

COMP41680

Classification and Evaluation

Derek Greene

UCD School of Computer Science
Spring 2023

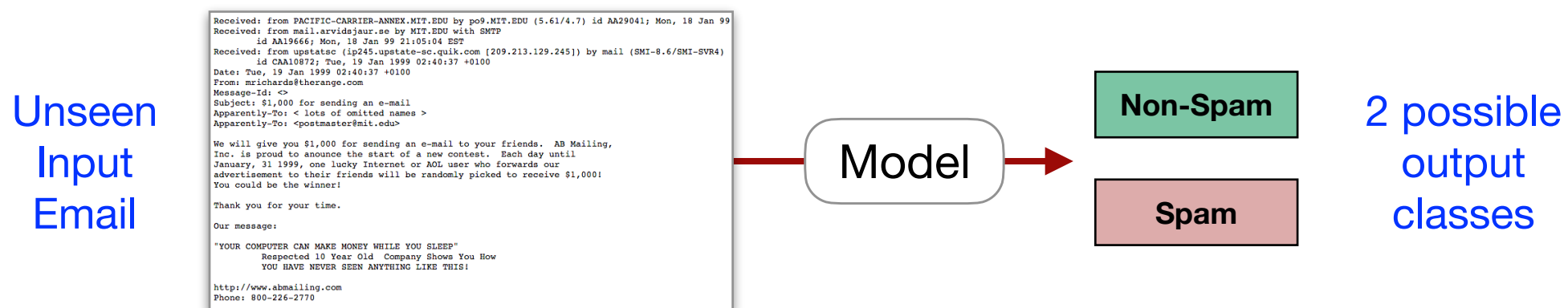


Modelling and Prediction Tasks

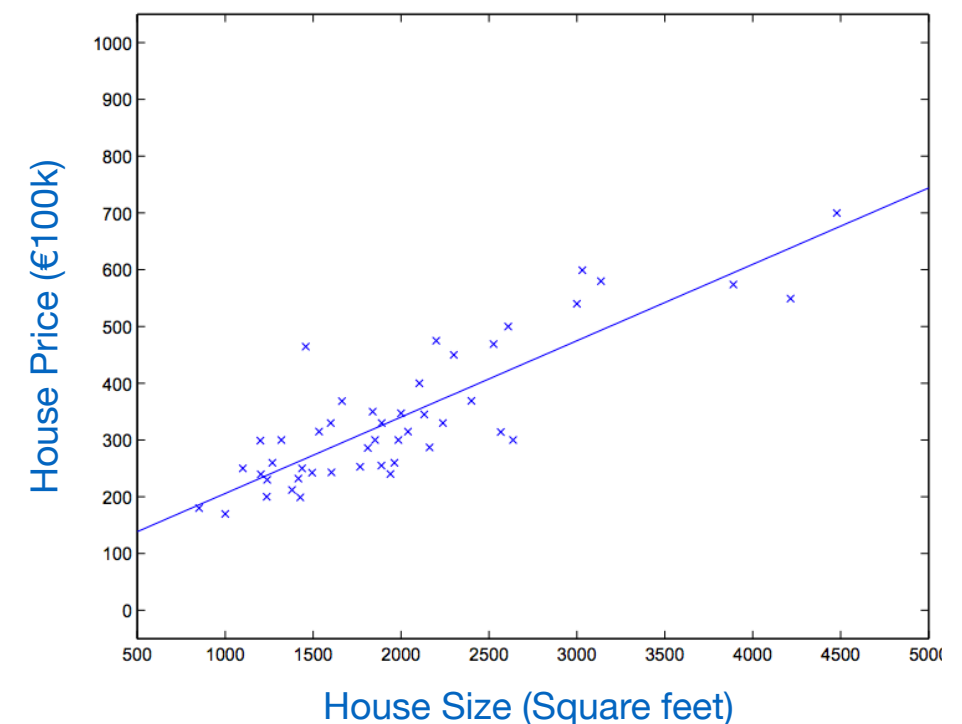
- **Predictive modelling** uses statistics to predict outcomes, based on historic data. Also referred to as **supervised machine learning**.
- **Examples:**
 - Spam filtering: predict if a new email is spam or non-spam, based on annotated examples of past spam / non-spam.
 - Car insurance: assign risk of accidents to policy holders and potential customers.
 - Healthcare: predict disease which a patient has, based on their symptoms.
 - Algorithmic trading: predictive models can be built for different assets like stocks, futures, currencies, etc, based on historic data and company information.

Reminder: Supervised Learning

- **Classification:** Learn from a labelled training set to make a prediction to assign a new "unseen" input example to one of a fixed number of classes (i.e. the output is a class).

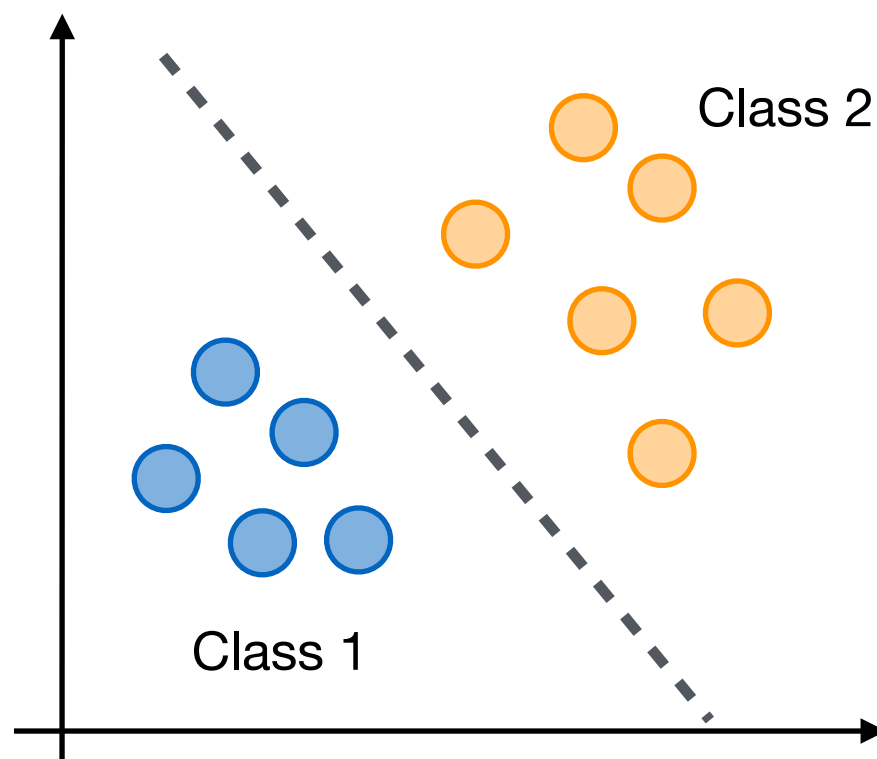


- **Regression:** Learn from an existing training set to decide the value of a continuous output variable (i.e. the output is a number).

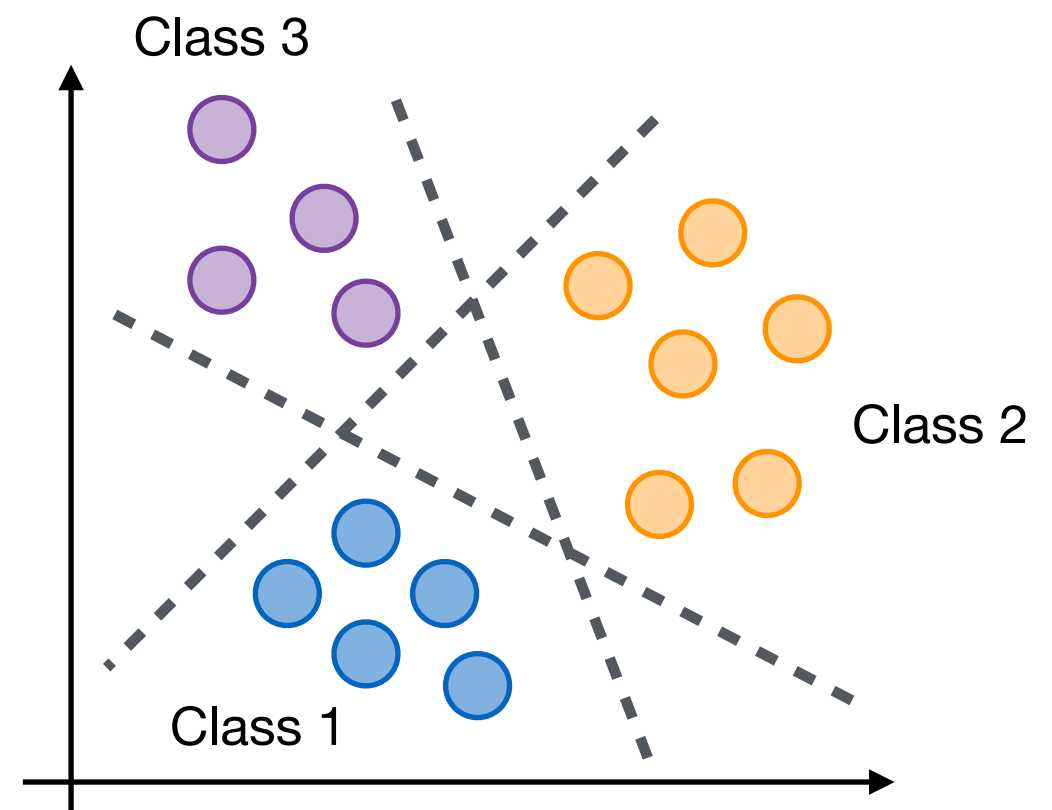


Types of Classification Tasks

- **Binary Classification:** Assign input examples into one of two different classes.
- **Multiclass Classification:** Assign input examples into one of three or more different classes.



