Calvin Khor

Last compiled: November 22, 2023

Link to these notes and their source code:
https://github.com/clvnkhr/Quick-notes-sk-learn

Contents

1 Introduction

While I was trying to learn these tools for data analysis, I found the available discussion online to be dated; for example, `this talk` with corresponding `Github repo` were helpful but suggest making custom estimators purely to return Pandas DataFrames. These notes aim to get you using the newer features of scikit-learn quickly, to the point where you are comfortable creating your own estimators.

1.1 Setup

I will assume semi-recent versions of python (3.11), numpy (1.26), scipy (1.11.x), scikit-learn (1.3.2) and so on.

In the first block of your Jupyter notebook I would keep all the imports that you add later, so that it is easy to restart. I would also recommend settings like the following:

```
import pandas as pd
pd.options.display.max_columns = 1000
pd.options.display.max_rows = 2000
pd.options.display.width = 1000
pd.options.display.max_colwidth = 400
```

Since `scikit-learn 1.2`, there is good interop with Pandas: you can configure all transformers to output pandas DataFrames globally.

```
from sklearn import set_config
set_config(transform_output="pandas")
```

2 Basic Usage

We will assume that we are trying to perform prediction on some labelled training data which we will store in `X`, `y`, and the test data in `X_test`.

```
X = pd.read_csv("X_train.csv")
y_raw = pd.read_csv("Y_train.csv")
X_test = pd.read_csv("X_test.csv")
y = y_raw["TARGET"]
```

We will first assume that we want to design a `Pipeline`, fit it to the training data, predict with a regressor, and try to evaluate our performance. Later, we will see how to modify this to allow for gridsearching.

2.1 Pipeline

A a `Pipeline` is a way to combine estimators and predictors in a way that is easy to modify and develop. Documentation on `Pipeline`s `here`. To understand them, we have to first explain what scikit-learn estimators are: these are the building blocks that either transform your data, or learn and predict from them. An estimator `MyEstimator` is implemented as a Python class (usually inheriting from `BaseEstimator`). If it transforms, it has an `MyEstimator.transform` method; learning is done with the `MyEstimator.fit` method, and prediction is done with the `MyEstimator.predict` method.

An example of a transformer is `StandardScaler` which[1] scales to mean 0 and variance 1. One needs to `fit` to learn the parameters and then `transform`, or equivalently, use the convenience method `fit_transform`:

[1] another useful scaler is the `RobustScaler` which uses quantile information and is therefore more robust to outliers.

```
from sklearn.preprocessing import StandardScaler
X_scaled = StandardScaler().fit(X).transform(X)
X_scaled = StandardScaler().fit_transform(X) # same as the above
```

We quickly remark that the above method chaining implies that the methods have[2] the signatures

```
class MyEstimator:
# ...
def fit(self, X: pd.DataFrame) -> MyEstimator:
    # ...
    return self
def transform(self, X: pd.DataFrame) -> pd.DataFrame:
    # ...
    # return a pd.DataFrame
```

A very convenient transformer is the `FunctionTransformer` which applies an arbitrary Python function to the Pandas `DataFrame`. The function should take the dataframe as input, and return a new dataframe, which is the output of the `transform` method (`fit` is empty for `FunctionTransformer`s.) A simple example is if you wanted to drop a column called `"Rubbish"`, you could use[3]

```
from sklearn.preprocessing import FunctionTransformer
X_clean = FunctionTransformer(lambda df: df.drop(["Rubbish"],
↪  axis=1)).transform(X)
```

Once we are happy with the preprocessing, we need to make the predictions. For example we can fit e.g.[4] `LinearRegression` and predict on the test data:

```
    from sklearn.linear_model import LinearRegression
    ols = LinearRegression().fit(X,y)
    y_test = ols.predict(X_test)
```

The last thing to do is to evaluate the performance of our model. This will be project specific. The short answer is to use cross-validation since it is a model agnostic method of estimating the true error on an unforseen dataset, which is good for iterating to more complicated models. Cross-validation is implemented as part of many different functions in `scikit-learn` but for starters one can `from sklearn.model_selection import cross_validate`.

