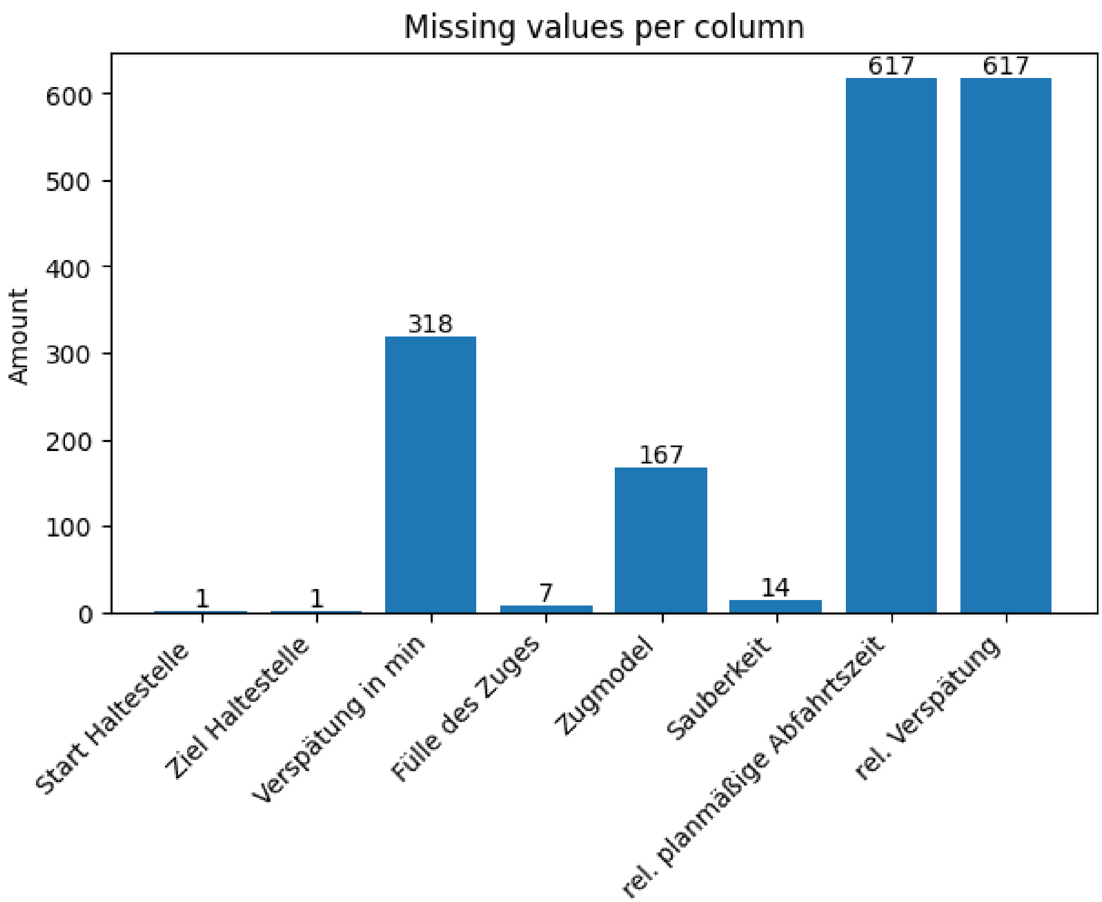





```
-- Statistics about Delay --
count    651.000000
mean       2.849462
std        9.762979
min       -5.000000
25%        0.000000
50%        0.000000
75%        3.000000
max      173.000000
Name: Verspätung in min, dtype: float64
'X' in Delay: 5; 'N' in Dealy: 4

-- Statistics about Relative Delay --
count     32.000000
mean      25.406250
std       36.036594
min      -29.000000
25%       -0.250000
50%       22.000000
75%       33.000000
max      127.000000
Name: rel. Verspätung, dtype: float64
'X' in Relative Delay: 2
```

