The Orange Book of Machine Learning

The essentials of making predictions using supervised regression and classification for tabular data

Carl McBride Ellis

ORCiD: 0000-0003-2966-7530

email: carl.mcbride@protonmail.ch
LinkedIn: www.linkedin.com/in/carl-mcbride-ellis
Book GitHub: github.com/Carl-McBride-Ellis/TOBoML

First 'Orange' edition, June 2024
(version 1.0.1)

Contents

This book is based on my five day course which I had the pleasure of teaching in the following Spanish cities: A Coruña, Algeciras, Alicante, Bilbao, Cáceres, Granada, Huesca, Jaén, Madrid, Málaga, Murcia, Sevilla, Valencia, Valladolid, and Zaragoza. I would like to take this opportunity to say a big thank you to all of my students *¡un gran placer!*

This book uses the `python` programming language largely in conjunction with the `scikit-learn` machine learning library, and `pandas` for data manipulation. All example notebooks use the Jupyter Notebook environment.

Special thanks to: Carlos Ortega, Santiago Mota Herce, Norberto Malpica, Charles H. Martin, Roland Stevenson, Adrian Olszewski, Valeriy Manokhin, and everyone on Kaggle.

For Cristina

1. Introduction

> Without data, you're just another person with an opinion.
>
> *William Edwards Deming*

1.1 The $\hat{\beta}$ and the $\hat{y}$

A statistical model, designed for explanation or inference, will use a simple predefined function (based on the assumption that the data can be modelled by a parametric distribution) and will then seek the best parameters, $\hat{\beta}$, for said function. A non-parametric (*i.e.* making no assumptions about the distribution of the data) machine learning model forgoes explanatory power for predictive power, with the emphasis on producing the best predictions, $\hat{y}$.

Equation: Fundamental assumption: our data is composed of signal + noise

$$y = f(x) + \varepsilon \tag{1.1}$$

where $f(x)$ is the *ground truth* and ε is the *irreducible error*.

In multiple dimensions $f(x)$ will form a surface, and our objective is to produce the best possible fit to this surface. This short book is dedicated to introducing us to the essentials of a number of techniques for producing such a fit: this book is all about the $\hat{y}$.

1.2 Interpolation and curve fitting

Interpolation passes through every single data point, whereas curve fitting does not have to do this. Indeed to some great extent machine learning can be thought of as little more than glorified cure fitting. Ideally we fit a curve to all of the signal, and to none of the noise. If it were not for noise, often leading to y being multivalued, then interpolation would have done the job just fine from a $\hat{y}$ point of view (but would be useless from a $\hat{\beta}$ viewpoint).

1.3 Errors and residuals

The error is the difference between the observed value and the ideal value. For example, a chocolate bar may have the ideal value printed on the packet, say 100g. However, no chocolate bar has ever been made that weights *exactly* 100g, and the difference between the actual weight and ideal 100g is the error. On the other hand the residual is the difference between the predicted value and the observed value. Note however that in this text when we use the word error correctly speaking we should really use the word residual, and we do this simply because of the prevalence of the word error in the machine learning literature, for example the mean absolute error (MAE) or the root mean square error (RMSE).

1.4 Sources of uncertainty: aleatoric and epistemic

Aleatoric, or stochastic, uncertainty is due to inherent random noise in a system/generating process. Epistemic uncertainty is due to a lack of data.

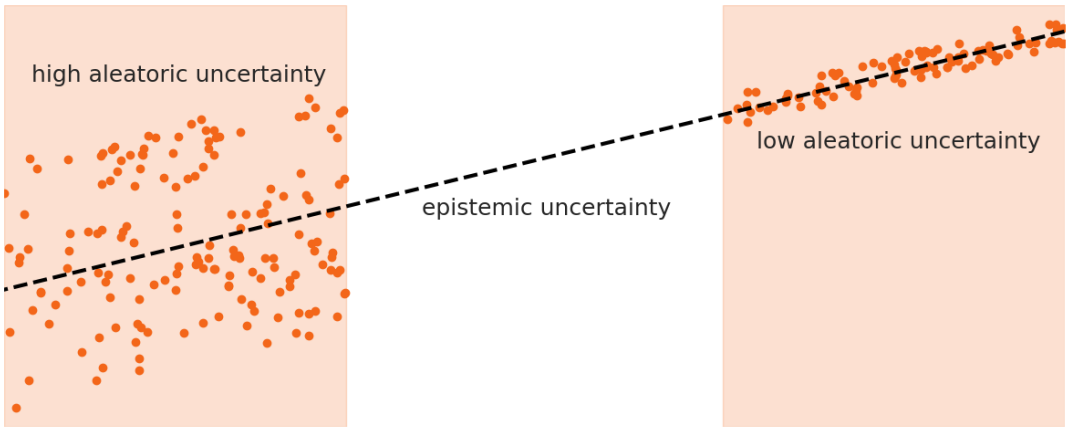


Figure 1.1: Aleatoric and epistemic uncertainty in a dataset.

1.5 Confidence and prediction intervals

Confidence intervals quantify the uncertainty in the parameters β. Prediction intervals quantify the uncertainty in the predictions $\hat{y}$. We shall see how to calculate regression prediction intervals using conformal prediction in Section 6.13.

1.6 Explainability and interpretability

- Explainability: the ability to understand both the 'how' and the 'why' of a prediction
- Interpretability: the ability of understand the 'why' of a prediction

Explainability is the *forte* of statistical models; the 'how' is in-built from the very start by using simple models for the data. On the other hand the focus of machine learning (ML) is almost exclusively on prediction performance. Only the simplest ML models are interpretable; decisions trees are in principle very easy to explain, but even at depth 3 (8 leaves) and having several features then in reality they quickly become hard to interpret.

There are packages such as SHAP (SHapley Additive exPlanations) and LIME (Local Interpretable Model-Agnostic Explanations) to facilitate interpretability. However, it has

been seen that often even the data scientists themselves do not know how to use these packages properly[1]:

> *"...results indicate that data scientists over-trust and misuse interpretability tools. Furthermore, few of our participants were able to accurately describe the visualizations output by these tools".*

which does not bode well. To make matters worse of late some of the uses that people make of the SHAP technique, such as posing counterfactual questions, have been brought into question[2,3]. At the end of the day machine learning models are designed for performance, and generally interpretability takes a back seat. However, in many circumstances model interpretability is of paramount importance; indeed under the EU General Data Protection Regulation (GDPR) 2016/679 (Article 15 1h)

> "*The data subject shall have the right to obtain... the following information: the existence of automated decision-making... meaningful information about the logic involved, as well as the significance and the envisaged consequences of such processing for the data subject.*"

With the ever increasing rôle that machine learning models play in modern society the creation of interpretability tools is an active area of development.

[1] Kaur *et al.* *"Interpreting Interpretability: Understanding Data Scientists' Use of Interpretability Tools for Machine Learning"*, Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems pp. 1-14 (2020)

[2] Huang, Marques-Silva *"The Inadequacy of Shapley Values for Explainability"*, arXiv:2302.08160 (2023)

[3] Bilodeau, Jaques, Wei Koh, Kim *"Impossibility Theorems for Feature Attribution"*, arXiv:2212.11870 (2024)

Recommended reading

Papers

- Leo Breiman *"Statistical Modeling: The Two Cultures"*, Statistical Science **16** pp. 199-231 (2001)
- Galit Shmueli *"To Explain or to Predict?"*, Statistical Science **25** pp. 289-310 (2010)
- Gruber *et al.* *"Sources of Uncertainty in Machine Learning - A Statisticians' View"*, arXiv:2305.16703 (2023)

Books

- Denis Rothman *"Hands-On Explainable AI (XAI) with Python"*, Packt Publishing Limited (2020)
- Serg Masís *"Interpretable Machine Learning with Python"*, Packt Publishing Limited (2023)
- Christoph Molnar *"Interpretable Machine Learning"*, (online book)

Packages

- SHAP
- LIME

2. Statistics

60% of the time, it works every time

Brian Fantana in 'Anchorman'

In this chapter we briefly cover some essential statistical concepts that are very useful when it comes to understanding our data and machine learning estimators.

2.1 Centrality: Mean, median, and mode

The mean and the median are measures of the central tendency of a distribution.

Equation: The arithmetic mean of a sample is given by

$$\bar{y} := \frac{1}{n}\sum_{i=1}^{n} y_i \tag{2.1}$$

The mean of the hypothetical population, from which the sample was supposedly obtained, is denoted by μ.

The median ($\tilde{y}$) of a set of values is a number such that half of the values are below the median value, and the other half are above the median value. The median is said to be a *robust* statistic in that it is less influenced by a few extreme or outlier values. For example here we calculate the mean and the median for an *array* of values using the python numpy library

```
import numpy as np

np.mean([1,2,3,4,5,6,1001])
146.0

np.median([1,2,3,4,5,6,1001])
4.0
```

The mode is the most frequently occurring value, and can be useful when working with a discrete distribution (such as count data), categorical features, or for quickly identifying a zero-inflated distribution.

2.2 Dispersion: Variance, MAD, and quartiles

In Figure 2.1 we have two distributions that have exactly the same mean value.

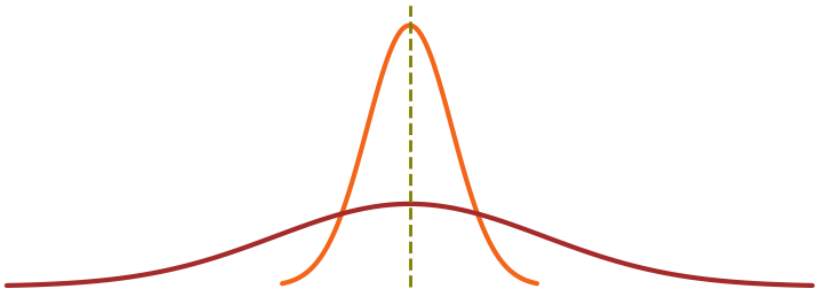


Figure 2.1: Two distributions with same central tendency.

We can see that centrality only tells part of the story and evidently we also need to be able to describe how spread out a distribution is as well.

Equation: The variance, σ^2, is given by

$$\sigma^2(y) := \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2 \tag{2.2}$$

and σ is known as the standard deviation.

Remark: When calculating the variance or standard deviation by default, unlike `numpy`, `pandas` applies the Bessel correction of $n-1$.

Equation: The sample covariance, between the ordered pair x and y, is given by

$$cov(x,y) = \frac{1}{(n-1)}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \tag{2.3}$$

In 2-dimensions we can now also construct a variance-covariance matrix:

$$K = \begin{bmatrix} \sigma^2(x) & cov(x,y) \\ cov(y,x) & \sigma^2(y) \end{bmatrix} \tag{2.4}$$

(note that $cov(x,y) = cov(y,x)$ so this matrix is symmetric).

The median absolute deviation is the robust statistic for dispersion.

Equation: The median absolute deviation (MAD) is given by

$$\text{MAD}(y) = \text{median}(|y_i - \tilde{y}|) \tag{2.5}$$

Remark: For some reason the `pandas mad` is actually the **mean** absolute deviation and not the median absolute deviation.

2.2.1 Quantiles, quartiles and the interquartile range (IQR)

The median is a special case of a quantile, and is denoted as Q2. Two other quantiles of note are Q1, below which lies 25% of the values, and Q3, above which also lies 25% of the data. Suffice to say the other 50% of the data has values between Q1 and Q3 and is known as the interquartile range (IQR).

```
import numpy as np

values = [1,2,3,4,5,6,7,8,9,10,11,12]
np.quantile(values, 0.25) # Q1
3.75
np.quantile(values, 0.5)  # Q2
6.5
np.quantile(values, 0.75) # Q3
9.25
np.quantile(values, 0.75) - np.quantile(values, 0.25) # IQR
5.5
```

Quantiles are excellent for producing prediction intervals as we shall see later in quantile regression.

