



## Anomaly detection - Part 4

*One should look for what is and not what he thinks should be. -Albert Einstein*

# Module completion checklist

Objective	Complete
Implement LOF to detect anomalies	
Describe the isolation forest algorithm	
Implement isolation forest	
Implement isolation forest to detect anomalies	

# Loading packages

Let's load the packages we will be using:

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pickle

from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM
from sklearn.ensemble import IsolationForest
```

# Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into variables
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your course materials folder
- `data_dir` be the variable corresponding to your data folder

```
# Set 'main_dir' to location of the project folder
from pathlib import Path
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

# Load pickled data from previous module

```
non_fraud = pickle.load(open((data_dir + "/non_fraud.sav"), "rb"))
test = pickle.load(open((data_dir + "/test.sav"), "rb"))
actual_test = pickle.load(open((data_dir + "/actual_test.sav"), "rb"))
performance_df = pickle.load(open((data_dir + "/performance_anomalies.sav"), "rb"))
```

# Data: load energy consumption

- Load the PJME.csv dataset and print the head

```
pjm_energy = pd.read_csv(str(data_dir)+"/PJME_hourly.csv")  
pjm_energy.head()
```

		Datetime	PJME_MW
0	2002-12-31	01:00:00	26498.0
1	2002-12-31	02:00:00	25147.0
2	2002-12-31	03:00:00	24574.0
3	2002-12-31	04:00:00	24393.0
4	2002-12-31	05:00:00	24860.0

# Data: preprocessing

- Let's convert the `Datetime` variable from type **object** to **datetime**

```
pjm_energy['Datetime'] = pd.to_datetime(pjm_energy['Datetime'])  
pjm_energy.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 145366 entries, 0 to 145365  
Data columns (total 2 columns):  
#   Column      Non-Null Count  Dtype  
---  ---  
0   Datetime    145366 non-null  datetime64[ns]  
1   PJME_MW     145366 non-null  float64  
dtypes: datetime64[ns](1), float64(1)  
memory usage: 2.2 MB
```

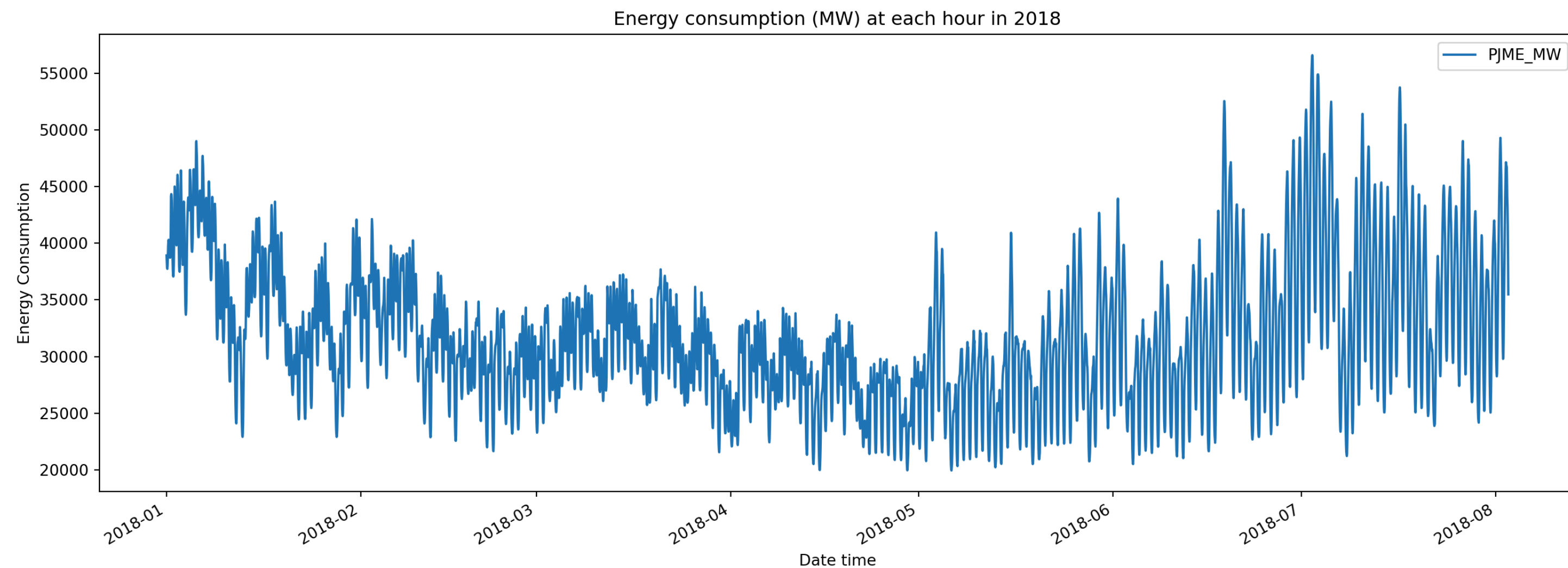
- We will filter the data to contain values for the year 2018

```
pjm_energy = pjm_energy[pjm_energy['Datetime'] > '2018-01-01 00:00:00']  
pjm_energy.shape
```

```
(5135, 2)
```

# Visualize the data: line plot

```
pjm_energy.plot(x='Datetime', y='PJME_MW', figsize=(17,6))  
plt.xlabel('Date time')  
plt.ylabel('Energy Consumption')  
plt.title('Energy consumption (MW) at each hour in 2018')  
plt.show()
```





# LOF model: energy consumption

- Since we don't have a target variable to evaluate the LOF model, we will fit the model on the entire data. This will enable the model to understand the underlying data distribution
- Once we have the model trained, we predict the anomalies on the same dataset
- The predicted values would be +1 for inliers and -1 for outliers
- We will implement the LOF model with `n_neighbors` set to **50** and `contamination` set to **0.01**

**Note:** As we have no evaluation metric, these values were picked based on experimenting with different parameters values and finalized with the one that showed good anomaly detection

# Create and fit LOF model: energy consumption

- We now will instantiate our LOF model with `n_neighbors` set to **50**

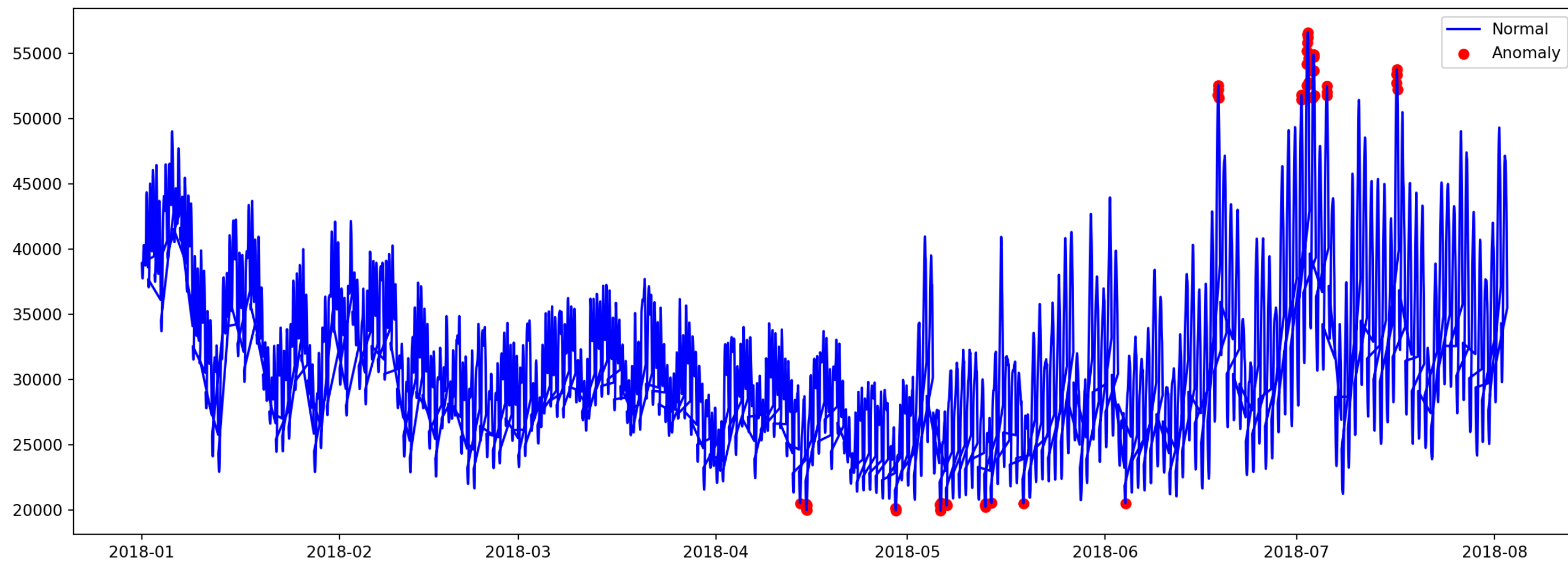
```
lof_energy_model = LocalOutlierFactor(n_neighbors = 50,  
                                     metric = "manhattan",  
                                     contamination = 0.01,  
                                     novelty = False)  
  
pjm_energy['anomaly'] = lof_energy_model.fit_predict(pd.DataFrame(pjm_energy['PJME_MW']))
```

