

# COMP47670

## Classification and Evaluation

Slides by Derek Greene

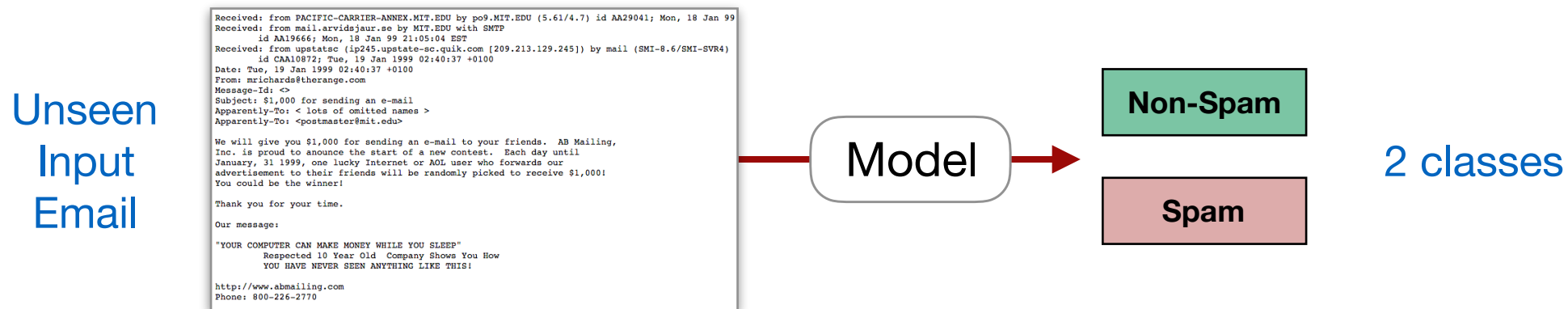
UCD School of Computer Science



# Reminder: Supervised Learning

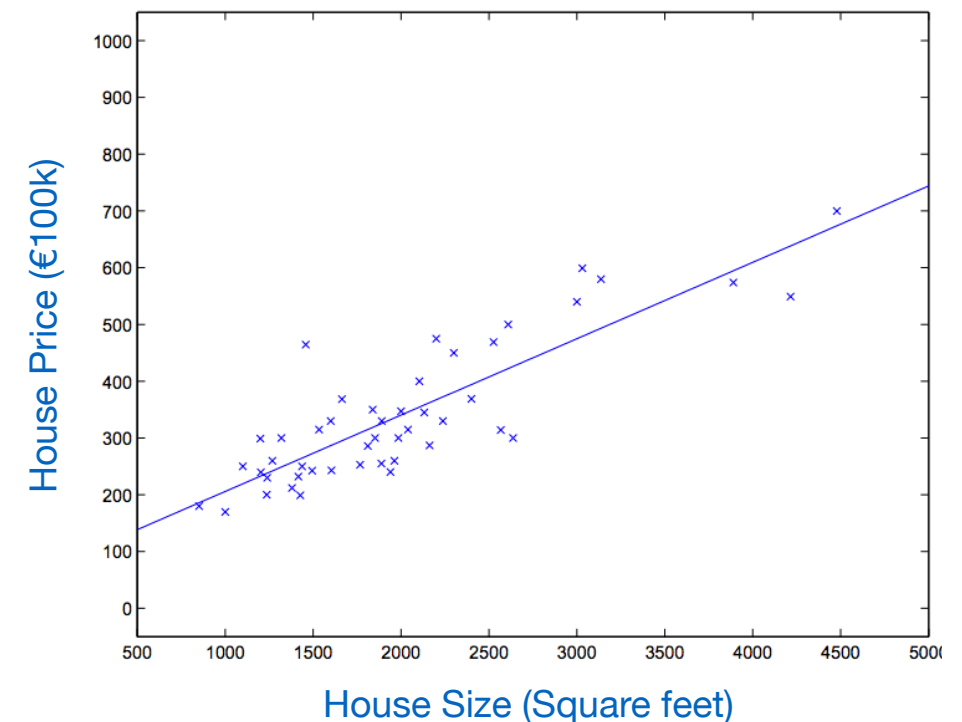
- Classification:**

Learn from a labelled training set to make a prediction to assign a new "unseen" example to one of a fixed number of classes.



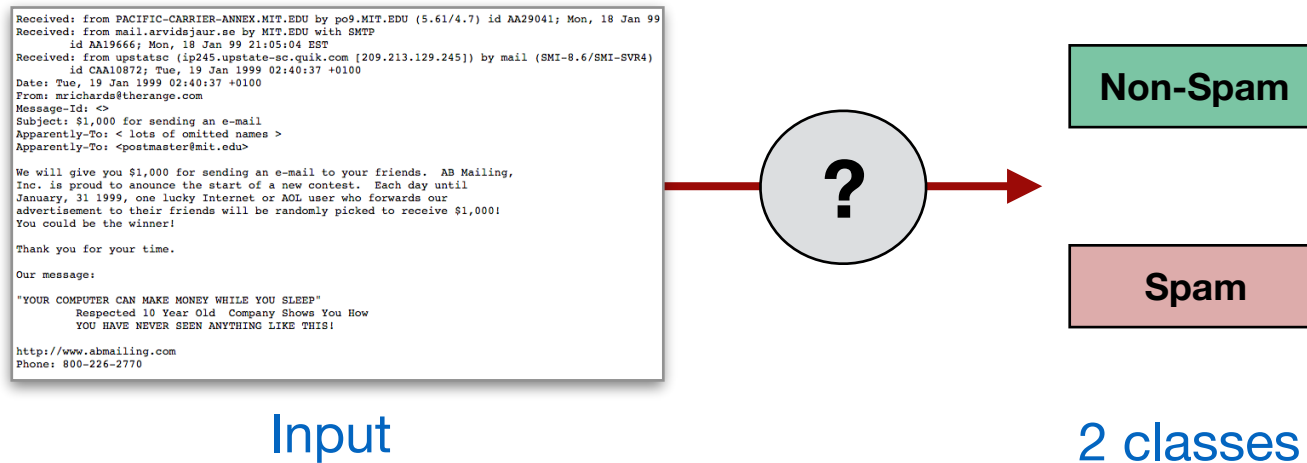
- Regression:**

Learn from a labelled 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 an input to one of two possible target class labels.



- **Multiclass Classification:**  
Assign an input to one of  $M$  different target class labels.

