

Sklearn, Généralisation & Overfitting

Jour 2 — Après-midi

Julien Rolland

Formation M2 Développement Fullstack

Jour 2

- 1 Le Vrai Objectif : Généraliser
- 2 L'Ennemi N°1 : l'Overfitting
- 3 Biais-Variance
- 4 Méthodologie : Train / Validation / Test
- 5 Cross-Validation
- 6 Scikit-Learn

Le Piège

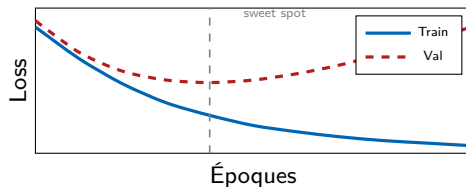
- Minimiser $J(\Theta, X_{\text{train}})$ = problème résolu
- La performance sur le train **ne compte pas**

Ce qu'on veut vraiment

- Données **non-vues** (unseen data)
- Capter la loi sous-jacente
- Ignorer le bruit du dataset

Définition

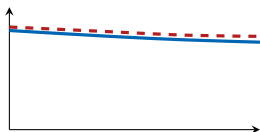
Généralisation : capacité à performer sur des exemples jamais vus pendant l'entraînement.



Underfitting vs Overfitting

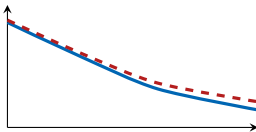
Underfitting

- Modèle trop **simple** — biais élevé
- $\text{Train} \approx \text{Val} \Rightarrow$ toutes deux élevées



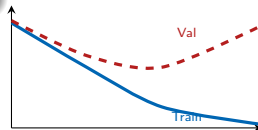
Bon fit

- Complexité adaptée aux données
- $\text{Train} \approx \text{Val} \Rightarrow$ toutes deux basses



Overfitting

- Modèle trop **complexe** — variance élevée
- $\text{Train} \ll \text{Val} \Rightarrow$ Val explose

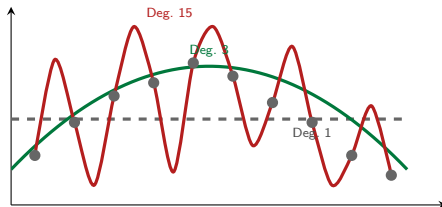


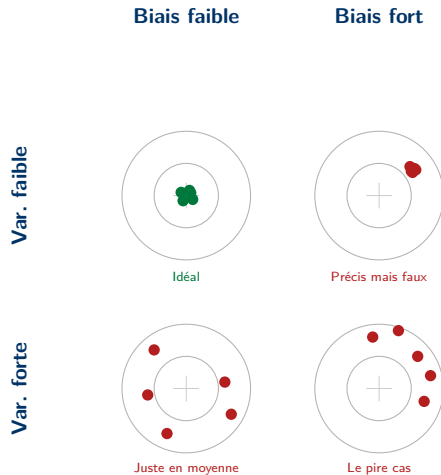
Théorème de Cover

- Dans \mathbb{R}^D , $N \leq D$ points sont toujours linéairement séparables
- Un modèle assez complexe peut **mémoriser** n'importe quel dataset
- Accuracy train = 100% ne signifie rien

Analogie MNIST

- Mémoriser les pixels exacts de chaque «3»
- Incapable de reconnaître un «3» écrit avec un stylo différent
- \Rightarrow 0% de généralisation





En Machine Learning

- **Biais** : erreur systématique
modèle trop simple
- **Variance** : sensibilité aux données
modèle trop complexe

$$\text{Erreur} = \text{Biais}^2 + \text{Variance} + \varepsilon$$

Leviers

- **Complexité** du modèle
- **Régularisation** (λ)
- **Volume** de données



3 ensembles distincts

- 1 **Train** — ajuster les paramètres Θ
- 2 **Validation** — choix du modèle et des hyperparamètres
- 3 **Test** — mesure finale

Règle d'or : Data Leakage

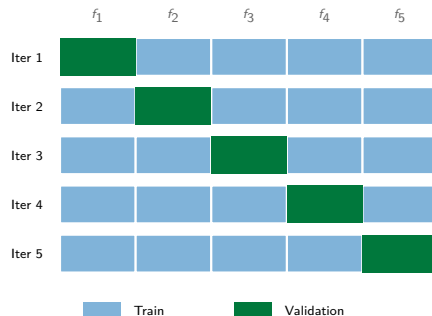
Ne **jamais** entraîner sur le test set.
Ne **jamais** utiliser le test pour choisir les HP.
Le test set doit rester **invisible**.

Le Problème

- Peu de données \Rightarrow split instable
- Un seul val set \Rightarrow variance élevée de l'estimation

Solution : K-Fold

- Découper le train en K morceaux (*folds*)
- K rotations : chaque fold sert de validation
- Score final = moyenne sur K runs
- Utilisation maximale des données disponibles



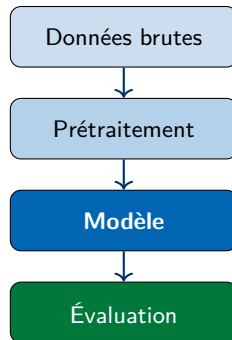
$$s = \frac{1}{K} \sum_{k=1}^K s_k$$

La bibliothèque ML de référence

- Open-source, basée sur NumPy / SciPy
- Standard de l'industrie pour le ML classique
- Interface cohérente pour tous les algorithmes

Ce qu'elle couvre

- **Prétraitement** : scaling, encodage, imputation
- **Modèles** : régression, classification, clustering
- **Validation** : cross-validation, grid search
- **Métriques** : accuracy, F1, AUC, ...