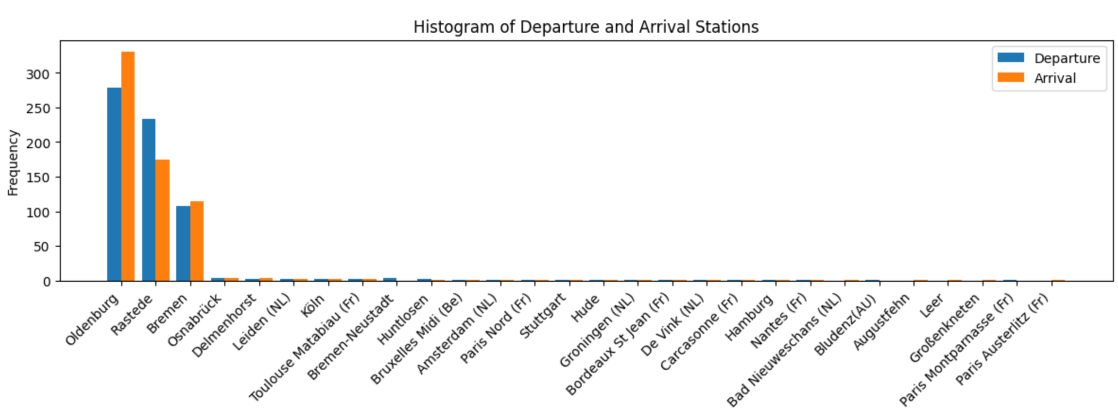
The statistics are not 100% accurate since X and N were converted to 0 for the delays, however there are not many records with an X or N. The boxplot shows quite clearly that the majority of delays are very low and mostly 0 since the 50% mark is at 0 minutes. Noteworthy is the outlier and maximum of 173 minutes delay. The relative delays are more spread out and the percentiles have a larger range.

[4]:
```
# Count missing values for each column, ignoring the last 4 columns (these are␣
↪fully empty/not relevant)
cols = journeys.columns[:-4]
missing_counts = journeys[cols].isna().sum() + (journeys[cols] == '').sum()
missing_counts = missing_counts[missing_counts > 0] # remove empty columns

# Create bar chart
plt.figure(figsize=(7, 4))
bars = plt.bar(missing_counts.index, missing_counts.to_numpy())
plt.bar_label(bars)
plt.ylabel('Amount')
plt.title('Missing values per column')
plt.xticks(rotation=45, ha='right')
plt.show()
```



The bar chart shows the missing values for all columns which have any missing values. Relative planned departure and relative delay have the most missing values. This is expected as relative delays are not expected to occur regularly and will be left empty when not relevant. The delay has also a lot of missing values since these are left empty when delay is 0. Similar for the train model which is not documented for every train. Notable are the couple missing values for departure station, arrival station, crowdedness and cleanliness. These will have to be taken care of during preprocessing or analysis.

[5]:
```
# Get value counts, drop missing values
departures = journeys['Start Haltestelle '].dropna().value_counts()
arrivals = journeys['Ziel Haltestelle'].dropna().value_counts()

# Get all unique stations
all_stations = set(departures.index).union(arrivals.index)

# Sort stations by combined counts (descending)
sorted_stations = departures.add(arrivals, fill_value=0).
↪sort_values(ascending=False).index.tolist()

# Get separate counts for all stations
departures_counts = [departures.get(station, 0) for station in sorted_stations]
arrivals_counts = [arrivals.get(station, 0) for station in sorted_stations]


# Create bar chart for departure and arrival
x = np.arange(len(sorted_stations))
width = 0.4

plt.figure(figsize=(12, 5))
plt.bar(x - width/2, departures_counts, width, label='Departure')
plt.bar(x + width/2, arrivals_counts, width, label='Arrival')
plt.xticks(x, sorted_stations, rotation=90)
plt.xlabel('Station')
plt.ylabel('Frequency')
plt.title('Histogram of Departure and Arrival Stations')
plt.legend()
plt.tight_layout()
plt.xticks(rotation=45, ha='right')
plt.show()
```



As discussed in the “Sample Data” section, most journeys either start or end in “Oldenburg”, “Rastede” or “Bremen”. The histogram does visualize this quite clearly. Most other stations have very few records in the data set and might not supply sufficient data for a deeper analysis.

3.2 Train announcements

TDB

4 Analysis Objectives

TODO: Need to be more concrete and with reasoning

Predictions and classifications could also help improve future decisions about which train connection to take for a minimal risk of delay.

The primary questions for this analysis are:

1. Is there a correlation between the train line and other journey statistics? This includes examining whether certain lines tend to have more delay or higher crowding.
2. Can journeys be classified based on key features? Using classification techniques it will be attempted to predict if a train is for example ‘likely to be crowded’.
3. Is the likelihood of ticket inspections dependent on the time of day or month?

02-preprocessing

July 15, 2025

1 Data preprocessing

The data exploration step in the previous notebook already showed that there are some missing values in the data set and some which are not in a good format for furhter analysis. Since the attributes of the data set are well known it is possible to filter out semantic inconsistencies. There might for example be records with the same departure and arrival station.

The goal of this step is to handle missing values, transform some values and export a clean data set which can be used for further analysis.

