Week 9: Classification

Classification Algorithms

Dr Giuseppe Brandi

Northeastern University London

What is classification?

A type of supervised learning, where each input example has a known target.

- Given an input vector x, assign to it one of K discrete classes (or labels) C_k, $k = 1, 2, \ldots K$
- The mapping is unique (one class only)

Geometrically, we can view classification as dividing the input space into *regions*.

Example problems

Example, real-world problems include:

Application	*Input*	*Output*
Spam detection	E-mails	True or false
Text recognition	Images	Characters & digits
Diagnostics	Symptoms	Disease(s)
Fradulent activity	Transactions	True or false

One-hot encodings

It is convenient to encode each class (or label) as "1-of-K":

- Each label is represented as a vector of length K
- The i-th label is a vector of zeros with a 1 at position i

One-hot encodings

For example, if $C = \{\text{cat}, \text{dog}, \text{mouse}\}$, we can define:

- $\text{cat} = (1, 0, 0)$,
- $\text{dog} = (0, 1, 0)$, and
- $\text{mouse} = (0, 0, 1)$

Intuitively, we can interpret such vectors as the probability of an input, say x, being a cat, dog, or a mouse:

$$\Big(p(\text{cat}|\text{x}), p(\text{dog}|\text{x}), p(\text{mouse}|\text{x})\Big)$$

The Linear Probability Model (LPM)

The Linear Probability Model (LPM) is a simple model, wich entails applying linear regression to binary outcomes:

- **LPM model:**

$$P(y = 1 \mid x) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_k x_k$$

- Each coefficient w_i represents the change in the probability of $y = 1$ for a one-unit change in x_i, assuming a linear relationship.
- **Interpreting Coefficients:** If $w_1 = 0.2$, a one-unit increase in x_1 raises the probability of $y = 1$ by 20%.
- **Intercept:** Represents the base probability when all predictors are zero.

Advantages of Linear Probability Model (LPM)

- **Simplicity**: Uses ordinary least squares (OLS) linear regression, easy to implement and understand.
- **Interpretability**: Coefficients represent marginal effects, providing straightforward interpretation (e.g., a 0.2 coefficient implies a 20% increase in probability).
- **Computational Efficiency**: LPM can be calculated quickly, even for large datasets.

Disadvantages of Linear Probability Model (LPM)

- **Probability Bounds**: Predictions can fall outside the 0-1 range, producing probabilities below 0 or above 1.
- **Non-Linearity**: Many relationships in binary outcomes are nonlinear, making LPM less accurate.
- **Sensitivity to Outliers**: Outliers can disproportionately affect predictions due to linearity.

Logistic regression

We know from linear regression models that:

- $y(\mathrm{x}, \mathrm{w})$ is a linear function of parameters w
- $y(\mathrm{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots = w_0 + \mathrm{w}^T \mathrm{x}$

But $y(\mathrm{x})$ is a real number, and we want to predict discrete labels; or better, $p(C_k|\mathrm{x})$ that are between 0 and 1.

Logistic regression

But $y(\mathrm{x})$ is a real number, and we want to predict discrete labels; or better, $p(C_k|\mathrm{x})$ that are between 0 and 1.

So, we apply a non-linear function to squash y:

$$p(C_k|\mathrm{x}) = y(\mathrm{x}) = f(w_0 + \mathrm{w}^T\mathrm{x})$$

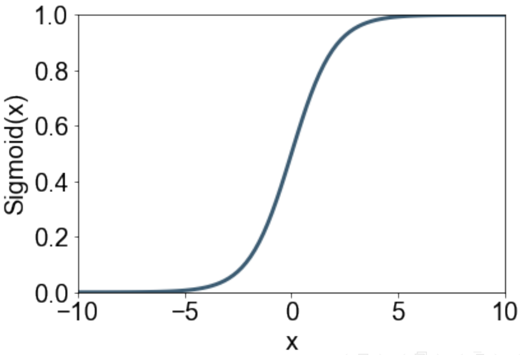
where f is:

- The sigmoid function for 2-class problems
- The softmax function for multi-class ones

Sigmoid function

The sigmoid function has a characteristic S-shape, given by:

$$f(x) = \frac{1}{1 + e^{-x}}$$



Logit Regression

- **Probability Constraints**: Ensures predictions remain between 0 and 1 using the logistic (sigmoid) function.
- **Nonlinear Relationship**: Better captures the underlying structure in binary outcomes, especially at the extremes (near 0 or 1).
- **More Robust**: Less sensitive to extreme values compared to LPM.