2.3 Gaussian distribution: additive

Equation: The Gaussian probability density function (PDF) is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{2.6}$$

where μ is the mean and σ is the standard deviation.

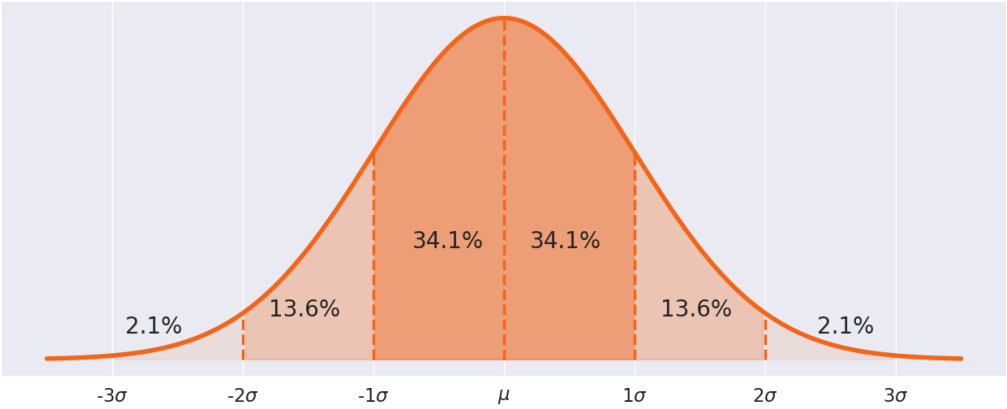


Figure 2.2: The Gaussian distribution

This leads to the easy to remember 68–95–99.7 rule.

Example

To calculate the percentage of data below 1σ we use the cumulative distribution function (CDF). This corresponds to the probability that a random sample will lie $\leq 1\sigma$.

```
import scipy.stats

mu = 0
sigma = 1
x = 1*sigma

scipy.stats.norm.cdf(x,mu,sigma)

0.8413447460685429
```

answer: $\approx 84\%$.
A Gaussian distribution can approximately be obtained as a result of a large number of independent additive samples.

N **Jupyter Notebook:** Animated histogram of the central limit theorem

2.3.1 Tests for normality

There are several statistical tests for normality[1]. Null hypothesis (H_0): the sample is from a Gaussian population

- Shapiro-Wilk
- D'Agostino
- Anderson-Darling

[1] B. W. Yap, C. H. Sim *"Comparisons of various types of normality tests"*, Journal of Statistical Computation and Simulation **81** pp. 2141-2155 (2011)

2.4 Chebyshev's inequality

Suffice to say that our data almost never has a Gaussian distribution. The Chebyshev inequality applies to virtually any distribution and of note is that in general at least 75% of the values lie within $\pm 2\sigma$ of the mean, and at least 89% lie within $\pm 3\sigma$ of the mean.

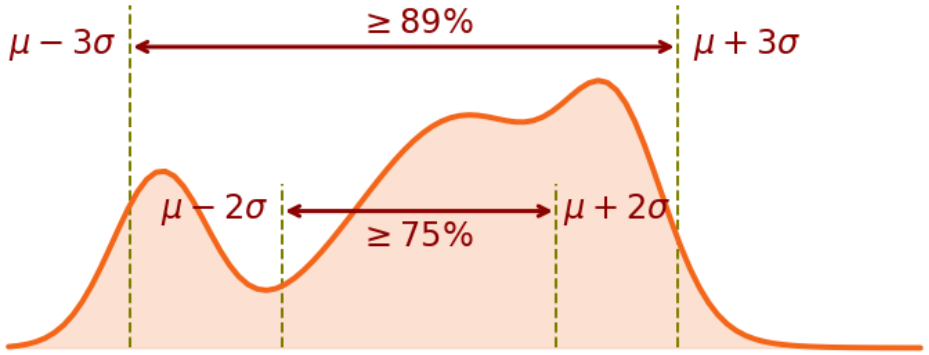


Figure 2.3: Chebyshev's inequality.

2.5 Galton distribution: multiplicative

The Galton distribution, or log-normal distribution is frequently seen in real datasets[2].

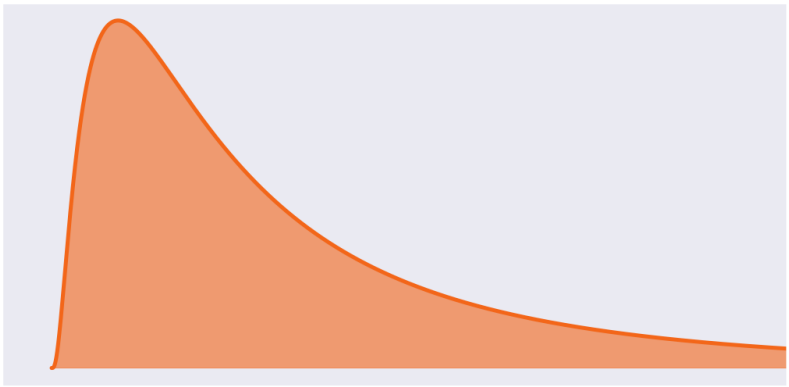


Figure 2.4: Galton or log-normal distribution

If one were to take the logarithm of the independent variable one would obtain a Gaussian distribution. A Galton distribution can approximately be obtained as a result of a large number of independent multiplicative samples (see also Gibrat's law).

N **Jupyter Notebook:** Simulating a Galton distribution

2.6 Skewness and kurtosis

Skewness is a measure of distribution asymmetry. The Galton distribution is an example of a distribution having positive skew. Unlike the symmetric Gaussian distribution with

[2] Eckhard Limpert, Werner A. Stahel, Markus Abbt *"Log-normal Distributions across the Sciences: Keys and Clues"*, BioScience **51** pp. 341-352 (2001)

similar tails on either side, here the long tail is located on the right hand side (RHS). For negative skew the long tail is on the left hand side (LHS). One can test for skewness using the `scipy.stats.skewtest`.

> **Remark:** Be wary of detecting skewness simply in terms of the relative locations of the mean and the median, especially for discrete distributions often seen in real data.

Kurtosis is a measure of tail thickness (and not peakedness[3]):

- leptokurtic ($Kurt - 3 > 0$): a distribution that has a fatter tail than a Gaussian
- mesokurtic ($Kurt - 3 = 0$): Gaussian-like
- platykurtic ($Kurt - 3 < 0$): a distribution that has a thinner tail than a Gaussian

One can test for kurtosis using `scipy.stats.kurtosis`.

Recommended reading

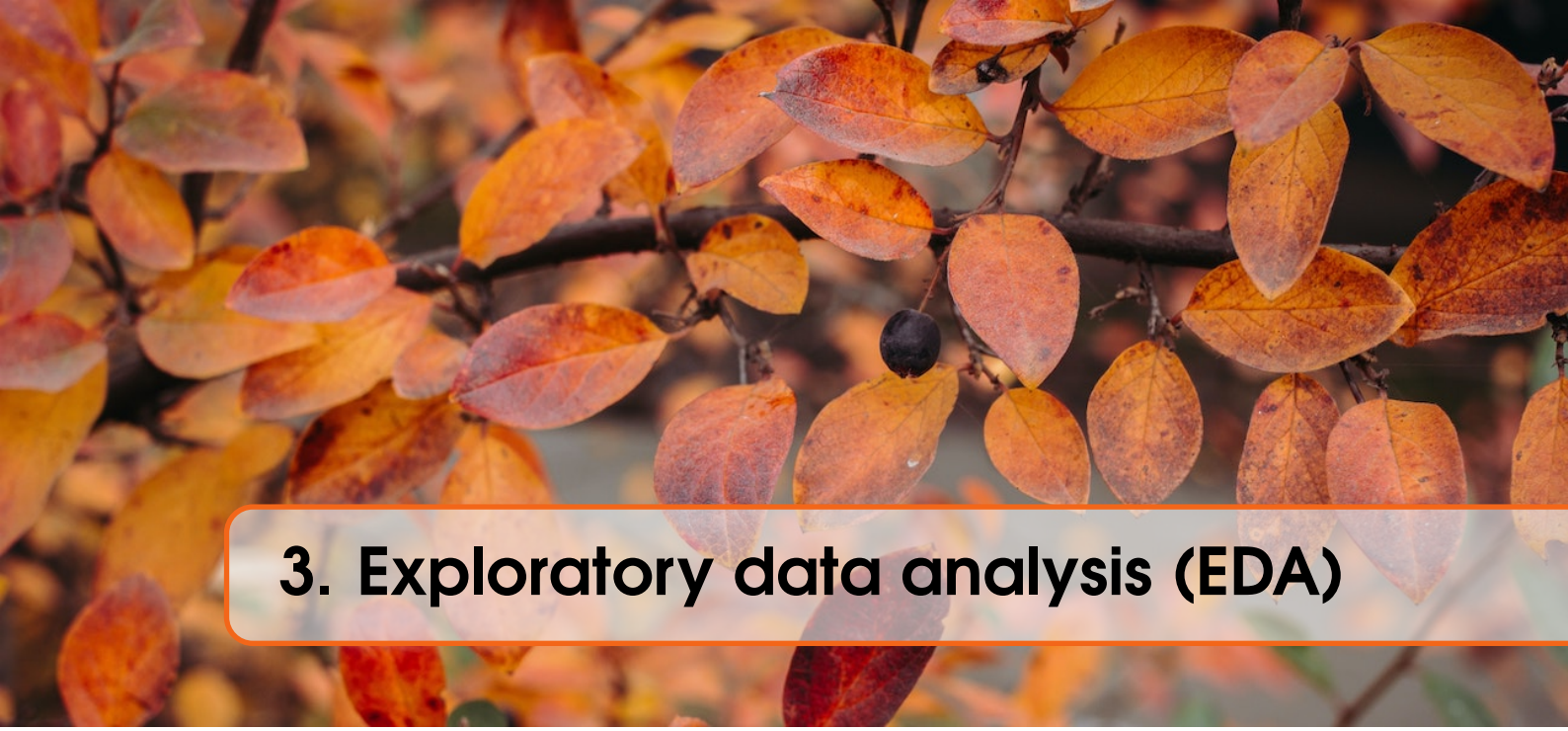
Books

- Hans Fischer "*A History of the Central Limit Theorem*", Springer (2011)
- Carlos Fernandez-Granda "*Probability and Statistics for Data Science*"

Packages

- NumPy
- SciPy

[3] Peter H. Westfall *"Kurtosis as Peakedness, 1905–2014. R.I.P."*, The American Statistician **68** pp. 191-195 (2014)



3. Exploratory data analysis (EDA)

> A computer should make both calculations and graphs. Both sorts of output should be studied; each will contribute to understanding.
>
> *Anscombe (1973)*

> garbage in, garbage out (GIGO)
>
> *unknown (1950's)*

The primary objectives of exploratory data analysis (EDA) in the context of predictive modeling are:

- to become familiar with the data we shall be working with
- awareness of any potential data quality issues
- obtain ideas for feature engineering

3.1 Data quality

In an insightful paper by Carlos Mougan *et al.*[1] they discuss five dimensions of data that lead to a project being able to leverage the most out of machine learning:

- Appropriateness: Is the data actually relevant to the data question(s) posed?
- Readiness: Is the data analysis-ready?
- Reliability: How biased is the data? Biases include historical, representation, measurement and aggregation.
- Sensitivity: How sensitive is the data? Does it contain personal information or data which carries legal, reputational, or political risks?
- Sufficiency: How much data is enough data?