1.1 Renaming columns

Firstly the columns will be renamed to have english labels as defined in the attribute description.

[1]:
```python
import pandas as pd

# The first three lines are invalid -> skip
journeys = pd.read_csv('../data/train-drives.csv', skiprows=3, encoding='utf-8')

# Rename columns, optional
journeys.rename(columns={
    'Lfd. Nummer': 'ID',
    'Linien Nummer': 'train_line',
    'Start Haltestelle ': 'departure_station',
    'Ziel Haltestelle': 'arrival_station',
    'Planmäßige Abfahrtszeit': 'planned_departure',
    'Datum': 'date',
    'Verspätung in min': 'delay',
    'Kontrolliert': 'ticket_checked',
    'Gleis verlegt': 'platform_changed',
    'Fülle des Zuges': 'crowdedness',
    'Zugmodel': 'train_model',
    'Sauberkeit': 'cleanliness',
    'rel. planmäßige Abfahrtszeit': 'relative_planned_departure',
    'rel. Verspätung': 'relative_delay',
    'Alternativer Anschluss': 'alternative_connection',
}, inplace=True)

# Drop unamed columns
```

```
journeys = journeys.loc[:, ~journeys.columns.str.contains('^Unnamed')]
# Drop unnecessary columns
journeys.drop(columns=['ID', 'Bemerkung', 'Definition Verspätung: '],␣
 ↪inplace=True)

# Calculate actual length for later evaluation
org_len = len(journeys)
non_boolean_cols = ['train_line', 'departure_station', 'arrival_station',␣
 ↪'planned_departure',
                    'date', 'delay', 'crowdedness', 'train_model',␣
 ↪'cleanliness',
                    'relative_planned_departure', 'relative_delay']
journeys.dropna(how='all', subset=non_boolean_cols, inplace=True)
non_empty_len = len(journeys)

journeys.head(5)
```

[1]:

```
  train_line departure_station arrival_station planned_departure        date  \
0         IC           Bremen       Oldenburg             11:53  20.11.2023
1       RE18         Oldenburg         Rastede             12:36  20.11.2023
2       RE18           Rastede       Oldenburg             06:41  21.11.2023
3       RS30         Oldenburg          Bremen             07:05  21.11.2023
4       RB58   Bremen-Neustadt     Delmenhorst             13:23  21.11.2023

  delay  ticket_checked  platform_changed  crowdedness train_model  \
0   NaN           False             False   Wenig voll         NaN
1     3           False              True   Wenig voll         Neu
2   NaN           False             False  Normal voll         Alt
3     4           False              True      Zu voll         Alt
4   NaN            True             False    Sehr voll         Neu

   cleanliness relative_planned_departure relative_delay  \
0           Ok                        NaN            NaN
1           Ok                        NaN            NaN
2  Sehr Sauber                        NaN            NaN
3          NaN                        NaN            NaN
4           Ok                        NaN            NaN

   alternative_connection
0                   False
1                   False
2                   False
3                   False
4                   False
```

[2]:

```
# Remove rows with uninteresting values
```

```python
# Drop all records with rare departure or arrival stations
rastede_bremen_stations = ['Rastede', 'Oldenburg', 'Hude', 'Delmenhorst',␣
↪'Bremen', 'Bremen-Neustadt']
journeys = journeys[
    journeys['departure_station'].isin(rastede_bremen_stations)
    & journeys['arrival_station'].isin(rastede_bremen_stations)
]

# Remove all records where delay is 'X' or 'N', case-insensitive
journeys = journeys[
    ~journeys['delay'].astype(str).str.upper().isin(['X', 'N'])
]
```