Functional Form:

$$P(y=1|x) = \frac{1}{1+e^{-(w_0+\mathrm{w}^T\mathrm{x})}}$$

Softmax Regression – Ideal for Multiclass Problems

- **Multiclass Capability**: Softmax is specifically designed for cases with multiple classes, generalizing Logit to multiple categories.
- **Probability Distribution**: Produces a probability for each class that sums to 1, ensuring interpretability and valid probability outputs.

Functional Form:

$$P(y = j|\mathrm{x}) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \quad j = 1, \ldots, K$$

Example

Consider a 4-class classification problem given 300 training data points.

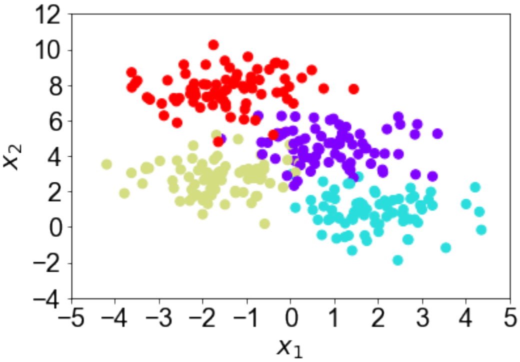
Generating sample data

```python
from sklearn.datasets import make_blobs as blobs

X, y = blobs(n_samples=300,
             centers=4,
             n_features=2,
             random_state=0,
             cluster_std=1.0)
```

Example

Consider a 4-class classification problem given 300 training data points.



Solving it in Python

Consider a 4-class classification problem given 300 training data points.

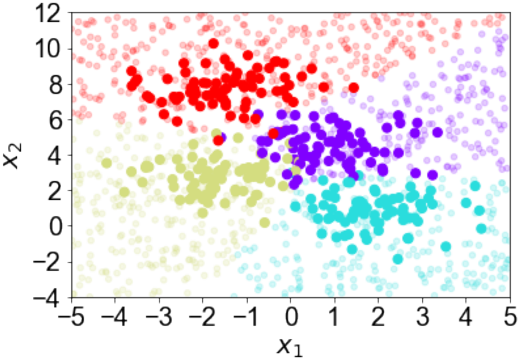
Logistic regression

```python
from sklearn.linear_model import LogisticRegression

# Create a logistic regression model
model = LogisticRegression()
# Fit the training data
model.fit(X, y);
```

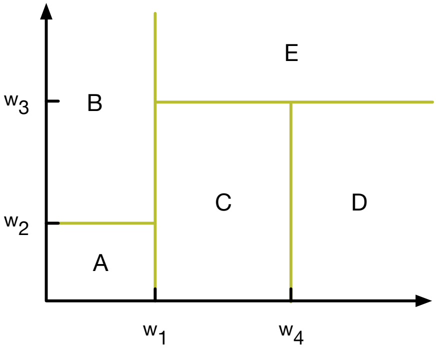
Solving it in Python

By calling `model.predict(samples)` on new data (`samples`), we can predict their label (in our case, colour).



Decision trees

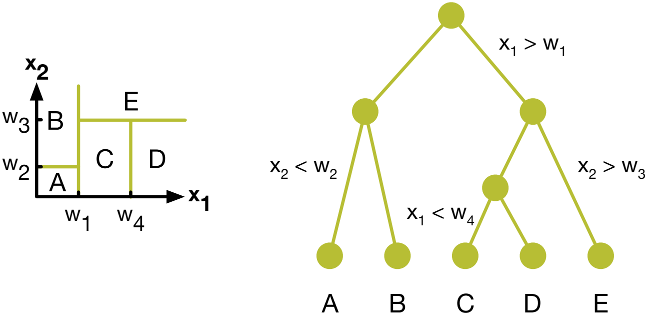
Decision trees partition the input space into cuboids (i.e., squares, cubes) and then assign a simple model to each region.[1]



[1]Example from C. Bishop's *Pattern recognition and machine learning*.

Inference using decision trees

A decision tree makes decisions by following a path through questions, dividing data at each step. Each question splits the data into smaller parts until reaching a final decision.



Training decision trees

How to determine the best structure of a decision tree?

