# CS221 Section 2: Learning

Ajay Sohmshetty & Benoit Zhou

# Topics

- Backpropagation
- SciKit Learn
  - *k*-nearest neighbors
  - Decision trees
  - Random forests
  - Hyperparameter search
  - Cross validation

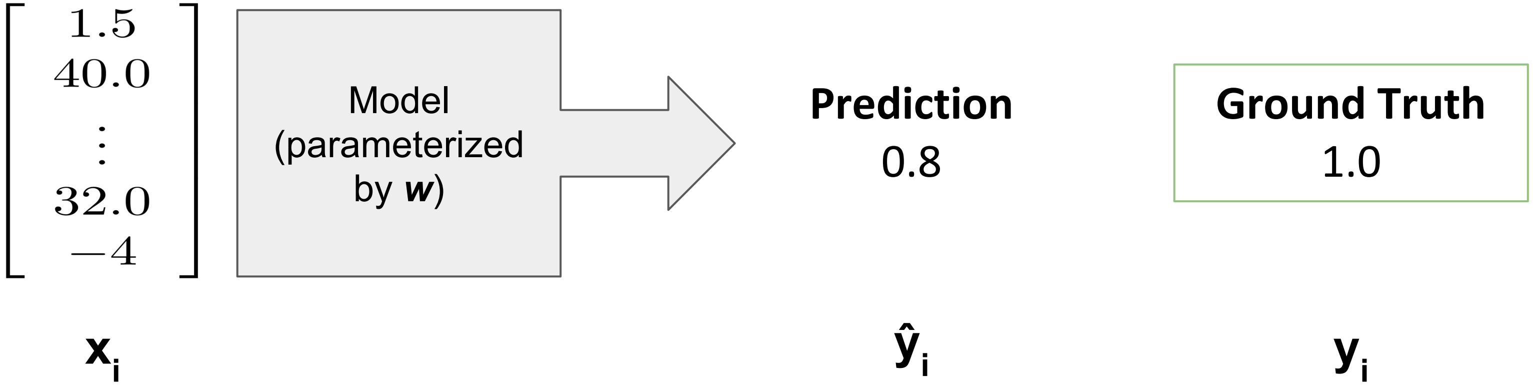
# Datapoints

$$\mathbf{x}_i = \begin{bmatrix} 1.5 \\ 40.0 \\ \vdots \\ 32.0 \\ -4 \end{bmatrix}$$