[3]:
```python
# Data conversion and encoding

# One-Hot encoding for train line
journeys = pd.get_dummies(journeys, columns=['train_line'], prefix='train_line')

# Remove invalid times for planned departure
journeys['planned_departure'] = pd.to_datetime(journeys['planned_departure'],␣
↪format='%H:%M', errors='coerce')
journeys = journeys.dropna(subset=['planned_departure'])
# Convert time into an hour column and group by 2 hour blocks
journeys['planned_departure_hour'] = journeys['planned_departure'].dt.hour
journeys['planned_departure_hour'] = (journeys['planned_departure_hour'] // 2)␣
↪* 2

# Convert date to datetime and remove invalid dates
journeys['date'] = pd.to_datetime(journeys['date'], format='%d.%m.%Y',␣
↪errors='coerce')
journeys.dropna(subset=['date'], inplace=True)

# Extract day of the week (Monday=1, Sunday=7) and month (1-12)
journeys['day_of_week'] = journeys['date'].dt.isocalendar().day
journeys['month'] = journeys['date'].dt.month

# Convert delay to integer
delay_numeric = pd.to_numeric(journeys['delay'], errors='coerce')
journeys['delay'] = delay_numeric.fillna(0).astype(int)

# Apply binary encoding for ticket_checked
journeys['ticket_checked'] = journeys['ticket_checked'].astype(bool).astype(int)

# Encode crowdedness with ordinal encoding
journeys = journeys[
    # Remove empty values
    journeys['crowdedness'].astype('str').str.strip() != ''
```

```
]
crowdedness_mapping = {
    '(Fast) Leer': 0,
    'Wenig voll': 1,
    'Normal voll': 2,
    'Sehr voll': 3,
    'Zu voll': 4
}
journeys['crowdedness'] = pd.to_numeric(journeys['crowdedness'].
 ↪map(crowdedness_mapping), errors='raise', downcast='integer')
```

[4]:

```
# Compare org_len to current length
new_len = len(journeys)
print(f'Original length (with empty records): {org_len}')
print(f'Original length (without empty records): {non_empty_len}, New length:␣
 ↪{new_len}, Removed: {non_empty_len - new_len}')

# Save the cleaned data to a new CSV file
journeys.to_csv('../data/train-drives-cleaned.csv', index=False,␣
 ↪encoding='utf-8')
journeys.head(5)
```

```
Original length (with empty records): 997
Original length (without empty records): 653, New length: 612, Removed: 41
```

[4]:

```
  departure_station arrival_station   planned_departure       date  delay  \
0            Bremen       Oldenburg 1900-01-01 11:53:00 2023-11-20      0
1         Oldenburg         Rastede 1900-01-01 12:36:00 2023-11-20      3
2           Rastede       Oldenburg 1900-01-01 06:41:00 2023-11-21      0
3         Oldenburg          Bremen 1900-01-01 07:05:00 2023-11-21      4
4   Bremen-Neustadt     Delmenhorst 1900-01-01 13:23:00 2023-11-21      0

   ticket_checked  platform_changed  crowdedness train_model  cleanliness  \
0               0             False            1         NaN           Ok
1               0              True            1         Neu           Ok
2               0             False            2         Alt  Sehr Sauber
3               0              True            4         Alt          NaN
4               1             False            3         Neu           Ok

   … alternative_connection train_line_IC  train_line_RB58  train_line_RE1  \
0  …                  False          True            False           False
1  …                  False         False            False           False
2  …                  False         False            False           False
3  …                  False         False            False           False
4  …                  False         False             True           False

   train_line_RE18  train_line_RS30  train_line_RS_3  planned_departure_hour  \
```

```
0           False            False            False                         10
1            True            False            False                         12
2            True            False            False                          6
3           False             True            False                          6
4           False            False            False                         12

   day_of_week  month
0            1     11
1            1     11
2            2     11
3            2     11
4            2     11

[5 rows x 22 columns]
```