# LOF - visualize anomalies

```
# visualization
fig, ax = plt.subplots(figsize=(17,6))

a = pjm_energy.loc[pjm_energy['anomaly'] == -1, ['Datetime', 'PJME_MW']] #anomaly

ax.plot(pjm_energy['Datetime'], pjm_energy['PJME_MW'], color='blue', label = 'Normal')
ax.scatter(a['Datetime'],a['PJME_MW'], color='red', label = 'Anomaly')
plt.legend()
plt.show();
```



# LOF - visualize anomalies

- Identify the lower and the upper range of anomalies detected

```
lower_threshold = pjm_energy['PJME_MW'].quantile(0.25)
upper_threshold = pjm_energy['PJME_MW'].quantile(0.75)
lof_anomalies = pjm_energy[pjm_energy['anomaly'] == -1]
```

```
# Upper range of anomalies
lof_upper_anomalies = lof_anomalies[lof_anomalies['PJME_MW'] > upper_threshold]['PJME_MW']

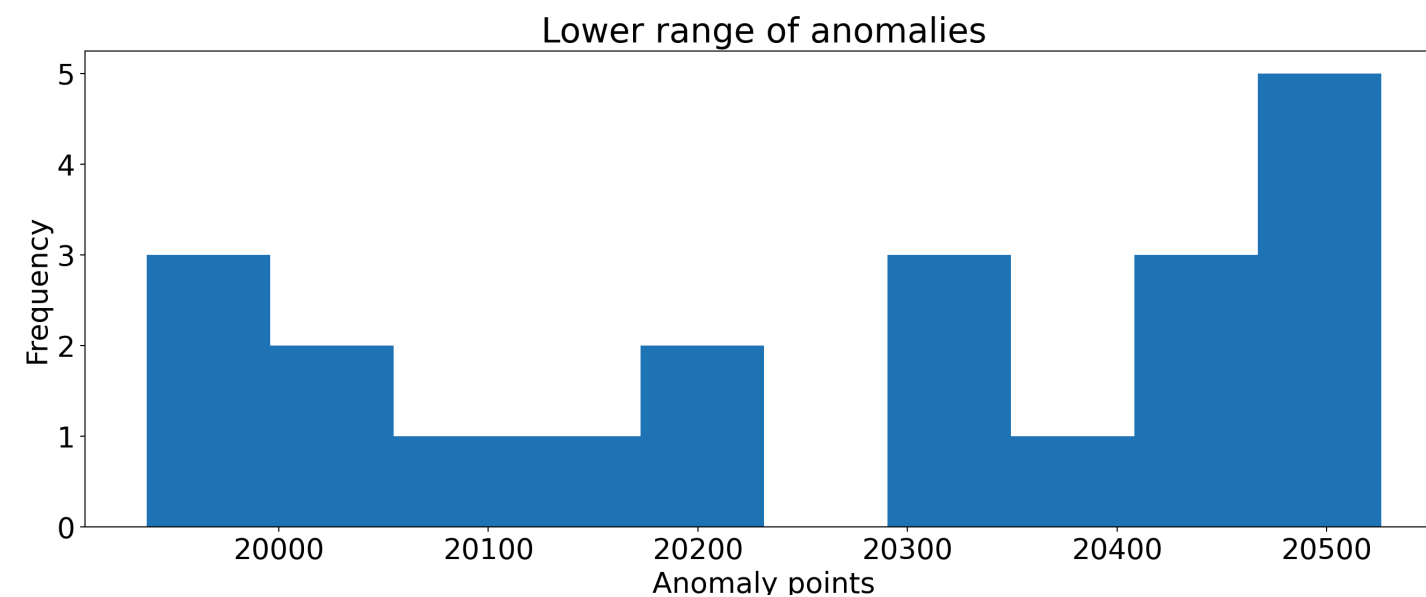
# Lower range of anomalies
lof_lower_anomalies = lof_anomalies[lof_anomalies['PJME_MW'] < lower_threshold]['PJME_MW']
```

# LOF - visualize anomalies

- Lower range of anomalies

```
plt.rcParams.update({'font.size': 20})  
plt.hist(lof_lower_anomalies)
```

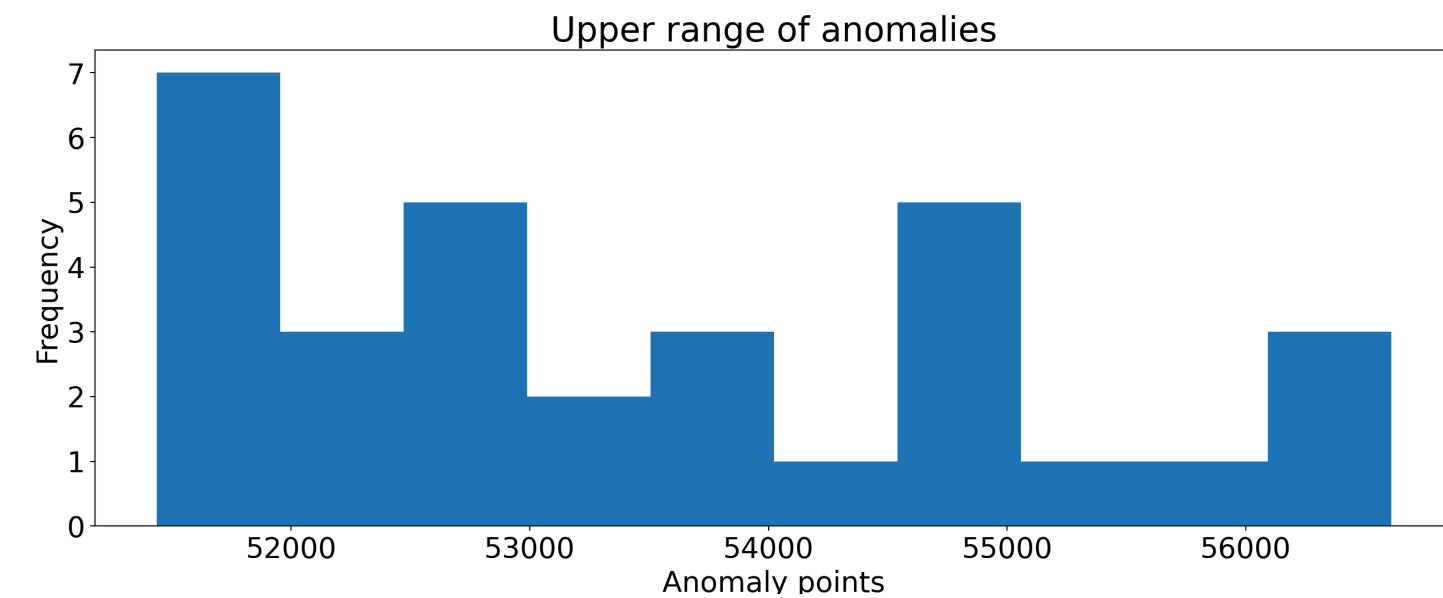
```
plt.xlabel("Anomaly points")  
plt.ylabel("Frequency")  
plt.title("Lower range of anomalies")  
plt.show()
```



