Data Science: Principles and Practice

Lecture 3: Classification

Ekaterina Kochmar

UNIVERSITY OF CAMBRIDGE

Recap: Supervised Learning

Dataset: $\{< x^{(1)}, y^{(1)} >, < x^{(2)}, y^{(2)} >, ..., < x^{(m)}, y^{(m)} >\}$

Input features: $(x_1^{(i)}, x_2^{(i)}, ..., x_n^{(i)})$

Known (desired) outputs: $y^{(1)}, y^{(2)}, ..., y^{(m)}$

Our goal: Learn the mapping $f : X \to Y$
such that $y^{(i)} = f(x^{(i)})$ for all $i = 1, 2, ..., m$

Application: Learn the function on the training set, then use it to predict $\hat{y}^{(j)} = f(x^{(j)})$ for all x_j in the test set

Recap: Supervised Learning

Dataset: $\{< x^{(1)}, y^{(1)} >, < x^{(2)}, y^{(2)} >, ..., < x^{(m)}, y^{(m)} >\}$

Input features: $(x_1^{(i)}, x_2^{(i)}, ..., x_n^{(i)})$

Known (desired) outputs: $y^{(1)}, y^{(2)}, ..., y^{(m)}$

Our goal: Learn the mapping $f : X \to Y$
such that $y^{(i)} = f(x^{(i)})$ for all $i = 1, 2, ..., m$

Application: Learn the function on the training set, then use it to predict $\hat{y}^{(j)} = f(x^{(j)})$ for all x_j in the test set

Last time we looked into regression tasks, today – **classification**

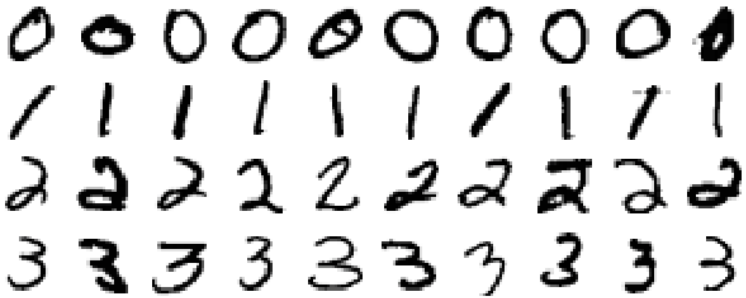
Recap: Regression vs. Classification

Regression tasks: the desired labels are continuous

Examples: House size, age, income → price
Weather conditions, time → number of rented bikes

Classification tasks: the desired labels are discrete

Examples: Pixel distribution in the image → digit label
Word distribution in movie reviews → sentiment (pos/neg/neut) label



Data Science: Principles and Practice

01 Binary classification

02 Data transformations

03 Model evaluation

04 Multi-class classification

05 Practical 2

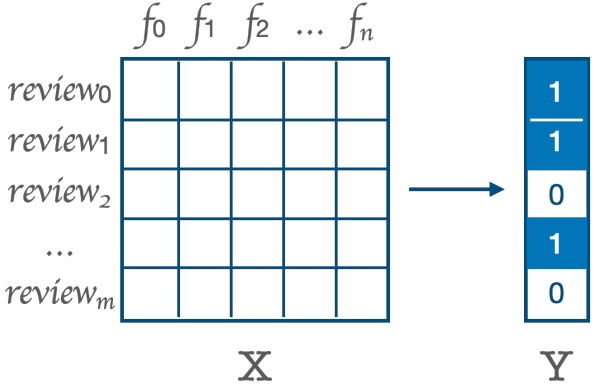
Binary classification

- Let's start with a simpler case – **binary classification** (i.e. distinguishing between 2 classes)

- **Task:** Sentiment analysis in movie reviews (Part IA CST Machine Learning and Real-world Data)

- **Data**: $m \times n$ matrix X with m reviews and n features (words)

- **Labels**: $y \in (0, 1)$ with 0 for *neg* and 1 for *pos*



Binary classification with Naïve Bayes

Naïve Bayes classifier:

- relies on probabilistic assumptions about the data
- makes “naïve” independence assumption about the features
- is fast and scalable compared to more sophisticated methods
- shows competitive results on a number of real-world tasks, despite its over-simplistic assumptions

Binary classification with Naïve Bayes

Prediction:

$$\hat{y}^{(i)} = argmax_{c \in (0,1)} p(y = c|x^{(i)}) = \begin{cases} 1, & \text{if } p(y = 1|x^{(i)}) > p(y = 0|x^{(i)}) \\ 0, & \text{otherwise} \end{cases}$$

where $x^{(i)} = (f_1^{(i)}, ..., f_n^{(i)})$ is the i-th review $x^{(i)}$ with its features $f_1^{(i)}, ..., f_n^{(i)}$

Flipping conditions: $\hat{p}(y = c|x^{(i)}) = \frac{p(c)p(x^{(i)}|c)}{p(x^{(i)})}$

where:

- $p(c)$ is the *prior*
- $p(x^{(i)}|c)$ is *likelihood*
- $p(x^{(i)})$ is *evidence* (note that it's irrelevant for the *argmax* estimation)
- $p(y = c|x^{(i)})$ is the *posterior*

Binary classification with Naïve Bayes

Naïve independence assumption:

$$p(f_1^{(i)}, ..., f_n^{(i)}|y) \approx \prod_{k=1}^{n} p(f_k^{(i)}|y)$$

conditional independence assumption – features are independent of each other given the class (naïve ⇒ do you think it always holds?)

Revised estimation: we've started with

$$\hat{y}^{(i)} = argmax_{c \in (0,1)} p(y = c|x^{(i)}) = \begin{cases} 1, & \text{if } p(y = 1|x^{(i)}) > p(y = 0|x^{(i)}) \\ 0, & \text{otherwise} \end{cases}$$

⇒ This is equivalent to

$$\hat{y}^{(i)} = argmax_{c \in (0,1)} p(y = c) \prod_{k=1}^{n} p(f_k^{(i)}|y = c)$$