03-classification

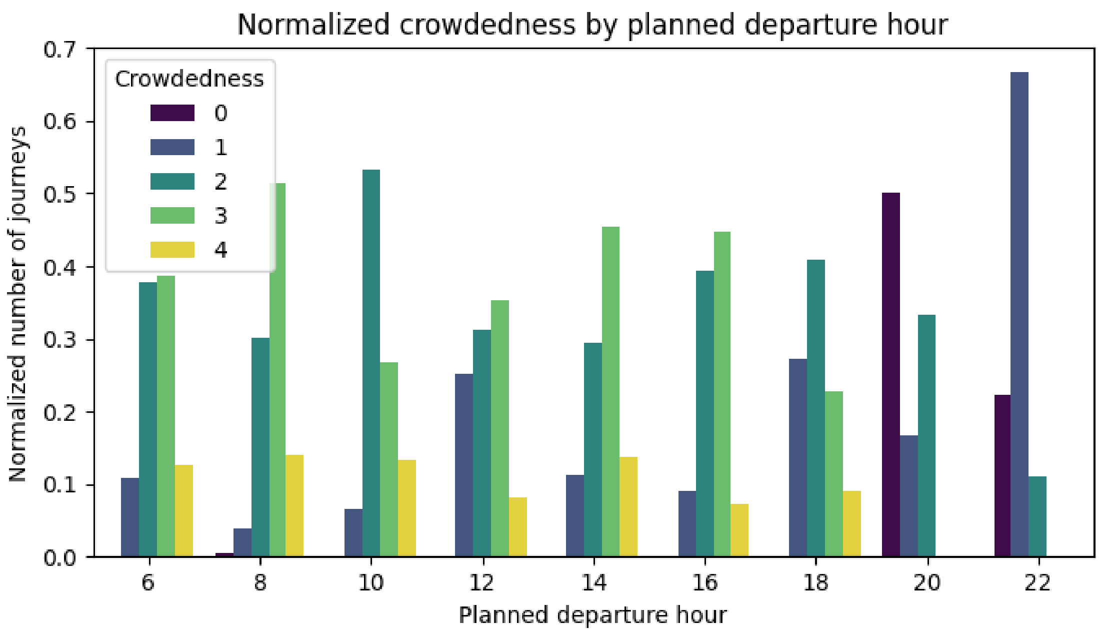
July 15, 2025

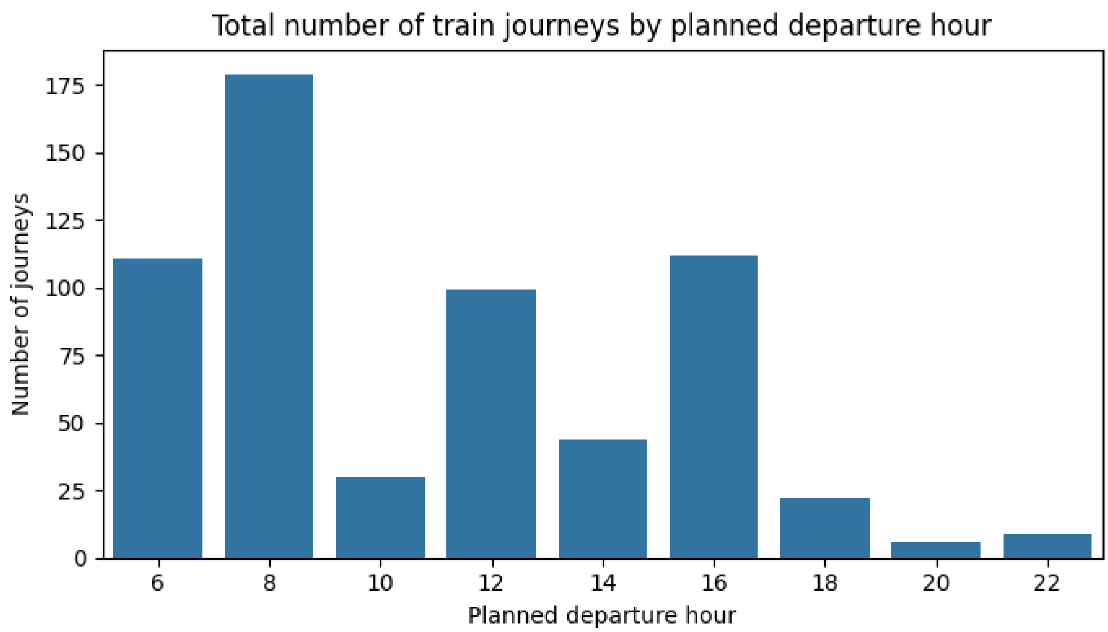
[1]:
```python
import pandas as pd
from sklearn.model_selection import train_test_split

# Load cleaned data from preprocessing step
journeys = pd.read_csv('../data/train-drives-cleaned.csv', encoding='utf-8')

# Select only relevant columns
journeys_filtered = journeys.loc[:, journeys.columns.str.
 ↪startswith('train_line') | journeys.columns.isin(['planned_departure_hour',␣
 ↪'day_of_week'])]
class_label_df = journeys['crowdedness']

# Split into train and test sets
journeys_train, journeys_test, class_label_train, class_label_test =␣
 ↪train_test_split(journeys_filtered, class_label_df, test_size=0.2,␣
 ↪random_state=123)
journeys_filtered.head()
```

[1]:
```
   train_line_IC  train_line_RB58  train_line_RE1  train_line_RE18  \
0           True            False           False            False
1          False            False           False             True
2          False            False           False             True
3          False            False           False            False
4          False             True           False            False

   train_line_RS30  train_line_RS_3  planned_departure_hour  day_of_week
0            False            False                      10            1
1            False            False                      12            1
2            False            False                       6            2
3             True            False                       6            2
4            False            False                      12            2
```