- Upper range of anomalies

```
plt.hist(lof_upper_anomalies)
```

```
plt.xlabel("Anomaly points")  
plt.ylabel("Frequency")  
plt.title("Upper range of anomalies")  
plt.show()
```



# Exercise 3



# Module completion checklist

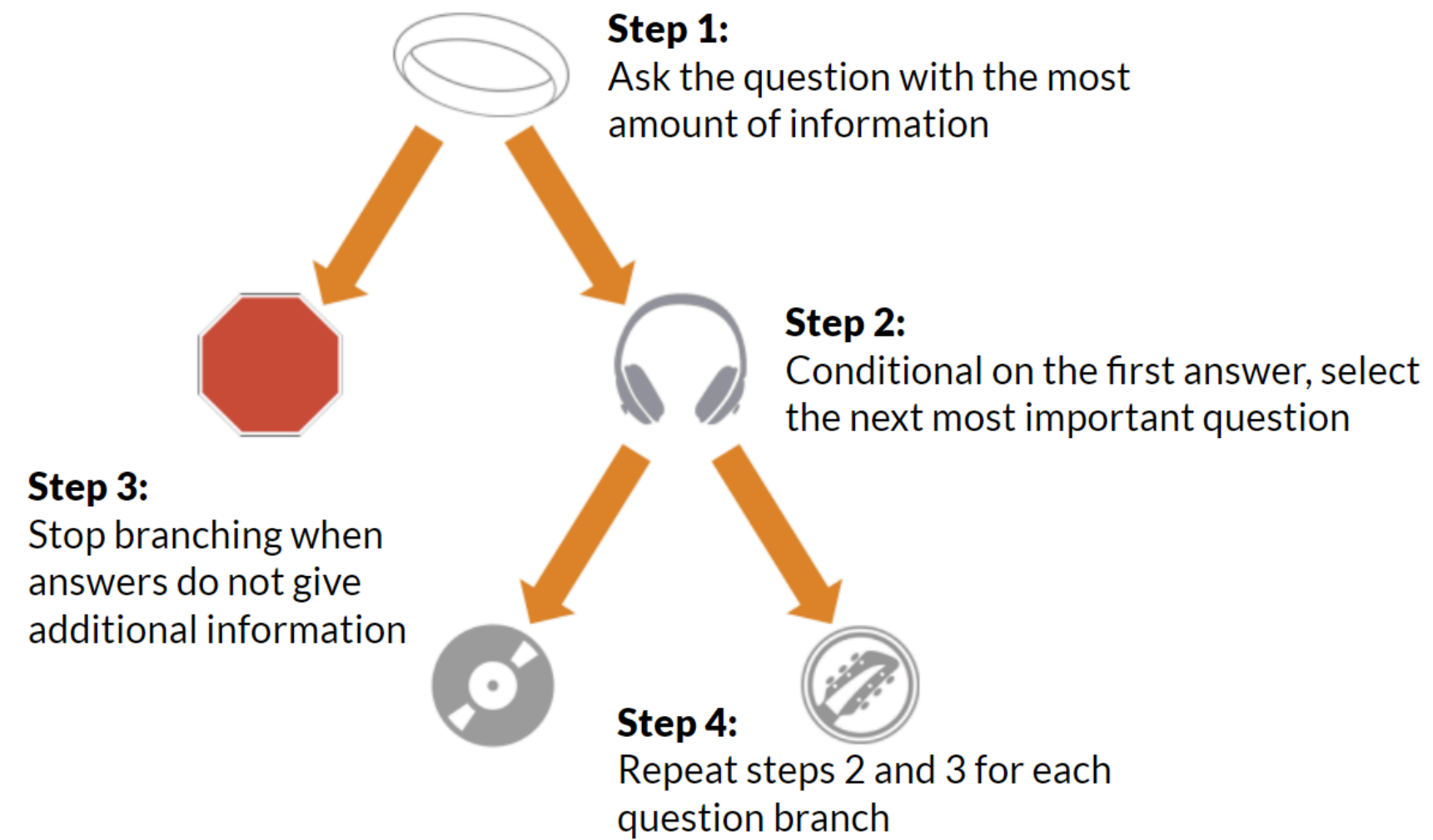
Objective	Complete
Implement LOF to detect anomalies	✓
Describe the isolation forest algorithm	
Implement isolation forest	
Implement isolation forest to detect anomalies	

# Decision trees: recap

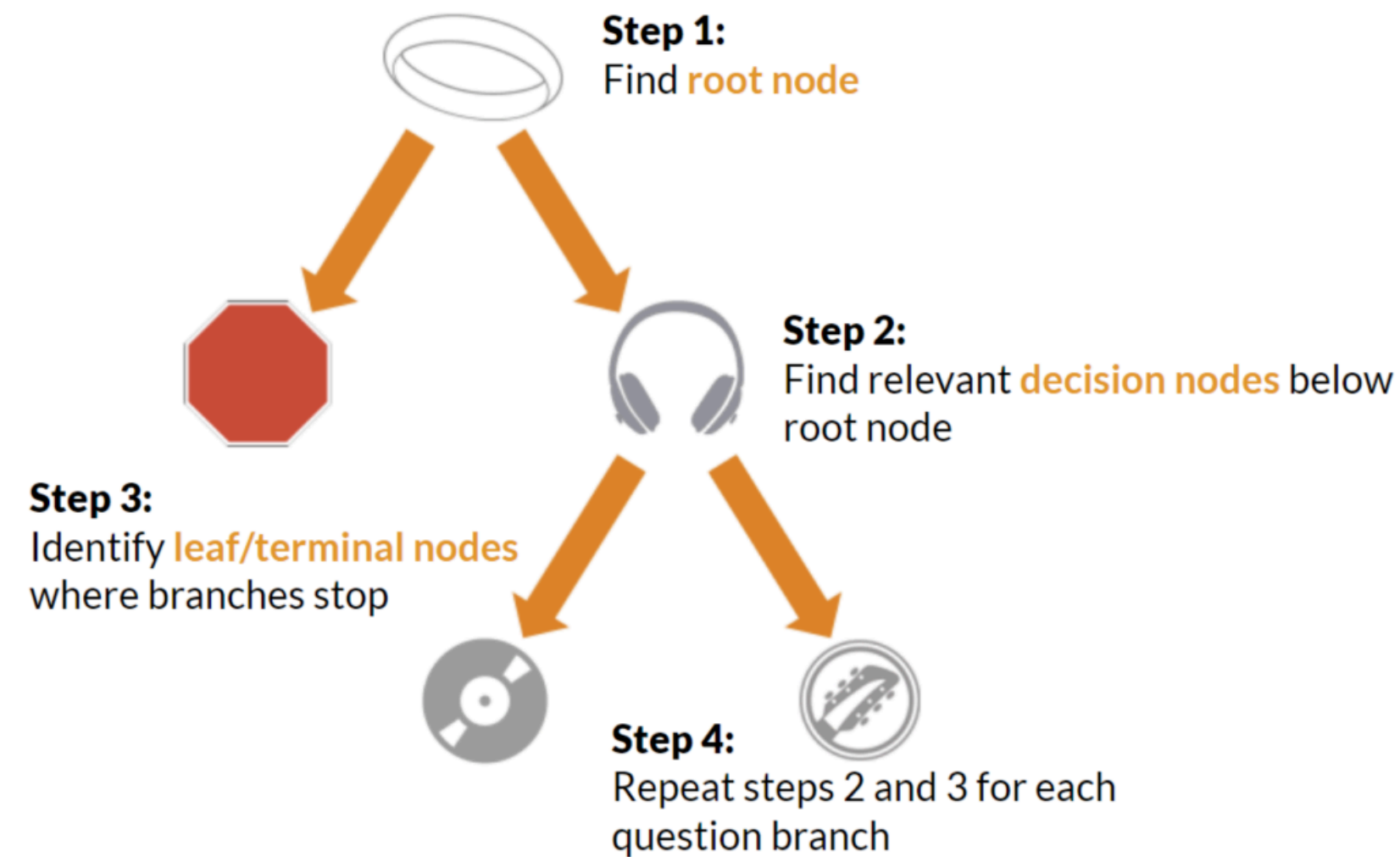
- Isolation forests are built on the basis of decision trees
- In decision trees, partitions are created by first randomly selecting a feature and then selecting a random split value between the minimum and maximum value of the selected feature



