Module 1 :

Machine Learning Review

Supervised Learning Algorithms

Géraldine Schaller-Conti, August 2024

Discussion Session

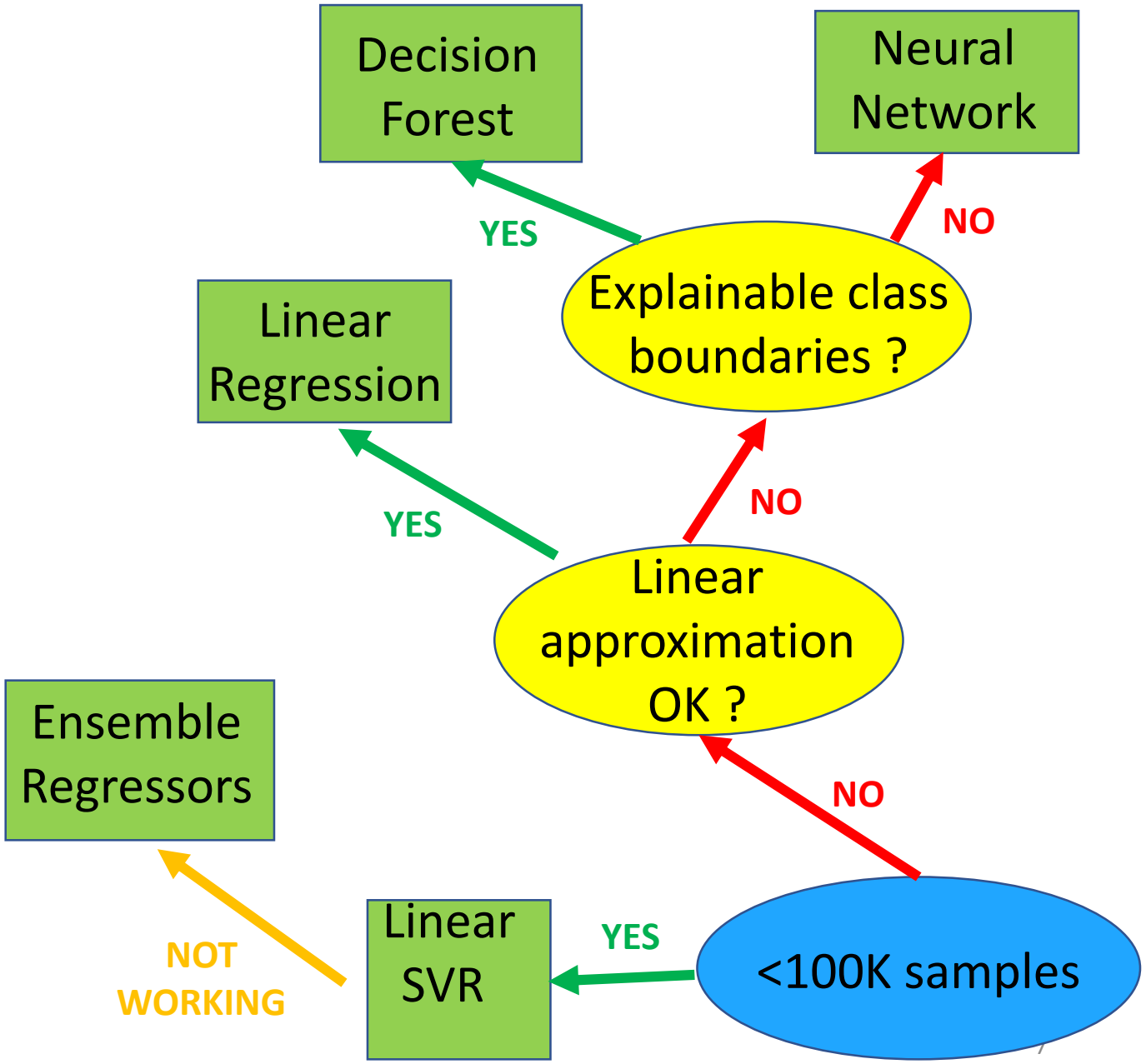
- Review of Notebook 1 (data preparation)
- Based on 02_end_to_end_machine_learning_project.ipynb (A. Geron)
  - Visualize data
  - Correlation matrix
  - Prepare data
  - Encoding
- Exercise (summary of algorithm jungle)

Bibliography

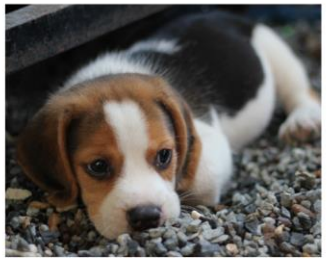
- Deep Learning book (Goodfellow, Bengio, Courville)
- Machine Learning @ Stanford (Prof Andrew Ng)
- Hands-On Machine Learning with Scikit-Learn & Tensorflow (Aurélien Géron)

Regression


$82000
$55500
???

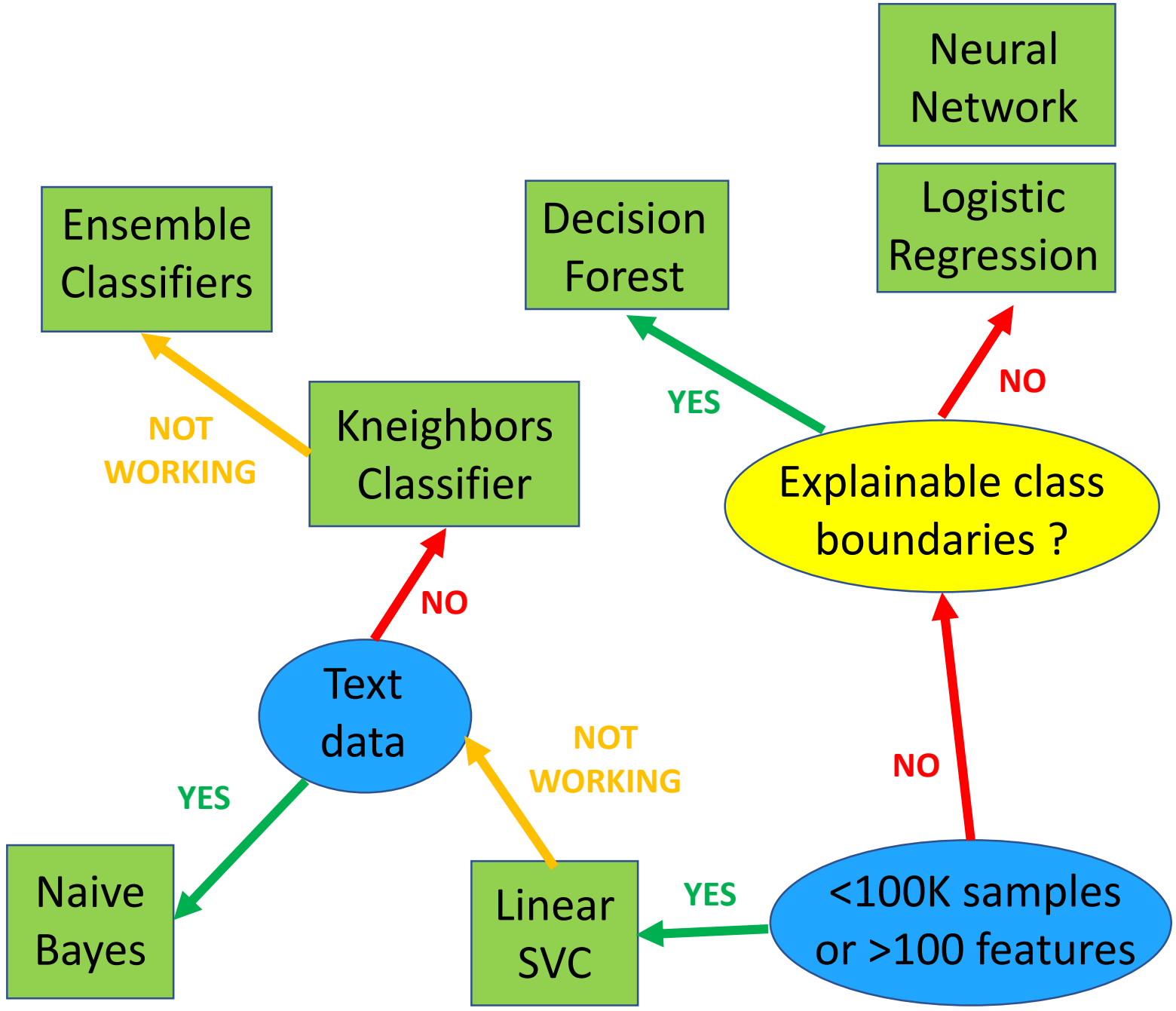

Decision Forest
Neural Network
YES
NO
Explainable class boundaries ?
Linear Regression
YES
NO
Linear approximation OK ?
Ensemble Regressors
NO
NOT WORKING
Linear SVR
YES
<100K samples

Classification



Dog: 96%
Cat: 29%
Duck: 2%
Bird: 0%

Dog: 36%
Cat: 94%
Duck: 2%
Bird: 1%



Learning Objectives

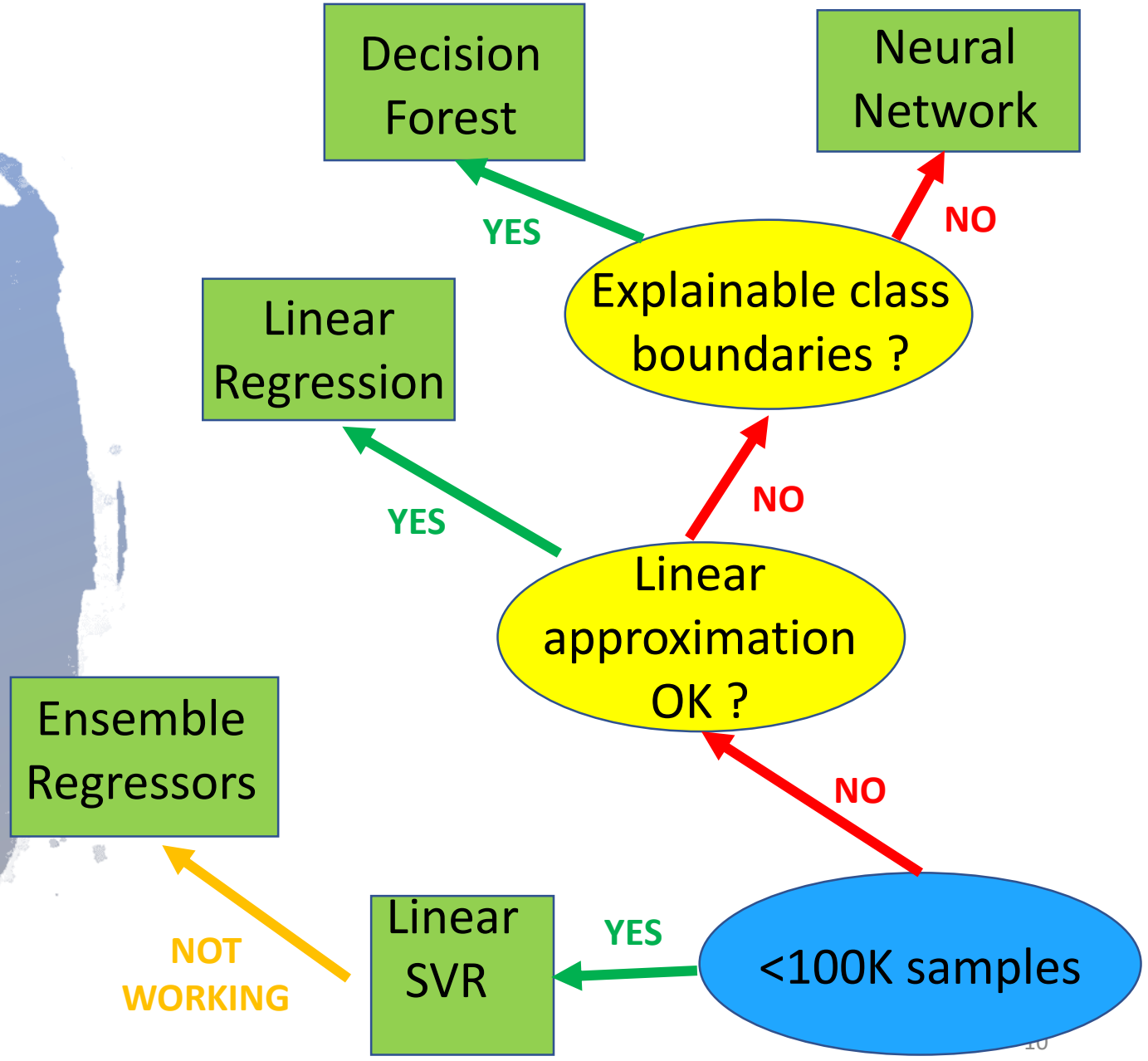
- Regression
  - Types : Linear, polynomial
  - Regularization : Ridge, LASSO, Elastic Net
  - Performance evaluation
- Classification
  - Logistic regression
  - K-nearest neighbors
  - Naïve Bayes
  - Performance evaluation
- Support Vector Machines (regression/classification)
  - SVC, SVR
- Ensemble methods (regression/classification)
  - Decision trees, random forests
  - Bagging, boosting