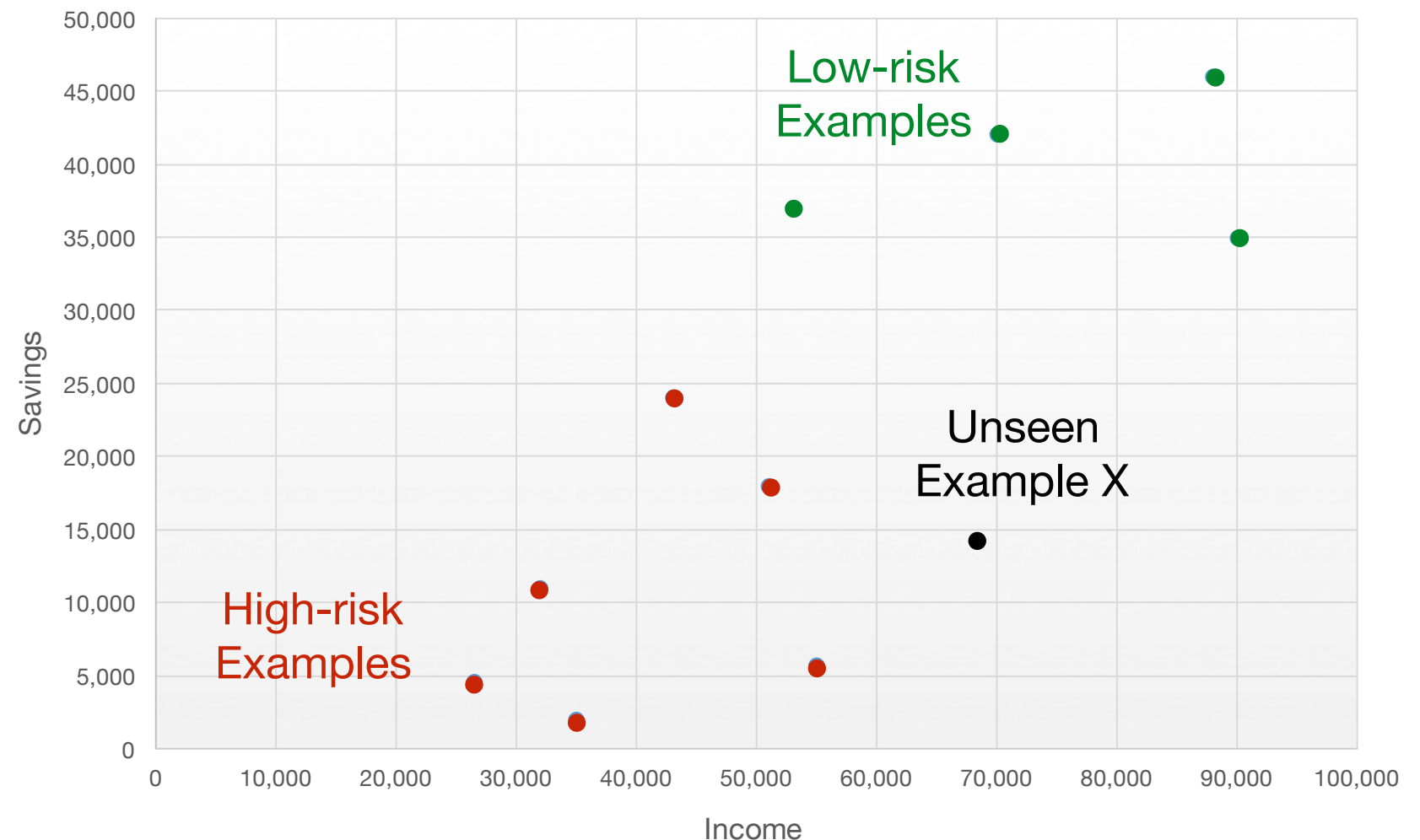




# Typical Binary Classification Task

## Example: Mortgage Approval

Manually classify mortgage applicants into two categories (**low-risk** & **high-risk**) based on savings and income data.

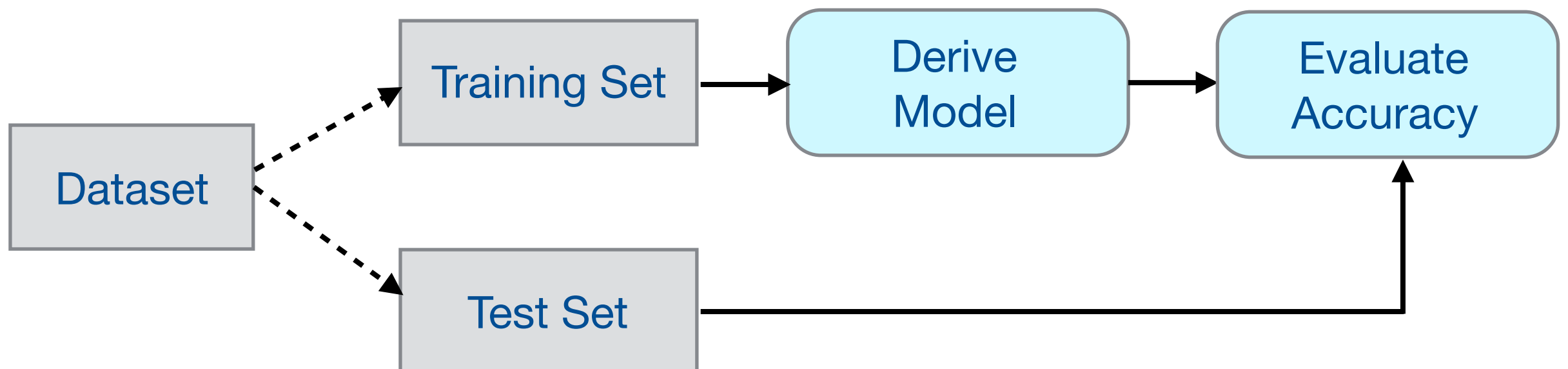


**Q.** Can we train an algorithm to learn to automatically classify a new application X as either **low-risk** or **high-risk**, based on the manually-classified examples of each type?

# Training v Test Data

---

- Standard evaluation approach for classification tasks is to split the set of examples into a *training set* and the *test set*.
- **Training set:** Examples provided to the classifier to build a model of the data. Each has been manually assigned a class label.
- **Test set:** Examples held back from the classifier, which are used to evaluate the accuracy of the classifier. Test examples are completely separate from the training set, so we can evaluate how well the model built by the classifier will generalise to new input examples.