Binary Classification



Multiclass Classification

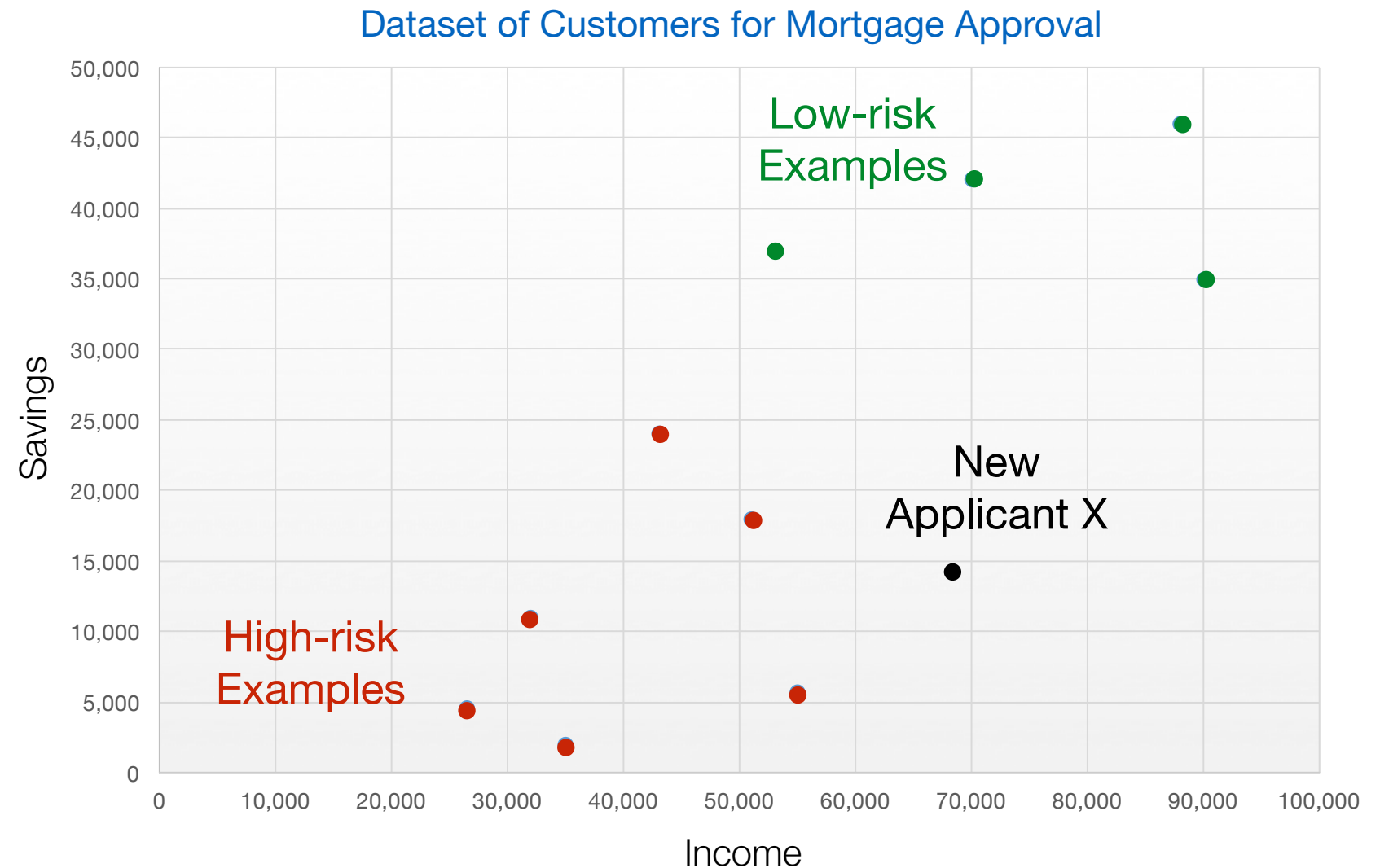
Binary Classification

- Binary classification tasks typically involve one class that represents the state of interest (the **positive class**) and one class which represents the "normal" state (the **negative class**).
- Example applications of binary classification:

Application	Positive Class	Negative Class
Spam email filtering	Email is spam	Email is non-spam
Medical diagnostics	Patient has disease X	Patient does not have disease X
Customer churn	Customer moves to a competitor	Customer does <u>not</u> move to a competitor
Conversion rate	Customer buys a product after seeing an advert	Customer does <u>not</u> buy a product after seeing an advert
Mortgage approval	Customer is <u>high risk</u> for a mortgage	Customer is <u>low risk</u> for a mortgage

Binary Classification

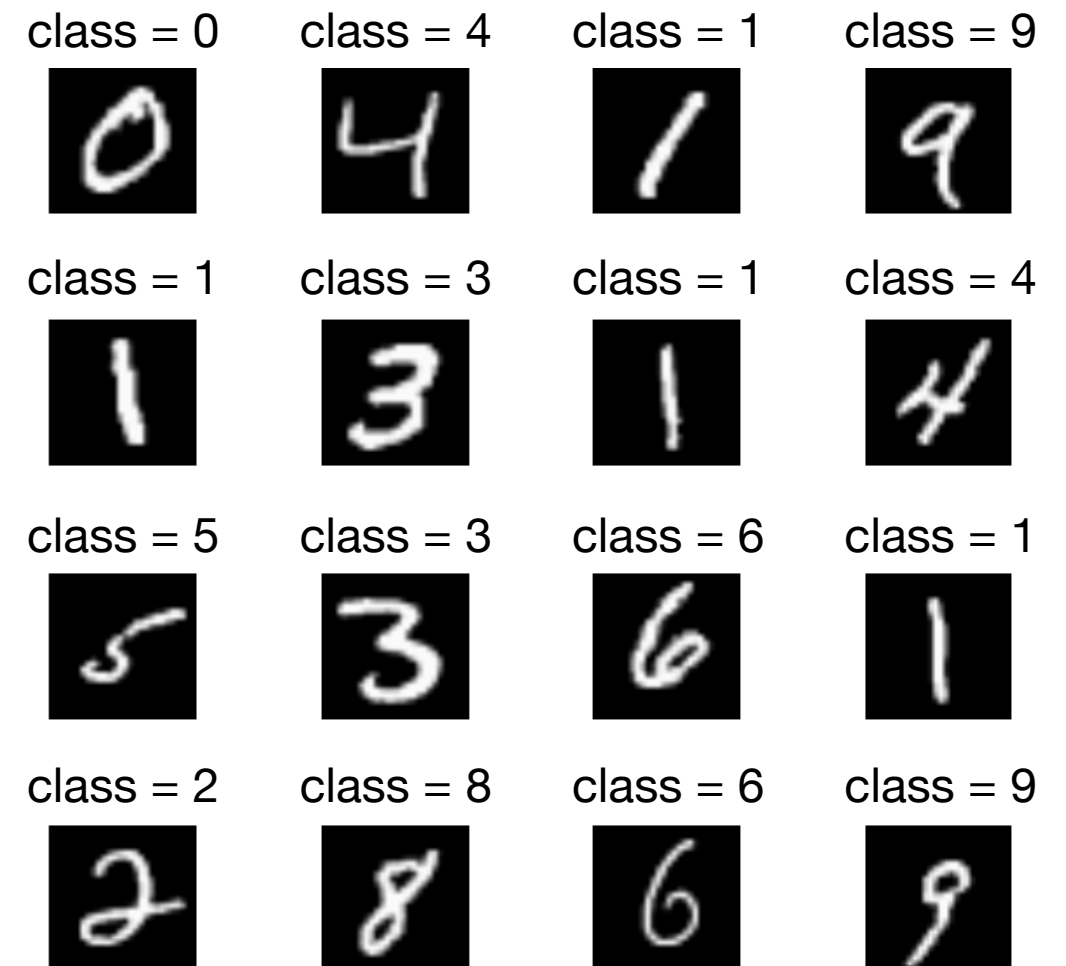
- **Example:**
Classify mortgage applicants with one of two labels (**low-risk**, **high-risk**) based on savings and income data.