[2]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

# == Plot the crowdedness by planned departure time ==

# Group by hour and crowdedness, count occurrences
```

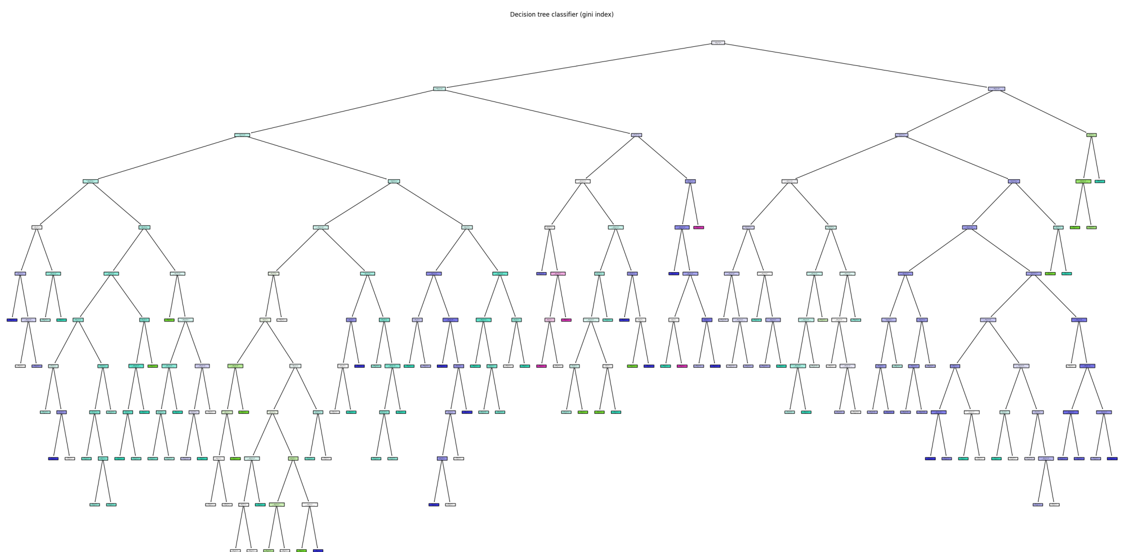
```python
journeys_visual = journeys.groupby(['planned_departure_hour', 'crowdedness']).
 ↪size().reset_index(name='count')

# Normalize the counts per hour
journeys_visual['normalized_count'] = journeys_visual.
 ↪groupby('planned_departure_hour')['count'].transform(lambda x: x / x.sum())

plt.figure(figsize=(7, 8))
plt.subplot(2, 1, 1)
sns.barplot(data=journeys_visual, x='planned_departure_hour',␣
 ↪y='normalized_count', hue='crowdedness', palette='viridis')
plt.title('Normalized crowdedness by planned departure hour')
plt.xlabel('Planned departure hour')
plt.ylabel('Normalized number of journeys')
plt.legend(title='Crowdedness', loc='upper left')
plt.tight_layout()


# == Plot the total number of journeys by planned departure time ==
journeys_total = journeys.groupby('planned_departure_hour').size().
 ↪reset_index(name='total_count')
plt.subplot(2, 1, 2)
sns.barplot(data=journeys_total, x='planned_departure_hour', y='total_count')
plt.title('Total number of train journeys by planned departure hour')
plt.xlabel('Planned departure hour')
plt.ylabel('Number of journeys')
plt.tight_layout()

plt.show()
```





[3]:
```python
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, classification_report

## Create decision tree with gini index
class_labels = class_label_df.unique().astype(str)
decision_tree_model = DecisionTreeClassifier(criterion='gini', random_state=123)
decision_tree_model.fit(journeys_train, class_label_train)
```

```python
# Plot resulting tree
plt.figure(figsize=(40, 20))
plot_tree(decision_tree_model, filled=True, feature_names=journeys_train.
 ↪columns, class_names=class_labels, rounded=False)
plt.title('Decision tree classifier (gini index)')
plt.show()