# 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.

# Similarity/Distance-based Learning

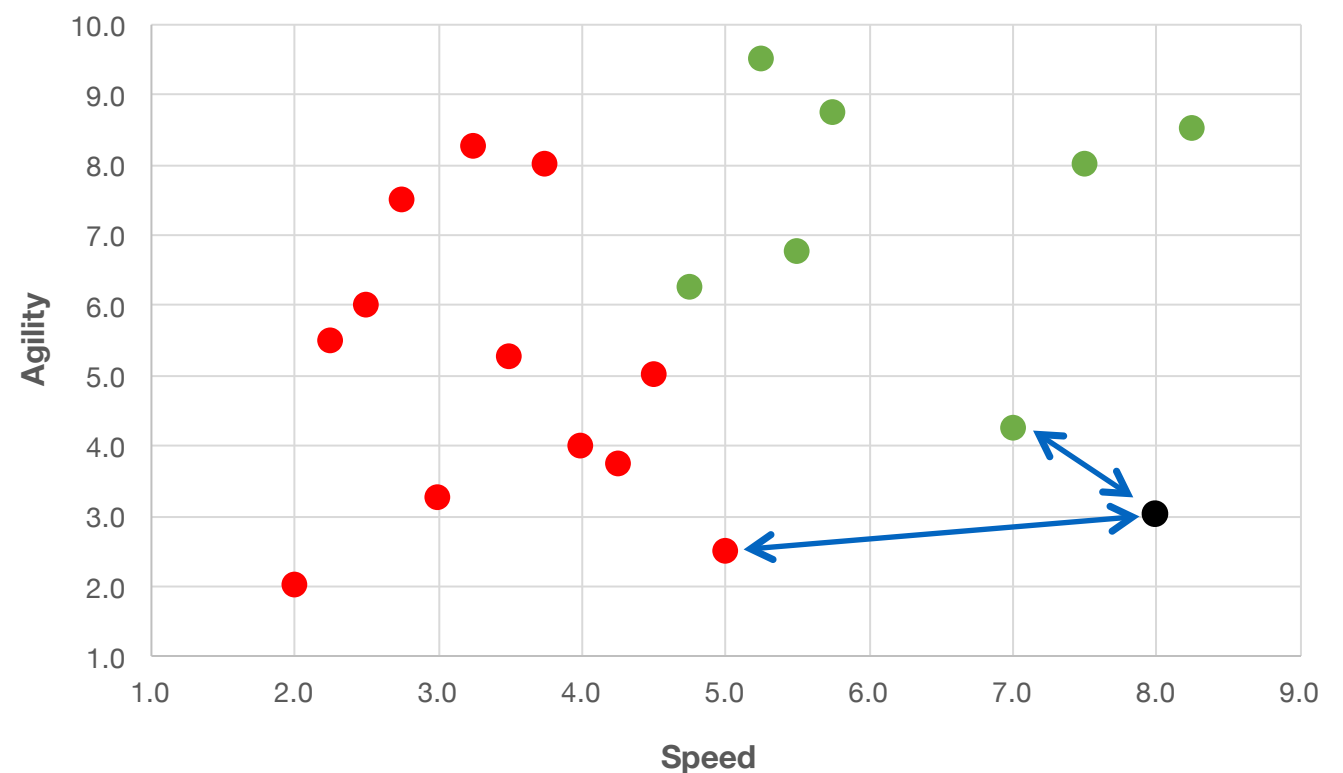
**Fundamental Strategy:** “Best way to make predictions is to look at past examples and repeat the same process again”.

## 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

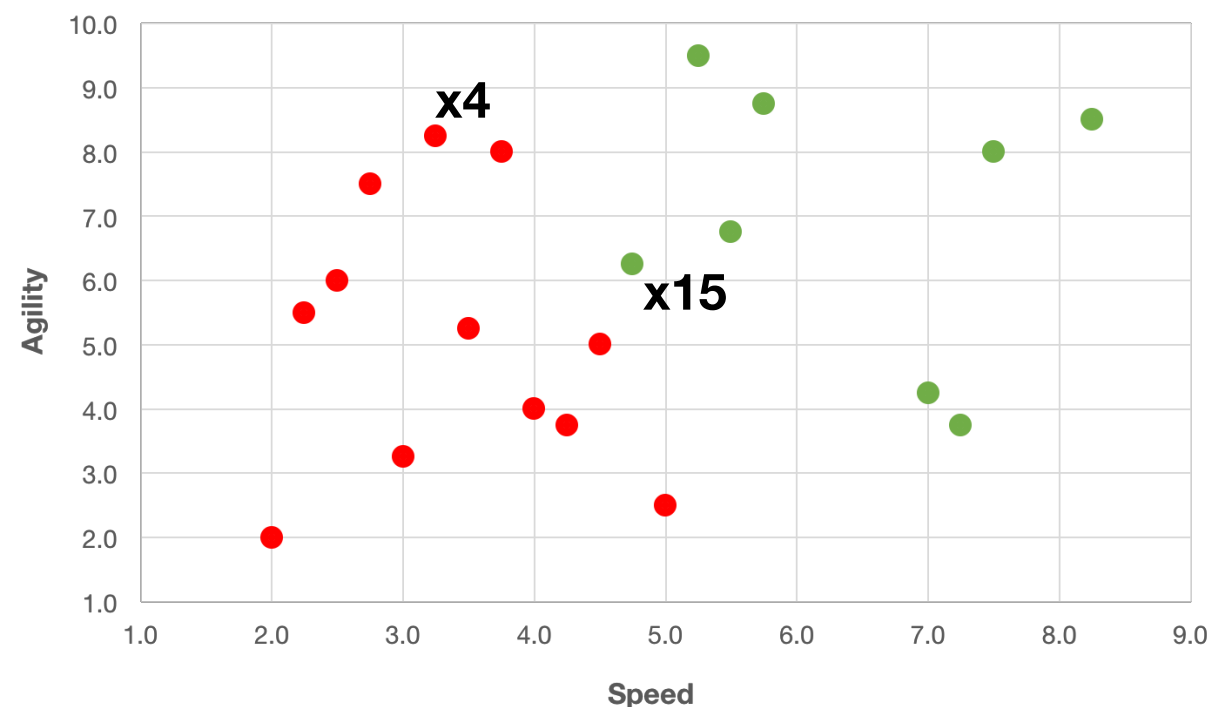
**Typically:**  $\text{Distance} = 1/\text{Similarity}$  OR  $\text{Distance} = 1 - \text{Similarity}$