Regression

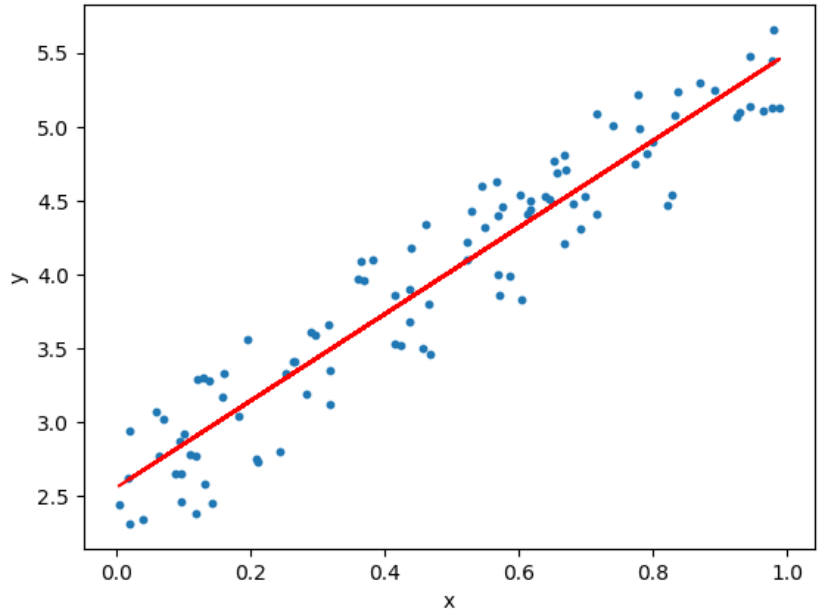
$82000

$55500

???



Linear Regression



- Regression equation (simplest form) : $y = b \cdot x + c$
  - Complexity = number of coefficients used in the model

- Cost function

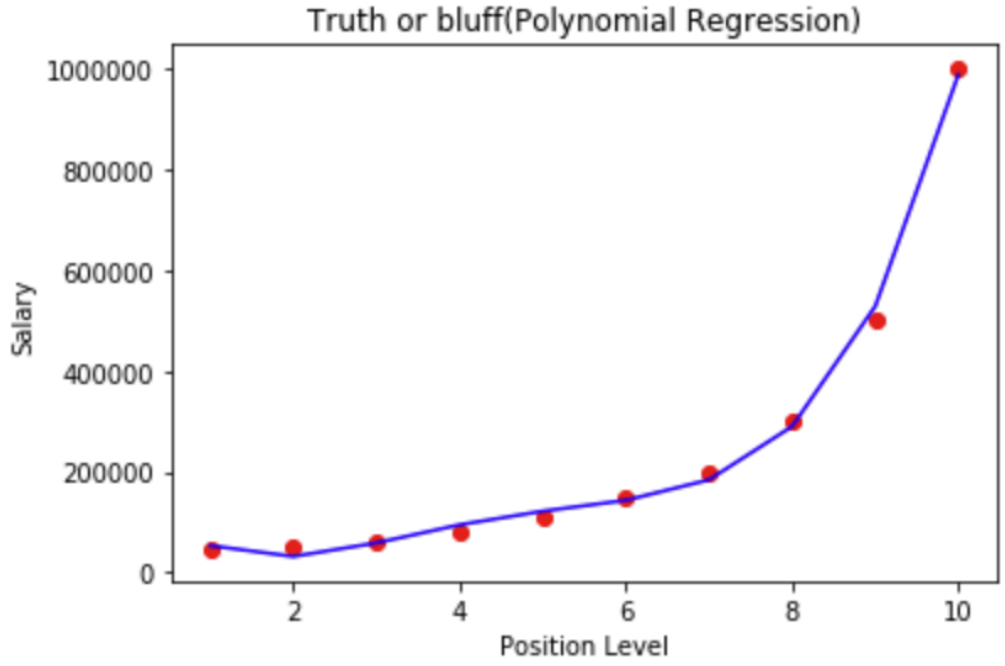
$$MSE(X, \hat{y}) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T \cdot x^{(i)} - y^{(i)}\right)^2$$

$$\hat{y} = \theta^T \cdot x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Linear Regression in practice

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
>>> # y = 1 * x_0 + 2 * x_1 + 3
>>> y = np.dot(X, np.array([1, 2])) + 3
>>> reg = LinearRegression().fit(X, y)
>>> reg.score(X, y)
1.0
>>> reg.coef_
array([1., 2.])
>>> reg.intercept_
3.0000...
>>> reg.predict(np.array([[3, 5]]))
array([16.])
```

Polynomial Regression



$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n + \varepsilon.$$