**Ground Truth**
1.0

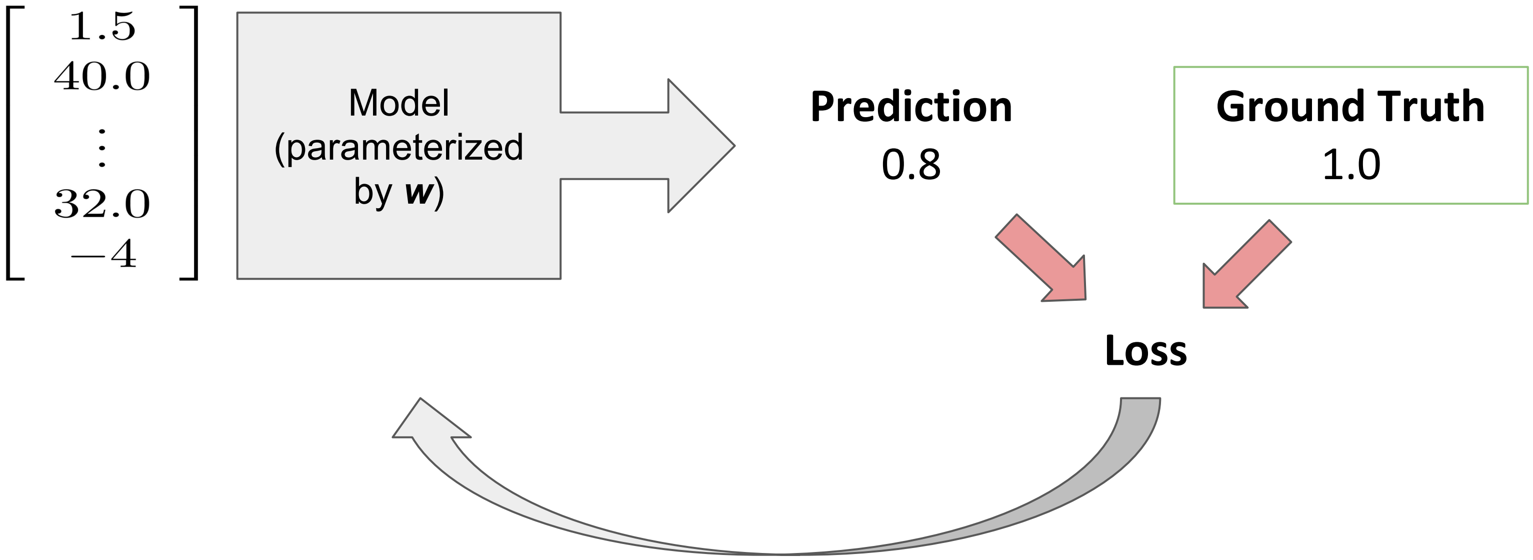
$\mathbf{y}_i$

# Model



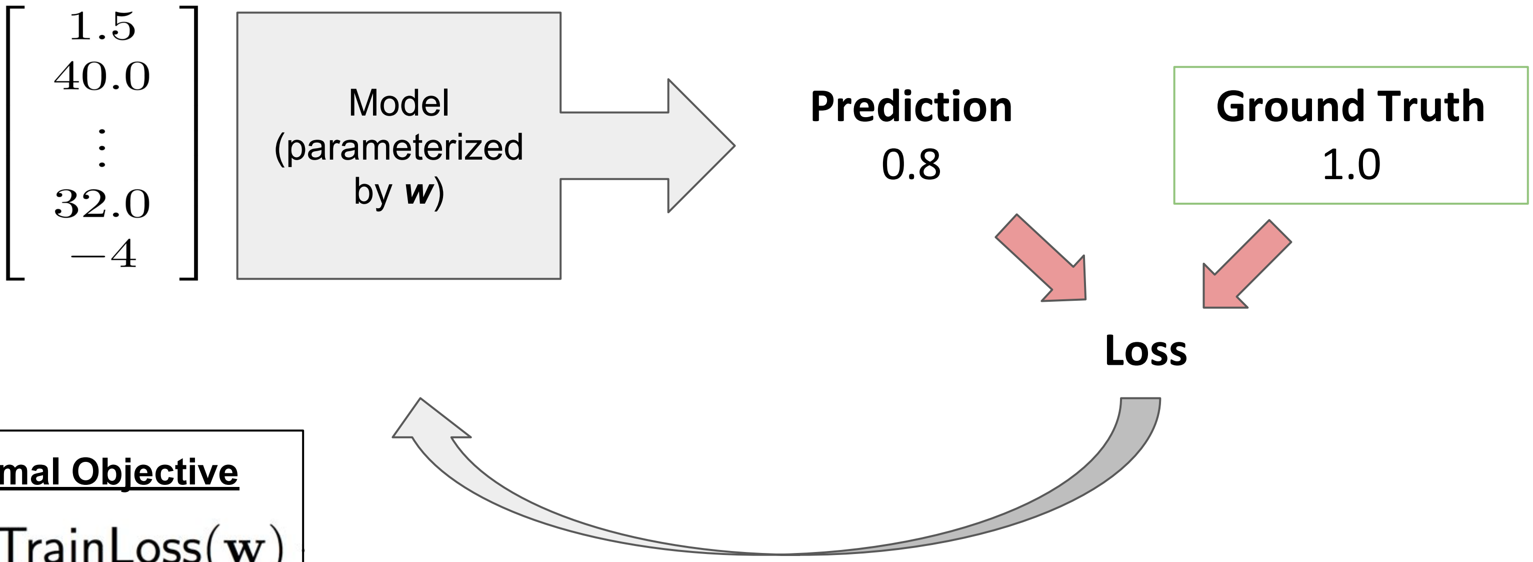
$\begin{bmatrix} 1.5 \\ 40.0 \\ \vdots \\ 32.0 \\ -4 \end{bmatrix}$
Model (parameterized by ***w***)
**Prediction**
0.8
**Ground Truth**
1.0
$\mathbf{x_i}$
$\mathbf{\hat{y}_i}$
$\mathbf{y_i}$