- Even with fixed number of nodes, it can be computationally infeasible – too many combinations
- Greedy construction starts at a root node and adds nodes one at a time.

Then, the question is:

- When to stop adding nodes? In other words,
- How deep should the decision tree be?

Solving it in Python

Consider a 4-class classification problem given 300 data points.

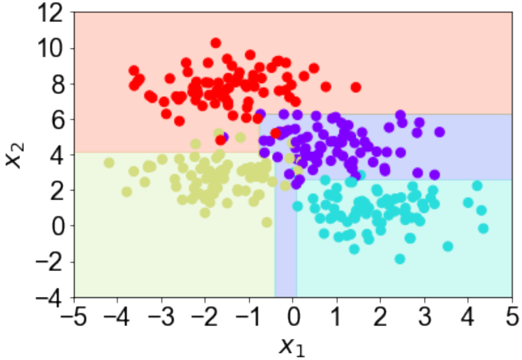
Logistic regression

```python
from sklearn.tree import DecisionTreeClassifier

# Create a decision tree of depth 4
classifier = DecisionTreeClassifier(max_depth=4)
# Fit the training data
classifier.fit(X, y);
```

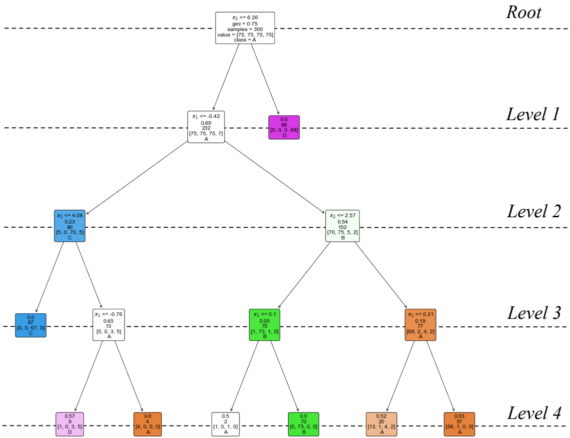
Solving it in Python

Let's visualise the regions found by the classifier:



Solving it in Python

Let's visualise the decision process:



Improving Decision Trees: Pruning and Bagging

Challenges with Decision Trees

- Decision trees are prone to overfitting, especially when fully grown.
- They can have high variance, making predictions sensitive to training data fluctuations.

Two Key Techniques to Improve Decision Trees

- **Pruning**: Simplifies the tree by removing unnecessary branches, reducing overfitting and improving generalization.
- **Bagging**: Creates an ensemble of trees by training multiple trees on different subsets of data and averaging predictions, reducing variance.

Pruning in Decision Trees

What is Pruning?

- Pruning reduces the size of a decision tree by removing sections that do not provide significant predictive power.
- This process is typically done after the tree has been fully grown.

Why Use Pruning?

- **Reduce Overfitting**: By cutting back branches that capture noise, pruning helps the model generalize better to unseen data.
- **Improve Interpretability**: A pruned tree is simpler and easier to interpret.

Bagging

Bagging is an ensemble technique capable of reducing the variance of an estimator (e.g., a decision tree):

- A *commitee* of classifiers, each fitting subsets of the data set
- Aggregating individual predictions (by voting or averaging) to make a final prediction.

Works well, but assumes individual models are uncorrelated

Solving it in Python

Consider a 4-class classification problem given 300 data points.

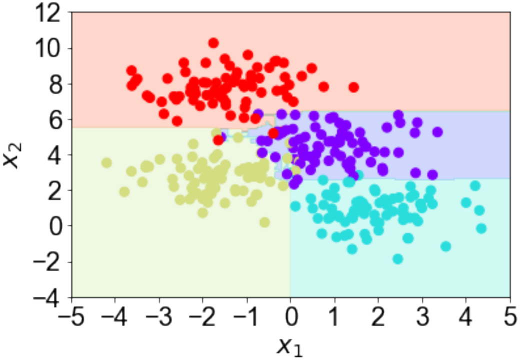
Logistic regression

```python
from sklearn.ensemble import BaggingClassifier

tree = DecisionTreeClassifier()
classifier = BaggingClassifier(tree,
                               n_estimators=100,
                               max_samples=0.8,
                               random_state=1)
# Fit the training data
baggingclassifier.fit(X, y);
```

Solving it in Python

Let's visualise the regions found by bagging:



Limitations of decision trees

Decision trees are **easy to interpret**. However,

- High sensitivity to input data
- Regions are squares,
  - Boundaries are aligned to axes
  - Can you think of a data set where trees are not appropriate?