Interface universelle

<code>.fit(X, y)</code>	Apprend les paramètres internes sur les données d'entraînement
<code>.predict(X)</code>	Prédit les labels / valeurs pour de nouveaux exemples
<code>.transform(X)</code>	Applique la transformation apprise (scalers, encodeurs...)
<code>.fit_transform(X)</code>	<code>.fit()</code> + <code>.transform()</code> en une seule passe
<code>.score(X, y)</code>	Retourne une métrique : accuracy (classif), R^2 (régression)

```
1 from sklearn.linear_model import LogisticRegression
2 from sklearn.preprocessing import StandardScaler
3
4 scaler = StandardScaler()
5 X_tr = scaler.fit_transform(X_train)  # fit sur train uniquement
6 X_te = scaler.transform(X_test)      # applique la meme normalisation
7
8 clf = LogisticRegression()
9 clf.fit(X_tr, y_train)
10 print(clf.score(X_te, y_test))
```

Principe

- Enchaîner Scaler → Modèle
- `.fit()` appliqué séquentiellement
- Compatible avec la cross-validation

```
1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import StandardScaler
3 from sklearn.linear_model import LogisticRegression
4
5 pipe = Pipeline([
6     ('scaler', StandardScaler()),
7     ('clf', LogisticRegression()),
8 ])
9
10 pipe.fit(X_train, y_train)
11 print(pipe.score(X_test, y_test))
```

train_test_split

- Découpe aléatoire du dataset
- test_size : proportion du test set
- random_state : reproductibilité
- stratify : préserve les proportions de classes

cross_val_score

- K-Fold intégré, compatible Pipeline
- Retourne un score par fold
- cv : nombre de folds
- scoring : métrique ('accuracy', ...)

```
1 from sklearn.model_selection import train_test_split, cross_val_score
2
3 X_train, X_test, y_train, y_test = train_test_split(
4     X, y, test_size=0.2, random_state=42, stratify=y)
5
6 scores = cross_val_score(pipe, X_train, y_train, cv=5)
7 print(f"CV: {scores.mean():.3f} +/- {scores.std():.3f}")
```

Classification

- `LogisticRegression` — linéaire, rapide
- `SVC` — à noyau, puissant
- `KNeighborsClassifier` — basé distance
- `RandomForestClassifier` — ensemble
- `GradientBoostingClassifier` — boosting

Régression

- `LinearRegression` — OLS
- `Ridge` / `Lasso` — régularisés
- `SVR` — à noyau
- `RandomForestRegressor`
- `GradientBoostingRegressor`

Interface universelle

`.fit(X, y)`

`.predict(X)`

`.score(X, y)`

Normalisation

- StandardScaler — $\mu = 0$, $\sigma = 1$
- MinMaxScaler — $[0, 1]$
- RobustScaler — robuste aux outliers

Encodage & Imputation

- OneHotEncoder — catégorielles → binaires
- LabelEncoder — labels → entiers
- SimpleImputer — valeurs manquantes

```
1 from sklearn.preprocessing import (  
2     StandardScaler, OneHotEncoder,  
3     SimpleImputer,  
4 )  
5  
6 # normalisation  
7 scaler = StandardScaler()  
8 X_sc = scaler.fit_transform(X_train)  
9  
10 # valeurs manquantes  
11 imp = SimpleImputer(strategy='mean')  
12 X_cl = imp.fit_transform(X_train)  
13  
14 # variables catégorielles  
15 enc = OneHotEncoder()  
16 X_cat = enc.fit_transform(X_train)
```

Classification

- accuracy_score
- f1_score — macro / weighted
- confusion_matrix
- classification_report
- roc_auc_score

Régression

- mean_squared_error (MSE)
- mean_absolute_error (MAE)
- r2_score — coefficient R^2

```
1 from sklearn.metrics import (  
2     accuracy_score,  
3     classification_report,  
4     confusion_matrix,  
5 )  
6  
7 y_pred = clf.predict(X_test)  
8  
9 print(accuracy_score(y_test, y_pred))  
10 print(classification_report(  
11     y_test, y_pred))  
12 print(confusion_matrix(  
13     y_test, y_pred))
```

GridSearchCV

- Teste toutes les combinaisons
- Cross-validation intégrée
- Compatible Pipeline

RandomizedSearchCV

- Échantillonnage aléatoire de l'espace
- Plus rapide sur de grands espaces
- `n_iter` itérations

```
1 from sklearn.model_selection import (  
2     GridSearchCV)  
3 from sklearn.svm import SVC  
4  
5 param_grid = {  
6     'C': [0.1, 1, 10],  
7     'kernel': ['linear', 'rbf'],  
8 }  
9  
10 grid = GridSearchCV(  
11     SVC(), param_grid, cv=5,  
12     scoring='accuracy')  
13 grid.fit(X_train, y_train)  
14  
15 print(grid.best_params_)  
16 print(grid.best_score_)
```


Généralisation

- Objectif : données **non vues**, pas le train
- Underfitting — biais élevé, modèle trop simple
- Overfitting — variance élevée, modèle trop complexe
- Compromis biais-variance : le *sweet spot*

Méthodologie

- Split **Train** / **Val** / **Test** — test intouchable
- **K-Fold** — estimation robuste, peu de données

API Scikit-Learn

- `.fit()` / `.predict()` / `.score()`
- Pipeline — prétraitement + modèle chaînés
- GridSearchCV — recherche d'hyperparamètres

Modules clés

- preprocessing — scalers, encodeurs
- linear_model / ensemble / svm
- metrics — accuracy, F1, R^2
- model_selection — CV, GridSearch