- Instead of doing this manually, can we use an algorithm to automatically classify a new mortgage applicant X as being either **low-risk** or **high-risk**, based on existing examples of each group?

Multiclass Classification

- In other cases, we want to train an algorithm to learn to automatically classify a new input into one of 3+ different classes.
- **Example:** Automatically determine what a handwritten digit image depicts. Each image is assigned to one of 10 classes - i.e. the digit classification categories 0 to 9.
- **Example:** Automatically assign a news article to one of several news categories, based on its content.



Training Data

- Classification algorithms rely on a set of examples for each of the relevant classes - i.e. a **training set**. These might originate from historic outcomes or human annotations.
- **Example:** Previous mortgage approval decisions can be used as training data to build a binary classifier for approvals.

Case	Income	Savings	Class Label
1	35,000	2,000	High-risk
2	51,000	18,000	High-risk
3	70,000	42,000	Low-risk
4	26,500	4,500	High-risk
5	32,000	11,000	High-risk
6	53,000	37,000	Low-risk
7	88,000	46,000	Low-risk
8	55,000	5,700	High-risk
9	90,000	35,000	Low-risk
10	43,000	24,000	High-risk

Training Data

- **Example:** The MNIST dataset contains 70,000 images of handwritten digits. Each image is assigned to one of 10 classes.

Example Images															Class Label
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	"0"
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	"1"
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	"2"
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	"3"
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	"4"
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	"5"
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	"6"
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	"7"
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	"8"
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	"9"

- The performance of classification algorithms is dependent on the availability of reliable training data. Often building a training set requires significant amount of effort by human annotators.

Classification Algorithms

- Many different learning algorithms exist for classification (e.g. k-nearest neighbour, decision tree, neural network, support vector machine).
- Problem dimensions will often determine which classification algorithm will be practically applicable, due to processing, memory, and storage requirements.
 1. Number of examples N .
 - Sometimes millions of input examples.
 2. Number of features (dimensions) D representing each example.
 - Often 10-1000, but sometimes far higher.
 3. Number of target classes M .
 - Often small (binary), but sometimes far higher.
 4. Certain types of models perform well empirically for certain domains.

Similarity/Distance-based Algorithms

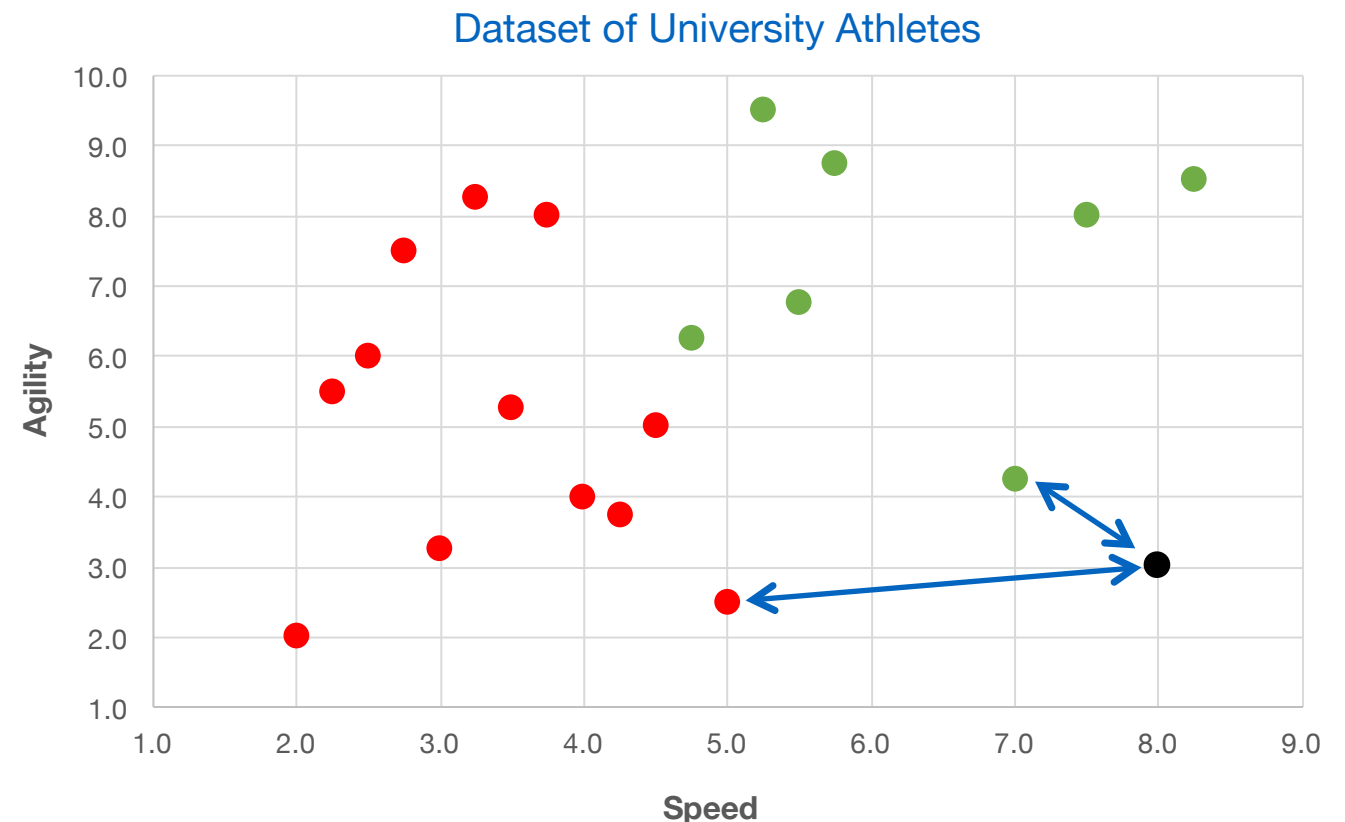
Fundamental Strategy: “Best way to make predictions is to look at past examples and repeat the same process again”.

Feature space:

A D -dimensional coordinate space used to represent the input examples for a given problem, with one coordinate per descriptive feature.

Similarity measure:

Some function to measure how similar (or distant) two input examples are from one another are in the D -dimensional coordinate space.



2 features describing each example (agility & speed)

→ 2 coordinate dimensions for measuring similarity

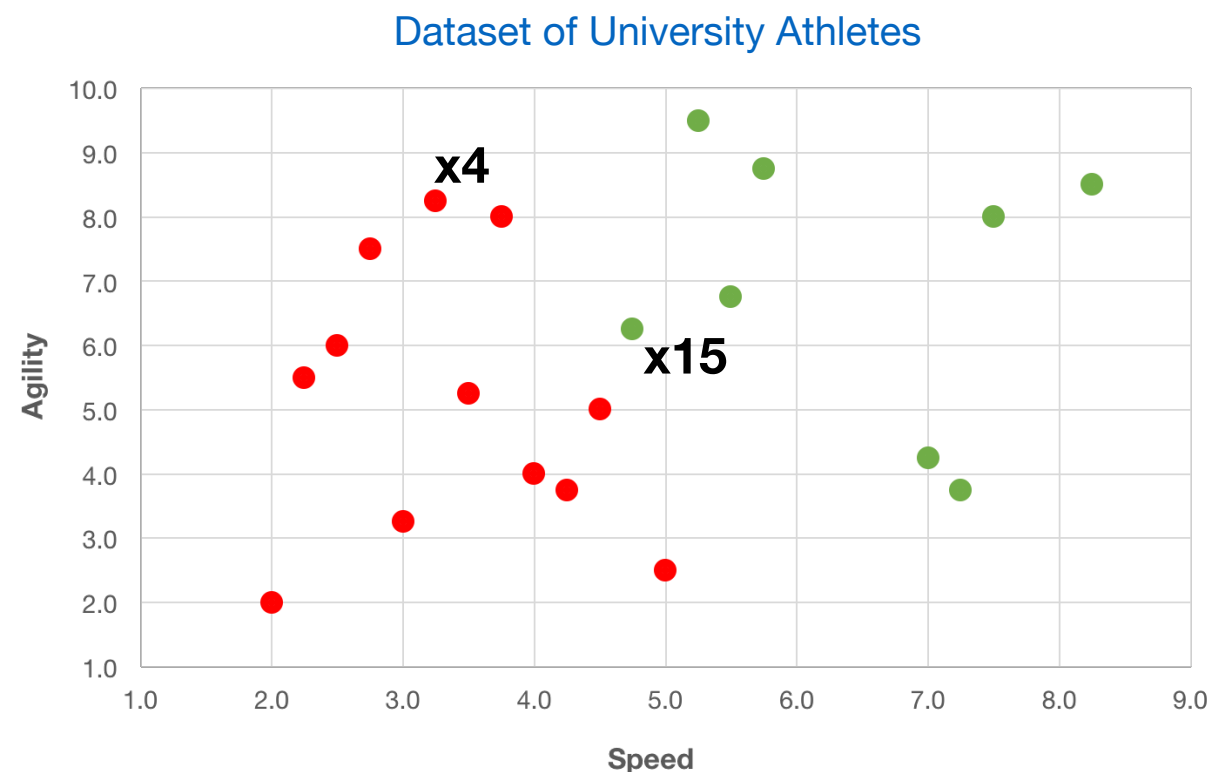
Typically: $\text{Distance} = 1/\text{Similarity}$ OR $\text{Distance} = 1 - \text{Similarity}$

