Méthodes d'agrégation avec Python

Novembre 2025

Cédric Dangeard
cedric.dangeard@orange.com

Cours 5 : Au programme

- Evaluation (QCM)
- Pipelines
- Faire ces propres estimateurs
- TP : Pipelines
- Arbres sur R
- TP : R

PipeLines

Objectif :

- Simplifier le flux de travail de prétraitement et de modélisation.
- Faciliter la gestion des étapes de transformation des données et du modèle.

PipeLines

Solution :

Utilisation de la classe `Pipeline` de `sklearn.pipeline`.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

clf = Pipeline(
    steps=[("scaler", StandardScaler()), ("classifier", LogisticRegression())]
)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

clf.fit(X_train, y_train)
```

- Composition d'un ensemble d'étapes de transformation et d'un modèle final.

ColumnTransformer

Objectif : Appliquer différentes transformations à différentes colonnes d'un DataFrame.

Solution : Utilisation de la classe `ColumnTransformer` de `sklearn.compose`.

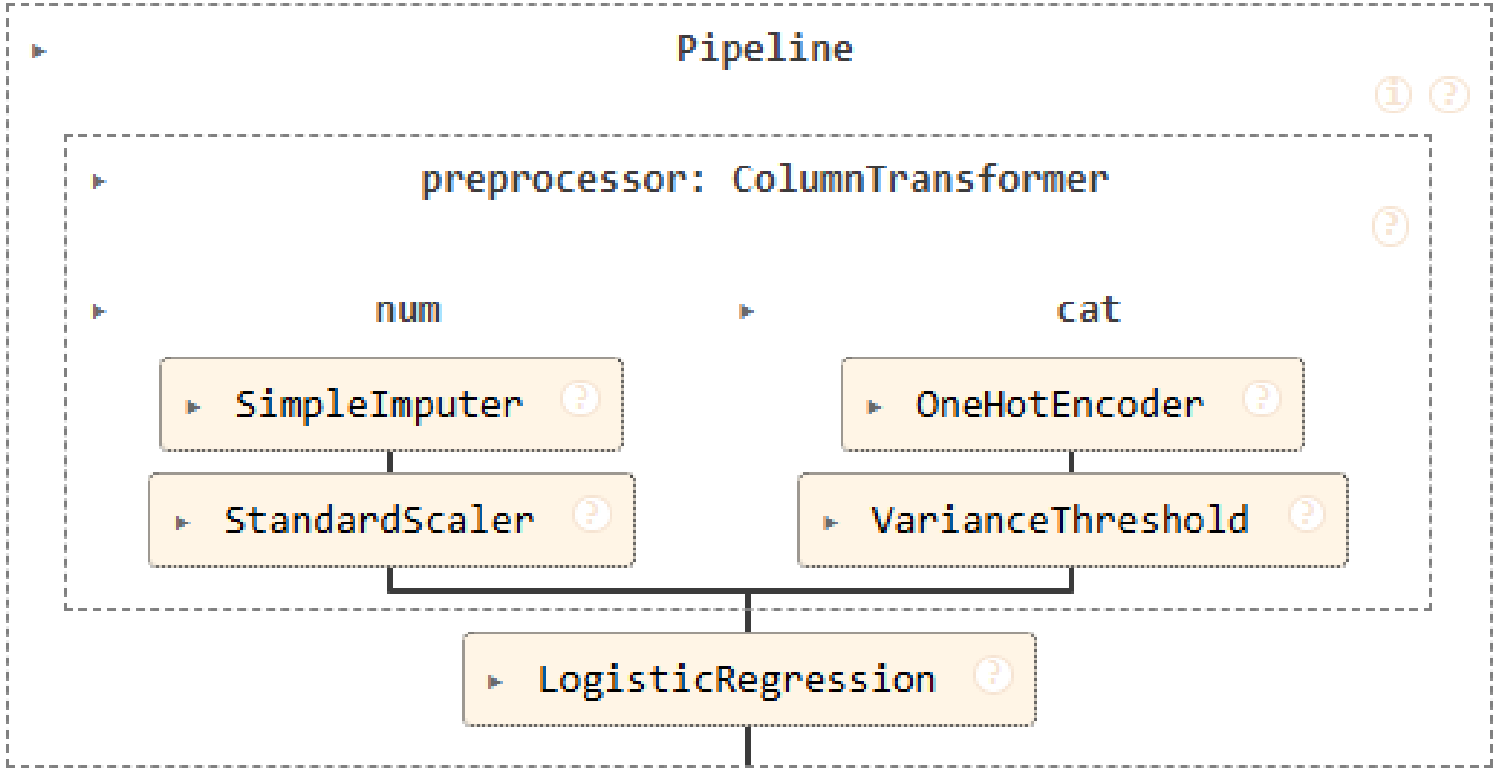
Exemple 2

```
numeric_features = ["age", "fare"]
categorical_features = ["embarked", "sex", "pclass"]


numeric_transformer = Pipeline(
    steps=[("imputer", SimpleImputer(strategy="median")), ("scaler", StandardScaler())]
)

categorical_transformer = Pipeline(
    steps=[
        ("encoder", OneHotEncoder(handle_unknown="ignore")),
        ("selector", VarianceThreshold(threshold=0.01)),
    ]
)
preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_features),
        ("cat", categorical_transformer, categorical_features),
    ]
)

clf = Pipeline(
    steps=[("preprocessor", preprocessor),
           ("classifier", LogisticRegression())]
)
```


Pipeline
preprocessor: ColumnTransformer
num
cat
SimpleImputer
StandardScaler
OneHotEncoder
VarianceThreshold
LogisticRegression

Faire ses propres estimateurs

Objectif : Créer des transformations personnalisées ou des modèles spécifiques.

Solution : Hériter des classes `BaseEstimator` et `TransformerMixin` de `sklearn.base`.

`BaseEstimator` : Fournit des fonctionnalités de base pour les estimateurs, comme la gestion des hyperparamètres.

`TransformerMixin` : Fournit la méthode `fit_transform` pour les transformations.

Exemple 3

```python
from sklearn.base import TransformerMixin

class MeanOrModImputer(TransformerMixin):
    """Remplace par la moyenne ou le mode"""
    def __init__(self) -> None:
        self.valeurToImpute = {}

    def fit(self, X, y=None):
        """
        Apprend les valeurs de remplacement pour les autres colonnes.
        """
        for col in X.columns:
            if X[col].dtype != 'float64':
                self.valeurToImpute[col] = X[col].mode()[0]
            else:
                self.valeurToImpute[col] = X[col].mean()
        return self

    def transform(self, X, y=None):
        """
        Applique le nettoyage aux données.
        """
        Xtransformed = X.copy()
        for col, val in self.valeurToImpute.items():
            Xtransformed[col] = Xtransformed[col].fillna(val)
        return Xtransformed
```

Quel différences?

```
imputeur = MeanOrModImputer()

X_train_clean = imputeur.fit_transform(X_train)
X_test_clean = imputeur.transform(X_test)
```

```
def imputeur_maison(X):
    Xtransformed = X.copy()
    for col in X.columns:
        if X[col].dtype != 'float64':
            Xtransformed[col].fillna(X[col].mode()[0])
        else:
            Xtransformed[col].fillna(X[col].mean())
    return Xtransformed

X_train_clean2 = imputeur_maison(X_train)
X_test_clean2 = imputeur_maison(X_test)
```