```
#fitting the polynomial regression model to the dataset
from sklearn.preprocessing import PolynomialFeatures
poly_reg=PolynomialFeatures(degree=4)
X_poly=poly_reg.fit_transform(X)
poly_reg.fit(X_poly,y)
lin_reg2=LinearRegression()
lin_reg2.fit(X_poly,y)
```

- You can use a linear model to fit nonlinear data
- Add powers of each feature as new features
- Use LinearRegression on this training data

Regularization

- To limit the number of features to solve
- Loss :

Regularized Loss = Loss Function + Constraint

- Constraint :
  - Ridge : $J(\theta) = 0.5 \cdot \alpha \sum_{i=1}^{n} \theta_i^2$
  - LASSO : $J(\theta) = \alpha \sum_{i=1}^{n} |\theta_i|$
  - Elastic Net : $J(\theta) = (1-r)/2 \cdot \alpha \sum_{i=1}^{n} \theta_i^2 + r \cdot \alpha \sum_{i=1}^{n} |\theta_i|$

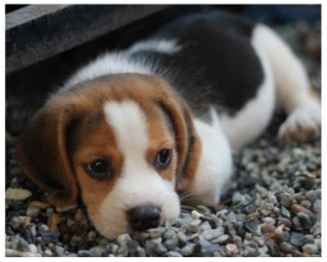
Performance Evaluation

- To evaluate an algorithm, we need a way to measure **how well it performs on the task**
- It is measured **on a separate set** (test set) from what we use to build the function f (training set)
- **Examples** :
  - Classification accuracy (portion of correct answers)
  - Error rate (portion of incorrect answers)
  - Regression accuracy (e.g. least squares errors)

Make your own summary table

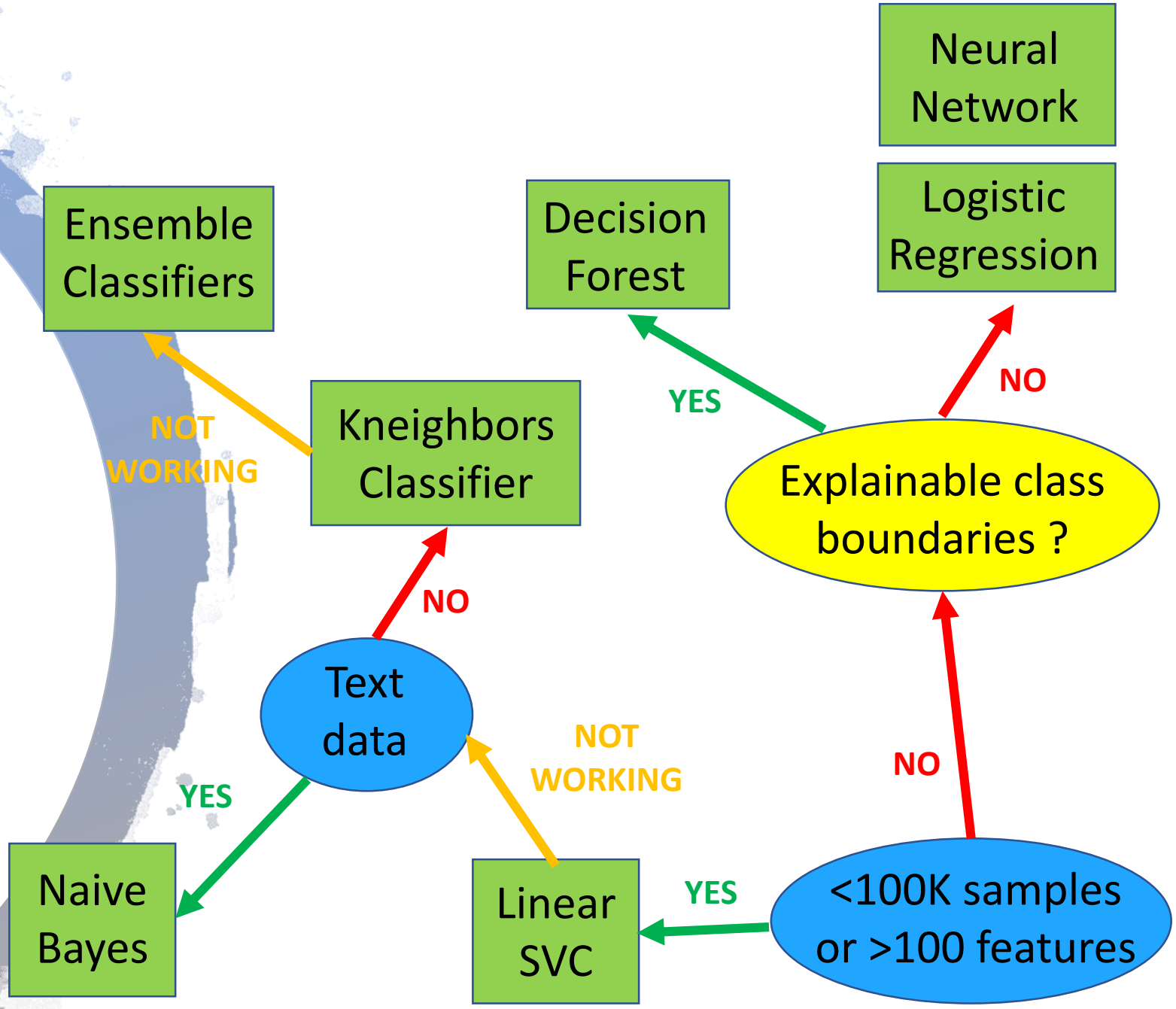
https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics

Classification



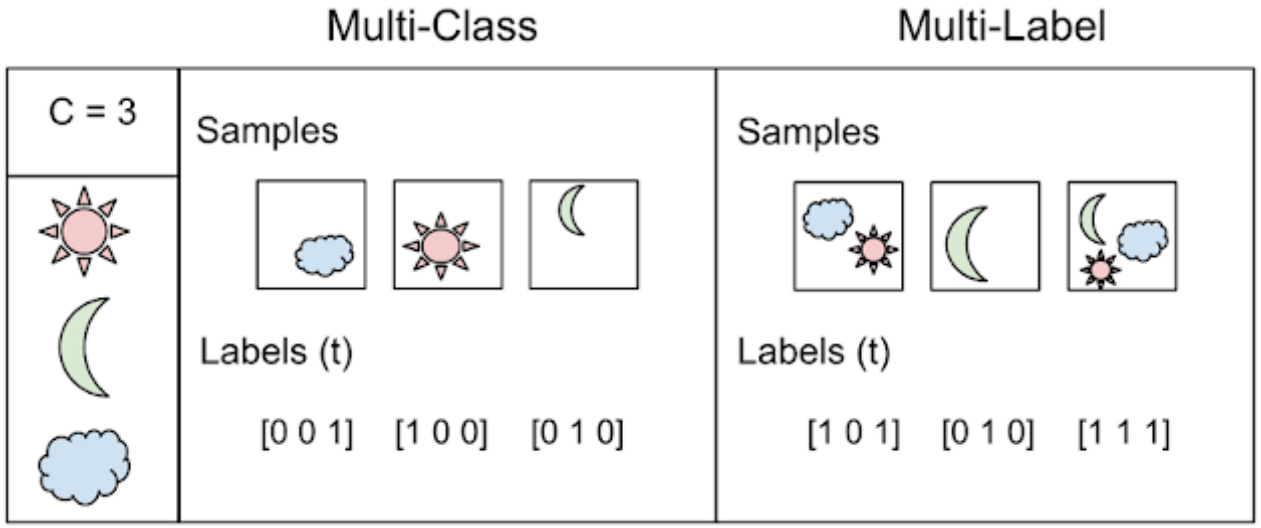
Dog: 96%
Cat: 29%
Duck: 2%
Bird: 0%

Dog: 36%
Cat: 94%
Duck: 2%
Bird: 1%



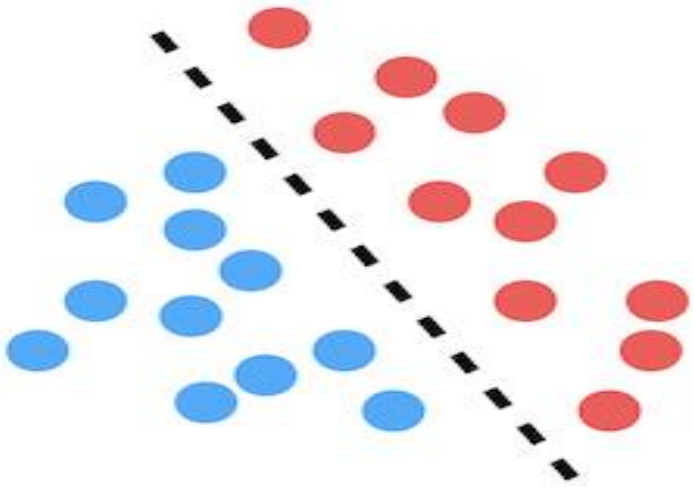
Types of Classifications

- Binary classification
  - Distinction between two classes
- Multiclass classification
  - Distinction between more than two classes
- Multilabel classification
  - Possible to have several classes selected



Types of Models

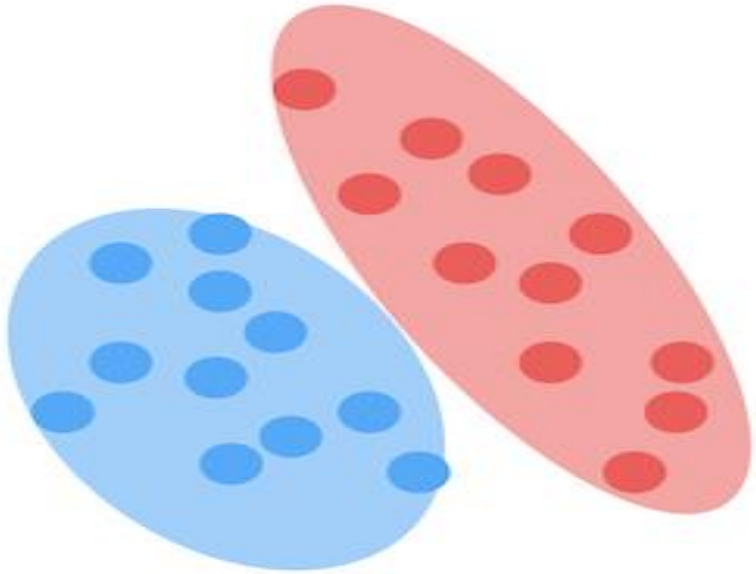
Discriminative



- Focus on the decision boundary
- Only supervised tasks

1. Logistic Regression
2. K-nearest neighbors

Generative



- Probabilistic “model” of each class
- Decision boundary is where one model becomes more likely
- Can use unlabeled data

3. Naïve Bayes

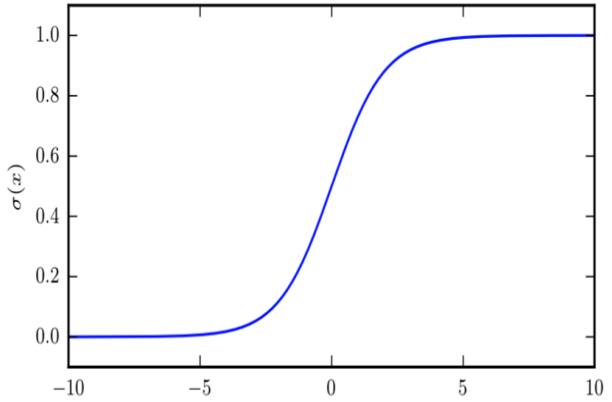
1. Logistic Regression

- Mainly used in cases where the output is True/False
- Data fit into linear regression model, which then be acted upon by a logistic function predicting the categorical target

$$\hat{p} = \sigma(\theta^T \cdot x)$$

  - Logistic function = sigmoid (*two-class*) :

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



- Cost function (convex) : cross-entropy

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

Generalization to multi-class

Softmax (*multi-class* classifier) :

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

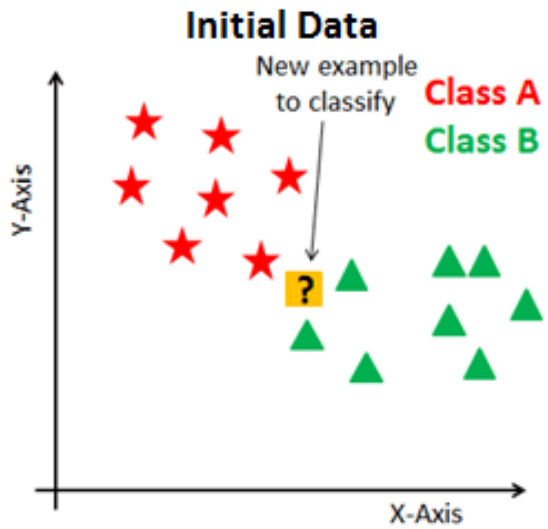
Cross-entropy cost function :

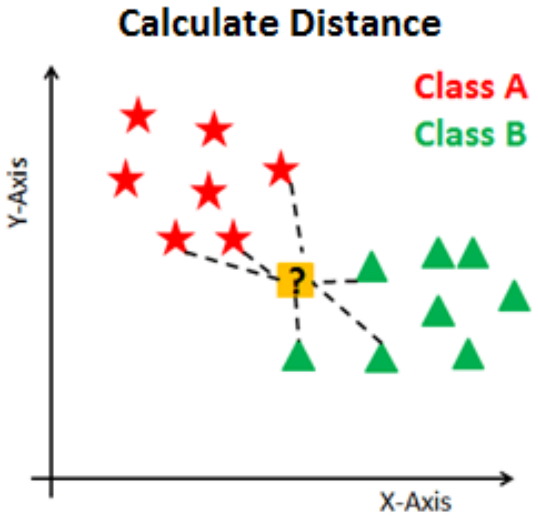
$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} {y_k}^{(i)} \log\left({\hat{p}_k}^{(i)}\right)$$

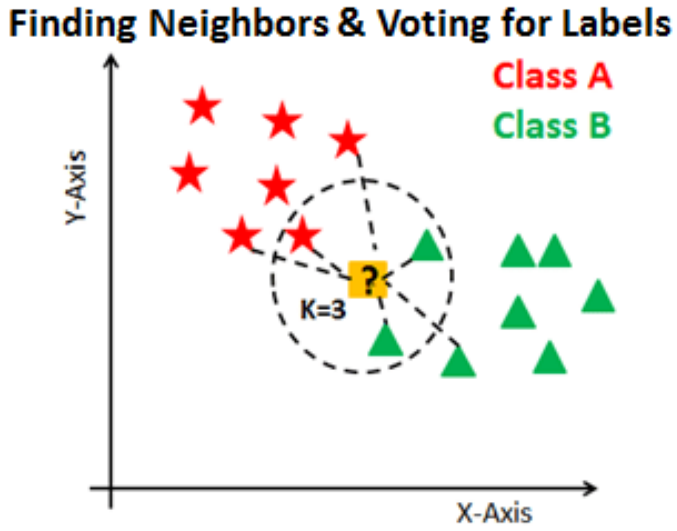
Logistic Regression in practice

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(random_state=0).fit(X, y)
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :])
array([[9.8...e-01, 1.8...e-02, 1.4...e-08],
       [9.7...e-01, 2.8...e-02, ...e-08]])
>>> clf.score(X, y)
0.97...
```

2. K-Nearest-Neighbors (k-NN)







- Steps : Take a data point
  1) Calculate distances
  2) Find k-closest neighbors
  3) Vote for label : the data point is assigned the label of the majority of the k closest points

Optimal k value for k-NN

- Cross-validation : evaluate different values of k and choose the one performing best on the validation set
- Odd values : for binary classification problems, better to avoid ties in the majority class
- Bias-variance trade-off : smaller values of k tend to result in a more complex model with low bias and high variance
- Ensemble methods : weighted k-NN, where different neighbors are given different weights based on their distance from the query point.
- Experimentation : ultimately, the choice of k might require some experimentation.

K-NN in practice

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.66666667 0.33333333]]
```

Can also be used for multilabel classification

Logistic Regression versus K-NN

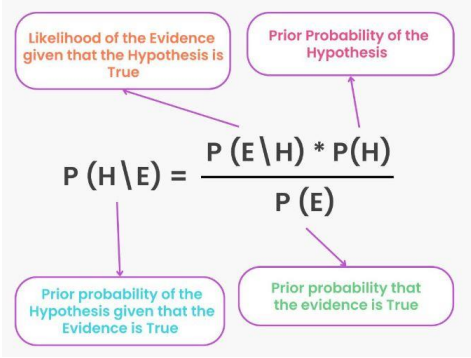
- LR is a parametric approach because it assumes a linear functional form for f(X), whereas K-NN is a non-parametric method

	Parametric (i.e. LR)	**Non-parametric (i.e. K-NN)**
Advantages	Easy to fit (small number of coefficients) Easy to interpret	Do not assume an explicit form for f(X)
Disadvantages	Strong assumptions about the form of f(X)	More complex to interpret
Use cases	Better if there is a small number of observations per predictor	

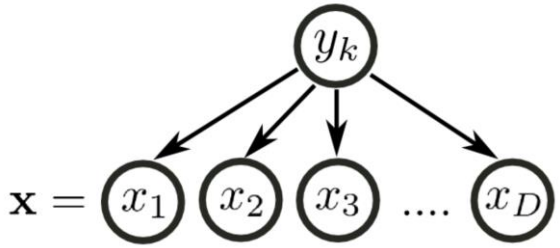
3. Naive Bayes

- Linear classifier based on probabilities. It learns the probability of every object, its features and which groups they belong to
  - Example : predict a fruit based on color and texture

- use Bayes theorem (Bayes)



- Use independence condition among features (Naïve)
  - "naive" because of the independence of the features

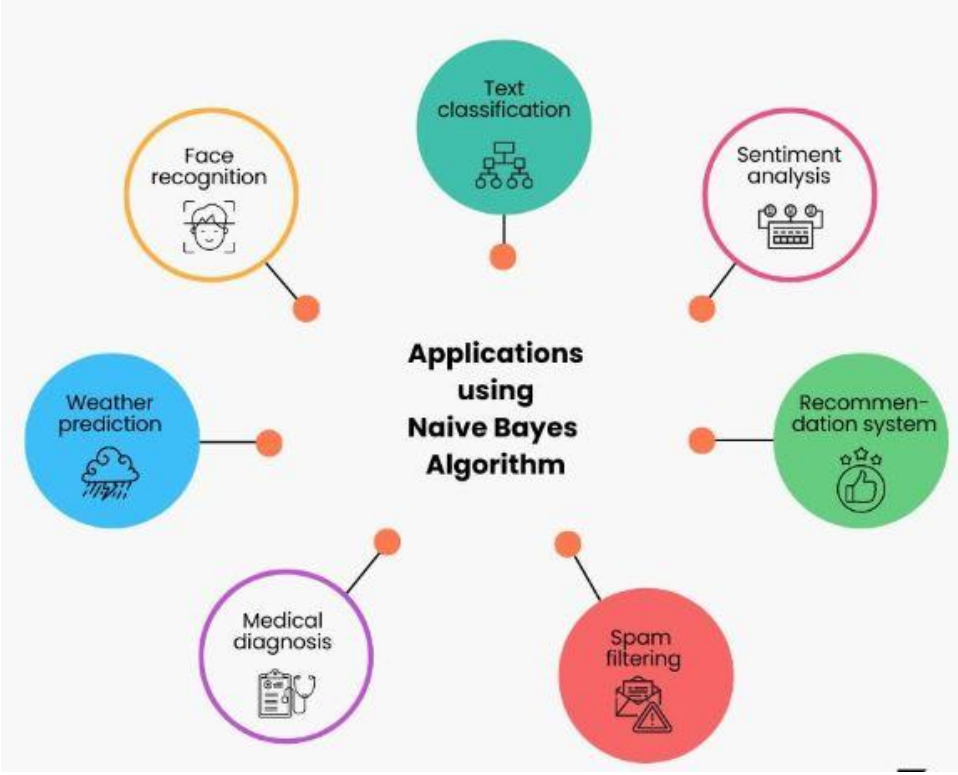


$$p(y_k|\mathbf{x}) = \frac{p(\mathbf{x}|y_k)p(y_k)}{p(\mathbf{x})} \propto p(\mathbf{x}|y_k)p(y_k)$$

$$p(y_k|\mathbf{x}) = p(y_k)\prod_{i=1}^{D} p(x_i|y_k)$$

Gaussian density function

Naive Bayes in practice