Similarity/Distance-based Algorithms

- For numeric data, the most common distance function used is the **Euclidean distance**.
- Calculated as square root of sum of squared differences for each feature f representing a pair of examples.

$$ED(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{f \in F} (q_f - p_f)^2}$$

Example	Speed	Agility
x4	3.25	8.25
x15	4.75	6.25

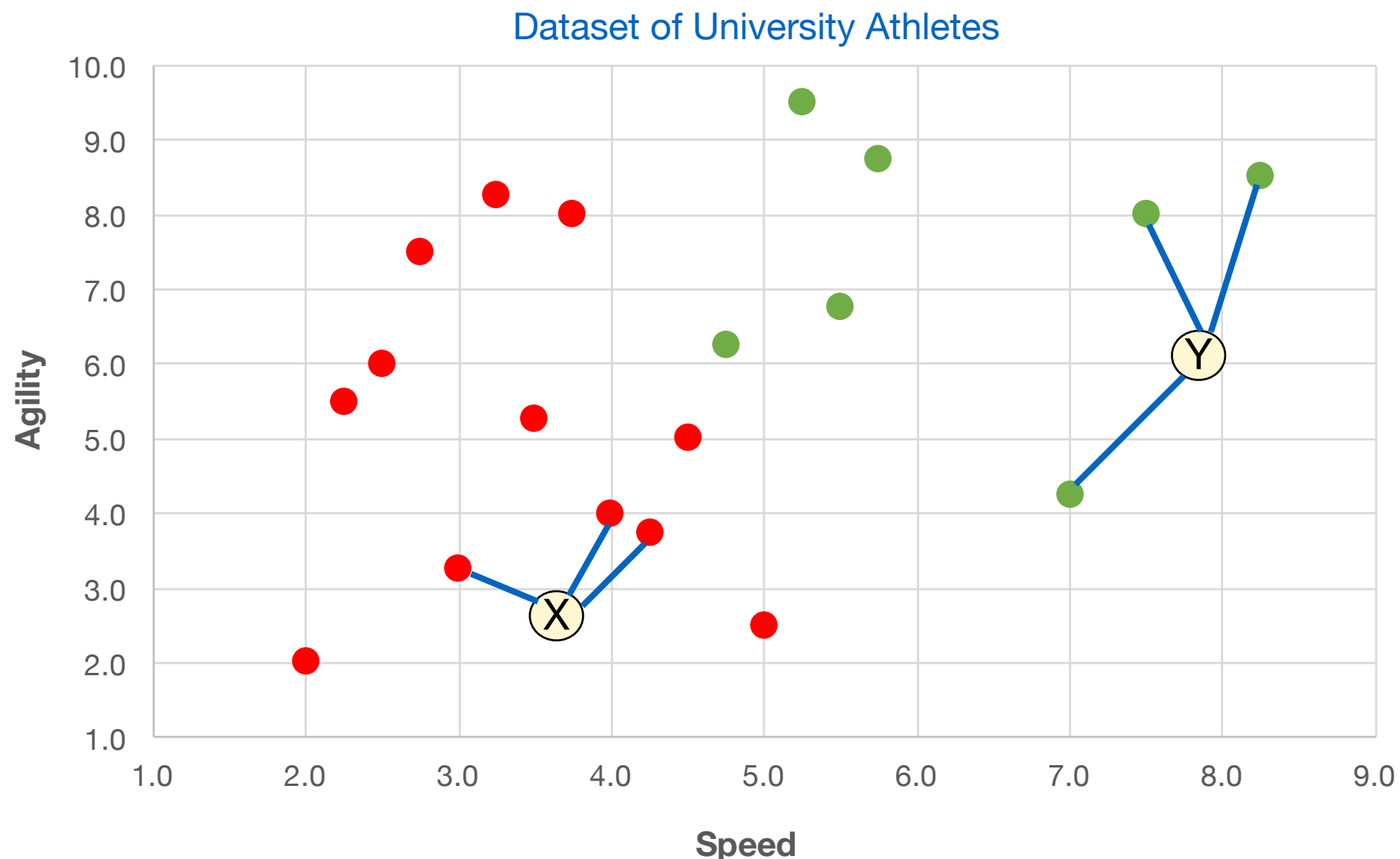


$$ED(x4, x15) = \sqrt{(3.25 - 4.75)^2 + (8.25 - 6.25)^2} = \sqrt{6.25} = 2.5$$

KNN Classifier

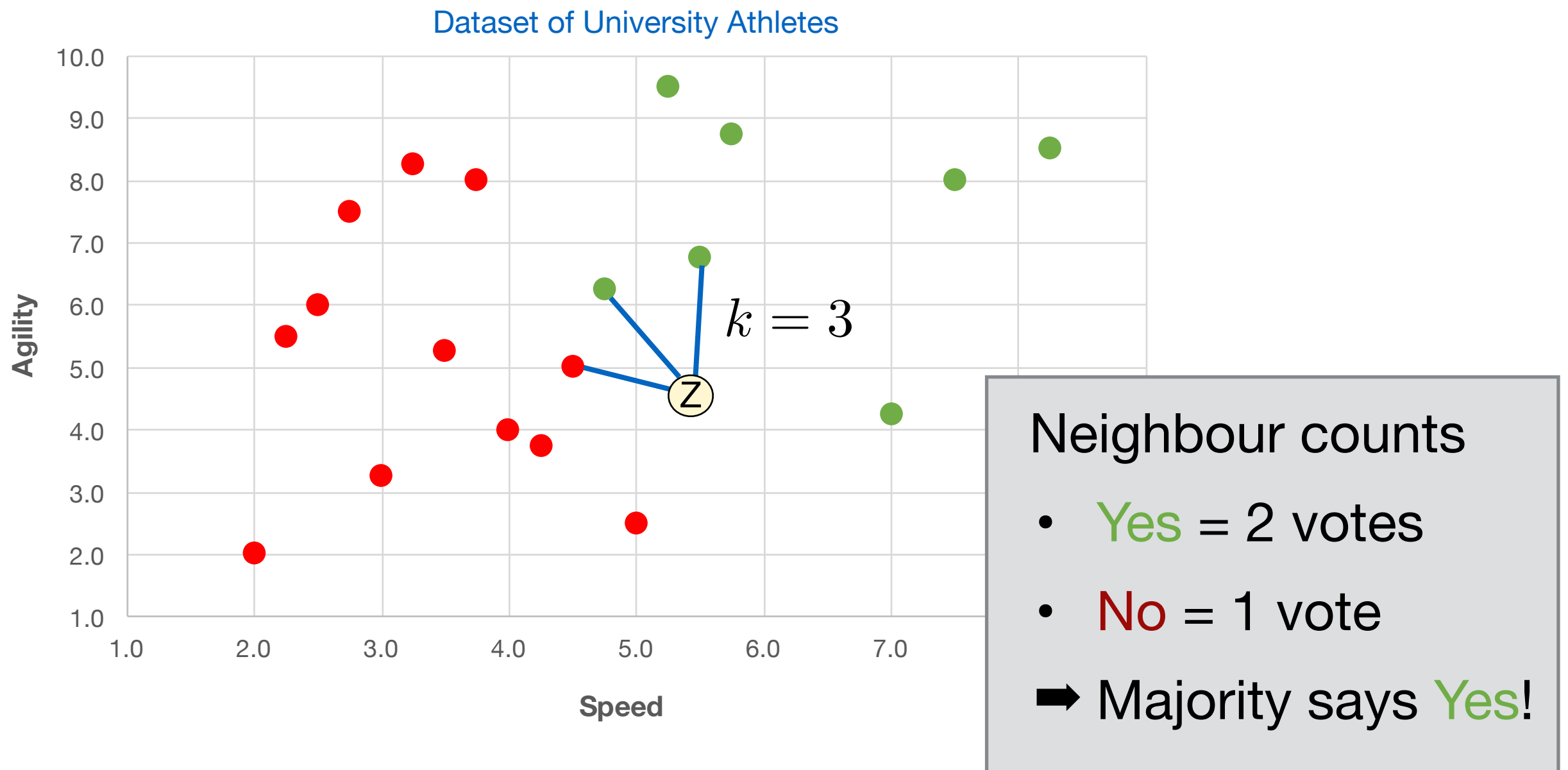
***k*-Nearest Neighbour Classifier:** Simple but effective "lazy" classifier. Find most similar previous examples for which a decision has already been made (i.e. their **nearest neighbours** from the training set).