[1] Mougan, Plant, Teng, Bazzi, Cabrejas-Egea, Sze-Yin Chan, Salvador Jasin, Stoffel, Jane Whitaker, Manser *"How to Data in Datathons"*, NeurIPS 2023 **458** pp. 10440-10456 (2023)

	Insufficient	Developing	Functional	Optimal
Appropriateness	Data unrelated to the research question(s).	Target variables are inappropriate for the research question.	Features aren't entirely appropriate for the research question.	Clear relation between features and target variables.
Readiness	No data collected or collection methodology.	Some collected data or clear collection methodology in place.	Data collected, but requires some merging and unifying.	Data collected and cleaned.
Reliability	Strongly biased data.	Indeterminate source of bias.	Determinate source of bias which can be accounted for.	Weak or clear of bias.
Sensitivity	Highly risky sensitive data.	Requires extensive security measures.	Data does not present significant risk but may be confidential.	Limited risks associated with the data, usually publicly available data.
Sufficiency	Prior experience shows low success rate.	Does not meet the level of previous succesful events.	Previous events have successfully operated at this level.	Extensive data in excess of previous successful events.

Figure 3.1: (Image credit: Mougan, Plant, Teng, Bazzi, Cabrejas-Egea, Sze-Yin Chan, Salvador Jasin, Stoffel, Jane Whitaker, Manser *"How to Data in Datathons"*, NeurIPS 2023 **458** pp. 10440-10456 (2023))

Assessing and addressing each of these dimensions forms part of the process of exploratory data analysis.

Regarding synthetic data

If one has insufficient data for machine learning there is sometimes a temptation to perform synthetic data augmentation, for example by using a pair of artificial neural networks known as a generative adversarial network (GAN), a prime example being CTGAN. All I can say on that matter is

> *Some people, when confronted with a small tabular dataset, think "I know, I'll use synthetic data." ...now they have two problems.*

For the same reasons that neural networks struggle to model tabular data, leads them to also fail to synthesize effective tabular data.

3.2 Getting to know your dataframe

We shall assume you have read in your dataset as a pandas dataframe, be it from a file stored in the popular `csv` format (for example with `pd.read_csv()`), in the highly efficient `parquet` format (`pd.read_parquet()`), or directly from a database via an SQL query, for example using something like `.to_dataframe()`. The following pandas methods are great for very quickly getting to know your dataframe `df`:

- `df.shape` - returns the number of rows and columns of the `df`
- `df.head(n)` - print out the first n rows
- `df.tail(n)` - print out the last n rows
- `df.dtypes` - print the data types of each column (float, integer, object, *etc.*)
- `df.describe()` - descriptive statistics for each column

Some useful `pandas` options are

```
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 30)
pd.set_option('display.max_colwidth', 15)
```

The very first thing to do when working with a new dataset is to know how much data one is working with. `df.shape` will tell us the number of rows ('*feature vectors*') and columns ('*features*') we have in the problem. As a rule of thumb for machine learning to be effective one should have `n_rows` $>$ 100 $\times$ `n_columns`.

We can also have too much data; the more data we have evidently the better the model we can make. But training a model on a very large dataset is computationally expensive, and for rapidly creating prototype models we may find it convenient to extract a subsample of the dataset to work with. For example we may like to use a randomly sampled subset of 25K rows of data (note that it is important to take a random sample of the data, and not simply the first 25K rows):

```
sample_df = df.sample(n=25_000, random_state=42)
```

where the `random_state` option permits the creation of a reproducible sample.

Remark: Python very helpfully allows the insertion of underscores within numbers. These are discarded at run-time, but are a great visual aid. (PEP 515)

Once we happy with our choice of estimator and hyperparameters we can then proceed to train our model on the full dataset.

3.2.1 The curse of dimensionality

The curse of dimensionality, a wonderfully florid term by Richard Bellman, is due to having too many columns of data. One can imagine each column of data as an axes in an n_c dimensional vector space. As the volume of this space increases the datapoints rapidly become sparse. For example, say we wish to describe the average person. If our only characteristic, or *feature*, is their height then we could do this rather well. Now say we add more features, such as hair color, eye colour, age, weight, *etc*. Eventually with enough features we would find that every single person becomes unique, and the term average no longer has any meaning. Sparse data is easily overfitted, and overfitting is the bane of machine learning.

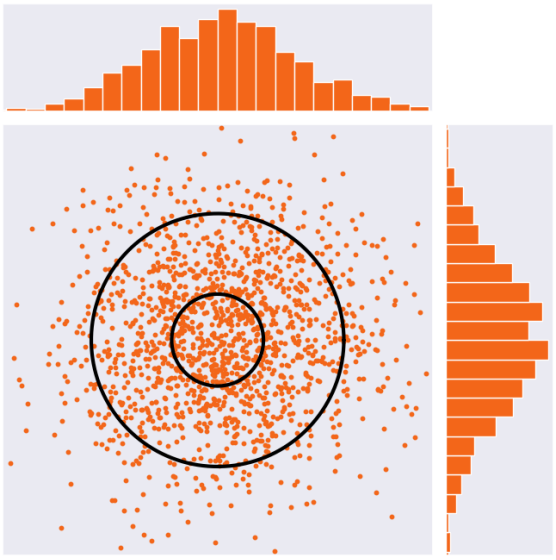


Figure 3.2: The curse of dimensionality in 2-dimensions. The inner circle contains 45% of the points, the outer circle contains 90% of the points.

In the following example we have hypothetical features that each have the same Gaussian distribution (*i.e.* we have a multivariate Gaussian distribution). In one dimension 45% of the data is located roughly between $\pm 0.60\sigma$, and 90% of the data between $\pm 1.64\sigma$, in other words the inner 45% lies along a line of length 1.2, and the outer 45% along a line of length 2.1. The ratio between the outer length and the inner length is 1.75 so on average in one dimension the inner 45% of the data is almost twice as dense as the outer 45%.

dimension	ratio
1	1.75
2	6.6
3	20
4	56
5	157
6	432
7	1191
8	3277
9	9018
10	24810

Table 3.1: Ratios of the volumes of an n-shell to an n-ball for $l_1 = 1$ and $l_2 = 2.75$.

From Table 3.1 we can see that for ten dimensions (*i.e.* 10 features) the outer 45% of the data is almost 25 thousand times more sparse than the central 45%. This is why it is very important in machine learning to have a very large number of rows of data.

3.2.2 Descriptive statistics

Descriptive statistics can be useful in quickly identifying data quality issues, for example perhaps the `Age` column has negative values, or ages over 120. It can give us an idea of the

scale of the columns; `sales` may be of the order of thousands or millions, whereas maybe `temperature` has a scale of a few degrees. Here is the output of a `df.describe().T` for the Titanic dataset

	count	mean	std	min	25%	50%	75%	max
PassengerId	891.000000	446.000000	257.353842	1.000000	223.500000	446.000000	668.500000	891.000000
Survived	891.000000	0.383838	0.486592	0.000000	0.000000	0.000000	1.000000	1.000000
Pclass	891.000000	2.308642	0.836071	1.000000	2.000000	3.000000	3.000000	3.000000
Age	714.000000	29.699118	14.526497	0.420000	20.125000	28.000000	38.000000	80.000000
SibSp	891.000000	0.523008	1.102743	0.000000	0.000000	0.000000	1.000000	8.000000
Parch	891.000000	0.381594	0.806057	0.000000	0.000000	0.000000	0.000000	6.000000
Fare	891.000000	32.204208	49.693429	0.000000	7.910400	14.454200	31.000000	512.329200

Figure 3.3: `pandas describe()` for the Titanic dataset

It reports centrality in the form of the mean and the median (*i.e.* the 50% column), and dispersion in the form of the standard deviation, the IQR (*i.e.* 75%-25%), and the range (*i.e.* $max - min$).

Remark: When using `pandas describe()` it is perhaps preferable to print the transpose (`.T`), especially if the dataset has a lot of features. This then makes for a long table as opposed to a wide table.

3.3 Anscombe's quartet

Anscombe's quartet was invented by Frank J. Anscombe in 1973[2] and consists of four very carefully crafted datasets (here called A, B, C, and D).

Statistic	**A**	**B**	**C**	**D**
$\bar{x}$	9.00	9.00	9.00	9.00
$\bar{y}$	7.50	7.50	7.50	7.50
β_1	0.50	0.50	0.50	0.50
β_0	3.00	3.00	3.00	3.00
$\sum(x_i - \bar{x})^2$	110.0	110.0	110.0	110.0
Regression sum of squares	27.51	27.50	27.47	27.49
Residual sum of squares of y	13.76	13.78	13.76	13.74
Standard error of the slope	0.12	0.12	0.12	0.12
R^2	0.67	0.67	0.67	0.67

Table 3.2: Descriptive statistics for Anscombe's quartet.

[2] F. J. Anscombe "*Graphs in Statistical Analysis*", The American Statistician **27** pp. 17-21 (1973)

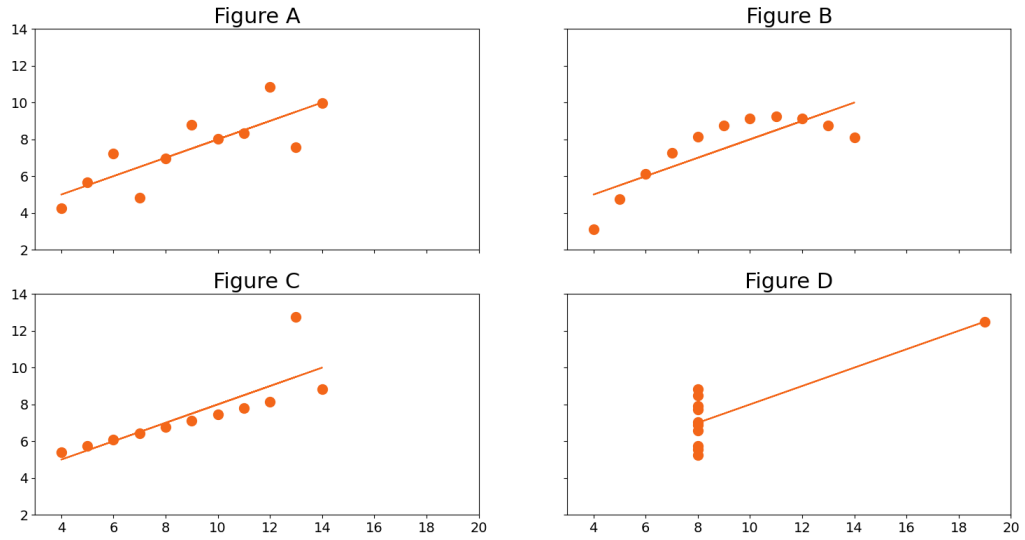


Figure 3.4: Anscombe's quartet

In Table 3.2 we can see that each of the four datasets have almost exactly the same descriptive statistics, but in Figure 3.4 when we visualize the data in a scatter plot we can see they are really rather different:

- in A we seem to have a good fit using linear regression
- in B here we are underfitting, and fitting a polynomial would be much better
- in C there seems to be an outlier point, whose removal would lead to a better fit
- in D we have a zero variance feature along with a 'leverage' point

The message is clear; it is very important to visualize ones data; descriptive statistics alone are simply not sufficient.

> **Remark:** There is a modern incarnation of Anscombe's quartet in the form of The Datasaurus Dozen whose animation is well worth watching.

