

# Machine Learning



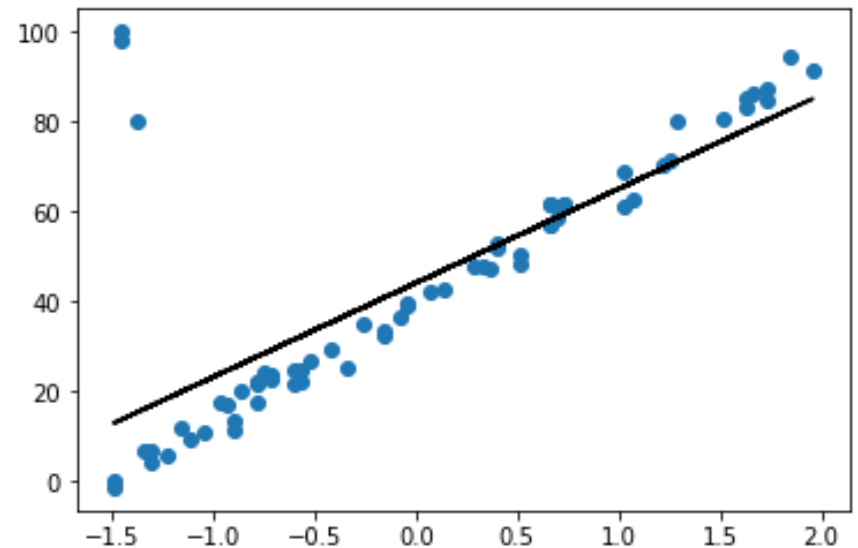
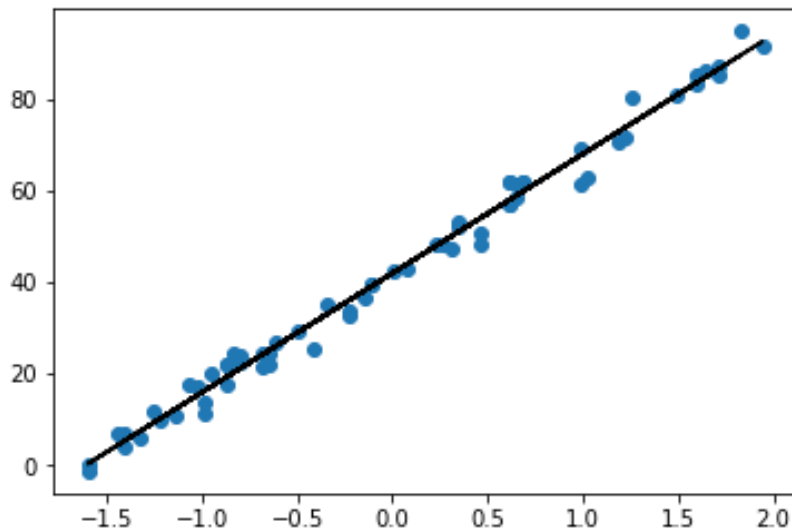
## Machine Learning

Part 1 – Data Preparation – Outlier Detection

Ted Scully

# Outlier Detection

- ▶ Outliers are data that differ significantly from other data in a sample.
- ▶ Outliers **skew your data distributions** and impact your basic statistical measures and can be responsible for **underperformance** of certain algorithms.
- ▶ Outliers might be caused by faulty equipment, human error such as data entry, transcribing results, data processing error, etc

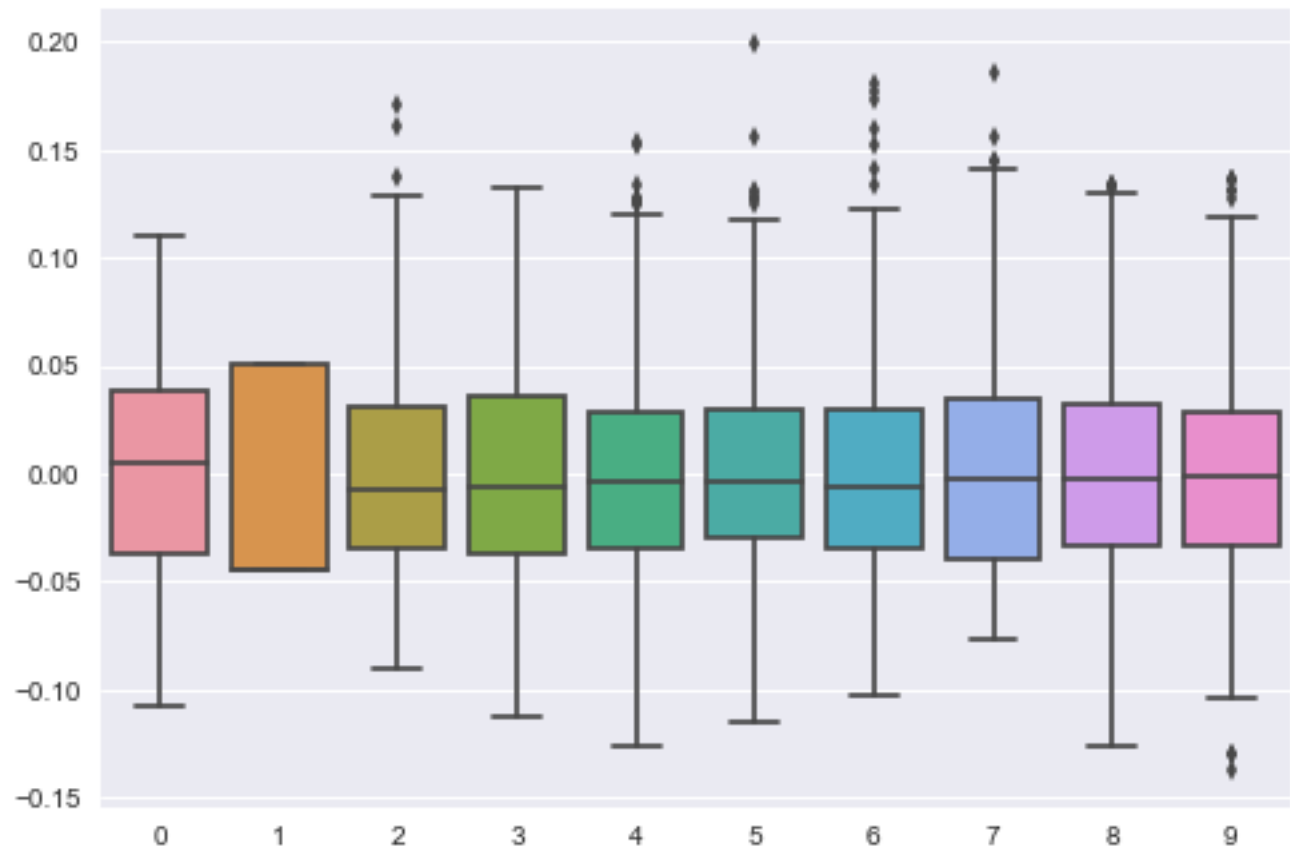


```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets

diab = datasets.load_diabetes()
X = diab.data
y = diab.target

sns.boxplot(data= pd.DataFrame(X))
plt.show()
```

- ▶ It is often useful to **plot all features together** to get an overview of the entire dataset (clearly this is only practical if all features have been normalized or standardized in advance or appear within the same range).
- ▶ We can do this by passing in a dataframe containing all the features from our dataset.

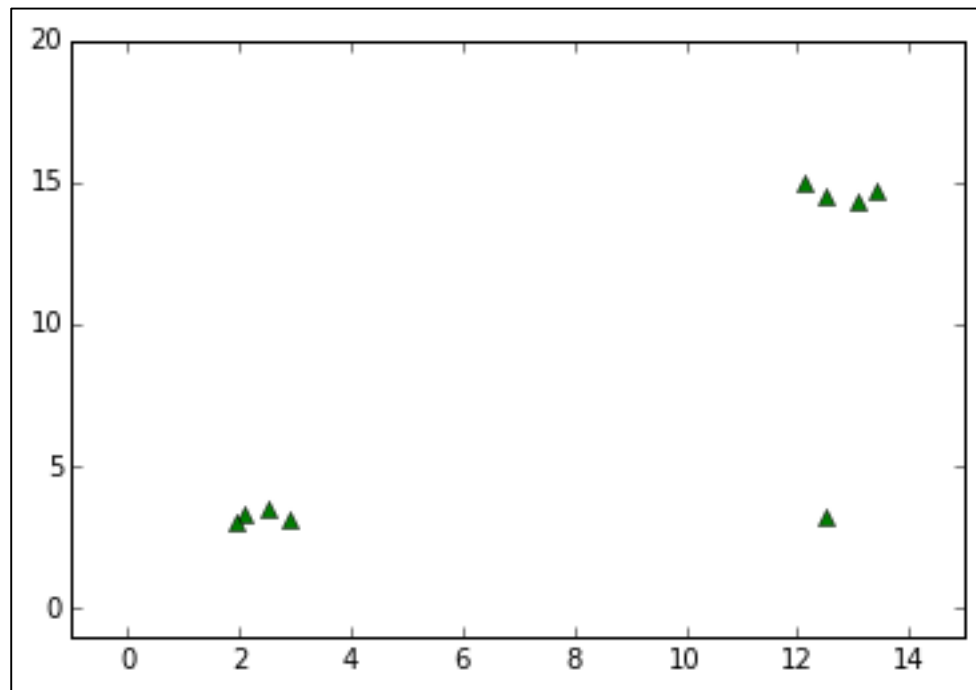


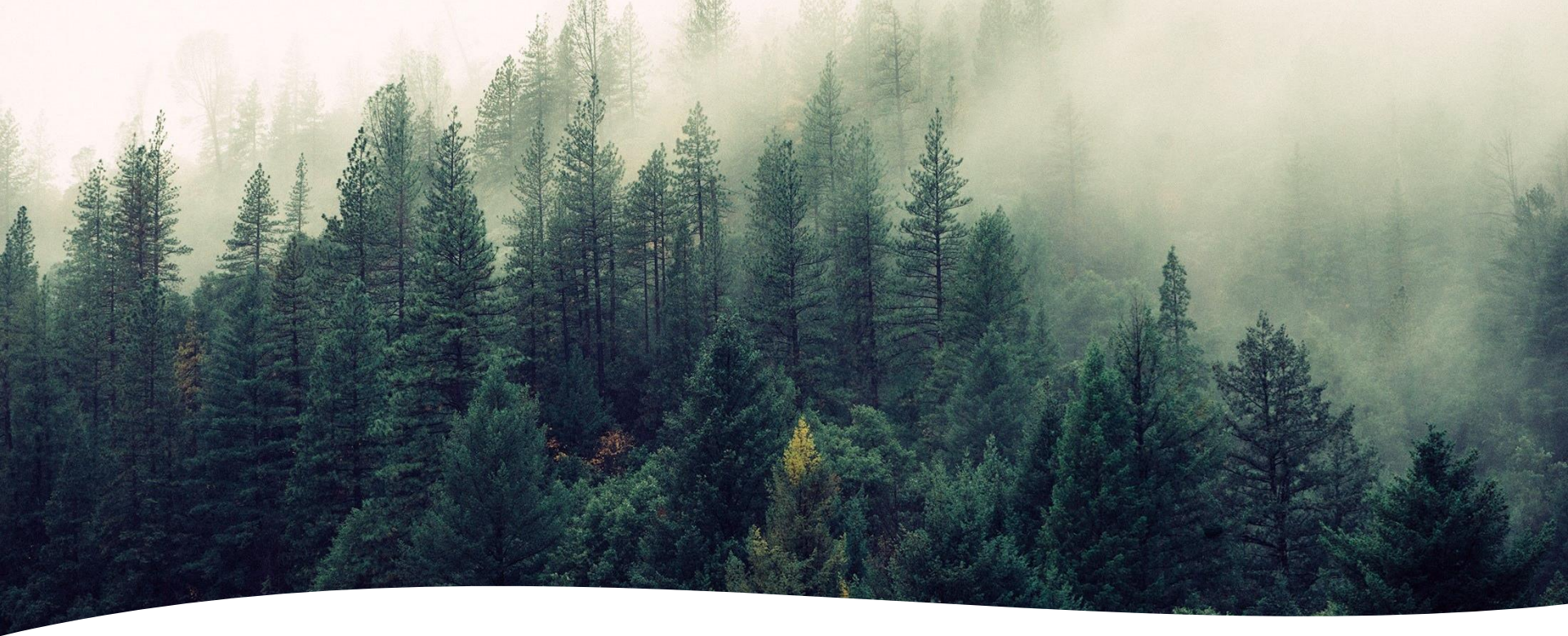
# What to do with the outliers

- ▶ **Recommendation**: Recommend that you “err on the side of caution” when dealing with outliers.
- ▶ If data points appear just outside the whisker boundaries or appear in significant quantity I would not recommend classifying them as outliers. Remember outliers are really datapoints that are significantly isolated and removed from the main distribution of data points.
- ▶ There are a number of different methods for dealing with outliers.
  - ▶ Deletion of the associated instance from the dataset
  - ▶ Clamping outlier values. Resetting the outlier value to some upper or lower boundary value (for example, upper outlier values can be set to  $\text{median} + 1.5 * \text{IQR}$ , lower outlier values will be set to  $\text{median} - 1.5 * \text{IQR}$ , lower )
  - ▶ Set the outlier as a missing value and impute

# Visualization for Outliers

- ▶ However, notice if we plot one feature against another we can see that one data point is significantly removed from another data point.
- ▶ Univariate outlier detection is limited in this regard as you aren't considering unusual combinations of multiple variables





# Isolation Forests for Multi-variate Outlier Detection

- Isolation forest are an **unsupervised learning** approach to identify isolated instances (outliers or anomalies) in feature space.
- The fundamental rationale behind an isolation forest (which consists of many isolation trees) is that it is much **easier to isolate an outlier data point in feature space** compare to isolating a data point that is part of an existing cluster.