# Evaluate model
predictions = decision_tree_model.predict(journeys_test)
accuracy = accuracy_score(class_label_test, predictions)
print(f'Accuracy of gini index decision tree: {accuracy:.2f}')
print(classification_report(class_label_test, predictions,␣
 ↪target_names=class_labels, zero_division=0))
```

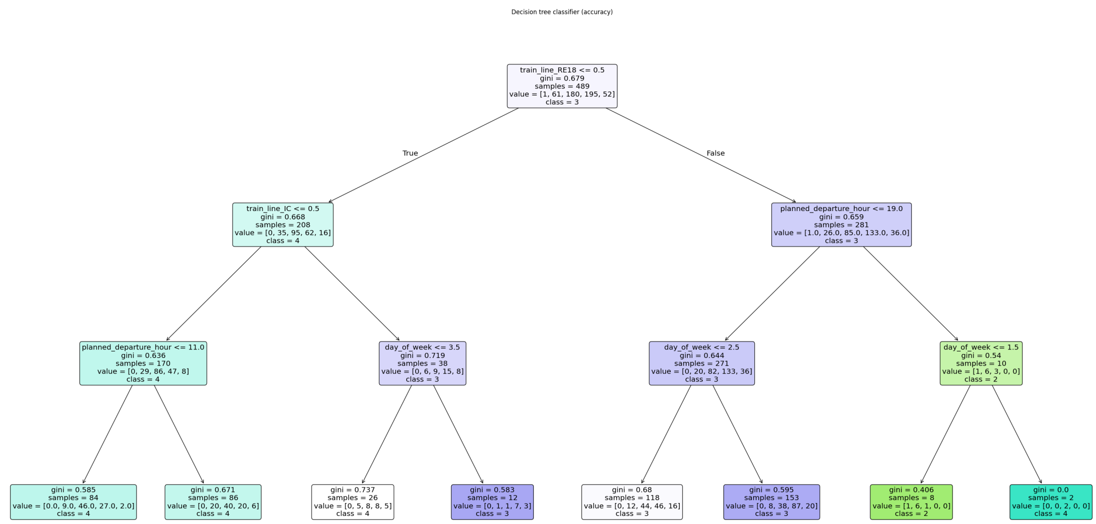



```
Accuracy of gini index decision tree: 0.44
              precision    recall  f1-score   support

           1       0.00      0.00      0.00         5
           2       0.24      0.38      0.29        13
           4       0.30      0.34      0.32        32
           3       0.58      0.66      0.62        58
           0       0.00      0.00      0.00        15

    accuracy                           0.44       123
   macro avg       0.22      0.28      0.25       123
weighted avg       0.38      0.44      0.41       123
```

[4]:
```python
from sklearn.model_selection import GridSearchCV

# Possible parameters for hyperparameter tuning
# We want to limit the size of the tree to avoid overfitting
param_grid = {
    'max_depth': [2, 3, 4, 5],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

decision_tree_model = DecisionTreeClassifier(random_state=123)

# Search for best parameters and train model
grid_search = GridSearchCV(estimator=decision_tree_model,
                           param_grid=param_grid,
                           cv=3,                # 5-fold cross-validation
                           scoring='accuracy',
                           n_jobs=-1)           # Use all CPU cores
grid_search.fit(journeys_train, class_label_train)

print("Best parameters found: ", grid_search.best_params_)
print("Best cross-validation score: ", grid_search.best_score_)

# Plot resulting tree
plt.figure(figsize=(40, 20))
plot_tree(grid_search.best_estimator_, filled=True,␣
 ↪feature_names=journeys_train.columns, class_names=class_labels, rounded=True)
plt.title('Decision tree classifier (accuracy)')
plt.show()

# Evaluate model
predictions = grid_search.best_estimator_.predict(journeys_test)
accuracy = accuracy_score(class_label_test, predictions)
print(f'Accuracy of gini index decision tree: {accuracy:.2f}')
print(classification_report(class_label_test, predictions,␣
 ↪target_names=class_labels, zero_division=0))
```

```
c:\git\hsb\datamining\train-data-mining\.venv\Lib\site-
packages\sklearn\model_selection\_split.py:805: UserWarning: The least populated
class in y has only 1 members, which is less than n_splits=3.
  warnings.warn(
```

```
Best parameters found:  {'max_depth': 3, 'min_samples_leaf': 1,
'min_samples_split': 2}
Best cross-validation score:  0.4764826175869121
```




```
Accuracy of gini index decision tree: 0.48
              precision    recall  f1-score   support

           1       0.00      0.00      0.00         5
           2       0.33      0.08      0.12        13
           4       0.37      0.66      0.47        32
           3       0.59      0.64      0.61        58
           0       0.00      0.00      0.00        15

    accuracy                           0.48       123
   macro avg       0.26      0.27      0.24       123
weighted avg       0.41      0.48      0.42       123
```

[5]:

```python
from imblearn.over_sampling import SMOTE
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler

# Scale the planned departure and day of week to make them comparable
cols_to_scale = ['planned_departure_hour', 'day_of_week']
journeys_train_scaled = journeys_train.copy()
journeys_test_scaled = journeys_test.copy()

scaler = StandardScaler()
journeys_train_scaled[cols_to_scale] = scaler.
 ↪fit_transform(journeys_train[cols_to_scale])
```

```
journeys_test_scaled[cols_to_scale] = scaler.
  ↪transform(journeys_test[cols_to_scale])