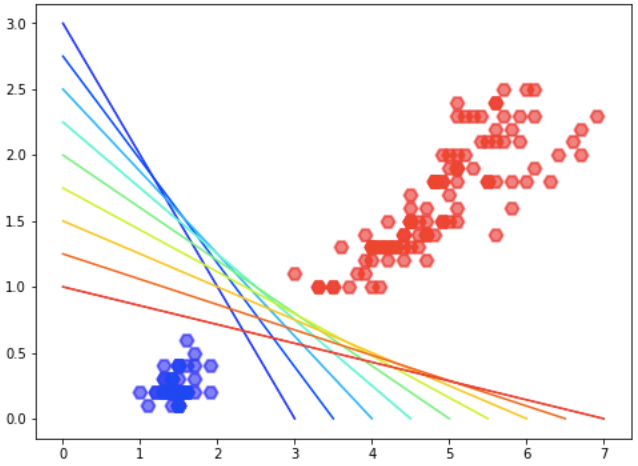
Practical notes on Naïve Bayes

- Probabilities $p(y = c)$ and $p(f_k^{(i)}|y = c)$ are estimated from the training data using *maximum a posteriori* (*MAP*) estimate
- Naïve Bayes models typically differ with respect to the assumptions about the distribution of features $p(x^{(i)}|y)$. Commonly used models include Gaussian NB, Multinomial NB, and Bernoulli NB.[1]

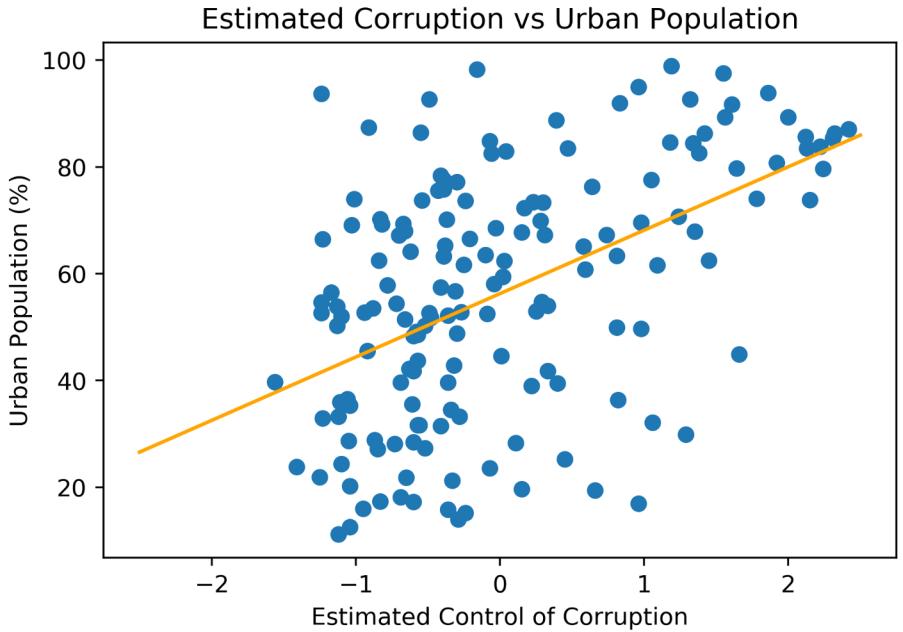
Recommended reading: A. McCallum and K. Nigam (1998). *A comparison of event models for Naïve Bayes text classification*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.1529

Linearly separable data

- **Linearly separable data** is the data where classes can be separated with a single line (or a *hyperplane* in a higher-dimensional Euclidean space)
- **Linear classifiers**, that try to learn a linear separation boundary between the classes, are well-suited for such data
- **Examples**: Logistic Regression, Perceptron, Support Vector Machines



Recap: Linear Regression



$$y = ax + b$$

Controls the angle

Controls the intercept

Logistic Regression vs Linear Regression

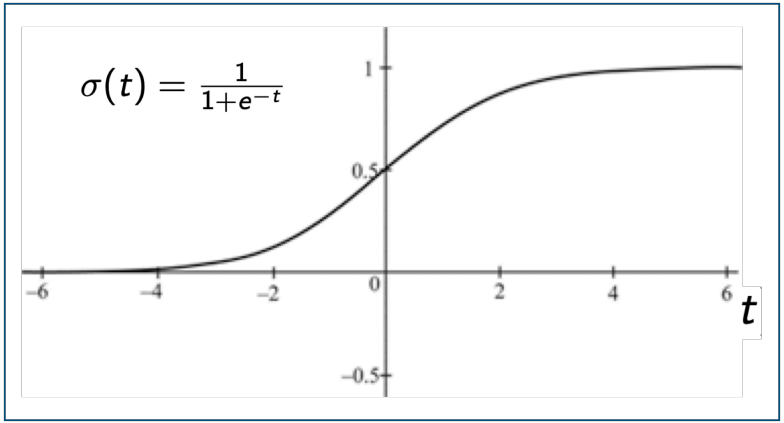
- Despite the similarity in name, Logistic Regression outputs a *discrete value*, i.e. it is used for *classification*
- Logistic Regression estimates whether the probability of an instance $\mathbf{x}^{(i)}$ belonging to class **c** is greater than 0.5. If it is, the instance is classified as **c**; otherwise it is classified as **¬c**.

$$\hat{y} = \begin{cases} c, & \text{if } p(x^{(i)} \in c) \geq 0.5 \\ \neg c, & \text{otherwise} \end{cases}$$

Logistic Regression

- First, estimate $w \cdot X$ as before, where w is the weight vector $(w_0, w_1, \ldots, w_n)$
- Next, apply a *sigmoid function* to the result $\hat{p} = \sigma(w \cdot X)$ where

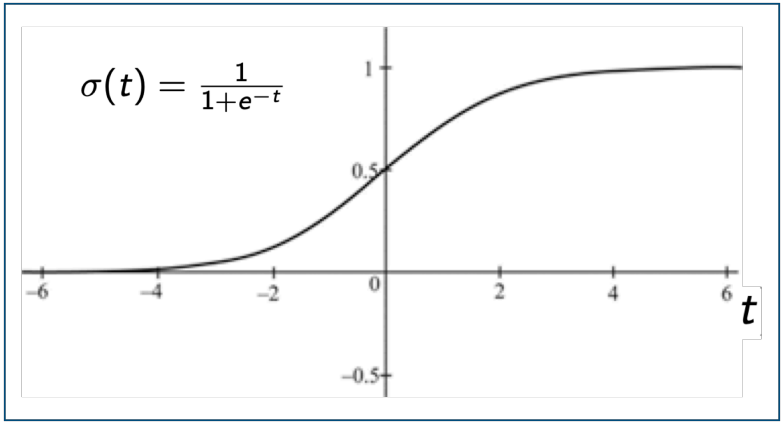
$$\sigma(t) = \frac{1}{1+exp(-t)} \qquad = t$$



