Notes on and around ISLP

Calvin Khor

Last compiled: November 21, 2023

1 Resources

1. The website for the book: https://www.statlearning.com
2. Python Solutions to ISLR: https://github.com/botlnec/islp
3. Videos accompanying ISLR: https://www.youtube.com/watch?v=LvySJGj-88U&list=PLoROMvodv4rOzrYsAxzQyHb8n_RWNuS1e
4. My errata: https://docs.oracle.com/javase/8/docs/api/java/lang/
5. These notes: https://dotty.epfl.ch/3.0.0/api

2 Statistical Learning

2.1 Definitions and Notation

In this book, usually $x_1, \ldots, x_n \in \mathbb{R}^p$ denotes training data, and x_0 is test (out-sample) data. When its capital letters $X_1, \ldots X_n$ its a random variable. The p denotes the number of predictors (i.e. independent variables/features/variables). The model is abstractly

$$Y = f(X) + \epsilon. \tag{2.1}$$

Y is the response/dependent variable. ϵ is noise inherent to the model: WLOG $\mathbb{E}\,\epsilon = 0$, $\operatorname{Var}\epsilon = 1$, and independent from the predictors (i.e. we cannot predict the error)

We denote the number of observations by n, so that the observations of the predictors are $x_j = (x_{1j}, x_{2j}, \ldots x_{nj})$, for $j = 1, \ldots, p$.

2.2 Parametric and non-parametric models

A parametric model is when the space of solutions can be parameterised by some subset of $\mathbb{R}^n$. A model is non-parametric otherwise; for instance the model may be parameterised over a subset of functions (e.g. cubic splines.)

2.3 Prediction Accuracy

2.4 Model Interpretability

2.5 Test MSE

Variance: ‘the amount by which $\hat{f}$ would change if we estimated it using a different training data set.’ - i.e. variance by treating the training data as random variables

Bias: the error that is introduced by approximating a real-life problem, which may be extremely complicated, by a much simpler model.

$$E|y_0 - \hat{f}(x_0)|^2 = \mathrm{Var}\,\hat{f}(x_0) + (\mathrm{Bias}\,\hat{f}(x_0))^2 + \mathrm{Var}\,\epsilon \tag{2.7}$$

Important quote from book:

> Here the notation $\mathbb{E}\,|y_0 - \hat{f}(x_0)|^2$ defines the expected test MSE at x_0, expected test MSE and refers to the average test MSE that we would obtain if we repeatedly estimated f using a large number of training sets, and tested each at x_0. The overall expected test MSE can be computed by averaging $\mathbb{E}\,|y_0 - \hat{f}(x_0)|^2$ over all possible values of x_0 in the test set.

In other words, the general process is as follows:

$$\boxed{\text{gather data}}$$

$$x_1, \ldots, x_n;\ y_1, \ldots, y_n$$

$$\downarrow$$

$$\boxed{\text{model choice}}$$

$$Y = f(X) + \epsilon$$

$$\downarrow$$

$$\boxed{\text{choose } \hat{f} \text{ using data}}$$

$$\downarrow$$

$$\boxed{\text{evaluate at out-sample(s)}}$$

$$\hat{y} = \hat{f}(x_0)$$

When we use $\hat{f}(x_0)$ in discussions of theoretical probability like the above say $\mathbb{E}\,g(\hat{f}(x_0), y_0)$, we mean to analyse the above process in the following manner: the expectations are always taken by assuming the data $x_1, \ldots, x_n$ are replaced by random variables $X_1, \ldots, X_n$. The dependence on these random variables can be written explicitly as $\hat{f}(x_0) = \hat{f}(X_1, \ldots, X_n; x_0)$, with the test data (x_0, y_0) kept non-random. Then the expected value at a particular test data is

$$\mathbb{E}\,g(\hat{f}(x_0), y_0) = \mathbb{E}_{X_1,\ldots,X_n}\, g(\hat{f}(X_1, \ldots, X_n; x_0), y_0)$$

One can from this model get the ‘overall expected test MSE’ by further averaging over all (x_0, y_0)s in some way.