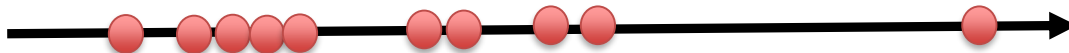
# Isolation Forests for Multi-variate Outlier Detection

Consider the values from a single feature represented graphically below (there are just 10 values for this feature).

We will **randomly pick** a point to partition this feature (a point randomly chosen between the **max** and **min** value for the feature).

Let's assume the partition divides the data in into two groups. In one group we will only have a single instance (data point) and in the other group we will have the remaining nine in another partition.

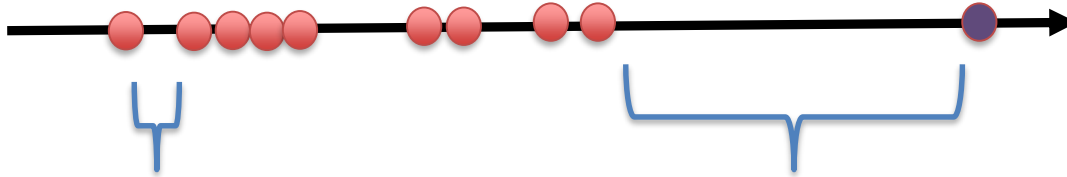
Which data point is most likely to be the single data point?



# Isolation Forests for Multi-variate Outlier Detection

Which data point is most likely to be the single data point.

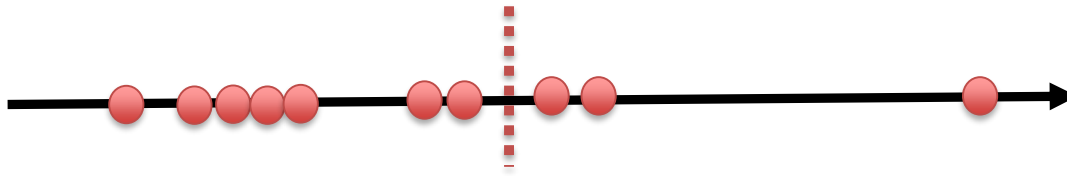
The data point on the right hand side is most likely to be the isolated instance as it is so far from the other data points.





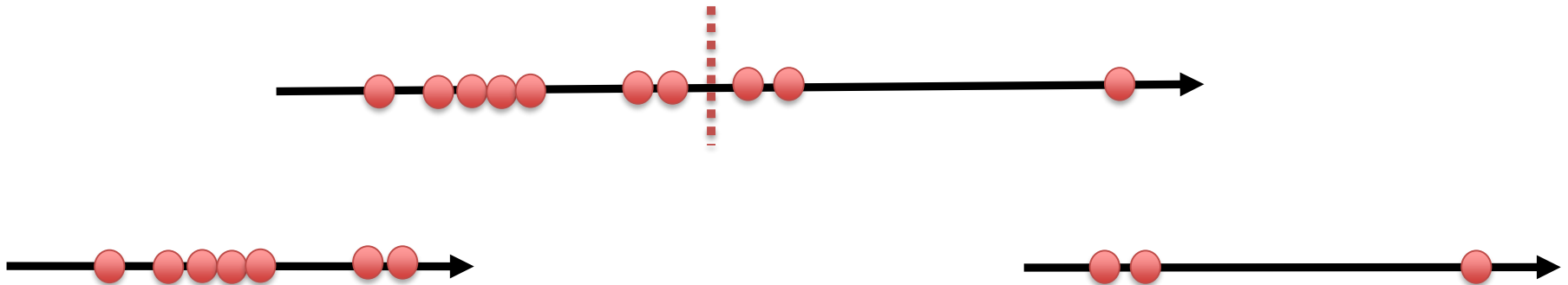
# Isolation Forests for Multi-variate Outlier Detection

Now let's say we were to repeat the process until we isolated one instance .... clearly the instance on the right is still the most likely to be isolated.



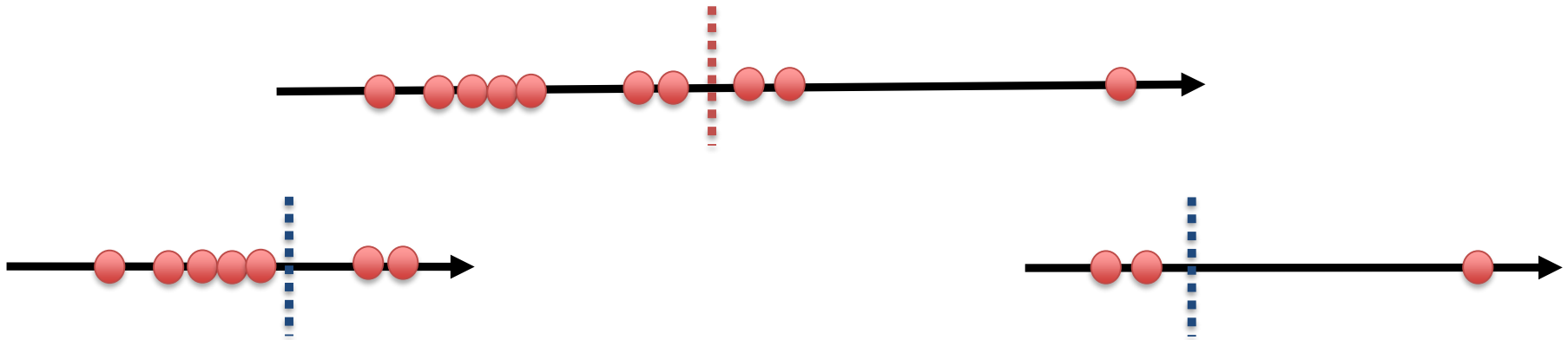
# Isolation Forests for Multi-variate Outlier Detection

Now let's say we were to repeat the process until we isolated one instance .... clearly the instance on the right is still the most likely to be isolated.



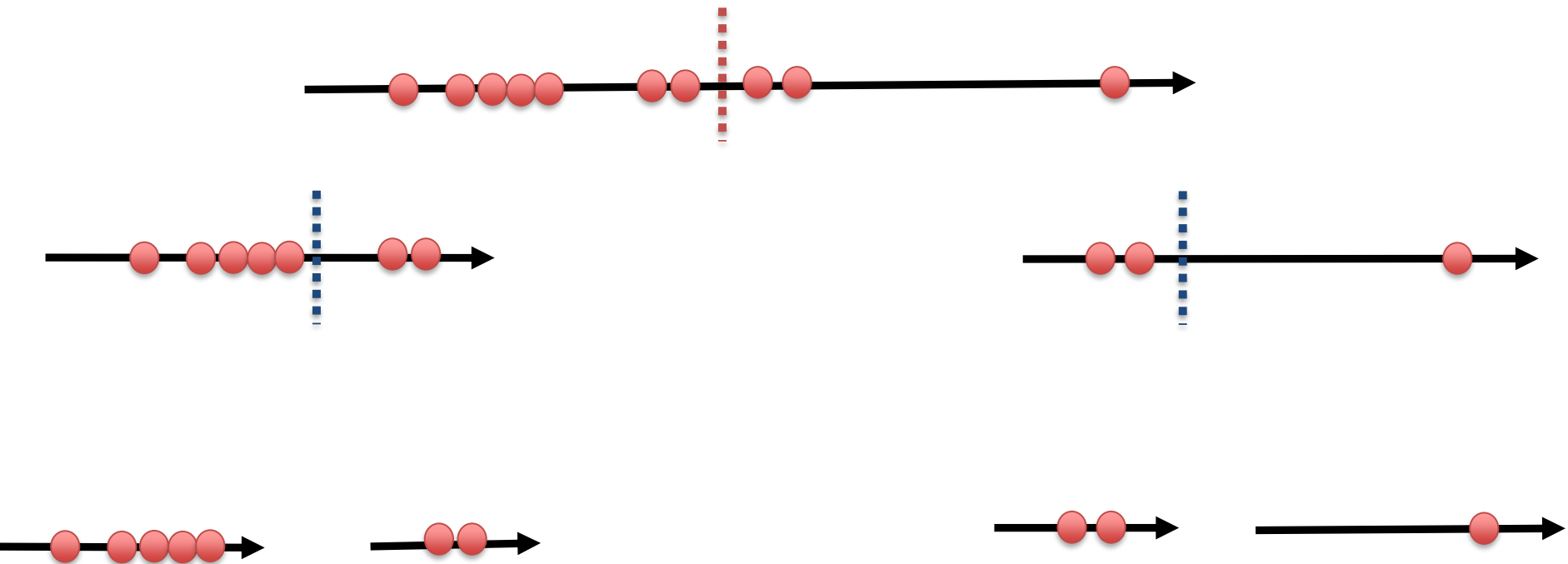
# Isolation Forests for Multi-variate Outlier Detection

Now let's say we were to repeat the process until we isolated one instance .... clearly the instance on the right is still the most likely to be isolated.



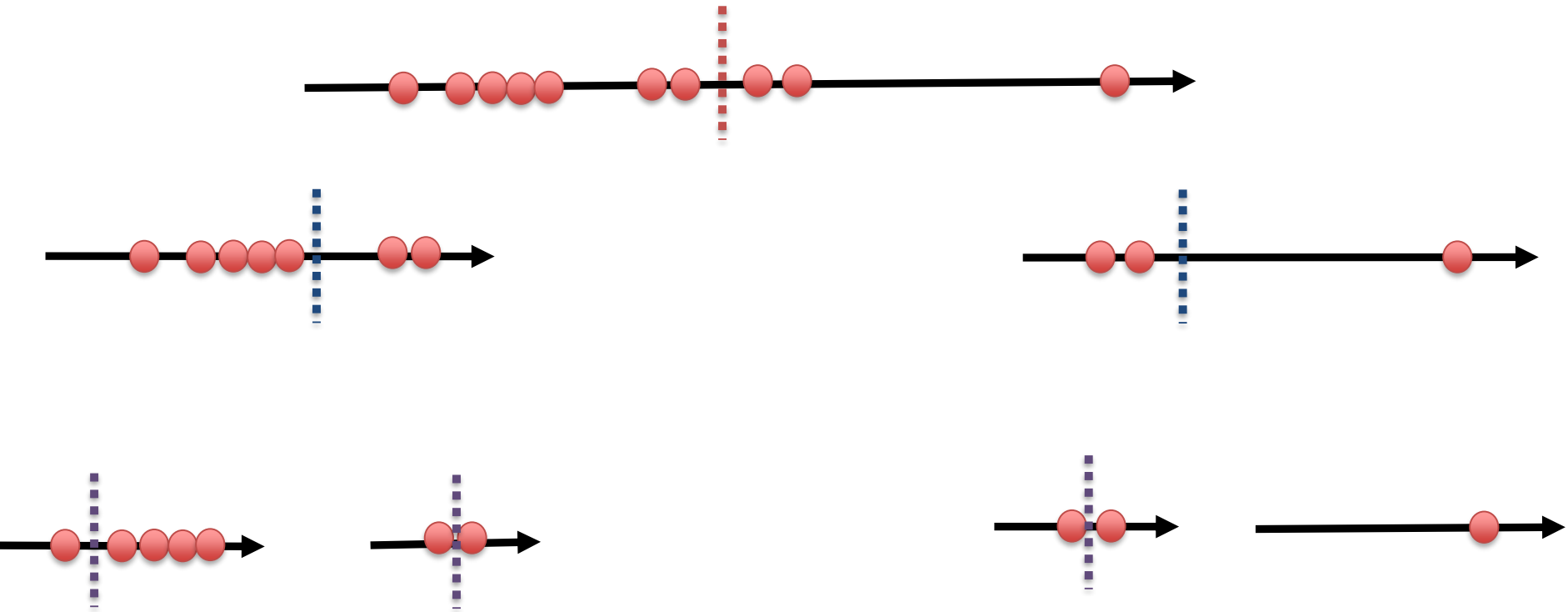
# Isolation Forests for Multi-variate Outlier Detection

Now let's say we were to repeat the process until we isolated one instance .... clearly the instance on the right is still the most likely to be isolated.