Prediction step with Logistic Regression

Estimate whether the probability of an instance $\mathbf{x}^{(i)}$ belonging to class **c** is greater than 0.5, i.e.:

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{p} \geq 0.5 \\ 0, & \text{otherwise} \end{cases} \Rightarrow \hat{y} = \begin{cases} 1, & \text{if } \frac{1}{1+exp(-t)} \geq 0.5 \\ 0, & \text{otherwise} \end{cases} \Rightarrow \hat{y} = \begin{cases} 1, & \text{if } t \geq 0 \\ 0, & \text{otherwise} \end{cases}$$



Training Logistic Regression

- **Learning objective**: we need to learn the weights *w* such that
  - has a high positive value for *y = 1*, and
  - has a high negative value for *y = 0*
- The following **cost function** answers this objective:

$$c(w) = \begin{cases} -log(\hat{p}), & \text{if } y = 1 \\ -log(1 - \hat{p}), & \text{if } y = 0 \end{cases}$$

Training Logistic Regression

- **Learning objective**: we need to learn the weights *w* such that
  - has a high positive value for *y = 1*, and
  - has a high negative value for *y = 0*
- The following **cost function** answers this objective:

$$c(w) = \begin{cases} -log(\hat{p}), & \text{if } y = 1 \\ -log(1-\hat{p}), & \text{if } y = 0 \end{cases} \quad \Leftrightarrow \quad c(w) = \begin{cases} -log(\hat{p}) \to 0, & \text{if } y = 1 \text{ and } \hat{p} \to 1 \\ -log(\hat{p}) \to \infty, & \text{if } y = 1 \text{ and } \hat{p} \to 0 \\ -log(1-\hat{p}) \to 0, & \text{if } y = 0 \text{ and } \hat{p} \to 0 \\ -log(1-\hat{p}) \to \infty, & \text{if } y = 0 \text{ and } \hat{p} \to 1 \end{cases}$$

Log-loss function

- Cost function over the whole training set:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(\hat{p}^{(i)}) + (1-y^{(i)})log(1-\hat{p}^{(i)})]$$

- No closed form solution for *w* that minimises this cost function, but since the function is convex, `Gradient Descent` (see previous lecture) can be used to find optimal weights: e.g. partial derivative of the cost function wrt the *j*-th model parameter is

$$\begin{aligned}\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{1}{m}\sum_{i=1}^{m}(\hat{p}^{(i)} - y^{(i)})x_j^{(i)} \\ &= \frac{1}{m}\sum_{i=1}^{m}(\sigma(\theta^T \cdot x^{(i)}) - y^{(i)})x_j^{(i)}\end{aligned}$$

Recap: the Gradient

It may be more convenient to work with vector notation.

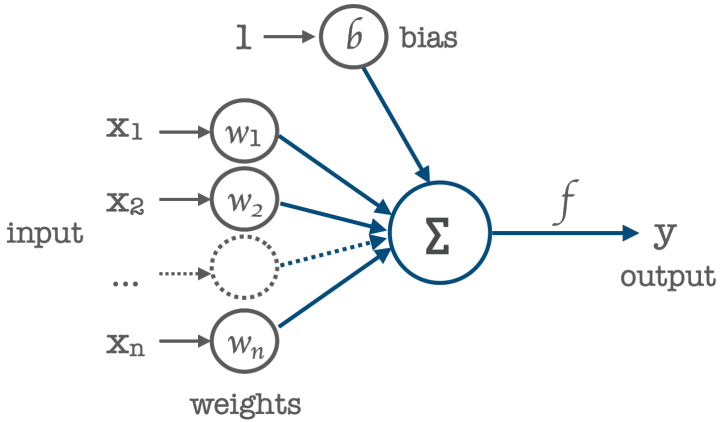
The gradient is a vector of all partial derivatives.

For a function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is

$$\nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \frac{\partial f(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_n} \end{bmatrix}$$

Single-layer perceptron

$$\hat{y}^{(i)} = \begin{cases} 1, & \text{if } w \cdot x^{(i)} + b > 0 \\ 0, & \text{otherwise} \end{cases}$$



where:

- $w \cdot x^{(i)}$ is the dot product of the weight vector w and the feature vector $x^{(i)}$ for instance i , i.e.

$$\sum_{j=1}^{n} w_j x_j^{(i)}$$

- b is the bias term

Single-layer perceptron training

Initialisation: Initialise the weights $w = (w_1, \ldots, w_n)$ and the bias term $b = w_0$ to some value (e.g., 0 or some other small value)

Single-layer perceptron training

Initialisation: Initialise the weights $w = (w_1, ..., w_n)$ and the bias term $b = w_0$ to some value (e.g., 0 or some other small value)

Estimation at time ***t*** for each instance ***i***:

$$\hat{y}^{(i)}(t) = f(w(t) \cdot x^{(i)}) = f(w_0(t) + w_1(t)x_1^{(i)} + ... + w_n(t)x_n^{(i)})$$

Single-layer perceptron training

Initialisation: Initialise the weights $w = (w_1, ..., w_n)$ and the bias term $b = w_0$ to some value (e.g., 0 or some other small value)

Estimation at time ***t*** for each instance ***i***:

$$\hat{y}^{(i)}(t) = f(w(t) \cdot x^{(i)}) = f(w_0(t) + w_1(t)x_1^{(i)} + ... + w_n(t)x_n^{(i)})$$

Update for the weights at time (***t + 1***) for each instance ***i*** and each feature $0 \leq j \leq n$

$$w_j(t+1) = w_j(t) + r(y^{(i)} - \hat{y}^{(i)}(t))x_j^{(i)}$$ where *r* is a predefined learning rate

Single-layer perceptron training

Initialisation: Initialise the weights $w = (w_1, ..., w_n)$ and the bias term $b = w_0$ to some value (e.g., 0 or some other small value)

Estimation at time ***t*** for each instance ***i***:

$$\hat{y}^{(i)}(t) = f(w(t) \cdot x^{(i)}) = f(w_0(t) + w_1(t)x_1^{(i)} + ... + w_n(t)x_n^{(i)})$$