# Growing decision trees steps

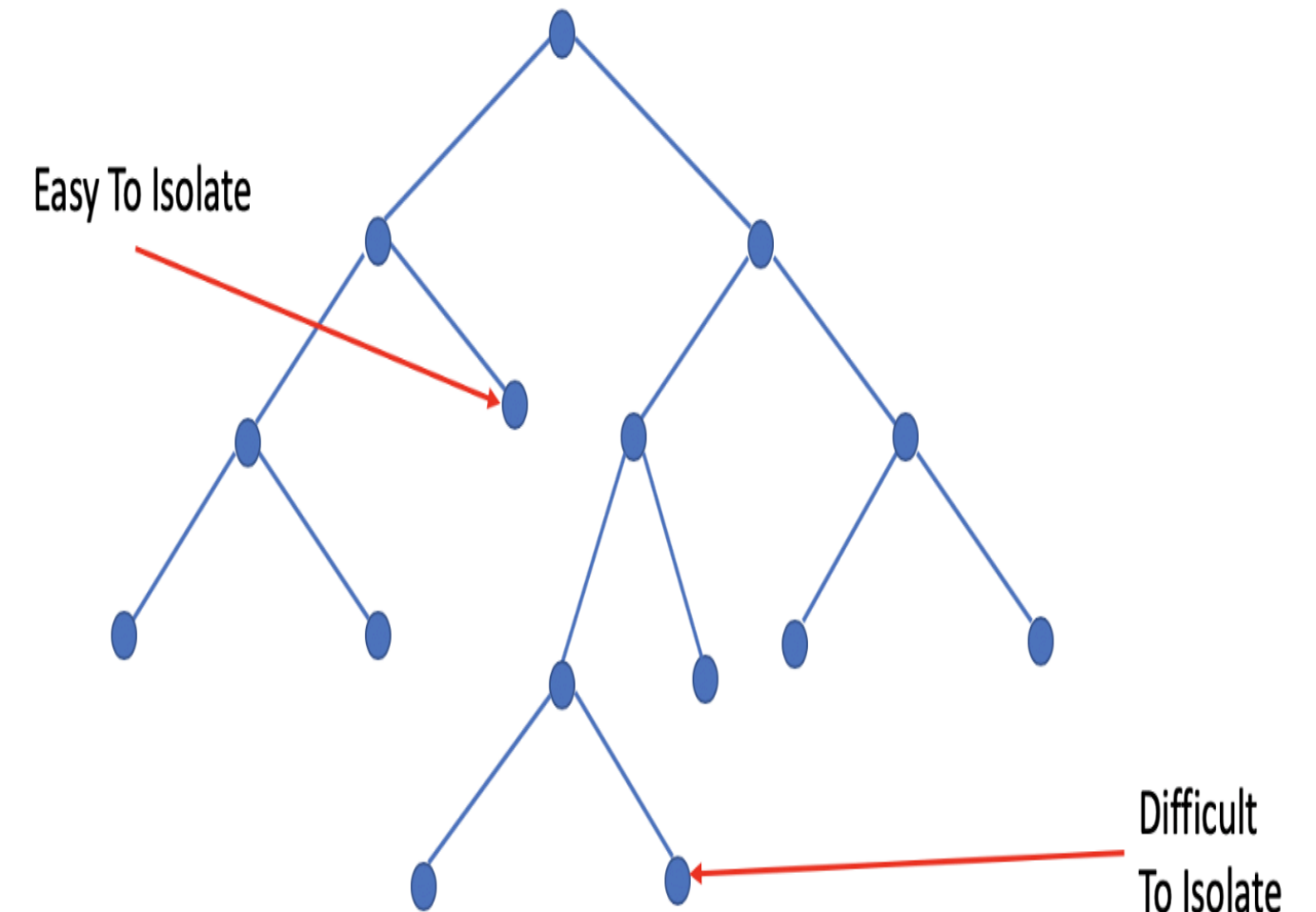


# Growing decision trees with vocabulary



# Isolation forest

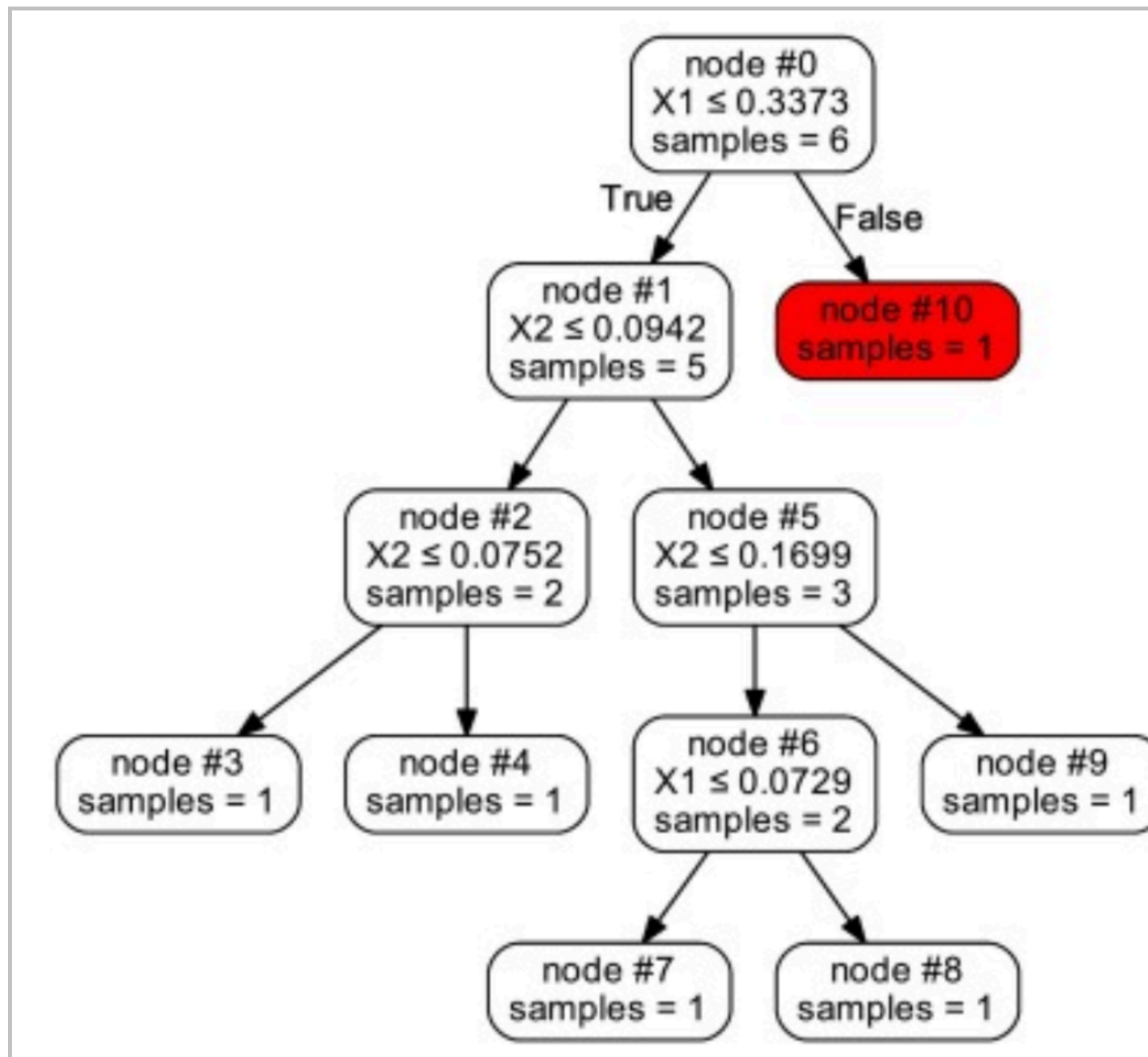
- We know that outliers are less frequent and differ from regular observations
- This means they will also be identified faster and closer to the root during the partitioning at each feature node
- An isolation forest algorithm calculates an anomaly score for each test observation which we want to classify based on the path length
- Path length is the number of nodes the observation travels down the decision tree
- Based on the anomaly score, it is classified either as an inlier (1) or outlier (-1)