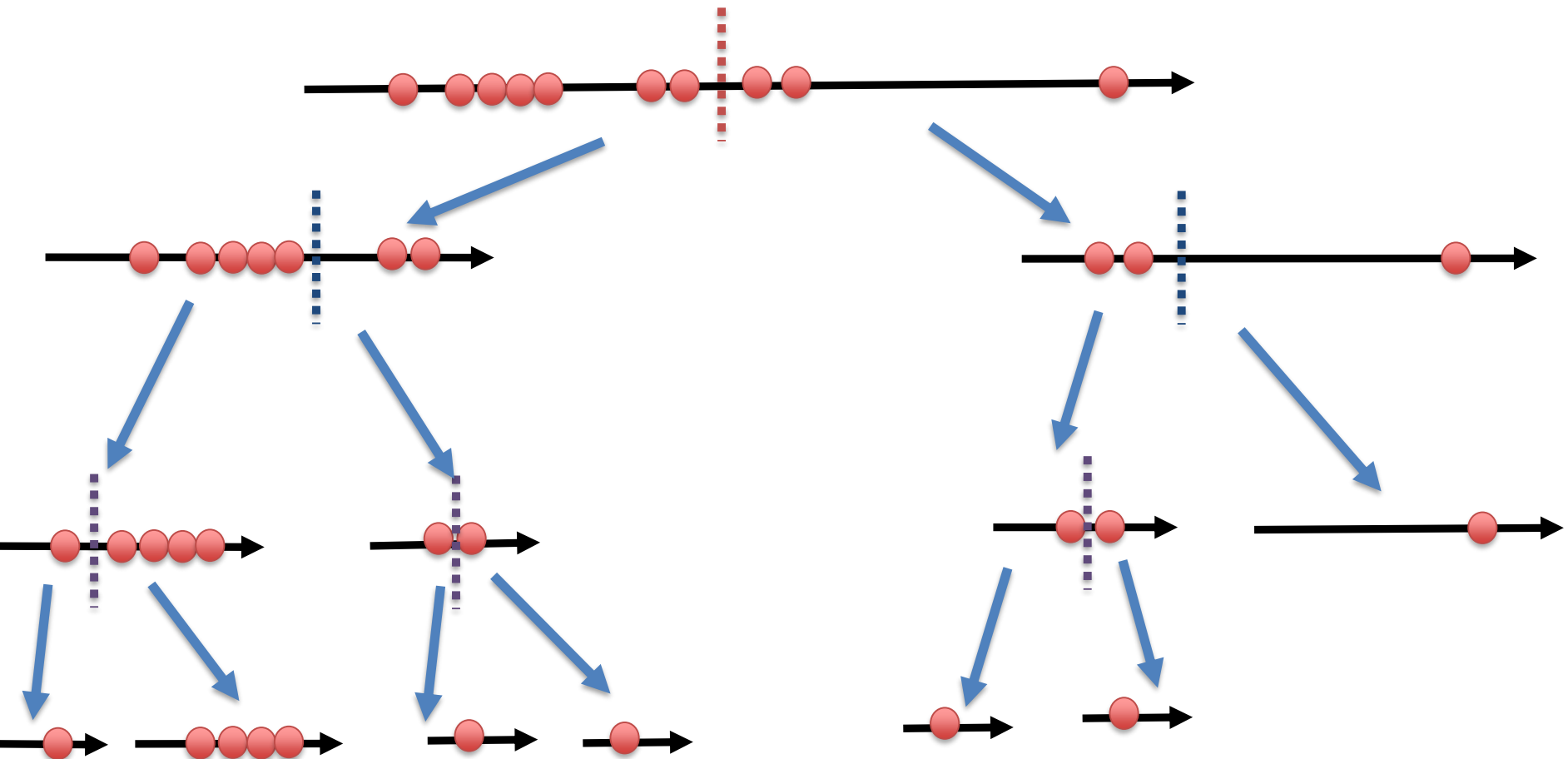
# Isolation Forests for Multi-variate Outlier Detection

Now let's say we were to repeat the process until we isolated one instance .... clearly the instance on the right is still the most likely to be isolated.



# Isolation Forests for Multi-variate Outlier Detection

Now let's say we were to repeat the process until we isolated one instance .... clearly the instance on the right is still the most likely to be isolated.

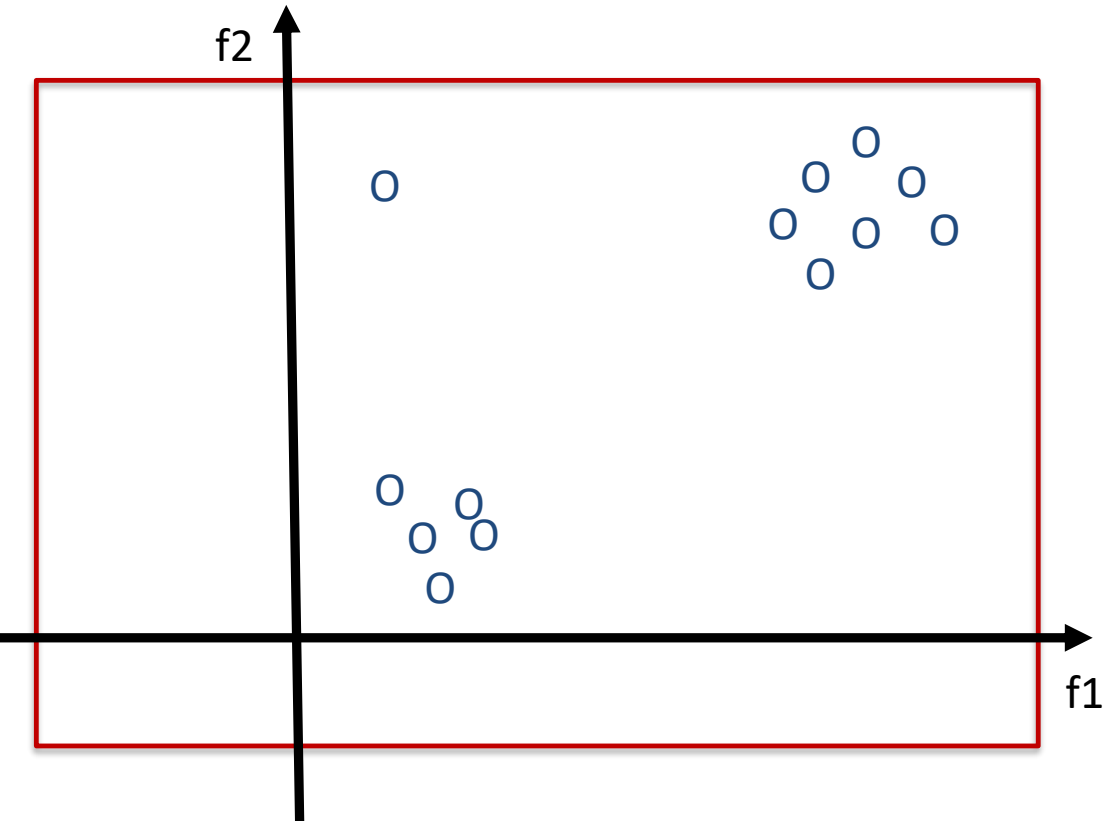


# Isolation Forests for Multi-variate Outlier Detection

Now let's consider a feature space with 2 features. An isolation forest will build many **isolation trees**.

To build each tree we will (1) **randomly select a feature** and will (2) **randomly select a value to partition** that feature. It will continue this process iteratively.

Below f1 is feature 1 and f2 is feature 2 from our dataset. We initially pick f2 at random.



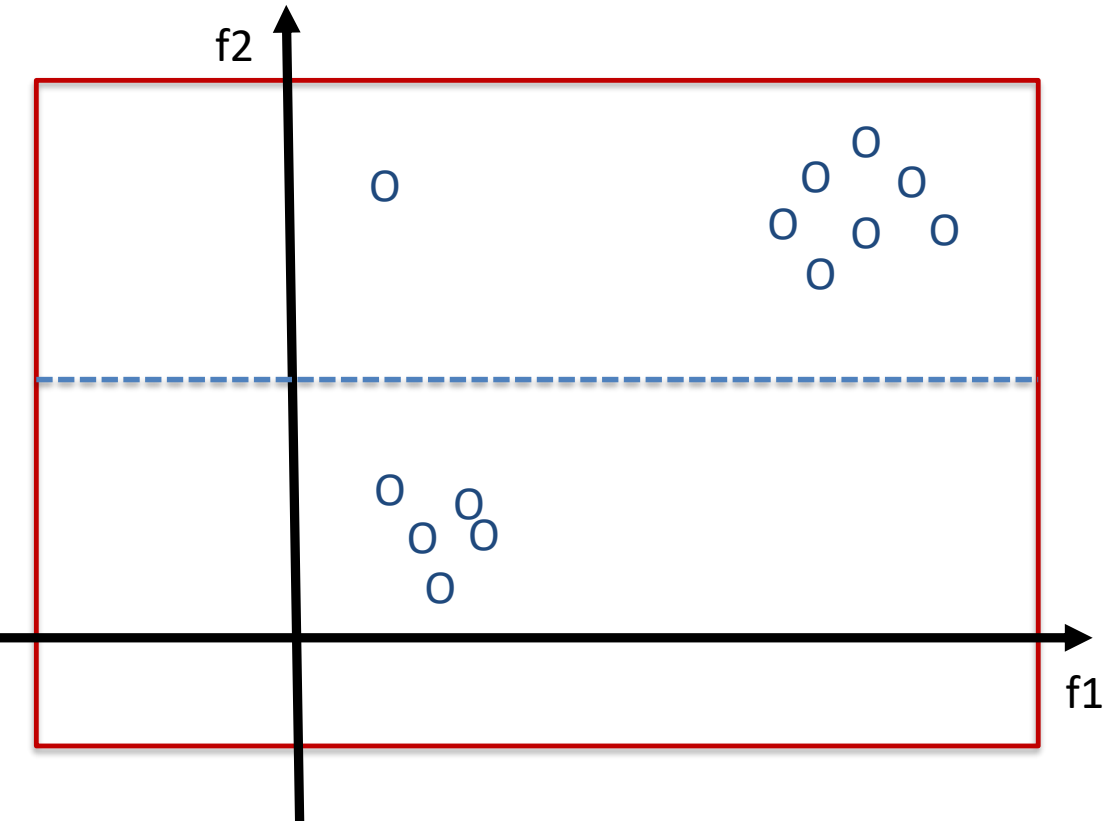


# Isolation Forests for Multi-variate Outlier Detection

Now let's consider a feature space with 2 features. An isolation forest will build many **isolation trees**.