Example: For new input examples X and Y, we look at the labels of their *k* nearest neighbours under both features (e.g. *k*=3 neighbours)



KNN Classifier

Majority voting: The predicted label for a new input example Z is decided based on the “votes” of its k nearest neighbours, where the neighbours are selected to minimise the distance $d(q, x_i)$



KNN Classifier in Python

- Scikit-learn includes a range of classification algorithms. In all cases, we call `fit()` to build a model on training data, then we call `predict()` to predict class labels for new inputs.
- Example:** Use the Penguins dataset, where the goal is to classify species based on a set of features describing each animal.

Read CSV file into a Pandas Data Frame

```
df = pd.read_csv('penguins_af.csv', index_col=0)
```

Extract 4 useful numeric features for the rows

```
data = df.iloc[:,2:6]
```

Extract the target label (species) for the rows

```
target = df.iloc[:,0]  
list(target.unique())
```

```
['Adelie', 'Gentoo', 'Chinstrap']
```

```
data.head()
```

	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
0	39.1	18.7	181.0	3750.0
1	39.5	17.4	186.0	3800.0
2	40.3	18.0	195.0	3250.0
4	36.7	19.3	193.0	3450.0
5	39.3	20.6	190.0	3650.0

```
target.head()
```

```
0    Adelie  
1    Adelie  
2    Adelie  
4    Adelie  
5    Adelie  
Name: species, dtype: object
```

KNN Classifier in Python

- Train a classifier to predict the species (class label) for a new data point describing a penguin. There are 3 possible species (Adelie, Gentoo, Chinstrap), so this is a multiclass problem.
- Step 1: Fit a KNN classifier with $K=3$ neighbours on the full dataset - this is our training data, using the `fit()` function.

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3)
model.fit(data, target)
```

$K=3$ neighbours
parameter

- Step 2: Make a prediction for a new input example using the trained KNN model, using the `predict()` function.

```
xinput = np.array([[51.0, 15.6, 235.0, 5400.0]])
model.predict(xinput)[0]

'Gentoo'
```

Note new input
example has same
4 numeric features
as training data

KNN Classifier in Python

- We can also generate predictions for multiple input examples using a single call to `predict()`, or make multiple calls to `predict()` to re-use the same trained model.

Randomly select 4 rows from the numeric data as inputs

```
xinputs = data.sample(n=4)
```

bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
59.6	17.0	230.0	6050.0
47.3	15.3	222.0	5250.0
43.5	18.1	202.0	3400.0
35.0	17.9	190.0	3450.0

Use the previous model to make predictions for the inputs

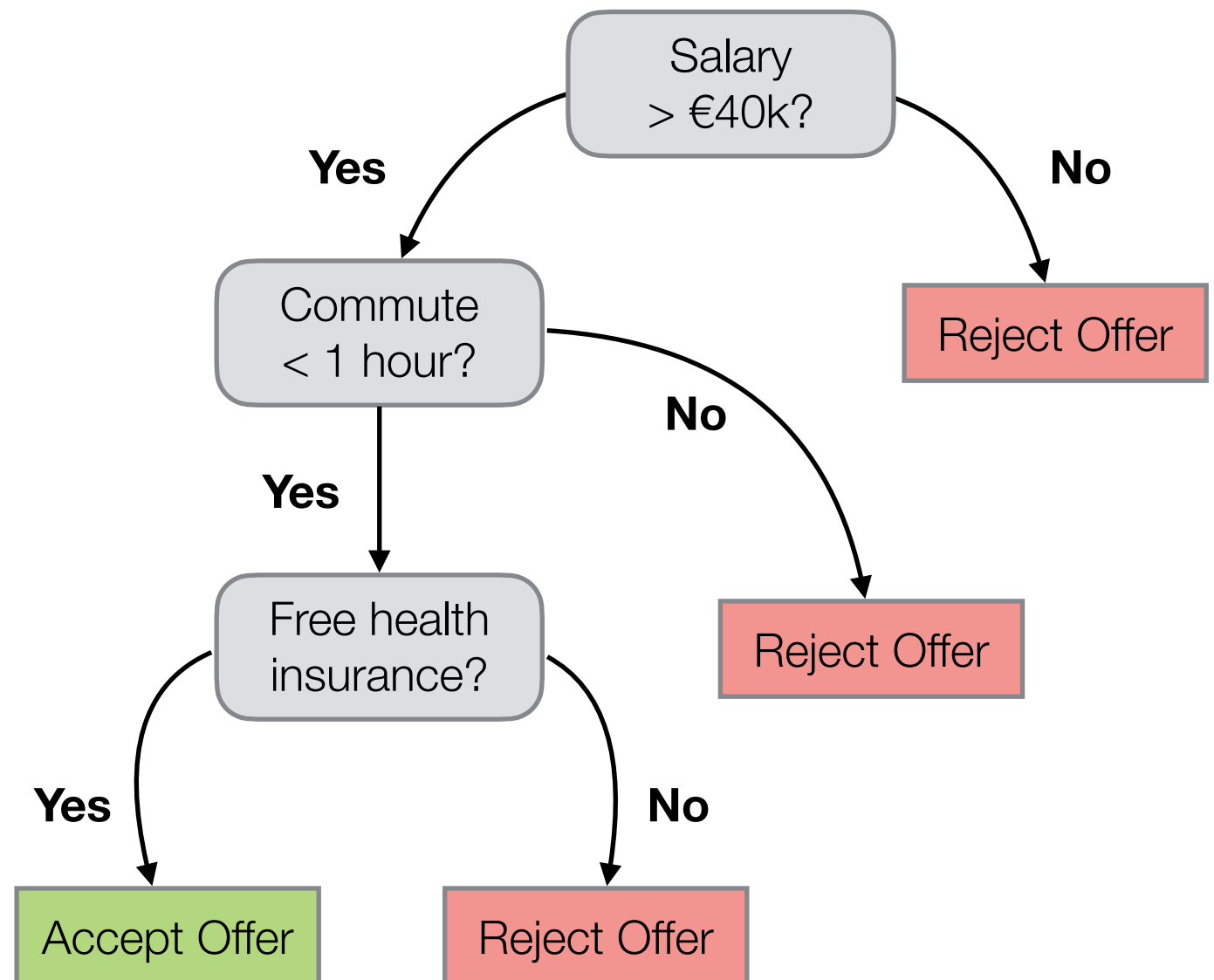
```
list(model.predict(xinputs))
```