# Working of isolation forest

- In the isolation forest, the model is built only on regular observations
- When we get a new test data point, it travels through each node and gets classified as either
  - inlier or normal observation (+1)
  - outlier or anomaly (-1)
- Like random forest, isolation forest has multiple decision trees where the results are aggregated
- Let's say we have a new observation  $a$ , which is an outlier
- As the new observation travels down the tree, we note that none of its features have the same range as the regular observation
- The features of the new observation are very different from the tree model
- Hence, the new observation is classified as an anomaly and assigned label  $-1$

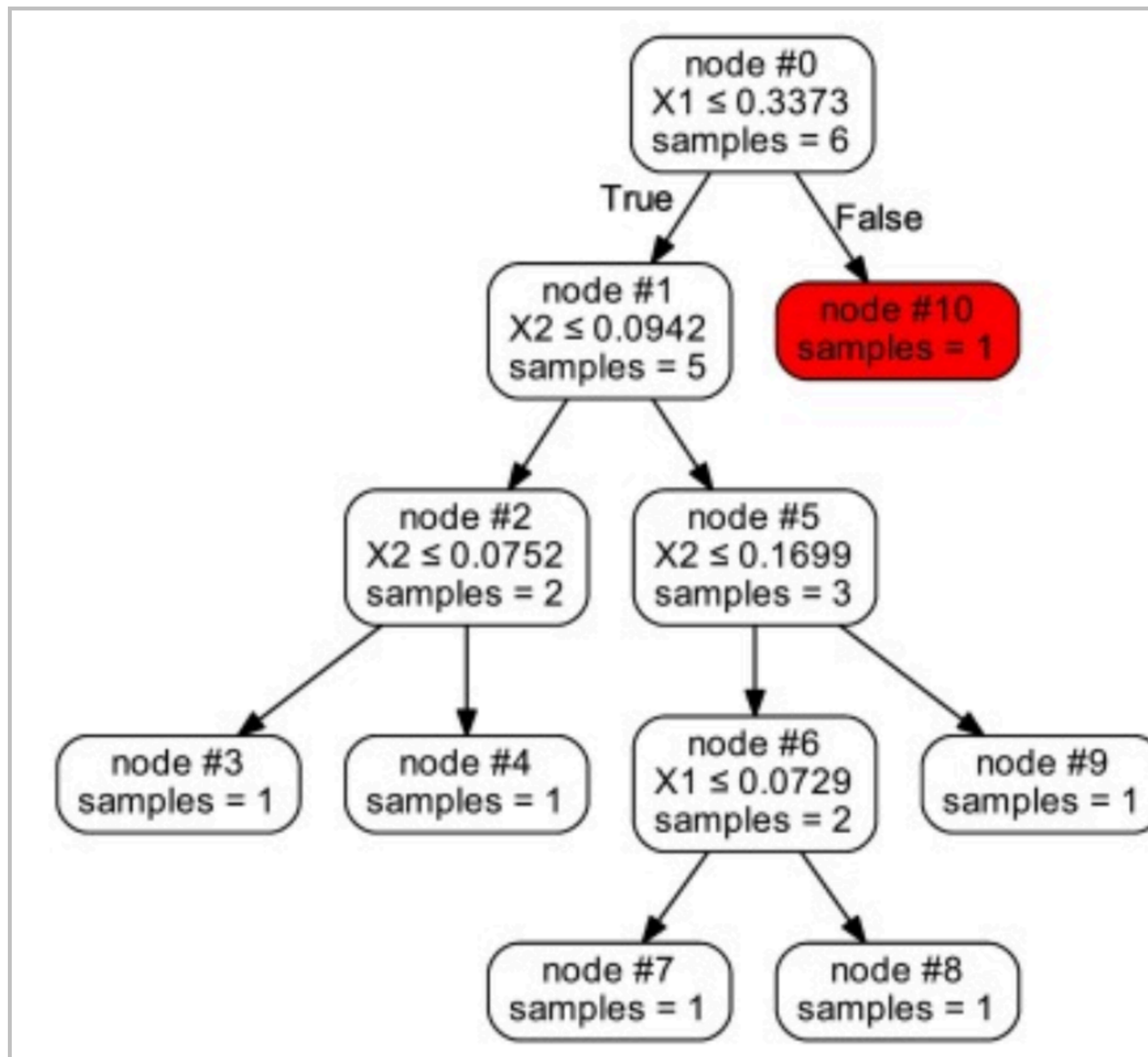
# Working of isolation forest with example 1



- Let's say we built our tree model with regular observation as shown
- We have a new observation which has a variable(X1) value greater than a threshold (0.3373), then it gets classified as an anomaly because all the observations in the tree model has  $X1 \leq \text{threshold (0.3373)}$



# Working of isolation forest with example 2



- Let's say we have another new observation which is also an anomaly but its  $X1 \leq \text{threshold}$  (0.3373)
- But its other variable ( $X2$ ) is greater than threshold2 (0.0942)
- That observation gets classified as outlier at the second node (node #1) because all the observations within the tree model data have values with  $X2 \leq \text{threshold2}$  (0.0942)

# Isolation forest in Python

- Our SciKit library has a package for isolation forest
- Read more on it [here](#)

## `sklearn.ensemble.IsolationForest`

```
class sklearn.ensemble. IsolationForest (n_estimators=100, max_samples='auto', contamination='legacy',  
max_features=1.0, bootstrap=False, n_jobs=None, behaviour='old', random_state=None, verbose=0,  
warm_start=False)
```

[\[source\]](#)

Isolation Forest Algorithm

Return the anomaly score of each sample using the IsolationForest algorithm

The IsolationForest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

# Module completion checklist

Objective	Complete
Implement LOF to detect anomalies	✓
Describe the isolation forest algorithm	✓
Implement isolation forest	
Implement isolation forest to detect anomalies	



# Create and fit isolation forest model

- We now will instantiate our isolation forest model and run it on non\_fraud data
- At first, we will simply run the model on our training data and predict on test
- We set default parameter for **n\_estimators = 100** and contamination as **0.1**

```
iforest = IsolationForest(n_estimators=100, contamination = 0.1)

# model fitting
iforest.fit(non_fraud)
```

```
IsolationForest(contamination=0.1)
```

# Test predictions

- Predict on the test data using the trained isolation forest

```
fraud_pred = iforest.predict(test.iloc[:, :-1])  
fraud_pred
```

```
array([1, 1, 1, ..., 1, 1, 1])
```