# Loss




**Key idea**: Use loss to inform updates to weights.

# Loss



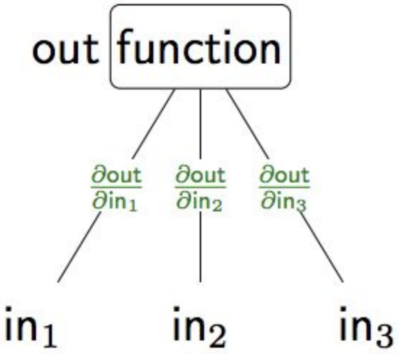

**<u>Formal Objective</u>**

$$\min_{\mathbf{w}} \mathsf{TrainLoss}(\mathbf{w})$$

**Key idea**: Use loss to inform updates to weights.

# Partial Derivatives / Gradients

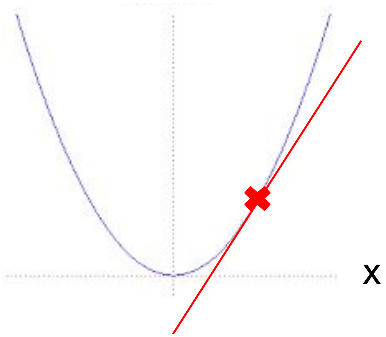
We want to know how each weight affects the training loss.
→ exactly what derivative (gradient in the vector case) tells us!




Partial derivatives (gradients): how much does the output change if an input changes?

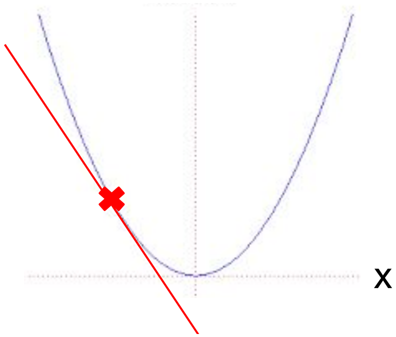
# Gradient Descent

We want to know how each weight affects the training loss.
→ exactly what derivative (gradient in the vector case) tells us!




Positive gradient = decrease x

**$y=x^2$**




Negative gradient = increase x

# Stochastic Gradient Descent

**Algorithm: stochastic gradient descent**

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

    For $(x, y) \in \mathcal{D}_{\text{train}}$:

        $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$

# Symbolic Gradient Computation - Two Layer NN

$$\text{TrainLoss}(\mathbf{V}, \mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w})$$

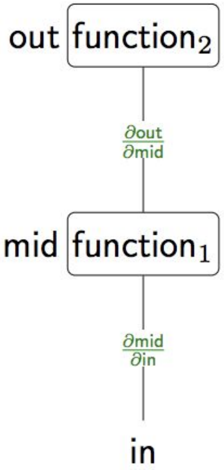
$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = (y - f_{\mathbf{V},\mathbf{w}}(x))^2$$

$$f_{\mathbf{V},\mathbf{w}}(x) = \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x))$$

**GOAL**

$$\nabla_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

# Backpropagation




Chain rule: $\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$

# Backprop Gradient Computation - Two Layer NN

$$\text{TrainLoss}(\mathbf{V}, \mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w})$$

$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = (y - f_{\mathbf{V},\mathbf{w}}(x))^2$$

$$f_{\mathbf{V},\mathbf{w}}(x) = \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x))$$

**<u>GOAL</u>**

$$\nabla_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

# Why Backpropagate?

- Don’t have to deal with the nastiness of the chain rule with deep neural networks
- Performance optimizations (can hold onto intermediary values, don’t have to recompute)
- Translates into a modular framework, so packages like Tensorflow and PyTorch will auto-differentiate for you!

# Backpropagation

Backprop: http://cs231n.github.io/optimization-2/

Vector, Matrix, and Tensor Derivatives: http://cs231n.stanford.edu/vecDerivs.pdf

# scikit-learn

```
import sklearn
import numpy as np
```

## Load a dataset

$X$ is a $n \times d$ matrix where each row is $\phi(x_i) \in \mathbb{R}^d$

$y$ is a $n$-dimensional vector where each entry $y_i$ is the label of the $i^{th}$ datapoint $x_i$

```
from sklearn.datasets import load_iris
X = load_iris().data
y = load_iris().target
```

```
X[0]
```

```
array([5.1, 3.5, 1.4, 0.2])
```

```
y[0]
```

```
0
```

## Split into training and test sets

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=3)
```

# Logistic Regression

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr = lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)
```

