Шпаргалка по Scikit-learn / Sklearn Cheatsheet (XeLaTeX)

Краткий справочник по основным операциям

Содержание

Scikit-learn: Ваш Швейцарский Нож в ML

Scikit-learn (sklearn) — это фундаментальная библиотека Python для классического машинного обучения. Она предоставляет единообразный интерфейс (**API**) для большинства алгоритмов и инструментов предобработки, оценки и выбора моделей. Знание её основ — **must-have**. **Ключевая сила — единообразный интерфейс**, позволяющий легко пробовать разные алгоритмы без необходимости изучать новый синтаксис для каждой модели. **Аналогия:** Представь себе ящик с инструментами. Sklearn — это такой ящик, где все инструменты (модели, препроцессоры) имеют похожие ручки (методы 'fit', 'predict', 'transform'), что сильно упрощает работу.

1 Основной API: Глаголы ML

Ключевые Методы Оценщиков (Estimators)

Все объекты sklearn, которые учатся на данных (модели, препроцессоры), называются **оценщиками (estimators)**. У них есть стандартные методы:

- `fit(X, y)`: "Обучись". Главный метод для тренировки модели. Принимает обучающие данные ('X') и целевую переменную ('y', если модель с учителем). 'y' **не требуется для большинства препроцессоров** (например, Scaler, Encoder, Imputer): `fit(X)`. **Аналогия:** Это как показать собаке команды ('X') и правильные реакции ('y'), чтобы она научилась. Или как настроить инструмент по эталону ('X').
- `predict(X)`: "Предскажи". Используется *после* `fit`. Генерирует предсказания для новых данных 'X'. **Аналогия:** Попросить обученную собаку выполнить команду на новых данных.
- `predict_proba(X)`: "Оцени вероятности". Используется *после* `fit` для **классификаторов**. Возвращает вероятности принадлежности к каждому классу (массив `[n_samples, n_classes]`). Очень полезно для оценки уверенности модели и для построения ROC-кривых. **Аналогия:** Спросить у собаки, насколько она уверена, что это именно та команда (например, 80
- `transform(X)`: "Преобразуй". Используется *после* `fit` для **препроцессоров** (например, 'StandardScaler', 'PCA', 'OneHotEncoder'). Применяет выученное преобразование к данным 'X'. **Аналогия:** Использовать настроенный инструмент (например, линейку, откалиброванную по `fit`) для измерения новых объектов.
- `fit_transform(X, y=None)`: "Обучись и преобразуй". Оптимизированная комбинация `fit(X)` и `transform(X)`. Часто используется для обучающей выборки, чтобы избежать повторных вычислений. **Важно:** Применять только к **обучающей** выборке, чтобы избежать **утечки информации из тестовой выборки** в процесс обучения препроцессора! Для тестовой выборки используется только `transform`. **Аналогия:** Найти среднее и стандартное отклонение (fit) и сразу же стандартизировать данные (transform) за один проход.
- `score(X, y)`: "Оцени качество". Используется *после* `fit`. Возвращает метрику по умолчанию (accuracy для классификаторов, R^2 для регрессоров) на данных 'X', 'y'. Удобно для быстрой проверки.

Базовый пример API

```python
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np

# Допустим, есть данные X (фичи) и y (цель)
# X = np.array(...)
# y = np.array(...)
# X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2)

# 1. Препроцессор (Scaler)
# Масштабирование часто улучшает сходимость и
производительность моделей, чувствительных к масштабу
признаков (линейные модели, SVM, нейросети, kNN).
scaler = StandardScaler()
# Обучаем на ТРЕНИРОВОЧНЫХ данных
scaler.fit(X_train)
# Преобразуем ТРЕНИРОВОЧНЫЕ и ТЕСТОВЫЕ данные
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Альтернатива для трейна: X_train_scaled =
scaler.fit_transform(X_train)

# 2. Модель (Классификатор)
model = LogisticRegression()
# Обучаем модель на масштабированных ТРЕНИРОВОЧНЫХ
данных
model.fit(X_train_scaled, y_train)

# 3. Предсказания
# Предсказания классов для теста
y_pred = model.predict(X_test_scaled)
# Предсказания вероятностей для теста
y_pred_proba = model.predict_proba(X_test_scaled)

print(f"Предсказанные классы: {y_pred[:5]}")
print(f"Вероятности классов: \n{y_pred_proba[:5]}")
# Оценка качества (accuracy по умолчанию для
классификатора)
# print(f"Точность на тесте:
{model.score(X_test_scaled, y_test)}")
```

2 Pipeline: Конвейер Обработки

Зачем нужен Pipeline?