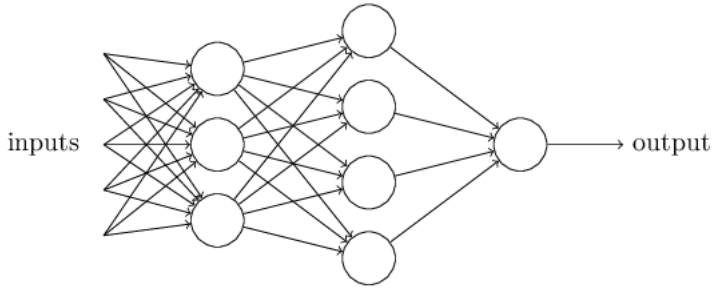
Update for the weights at time (***t + 1***) for each instance ***i*** and each feature $0 \leq j \leq n$

$$w_j(t+1) = w_j(t) + r(y^{(i)} - \hat{y}^{(i)}(t))x_j^{(i)}$$ where *r* is a predefined learning rate

Stopping criteria: convergence to an error below a predefined threshold γ, or after a predefined number of iterations $t \leq T$

Notes on single-layer perceptron

- If the data is **linearly separable**, the perceptron algorithm is guaranteed to converge
- If the data is **not linearly separable**, the perceptron will never be able to find a solution to separate the classes in the training data
- A **single-layer perceptron** is a simple linear classifier, often used to illustrate the simplest feedforward neural network.
- **Multilayer perceptrons** combine multiple layers and use non-linear activation functions, which makes them capable of classifying data that is not linearly separable (more on this in later lectures)

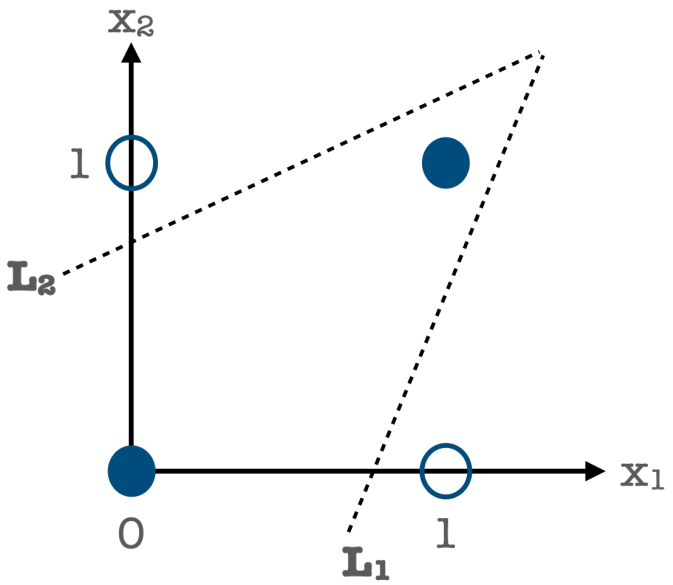


Non-linearly separable data

Consider the following classic example of the XOR problem $y = x_1 \oplus x_2$

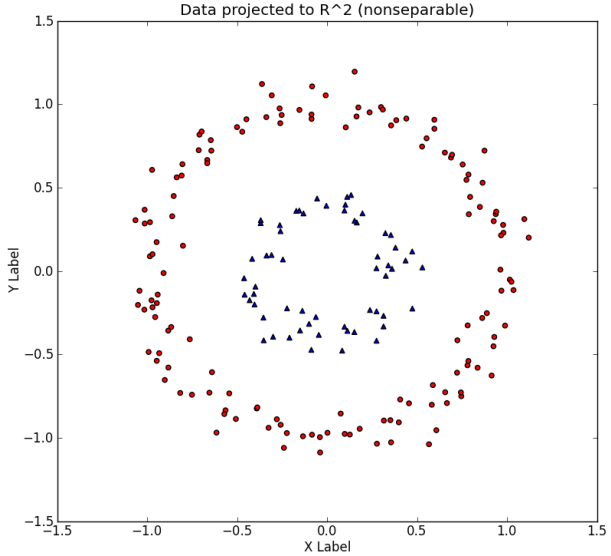
x_1	x_2	y
0	0	0
0	1	**1**
1	0	**1**
1	1	0

$y = x_1 \oplus x_2$

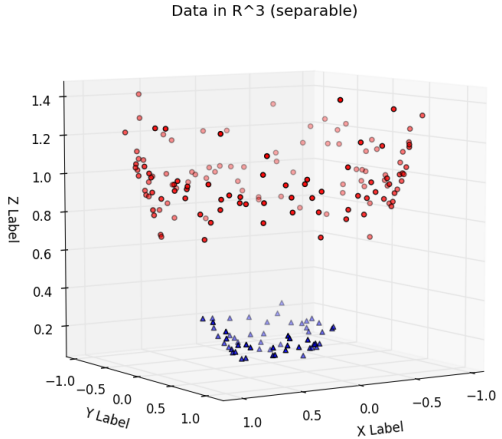


Non-linearly separable data

Actual (raw) data: two classes non-linearly separable

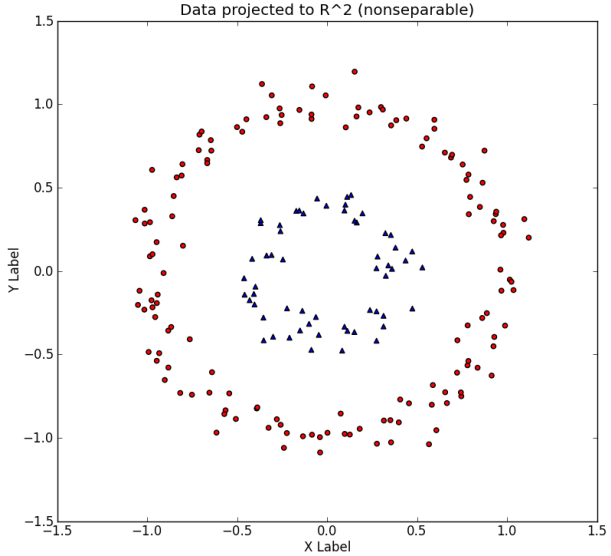


Objective: transform the data using additional dimensions such that it becomes possible to separate the classes linearly

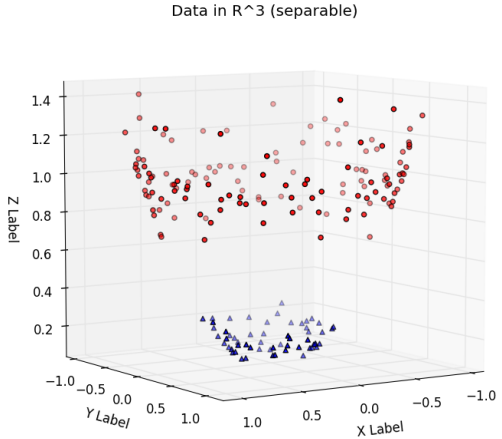


Non-linearly separable data

Actual (raw) data: two classes non-linearly separable



Objective: transform the data using additional dimensions such that it becomes possible to separate the classes linearly