Derivation of (2.7):

$$\begin{aligned}
\text{LHS}(2.7) &= \mathbb{E}\,|f(x_0) - \hat{f}(x_0) + \epsilon|^2 \\
&= \mathbb{E}\,|f(x_0) - \mathbb{E}\,\hat{f}(x_0) + (\mathbb{E}\,\hat{f}(x_0) - \hat{f}(x_0)) + \epsilon|^2 \\
&= \mathbb{E}\,|f(x_0) - \mathbb{E}\,\hat{f}(x_0)|^2 + \mathbb{E}\,|\epsilon|^2 + \mathbb{E}\,|\hat{f}(x_0) - \mathbb{E}\,\hat{f}(x_0)|^2 + 2\,\mathbb{E}\,\epsilon(f(x_0) - \hat{f}(x_0)) \\
&= |f(x_0) - \mathbb{E}\,\hat{f}(x_0)|^2 + \mathbb{E}\,|\epsilon|^2 + \mathbb{E}\,|\hat{f}(x_0) - \mathbb{E}\,\hat{f}(x_0)|^2 + 2\,\mathbb{E}\,\epsilon(f(x_0) - \hat{f}(x_0)) \\
&=: \text{Bias}^2 + \text{Var}\,\epsilon + \text{Var}\,\hat{f}(x_0) + \underbrace{0}_{\text{since } \epsilon \text{ is independent}} = \text{RHS}(2.7).
\end{aligned}$$

2.6 Bias-Variance Trade-off

3 Linear Regression

3.1 Simple Linear Regression

Model:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

in particular then $\mathbb{E}[Y|X] = \beta_0 + \beta_1 X$. We say "$Y$ regresses on X" to indicate that Y is the dependent variable that depends on the independent variable X.

Asides: assumptions on ϵ affects the statistical results one can prove. Ways to work around somewhat: quasi maximum likelihood ("departure from Gaussianity". Note Student's t distribution has infinite variance if $nu < 2$ and infinite skewness if $\nu < 3$. Kolmogorov-Smirnov test, Kurtosis https://en.wikipedia.org/wiki/Kurtosis

Prediction = find β_i so that

$$\hat{y} := \hat{\beta}_0 + \hat{\beta}_1 X$$

minimises the loss. The estimation error at x is

$$e \equiv y - \hat{y} = y - (\hat{\beta}_0 + \hat{\beta}_1 x)$$

which may be different from ϵ. OLS = minimising residual sum of squares, RSS $= \sum_{i=1}^n e_i^2$. RSS defines your "risk tolerance"; there are different loss functions.

3.1.1 Estimating the Coefficients

$$\hat{\beta}_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \quad (\frac{\text{sample covariance}}{\text{sample variance of } x}), \qquad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

3.1.2 Assessing the Accuracy of the Coefficient Estimates

This subsection deals with CE and hypothesis testing to check if there is a (linear) relationship between the data.

The standard error of a statistic (usually an estimate of a parameter) is the standard deviation of its sampling distribution or an estimate of that standard deviation. If the statistic is the sample mean, it is called the standard error of the mean (SEM). The standard error is a key ingredient in producing confidence intervals.

The formula for the (approximate) 95% CE of X when X is Gaussian is $\hat{X} \pm 2\,\mathrm{SE}(\hat{X})$. For CEs with different α it would be $\hat{X} \pm z_{\alpha/2}\,\mathrm{SE}(\hat{X})$. where z_θ is a Z-statistic.

In the case of the mean The theorem that the true value lies in the constructed α-confidence interval (CE) α% of the time assumes that the error term is Gaussian. Similarly for hypothesis testing.

Hypothesis test if there is a (linear) relationship between Y and X: WLOG subtracting the mean, we simply need to test if $\beta_0 = 0$ (the null) or not (the alternative). Accomplished with t-statistic $t = \frac{\hat{\beta}_1 - 0}{\mathrm{SE}(\hat{\beta}_1)} \sim t_{n-2}$. Then compute a p-value: the probability of getting an observation at least as large as the data, under the null.

There is a correspondence: We reject the null that $\beta_1 = 0$ iff the confidence interval constructed for β_1 does not contain 0.

3.1.3 Assessing the Accuracy of the Model

After concluding that the model fits the data, we want to quantify how much the model fits.