- We know that an isolation forest classifies data points as -1 and +1 instead of 1 and 0
- Let's replace these values into 0 and 1 as we have in our Paysim dataset

```
fraud_pred[fraud_pred == 1] = 0  
fraud_pred[fraud_pred == -1] = 1
```

# Find TPR and TNR

- Let's evaluate the isolation forest model

```
tn, fp, fn, tp = confusion_matrix(actual_test, fraud_pred).ravel()  
non_fraud_eval = tn / (tn + fp)  
print(non_fraud_eval)
```

```
0.9005506424161521
```

```
fraud_eval = tp / (tp + fn)  
print(fraud_eval)
```

```
0.34545454545454546
```

# Load performance\_df dataframe

- Append the scores of the isolation forest to the performance\_df dataframe

```
s = pd.Series(['Isolation Forest', fraud_eval, non_fraud_eval],  
              index=['model_name', 'TPR', 'TNR'])  
performance_df = performance_df.append(s, ignore_index = True)  
performance_df
```

	model_name	TPR	TNR
0	Decision_tree_baseline	0.671642	0.999667
1	SMOTE	0.865672	0.991332
2	LOF	0.772727	0.890927
3	LOF	0.759091	0.903280
4	LOF	0.750000	0.894868
5	LOF	0.745455	0.900367
6	LOF	0.754545	0.892982
7	LOF	0.722727	0.893376
8	Isolation Forest	0.345455	0.900551

# Isolation model: hyperparameter tuning

- The hyperparameters that can be tuned for optimizing the isolation forest model are shown below
  - **n\_estimators**: the number of base estimators in the ensemble
  - **contamination**: the amount of contamination of the data set, i.e., the proportion of outliers in the data set
  - **max\_features**: the number of features to draw from X to train each base estimator
  - **max\_samples**: the number of samples to draw from X to train each base estimator
- We wouldn't be tuning the hyperparameters for isolation model here, as the model results showed no significant improvement on this dataset

# Knowledge check 3



# Exercise 4



# Module completion checklist

Objective	Complete
Implement LOF to detect anomalies	✓
Describe the isolation forest algorithm	✓
Implement isolation forest	✓
Implement isolation forest to detect anomalies	



# Isolation forest on time series data

- We now will instantiate our isolation forest model and run it on time series data
- We set default parameter for **n\_estimators = 100** and contamination as **0.01**

```
isolation_energy = IsolationForest(n_estimators=100, contamination = 0.01)

# model fitting
isolation_energy.fit(pd.DataFrame(pjm_energy['PJME_MW']))
```

```
IsolationForest(contamination=0.01)
```

```
pjm_energy['anomaly'] = isolation_energy.predict(pd.DataFrame(pjm_energy['PJME_MW']))
```

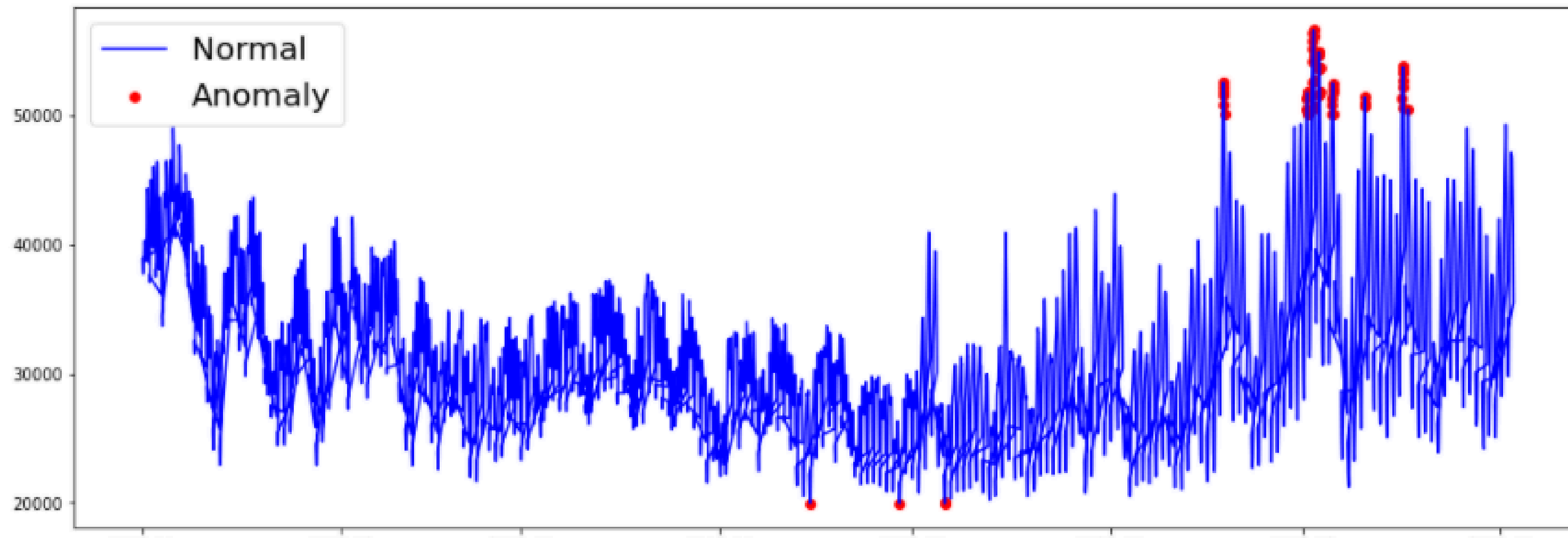
# Isolation forest - visualize anomalies

- We will now visualize the anomalies detected by isolation forest

```
# visualization
fig, ax = plt.subplots(figsize=(15,5))

a = pjm_energy.loc[pjm_energy['anomaly'] == -1, ['Datetime', 'PJME_MW']] #anomaly

ax.plot(pjm_energy['Datetime'], pjm_energy['PJME_MW'], color='blue', label = 'Normal')
ax.scatter(a['Datetime'],a['PJME_MW'], color='red', label = 'Anomaly')
plt.legend()
plt.show()
```