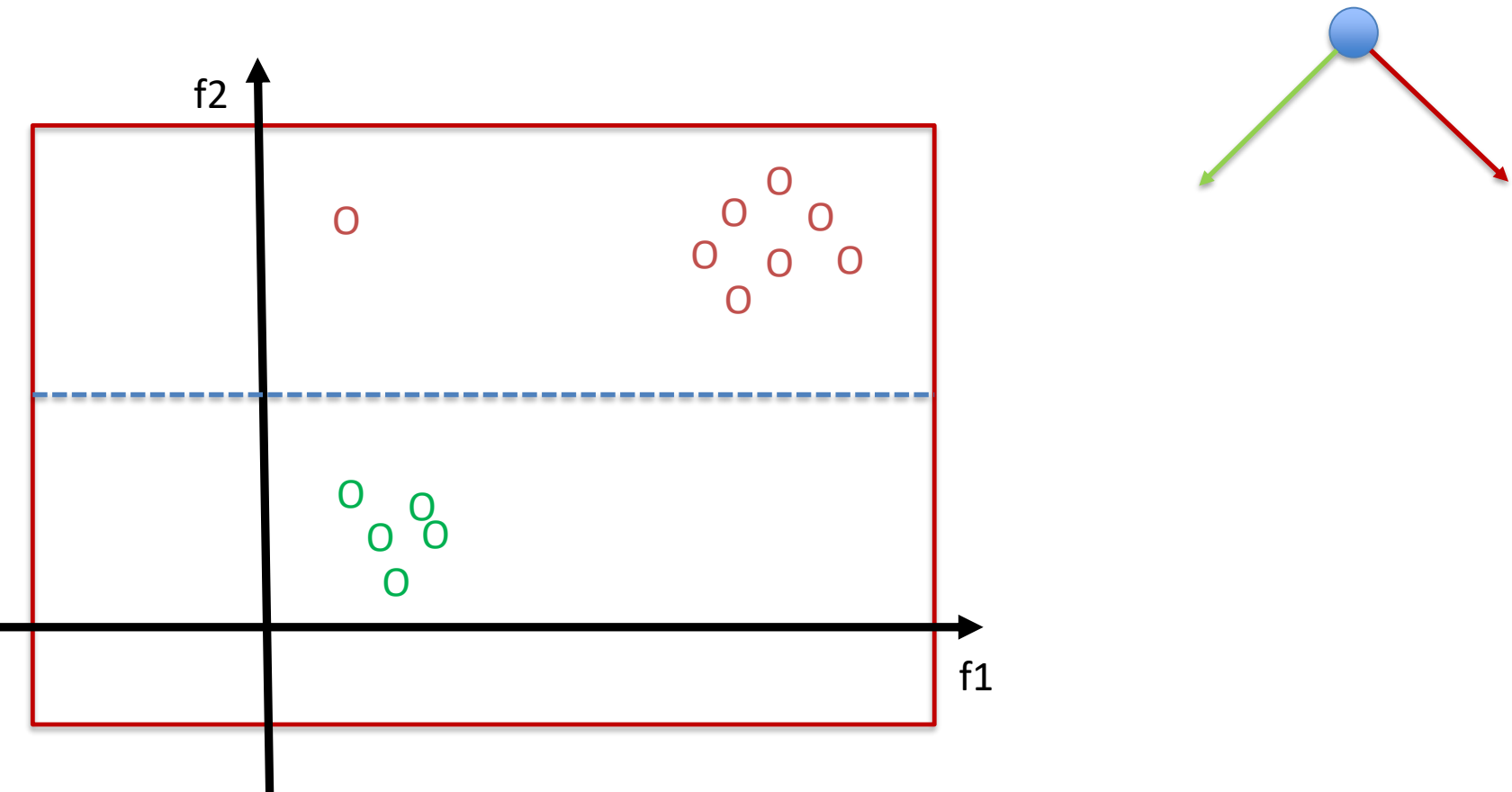
To build each tree we will (1) **randomly select a feature** and will (2) **randomly select a value to partition** that feature. It will continue this process iteratively.

Below f1 is feature 1 and f2 is feature 2 from our dataset. We initially pick f2 at random.



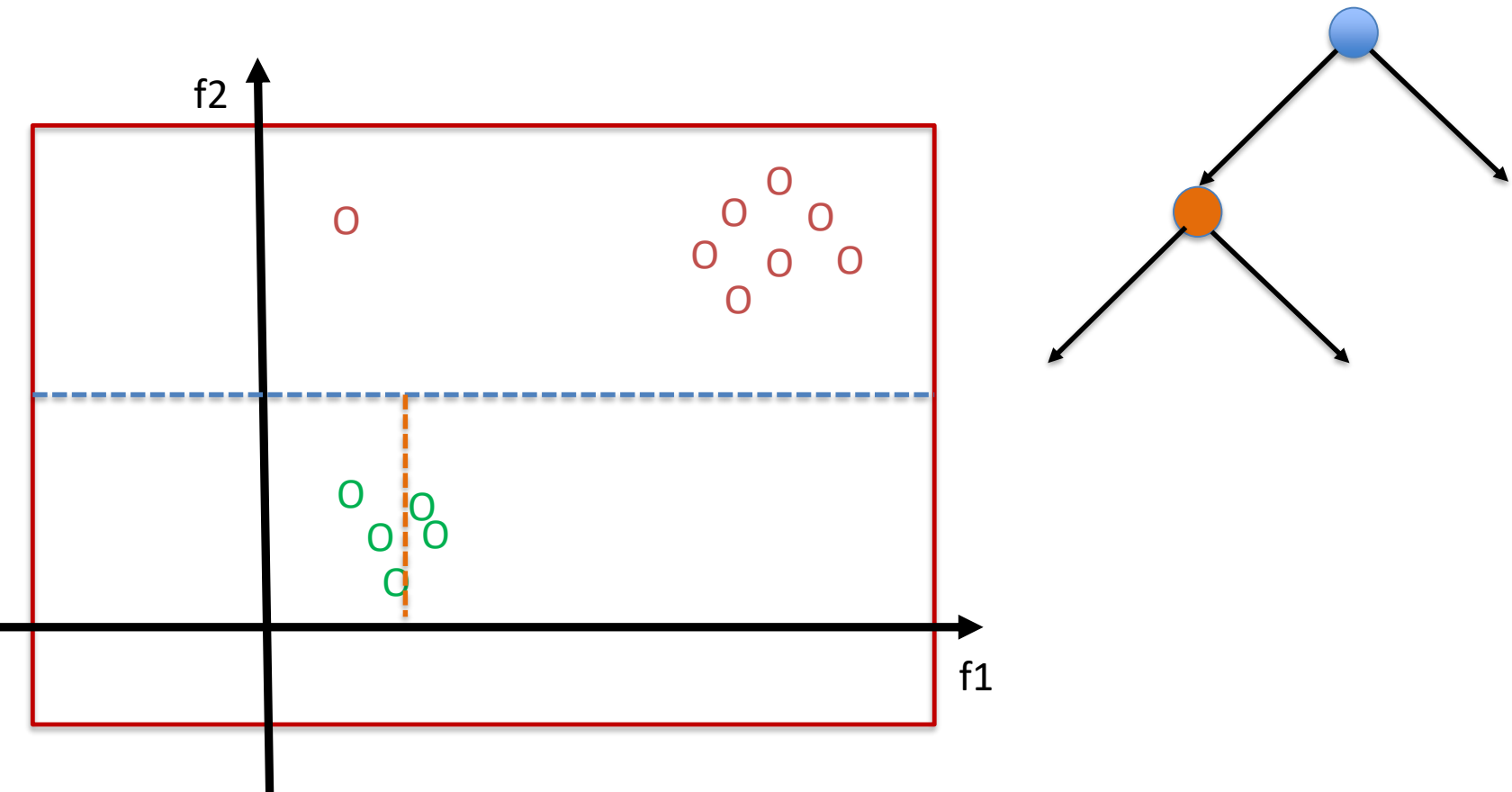
# Isolation Forests for Multi-variate Outlier Detection

We can visualize this process as a binary tree. Every instance that has a  $f_2$  value less than the select partition flows down one branch and every instance with an  $f_2$  value greater than the partition points flows down the other branch



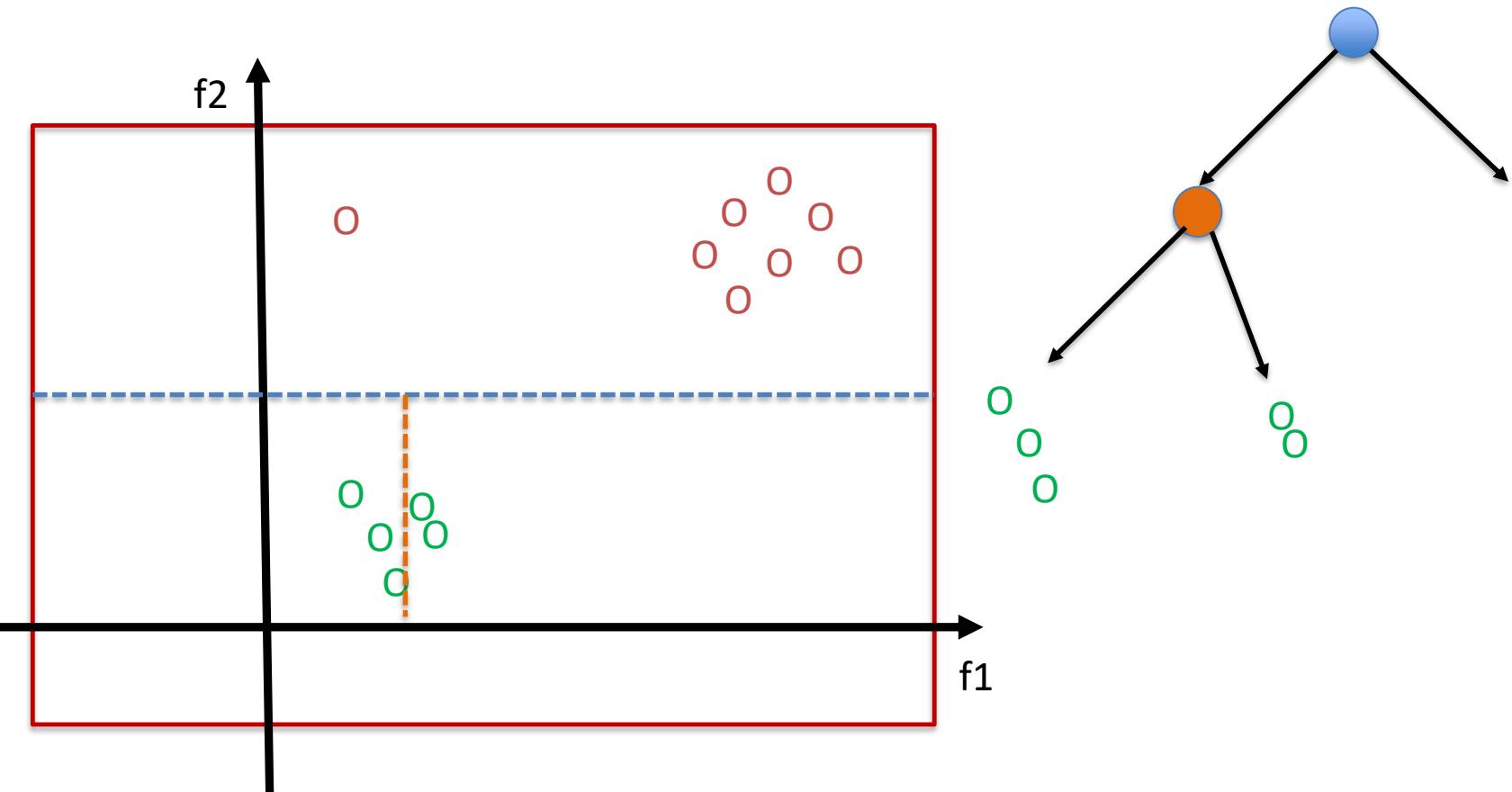
# Isolation Forests for Multi-variate Outlier Detection

This iterative process then continues. For all those instances that flowed down the left hand side of our graph, we again randomly select a feature (for simplicity we assume it is  $f_1$ ). Then we randomly partition the data using  $f_1$ .