# Use SMOTE to generate synthetic samples for minority class
smote = SMOTE(random_state=123, k_neighbors=3)
journeys_train_balanced, class_label_train_balanced = smote.
  ↪fit_resample(journeys_train_scaled, class_label_train)

# Use cross-validation to find best k
k_range = range(1, 21)
cv_scores = []

for k in k_range:
    knn_classifier = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn_classifier, journeys_train_balanced,␣
  ↪class_label_train_balanced, cv=3, scoring='accuracy')
    cv_scores.append(scores.mean())


best_k = k_range[cv_scores.index(max(cv_scores))]
print(f'Best k found by cross-validation: {best_k}')

# Train model again with bset k
knn_classifier = KNeighborsClassifier(n_neighbors=3)
knn_classifier.fit(journeys_train_balanced, class_label_train_balanced)

# Evaluate the KNN model
knn_predictions = knn_classifier.predict(journeys_test_scaled)
knn_accuracy = accuracy_score(class_label_test, knn_predictions)
print(f'Accuracy of KNN Classifier: {knn_accuracy:.3f}')
print(classification_report(class_label_test, knn_predictions,␣
  ↪target_names=class_labels, zero_division=0))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 18
     16 # Use SMOTE to generate synthetic samples for minority class
     17 smote = SMOTE(random_state=123, k_neighbors=3)
---> 18 journeys_train_balanced, class_label_train_balanced =␣
  ↪smote.fit_resample(journeys_train_scaled, class_label_train)
     20 # Use cross-validation to find best k
     21 k_range = range(1, 21)

File c:\git\hsb\datamining\train-data-mining\.
  ↪venv\Lib\site-packages\imblearn\base.py:202, in BaseSampler.fit_resample(self,␣
  ↪X, y, **params)
```

```
    181 def fit_resample(self, X, y, **params):
    182     """Resample the dataset.
    183 
    184     Parameters
   (…)    200         The corresponding label of `X_resampled`.
    201     """
--> 202     return super().fit_resample(X, y, **params)

File c:\git\hsb\datamining\train-data-mining\.venv\Lib\site-packages\sklearn\base.py:1389, in _fit_context.<locals>.decorator.<locals>.wrapper(estimator, *args, **kwargs)
   1382     estimator._validate_params()
   1384 with config_context(
   1385     skip_parameter_validation=(
   1386         prefer_skip_nested_validation or global_skip_validation
   1387     )
   1388 ):
-> 1389     return fit_method(estimator, *args, **kwargs)

File c:\git\hsb\datamining\train-data-mining\.venv\Lib\site-packages\imblearn\base.py:105, in SamplerMixin.fit_resample(self, X, y, **params)
     99 X, y, binarize_y = self._check_X_y(X, y)
    101 self.sampling_strategy_ = check_sampling_strategy(
    102     self.sampling_strategy, y, self._sampling_type
    103 )
--> 105 output = self._fit_resample(X, y, **params)
    107 y_ = (
    108     label_binarize(output[1], classes=np.unique(y)) if binarize_y else output[1]
    109 )
    111 X_, y_ = arrays_transformer.transform(output[0], y_)

File c:\git\hsb\datamining\train-data-mining\.venv\Lib\site-packages\imblearn\over_sampling\_smote\base.py:359, in SMOTE._fit_resample(self, X, y)
    356 X_class = _safe_indexing(X, target_class_indices)
    358 self.nn_k_.fit(X_class)
--> 359 nns = self.nn_k_.kneighbors(X_class, return_distance=False)[:, 1:]
    360 X_new, y_new = self._make_samples(
    361     X_class, y.dtype, class_sample, X_class, nns, n_samples, 1.0
    362 )
    363 X_resampled.append(X_new)

File c:\git\hsb\datamining\train-data-mining\.venv\Lib\site-packages\sklearn\neighbors\_base.py:854, in KNeighborsMixin.kneighbors(self, X, n_neighbors, return_distance)
    852     else:
    853         inequality_str = "n_neighbors <= n_samples_fit"
```

```
--> 854     raise ValueError(
    855         f"Expected {inequality_str}, but "
    856         f"n_neighbors = {n_neighbors}, n_samples_fit = {n_samples_fit}, "
    857         f"n_samples = {X.shape[0]}"  # include n_samples for common tests
    858     )
    860 n_jobs = effective_n_jobs(self.n_jobs)
    861 chunked_results = None

ValueError: Expected n_neighbors <= n_samples_fit, but n_neighbors = 4,
↪n_samples_fit = 1, n_samples = 1
```