3.4 Box, violin and raincloud plots

In Figure 3.5 is the first ever box plot, produced by Mary Spear in her 1952 book *"Charting Statistics"*

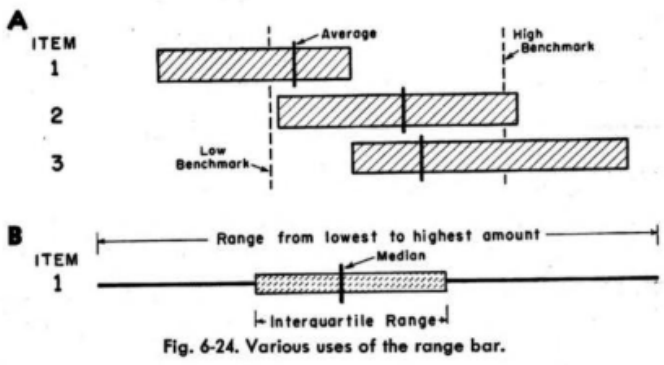


Figure 3.5: The first ever box plot. (Image credit: Mary Spear "Charting Statistics", McGraw-Hill Book Co. (1952))

In Figure 3.6 we draw a box plot as per both matplotlib `boxplot` and pandas `boxplot`, where the box ranges from $Q1$ to $Q3$. We also have the 'Tukey fences', here at $Q1 - 1.5\text{IQR}$ and $Q3 + 1.5\text{IQR}$. Data beyond these fences are deemed to be outliers and are plotted as points. The whiskers then extend to the smallest and largest non-outlier points.

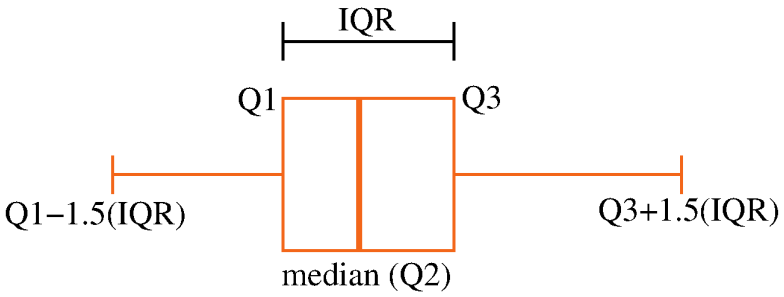


Figure 3.6: Box plot.

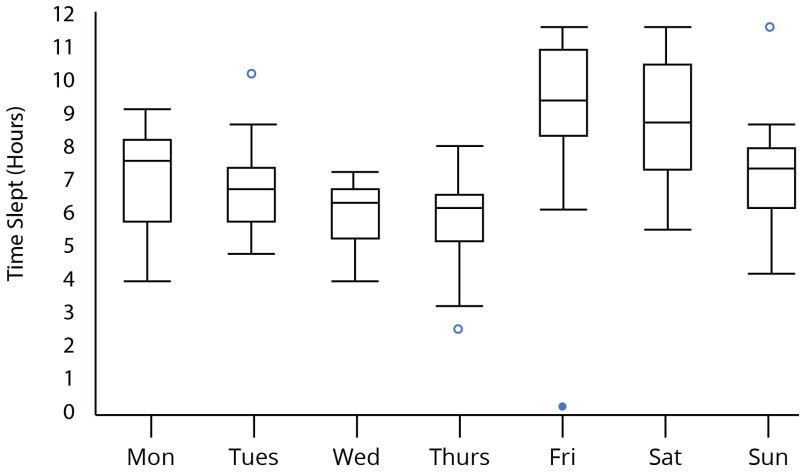


Figure 3.7: A typical box plot in action

However, I would suggesting reading a wonderful article by Nick Desbarats *"I've Stopped Using Box Plots. Should You?"* highlighting the shortcomings of this venerable technique. In Figure 3.8 is an example of a violin plot

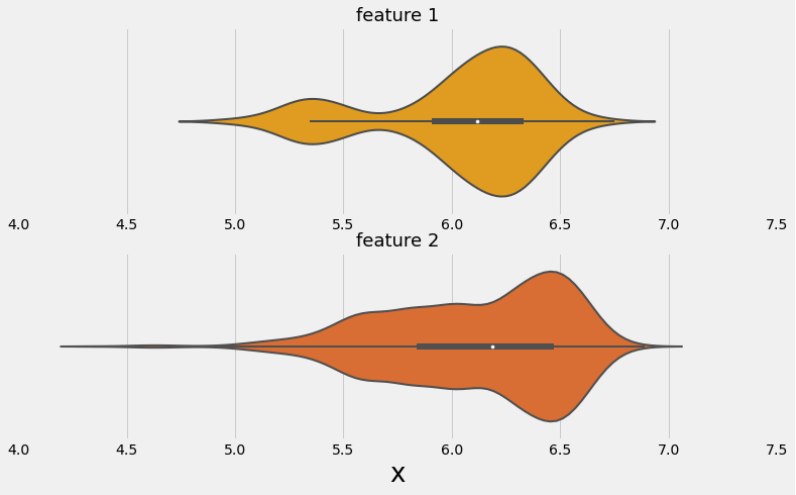


Figure 3.8: A violin plot.

which was produced using `seaborn.violinplot`. Finally, in Figure 3.9 a raincloud plot[3] is a combination of the upper half of a violin plot, a scatter plot, as well as a traditional box plot for good measure, created using the `PtitPrince` package.

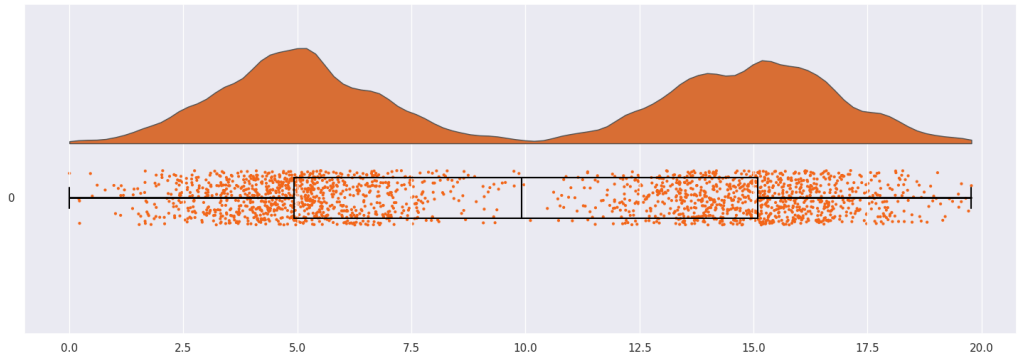


Figure 3.9: A raincloud plot

Jupyter Notebook: Box + strip + violin = raincloud plot

3.5 Outliers, inliers and extreme values

Firstly is is very important to differentiate between outliers and extreme values. Extreme values are not bad values at all and should definitely be kept (they also help to expand the convex hull of the training data). Possible causes of outliers:

- human error when inputting data
- measurement error (broken device *etc*)
- ...

For example

[3] Micah Allen *et al.* *"Raincloud plots: a multi-platform tool for robust data visualization"*, Wellcome Open Research **4** 63 (2021)

- in a dataset containing peoples heights we discover someone who is 6 meters tall: this could well be due to someone recording data in feet rather than meters. Here we can correct this error and keep this datapoint.
- a remote sensing device in the middle of the Sahara desert constantly returns a temperature of -150 degrees: the device is malfunctioning and this data should be discarded

The traditional method for detecting outliers is the z-score

Equation: The z-score is given by

$$z\text{−score} = \frac{X - \bar{x}}{\sigma} \tag{3.1}$$

In python one can use

```
from scipy.stats import zscore

df_z = df.apply(zscore)
```

where values $> |3|$ are beyond the $\pm 3\sigma$ (99.7%) of the rest of the data as per the Gaussian distribution. However, this is predicated on our data actually having a unimodal Gaussian distribution in the first place, which for real data is very rarely the case. Furthermore, the z-score is completely oblivious to *inliers*; potentially bad data that is situated within the bulk of the distribution.

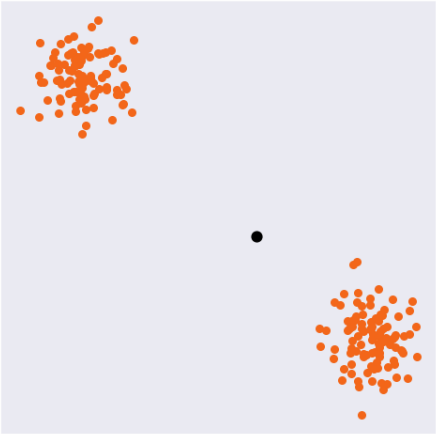


Figure 3.10: An outlier point (black) that is actually an inlier, and will not be found from a z-score.

We can also have multi-dimensional outliers; for example we have a weather dataset that has a temperature column, and also a snowfall column. A univariate analysis of each of these columns may appear to be completely reasonable. However, there could be a row of data that has both a temperature of 39 degrees and at the very same time a snowfall of 3 cm, which is evidently incongruous. In a large dataset, due to the curse of dimensionality it is very difficult to detect such inliers because if there are a large number of features, all datapoints are 'unusual' (see also the isolation forest technique).

Remark: The Mahalanobis distance is the multivariate generalization of the z-score.

3.6 Correlation coefficients

Pearson

Equation: The Pearson correlation coefficient, r_{xy}, measures linear correlation between two sets of data, x and y:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} = \frac{cov(x,y)}{\sigma_x \sigma_y} \tag{3.2}$$

where cov is the covariance and σ is the standard deviation.
Pandas: `df.corr(method='pearson')`

In Figure 3.11 are some values of the Pearson correlation coefficient for various datasets

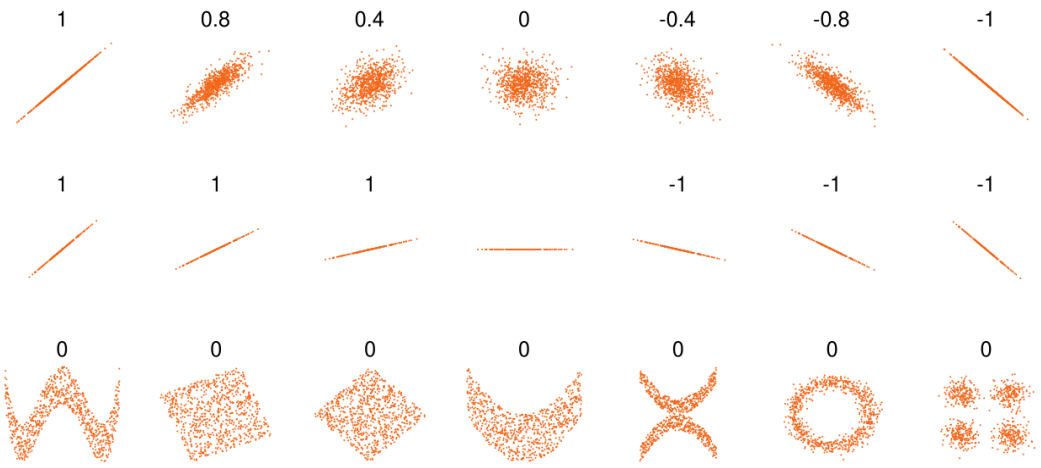


Figure 3.11: Pearson correlation coefficient examples (Image credit: Wikimedia Commons).

The following is an example of a *correlation matrix*. Note that the main diagonal values are always 1, and that the matrix is symmetric ($A = A^{\mathsf{T}}$).

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
PassengerId	1.00	-0.01	-0.04	0.04	-0.06	-0.00	0.01
Survived	-0.01	1.00	-0.34	-0.08	-0.04	0.08	0.26
Pclass	-0.04	-0.34	1.00	-0.37	0.08	0.02	-0.55
Age	0.04	-0.08	-0.37	1.00	-0.31	-0.19	0.10
SibSp	-0.06	-0.04	0.08	-0.31	1.00	0.41	0.16
Parch	-0.00	0.08	0.02	-0.19	0.41	1.00	0.22
Fare	0.01	0.26	-0.55	0.10	0.16	0.22	1.00

Figure 3.12: Pearson correlation matrix for the Titanic dataset.