Часто рабочий процесс ML включает несколько шагов предобработки (масштабирование, кодирование категорий, извлечение признаков) перед обучением модели. **Pipeline** позволяет объединить эти шаги в единый объект (оценщик). **Преимущества:**

- **Удобство:** Все шаги выполняются одной командой `fit` и `predict`/`transform`.
- **Предотвращение утечки данных (Data Leakage):** Гарантирует, что `fit` препроцессоров вызывается *только* на обучающих фолдах при кросс-валидации, а `transform` — на обучающих и валидационных/тестовых. Это **критически важно** для получения несмещенной оценки качества модели.
- **Легкость настройки:** Гиперпараметры шагов пайплайна можно подбирать с помощью 'GridSearchCV'/'RandomizedSearchCV'.

Аналогия: Пайплайн — это как сборочный конвейер на заводе. Сырье (данные) проходит через несколько станций обработки (препроцессоры) и на выходе получается готовый продукт (предсказание модели).

Пример Pipeline

```python
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer # Для заполнения пропусков
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
# Допустим, есть X_train, y_train

# Создаем пайплайн:
# Шаг 1: Заполнение пропусков средним
# Шаг 2: Масштабирование числовых признаков
# Шаг 3: Обучение модели
pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')), # Имя шага, объект
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

# Обучаем весь пайплайн как единое целое
pipe.fit(X_train, y_train)

# Делаем предсказания (данные пройдут через imputer -> scaler -> predict)
# y_pred_pipe = pipe.predict(X_test)
# print(f"Точность пайплайна: {pipe.score(X_test, y_test)}")
```

3 ColumnTransformer: Работа с Разными Типами Признаков

Обработка Гетерогенных Данных

Реальные данные часто содержат столбцы разных типов: числовые, категориальные. К ним нужно применять разную предобработку (например, масштабирование к числовым, One-Hot Encoding к категориальным). **ColumnTransformer** позволяет применять разные трансформаторы к разным подмножествам столбцов. Сам ‘ColumnTransformer‘ является таким же *оценщиком (estimator)* sklearn и может использоваться как самостоятельно, так и (что чаще всего) в качестве шага внутри ‘Pipeline‘. **Аналогия:** Представь, что в больницу пришел пациент (строка данных). У него есть разные проблемы (признаки): перелом (числовой признак), аллергия (категориальный). ‘ColumnTransformer‘ — это регистратура, которая направляет пациента к нужным специалистам: травматологу (масштабирование) и аллергологу (One-Hot Encoding).

Пример ColumnTransformer в Pipeline

```python
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import pandas as pd

# Допустим, есть DataFrame df с числовыми и категориальными фичами
# df = pd.DataFrame(...)
# X = df.drop('target', axis=1)
# y = df['target']
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Определяем числовые и категориальные столбцы
numeric_features = X_train.select_dtypes(include=['int64', 'float64']).columns
categorical_features = X_train.select_dtypes(include=['object', 'category']).columns

# Создаем пайплайны для каждого типа признаков
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))]) # handle_unknown='ignore' важен!

# Создаем ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features), # Имя, трансформер, колонки
        ('cat', categorical_transformer, categorical_features)])

# Создаем основной пайплайн, включающий препроцессор и модель
model_pipe = Pipeline(steps=[('preprocessor', preprocessor),
                             ('classifier', LogisticRegression())])

# Обучаем и используем как обычно
model_pipe.fit(X_train, y_train)
# y_pred = model_pipe.predict(X_test)
# print(f"Точность модели с ColumnTransformer: {model_pipe.score(X_test, y_test)}")
```

4 Подбор Гиперпараметров

GridSearchCV vs RandomizedSearchCV

Большинство моделей имеют **гиперпараметры** — настройки, которые не выучиваются из данных, а задаются до обучения (например, глубина дерева, коэффициент регуляризации). Их нужно подбирать.