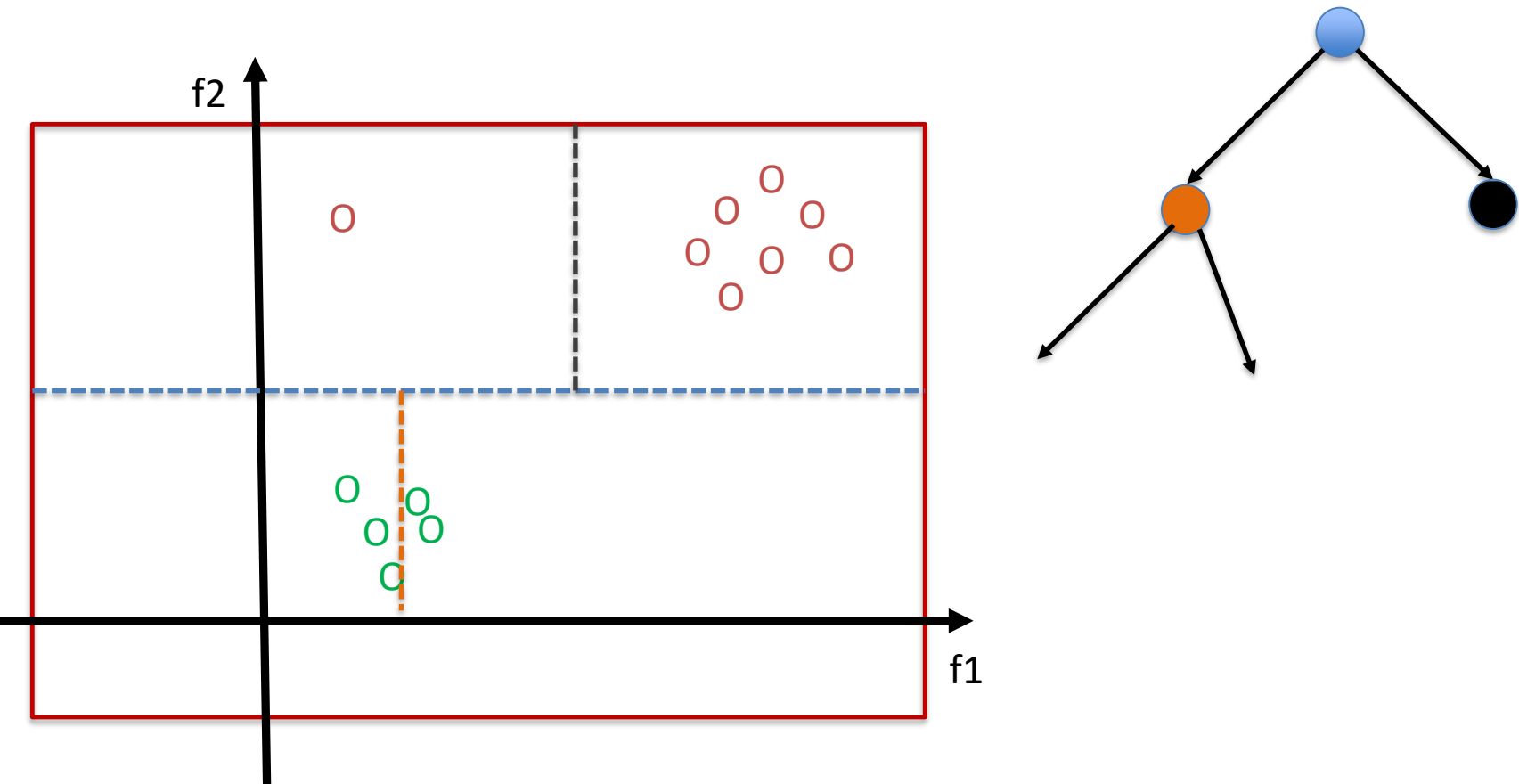
# Isolation Forests for Multi-variate Outlier Detection

This iterative process then continues. For all those instances that flowed down the left hand side of our graph, we again randomly select a feature (for simplicity we assume it is f1). Then we randomly partition the data using f1.



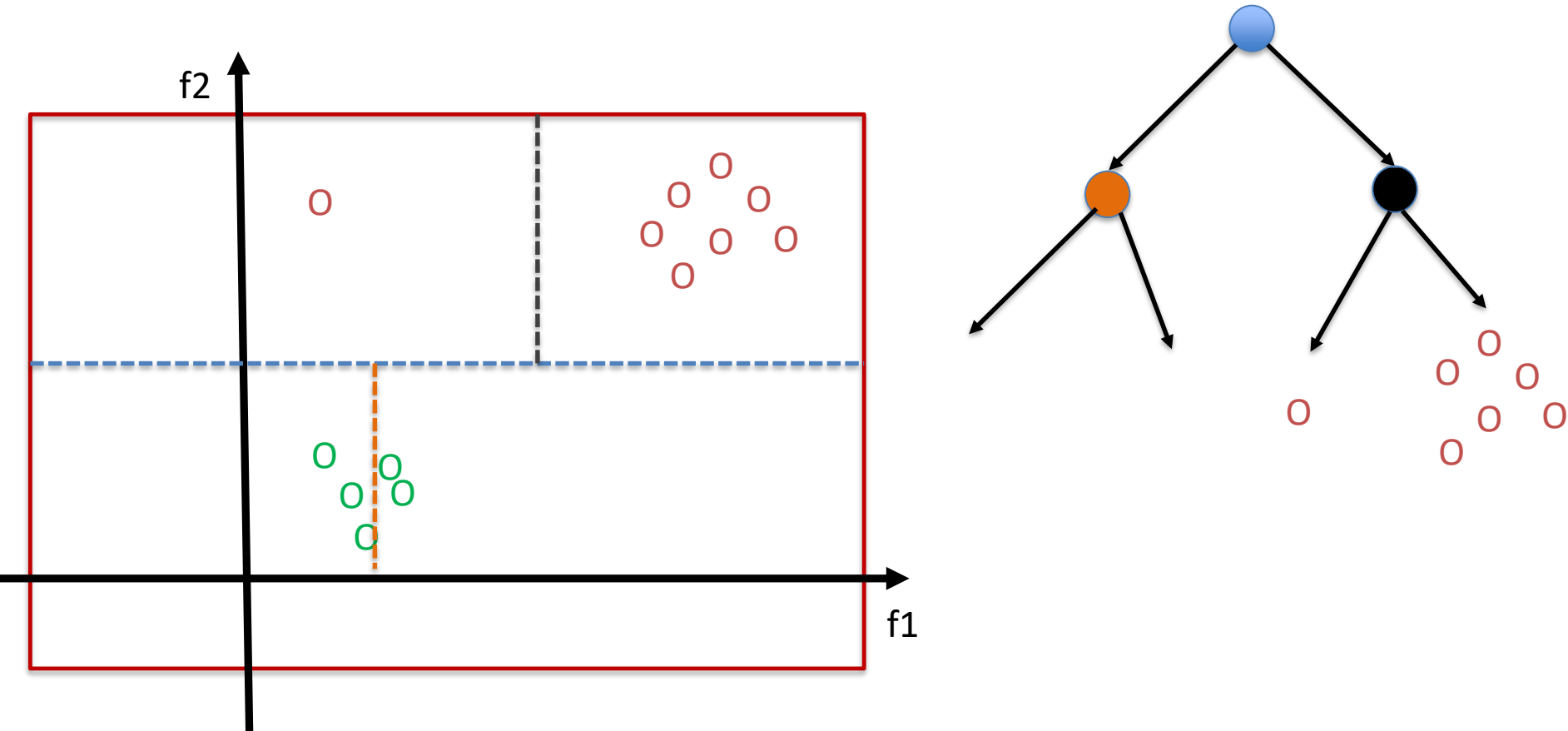
# Isolation Forests for Multi-variate Outlier Detection

We repeat the same process for all those that flow down the right hand side of the original partition. We again pick a random a feature (f1 selected) and partition the feature space randomly by f1.



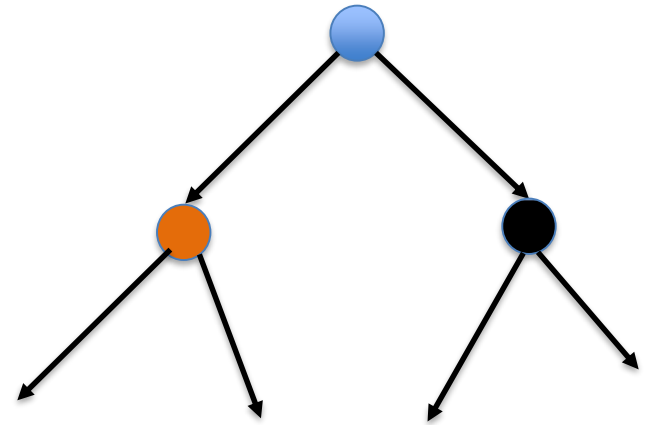
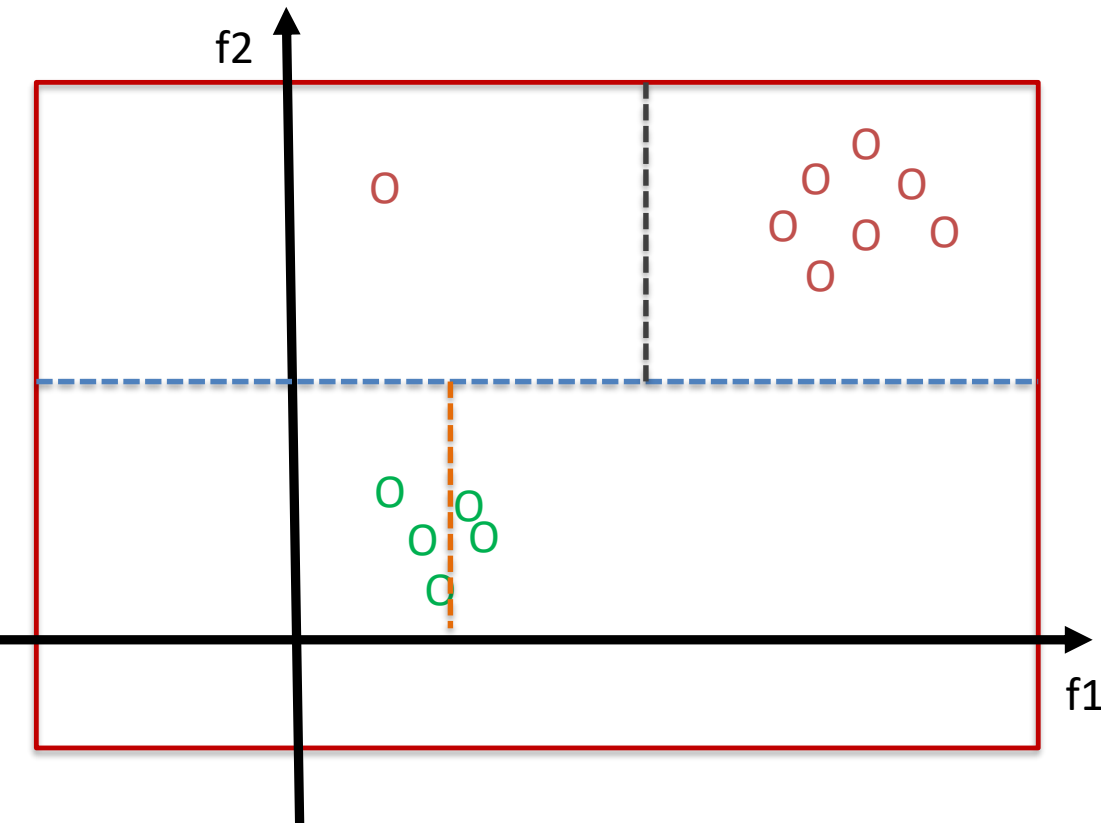
# Isolation Forests for Multi-variate Outlier Detection

Notice we have now reached a point in the tree, where a single instance is on it's own (notice that this instance is the outlier). The process of building the tree continues until we reach a point where a particular depth of the tree is reached or until everyone single instance is alone at the end of the branch.



# Isolation Forests for Multi-variate Outlier Detection

Notice we have now reached a point in the tree, where a single instance is on it's own (notice that this instance is the outlier). The process of building the tree continues until we reach a point where a particular depth of the tree is reached or until everyone single instance is alone at the end of the branch.