```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.naive_bayes import GaussianNB
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=0)
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(X_train, y_train).predict(X_test)
>>> print("Number of mislabeled points out of a total %d points : %d"
...       % (X_test.shape[0], (y_test != y_pred).sum()))
Number of mislabeled points out of a total 75 points : 4
```

Logistic Regression versus Naive Bayes

	Logistic Regression	**Naïve Bayes**
Advantages	Lower bias	Asymptotic solution faster (O(log n)) Lower variance
Disadvantages	Slower (O(n)) higher variance	Higher bias (because of assumption on features)

Performance Evaluation : Case Study

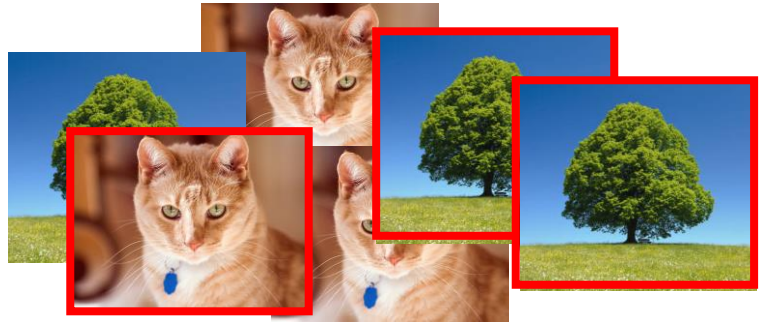
- You want to find cats in images

- Classification error (portion of wrong answers) used as an evaluation metric

Algorithm	Classification error (%)
A	**3%**
B	**5%**

Which one is best ?

Evaluation Metrics



- Precision (p)

$$\text{Precision (\%)} = \frac{True\ positive}{Number\ of\ predicted\ positive} x\ 100 = \frac{True\ positive}{(True\ positive + False\ positive)} x\ 100$$

$$\frac{2}{2+1} \times 100 = 66\%$$

- Recall (r)

$$\text{Recall (\%)} = \frac{True\ positive}{Number\ of\ predicted\ actually\ positive} x\ 100 = \frac{True\ positive}{(True\ positive + False\ negative)} x\ 100$$

$$\frac{2}{2+2} \times 100 = 50\%$$

F1-score

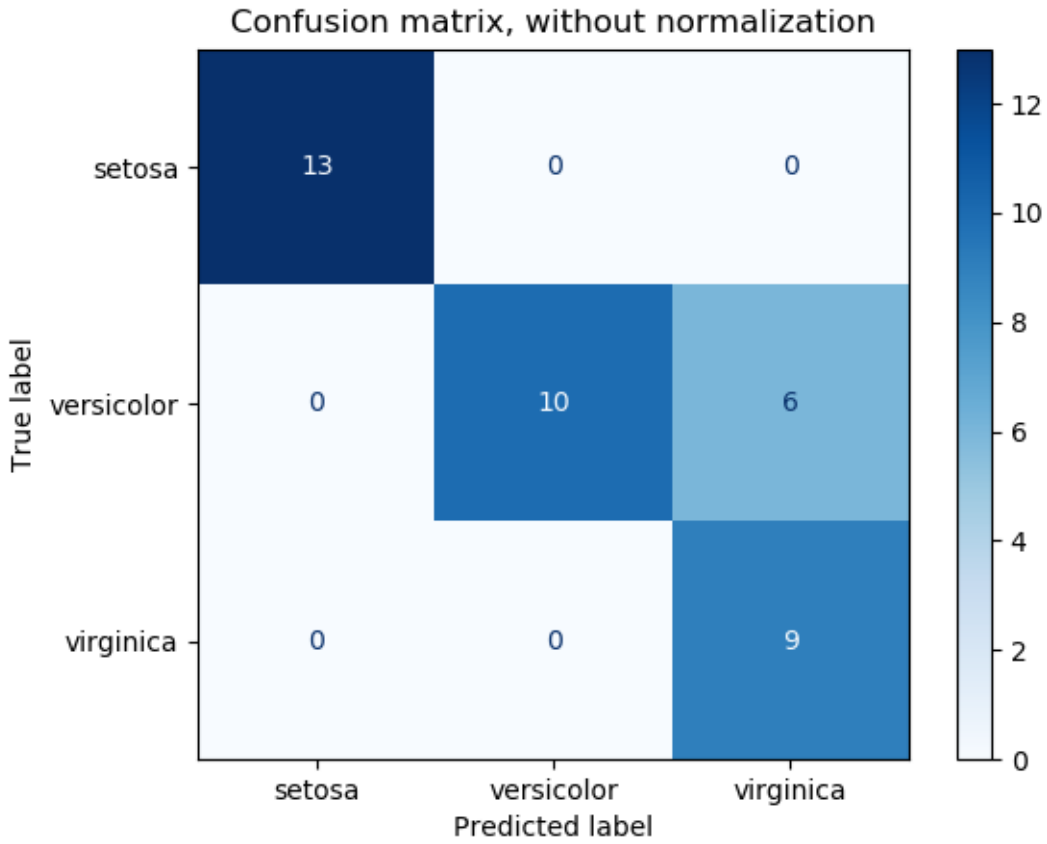
- F1-score is a harmonic mean combining p and r

$$\text{F1-Score} = \frac{2}{\frac{1}{p}+\frac{1}{r}}$$

	Precision	Recall	F1 Score	