Highly correlated features (columns) are not a good thing to have in our model, and can complicate the interpretation of a fit. There is even a good chance that two very highly correlated features are simply different ways of representing the very same thing, leading to multicollinearity. For example there may be a column with the speed of a car in *kmph* and another column in *mph*. If this is the case it is convenient to remove (`drop`) one of the offending features:

```
df = df.drop(["mph"], axis=1)
```

N **Jupyter Notebook:** Explainability, collinearity and the variance inflation factor (VIF)

Sample code

Here is some code to create a sortable list of the correlations between columns of a dataframe

```
import itertools

columns_list = df.select_dtypes(include='number').columns.tolist()
pairs = itertools.combinations(columns_list, 2)
correlation_list = []

for pair in (pairs):
    correlation_list.append([pair[0], pair[1],
                             df[pair[0]].corr(df[pair[1]])] )

correlation_df = pd.DataFrame(correlation_list)
correlation_df.sort_values(by=[2], ascending=False)
```

Spearman

Equation: The Spearman rank correlation coefficient measures monotonic correlation

between the ranks (R) of two sets of data, x and y both of size n:

$$r_{xy} = 1 - \frac{6\sum(R(x_i) - R(y_i))^2}{n(n^2-1)} \tag{3.3}$$

Pandas: `df.corr(method='spearman')`

We shall now compare these correlation coefficients with the linear regression coefficient β_1 for the toy dataset illustrated in Figure 3.13 the results of which are in Table 3.3.

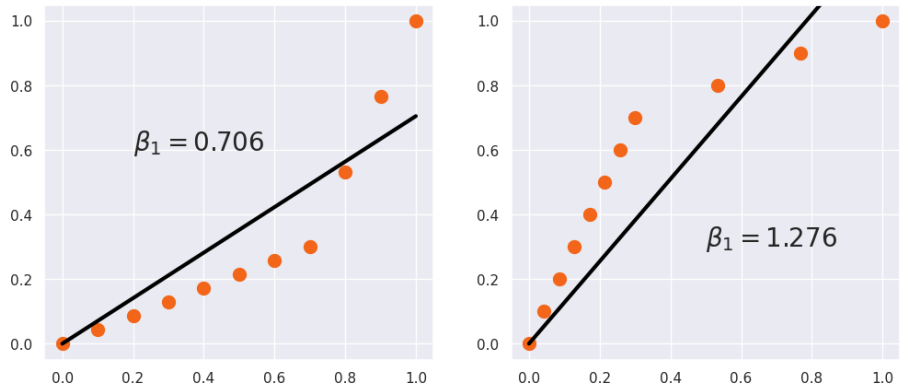


Figure 3.13: Toy dataset.

Pearson(x,y) = Pearson(y,x)	0.915
Spearman(x,y) = Spearman(y,x)	1.000
$y = \beta_1 x$	$\beta_1 = 0.706$
$x = \beta_1 y$	$\beta_1 = 1.276$

Table 3.3: Correlation coefficient results for the toy dataset in Figure 3.13.

We have a linear Pearson correlation coefficient of 0.915. The non-parametric Spearman correlation, based on rank, where in this example each x,y pair has a greater value than the last is thus a perfect 1. The linear regression coefficient β_1 for $y = \beta_1 x$ is 0.706, and if we swap the variables then for $x = \beta_1 y$ it becomes 1.276. However, if we standardize both x and y then in both cases β_1 becomes 0.915, *i.e.* the same as the Pearson coefficient.

3.6.1 Mutual information (MI)

Both of the datasets plotted in Figure 3.14 have Pearson correlation coefficients of zero as it only picks up on linear relations. However, intuitively we can see that there is indeed a relation between the two axes (for the LHS example for a circle $x^2 + y^2 = r^2$).

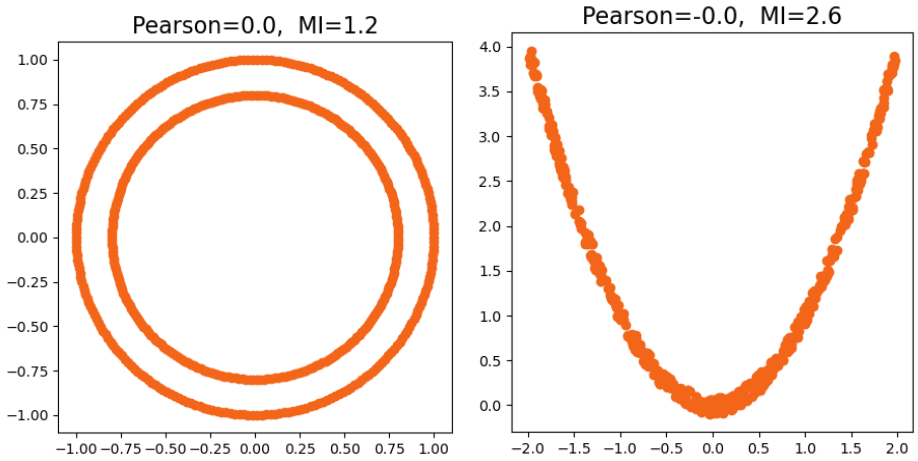


Figure 3.14: Example Pearson correlation and mutual information scores.

The mutual information[4] can be calculated via:

```
from sklearn.feature_selection import mutual_info_regression

mutual_info_regression(X, y)
```

N **Jupyter Notebook:** Pearson correlation coefficient, mutual information (MI) and Predictive Power Score (PPS)

3.7 Scatter plot

I would suggest that the humble scatter plot is one of the most useful tools when it comes to visual inspection of data.

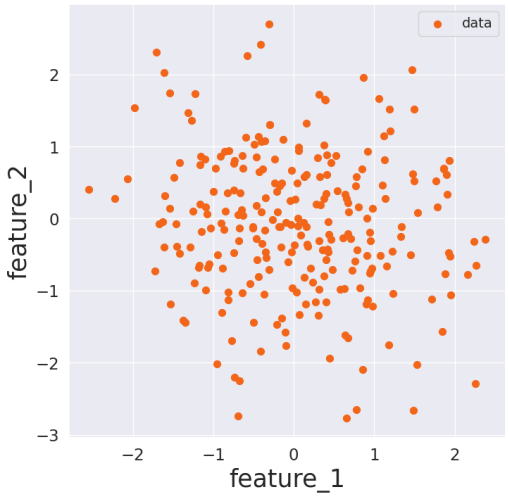


Figure 3.15: Scatter plot

[4] Kraskov, Stögbauer, Grassberger *"Estimating mutual information"*, Physical Review E **83** 019903 (2011)

Remark: If one intends to later apply feature scaling it may not be a bad idea to visualize scatter plots using an aspect ratio of 1:1.

One can also produce interactive 3D scatter plots

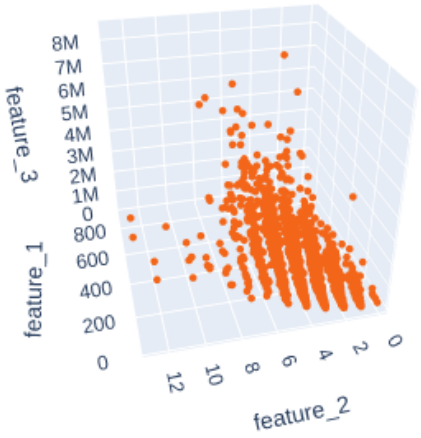


Figure 3.16: 3D Scatter plot

Sample code

Code to produce an interactive 3D scatter plot using `plotly.express`

```
import plotly.express as px
import plotly.graph_objects as go

fig = go.Figure()
fig = px.scatter_3d(x= dataset["feature_1"],
                    y= dataset["feature_2"],
                    z= dataset["feature_3"],
                    title="3D scatter plot",
                    labels={"x": "feature_1",
                            "y": "feature_2",
                            "z": "feature_3"}
                   )
fig.update_traces(marker=dict(size=1.5))
fig.show()
```

3.8 Histograms and eCDF

Histograms are fantastic for visualizing the distribution of the data; to see whether a feature is unimodal, bimodal, or multimodal, or for visually detecting skewness or kurtosis. However, they have a potentially major drawback in that the information conveyed can change considerably depending on the choice of bin width. In Figure 3.17 both of the plots are for the very same dataset. On the other hand the empirical cumulative distribution function (eCDF) is independent of the width of the bins.

Remark: In `seaborn` one can set `cumulative=True` or use the `seaborn.ecdfplot`)

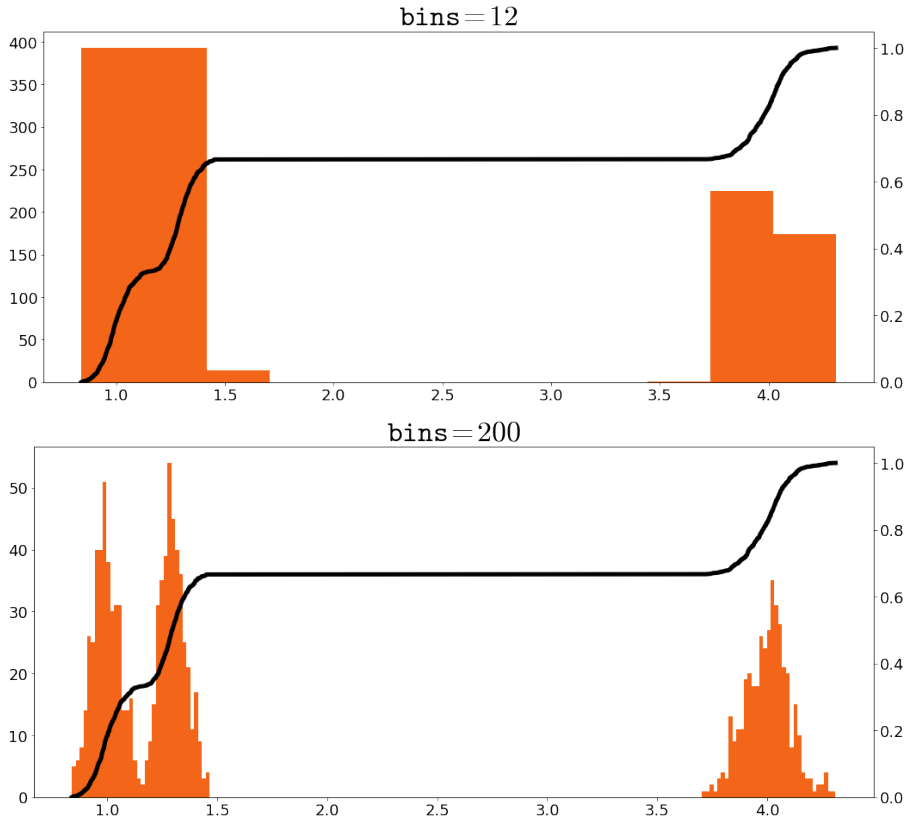


Figure 3.17: Histograms (orange) with eCDF (black). The upper histogram has 12 bins as suggested by Sturges's rule. The lower histogram has 200 bins.

The main thing we are looking out for are plateaux on the eCDF; we can see a small plateau on the LHS due to the lack of data between the LHS bimodal peaks, whilst this bimodality is not seen at all in the upper histogram in the figure thus observing such a plateau in the eCDF is a warning that our bins are too wide and we should make them narrower. At the other extreme if there are too many bins then each datapoint will have its very own bin and we learn very little.