There is an issue when having too many data points, since the true value of β_0 is unlikely to be exactly zero in the many observations limit: to avoid this one can use the Akaike Information Criterion, Bayesian Info Criterion: good to use when too many data points so the hypothesis test is statistically significant but not practically significant. Also adjusted R^2. "Avoids overfitting". RSE is the Residual Standard Error, and R^2 is the fraction of variance explained:

$$\mathrm{RSE} := \sqrt{\frac{\mathrm{RSS}}{n-2}}, \quad R^2 := \frac{\mathrm{TSS} - \mathrm{RSS}}{\mathrm{TSS}} = 1 - \frac{\mathrm{RSS}}{\mathrm{TSS}} = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} = 1 - \frac{|y - \hat{y}|^2}{|y - \bar{y}\mathbf{1}|^2}$$

Note: since OLS is L^2 orthogonal projection (after subtracting off the mean),

$$|y - \bar{y}\mathbf{1}|^2 = |y - \hat{y}|^2 + |\hat{y} - \bar{y}\mathbf{1}|^2$$

In simple linear regression (1 predictor 1 response), $R^2 = r^2$ where r is the Pearson correlation coefficient:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}}.$$

Proof (see Exercise 7, solution online): without loss $\bar{y} = 0, \bar{x} = 0$. Then

$$R^2 = 1 - \frac{|y - \hat{y}|^2}{|y|^2} = \frac{|\hat{y}|^2}{|y|^2} = \frac{|(X^T X)^{-1} X^T y|^2}{|y|^2}$$

For simple linear regression, $X \in \mathbb{R}^{n \times 1}$ is a column vector so $X^T X = |X|^2$ and $X^T y = X \cdot y$. Hence $R^2 = \left|\frac{X \cdot y}{|X||y|}\right|^2 = \left|\frac{s_{xy}}{s_x s_y}\right|^2$.

3.2 Multiple Linear Regression

Hypothesis testing. $H_0 : \beta = (\beta_0, \ldots, \beta_n) = 0$. H_1: there exists one nonzero coefficient. Instead of t-statistic, use F-statistic. $y = X\beta + \epsilon$. Hard to interpret coefficients. Model selection. Dummy variables. Nonlinearities.

3.3 Other things that came up in class

- https://en.wikipedia.org/wiki/Projection_matrix (i.e. the hat matrix that puts a hat on y, $\hat{y} = X(X^TX)^{-1}X^Ty$.

- Random math bit: Let $f(x) = (1-x^4)^{1/7}$. Observe that on $[0,1]$, f is monotone decreasing with inverse $g(x) = (1-x^7)^{1/4}$. Then if μ denotes standard Lebesgue measure, we have

  $$\begin{aligned}\int_0^1 (1-x^4)^{1/7}\,\mathrm{d}x &= \int_0^1 f(x)\,\mathrm{d}x = \int_0^1\int_0^{f(x)} \mathrm{d}t\,\mathrm{d}x = \int_0^1 \mu(x\in[0,1] : t < f(x))\,\mathrm{d}t = \int_0^1 \mu(x\in[0,1] : x < g(t))\,\mathrm{d}t = \int_0^1 g(t)\,\mathrm{d}t \\ &= \int_0^1 (1-x^7)^{1/4}\,\mathrm{d}x\end{aligned}$$

  PS $d_f(t) := \mu(x : f(x) > t)$ is the *distribution function* of f.

- Joy's question (difference between interaction features and fitting separate models) Say we have a problem we're modelling as $y = X\beta + \epsilon \quad (\in \mathbb{R})$. We can then e.g. choose to double the number features by elementwise multiplying by appropriate indicators: i.e. if $X^C = X\star.\mathbf{1}_C$ for some condition C (forgive my abuse $(\mathbf{1}_C)_i := \mathbf{1}_{i\in C}$ for $i = 1,\dots,n$) then $X = X^C + X^{C^c}$. To fix notation, WLOG set $C = [1,c]$, $X^C = \binom{A}{0}$ and $X^{C^c} = \binom{0}{B}$ where $A \in \mathbb{R}^{c\times p}$ and $B \in \mathbb{R}^{d\times p}$, $c + d = n$ (so that $\tilde{X} = \begin{pmatrix} A & 0 \\ 0 & B\end{pmatrix} \in \mathbb{R}^{(c+d)\times 2p}$ is the new features matrix). Then the $2p$ predictors model can be written

  $$\begin{pmatrix} y^1 \\ y^2 \end{pmatrix} = y = \tilde{X}\tilde{\beta} + \epsilon = A\beta^1 + B\beta^2 + \epsilon.$$