Algo 1 →	0.5	0.4	0.444	✓
Algo 2 →	0.7	0.1	0.175	
Algo 3 →	0.02	1.0	0.0392	

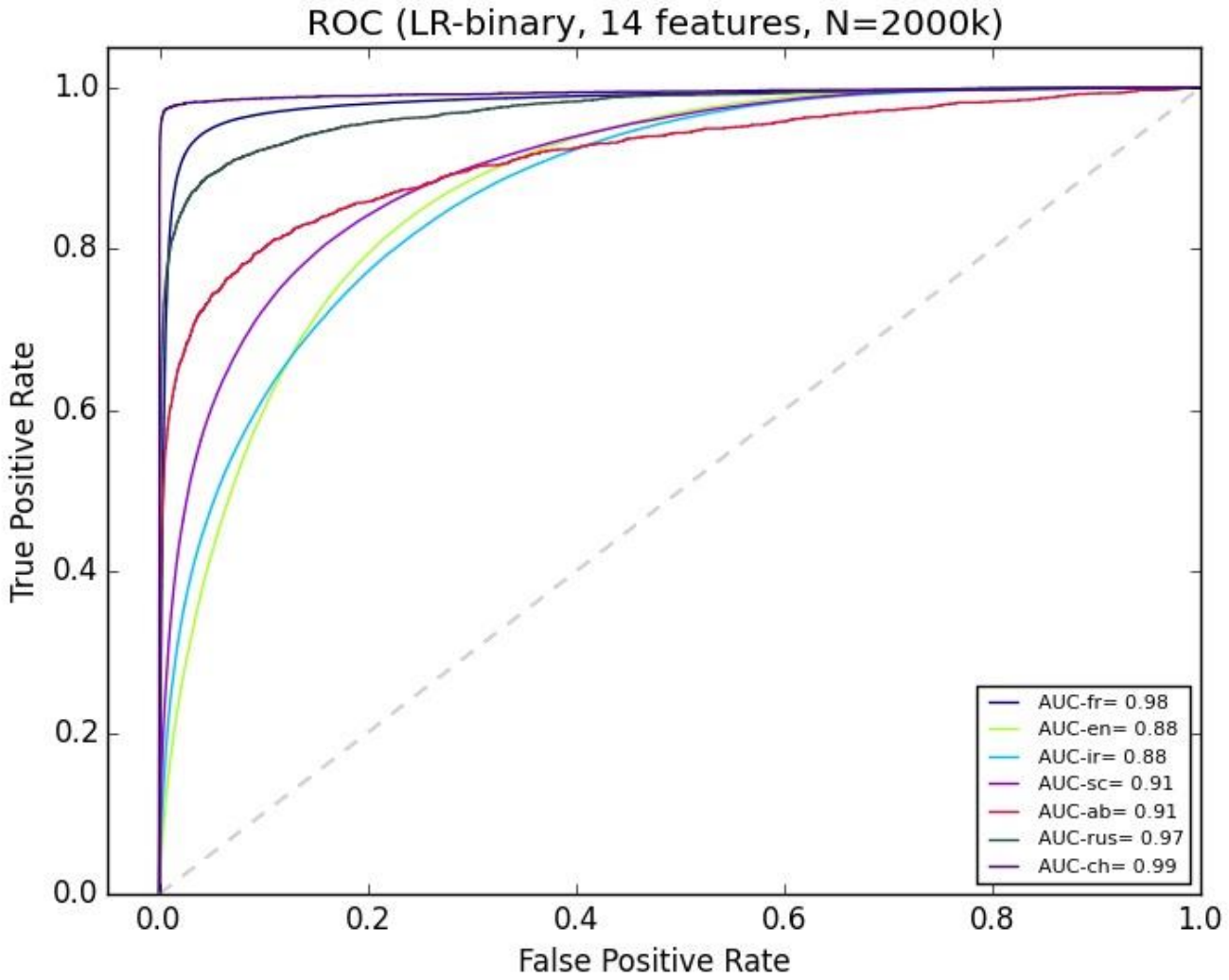
Confusion Matrix

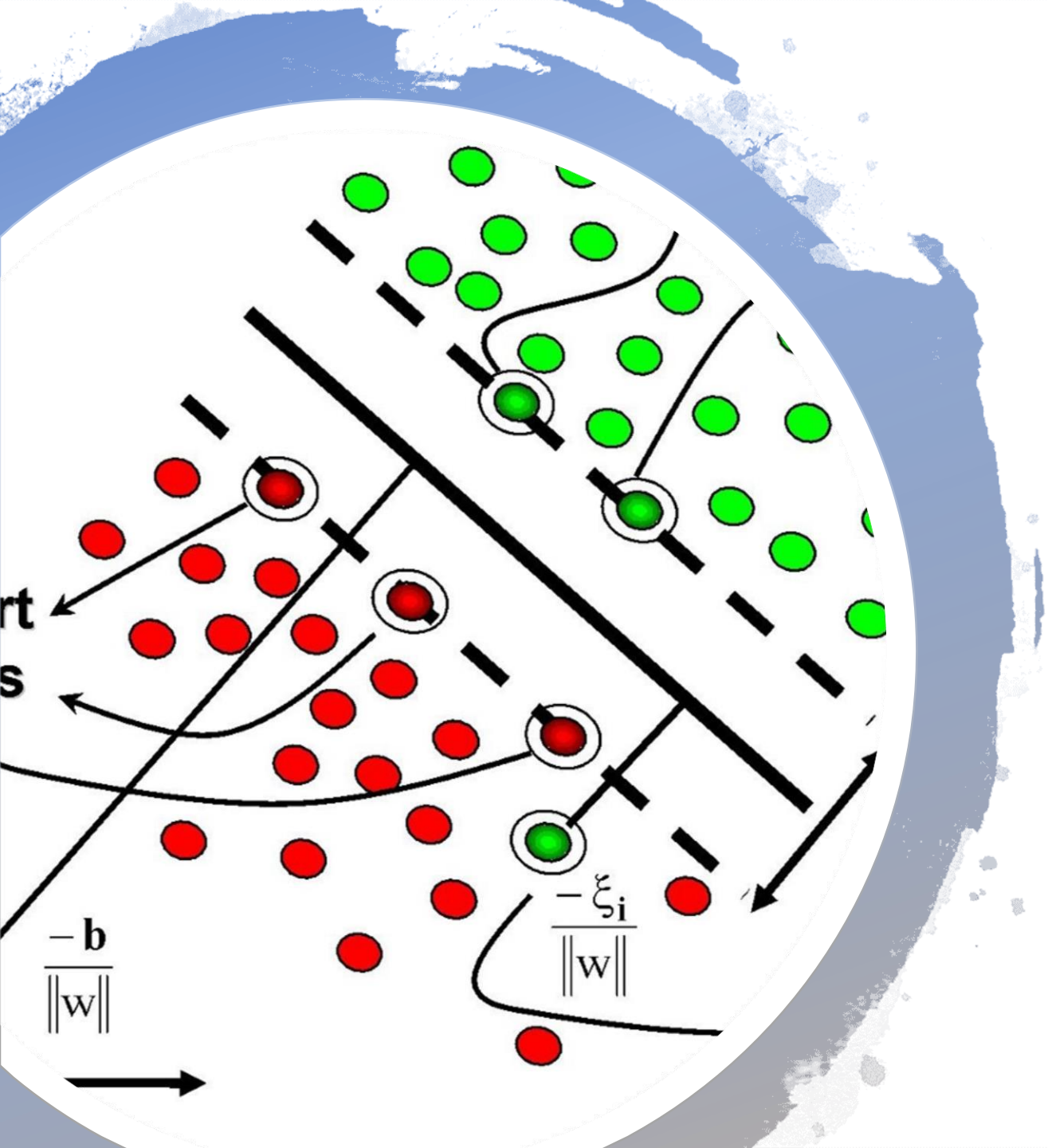
- To evaluate the performance of a classifier
- Count the number of times instances of class A are classified as class B



ROC Curve

- Tool used with binary classifiers for accuracy



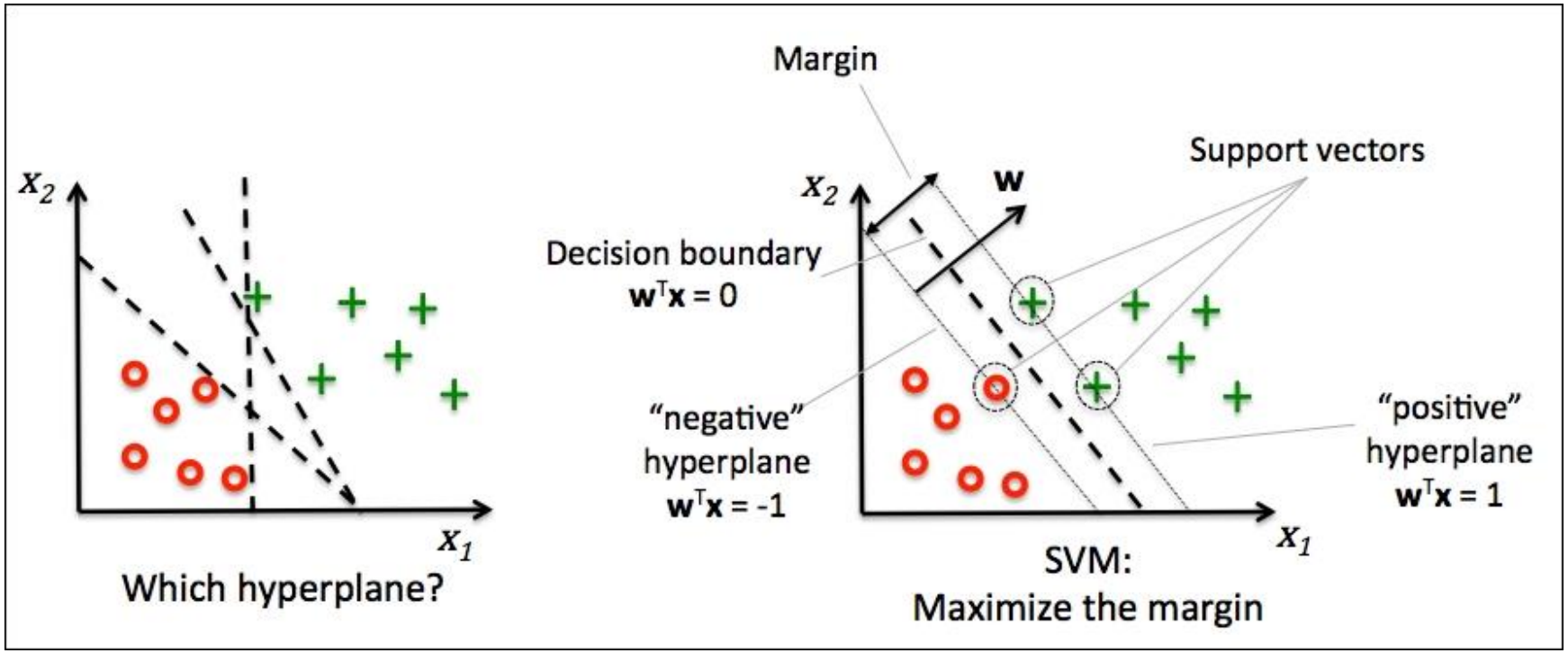


Support Vector Machines (SVMs)

first developed for classification (SVC), then generalized to handle regression (SVR)

SVC

- "Fit the widest possible street between classes"



- binary classifier (important to scale the input features)
- Convex optimization → *unique minimum*
- Use cases: object identification, text recognition, bioinformatics, speech recognition,…

Kernel

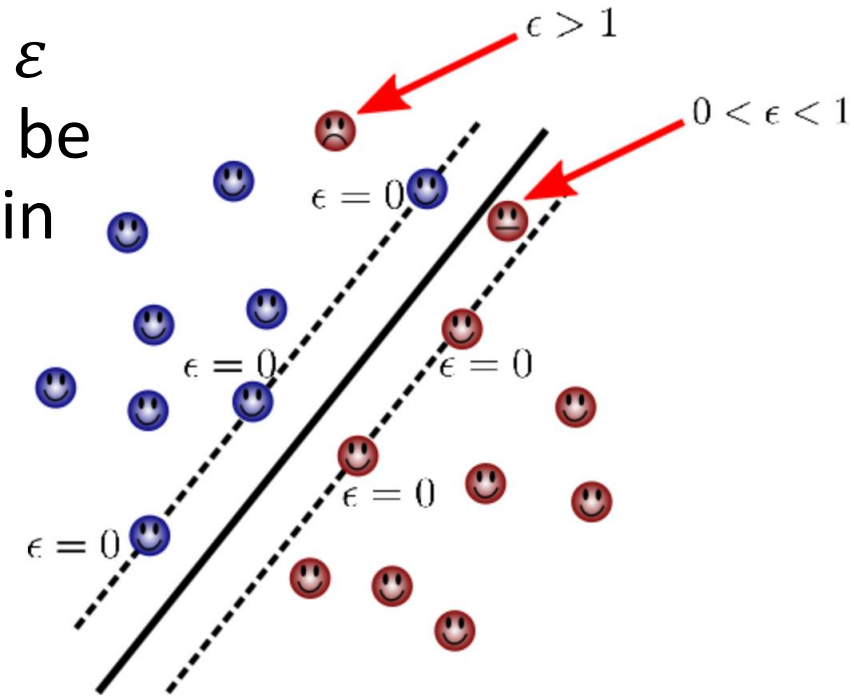
- Both linear and non-linear cases covered depending on the *choice* of the kernel (parameter)

- Linear $K(a, b) = aT \cdot b$
- Polynomial $K(a, b) = \gamma(a^T \cdot b + r)$
- Gaussian RBF $K(a, b) = \exp(-\gamma \|a - b\|^2)$

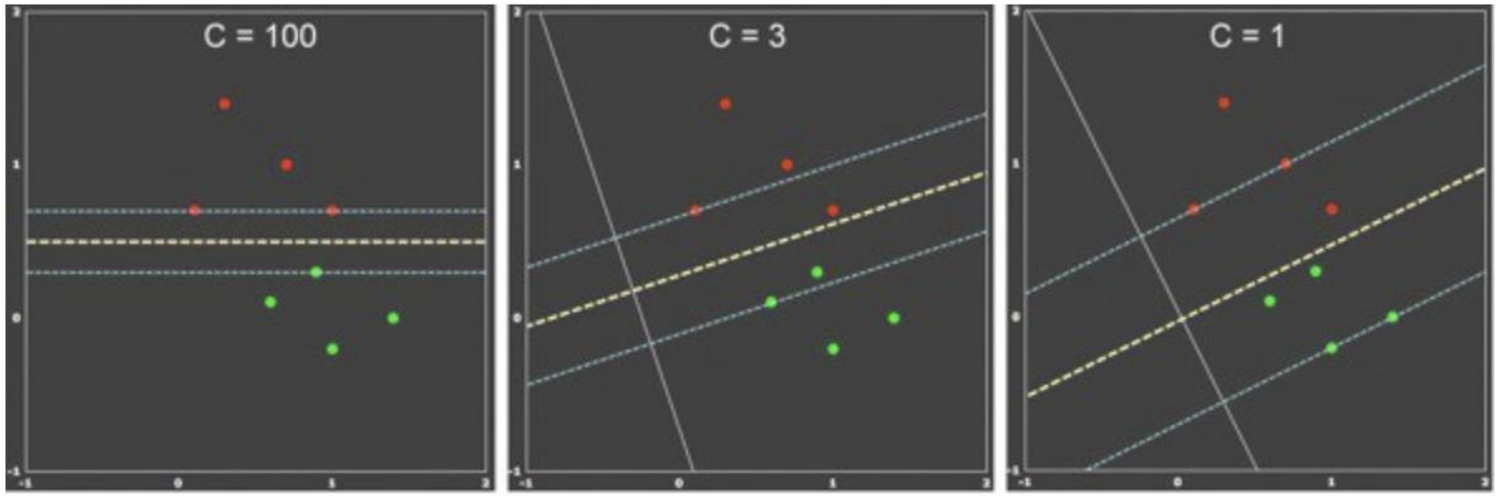
- Disadvantages :
  - Cost of training high with large datasets → SVC suited for complex but small- or medium-sized datasets
  - Generic kernels struggle to generalize well

Soft Margin SVM

- Define a tolerance variable ε to allow for some points to be a bit farther from the margin boundary (epsilon>1)



- New variable C
  - Large C → strict margin
  - Small C → looser margin



SVC in practice