Equation: Sturges's rule suggests that a reasonable estimate for the number of bins for a series is given by

$$\texttt{n\_bins} = \lceil 1 + 3.322 \log_{10}(N) \rceil \tag{3.4}$$

where N is the number of samples. However, this rule assumes the data is from a unimodal Gaussian distribution, which is frequently not the case and Sturges's rule will under-estimate the number of bins.

Sample code

Calculate the eCDF data via

```
import numpy as np

x_eCDF = np.sort(data)
y_eCDF = np.arange(1, len(data)+1 ) / len(data)
```

and to plot both the histogram and the eCDF together

```
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots(figsize=(15, 7))
twin = ax1.twinx()
ax1.hist(data, bins=12, density=False)
twin.plot(x_eCDF, y_eCDF, linewidth=5, color="red")
twin.set_ylim(bottom=0, top=None)
plt.show();
```

3.8.1 Kolmogorov-Smirnov test

The (two-sample) Kolmogorov-Smirnov test compares the eCDF for two distributions. This can be very useful in detecting if there is significant covariate shift (or concept drift if comparing the target columns) between the data the model was trained on and the data that the model is being asked to predict for.

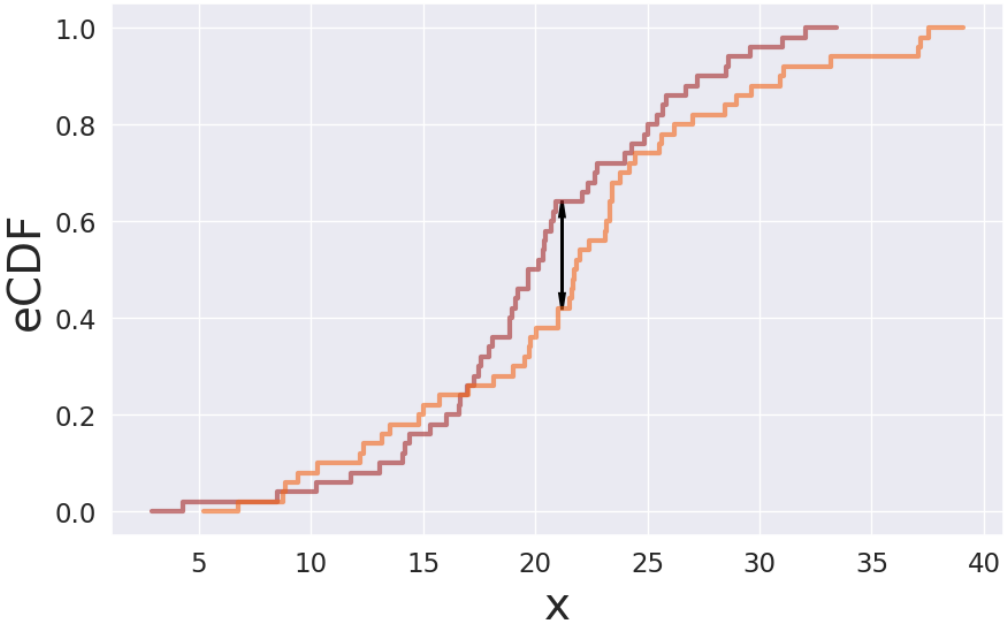


Figure 3.18: The two-sample Kolmogorov-Smirnov test.

It is based on finding the largest vertical distance between the two eCDF

$$D_n = \sup_x |\text{eCDF}_A(x) - \text{eCDF}_B(x)| \tag{3.5}$$

and can be computed via

```
from scipy import stats

KS_statistic, p_value = stats.kstest(eCDF_A, eCDF_B)
```

3.9 Pairplots (or not)

Pairplots are rather convenient, especially if one likes penguins or is interested in varieties of iris flowers, but unwieldy if there are say more than 10 features as this will lead to a 10×10 grid of tiny and basically useless figures.

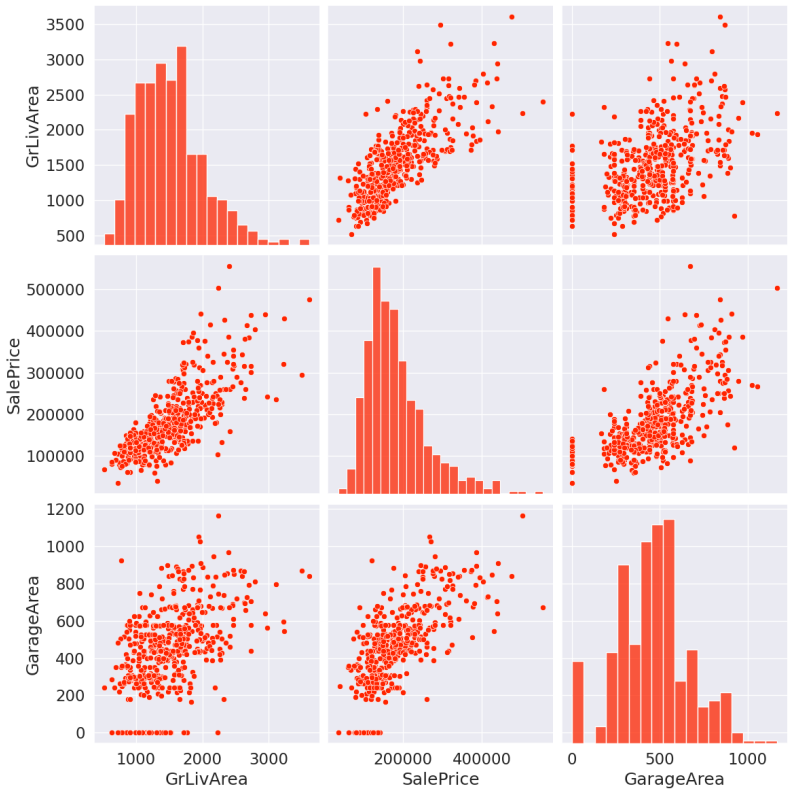


Figure 3.19: An example pairplot.

If one has more than 10 features I would suggest, although not elegant, printing out the scatter plots one after the other in a column to aid detailed visual inspection. For n features the following script will produce $(n^2 - n)/2$ plots:

Sample code

```
import seaborn as sns
import itertools

df = dataset.select_dtypes(include='number')
numeric_features = df.columns.tolist()
pairs = itertools.combinations(numeric_features, 2)
for pair in (pairs):
```

```
    sns.lmplot(data=df, x=pair[0], y=pair[1],
               line_kws={'color': 'orange'},
               aspect=1 )
    plt.show();
```

Recommended reading

Books

- John W. Tukey "*Exploratory Data Analysis*", Pearson (1977)

Packages

- Pandas
- matplotlib
- seaborn
- Plotly
- PtitPrince by Davide Poggiali
- ydata-profiling

Other

- PyViz: lists of libraries for visualizing data in Python
- ISO/IEC 25012:2008 general data quality model
- ISO/IEC 25024:2015 measurement of data quality

4. Data cleaning

> ...the tasks of exploratory data mining and data cleaning constitute 80% of the effort that determines 80% of the value of the ultimate data mining results.
>
> *"Exploratory Data Mining and Data Cleaning" by Tamraparni Dasu and Theodore Johnson (2003)*

Machine learning is a numerical technique and as such can only be performed on a complete dataset, composed entirely of numbers. Indeed we will not be far wrong if we start thinking of our dataset as a two dimensional matrix. However, there may be missing values or features encoded alphanumerically.

4.1 Missing values: `NULL` and `NaN`

It is far from unusual for there to be missing values in a dataset, for many reasons including the merging of distinct tables. These missing values are represented by a placeholder, such as `NULL` or by `NaN` (**N**ot **a** **N**umber).

4.1.1 Visualization of `NaN` with `missingno`

A great visual tool to get an overall idea as to how much data is missing, and where, is the `missingno` package.

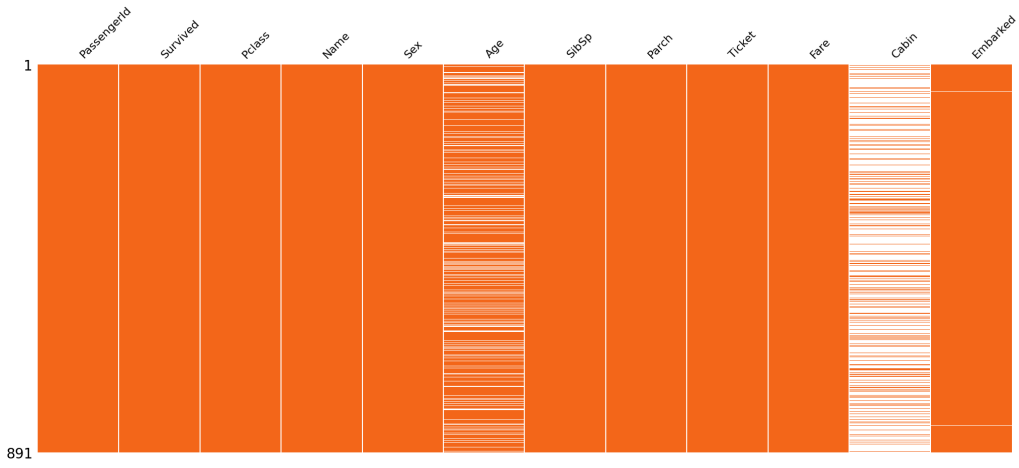


Figure 4.1: `missingno` for the Titanic data

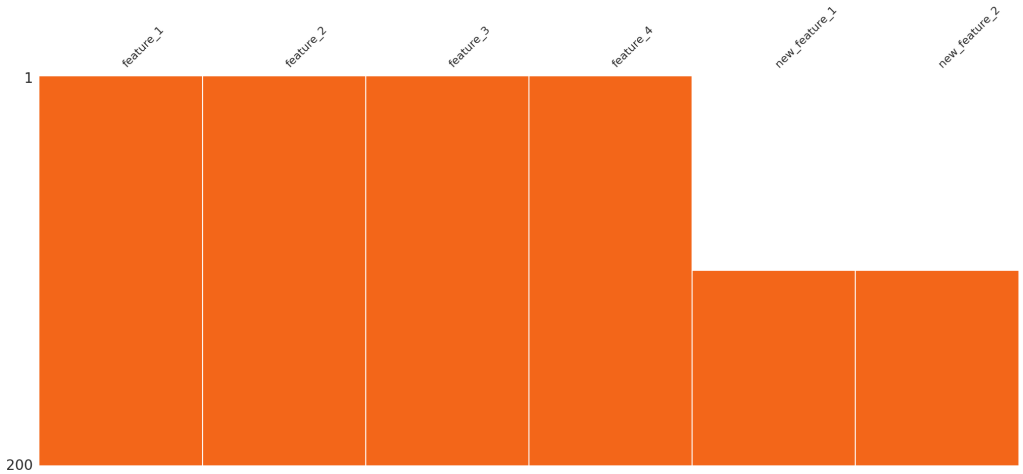


Figure 4.2: `missingno` of two concatenated dataframes: the lower `df` had two new features, leading to `NaN` values in the upper `df`.

and perform a quantitative analysis with `pandas`:

- `df.isna().sum()` - count the number of missing numbers in each column
- `df.isna().sum().sum()` - the total number of missing numbers in the `df`

Remark: The pandas method `df.isnull()` can be used too, as it is an alias for `df.isna()`. (I am not sure why the latter method was not more intuitively named `df.isnan()`).

What are some of the options we have to deal with missing values? Imputation:

- fill with a constant value
- deletion
- fill with an average value
- ...

4.1.2 MCAR, MAR, and MNAR

Donald Rubin created a taxonomy for different types of missing data, depending on how

the data came to be missing in the first place.