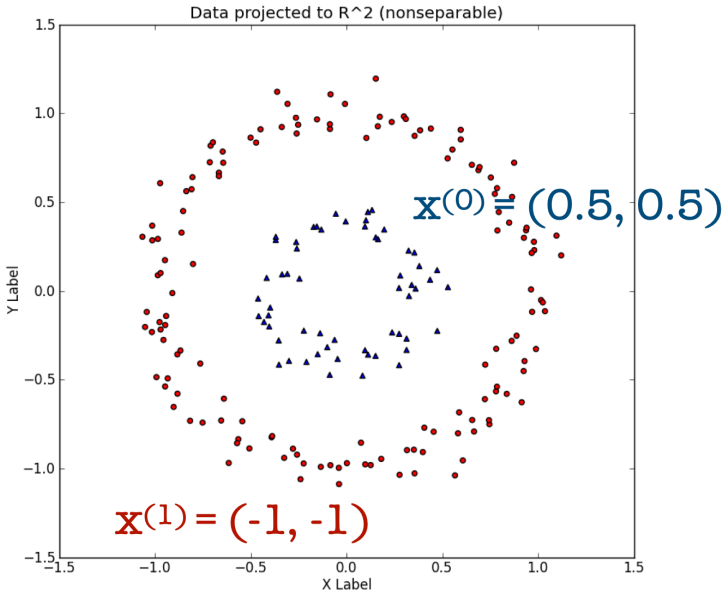


Method: data transformations / feature maps that transform the data into higher dimensional space (e.g. *kernel trick*)

Toy example

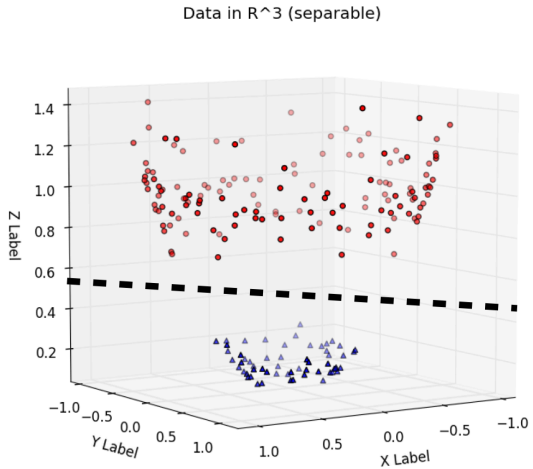
Suppose non-linearly separable classes 0 and 1 such that:

- $x^{(0)} = (0.5, 0.5)$ – blue dot
- $x^{(1)} = (-1, -1)$ – red dot



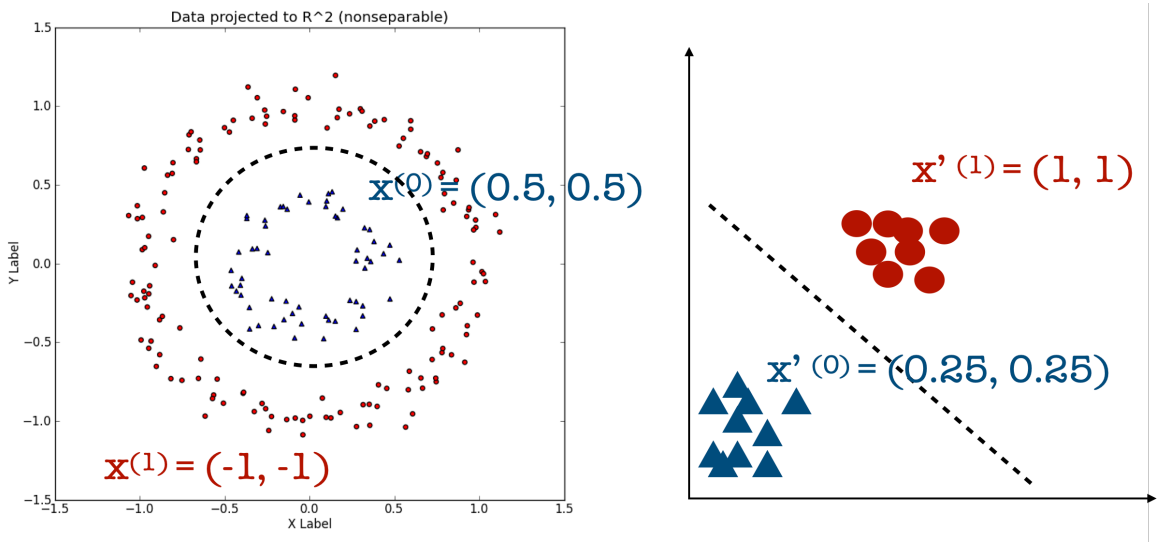
Consider using a square function:

- $x^{(0)} \rightarrow x'^{(0)} = (0.25, 0.25)$
- $x^{(1)} \rightarrow x'^{(1)} = (1, 1)$



Kernel trick and feature maps

- With the new data representation, the instances of class 0 (blue) end up in the lower left corner, and the instances of class 1 (red) end up in the upper right corner
- *Kernel trick* and feature maps allow us to cast the original data into higher dimensional data: e.g., $(x, y) \rightarrow (x^2, xy, y^2)$



Performance measures: Accuracy

- **Task**: suppose you select a digit in the handwritten digit dataset (e.g., 5). You perform a binary classification task of detecting 5 vs ¬5 in a balanced dataset of 10 digits

- **Evaluation**: the most straightforward way to evaluate the results is to calculate the proportion of correct predictions, i.e.:

$$ACC = \frac{num(\hat{y}==y)}{num(\hat{y}==y)+num(\hat{y}!=y)}$$

Performance measures: Accuracy

- **Task**: suppose you select a digit in the handwritten digit dataset (e.g., 5). You perform a binary classification task of detecting 5 vs ¬5 in a balanced dataset of 10 digits

- **Evaluation**: the most straightforward way to evaluate the results is to calculate the proportion of correct predictions, i.e.:

$$ACC = \frac{num(\hat{y}==y)}{num(\hat{y}==y)+num(\hat{y}!=y)}$$

Suppose you get an accuracy of 91%. Is this a good accuracy score?