- sklearn.svm.SVC : fit time scales at least quadratically with the number of samples and may be impractical beyond 10000 samples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import SVC
>>> clf = SVC(gamma='auto')
>>> clf.fit(X, y)
SVC(gamma='auto')
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Kernel function = 'linear', 'poly', 'rbf', 'sigmoid', … only with the SVC class, not the LinearSVC !

C=1 (default value)

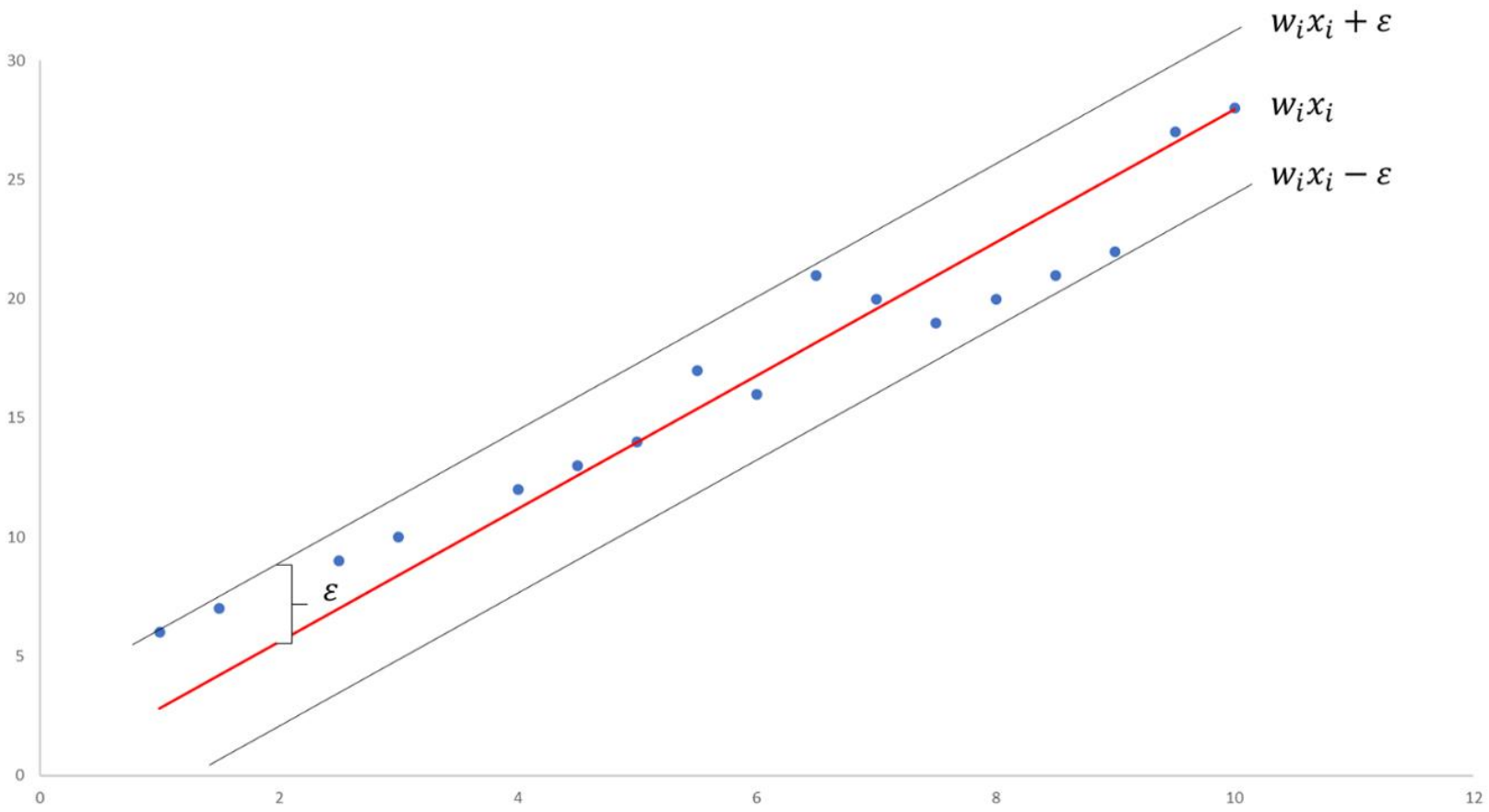
- sklearn.svm.LinearSVC : can be used with larger datasets (up to 100000 samples)

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = LinearSVC(random_state=0, tol=1e-5)
>>> clf.fit(X, y)
LinearSVC(random_state=0, tol=1e-05)
>>> print(clf.coef_)
[[0.085... 0.394... 0.498... 0.375...]]
>>> print(clf.intercept_)
[0.284...]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

LinearSVC much faster than SVC(kernel='linear'), as based on the liblinear library

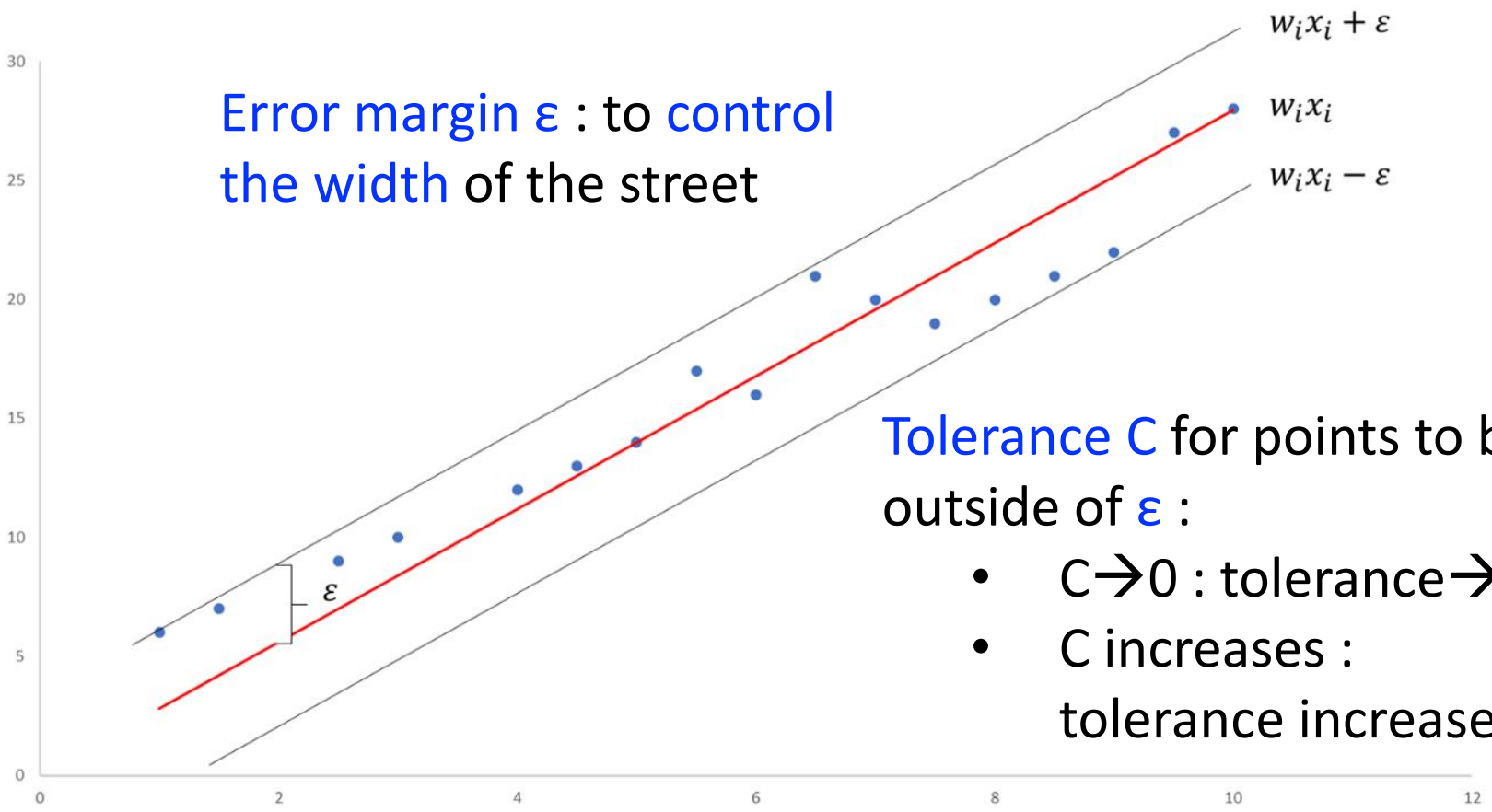
SVR

- Trick is to reverse the objective: try to fit as many instances as possible *on* the street while limiting margin violations



Hyperparameters

- Trick is to reverse the objective: try to fit as many instances as possible *on* the street while limiting margin violations



Error margin ε : to control the width of the street

Tolerance C for points to be outside of ε :

- C→0 : tolerance→0
- C increases : tolerance increases

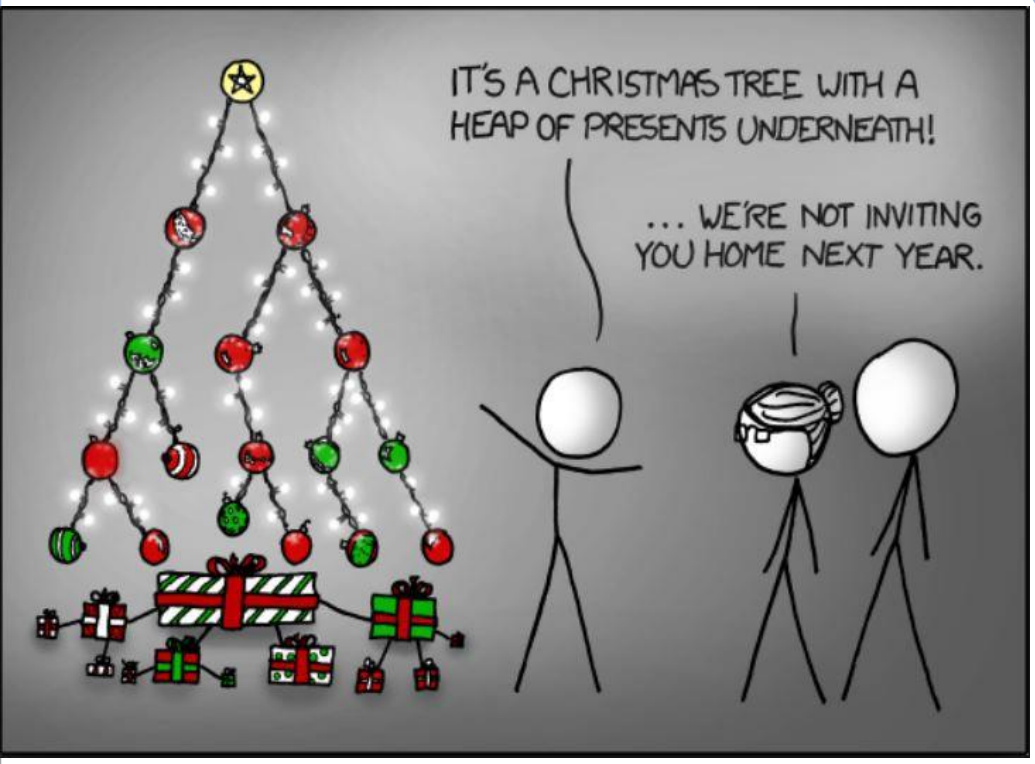
SVR in practice

- sklearn.svm.SRV :
  - free parameters : C and epsilon
  - fit time : scales at least quadratically with the number of samples and may be impractical beyond 10000 samples