One can also use the training error as a rough upper bound but don't get too attached to it. Since we have the target predictions on the training set, we can plot the data. Below, I have some convenience functions defined in order to quickly evaluate the cross-validation score and plot the model's predictions against the target predictions. I'm using Spearman's rank correlation coefficient as a scoring method, which works better for nonlinear data.

[2] actually `fit` takes an optional `y` input for supervised learning. Estimators normally also inherit from `BaseEstimator`, which we say again later.

[3] Keep in mind though lambdas will prevent the transformer from pickling.

[4] This includes the intercept term by default.

```
from scipy.stats import spearmanr
from sklearn.metrics import make_scorer
from sklearn.preprocessing import QuantileTransformer
import matplotlib.pyplot as plt
import seaborn as sns


def spearman_metric(y_pred, y=y):
    """y_pred is the model prediction; y is the training data target"""
    return spearmanr(y_pred, y).correlation
spearman_scorer = make_scorer(spearman_metric)


def grade(y_pred, y=y) -> None:
    Xy = X[["COUNTRY"]].copy()
    Xy["TARGET"] = y
    Xy["PREDICTED"] = y_pred
    Xy[["TARGET", "PREDICTED"]] = QuantileTransformer().fit_transform(
        Xy[["TARGET", "PREDICTED"]]
    )

    _, ax = plt.subplots()
    plt.plot(Xy["TARGET"], Xy["TARGET"], label="y=x (perfect model)",
    ↪  alpha=0.3)
    sns.scatterplot(Xy, y="PREDICTED", x="TARGET", hue="COUNTRY",
    ↪  alpha=0.8, s=20)
    plt.xlabel("Actual Values" + (" (quantile)" if quantile else ""))
    plt.ylabel("Predicted Values" + (" (quantile)" if quantile else ""))
    plt.title(
        "Output vs Training Data\nSpearman correlation for the train set:
        ↪  {:.1f}%".format(
            100 * spearman_metric(y_pred, y)
        )
    )
    ax.legend(title=None)
    plt.show()


def perform_cv(
    estimator, data, cv=5, scorer=spearman_scorer, show=True, y=y,
    ↪  n_jobs=1, verbose=0
) -> pd.DataFrame:
    """displays cv test scores and returns the result from the cv.
    """
    cv_results = cross_validate(
```

```
        estimator, data, y, cv=cv, scoring=scorer, n_jobs=n_jobs,
        ↪  verbose=verbose
    )
    if show:
        # Print the mean and standard deviation of the test scores
        print(
            "Spearman correlation for the cross validation: {:.1f}% ±
            ↪  {:.1f}%".format(
                100 * cv_results["test_score"].mean(),
                100 * cv_results["test_score"].std(),
            )
        )
        print(f"Spearman correlation for each fold:
        ↪  {cv_results['test_score']}")
    return pd.DataFrame(cv_results)
```

2.2 Finally, the Pipeline

The upshot of the above code is that Pipelines allow me to perform the entire data analysis in a very short Jupyter code block:

```
pipe = Pipeline(
    [
        ("drop", FunctionTransformer(lambda df: df.drop(["COUNTRY"],
        ↪  axis=1))),
        ("scale", RobustScaler()),
        ("ols", LinearRegression()),
    ]
)
pipe.fit(X, y)
y_pred = pipe.predict(X)
grade(y_pred, y)
perform_cv(pipe, X)
```

What a Pipeline is then, is a way to convert a sequence of transformers and a final predictor into a single estimator. Calling `pipe.fit(X,y)` is equivalent to calling `fit_transform` on every transformer and fit on the predictor; calling `pipe.predict` calls `transform` on all the transformers and then `predict`:

```
# need to define the estimators separately if not using a pipeline
drop = FunctionTransformer(lambda df: df.drop(["COUNTRY"], axis=1))
scale = RobustScaler()
ols =  LinearRegression()
# below is the same as pipe.fit(X,y)
ols.fit(scale.fit_transform(drop.fit_transform(X)),y)
# below is the same as y_pred = pipe.predict(X)
y_pred = ols.predict(scale.transform(drop.transform(X)))
```

Note in particular that the order of appearance of each estimator in the pipeline corresponds to the order in which they are called, but it is reversed (and nested) in the non-pipeline version.