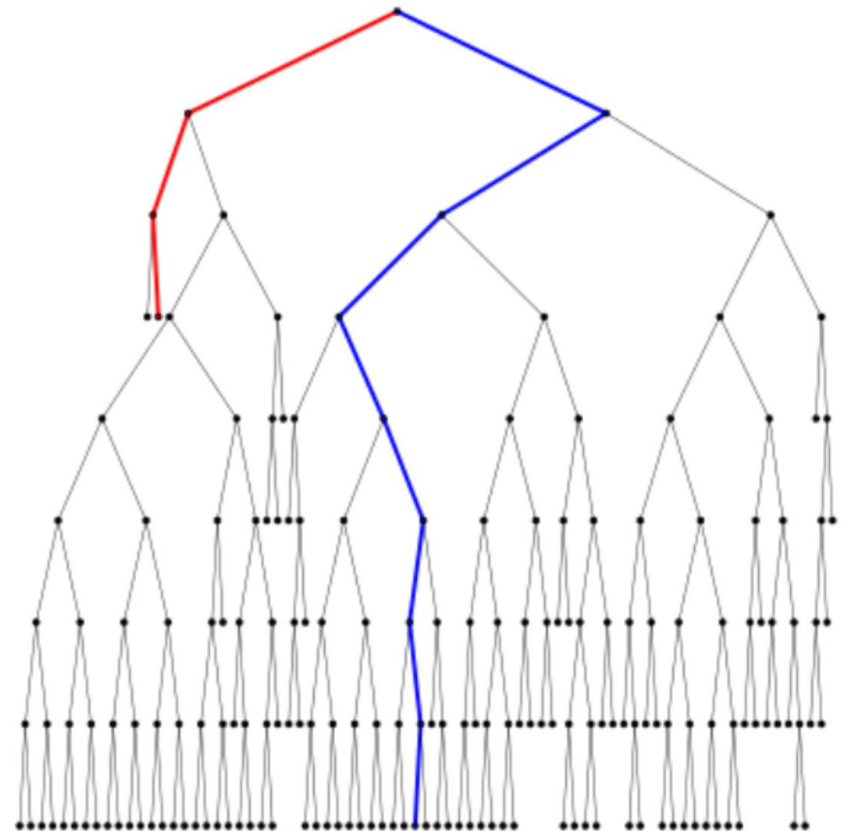


We refer to the tree above as an **isolation tree**. We continue growing the tree until all data points are isolated or until we have reached a particular maximum depth.

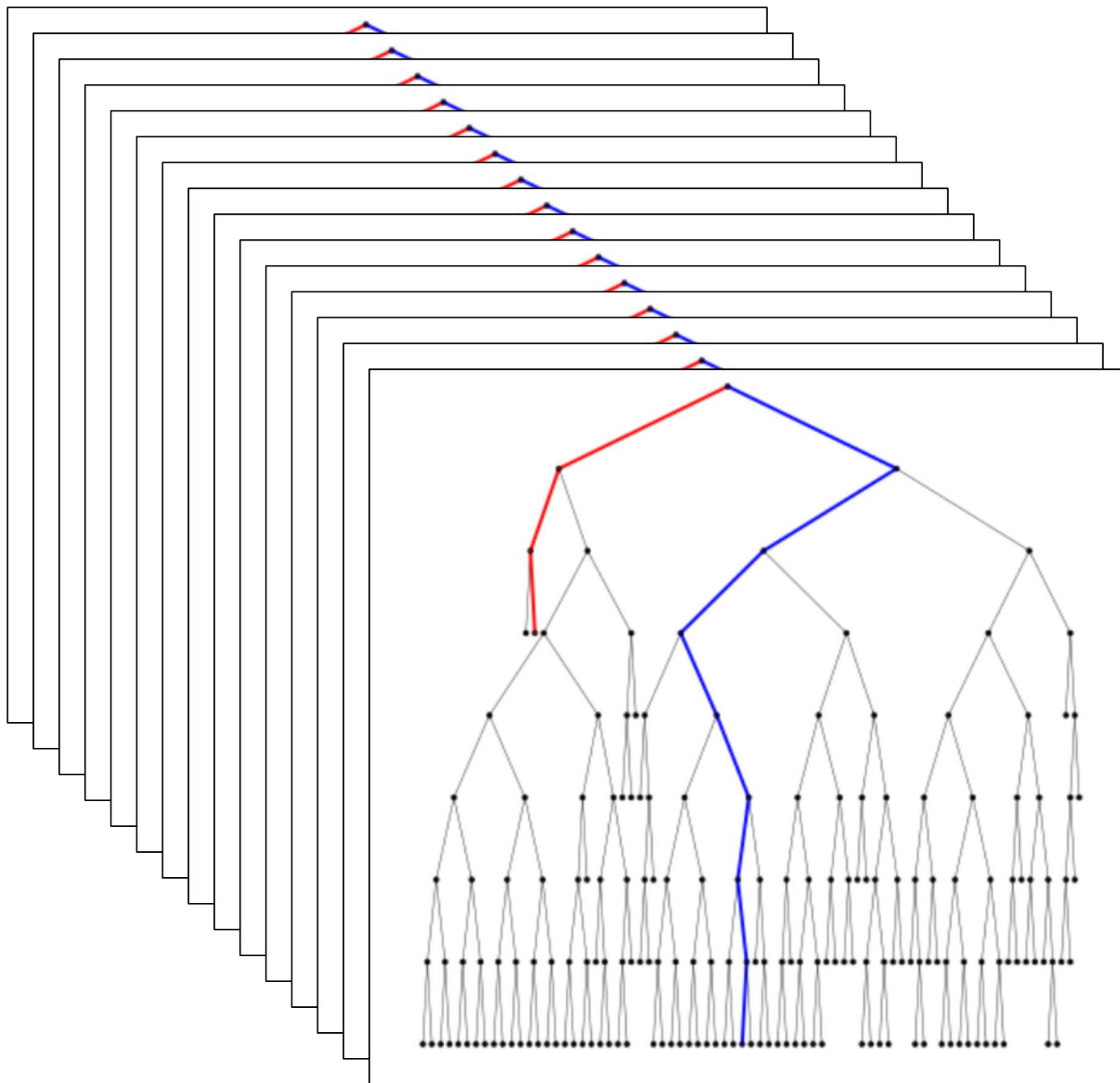


# Isolation Forests for Multi-variate Outlier Detection

- At the core of the Isolation Forest algorithm is the observation that outlier instances in a dataset are easier to separate (isolate) from the rest of the sample (isolate).
- Consider the single isolation tree on the right.
- The number of **edges on the red path is 3**, whereas the number of edges along the **blue path is 8**. The data point at the terminal of the red path is **more likely to be an outlier** compared to the data point at the terminal of the blue path.



<https://arxiv.org/pdf/1811.02141.pdf>



We build many isolation trees.

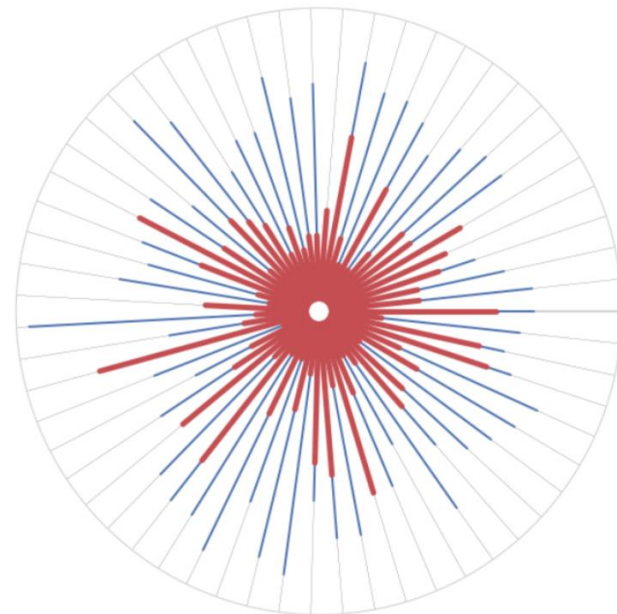
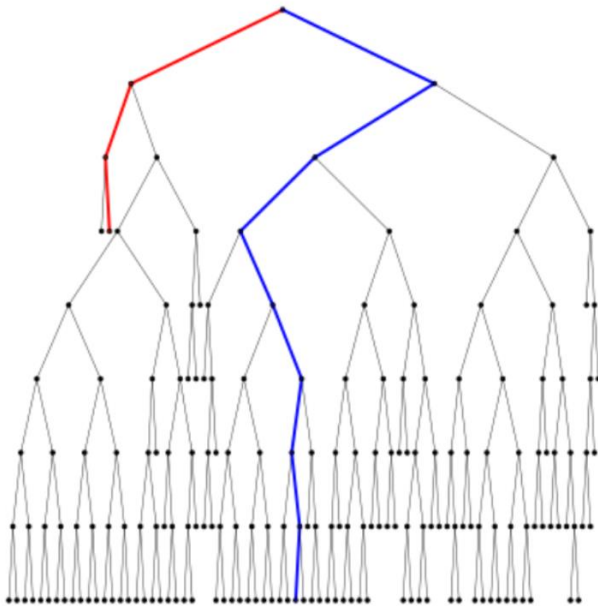
Each time we go to build a tree a random sample of the original data is extracted and the isolation tree is built using this subset.

The image below (on the right) depicts an isolation forest consisting of **60 isolation trees**. Each radial line represents one tree, while the outer circle represents the maximum depth limit.

The **red lines** represent the **depth** in the trees of the anomalous point, while the **blue lines** represent the **depths** of the normal point.

Clearly as can be seen, on average, the blue lines achieve a much larger radius than the red ones.

By creating many isolation trees, we can use the average depths of the branches to assign anomaly scores to each training instance.



# Isolation Forests for Multi-variate Outlier Detection

Each time we go to build a tree a random sample of the original data is extracted and the isolation tree is built using this subset.

---

**Algorithm**  $iForest(X, t, \psi)$

---

**Inputs:**  $X$  - input data,  $t$  - number of trees,  $\psi$  - sub-sampling size

**Output:** a set of  $t$  *iTrees*

- 1: **Initialize**  $Forest$
- 2: set height limit  $l = ceiling(\log_2 \psi)$
- 3: **for**  $i = 1$  to  $t$  **do**
- 4:    $X' \leftarrow sample(X, \psi)$
- 5:    $Forest \leftarrow Forest \cup iTree(X', 0, l)$
- 6: **end for**
- 7: **return**  $Forest$

# Isolation Forests for Multi-variate Outlier Detection

---

**Algorithm**  $iTree(X, e, l)$

---

**Inputs:**  $X$  - input data,  $e$  - current tree height,  $l$  - height limit

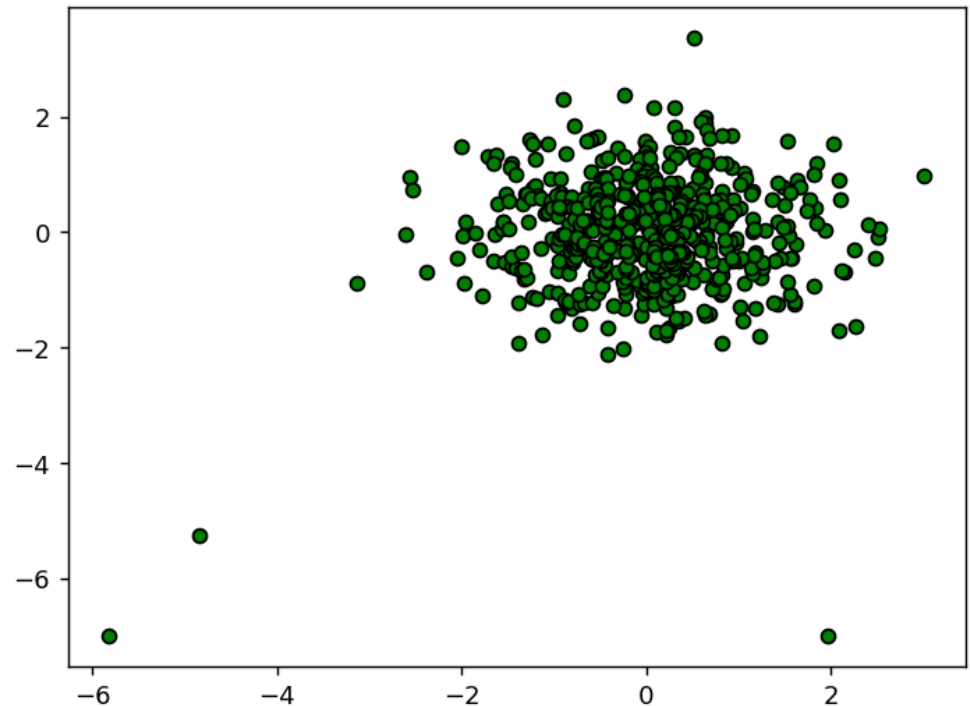
**Output:** an iTree

```
1: if  $e \geq l$  or  $|X| \leq 1$  then
2:   return  $exNode\{Size \leftarrow |X|\}$ 
3: else
4:   let  $Q$  be a list of attributes in  $X$ 
5:   randomly select an attribute  $q \in Q$ 
6:   randomly select a split point  $p$  from  $max$  and  $min$ 
     values of attribute  $q$  in  $X$ 
7:    $X_l \leftarrow filter(X, q < p)$ 
8:    $X_r \leftarrow filter(X, q \geq p)$ 
9:   return  $inNode\{Left \leftarrow iTree(X_l, e + 1, l),$ 
10:               $Right \leftarrow iTree(X_r, e + 1, l),$ 
11:               $SplitAtt \leftarrow q,$ 
12:               $SplitValue \leftarrow p\}$ 
13: end if
```

---

# Using an Isolation Forest in Scikit - Learn

- ▶ In this example, we will use Scikit Learn to apply an isolation forest.
- ▶ We will provide as input the dataset below defined by just two features.