```
>>> from sklearn.svm import SVR
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = SVR(C=1.0, epsilon=0.2)
>>> clf.fit(X, y)
SVR(epsilon=0.2)
```

- sklearn.svm.LinearSVR :
  - Similar to SVR with kernel='linear', but use of another library
  - Scale better to large number of samples (up to 100000)

```
>>> from sklearn.svm import LinearSVR
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, random_state=0)
>>> regr = LinearSVR(random_state=0, tol=1e-5)
>>> regr.fit(X, y)
LinearSVR(random_state=0, tol=1e-05)
>>> print(regr.coef_)
[16.35... 26.91... 42.30... 60.47...]
>>> print(regr.intercept_)
[-4.29...]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-4.29...]
```

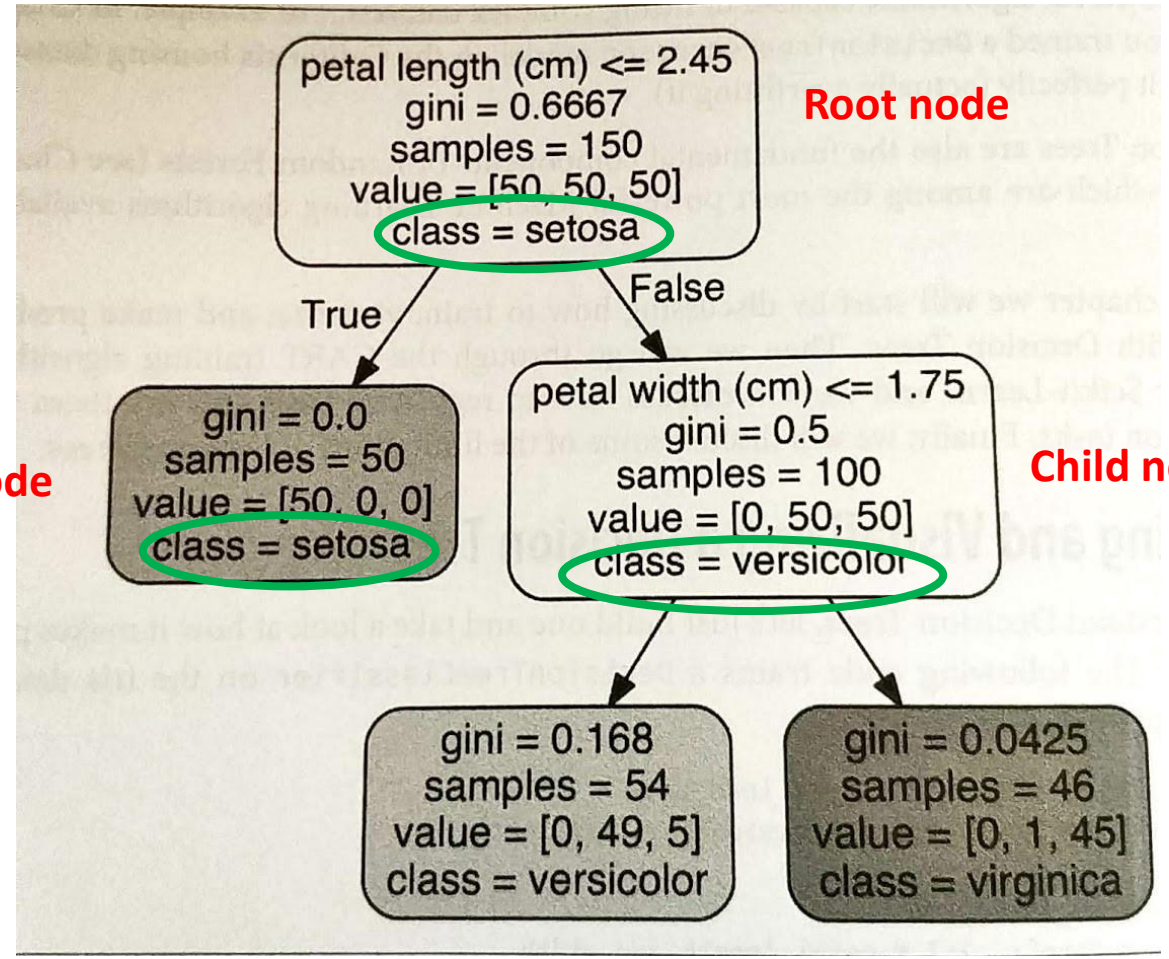


Ensemble Methods

Decision Trees

- Fundamental components of Random Forests

Classification example :

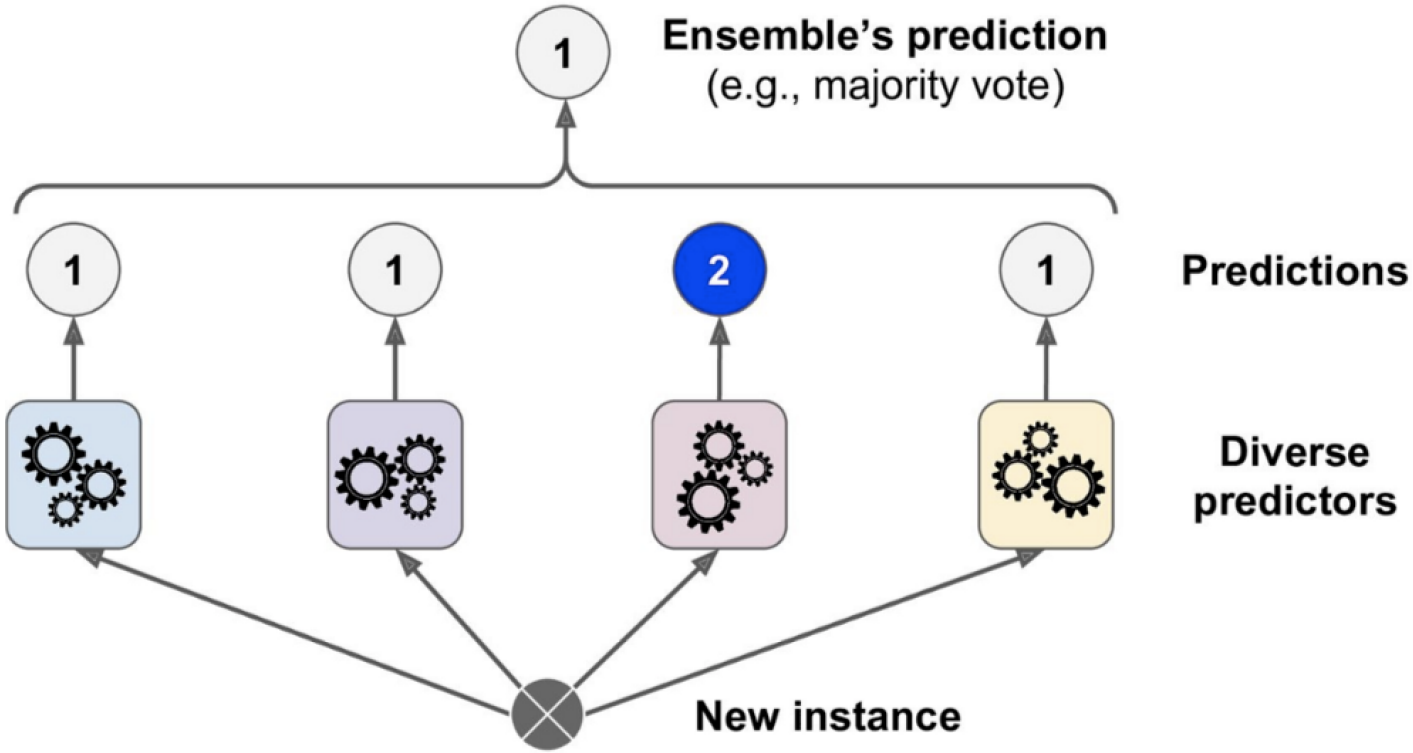


- Regularization : maximum depth of the tree

Ensemble Methods

- Decision Trees are very sensitive to small variations in the training data.

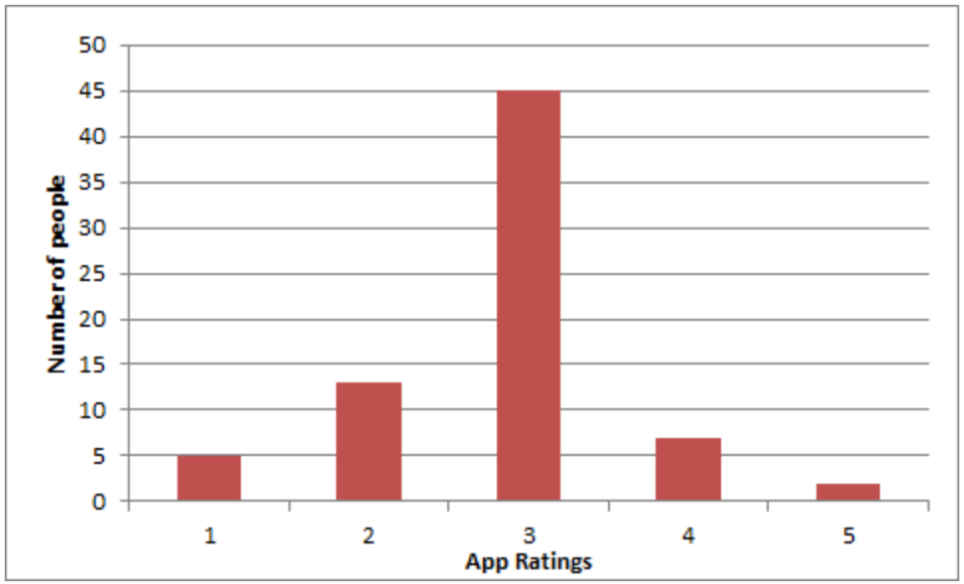
- *Wisdom of the crowd :* aggregate predictions of a group of predictors → Ensemble methods



Types of Ensemble Techniques

- Simple ensemble techniques
  - Mode, average, weighted average
- Advanced ensemble techniques
  - Bagging (Bootstrap AGGregatING)
  - Boosting

Simple Ensemble Techniques



Person	Professional	Weight	Rating
A	Y	0.3	3
B	Y	0.3	2
C	Y	0.3	2
D	N	0.15	4
E	N	0.15	3

1) Take the mode of the results

MODE=3, as majority people voted this

2) Take the average of the results (rounded to the nearest integer)

AVERAGE= (1*5)+(2*13)+(3*45)+(4*7)+(5*2)/72 = 2.833 = 3

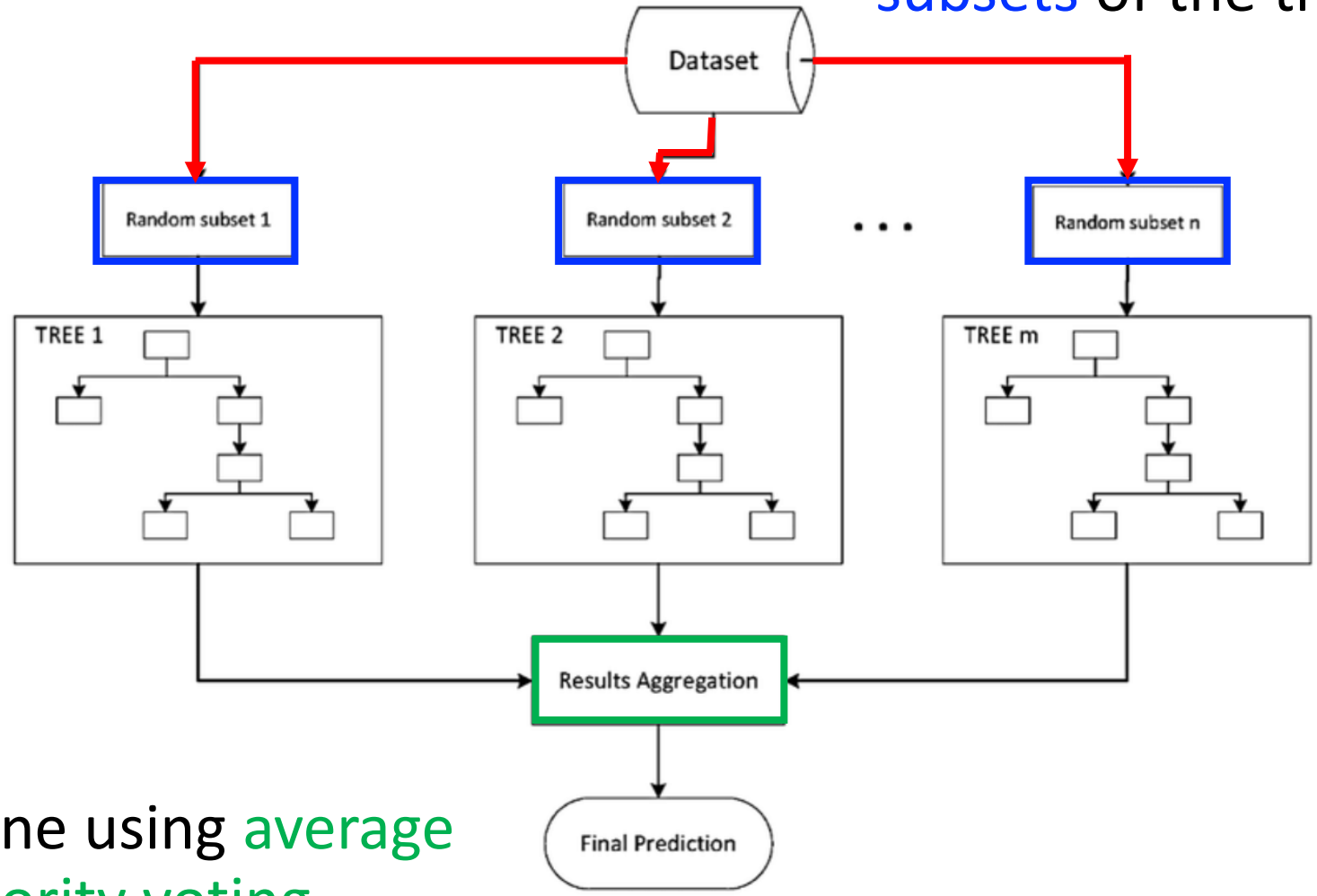
3) Take the weighted average of the results

WEIGHTED AVERAGE= (0.3*3)+(0.3*2)+(0.3*2)+(0.15*4)+(0.15*3) =3.15 = 3

Bagging

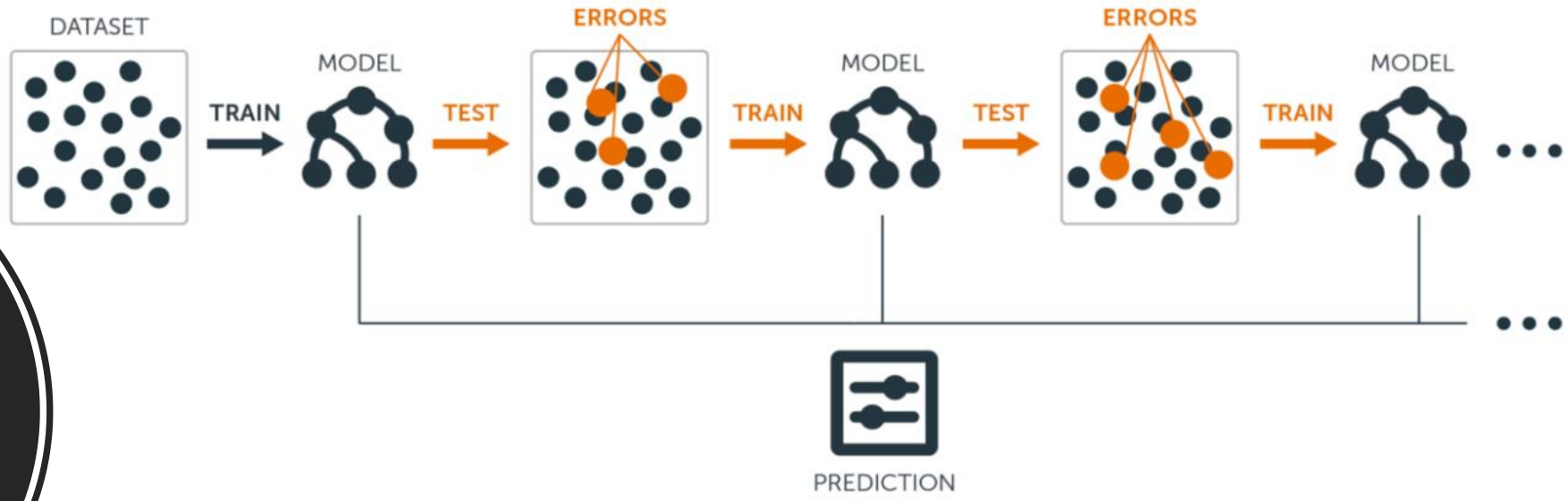
Use the same training algorithm for every predictor (e.g. classification tree)

Train on different random subsets of the training set



Combine using average or majority voting

Boosting



- Train predictors sequentially, each trying to correct the predecessor
  - combine several weak learners into a strong learner

- Examples of methods : AdaBoost and Gradient Boosting

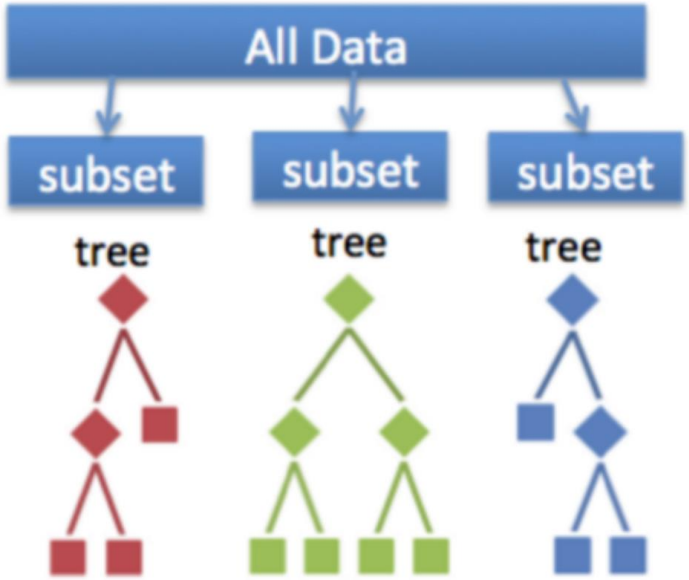
Which one is which ?

Comparison

Similarities	• Uses voting • Combines models of the same type	
Differences	Individual models are built separately	Each new model is influenced by the performance of those built previously
	Equal weight is given to all models	Weights a model's contribution by its performance
Primary error reduction	Variance	Bias
Overfitting	Prevented, as each model only sees part of the data (helps decreasing the variance error)	Tends to overfit the training data (parameter tuning is crucial)

- can be thought of as Bagging, with a slight tweak

Random Forest

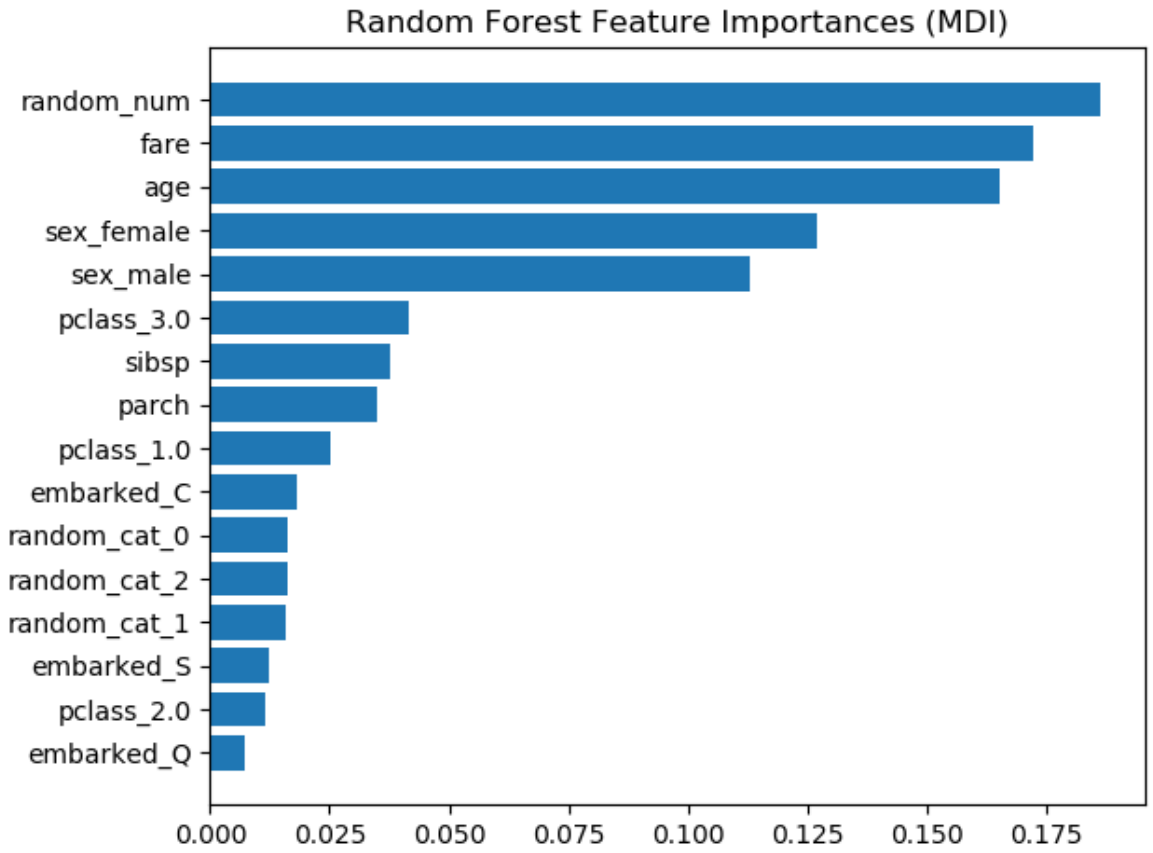


- Similarity : bootstrapped subsamples are pulled from a larger dataset.
- Difference : it searches for the best feature among a random subset of features

- greater tree diversity, which trades a higher bias for a lower variance

Feature Importance

- Relative importance of each feature, sum=1
- Useful for feature selection



Ensemble Methods Pros and Cons

- Pros :
  - More accurate prediction results