```
['Gentoo', 'Gentoo', 'Chinstrap', 'Adelie']
```

- As with many classification algorithms, the choice of parameters (i.e. value of k) can have a significant impact on KNN predictions.

Decision Tree Classifier

- Create a classification model to predict class labels by learning decision rules learned from training data.



Decision Tree Classifier

- **Basic Idea:** Build a tree model that splits the training set into subsets in which all examples have the same class.
- Splitting is done using rules inferred from the training set.
- Each rule is based on a feature, and the split corresponds to the values it can take:
 - e.g. `insured = {true, false}`
 - e.g. `wind = {weak, strong}`
 - e.g. `income = {low, average, high}`
 - e.g. `height < 6ft, height ≥ 6ft`
- If necessary, each subset can be split again using another feature, and so on until all examples have the same class.
- Once the tree is built, we can use it to quickly classify new input examples - this is an eager learning strategy.

Decision Tree Example

Q. “Will a customer wait for a restaurant table?”

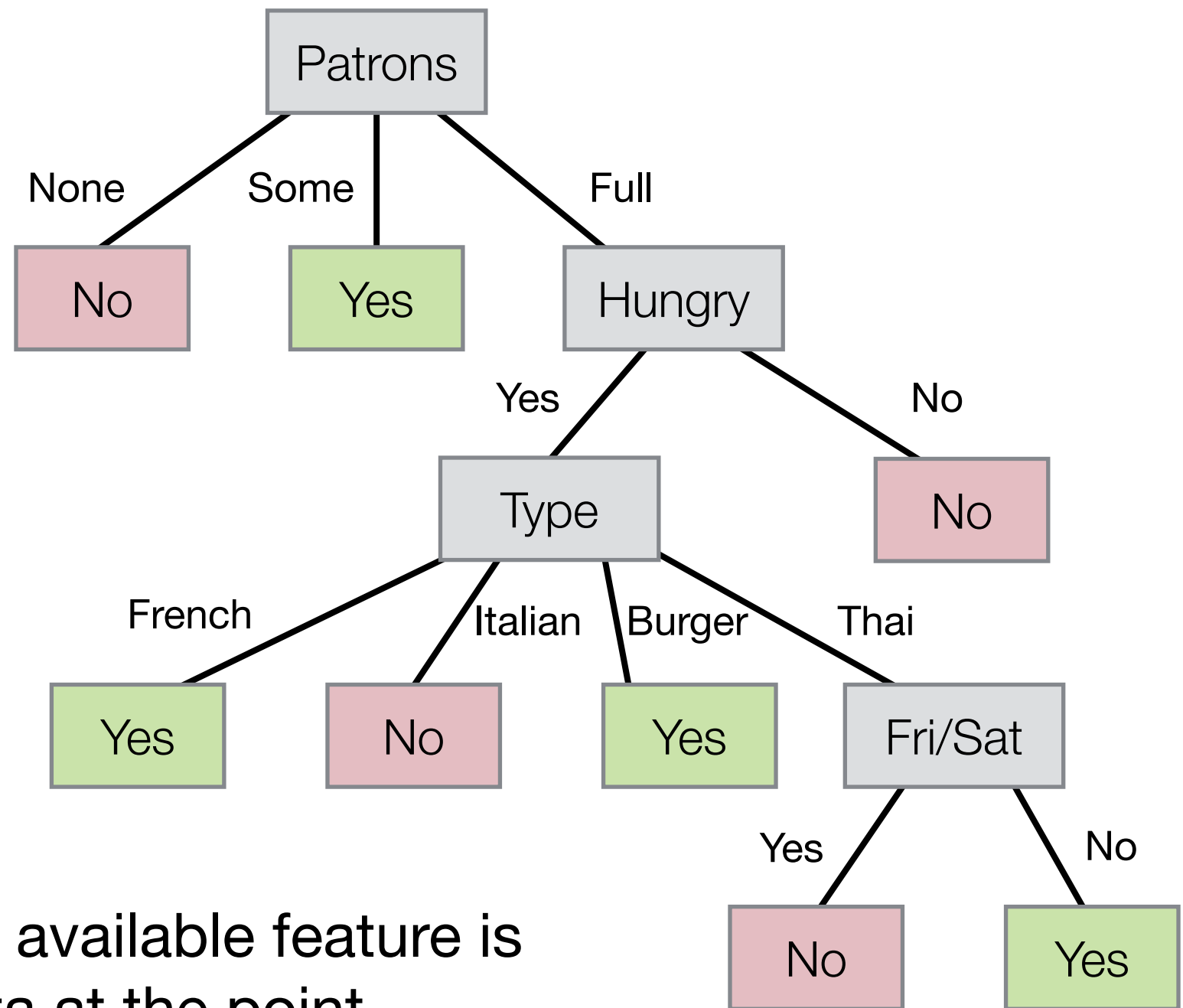
Russell & Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2009.

Binary classification task (WillWait = {Yes, No}), with examples described by 10 different features:

Example	Alternate	Bar	Fri/Sat	Hungry	Patrons	Price	Raining	Reservation	Type	WaitEst	WillWait?
1	Yes	No	No	Yes	Some	€€€	No	Yes	French	0-10	Yes
2	Yes	No	No	Yes	Full	€	No	No	Thai	30-60	No
3	No	Yes	No	No	Some	€	No	No	Burger	0-10	Yes
4	Yes	No	Yes	Yes	Full	€	No	No	Thai	10-30	Yes
5	Yes	No	Yes	No	Full	€€€	No	Yes	French	>60	No
6	No	Yes	No	Yes	Some	€€	Yes	Yes	Italian	0-10	Yes
7	No	Yes	No	No	None	€	Yes	No	Burger	0-10	No
8	No	No	No	Yes	Some	€€	Yes	Yes	Thai	0-10	Yes
9	No	Yes	Yes	No	Full	€	Yes	No	Burger	>60	No
10	Yes	Yes	Yes	Yes	Full	€€€	No	Yes	Italian	10-30	No
11	No	No	No	No	None	€	No	No	Thai	0-10	No
12	Yes	Yes	Yes	Yes	Full	€	No	No	Burger	30-60	Yes

Decision Tree Example

- A “good” decision tree will classify all examples correctly using as few tree nodes as possible.

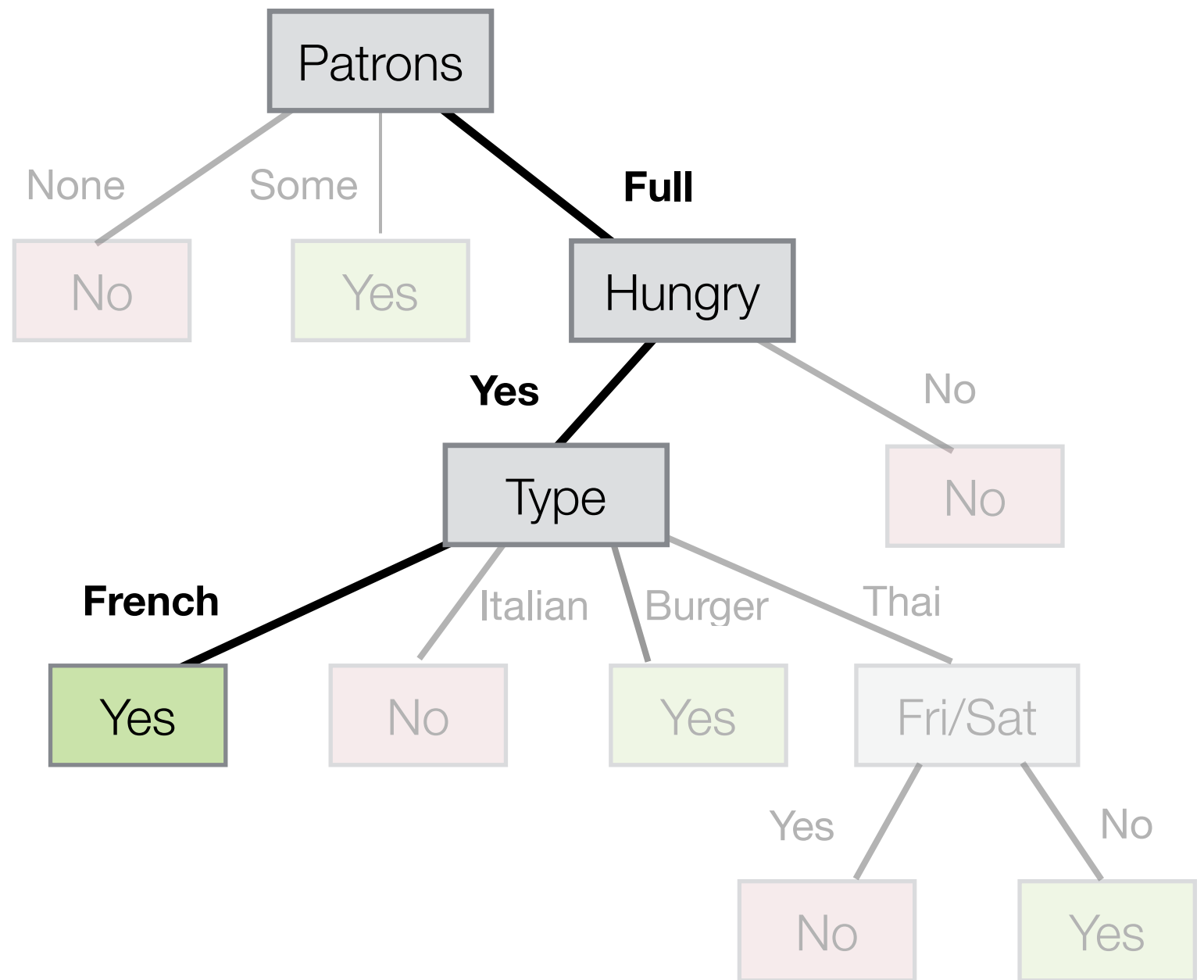


At each level, the best available feature is chosen to split the data at the point.

Decision Tree Example

- Once we have built a decision tree from the training set, we can use the rules in the tree to quickly classify new input examples.

Feature	Query <i>x1</i>
<i>Alternate</i>	Yes
<i>Bar</i>	No
<i>Fri/Sat</i>	Yes
<i>Hungry</i>	Yes
<i>Patrons</i>	Full
<i>Price</i>	€€
<i>Raining</i>	No
<i>Reservation</i>	No
<i>Type</i>	French
<i>WaitEstimate</i>	10-30



⇒ Based on tree, output for *x1* is "Yes"

Decision Tree Classifier in Python

- **Example:** Apply a Decision Tree to the *Penguins* dataset.
- Step 1: Apply a Decision Tree classifier on the full dataset, again using the `fit()` function to create the model.