To use a pipeline, simply pass a list of tuples to the constructor. The second part of the tuple is simply the estimator, and the first part[5] is a name that can be used to inspect parts of the pipe:

```
# this pulls out the coefficients computed from ols
pipe_bench.named_steps["ols"].coef_
```

3 Gridsearching

Suppose instead of `LinearRegression`, we wanted to use `Lasso`, which modifies the loss function for least squares by an L^1 penalty term for the coefficients, i.e.

$$J(\beta) = \sum_{i=1}^{n} |y_i - (X\beta)_i|^2 + \alpha \sum_{j=1}^{p} |\beta_j|$$

The parameter α can be interpreted as a Lagrange multiplier. But since this minimisation problem cannot be solved symbolically, we have to treat it as a *tuning parameter* and determine it experimentally.

To use lasso, we import it and set up our pipeline:

```
from sklearn.linear_model import Lasso
pipe = Pipeline(
    [
        ("drop", FunctionTransformer(lambda df: df.drop(["COUNTRY"],
        ↪  axis=1))),
        ("scale", RobustScaler()),
        ("lasso", Lasso()),
    ]
)
```

`scikit-learn` has many ways to search for an optimal parameter. The simplest is `GridSearchCV`, which performs an exhaustive search in the given parameter space. I have written some helper functions (`display_grid_params` and `report`) as well. The overall code is as follows:

```
from icecream import ic
import time
# pipe code from above goes here
tick = time.time()
pipe.fit(X, y)
time_for_one_fit = time.time() - tick
ic(time_for_one_fit)
param_grid = {
```

[5] There is a variant, `make_pipeline` that avoids needing a name by creating a default one from the transformer.

```
    "model__alpha": [ 0.2 * np.exp(0.01 * k) for k in range(-5, 5)],
}
display_grid_params(param_grid, time_for_one_fit)
grid = GridSearchCV(
    pipe, param_grid=param_grid, cv=5, n_jobs=-1, scoring=spearman_scorer
)
grid.fit(X, y)
report(grid)
print("Predicting on train set using best params above:")
y_best = grid.predict(X)
grade(y_best, y)
```

For completeness, the helper functions are:

```
from icecream import ic
import functools
from operator import mul
def display_grid_params(params, time_for_one_fit=None):
    params_size = functools.reduce(mul, (len(params[k]) for k in params))
    note = f"The params grid has size {params_size}. "
    if time_for_one_fit:
        min, sec = divmod(time_for_one_fit * params_size, 60)
        hr, min = divmod(min, 60)
        note += f"Estimated time to completion: {hr}h {min}m {sec:.1f}s"
    ic(note)
    ic(params)


def report(grid, n_top=3):
    """Usage: fit outside the report with grid.fit(X,y). Then pass the
    ↪ cv_results_ to report.
    """
    cv_results_ = grid.cv_results_
    grid_df = pd.DataFrame(cv_results_)[
        ["params", "mean_test_score", "std_test_score", "rank_test_score"]
    ].sort_values(by="rank_test_score")

    if n_top != 0:
        ic(grid.best_params_, grid.best_score_)
    if n_top > 0:
        display(grid_df.head(n_top))
    elif n_top < 0:
        display(grid_df)
    return grid_df
```

See this part of the User Guide for more complicated (and potentially more efficient) methods of tuning hyper-parameters.