# Isolation forest - visualize anomalies

- Identify the lower and the upper range of anomalies detected

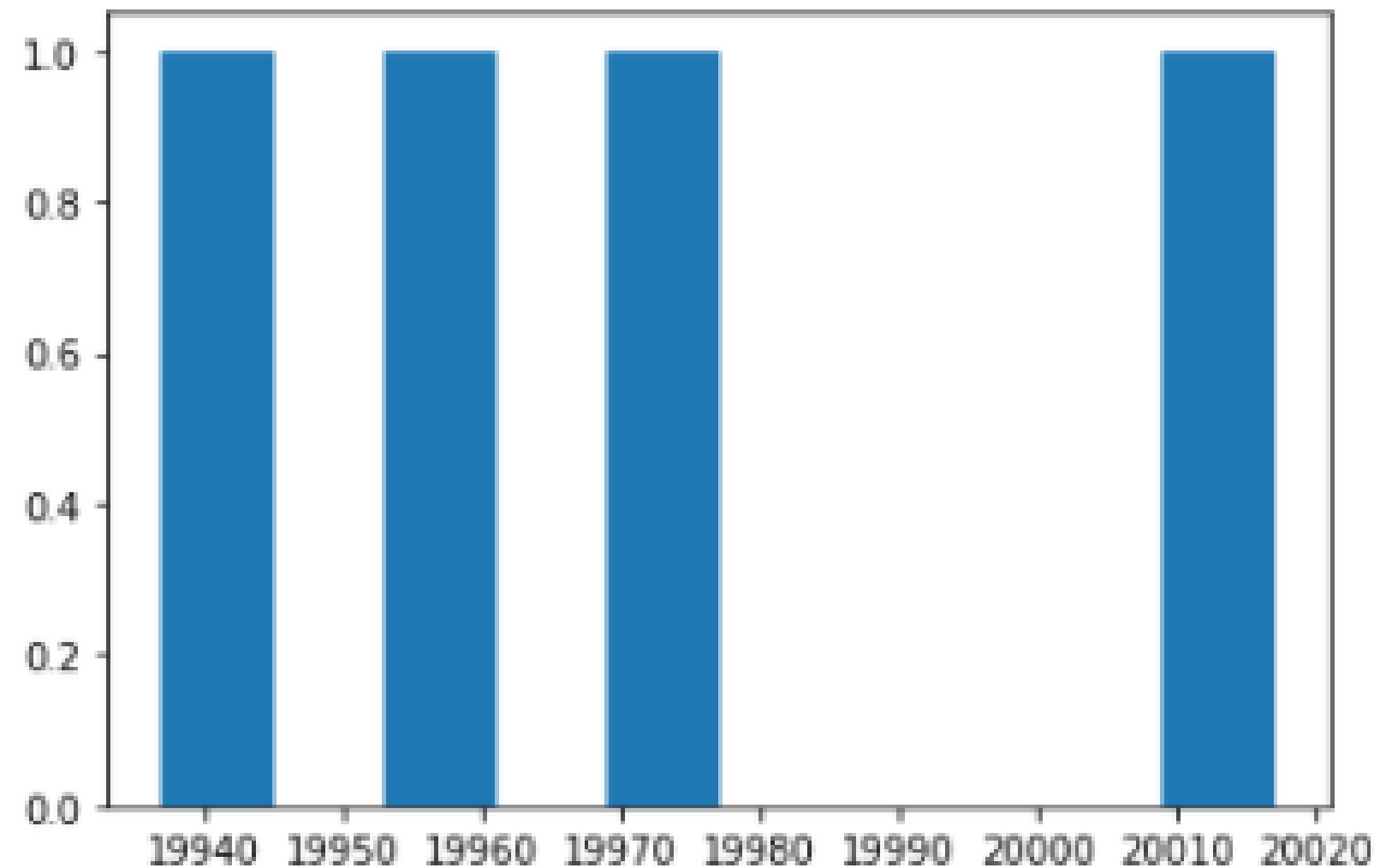
```
# visualization
lower_threshold = pjm_energy['PJME_MW'].quantile(0.25)
upper_threshold = pjm_energy['PJME_MW'].quantile(0.75)
if_anomalies = pjm_energy[pjm_energy['anomaly'] == -1]

if_upper_anomalies = if_anomalies[if_anomalies['PJME_MW'] > upper_threshold]['PJME_MW']
if_lower_anomalies = if_anomalies[if_anomalies['PJME_MW'] < lower_threshold]['PJME_MW']
```

# Isolation forest - visualize anomalies

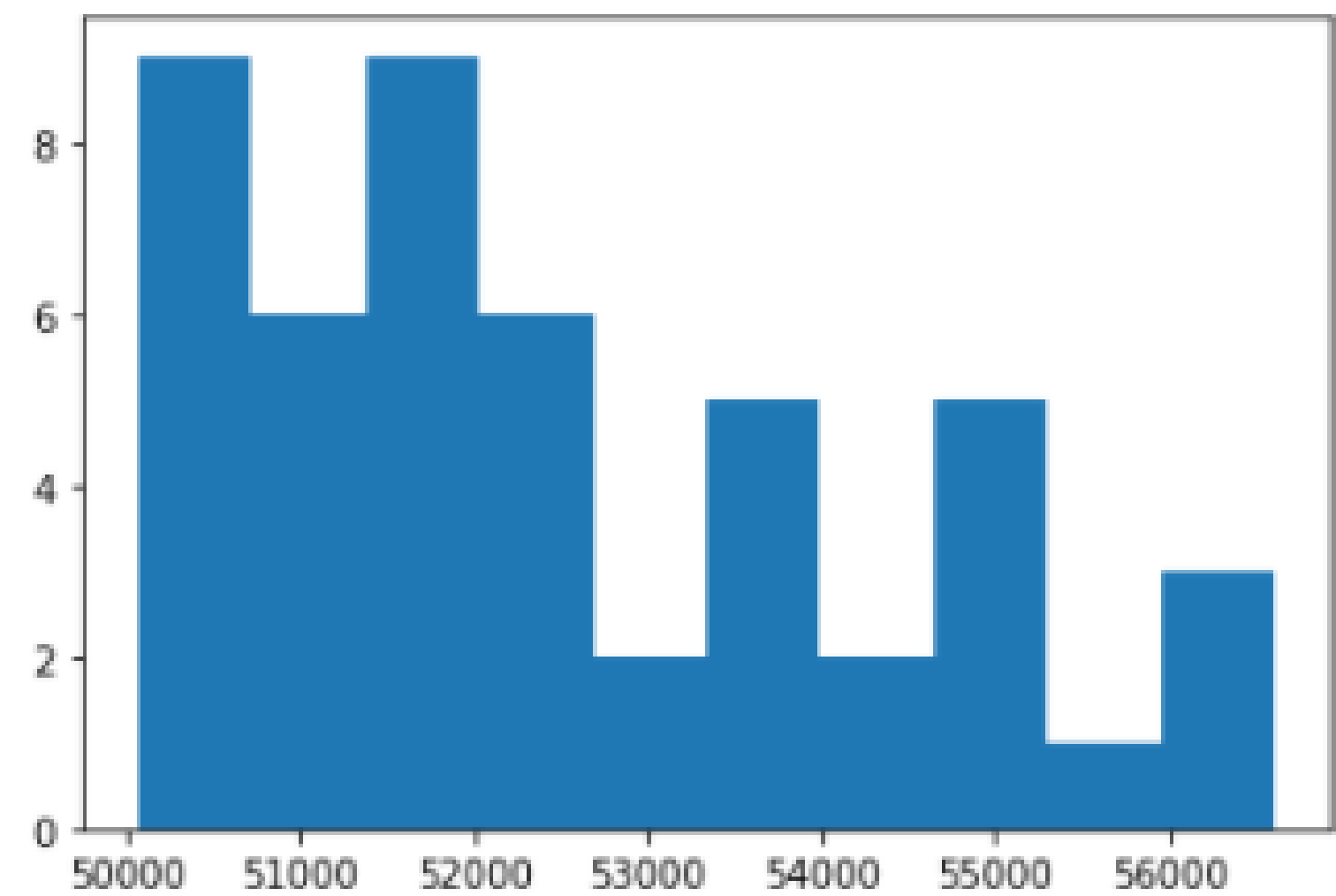
- Lower range of anomalies

```
plt.hist(if_lower_anomalies)
plt.xlabel("Anomaly points")
plt.ylabel("Frequency")
plt.title("Lower range of anomalies")
plt.show()
```