- **GridSearchCV**: ”Поиск по сетке”. Перебирает **все возможные комбинации** заданных гиперпараметров. Тщательно, но медленно, особенно если параметров много. **Аналогия:** Пекарь пробует испечь хлеб при всех комбинациях температуры (180, 200, 220) и времени (30, 40, 50 минут) — всего 3*3=9 попыток.
- **RandomizedSearchCV**: ”Случайный поиск”. Выбирает **случайные комбинации** гиперпараметров из заданных диапазонов или распределений. Работает быстрее GridSearchCV, часто находит достаточно хорошие параметры за меньшее число итераций (`n_iter`). Особенно эффективен, когда параметров много или не все параметры одинаково важны (позволяет быстрее найти ”достаточно хорошую” область). **Аналогия:** Пекарь пробует случайные 5 комбинаций температуры и времени из возможных диапазонов. Может не найти идеал, но быстро найдет хороший вариант.

Оба инструмента используют **кросс-валидацию (CV)** для оценки качества каждой комбинации параметров на обучающей выборке, чтобы избежать переобучения на конкретном разбиении.

Пример GridSearchCV (с Pipeline)

```python
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
# Используем model_pipe из примера с ColumnTransformer, но заменим LogReg на RF
# model_pipe = Pipeline(steps=[('preprocessor', preprocessor),
#                              ('classifier', RandomForestClassifier(random_state=42))])

# Задаем сетку параметров для поиска
# Обратите внимание на синтаксис для доступа к параметрам шага пайплайна:
# 'имя_шага__имя_параметра'
param_grid = {
    'preprocessor__num__imputer__strategy': ['mean', 'median'], # Параметр вложенного пайплайна
    'classifier__n_estimators': [100, 200], # Параметр модели
    'classifier__max_depth': [None, 10, 20],
    'classifier__min_samples_split': [2, 5]
}

# Создаем объект GridSearchCV
# cv=5 означает 5-кратную кросс-валидацию
# n_jobs=-1 использует все доступные ядра процессора
# Важно выбрать метрику (scoring), соответствующую задаче! Не всегда accuracy - лучший выбор.
grid_search = GridSearchCV(model_pipe, param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=1)

# Запускаем поиск (может занять время!)
grid_search.fit(X_train, y_train)

# Лучшие параметры и лучший скор
print(f"Лучшие параметры: {grid_search.best_params_}")
print(f"Лучший скор на CV: {grid_search.best_score_:.4f}")

# Лучшая модель уже обучена на всех трейн данных с лучшими параметрами
best_model = grid_search.best_estimator_
# y_pred_best = best_model.predict(X_test)


# print(f"Точность лучшей модели на тесте: {best_model.score(X_test, y_test)}")

# Для RandomizedSearchCV синтаксис похож, но вместо param_grid
# используется param_distributions (словари с распределениями scipy.stats)
# и добавляется параметр n_iter.
```

5 Часто Используемые Импорты

Джентльменский Набор для Старта

Этот список поможет быстро начать работу над типичной задачей.

Основные импорты моделей и метрик

```python
# --- Модели ---
# Линейные модели
from sklearn.linear_model import LinearRegression # Регрессия
from sklearn.linear_model import LogisticRegression # Классификация
from sklearn.linear_model import Ridge, Lasso # Регрессия с регуляризацией

# Метод опорных векторов (SVM)
from sklearn.svm import SVC # Классификация
from sklearn.svm import SVR # Регрессия

# Деревья решений
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import DecisionTreeRegressor

# Ансамбли: Бэггинг (случайный лес)
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import RandomForestRegressor

# Ансамбли: Бустинг
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingRegressor
# Популярные библиотеки бустинга (устанавливаются отдельно, часто производительнее)
# import xgboost as xgb # pip install xgboost
# import lightgbm as lgb # pip install lightgbm
# import catboost as cb # pip install catboost

# --- Предобработка ---
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline, make_pipeline # make_pipeline - упрощенное создание Pipeline без явного именования шагов

# --- Выборка и Оценка Моделей ---
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, KFold, StratifiedKFold
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

# --- Метрики ---
# Классификация
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.metrics import confusion_matrix, classification_report
# Регрессия
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# --- Другое ---
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```