```
df = pd.read_csv('penguins_af.csv', index_col=0)
data = df.iloc[:,2:6]
target = df.iloc[:,0]
```

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(data, target)
```

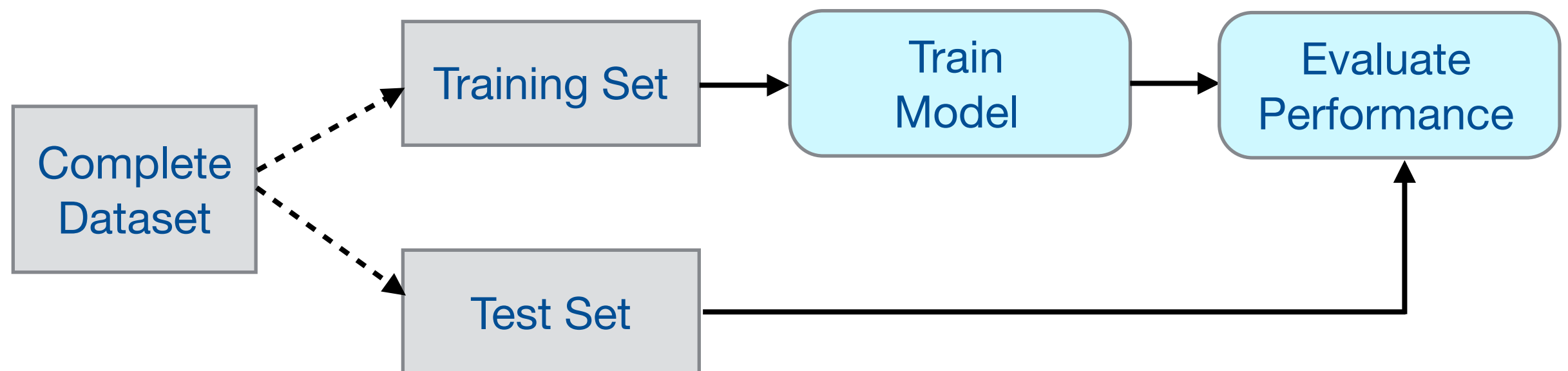
- Step 2: Make a prediction for a new unseen input example using the trained tree model, using the `predict()` function.

```
xinput = np.array([[51.0, 15.6, 235.0, 5400.0]])
model.predict(xinput)[0]
```

```
['Gentoo']
```

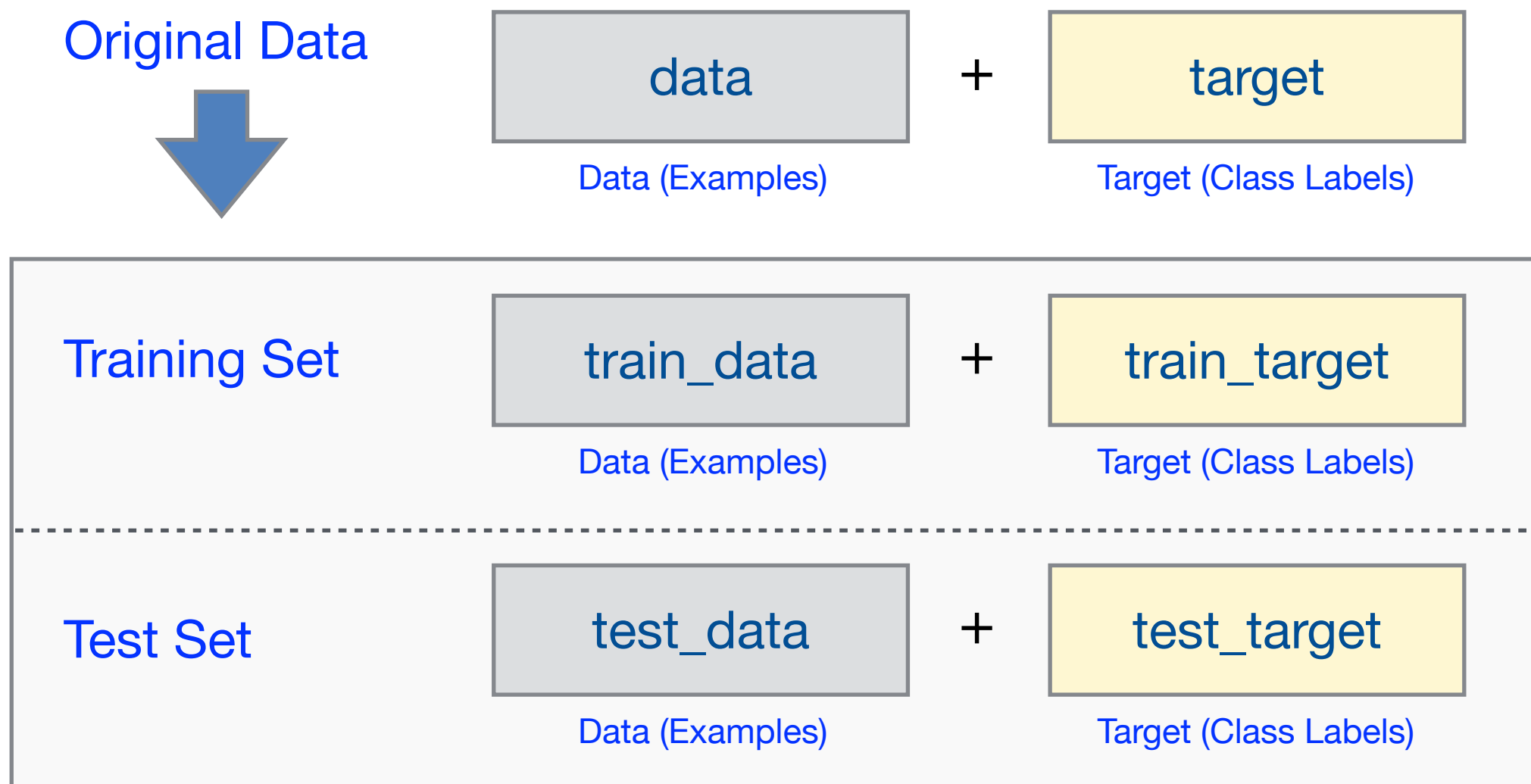

Evaluating Classifiers

- A key aspect of applying classification involves quantitatively assessing how well an algorithm is performing at the task at hand.
- Standard evaluation approach for classification tasks is to split the complete original set of annotated examples into two distinct sets:
 1. **Training set:** Set of examples provided to the classifier to build a model of the data. Each has been assigned a class label.
 2. **Test set:** Separate set of examples which are used to evaluate the performance of the classifier. The class labels are "hidden" from the algorithm.



Evaluating Classifiers

- **Hold-out strategy:** A common approach for generating training and test sets is to take the complete dataset and randomly "hold back" some examples (e.g. 20%) in the test set for evaluation.
- In Scikit-learn, this split involves creating 4 new variables:



Evaluating Classifiers

- We can use the `train_test_split()` function to randomly split a dataset (and its associated class labels) into two distinct training and test sets for evaluation purposes.

```
from sklearn.model_selection import train_test_split
train_data, test_data, train_target, test_target =
    train_test_split(data, target, test_size=0.2)
```

Randomly split: 80% of the data for training, 20% for testing

```
model = KNeighborsClassifier(n_neighbors=3)
model.fit(train_data, train_target)
```

Fit a KNN classifier with K=3 neighbours on the data in the training set

```
predicted = model.predict(test_data)
print(predicted)
```

```
[0 1 1 0 2 1 1 2 1 2 1 1 1 2 1 2 0 0 0 2 1 2 1 1 0
 2 0 1 1 1]
```

Make a prediction for the examples in the test data

```
print(test_target)
```

```
[0 1 1 0 2 1 1 2 1 2 1 1 1 2 1 2 0 0 0 1 1 2 1 1 0
 2 0 1 1 2]
```

Compare the predictions to the "correct" labels for the test data

Overfitting

- Q. Why not just use all of the original data to build our model?
- For real-world tasks, we are interested in the **generalisation accuracy** of a classification system.
- **Overfitting**: Model is fitted too closely to the training data (including its noise). The model cannot generalise to situations not presented during training, so it is not useful when applied to unseen data.
- A good model must not only fit the training data well, but also accurately classify examples that it has never seen before.
- Using a hold-out set avoids **peeking** - when the performance of a model is evaluated using the same data used to train it.

Evaluation Measures

When making predictions, we need some kind of measure to capture how often the model makes correct or incorrect predictions, and how severe the mistakes are.











Accuracy:

Simplest measure. Fraction of correct predictions made by the classifier.

$$ACC = \frac{\# \text{ correct predictions}}{\text{total predictions}}$$

Example: Predictions made for test set of 10 emails

$$ACC = \frac{7}{10} = 0.7$$

Email	Label	Prediction	Correct?
1	spam	non-spam	
2	spam	spam	
3	non-spam	non-spam	
4	spam	spam	
5	non-spam	spam	
6	non-spam	non-spam	
7	spam	spam	
8	non-spam	spam	
9	non-spam	non-spam	
10	spam	spam	

Measuring Accuracy in Python

- The `sklearn.metrics` package provides a range of measures for evaluating classifiers.

```
from sklearn.model_selection import train_test_split
train_data, test_data, train_target, test_target =
    train_test_split(data, target, test_size=0.2)
```

Randomly split: 80% of the data for training, 20% for testing

```
model = KNeighborsClassifier(n_neighbors=3)
model.fit(train_data, train_target)
```

Fit a KNN classifier on the data in the training set

```
predicted = model.predict(test_data)
print(predicted)
```

```
[0 1 1 0 2 1 1 2 1 2 1 1 1 2 1 2 0 0 0 2 1 2 1 1 0
 2 0 1 1 1]
```

Make a prediction for the examples in the test data

```
from sklearn.metrics import accuracy_score
accuracy_score(test_target, predicted)
```

```
0.9333333333333333
```

Calculate accuracy score for the predictions, based on actual target labels for test data

Confusion Matrix











- Accuracy does not provide the full picture of the performance of a classifier - where do the errors lie?
- A **confusion matrix** summarises different aspects of classifier performance - i.e. where a classifier performs well and badly.