What accuracy score is missing

- Note that if the classifier always predicts ¬5 (i.e., essentially does nothing), on a balanced dataset with 10 digits its accuracy will be *ACC=90%*
- It is also unclear what exactly the classifier gets wrong

What accuracy score is missing

- Note that if the classifier always predicts ¬5 (i.e., essentially does nothing), on a balanced dataset with 10 digits its accuracy will be *ACC=90%*
- It is also unclear what exactly the classifier gets wrong, e.g. all of the following classifiers get *ACC=90%*, yet their decisions (and errors) are very different:

	predicted ¬5	predicted 5
actual ¬5	90	0
actual 5	10	0

Classifier 1 detects only ¬5's

	predicted ¬5	predicted 5
actual ¬5	85	5
actual 5	5	5

Classifier 2 detects some 5's

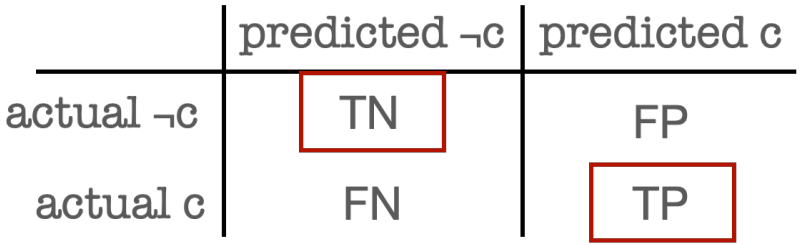
	predicted ¬5	predicted 5
actual ¬5	80	10
actual 5	0	10

Classifier 3 detects all 5's

Confusion matrix

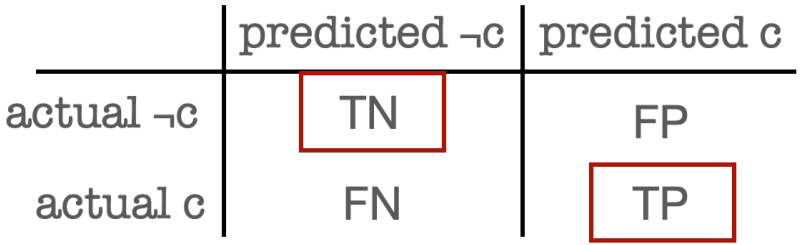
	predicted ¬c	predicted c
actual ¬c	TN	FP
actual c	FN	TP

Confusion matrix



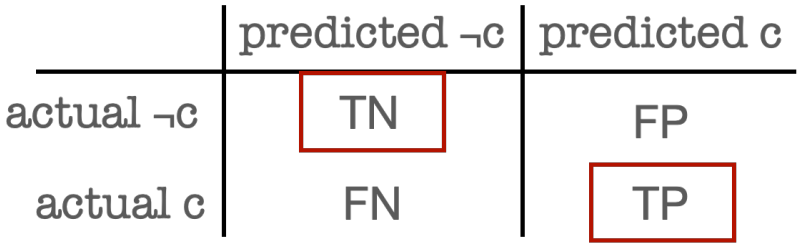
- **True negatives** (**TN**) – actual instances of ¬5 correctly classified as ¬5

Confusion matrix



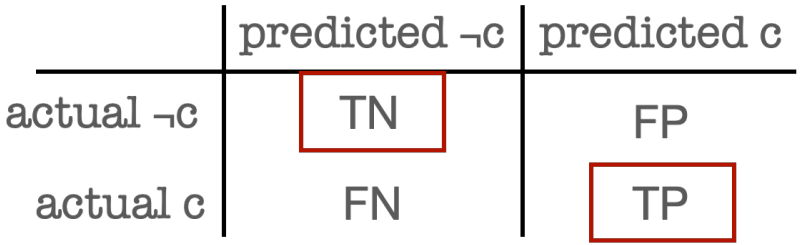
- **True negatives** (**TN**) – actual instances of ¬5 correctly classified as ¬5
- **False negatives** (**FN**) – actual instances of 5 missed by the classifier

Confusion matrix



- **True negatives** (**TN**) – actual instances of ¬5 correctly classified as ¬5
- **False negatives** (**FN**) – actual instances of 5 missed by the classifier
- **True positives** (**TP**) – actual instances of 5 correctly classified as 5

Confusion matrix



- **True negatives** (**TN**) – actual instances of ¬5 correctly classified as ¬5
- **False negatives** (**FN**) – actual instances of 5 missed by the classifier
- **True positives** (**TP**) – actual instances of 5 correctly classified as 5
- **False positives** (**FP**) – actual instances of ¬5 misclassified as 5

Performance measures

- **Accuracy:** proportion of correctly classified instances

$$ACC = \frac{TP+TN}{TP+TN+FP+FN}$$

Performance measures

- **Accuracy:** proportion of correctly classified instances

$$ACC = \frac{TP+TN}{TP+TN+FP+FN}$$

- **Precision**: “trustworthiness” of your classifier when it predicts class *c*

$$P = \frac{TP}{TP+FP}$$

Performance measures

- **Accuracy:** proportion of correctly classified instances

$$ACC = \frac{TP+TN}{TP+TN+FP+FN}$$

- **Precision**: “trustworthiness” of your classifier when it predicts class *c*

$$P = \frac{TP}{TP+FP}$$

- **Recall**: “coverage” with respect to the class *c*

$$R = \frac{TP}{TP+FN}$$

Performance measures

- **Accuracy:** proportion of correctly classified instances

$$ACC = \frac{TP+TN}{TP+TN+FP+FN}$$

- **Precision**: “trustworthiness” of your classifier when it predicts class *c*

$$P = \frac{TP}{TP+FP}$$

- **Recall**: “coverage” with respect to the class *c*

$$R = \frac{TP}{TP+FN}$$

- **F_1-score:** harmonic mean between precision and recall

$$F_1 = 2 \times \frac{P \times R}{P+R} \; [F_\beta = (1+\beta^2) \times \frac{P \times R}{\beta^2 \times P+R}]$$

Precision-recall trade-off

Most likely your classifier won't show both perfect precision and recall:

- You can reach perfect precision by identifying a single instance of class $c \Rightarrow$ low recall
- You can reach perfect recall by always predicting class $c \Rightarrow$ low precision

Some tasks require higher recall and some higher precision, e.g.:

- Detection of a potentially cancerous case that needs further tests?

Precision-recall trade-off

Most likely your classifier won't show both perfect precision and recall:

- You can reach perfect precision by identifying a single instance of class *c* ⇒ low recall
- You can reach perfect recall by always predicting class *c* ⇒ low precision

Some tasks require higher recall and some higher precision, e.g.:

- Detection of a potentially cancerous case that needs further tests? → **recall**
- Detection of suspicious activity on a credit card?