# Similarity/Distance-based Learning

- 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



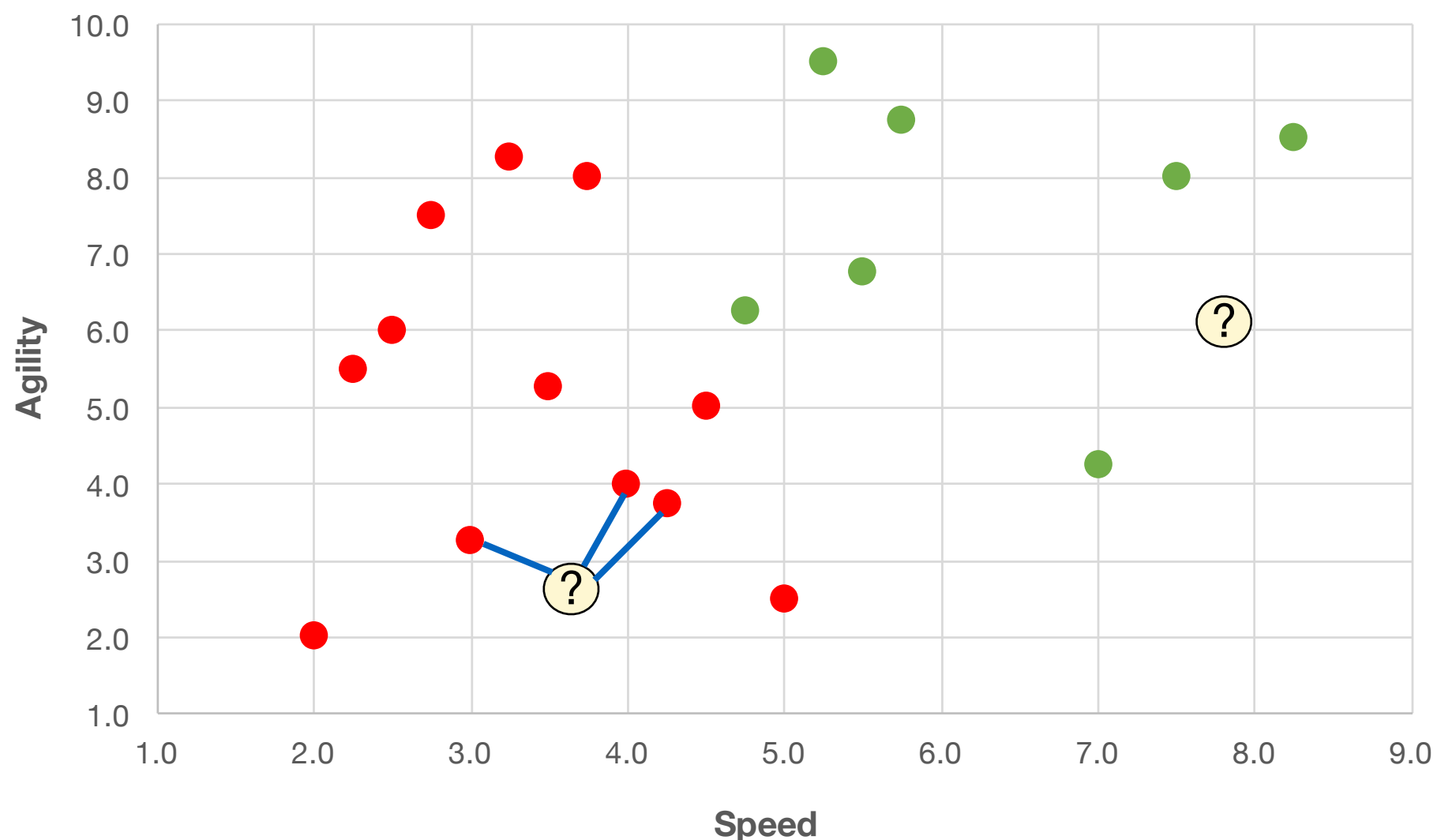
$$ED(x4, x15) = \sqrt{(3.25 - 4.75)^2 + (8.25 - 6.25)^2} = \sqrt{6.25} = 2.5$$



# KNN Classifier

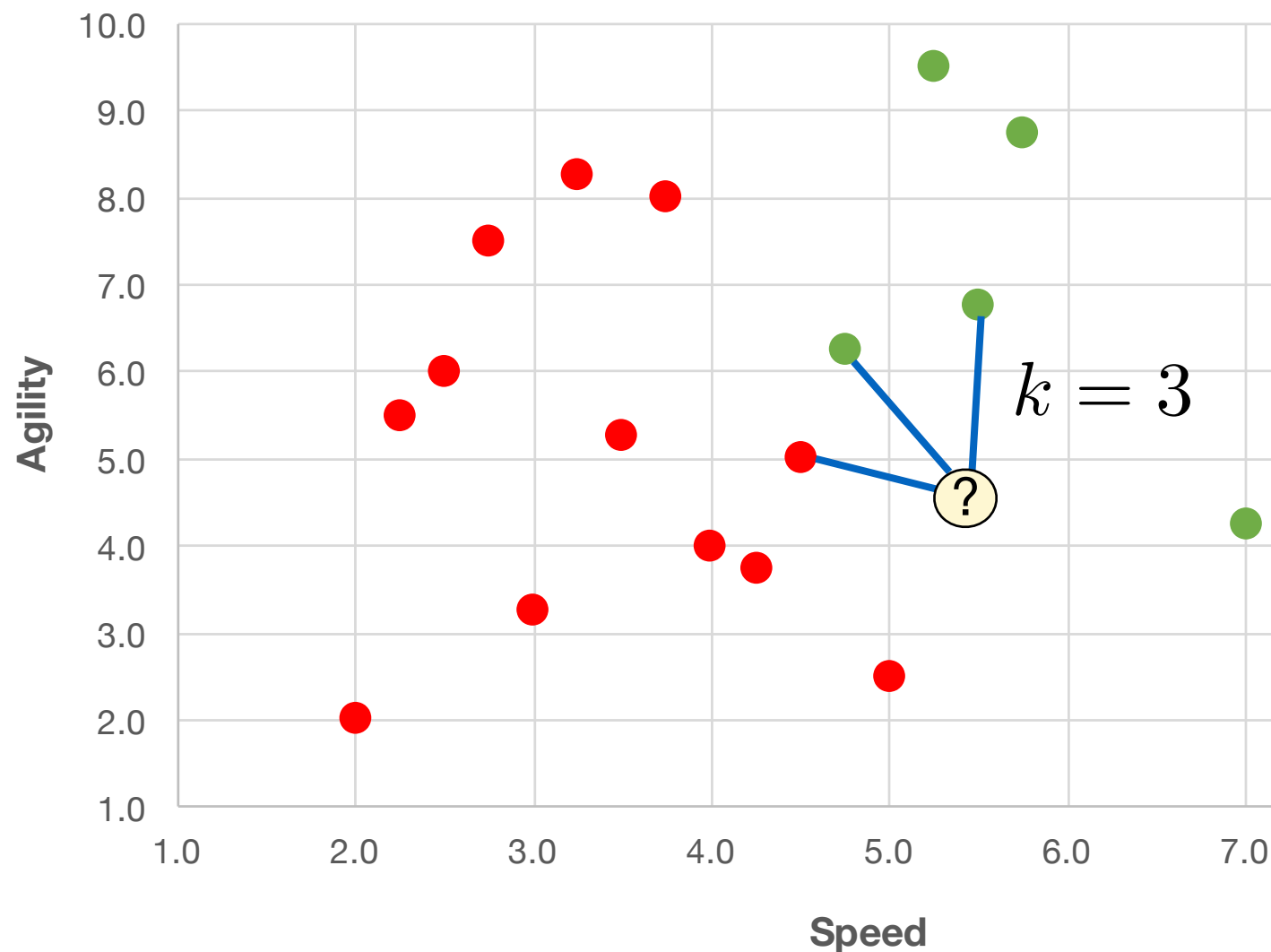
**$k$ -Nearest Neighbour Classifier:** Simple but effective "lazy" classifier. Find most similar previous examples for which a decision has already been made (i.e. their **nearest neighbours** from the training set).