    - better performance on unseen data as compared to the individual models in most of the cases
  - Stable and more robust
    - aggregate result of multiple models is always less noisy than the individual models
  - Used to capture the linear/non-linear relationships in data

- Cons :
  - Computation and design time is high
    - not good for real time applications
  - Selection of models for creating an ensemble is an Art !

Averaging methods in practice

	Classification	Regression

Bagging	`>>> from sklearn.ensemble import BaggingClassifier`
	`>>> from sklearn.neighbors import KNeighborsClassifier`
	`>>> bagging = BaggingClassifier(KNeighborsClassifier(),`
	`...  max_samples=0.5, max_features=0.5)`
		`>>> from sklearn.svm import SVR`
		`>>> from sklearn.ensemble import BaggingRegressor`
		`>>> from sklearn.datasets import make_regression`
		`>>> X, y = make_regression(n_samples=100, n_features=4,`
		`...  n_informative=2, n_targets=1,`
		`...  random_state=0, shuffle=False)`
		`>>> regr = BaggingRegressor(base_estimator=SVR(),`
		`...  n_estimators=10, random_state=0).fit(X, y)`
		`>>> regr.predict([[0, 0, 0, 0]])`
		`array([-2.8720...])`

RandomForest	`>>> from sklearn.ensemble import RandomForestClassifier`
	`>>> X = [[0, 0], [1, 1]]`
	`>>> Y = [0, 1]`
	`>>> clf = RandomForestClassifier(n_estimators=10)`
	`>>> clf = clf.fit(X, Y)`
		`>>> from sklearn.ensemble import RandomForestRegressor`
		`>>> from sklearn.datasets import make_regression`
		
		`>>> X, y = make_regression(n_features=4, n_informative=2,`
		`...  random_state=0, shuffle=False)`
		`>>> regr = RandomForestRegressor(max_depth=2, random_state=0)`
		`>>> regr.fit(X, y)`
		`RandomForestRegressor(max_depth=2, random_state=0)`
		`>>> print(regr.feature_importances_)`
		`[0.18146984 0.81473937 0.00145312 0.00233767]`
		`>>> print(regr.predict([[0, 0, 0, 0]]))`
		`[-8.32987858]`

Boosting methods in practice

* For >10000 samples : HistGradientBoosting is faster

AdaBoost

Classification

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier

>>> X, y = load_iris(return_X_y=True)
>>> clf = AdaBoostClassifier(n_estimators=100)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.9...
```

Regression

```
>>> from sklearn.ensemble import AdaBoostRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, n_informative=2,
...                        random_state=0, shuffle=False)
>>> regr = AdaBoostRegressor(random_state=0, n_estimators=100)
>>> regr.fit(X, y)
AdaBoostRegressor(n_estimators=100, random_state=0)
>>> regr.feature_importances_
array([0.2788..., 0.7109..., 0.0065..., 0.0036...])
>>> regr.predict([[0, 0, 0, 0]])
array([4.7972...])
>>> regr.score(X, y)
0.9771...
```

GradientBoosting*

Classification

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.913...
```

Regression

```
>>> import numpy as np
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor

>>> X, y = make_friedman1(n_samples=1200, random_state=0, noise=1.0)
>>> X_train, X_test = X[:200], X[200:]
>>> y_train, y_test = y[:200], y[200:]
>>> est = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
...     max_depth=1, random_state=0, loss='ls').fit(X_train, y_train)
>>> mean_squared_error(y_test, est.predict(X_test))
5.00...
```

Quiz

https://b.socrative.com/login/student/

Room : CONTI6128