Precision-recall trade-off

Most likely your classifier won't show both perfect precision and recall:

- You can reach perfect precision by identifying a single instance of class c ⇒ low recall
- You can reach perfect recall by always predicting class c ⇒ low precision

Some tasks require higher recall and some higher precision, e.g.:

- Detection of a potentially cancerous case that needs further tests? → **recall**
- Detection of suspicious activity on a credit card? → **recall**
- Automated change of drug dosage for a hospital patient?

Precision-recall trade-off

Most likely your classifier won't show both perfect precision and recall:

- You can reach perfect precision by identifying a single instance of class c ⇒ low recall
- You can reach perfect recall by always predicting class c ⇒ low precision

Some tasks require higher recall and some higher precision, e.g.:

- Detection of a potentially cancerous case that needs further tests? → **recall**
- Detection of suspicious activity on a credit card? → **recall**
- Automated change of drug dosage for a hospital patient? → **precision**
- Detection of videos safe for kids?

Precision-recall trade-off

Most likely your classifier won't show both perfect precision and recall:

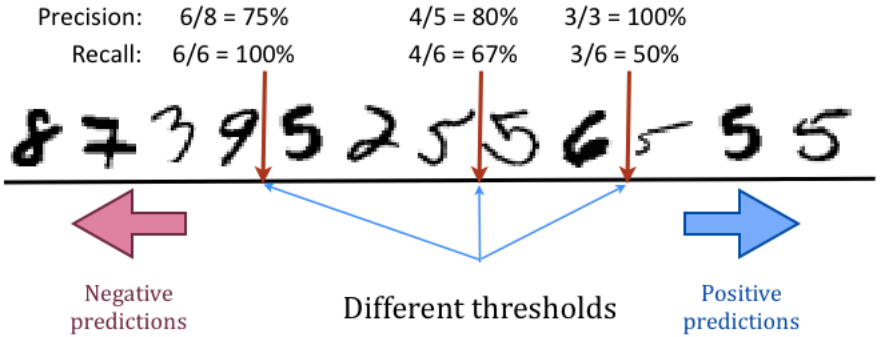
- You can reach perfect precision by identifying a single instance of class c ⇒ low recall
- You can reach perfect recall by always predicting class c ⇒ low precision

Some tasks require higher recall and some higher precision, e.g.:

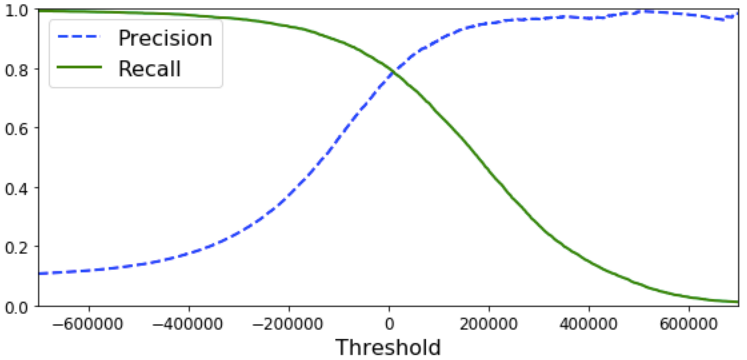
- Detection of a potentially cancerous case that needs further tests? → **recall**
- Detection of suspicious activity on a credit card? → **recall**
- Automated change of drug dosage for a hospital patient? → **precision**
- Detection of videos safe for kids? → **precision**

Confidence thresholds and P-R curve

By changing your classifier's confidence threshold you can change how conservative it should be in its decisions (the more conservative, the higher its precision and lower its recall)

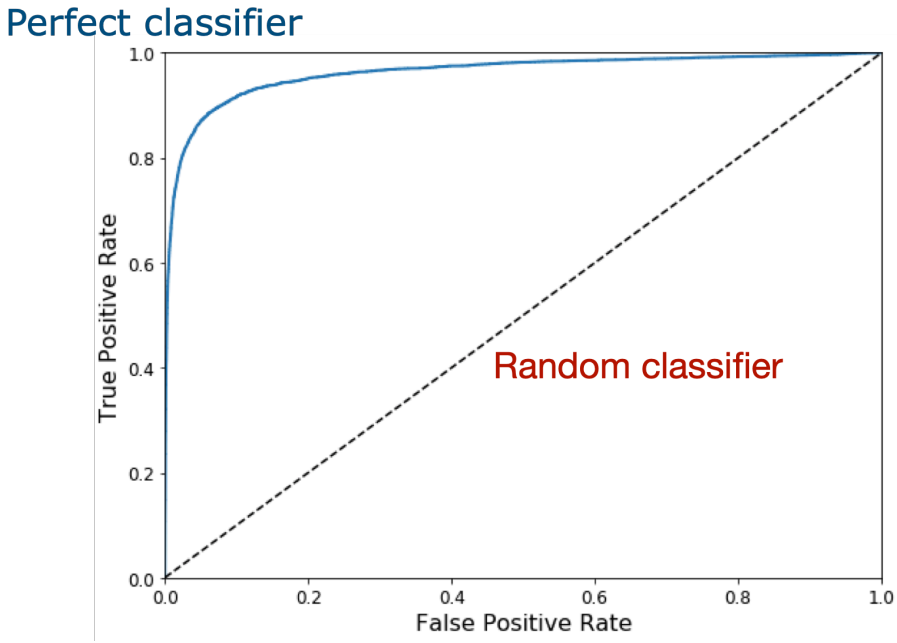


You can plot precision and recall as functions of the threshold value



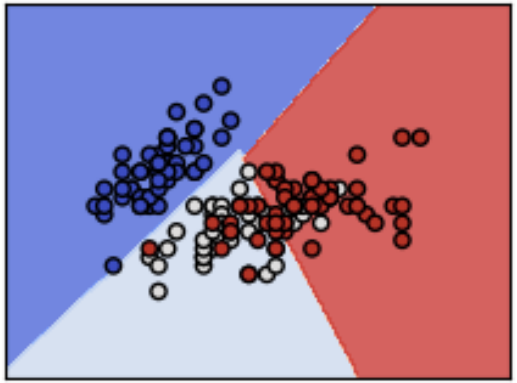
Receiver Operating Characteristic (ROC)