**Example:** For new unseen examples, look at the label of  $k$  nearest neighbours under both features (e.g.  $k=3$  neighbours)



# KNN Classifier

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



Neighbour counts

- **Yes** = 2 votes

- **No** = 1 vote

➡ Majority says **Yes**!

# KNN Classifier in Python

- Scikit-learn has a range of different classification algorithms:  
[http://scikit-learn.org/stable/supervised\\_learning.html](http://scikit-learn.org/stable/supervised_learning.html)
- In all cases, we call the `fit()` function to build a model on training data and then the `predict()` function on unseen data.
- **Example:** Classification using the *Iris* flower dataset, which is built into scikit-learn.

```
from sklearn.datasets import load_iris  
iris = load_iris()
```
- When we inspect the dataset, we see that it has a numpy array of feature values (data) and a set of target labels (target).

```
from sklearn.datasets import load_iris
iris = load_iris()
```

```
print(iris.data)
```

```
[ [ 5.1    3.5    1.4    0.2 ]
  [ 4.9    3.     1.4    0.2 ]
  [ 4.7    3.2    1.3    0.2 ]
  [ 4.6    3.1    1.5    0.2 ]
  [ 5.     3.6    1.4    0.2 ]
  ... ]
```

```
print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

```
print(iris.target)
```

[illegible]

# KNN Classifier in Python

---

- Goal: Train a classifier to predict the species (class label) for a new numeric record describing a flower.
- Three possible species (class labels), so this is a multi-class problem: 'setosa', 'versicolor', or 'virginica'.
- 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(iris.data, iris.target)
```

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

```
xinput = np.array([[3.0, 5.0, 4.1, 2.0]])
pred_class_number = model.predict(xinput)
print( iris.target_names[pred_class_number] )
```

```
['virginica']
```

# KNN Classifier in Python

---

- We can also generate predictions for multiple input examples using a single call to `predict()` - i.e. apply classifier to a new test set.
- In general we will have 4 variables:
  1. `dataset_train`: The set of data on which to build the model.
  2. `target_train`: The true class labels for the training data.
  3. `dataset_test`: The set of data on which to test the model.
  4. `target_test`: The true class labels for the test data (if available).

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

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

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

Make a prediction for the test set

```
[ 1.  0.  0.  0.  1.  0.  0.  0.  0.  1.  0.  1. ]
```

A prediction is made for each  
input in the test set

```
print(target_test)
```

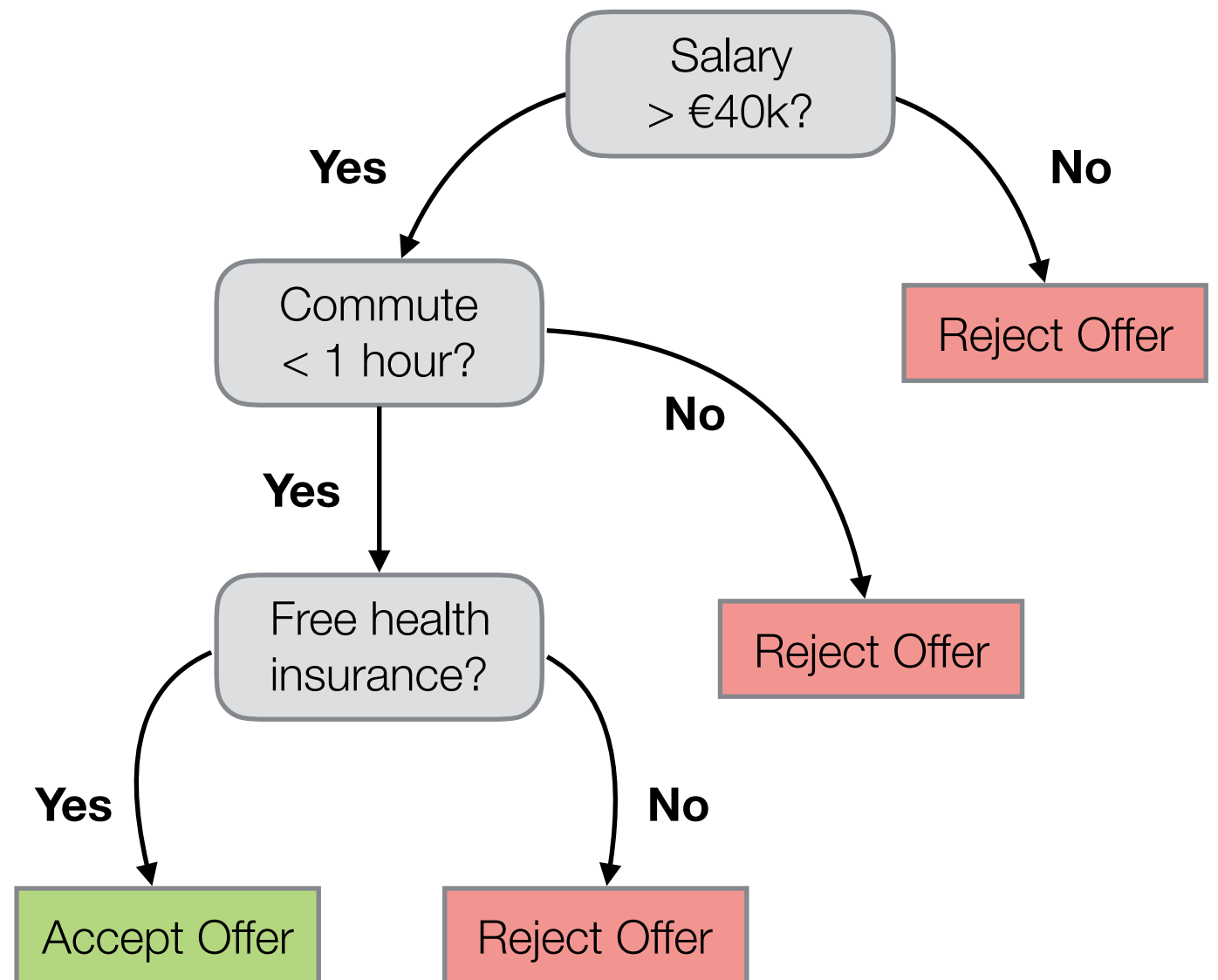
```
[ 1.  0.  0.  0.  1.  0.  0.  1.  1.  0.  0.  1. ]
```