- **MCAR** (missing completely at random) whether the data is missing or not has nothing to do with the data itself.
- **MAR** (missing at random)
- **MNAR** (missing not at random) the probability of this data being missing is a function of the missing value itself.

Knowing if there is potential reason for the data being missing can help in ones effort to solve the problem.

4.1.3 Global fill

Suffice to say this is not a good idea at all, but to get going quickly on a project the easiest thing to do is replace all of the NaN with a single value, say for example with zero:

```
df = df.fillna(0)
```

4.1.4 Global delete

Another somewhat drastic quick fix, and again ill advisable in the long run, is to simply delete all of the NaN values. To delete all the columns that contain even just one missing value:

```
df = df.dropna()
```

To delete all of the rows that have one or more missing values:

```
df = df.dropna(axis='columns')
```

One can also set a threshold (`thresh`) for deletion. For example to delete columns only if more than 80% of the data in the column is missing:

```
df.dropna(thresh= int(0.2*df.shape[0]))
```

Be aware that if the data is MNAR then deletion can introduce a bias in the resulting model.

4.1.5 Average value imputation

A very commonly used approach is to fill the missing values in the column with the average of said column, be it the mean, median, or mode. How to fill `NaN` with the mean:

```
df = df.fillna(df.mean())
```

How to fill `NaN` with the mode:

```
df['col'] = df['col'].fillna(df['col'].mode()[0])
```

Note that this operation is a little more elaborate because the most frequent value is not necessarily unique.

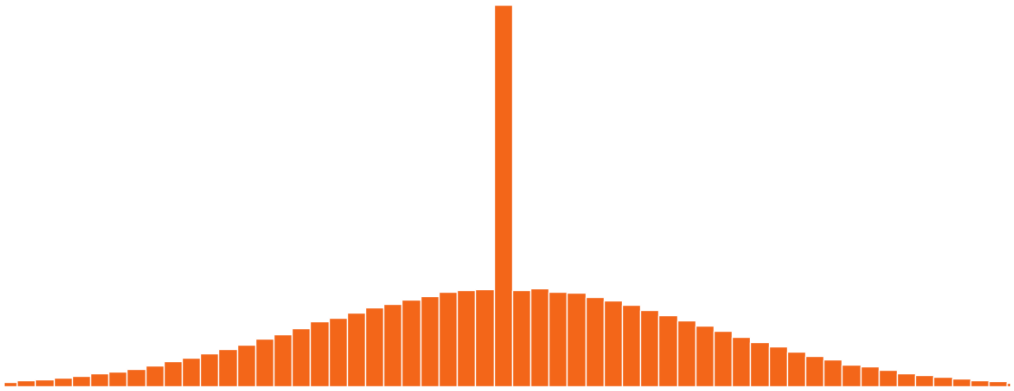


Figure 4.3: The effect of mean imputation (for 10% missing).

It is worth mentioning that mean imputation is less deleterious for predictive models than it is for inferential models[1].

Remark: `Pandas` cannot deal with infinite values (produced for example by the division by zero). The trick is to first replace them with NaN values
`df = df.replace([np.inf, -np.inf], np.nan)`

4.1.6 Multiple imputation

Multiple imputation methods, multivariate imputation by chained equations (MICE) being a well known example, involve treating a column with missing values as the target column and predicts the missing values using the other features. This is repeated for all of the columns that have missing data. This whole process is then repeated multiple times.

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

imputer = IterativeImputer(random_state=42, max_iter=10)
imputer.fit(dataset)
dataset = imputer.transform(dataset)
```

4.1.7 Do nothing!

Global fill with missing flag

For tree based estimators it is entirely possible that the best thing to do is nothing and leave them to the algorithm. Either leave the NaN as they are or globally replace them with an integer unlikely to occur in the dataset (such as `-99999`), then specify the missing flag: `df.fillna(-99999)` then (XGBoost parameter) `missing=-99999`
Indeed it has been found that

[1] Josse, Chen, Prost, Scornet, Varoquaux "*On the consistency of supervised learning with missing values*", arXiv:1902.06931 (2024)

> *"Native support for missing values in supervised machine learning predicts better than state-of-the-art imputation with much less computational cost".*[2]

4.1.8 Binary indicator column

When data is MNAR adding a binary 'dummy' feature flagging where data is missing can become an informative feature. For example, missing financial data could turn out to be a good indicator of fraud.

```
df["col_is_NaN"] = np.where(np.isnan(dataset["col"].values), 1, 0)
```

4.2 Outliers and inliers

4.2.1 Outliers

If we are decided that we have an unwelcome outlier value there are two basic options:

deletion

Example of deleting rows whose value in the age column `Age` are below 0 or above 120 from the dataframe

```
df = df[ (df["Age"] >= 0) & (df["Age"] <= 120) ]
```

or a less drastic option is to replace the offending values with a `NaN` to then be imputed afterwards; here we replace all values below 10 in the column `col` with `NaN`:

```
df["col"] = np.where(df["col"]<10, np.nan, df["col"])
```

clipping / Winsorizing

Another option is to clip values below (or above) a value to be said value (this is also known as Winsorizing, named after Charles Paine Winsor). Here all values below 0 become 0, and all values above 120 become 120:

```
df["col] = df["col].clip(lower=0, upper=120)
```

4.2.2 Inliers: Isolation forest

When it comes to inliers one option is to use the unsupervised classification (an area of machine learning not covered in this book) technique known as the `Isolation Forest`

Sample code

```
from sklearn.ensemble import IsolationForest

IF_classifier = IsolationForest(n_estimators  = 100,
                                contamination = 0.05,
                                random_state  = 42)

y_pred_IF = IF_classifier.fit_predict(X_train)
```

[2] Perez-Lebel, Varoquaux, Le Morvan, Josse, Poline *"Benchmarking missing-values approaches for predictive models on health databases"*, GigaScience **11** giac013 (2022)

```
X_train_filtered, y_train_filtered = X_train[(y_pred_IF == 1)], \
                                     y_train[(y_pred_IF == 1)]
```

The contamination hyperparameter is the fraction of datapoints to identify as outliers, and subsequently delete these outlier rows from the dataset.

N **Jupyter Notebook:** Filtering outliers using the Isolation Forest

4.3 Duplicated rows

Duplicated rows can unduly bias a machine learning model, and can arise for example in a dataset created via an SQL `UNION ALL` command or a pandas `concat`. To count the number of duplicated rows in a dataset one can use
`len(df) - len(df.drop_duplicates())`
and to delete them
`df = df.drop_duplicates()`

4.4 Boolean columns

To convert a Boolean column (`True` or `False`) into a binary column:
`df["col"] = df["col"].astype("int")`

4.5 Zero variance columns

As far as machine learning is concerned columns that have zero variance have no predictive power whatsoever and are best dropped. This can be done using
`df = df.loc[:, df.var() != 0]`
If there are categorical columns, where the variance is not defined, one could use:
`df = df.drop(df.columns[df.nunique() == 1], axis=1)`
Such a situation could occur if the dataset is the result of filtering. For example, say one had a dataset of global sales, but decided to create a model for the sales in Spain. Filtering the country column for Spain would naturally lead to zero variance for the country feature. Another reason to drop such columns is that the model is completely unprepared to produce a prediction if somehow a new value is encountered in the corresponding column in the test or production data.

Remark: Scikit learn has a filter `sklearn.feature_selection.VarianceThreshold` although I would be hesitant to recommend it; zero variance columns can be removed with impunity, but even a slight variance in some features could be informative.

4.6 Feature scaling: standardization and normalization

From a numerical standpoint some solvers, such as gradient descent or L-BFGS-B used in logistic regression, work best when the data is either standadized or normalized.

Equation: Standardization is when we set the mean to be zero and then scale to unit

variance ($\sigma = 1$)

$$z = \frac{(x - \mu)}{\sigma} \tag{4.1}$$

Sample code

```
# fit
x_train_mean = x_train.mean()
x_train_std  = x_train.std()

# transform the training data
z_train = (x_train - x_train_mean) / x_train_std

# transform the test data
z_test = (x_test - x_train_mean) / x_train_std
```

Note that we fit the parameters μ and σ to the training data, and use the same parameters when transforming the test data. This is done to prevent data leakage.

Equation: Normalization scales the data to lie in the range $[0, 1]$

$$z = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{4.2}$$

Remark: Both standardization and especially normalization are very sensitive to the presence of outliers.

I would suggest the best scaler is the `sklearn.preprocessing.RobustScaler` which removes the median and scales the data according to the interquartile range (IQR).

Equation: Robust scaling

$$z = \frac{x - Q_2(x)}{Q_3(x) - Q_1(x)} \tag{4.3}$$

Sample code

```
from sklearn.preprocessing import RobustScaler

RS = RobustScaler()
train_data = RS.fit_transform(train_data)
test_data  = RS.transform(test_data)
```

Feature scaling can be performed using the following scikit-learn routines:

- `sklearn.preprocessing.MinMaxScaler`
- `sklearn.preprocessing.StandardScaler`
- `sklearn.preprocessing.RobustScaler`

4.7 Categorical features

Categoricals are discrete features composed of values that belong to a set. They may be numerical or they may be strings (objects). There are two types of categorical features; ordinal and nominal.

- **ordinal**: have an order to them, *i.e.* a $<$ b $<$ c
- **nominal**: nominated values, *i.e.* Spain, USA, China

4.7.1 Ordinal

An example of an ordinal feature would be a dataset of Olympic athlete performance where there is a `Medals` column, which may contain the values `Gold`, `Silver`, `Bronze`, and `none` (or `NULL`). What do we do with this feature? We simply numerically re-code the column:

- label encoding: a $\rightarrow$ 1, b $\rightarrow$ 2, c $\rightarrow$ 3
  `sklearn.preprocessing.OrdinalEncoder()`
- encoding by hand using a dictionary, for example:
  `df["Medals"] = df["Medals"].replace({"Gold":3,"Silver":2,"Bronze":1,"none":0})`
  or via a mapping
  `df["Medals"] = df["Medals"].map({"Gold":3,"Silver":2,"Bronze":1,"none":0})`

4.7.2 Nominal

Nominal features contain 'nominated' values, a good example are post codes (ZIP codes). Although numerical in nature these are not actually a meaningful numbers; it was nominated. What do we do with nominal features: *one-hot encoding*. Example: here the feature `orientación` of type `object` has three distinct values:

orientación
norte
este
norte
sur

which is converted into three new features, now with binary values (and delete the original feature):

orientación_norte	orientación_este	orientación_sur
1	0	0
0	1	0
1	0	0
0	0	1

This can be done via

- One-hot encoding creates a bit-mask leading to n new columns
  `sklearn.preprocessing.OneHotEncoder()`

- One-hot encoding can also be performed using `pandas.get_dummies`:
  `df = pd.get_dummies(df, drop_first=True)`

Remark: Dummy Variable Trap: One hot encoding will create multicollinearity and the solution is to drop one of the columns, for example the very first column via `drop_first=True`.

Note that, for example, in Spain there are 11,752 post codes. One-hot encoding the post code feature could create up to 11751 columns of very sparse data. The post codes for Madrid all start with 28, whereas the post codes for Barcelona start with 08. Madrid is not 28 because it is more than Barcelona or vice versa; these codes were assigned alphabetically by province. One could drastically reduce the sparsity of the dataset by, for example, just using the first two digits of the post code as the features to be encoded, which in would lead to a much more manageable 52 columns of data.

Frequency or count encoding

An alternative worth exploring which does not involve the creation of new columns is to use the frequency that we see a given category as the value that is encoded.