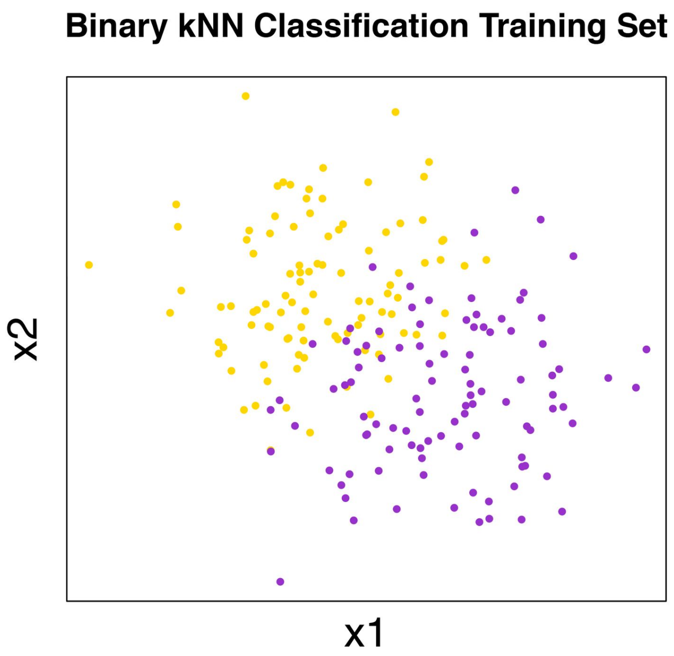
```
y_pred
```

```
array([0, 0, 0, 0, 0, 2, 1, 0, 2, 1, 1, 0, 1, 1, 2, 0, 2, 2, 2, 0, 2, 2,
       2, 1, 0, 2, 2, 1, 1, 1, 0, 0, 2, 1, 0, 0, 2, 0, 2, 1, 2, 1, 0, 0,
       2, 1, 0, 2, 2, 2])
```

```
np.mean(y_pred == y_test)
```

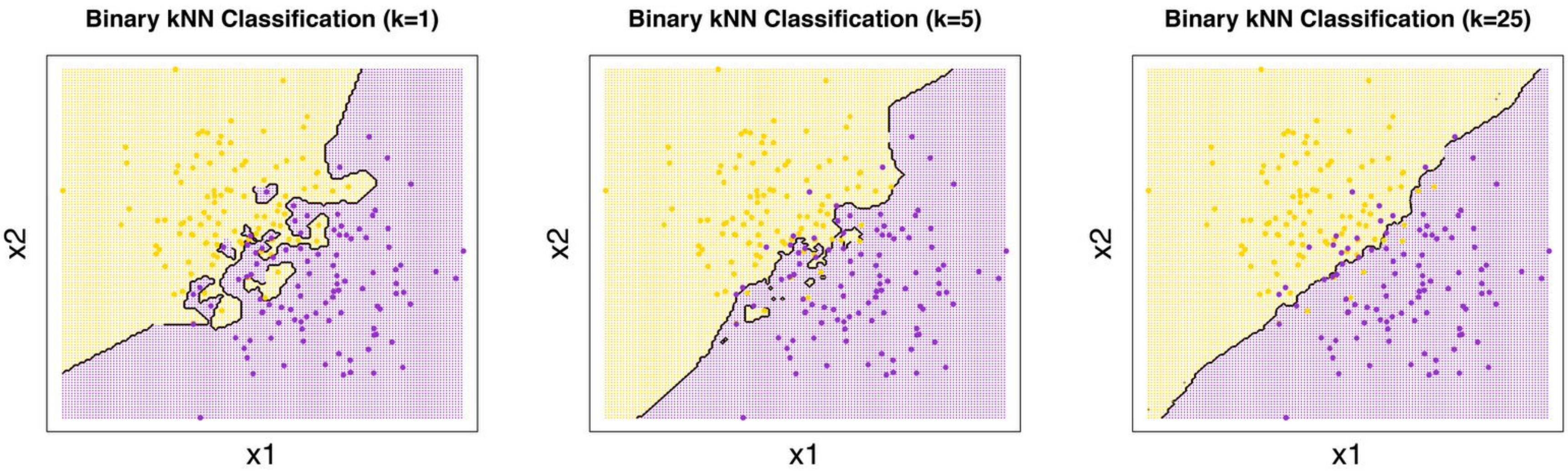
```
0.92
```

# *k*-nearest neighbors



# *k*-nearest neighbors

Effect of *k*:



# *k*-nearest neighbors

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn = knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
```

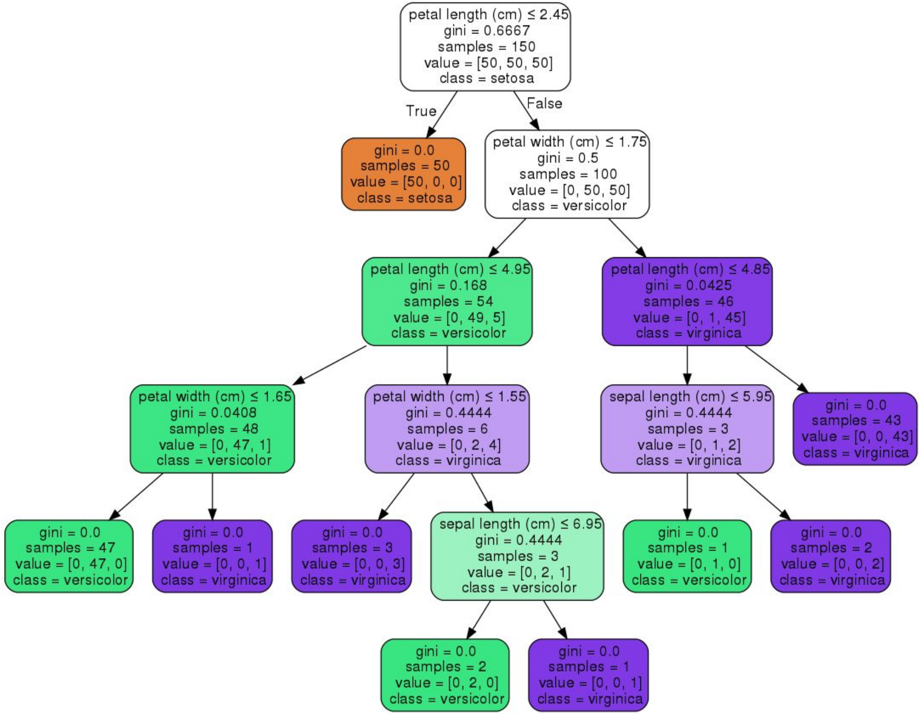
```
y_pred
```

```
array([0, 0, 0, 0, 0, 2, 1, 0, 2, 1, 1, 0, 1, 1, 2, 0, 2, 2, 2, 0, 2, 2,
       2, 1, 0, 2, 2, 1, 1, 1, 0, 0, 2, 1, 0, 0, 2, 0, 2, 1, 2, 1, 0, 0,
       2, 1, 0, 2, 2, 1])
```

```
np.mean(y_pred == y_test)
```

```
0.94
```

# Decision Tree



# Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier()
dt = dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
```

```
y_pred
```

```
array([0, 0, 1, 1, 0, 2, 1, 1, 0, 1, 2, 1, 1, 0, 0, 2, 1, 0, 0, 2, 1, 2,
       0, 1, 0, 1, 2, 2, 1, 2, 1, 1, 0, 0, 2, 2, 1, 2, 0, 1, 2, 0, 0, 0,
       2, 0, 1, 2, 0, 1, 1, 2, 0, 0, 2, 0, 0, 1, 2, 0])
```

```
np.mean(y_pred == y_test)
```

```
0.9166666666666666
```

# Random Forest

Decision trees have low bias but high variance

Random forests reduce the variance with bagging (bootstrap aggregating)

```
for b = 1, …, B:         # B is a parameter that you choose

    sample with replacement n training examples from (X, y); call these
(X_b, y_b)

    train a decision tree on (X_b, y_b), with a subset of the features

Take the average (for regression) or the majority vote (for
classification) of the B decision trees
```

# Random Forest

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
rf = rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
```

```
y_pred
```

```
array([0, 0, 1, 1, 0, 2, 1, 1, 0, 1, 2, 1, 1, 0, 0, 2, 1, 0, 0, 2, 1, 2,
       0, 1, 0, 1, 1, 2, 1, 2, 1, 1, 0, 0, 2, 1, 1, 2, 0, 1, 2, 0, 0, 0,
       2, 0, 1, 2, 0, 1, 1, 2, 0, 0, 1, 0, 0, 1, 2, 0])
```