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler

centers = [(5, 5)]
X, y = make_blobs(n_samples=500, n_features=2, cluster_std=0.5,
                  centers=centers, shuffle=False, random_state=42)
# create some outliers in feature space
X= np.vstack((X, [[2, 1], [6, 1], [2.5, 2]]))

scaler = StandardScaler().fit(X)
X = scaler.transform(X)

# fit the model
clf = IsolationForest(contamination = 0.01)
clf.fit(X)

results = clf.predict(X)
```



```
clf = IsolationForest(contamination = 0.01)  
clf.fit(X)
```

```
#predict returns an array contains 1 (not outlier) and -1 (outlier) values  
results = clf.predict(X)
```

```
outliers = X[results == -1]  
normal = X[results == 1]
```

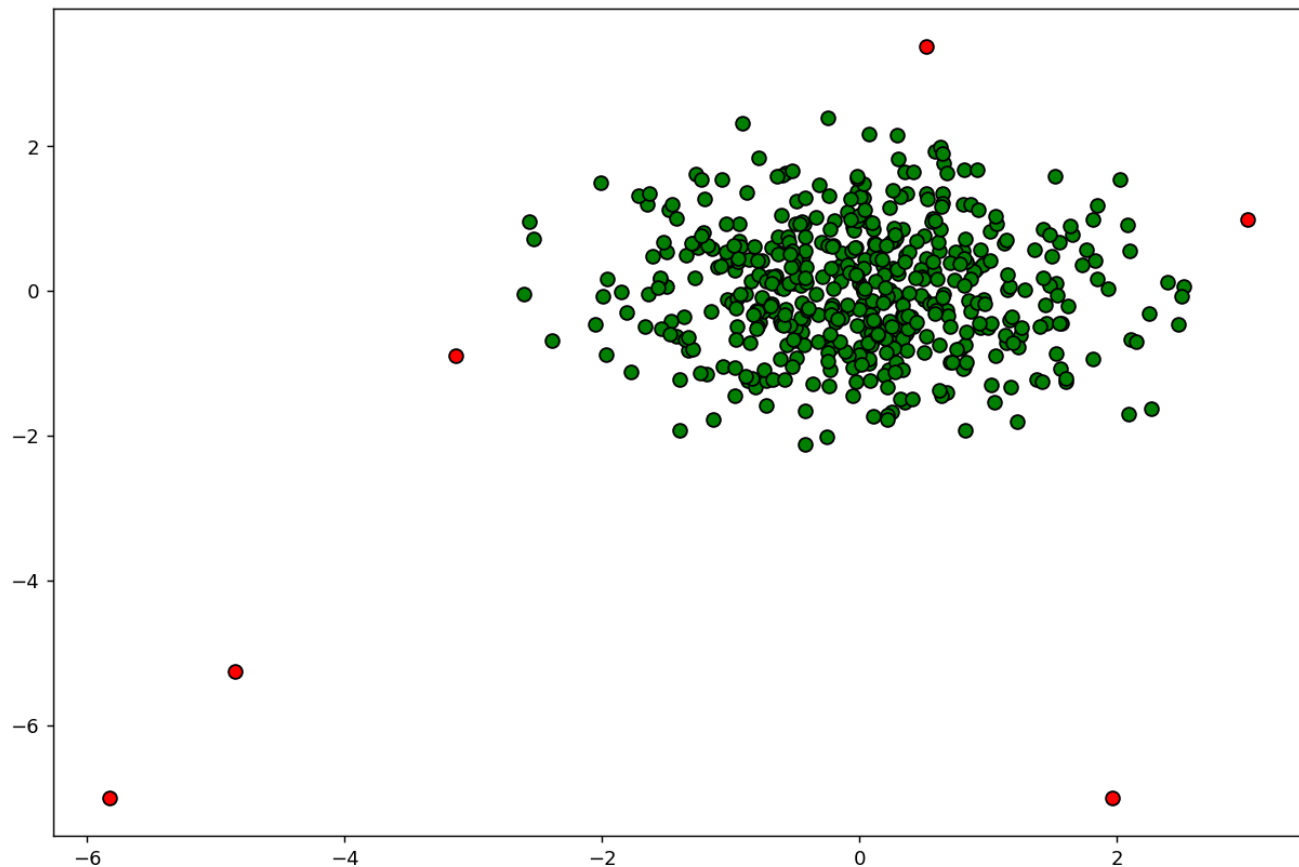
```
b1 = plt.scatter(outliers[:, 0], outliers[:, 1], c='red', s=30, edgecolor='k')  
b2 = plt.scatter(normal[:, 0], normal[:, 1], c='green', s=30, edgecolor='k')
```

```
clf = IsolationForest(contamination = 0.01)
clf.fit(X)
```

```
#predict returns an array contains 1 (not outlier) and -1 (outlier) values
results = clf.predict(X)
```

```
outliers =
normal =
```

```
b1 = plt.s
b2 = plt.s
```



```
clf = IsolationForest()
```

```
clf.fit(X)
```

```
#predict returns an array contains 1 (not outlier) and -1 (outlier) values
```

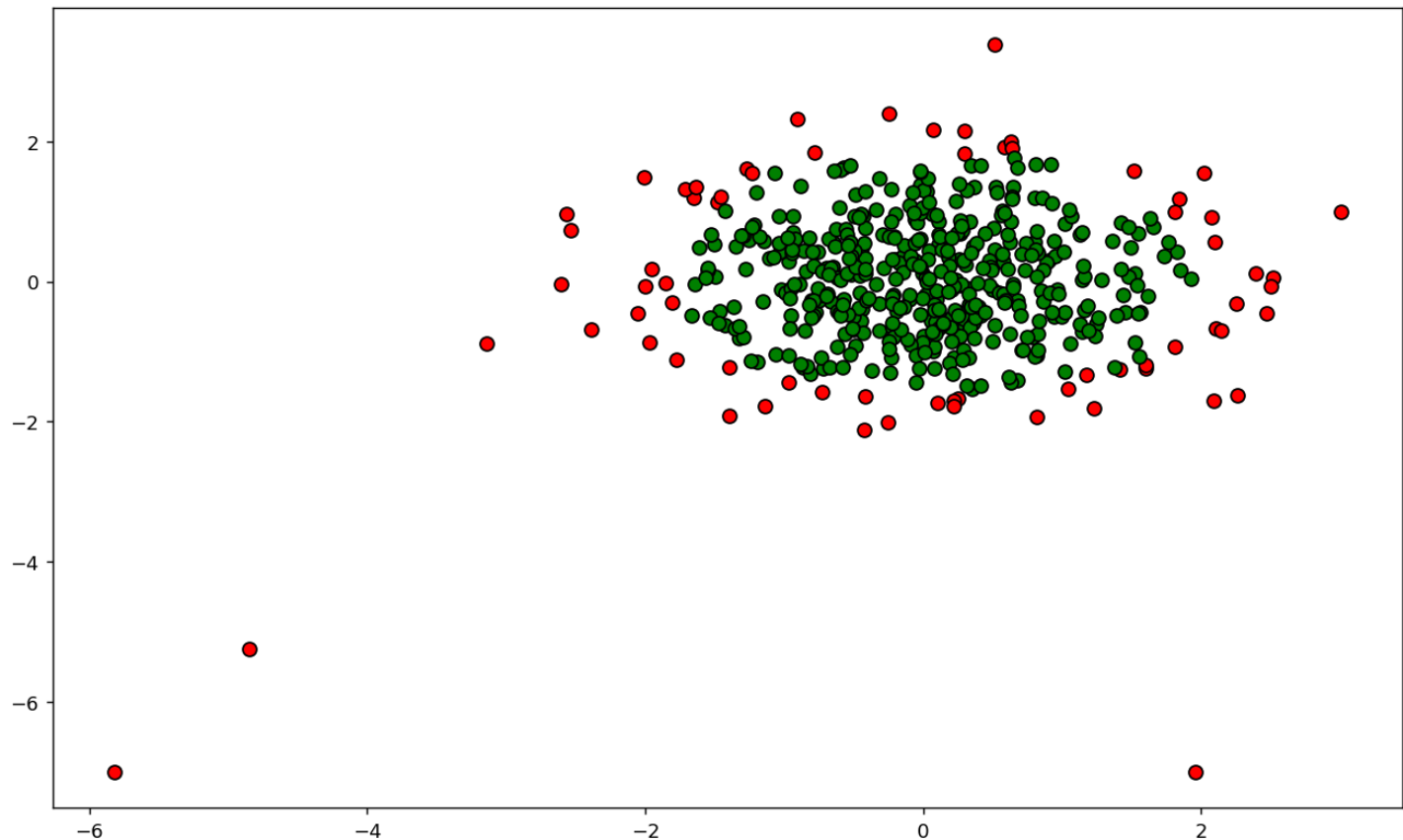
```
results = clf.predict(X)
```

```
outliers = X[results == -1]
```

```
normal = X[results == 1]
```

```
b1 = plt.scatter
```

```
b2 = plt.scatter
```



# Isolation Forests (Some Advice)

- With isolation forests the **contamination parameter** is an important hyper-parameter. It allows you to specify the anticipated percentage of outliers in the dataset.
- Typically the **number of outliers in a dataset is small**. Clearly this is dataset dependent but you may only have 1 -2 % of outliers.
- However, we **do not know the number of outliers** in advance. Contamination parameter can be treated as a parameter to be **optimized**. Be careful about identifying large number of outliers.
- An isolation forest is a machine learning algorithm therefore, it will typically have to take place after you have dealt with missing values, converted categorical features and performed scaling.

# Data Pre-processing for Scikit Learn

- ▶ Dealing with Outliers
- ▶ **Dealing with Missing Values**
- ▶ Handling Categorical Data
- ▶ Scaling Data
- ▶ Handling Imbalance
- ▶ Feature Selection

# Dealing with Missing Values

- ▶ It is common in real-world applications that some features are missing one or more values for various reasons. There could be an **error in the data collection process**, certain fields may have been **left blank in a survey, power outage**, etc.
- ▶ Most ML algorithms are unable to process these missing values and as such it is important that we deal with all missing values within the data before sending it to the ML algorithm.

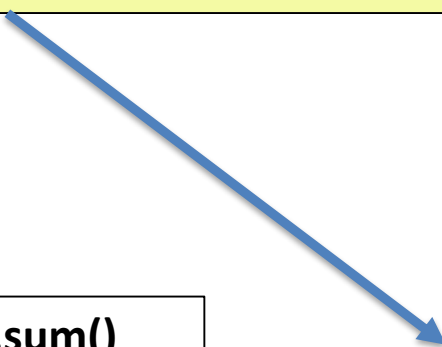
```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3), index=['a', 'b', 'c'])
seriesB = pd.Series(np.random.rand(4), index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(3), index=['b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                   'three' : seriesC})

print (df)
print (df.isnull().sum())
```

Notice the **.sum()** method will count the number of missing values in each column.



	one	three	two
a	0.376060	<b>NaN</b>	0.111221
b	0.400020	0.164076	0.548106
c	0.507972	0.325337	0.137571
d	<b>NaN</b>	0.823270	0.816618

one	1
three	1
two	0
dtype: int64	



# Dealing with Missing Values

- ▶ Please note that the call to **df.isnull().sum()** will only identify the missing values that are specified as **NaN**.
- ▶ Missing values may also be represented using dummy values such as '?', -1, -99, -999 in your dataset. Missing values such as these will not show up using `isnull()`. You should be able to identify these in your data exploration stage or when looking for outliers.
- ▶ Some of the methods we look at will allow you to specify how a missing values is presented in your dataset.
- ▶ However, the first method we will look at will only consider NaN value therefore, you may have to encode your missing values as NaN values.
- ▶ A simple way of dealing with this is by replacing the non-standard missing values with NaN using `df.replace`.

# Dealing with Missing Values