- **Specificity**: $\frac{TN}{TN+FP}$
- **False positive rate** (**FPR**) / fall-out / probability of false alarm = (1– *specificity*)
- **True positive rate** (**TPR**) / sensitivity / probability of detection = **recall**
- **Area under the curve (AUC)** – close to 1.0 for the perfect classifier



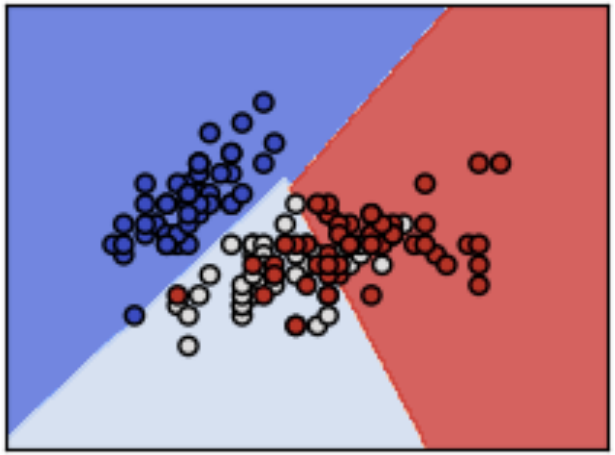
Multiclass classification

- So far, we have been looking into binary classification (distinguishing between exactly two classes)
- Some classifiers naturally allow for multiple classes, e.g. Naïve Bayes – simply output the most probable class
- Linear classifiers are strictly binary classifiers, so how can they handle multiple classes?



Two strategies for linear classifiers

- **One-vs-all** (**OvA**) or **one-vs-rest** (**OvR**):
  - Train *n* binary classifiers to detect one class each (e.g., 10 binary digit detectors)
  - At test time, output the class with the highest score
- **One-vs-one** (**OvO**):
  - Train $\frac{N(N-1)}{2}$ binary class-vs-class classifiers (e.g., 45 binary digit-vs-digit classifiers)
  - At test time, output the class that wins most of the time

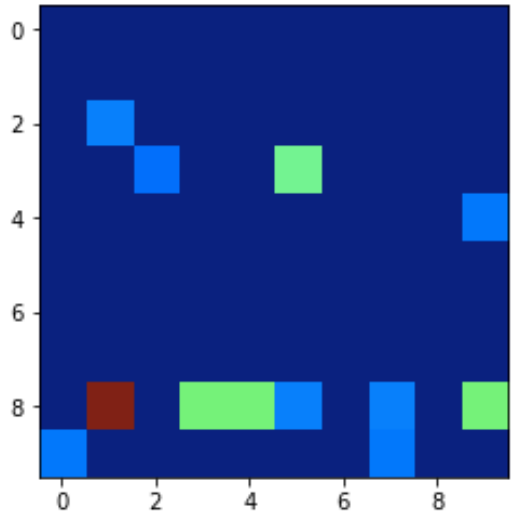


Error analysis for multiclass classification

Confusion matrix

```
array([[36,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 36,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  1, 34,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  1, 34,  0,  2,  0,  0,  0,  0],
       [ 0,  0,  0,  0, 35,  0,  0,  0,  0,  1],
       [ 0,  0,  0,  0,  0, 37,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0, 36,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 36,  0,  0],
       [ 0,  4,  0,  2,  2,  1,  0,  1, 23,  2],
       [ 1,  0,  0,  0,  0,  0,  0,  1,  0, 34]])
```

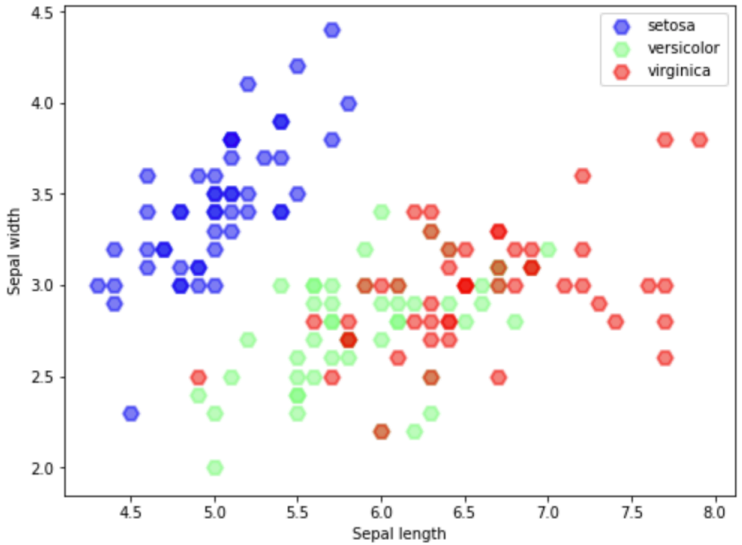
Confusion heatmap

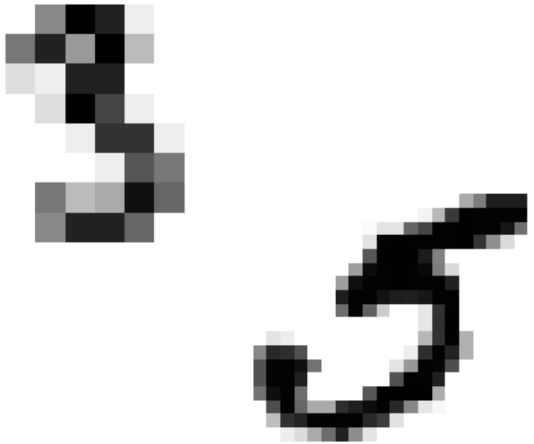


Practical 2

Data

- **Two datasets:** iris flower dataset (150 samples, 3 classes, 4 features), and the hand-written digits dataset (≈ 1.8K samples, 10 classes, 64 features)





Your task: Learning objectives

- Learn about binary and multiclass classification in practice

- Investigate whether data is linearly separable, and what to do when it is not

- Apply 3 classifiers discussed in this lecture

- Focus on evaluation of the classifiers

- One dataset is used to illustrate the ML techniques; your task is to implement the above steps for the other one

Practical 2 Logistics

- Data and code for Practical 2 can be found on: Github (https://github.com/ekochmar/cl-datasci-pnp-2021/tree/master/DSPNP_practical2)

- Practical ('ticking') session over Zoom at the time allocated by your demonstrator

- At the practical, be prepared to discuss the task and answer the questions about the code to get a 'pass'

- Upload your solutions (Jupyter notebook or Python code) to Moodle by the deadline (Thursday 12 November, 4pm)