```
np.mean(y_pred == y_test)
```

```
0.9666666666666667
```

# Gradient Boosting

Random forests try to decrease the high variance of decision trees while keeping the low bias

Gradient boosting trees try to decrease a high bias while keeping a low variance

Start with a weak learner, i.e. a really simple tree

Each iteration, in order to minimize the loss, add a tree without modifying the existing trees

# Gradient Boosting

```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier()
gb = gb.fit(X_train, y_train)
y_pred = gb.predict(X_test)
```

```
y_pred
```

```
array([0, 0, 1, 1, 0, 2, 1, 1, 0, 1, 2, 1, 1, 0, 0, 2, 1, 0, 0, 2, 1, 2,
       0, 1, 0, 1, 2, 2, 1, 2, 1, 1, 0, 0, 2, 2, 1, 2, 0, 1, 2, 0, 0, 0,
       2, 0, 1, 2, 0, 1, 1, 2, 0, 0, 2, 0, 0, 1, 2, 0])
```

```
np.mean(y_pred == y_test)
```

```
0.9833333333333333
```

# Hyperparameters

**Number of trees**: usually, the more, the better

**Max depth of trees**: a deeper tree is more likely to overfit

**Max number of features**: sqrt($p$) or log($p$) as a rule of thumb

# Hyperparameter search

**Grid search**: exhaustive, suffers from curse of dimensionality, parallelizable

**Randomized search** (recommended): faster, allows including prior knowledge by specifying distribution from which to sample, parallelizable, good performance

**Bayesian optimization**: builds a probabilistic model to find the best hyperparameters

# Validation

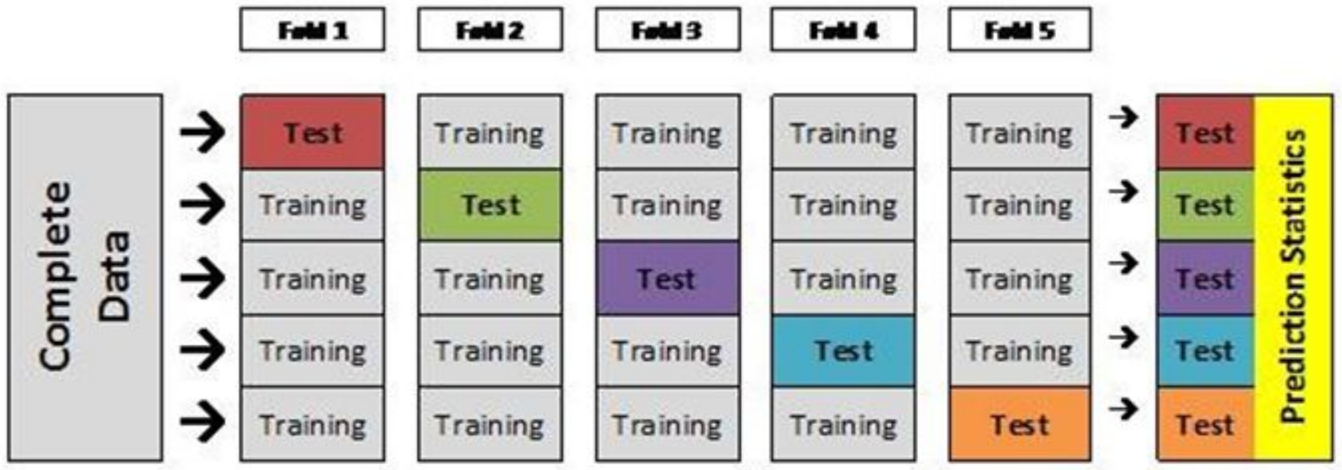
Solution: randomly take out 10-50% of training data and use it instead of the test set to estimate test error.

$\mathcal{D}_{\text{train}} \backslash \mathcal{D}_{\text{val}}$ | $\mathcal{D}_{\text{val}}$ | $\mathcal{D}_{\text{test}}$

**Definition: validation set**

A **validation (development) set** is taken out of the training data which acts as a surrogate for the **test set**.

# *k*-fold cross-validation



# *k*-fold cross-validation

```
from sklearn.model_selection import cross_val_score
lr = LogisticRegression()
scores = cross_val_score(lr, X, y, cv=10)
```

```
scores
```

```
array([1.        , 1.        , 1.        , 0.93333333, 0.93333333,
       0.93333333, 0.8       , 0.93333333, 1.        , 1.        ])
```

```
np.mean(scores)
```

```
0.9533333333333334
```