We can compare the predictions to  
the true labels for the test data



# 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.*

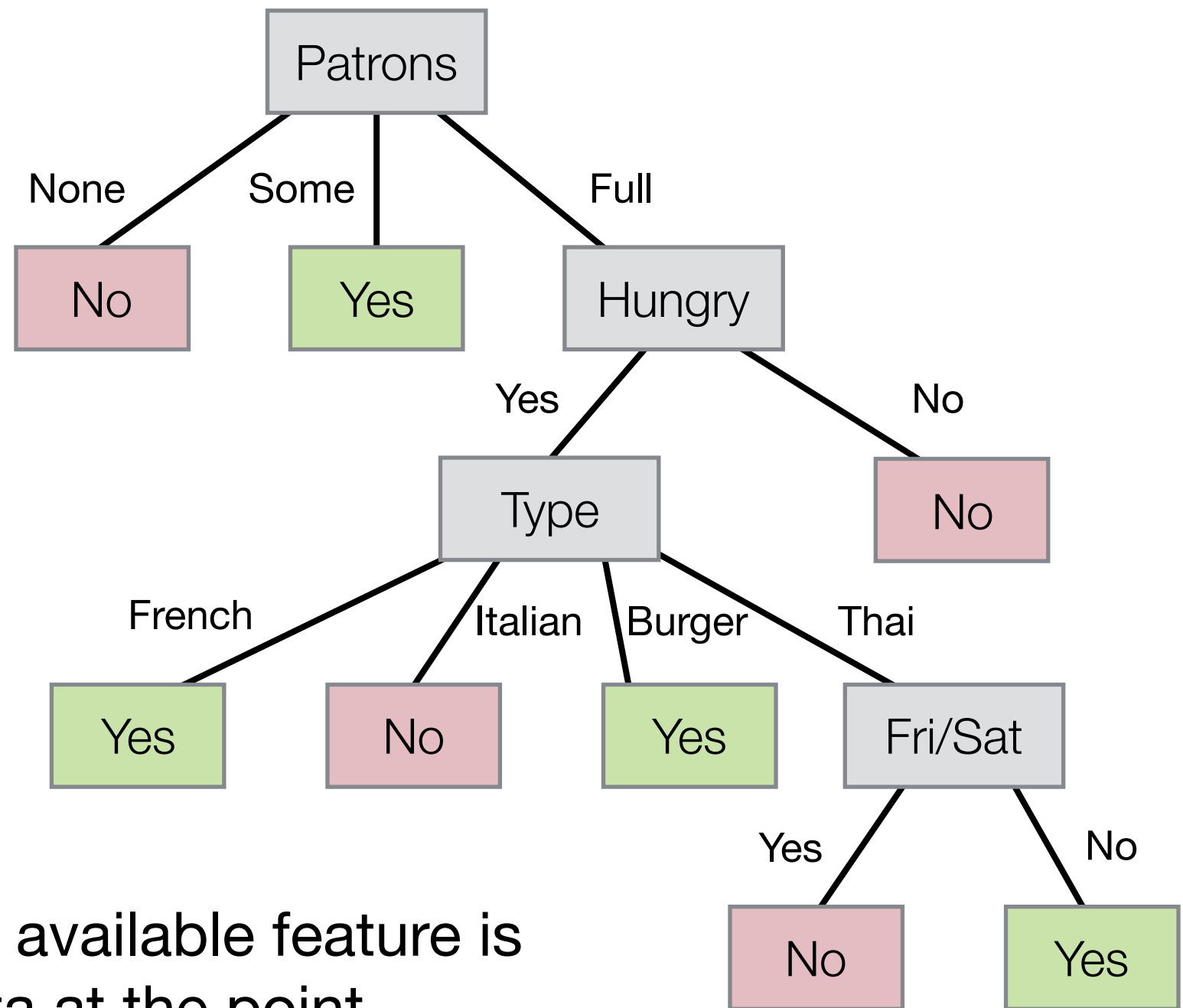
Binary classification task (WillWait = {Yes,No}), with examples described by 10 different features:

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.