4 The need for custom estimators

The built-in estimators are powerful: you can scale, impute missing values, select features, combine predictors together, and so on (see the User Guide.) But there are times when one has an idea that is hard to express with the defaults. For this one needs to know how to create a custom estimator. See scikit-learn's own tutorial. We can start from the following useful but simple example, which I call Tap:

```
class Tap(BaseEstimator, TransformerMixin):
    """debugger"""

    def __init__(self) -> None:
        pass

    def fit(self, X: pd.DataFrame, y=None):
        self.X_ = X.copy()
        return self

    def transform(self, X):
        return X
```

Essentially, we always inherit from BaseEstimator (which defines .get_params and .set_params). Adding the TransformerMixin defines[6] fit_transform, given that fit and transform are defined.

The only point of this class is so to save the DataFrame passed to it so that it can be inspected later. This helps with the development of other estimators and understanding your model.

Tap doesn't need any parameters so the initialiser is empty. For more complicated estimators, I first quote from scikit-learn's own tutorial an important point for interop with the scikit-learn estimators:

> The object's __init__ method might accept constants as arguments that determine the estimator's behavior (like the C constant in SVMs). It should not, however, take the actual training data as an argument, as this is left to the fit() method:
>
> ```
> clf2 = SVC(C=2.3)
> clf3 = SVC([[1, 2], [2, 3]], [-1, 1]) # WRONG!
> ```
>
> The arguments accepted by __init__ should all be keyword arguments with a default value. In other words, a user should be able to instantiate an estimator without passing any arguments to it. The arguments should all correspond to hyperparameters describing the model or the optimisation problem the estimator tries to solve. These initial arguments (or parameters) are always remembered by the estimator. Also note that they should not be documented under the "Attributes" section, but rather under the "Parameters" section for that estimator.

[6] it also defines set_output for Pandas, but the global setting is enough.

> In addition, every keyword argument accepted by `__init__` should correspond to an attribute on the instance. Scikit-learn relies on this to find the relevant attributes to set on an estimator when doing model selection.
>
> To summarize, an `__init__` should look like:

```
def __init__(self, param1=1, param2=2):
    self.param1 = param1
    self.param2 = param2
```

> There should be no logic, not even input validation, and the parameters should not be changed. The corresponding logic should be put where the parameters are used, typically in fit.
>
> [...]
>
> The reason for postponing the validation is that the same validation would have to be performed in `set_params`, which is used in algorithms like `GridSearchCV`.

Notably, the above convention is at odds with the usual Python conventions. With that out of the way, I present my `ColumnSubset` meta-estimator, which allows you to specify a column name, a list of names, or a function that transforms `X.columns` into the required list of column names, and then apply a transformer only to those columns. This *can* be done in the simpler cases with `FeatureUnion` or `ColumnTransformer` which come with `scikit-learn`, but I didn't like how `ColumnTransformer` changed the names of my columns, and I wanted more flexibility in choosing the columns.

```
Estimator = Pipeline # just for type hinting
def column_subset(
    X: pd.DataFrame,
    columns: str | list[str] | Callable | None = None,
    ignore_columns: str | list[str] | Callable | None = None,
):
    if isinstance(columns, str):
        out = [columns]
    elif isinstance(columns, list):
        out = columns
    elif callable(columns):
        out = columns(X.columns)
    elif columns is None:
        out = X.columns
    else:
        raise TypeError(f"Invalid type for columns={columns}")

    if isinstance(ignore_columns, str):
        out = [c for c in out if c != ignore_columns]
    elif isinstance(ignore_columns, list):
        out = [c for c in out if c not in ignore_columns]
    elif callable(ignore_columns):
```

```
        out = [c for c in out if c not in ignore_columns(X.columns)]
    elif ignore_columns is None:
        pass
    else:
        raise TypeError(f"Invalid type for
        ↪  ignore_columns={ignore_columns}")

    return (out, [c for c in X.columns if c not in out])


class ColumnSubset(BaseEstimator, TransformerMixin):
    def __init__(
        self,
        estimator: Estimator,
        columns: str | list[str] | Callable | None = None,
        ignore_columns: str | list[str] | Callable | None = None,
    ) -> None:
        self.estimator = estimator
        self.columns = columns
        self.ignore_columns = ignore_columns

    def fit(self, X: pd.DataFrame, y=None):
        self.cols_, self.other_cols_ = column_subset(
            X, columns=self.columns, ignore_columns=self.ignore_columns
        )
        self.estimator.fit(X[self.cols_], y)
        return self

    def transform(self, X: pd.DataFrame):
        return pd.merge(
            X[self.other_cols_],
            self.estimator.transform(X[self.cols_]),
            left_index=True,
            right_index=True,
```