  OLS will find the unique $\tilde{\beta} = \binom{\beta^1}{\beta^2}$ that minimises

  $$\begin{aligned}\sum_{i=1}^{n}\left|y_i - \sum_{j=1}^{2p}\tilde{X}_{ij}\tilde{\beta}_j\right|^2 &= \sum_{i\in C}\left|y_i - \sum_{j=1}^{2p}\tilde{X}_{ij}\tilde{\beta}_j\right|^2 + \sum_{i\notin C}\left|y_i - \sum_{j=1}^{2p}\tilde{X}_{ij}\tilde{\beta}_j\right|^2 \qquad (1)\\ &= \sum_{i\in C}\left|y_i - \sum_{j=1}^{p}X^C_{ij}\beta^1_j\right|^2 + \sum_{i\notin C}\left|y_i - \sum_{j=1}^{p}X^{C^c}_{ij}\beta^2_j\right|^2. \qquad (2)\end{aligned}$$

  Since β^i are independently chosen this solution is (uniquely) given by the minimisers of each of the two summands. We can also relate this to smaller models that are fit separately on the $i \in C$ and $i \notin C$ data:

  $$\sum_{i=1}^{n}\left|y_i - \sum_{j=1}^{2p}\tilde{X}_{ij}\tilde{\beta}_j\right|^2 = \sum_{i=1}^{c}\left|y^1_i - \sum_{j=1}^{p}A_{ij}\beta^1_j\right|^2 + \sum_{i=1}^{d}\left|y^2_i - \sum_{j=1}^{p}B_{ij}\beta^2_j\right|^2. \tag{3}$$

  One can plainly read off the two summands as the loss functions for the separate models. And just for completeness, the OLS solution for the $2p$ predictor model is

  $$\begin{aligned}\hat{\tilde{\beta}} &= (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^Ty \qquad (4)\\ &= \begin{pmatrix} (A^TA)^{-1} & 0 \\ 0 & (B^TB)^{-1}\end{pmatrix}\begin{pmatrix} A^T & 0 \\ 0 & B^T\end{pmatrix}\begin{pmatrix} y^1 \\ y^2\end{pmatrix} = \begin{pmatrix} (A^TA)^{-1}A^Ty^1 \\ (B^TB)^{-1}B^Ty^2\end{pmatrix}. \qquad (5)\end{aligned}$$

4 Classification

4.1 Overview

4.2 Why not Linear Regression

4.3 Logistic Regression

4.3.1 Multiple Logistic Regression

4.3.2 Multinomial Logistic Regression

4.4 Generative Models for CLassification

4.4.1 Linear Discriminant Analysis

4.4.2 Quadratic Discriminant Analysis

4.4.3 Naive Bayes

4.5 Comparison to KNN

4.6 Generalised Linear Models

5

6

7

8 Tree-based Methods

9

10

11 Quick notes on using scikit-learn and Pandas

Structure: setup. Basic usage. Estimators, Transformers, Regressors, Classifiers, Pipelines. GridsearchCV, Random. Halving. Custom Estimators. Typing.

11.1 Introduction

While I was trying to learn these tools for data analysis, I found the available discussion online to be dated. 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.

11.1.1 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")
```

11.2 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.

11.2.1 Pipeline

A a Pipeline is a way to combine estimators and predictors in a way that is easy to modify and develop. Documentation on Pipelines 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 the convenience method fit_transform:

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

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

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 `FunctionTransformers`.) A simple example is if you wanted to drop a column called `"Rubbish"`, you could use[2]

```
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.[3] `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.

```
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
```

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

[3] This includes the intercept term by default.

```python
    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)
```

11.2.2 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:

```python
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`:

```python
# 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[4] is a name that can be used to inspect parts of the pipe:

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

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

11.3 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 = {
    "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:

```python
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.

11.4 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:

```python
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[5] `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:
>
> ```python
> 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.
>
> 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:
>
> ```python
> 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.
>
> […]

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

> 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.

```python
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(
```

```python
            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.