Performance Metrics for Classification

To evaluate and compare the effectiveness of classification models, we use several performance metrics:

- **Accuracy**: Measures the overall proportion of correct predictions among total predictions.
- **Precision**: Indicates the proportion of true positive predictions out of all positive predictions made by the model. Useful when the cost of false positives is high.
- **Recall (Sensitivity)**: Measures the proportion of true positives out of all actual positives. Important when missing positive cases (false negatives) has a high cost.
- **F1 Score**: Combines precision and recall into a single metric by calculating their harmonic mean.

Performance Metrics for Classification

		true class: True	true class: False
predicted class	True	**True Positives (TP)**	**False Positives (FP)**
	False	**False Negatives (FN)**	**True Negatives (TN)**

$$\mathrm{PR} = \frac{\mathrm{TP}}{\mathrm{TP+FP}}$$

$$\mathrm{RE} = \frac{\mathrm{TP}}{\mathrm{TP+FN}}$$

$$\mathrm{CA} = \frac{\mathrm{TP+TN}}{\mathrm{TP+TN+FP+FN}}$$

$$\mathrm{F}_1 = \frac{2\mathrm{TP}}{2\mathrm{TP+FP+FN}}$$

Formulas for Performance Metrics: Recap

- **Accuracy**:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Predictions}}$$

- **Precision**:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

- **Recall (Sensitivity)**:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **F1 Score**:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Appendix: Naive Bayes

Appendix: Naive Bayes

We want to find the conditional probability distribution:

$$p(C_k|\text{features})$$

One approach is to compute it using Bayes' theorem:

$$p(C_k|\text{features}) \propto p(\text{features}|C_k)p(C_k)$$

So, we need to compute:

- $p(C_k)$, otherwise known as the *prior* probability of C_k
- $p(\text{features}|C_k)$, the class-conditional probability of x

Why Naive Bayes?

The prior $p(C_k)$ can be estimated from the data set:

- How many instances of C_k exist in a dataset of size n.

But it is hard to estimate:

$$p(x_1, x_2, \dots, x_n | C_k)$$

Unless we simplify the problem by assuming independence:

$$p(x_1, x_2, \dots, x_n | C_k) = p(x_1 | C_k) p(x_2 | C_k) \cdots p(x_n | C_k)$$

Computing $p(\mathrm{x}|C_k)$

We make assumptions about input features x_i, based on whether values are *discrete* or *continuous*.

If continuous, we typically assume a Gaussian (or normal) distribution. For each class C_k, $k = 1, 2, \dots K$:

- Filter all x whose label is C_k
- For each x_i in x
  - Compute mean, μ_{x_i}, and standard deviation, σ_{x_i}
  - Then, $p(x_i|C_k) \sim N(\mu_{x_i}, \sigma_{x_i})$

Solving it in Python

Consider a 4-class classification problem given 300 training data points.

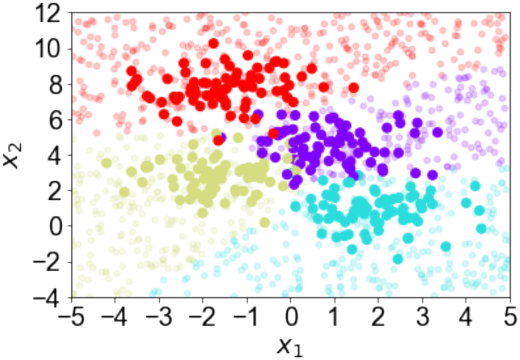
Gaussian Naive Bayes

```python
from sklearn.naive_bayes import GaussianNB

# Create a Gaussian Naive Bayes model
model = GaussianNB()
# Fit the training data
model.fit(X, y);
```

Solving it in Python

By calling `model.predict(samples)` on new data (`samples`), we can predict their label (in our case, colour).



Solving it in Python

Given $\mathrm{x} = (0.74, 6.45)$, we computed:

- $p(C_1|\mathrm{x}) = 0.72$
- $p(C_2|\mathrm{x}) = p(C_3|\mathrm{x}) = 0.00$
- $p(C_4|\mathrm{x}) = 0.28$