```
import pandas as pd
import numpy as np

seriesA = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesB = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesC = pd.Series([8, 1, '?', 43], index=['a', 'b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                   'three' : seriesC})

df = df.replace('?', np.NaN)

print(df)
print(df.isnull().sum())
```

In this case we use replace to replace any occurrence of '?' within the dataframe with NaN.

	one	two	three
a	12	12	8
b	4	4	1
c	3	3	?
d	?	?	43

	one	three	two
a	12	8	12
b	4	1	4
c	3	NaN	3
d	NaN	43	NaN

one	1
three	1
two	1
dtype:	int64

# Dealing with Missing Values - Removal

- ▶ One of the easiest ways to deal with missing values (although not always the most appropriate) is to simply remove the corresponding features (columns) or rows from the dataset entirely.
  - ▶ Rows
    - ▶ **df.dropna()** will remove any rows that contain a missing value.
    - ▶ **df.dropna(subset=['A'])** only drop rows where missing values appear in a specific column in this case column A.
    - ▶ **df.dropna(thresh=3)** the parameter thresh specifies the number of non-NAN values that a row must have in order to be retained.
  - ▶ Columns
    - ▶ **df.dropna(axis = 1)** will drop **columns** that have at least one missing value
    - ▶ if you want to drop a column of a specific name you can call **df.drop(['ColumnName'], axis=1)**
- ▶ Each of the above will return a separate copy of the dataset with the new changes (if you want the changes to take place on the current dataframe then you can set the parameter inplace=True)

```

import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3), index=['a', 'b', 'c'])
seriesB = pd.Series(np.random.rand(4), index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(3), index=['b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                   'three' : seriesC})

print (df)
print
newdf = df.dropna()
print (newdf)

```

Here we drop any rows from our dataframe that contain a missing value.

	one	three	two
a	0.867059	<b>NaN</b>	0.255192
b	0.722719	0.420534	0.212348
c	0.328197	0.141678	0.237098
d	<b>NaN</b>	0.458063	0.503182

	one	three	two
b	0.722719	0.420534	0.212348
c	0.328197	0.141678	0.237098

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(np.random.rand(5))
seriesD = pd.Series(np.random.rand(7))

df = pd.DataFrame({'one' : seriesA,
                  'two' : seriesB,
                  'three' : seriesC,
                  'four' : seriesD})

print (df)
df = df.dropna(thresh=4, axis=1 )
print (df)
```

	four	one	three	two
0	0.766476	0.909878	0.476737	0.084872
1	0.810370	0.285238	0.386073	0.268438
2	0.567595	0.162616	0.213230	0.389272
3	0.958997	NaN	0.579480	0.689228
4	0.141135	NaN	0.986758	NaN
5	0.612904	NaN	NaN	NaN
6	0.193091	NaN	NaN	NaN

```

import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(np.random.rand(5))
seriesD = pd.Series(np.random.rand(7))

df = pd.DataFrame({'one' : seriesA,
                  'two' : seriesB,
                  'three' : seriesC,
                  'four' : seriesD})

print (df)
df = df.dropna(thresh=4, axis=1 )
print (df)

```

Notice above the threshold value is 4. Therefore, a column must have four or more non-NA values in order to be retained.

	four	one	three	two
0	0.766476	0.909878	0.476737	0.084872
1	0.810370	0.285238	0.386073	0.268438
2	0.567595	0.162616	0.213230	0.389272
3	0.958997	NaN	0.579480	0.689228
4	0.141135	NaN	0.986758	NaN
5	0.612904	NaN	NaN	NaN
6	0.193091	NaN	NaN	NaN

	four	three	two
0	0.766476	0.476737	0.084872
1	0.810370	0.386073	0.268438
2	0.567595	0.213230	0.389272
3	0.958997	0.579480	0.689228
4	0.141135	0.986758	NaN
5	0.612904	NaN	NaN
6	0.193091	NaN	NaN

# Dealing with Missing Values

- ▶ Although the removal of missing values may seem convenient it also comes with **significant disadvantages**.
- ▶ If you delete a row that has a missing value then you could potentially be **deleting useful feature information**.
- ▶ In many cases you may not have an abundance of data and we may end up removing many samples which could in turn **reduce the accuracy** of our model.
- ▶ An alternative (which is typically more preferable) to use **imputation** techniques to estimate the missing values from the other training samples in our dataset.
- ▶ One of the most common imputation techniques is **mean imputation** where we replace a missing value with the mean of the data items in that column.

# Dealing with Missing Values

- ▶ Scikit learn provides an easy way of applying this method by using the [SimpleImputer](#) class.
- ▶ When creating a SimpleImputer object we can specify the **strategy** used for imputing missing values. The most typical is **strategy = mean**, however you can also impute by specifying **strategy = median**, **strategy = most\_frequent** (mode) or **strategy = constant**.
- ▶ The **most\_frequent** strategy replaces the missing value by the most frequent values. This is a strategy that is useful to use for **categorical** features.
- ▶ Another feature of the SimpleImputer is that we can **specify the missing value** using the parameter **missing\_values** (np.nan by default).



```

from sklearn.impute import SimpleImputer
# note we are using the dataframe we defined at the start of this section

print (df)

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')

imputer.fit(df)

allValues = imputer.transform(df)

print (allValues)

```

In this case we can impute for an entire dataset by passing in a pandas **dataframe** (we could also pass in a **NumPy** array).

The array we pass in must be a numerical array. The transform operation will always returns a **NumPy** array.

	one	three	two
a	0.030666	NaN	0.921680
b	0.351147	0.740355	0.478344
c	0.449259	0.299911	0.937952
d	NaN	0.897354	0.168368

```

[[ 0.03066623  0.64587346  0.92168033]
 [ 0.3511469   0.7403552   0.47834361]
 [ 0.4492592   0.29991101  0.93795242]
 [ 0.27702411  0.89735416  0.16836778]]

```

```
from sklearn.impute import SimpleImputer
# note we are using the dataframe we defined at the start of this section

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')

imputer.fit(df)

allValues = imputer.transform(df)

print (allValues)

df = pd.DataFrame(data = allValues, columns = df.columns)

print(df)
```

Notice that above we **convert the NumPy array back into a Pandas dataframe** using **pd.DataFrame**

```
[[ 0.13159694  0.57903764  0.39386208]
 [ 0.73002787  0.92954245  0.16350405]
 [ 0.67807006  0.58945908  0.60437333]
 [ 0.51323162  0.21811138  0.91419672]]
```

	one	three	two
0	0.131597	0.579038	0.393862
1	0.730028	0.929542	0.163504
2	0.678070	0.589459	0.604373
3	0.513232	0.218111	0.914197

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
```

```
seriesA = pd.Series(np.random.rand(5))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(["Blue", "Blue", "Yellow", "Brown"])
```

```
df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                   'three' : seriesC})
```

```
print (df)
```

	one	two	three
0	0.418927	0.734337	Blue
1	0.589525	0.347585	Blue
2	0.863817	0.297105	Yellow
3	0.002752	0.101646	Brown
4	0.745341	NaN	NaN

In the example on the next slide we will impute for missing values for one specific numerical column and one categorical column

## continued from previous slide

```
imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')
```

```
imputer.fit( df[['two']] )
```

```
df['two'] = imputer.transform( df[['two']] )
```

```
imputer = SimpleImputer(missing_values = np.NaN, strategy="most_frequent")
```

```
imputer.fit( df[['three']] )
```

```
df['three'] = imputer.transform( df[['three']] )
```

```
print (df)
```

	one	two	three
0	0.418927	0.734337	Blue
1	0.589525	0.347585	Blue
2	0.863817	0.297105	Yellow
3	0.002752	0.101646	Brown
4	0.745341	<b>0.370168</b>	<b>Blue</b>

# Multivariate Feature Imputation

- ▶ A new (and very welcome) addition to Scikit Learn is a method of multi-variate feature imputation called [IterativeImputer](#).
- ▶ It builds a separate model that takes in a **set of features** from the dataset and uses those to **predict the values** for the feature with the **missing values**.
- ▶ At each step [IterativeImputer](#) will:
  - ▶ **Select a feature column** as the target output  $y$  and the other feature columns are treated as inputs features  $X$ .
  - ▶ A **regression model** is fit on  $(X, y)$  for known  $y$ .
  - ▶ Then, the regression model is used to **predict the missing values of  $y$** . This is done for each feature in an iterative fashion.
- ▶ Then is repeated for `max_iter` imputation rounds.

## Note from Scikit Learn Website

This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_iterative_imputer`.

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	NaN
4	0.570394	NaN	NaN

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	<b>0.5680</b>
4	0.570394	<b>0.73472</b>	<b>0.5680</b>

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	NaN
4	0.570394	NaN	NaN

**Build model for predicting missing values in column two using the other columns as the feature (by default we start with the column with the least missing values)**

	one	three
0	0.883211	0.604766
1	0.072465	0.297758
2	0.958505	0.001842
3	0.923956	<b><u>0.5680</u></b>

	two
0	0.281865
1	0.395314
2	0.757694
3	0.373335

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	<b>0.5680</b>
4	0.570394	<b>0.73472</b>	<b>0.5680</b>

Notice, we only take those rows where there isn't a missing value in column 2.

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	NaN
4	0.570394	NaN	NaN

**Build model for predicting missing values in column two using the other columns as the feature**

	one	three
0	0.883211	0.604766
1	0.072465	0.297758
2	0.958505	0.001842
3	0.923956	<u>0.5680</u>

	two
0	0.281865
1	0.395314
2	0.757694
3	0.373335

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	0.5680
4	<b>0.570394</b>	0.73472	<b>0.5680</b>

(0.570394, 0.5680)

**Model**

**Predict missing values for column 2**



**Build model for predicting missing values in column two using the other columns as the feature**

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	NaN
4	0.570394	NaN	NaN

	one	three
0	0.883211	0.604766
1	0.072465	0.297758
2	0.958505	0.001842
3	0.923956	<u>0.5680</u>

	two
0	0.281865
1	0.395314
2	0.757694
3	0.373335

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	0.5680
4	0.570394	0.73472	0.5680

(0.570394, 0.5680)

**Model**

**Predict missing values for column 2**

We replace the NaN value for feature 2 with the predicted value. This process continues we will build a model for feature 3 next. It is then repeat all over again for a certain number of iterations (10 by default)

```
import pandas as pd
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
```

```
np.random.seed(0)
seriesA = pd.Series(np.random.rand(5))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(np.random.rand(3))

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                   'three' : seriesC})
```

```
print (df)
```

```
imp = IterativeImputer(random_state=0)
allValues = imp.fit_transform(df)
```

```
df = pd.DataFrame(data = allValues, columns = df.columns)
```

```
print(df)
```

	one	two	three
0	0.548814	0.645894	0.383442
1	0.715189	0.437587	0.791725
2	0.602763	0.891773	0.528895
3	0.544883	0.963663	NaN
4	0.423655	NaN	NaN

	one	two	three
0	0.548814	0.645894	0.383442
1	0.715189	0.437587	0.791725
2	0.602763	0.891773	0.528895
3	0.544883	0.963663	0.472460
4	0.423655	0.797768	0.482262

# Missing Values - Advice

- ▶ Typically if there is **50%** or more missing values from a feature, then that feature would commonly be **removed** entirely.
- ▶ One alternative to deleting a feature that suffers from such a large number of missing values is to create a new **separate binary feature** that indicates if there is a missing value in the original column or not.
- ▶ This approach can be useful if the reason for the original missing value has some relationship to the target variable.
- ▶ For example, if the missing feature may contain some sensitive personal information, a persons willingness to provide this data may tell us something about that person that might be related to the final target class.
- ▶ Typically the binary feature replaces the original feature.



# Missing Values - Advice

- ▶ As we mentioned previously, **deleting instances (rows)** that contains one or more missing values can result in a very large amount of data being lost. Unless you have an **abundance of data** and a relatively **small amount of missing values** I would not advice employing brute force deletion of rows.
- ▶ Imputation is a useful technique but it is not advisable when features have a very large number of missing values.
- ▶ A commonly observed rule is that **imputation** is not advisable on any feature that has **30% or more missing values** and should never be used on a feature that has **50%** or more missing values.