```python
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
```

11.5 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.

12 Python

12.1 Conventions and patterns

- Subscript at the end of a variable name means we are looking at a coefficient or other computed quantity of an estimator.

- Further to the above, see how to make custom estimators on conventions in the init and other functions (they differ from usual Python classes

- Don't use `inplace=True`.

- Saving a model using Joblib:

  ```
  import joblib
  # saving; the filetype extension is only for the user/reader
  joblib.dump(pipe, 'filename.joblib')

  # loading
  loaded = joblib.load('filename.joblib')
  ```

  Make sure you use the same environment (module version etc.) or the model may change.

- Pipelines are DataFrame-friendly[6], if the component transformers are as well. If they are not, we can wrap the transformer so that it returns a DataFrame. For instance, `StandardScaler` which normally returns a `np.array`:

[6] This and the next bullet point is from `https://www.youtube.com/watch?v=BFaadIqWlAg&list=PLzERW_Obpmv_t55kNFRet-E0h1nKeswWF&index=26`, with Github repo at `https://github.com/jem1031/pandas-pipelines-custom-transformers`.

```
class DFStandardScaler(TransformerMixin):
    def __init__(self):
        self.ss = None
    def fit(self, X, y=None):
        self.ss = StandardScaler().fit(X)
        return self
    def transform(self, X):
        Xss = self.ss.transform(X)
        Xscaled = pd.DataFrame(Xss, index=X.index, columns=X.columns)
        return Xscaled
```

- Pipelines compose[7], e.g.:

```
pipeline = Pipeline([
    ('features', DFFeatureUnion([
        ('categoricals', Pipeline([
            ('extract', ColumnExtractor(CAT_FEATS)),
            ('dummy', DummyTransformer())
        ])),
        ('numerics', Pipeline([
            ('extract', ColumnExtractor(NUM_FEATS)),
            ('zero_fill', ZeroFillTransformer()),
            ('log', Log1pTransformer())
        ]))
    ])),
    ('scale', DFStandardScaler())
])
```

12.2 Custom Scikit-learn classes

From https://www.youtube.com/watch?v=WGirN6zBJ4s&list=PLzERW_Obpmv_t55kNFRet-E0h1nKeswWF&index=1. There are `Estimator`, `Predictor`, `Transformer`, and `Model` classes. An `Estimator` must implement

- `.fit(X,y)`, fitting the estimator to `X` and `y`
- `.get_params()` return the parameters of the estimator
- `.set_params(**params)` change the parameters of the estimator (e.g. for copying)

Sklearn's `Predictor` needs

- `.predict(X)`

Sklearn's `Transformer` needs

- `.transform(X, y=None)`

Sklearn's `Model` needs

- `.score(X,y)`

[7] see also http://zacstewart.com/2014/08/05/pipelines-of-featureunions-of-pipelines.html

12.2.1 Inheritance and Mixins

- You can implement `.get_params()` and `.set_params()` by inheriting from `BaseEstimator`
- There is also `base.TransformerMixin`, `base.RegressorMixin`, `base.ClassifierMixin`, `base.ClusterMixin`, `feature_selection.SelectorMixin`,...

12.2.2 Example 1: Scaler

```
import numpy as np
from sklearn.base import BaseEstimator, TransformerMixin

class Standardizer(BaseEstimator, TransformerMixin):

def __init__(self,mean_after_transform = 0):
    self.mean_after_transform = mean_after_transform

def fit(self, X, y=None):
    self.mean_ = np.mean(X, axis=0) # columwise mean
    self.std_ = np.std(X, axis=0) # columwise std
    return self

def transform(self, X):
    return (X-self.mean_) / self.std_ + self.mean_after_transform
```

12.2.3 Example 2: Regression

Basic regressor that just predicts using the mean or median, using `RegressorMixin` (A regressor is a type of `Model`, so it needs a predict. The mixin gives us a `.score` for free):

```
import numpy as np
class MyDummyRegression(BaseEstimator, RegressorMixin):

def __init__(self, use_median=False):
    self.use_median = use_median

def fit(self, X, y):
    if self.use_median:
        self.value_ = np.median(y)
    else:
        self.value_ = np.mean(y)
    return self

def predict(self, X):
    out = np.empty (len(X))
    out.fill(self.value_)
    return out
```

12.3 FeatureUnion, FunctionTransformer, ColumnTransformer

ColumnTransformer video: https://www.youtube.com/watch?v=to2mukSyvLk&list=PLzERW_Obpmv_t55kNFRet-E0h1nKeswWF&index=22

12.4 Keep the index when defining transform

Example:

12.5 Don't assign multiple times: concat instead

This holds for `df.assign(key=col)`, `df1.merge(df2)`, and `df[key]=col`.