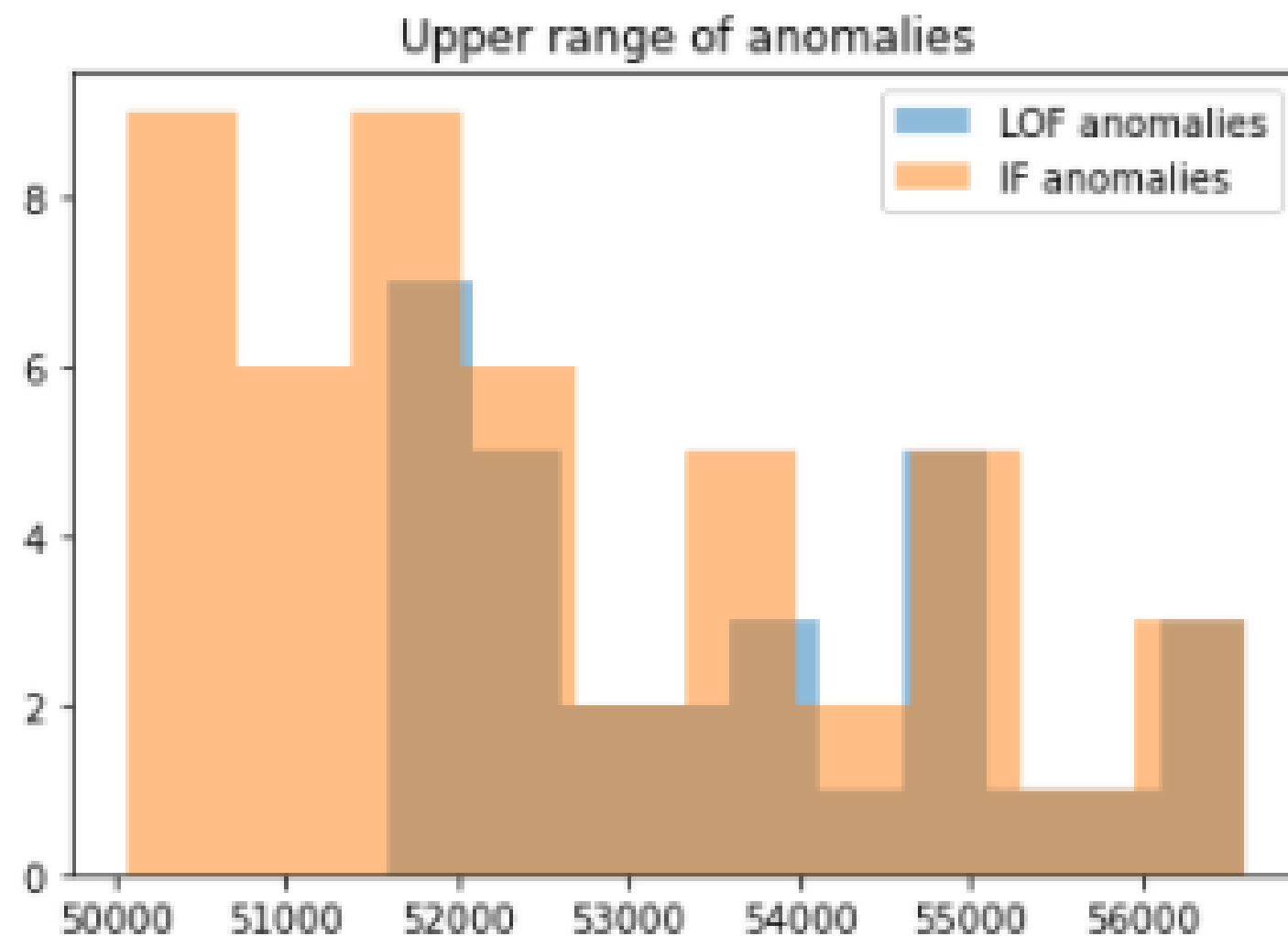
- Upper range of anomalies

```
plt.hist(if_upper_anomalies)
plt.xlabel("Anomaly points")
plt.ylabel("Frequency")
plt.title("Upper range of anomalies")
plt.show()
```

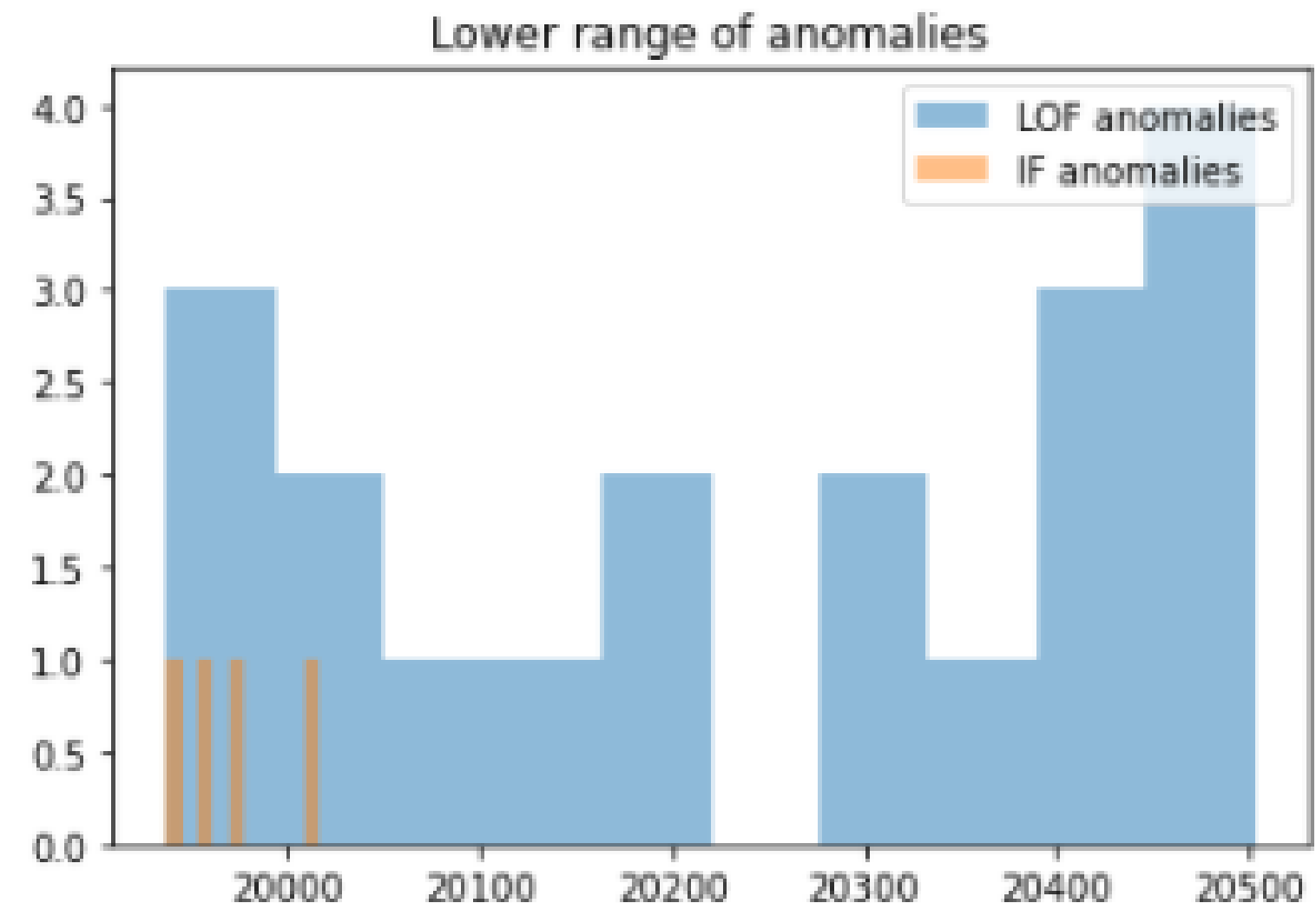


# Compare anomalies

```
plt.hist(lof_upper_anomalies, alpha =  
0.5, label='LOF anomalies')  
plt.hist(if_upper_anomalies, alpha = 0.5, label='IF  
anomalies')  
plt.title('Upper range of anomalies')  
plt.legend(loc='upper right')  
plt.show()
```



```
plt.hist(lof_lower_anomalies, alpha =  
0.5, label='LOF anomalies')  
plt.hist(if_lower_anomalies, alpha = 0.5, label='IF  
anomalies')  
plt.title('Lower range of anomalies')  
plt.legend(loc='upper right')  
plt.show()
```



# Exercise 5



# Module completion checklist

Objective	Complete
Implement LOF to detect anomalies	✓
Describe the isolation forest algorithm	✓
Implement isolation forest	✓
Implement isolation forest to detect anomalies	✓

# What's next?

- In this module we've learned about two anomaly detection techniques - LOF and isolation forest
- We implemented these techniques to detect fraud and anomalies in energy consumption
- In the next module we will learn:
  - concepts of time series modeling and its implementation
  - how ARIMA model can be used for anomaly detection



# Congratulations on completing this module!