		Predicted Label	
		Positive	Negative
Actual Label	Positive	TP True positive Correct!	FN False Negative (Type II error)
	Negative	FP False Positive (Type I error)	TN True Negative Correct!

Confusion Matrix

Example: Predict an email as *positive* (spam) or *negative* (non-spam)

- **TP** = Spam emails correctly predicted as spam
- **FP** = Non-spam emails incorrectly predicted as spam
- **TN** = Non-spam emails correctly predicted as non-spam
- **FN** = Spam emails incorrectly predicted as non-spam

Email	Label	Prediction	Correct?	Outcome
1	spam	non-spam		FN
2	spam	spam		TP
3	non-spam	non-spam		TN
4	spam	spam		TP
5	non-spam	spam		FP
6	non-spam	non-spam		TN
7	spam	spam		TP
8	non-spam	spam		FP
9	non-spam	non-spam		TN
10	spam	spam		TP

Predicted Label		
Spam	Non	
TP=4	FN=1	Spam
FP=2	TN=3	Non

Actual Label

➡ A perfect classifier would produce a pure diagonal matrix...

Confusion Matrix in Python

The function `confusion_matrix()` compares real target labels to predictions, and produces a NumPy array:

```
model = KNeighborsClassifier(n_neighbors=3)
model.fit(data_train, target_train)
```

Apply KNN classifier to training data with target labels

```
predicted = model.predict(data_test)
```

Make predictions for the 20 examples in the test set

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(target_test, predicted, labels=[1,-1])
print(cm)
```

Calculate and print the confusion matrix

```
[[1 7]
 [0 12]]
```

Predicted Label

1	-1	
TP=1	FN=7	1
FP=0	TN=12	-1

Actual Label

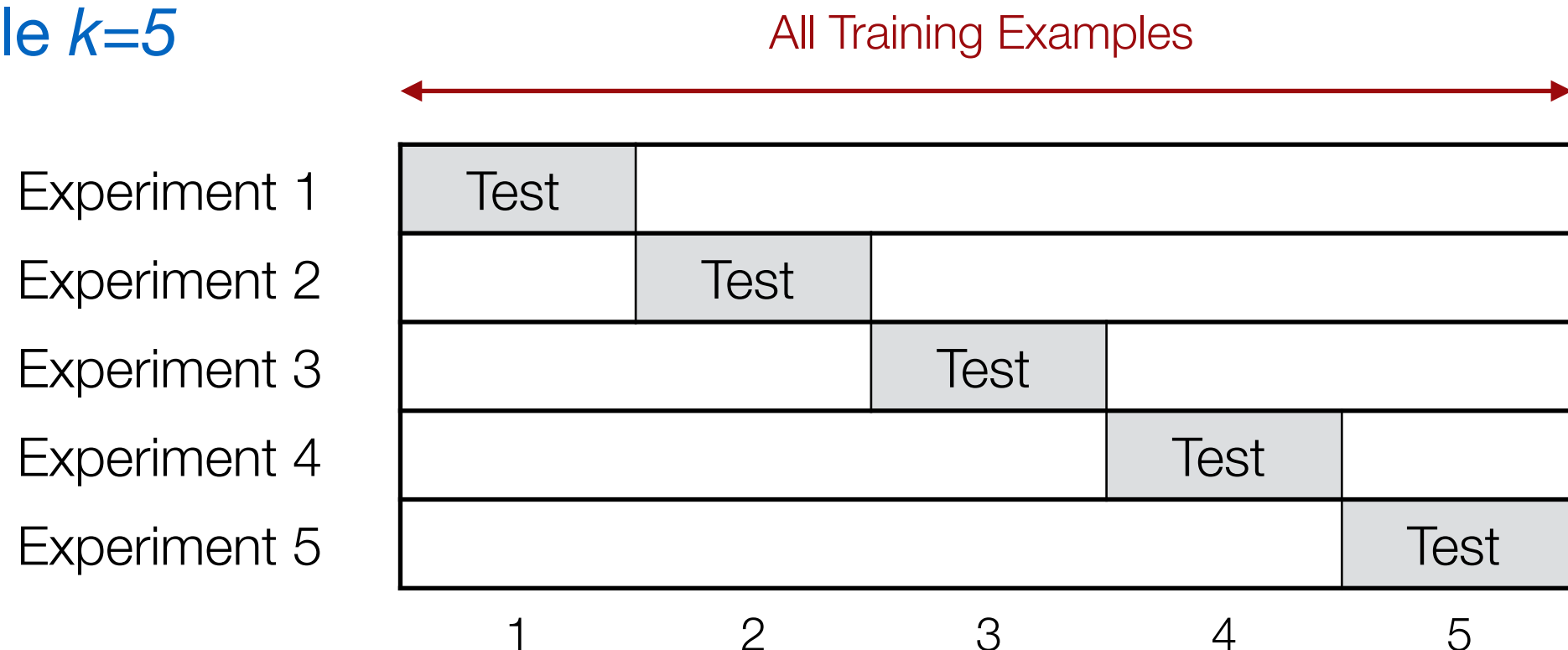
Accuracy is 65%, but confusion matrix shows high level of False Negatives.

7 test examples predicted as "-1" but real label is "1"

Cross-Validation

- A problem with simply randomly splitting a dataset into two sets is that each random split might give different results.
- ***k*-Fold Cross Validation:**
 1. Divide the data into k disjoint subsets - “folds” (e.g. $k=5$).
 2. For each of k experiments, use $k-1$ folds for training and the selected one fold for testing.
 3. Repeat for all k folds, average the accuracy/error rates.

Example $k=5$



Cross-Validation in Python

- Scikit-learn allows us to run cross-validation using a single function call `cross_val_score()`.
- We specify the number of folds required and the evaluation measure used to quantify performance for each fold.

```
model = KNeighborsClassifier(n_neighbors=3)
```

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, data, target, cv=5,
                          scoring="accuracy")
print(scores)
```

```
[ 0.567  0.6   0.625  0.6   0.556]
```

```
scores.mean()
```

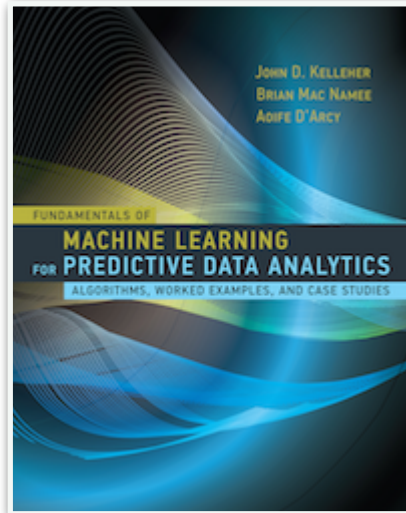
```
0.59
```

Apply cross-validation to full data and target labels for 5 folds, calculating accuracy each time

Result is a NumPy array containing 5 accuracy scores

Overall accuracy for the classifier is mean across folds

Further Reading

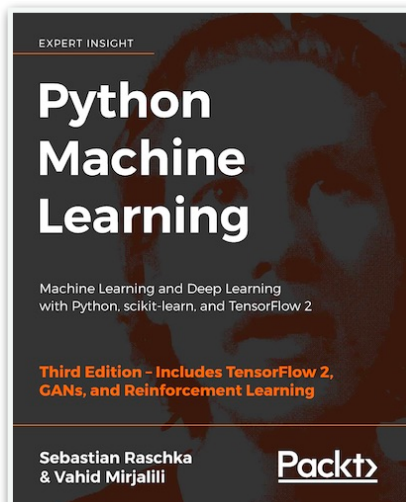
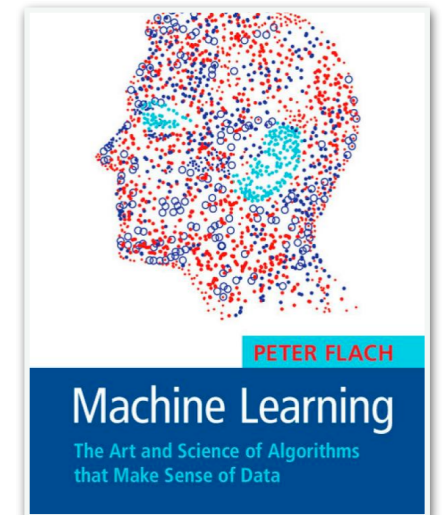


Fundamentals of Machine Learning for Predictive Data Analytics

John D. Kelleher, Brian Mac Namee,
Aoife D'Arcy

Machine Learning: The Art and Science of Algorithms that Make Sense of Data

Peter Flach



Python Machine Learning, 3rd Edition

Sebastian Raschka