I also created `ModelTransformer`, for using an (unsupervised) model's predictions to transform my features:

```
class ModelTransformer(BaseEstimator, TransformerMixin):
    """The `ModelTransformer` class is a custom transformer that fits a
    ↪  model on specified independent and
    response columns, and transforms the input data by predicting the
    ↪  response values using the fitted
    model."""

    def __init__(
        self,
```

```
        model: Estimator,
        indep_cols: list[str],
        response_cols: list[str],
    ):
        self.model = model
        self.indep_cols = indep_cols
        self.response_cols = response_cols

    def fit(self, X, y=None):
        self.model.fit(X[self.indep_cols], X[self.response_cols])
        return self

    def transform(self, X: pd.DataFrame):
        pre_out = pd.DataFrame(
            self.model.predict(X[self.indep_cols]),
            columns=[f"MT_{c}" for c in self.response_cols],
            index=X.index,
        )

        return = pd.merge(
            X,
            pre_out,
            left_index=True,
            right_index=True,
        )
```

Finally, I want to share my `ModelSelector`, which switches between predictors based on a categorical variable. This allows you to fit two (or inductively, any number) different models in a single Pipeline.

```
class ModelSelector(BaseEstimator, RegressorMixin):
def __init__(
    self,
    model_0: Estimator,
    model_1: Estimator,
    cat_var: str,
    drop_cat_var: bool = False,
):
    self.model_0 = model_0
    self.model_1 = model_1
    self.cat_var = cat_var
    self.drop_cat_var = drop_cat_var

def fit(self, X: pd.DataFrame, y):
    # split the data based on the value of the categorical variable
    X_0 = X[X[self.cat_var] == 0]
    y_0 = y[X[self.cat_var] == 0]
```

```
        X_1 = X[X[self.cat_var] == 1]
        y_1 = y[X[self.cat_var] == 1]
        if self.drop_cat_var:
            X_0 = X_0.drop(columns=[self.cat_var])
            X_1 = X_1.drop(columns=[self.cat_var])
        # fit the models on the corresponding subsets of data
        self.model_0.fit(X_0, y_0)
        self.model_1.fit(X_1, y_1)
        return self

    def predict(self, X):
        # split the data based on the value of the categorical variable
        X_0 = X[X[self.cat_var] == 0]
        X_1 = X[X[self.cat_var] == 1]
        if self.drop_cat_var:
            X_0 = X_0.drop(columns=[self.cat_var])
            X_1 = X_1.drop(columns=[self.cat_var])
        # predict using the models on the corresponding subsets of data
        y_pred_0 = self.model_0.predict(X_0)
        y_pred_1 = self.model_1.predict(X_1)
        # combine the predictions into a single array
        y_pred = np.empty(len(X))
        y_pred[X[self.cat_var] == 0] = y_pred_0
        y_pred[X[self.cat_var] == 1] = y_pred_1
        return y_pred
```

5 Further reading

I have made other more complicated estimators but they are too specific to the dataset. Hopefully the above examples have helped you learn how to use `scikit-learn` effectively. There are many more examples on the website and the User Guide and the API docs are very helpful.