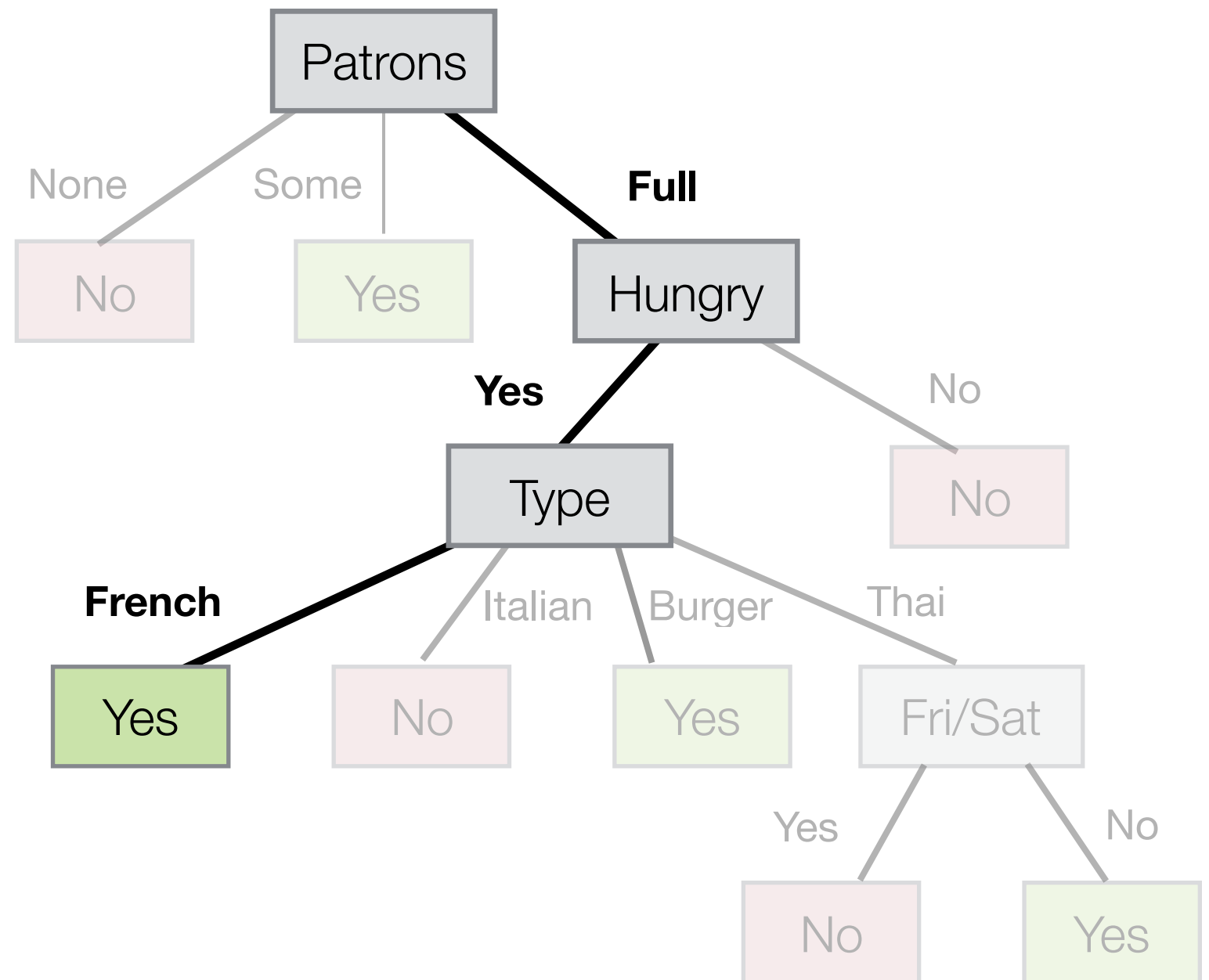


At each level, the best available feature is chosen to split the data at the point.

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

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

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

```
xinput = np.array([[2.9, 5.1, 4.1, 1.8]])
pred_class_number = model.predict(xinput)
print( iris.target_names[pred_class_number] )
```

```
['versicolor']
```

# Alternative Classifiers

---

- **Naive Bayes:** Simple classifier based on counts. Requires less training data, assumes all features are independent.
- **Support Vector Machines:** SVMs use a kernel function to accurately separate different classes. Difficult to interpret, can require parameter tuning.
- **Logistic Regression:** Probabilistic model, works well in an online setting.

```
from sklearn.naive_bayes import MultinomialNB
model = MultinomialNB()
model.fit(data_train, target_train)
predicted = model.predict(data_test)
```

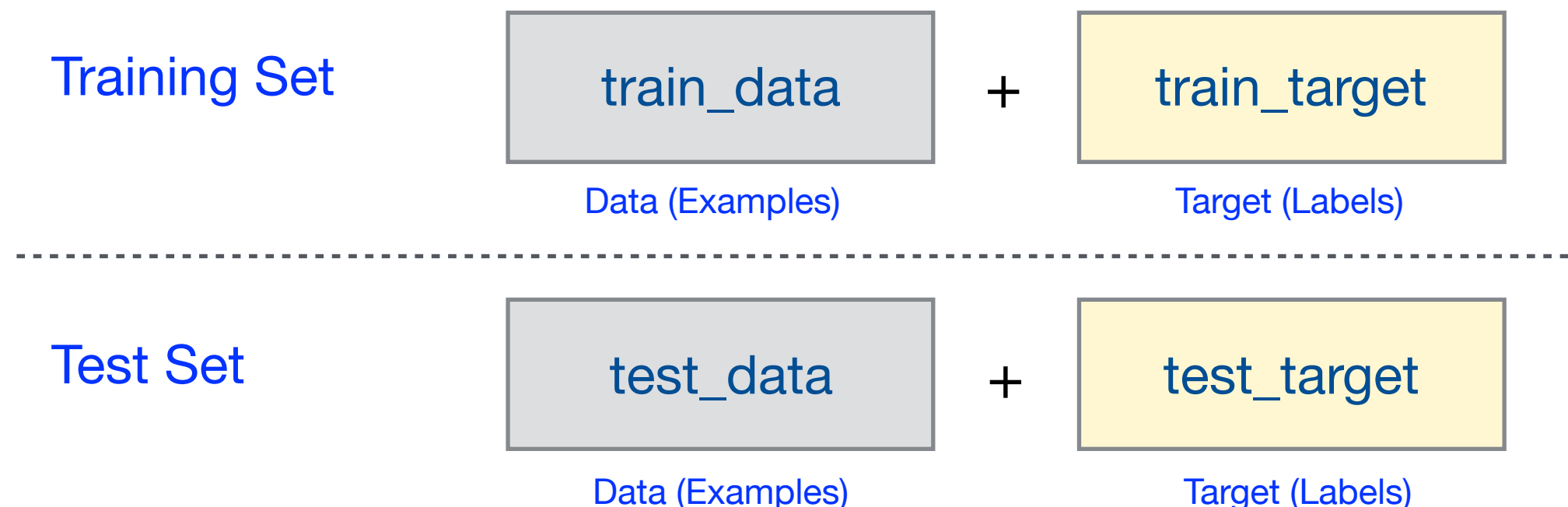
```
from sklearn.svm import SVC
model = SVC()
model.fit(data_train, target_train)
predicted = model.predict(data_test)
```

```
from sklearn import linear_model
model = linear_model.LogisticRegression()
model.fit(data_train, target_train)
predicted = model.predict(data_test)
```

# Evaluating Classifiers

---

- Reminder - Standard evaluation approach for classification tasks is to split the set of examples into two sets:
  1. **Training set:** 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 accuracy of the classifier.
- **Hold-out strategy:** Randomly hold back some examples (e.g. 20%) for evaluation. In Scikit-learn, this split involves 4 variables:



# Overfitting

---

- Why not just use all of the 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 evaluation functionality.
- To demonstrate, we will create an artificial dataset, where the examples have 2 binary class labels `[-1,1]`:

```
from sklearn.datasets import make_hastie_10_2
data, target = make_hastie_10_2(100)
```

100 examples in total, both the data and the target labels

```
from sklearn.model_selection import train_test_split
data_train, data_test, target_train, target_test =
train_test_split(data, target, test_size=0.2)
```

Randomly split the data into training and test sets. Test set size is 20% (0.2)

```
from sklearn.neighbors import KNeighborsClassifier
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 test data

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

```
0.65
```

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.

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

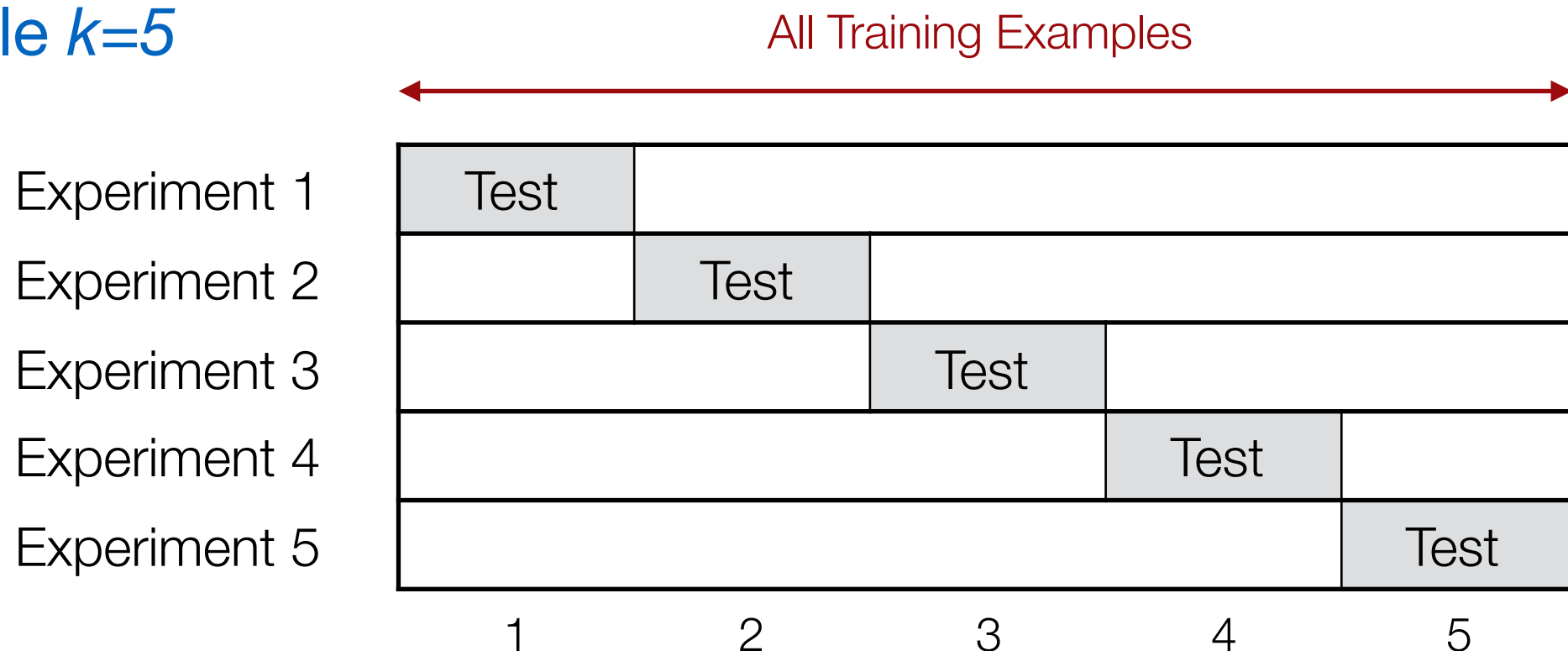
Accuracy is 65%, but confusion matrix shows high level of False Negatives.

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)
```

Create a single classifier

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

Apply to full data and target labels for 5 folds, calculate accuracy each time

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

Result is NumPy array containing 5 accuracy scores

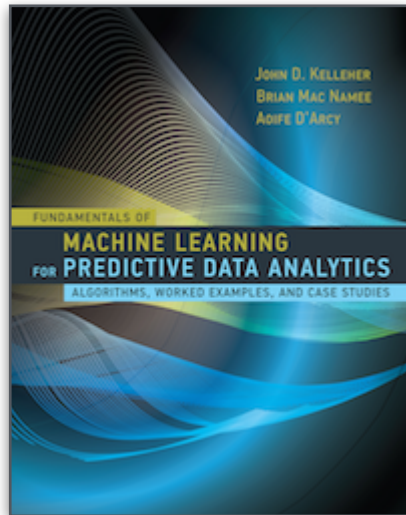
```
scores.mean()
```

Overall accuracy for the classifier is mean across folds

```
0.59
```

# Further Reading

---

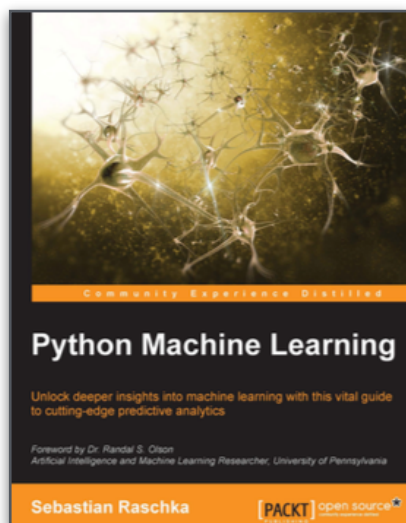
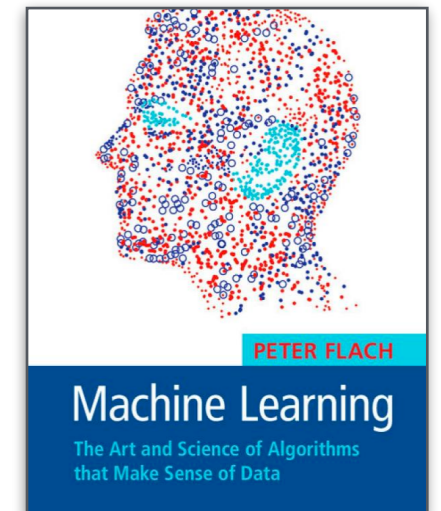


## *Fundamentals of Machine Learning for Predictive Data Analytics*

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*

Sebastian Raschka