```
count_map = X_train["cat_col"].value_counts().to_dict()
X_train["cat_col"] = X_train["cat_col"].map(count_map)
X_test["cat_col"] = X_test["cat_col"].map(count_map)
```

Indeed the package Category Encoders has no less than 20 different encoders for transforming ones categorical data.

Remark: The SoTA GBDT estimators have support for categorical features:

- XGBoost set: `enable_categorical=True`
- LightGBM parameter: `categorical_feature=`
- CatBoost parameter: `cat_features=`

Sample code

```
categorical_features = ['has_car', 'has_house', 'has_yacht']

for feature in categorical_features:
    df[feature] = df[feature].astype('category')
```

Recommended reading

Papers

- Yufeng Ding, Jeffrey S. Simonoff *"An Investigation of Missing Data Methods for Classification Trees Applied to Binary Response Data"*, Journal of Machine Learning Research **11** pp. 131-170 (2010)

Books

- "DAMA DMBOK2", Technics Publications (2017)
- Stef van Buuren *"Flexible Imputation of Missing Data"*, CRC Press (2018)

Packages

- scikit-learn
- missingno by Aleksey Bilogur
- Python Outlier Detection (PyOD)
- Category Encoders
- missingpy
- AWS Deequ

5. Cross-validation

All analytics models do well at what they are biased to look for.

Matthew Schneider

Once we have explored our data and cleaned it up we are now ready to create our machine learning model. Perhaps the biggest difference between the statistical modeling ('$\hat{\beta}$') and the machine learning modeling ('$\hat{y}$') paradigms is how model quality is evaluated. In statistical modeling the emphasis is on goodness-of-fit tests; ¿how well does the model fit to the data that it was fitted on? On the other hand, machine learning modeling tests how well the model does at predicting the target for data that it has never seen before. In this sense machine learning modeling is more akin to the scientific method, and maybe the reason for the term 'data science'. In this chapter we see how our dataset is partitioned into a number of disjoint datasets for the purpose of estimator evaluation and for hyperparameter tuning.

5.1 Train test split

The very simplest incarnation of performing the 'train-test split' is shown in Figure 5.1. Our dataset is composed of our feature columns ($\mathbf{X}$) and our target column (y).

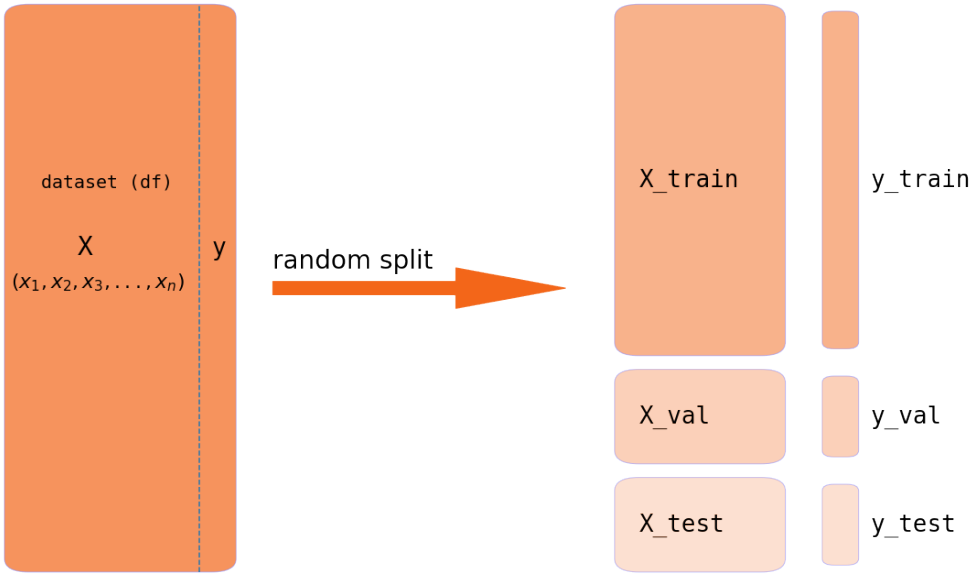


Figure 5.1: Splitting a dataset

Our first operation is to split the data vertically, separating the feature columns from the target column. Then we create (at least) three disjoint datasets, namely train, validation and test:

- `train`: the training data is used to find the parameters of the model
- `validation`: the validation dataset is used to find the best hyperparameters
- `test`: (or *hold-out*) is used as a performance 'reality-check':

> *"...the cautious statistician who sets aside a randomly selected part of his sample without looking at it and then plays without inhibition on what is left, confident in the knowledge that the set-aside data will deliver an unbiased judgment on the efficacy of his analysis."* M. Stone[1] (1974)

A rough guide is to perform a 'Pareto' split: 80% of the data for training purposes, and the remaining 20% is used for the validation and the test sets, which should be around the same size (same number of rows). It is very important that each dataset should be representative of the whole (or be independent and identically distributed (i.i.d.) random variables in statistical parlance). A very simple way to achieve this is simply to randomly shuffle the rows before performing the splits.

Sample code

First create the 'vertical' split

```
X = dataset
X = X.drop(["Target"], axis="columns")
y = dataset["Target"]
```

[1] M. Stone *"Cross-Validatory Choice and Assessment of Statistical Predictions"*, Journal of the Royal Statistical Society: Series B (Methodological) **36** pp. 111-133 (1974)

then the 'horizontal' splits, scikit-learn provides a helpful routine to take care of this task for us, namely the `train_test_split`

```
from sklearn.model_selection import train_test_split

X_train, X_tmp, y_train, y_tmp = train_test_split(X, y,
                                                  test_size=0.2,
                                                  random_state=42)
X_val, X_test, y_val, y_test   = train_test_split(X_tmp, y_tmp,
                                                  test_size=0.5,
                                                  random_state=42)
del X_tmp, y_tmp
```

When performing classification, to ensure equal proportions of each class are assigned to each split one can use `stratify=y`. It is a good idea to also use `random_state=42` in order to make the random split reproducible for testing purposes. This seed value can also be periodically changed to create 'fresh' datasets in order to prevent overfitting to a specific validation dataset.

5.2 Cross-validation

Cross-validation is a clever technique designed to make the very most out of a small dataset where one may not be able to afford the luxury of holding out a sizable chunk of data for validation purposes. In cross-validation the training dataset also becomes the validation dataset. This is achieved by k-fold cross-validation, drawn schematically in Figure 5.2 using $k = 5$:

Remark: The choice of $k = 5$ is so popular that it is often abbreviated as simply 5CV.

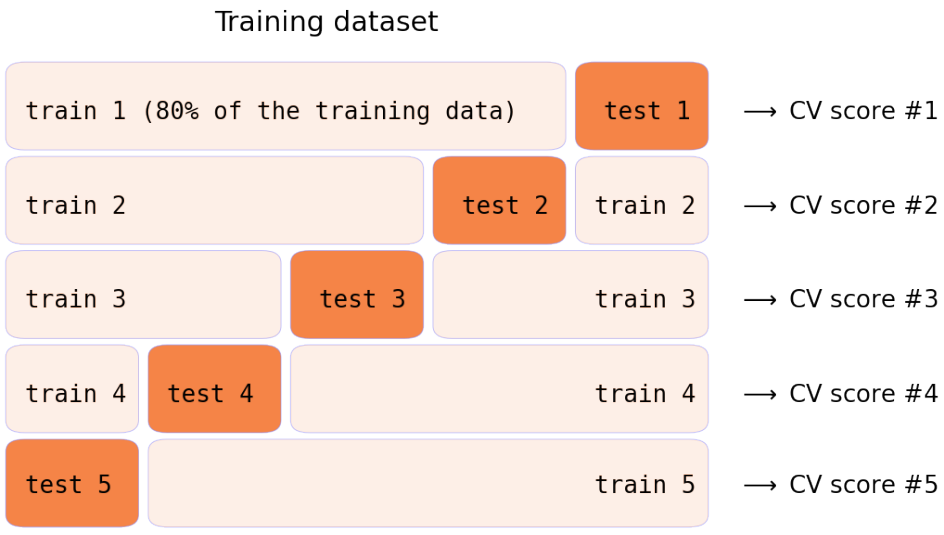


Figure 5.2: Schematic of 5-fold cross validation

Taken to the extreme one can even perform *leave-one-out* CV, where each time the validation set consists of just a single row of the training data. However, this implies training as many models as there are rows of training data. For obvious reasons this is not an advisable strategy when using large datasets.

Sample code

Scikit-learn helpfully provides a `cross_val_score` routine

```
from sklearn.model_selection import cross_val_score
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error

RMSE = make_scorer(mean_squared_error, squared=False)
CV_scores = cross_val_score(regressor, X, y, cv=5, scoring=RMSE)
print("* CV scores: ",CV_scores)
print("* RMS of CV RMSE: ",round(np.sqrt(np.mean(np.square(CV_scores))),1))
print("* Std. dev of CV score: ",round(np.std(CV_scores),1))
```

Remark: Note that when using the RMSE metric one should not directly average over the k individual CV scores but rather

$$RMSE = \sqrt{\frac{\sum (RMSE)^2}{k}}$$

where it is assumed that there are equal numbers of samples in each split.

However the `cross_val_score` admits the possibility of data leakage. Here is an example of performing cross-validation where we can apply bespoke `cleaning` functions to the `X` data, such as fits and transformations to the `X_train` data, and apply corresponding transformations of the `X_test` data

```
from sklearn.model_selection import KFold

#n_folds = len(X) # LOOCV
n_folds = 5 # 5CV
kf = KFold(n_splits=n_folds,
           random_state=42,
           shuffle=True)

def fold_training(regressor, X, y):
    OOF_predictions = []
    i=1
    for train_index, test_index in kf.split(X, y):
        #print("Running fold: ",i)

        X_train = X.iloc[train_index]
        X_train = cleaning_train(X_train)
```

```
        X_test  = X.iloc[test_index]
        X_test  = cleaning_test(X_test)

        y_train   = y.iloc[train_index]
        y_test    = y.iloc[test_index]

        regressor.fit(X_train,y_train)
        y_pred = regressor.predict(X_test)

        # save the OOF predictions for later:
        preds_df = pd.DataFrame(test_index, columns=["idx"])
        preds_df["y_pred"] = y_pred
        OOF_predictions.append( preds_df )

        i += 1
    return OOF_predictions

OOF_predictions = fold_training(regressor, X_train, y_train)
df_pred = pd.concat(OOF_predictions).sort_values(by=["idx"]).set_index("idx")

# using the MAE metric
from sklearn.metrics import mean_absolute_error
print(mean_absolute_error(y, df_pred["y_pred"]))
```

This routine is also useful for saving the out-of-fold (OOF) predictions of the model for later use in a stacking ensemble.

5.3 Nested cross-validation

The inner loop is for model selection where we find the best hyperparameters via a `GridSearchCV`, and the outer loop is for testing generalization using the `cross_val_score`. Thus here the model hyperparameters may vary in each CV fold, unlike in the un-nested CV where we are assessing the performance of one specific set of hyperparameters.

Sample code

```
param_grid  = {"max_depth": [3, 4, 5]}

estimator_GS = GridSearchCV(estimator=DT, param_grid=param_grid, cv=5)
nested_scores = cross_val_score(estimator_GS, X=X, y=y, cv=4)
```

5.4 Data leakage

Data leakage is when, somehow, information from either the target we are predicting or the test dataset become encoded within our model. This can lead an overoptimistic assessment of the model performance. There are two types of leakage:

- **target leakage**: between feature columns and the target column

- **future leakage**: between training rows and test rows

Target leakage is when the information we are predicting also happens to be present in one or more of the features that we are using to predict the target. For example we are predicting the price of a house (€) and we have the features m^2 and €/m^2 in our dataset.

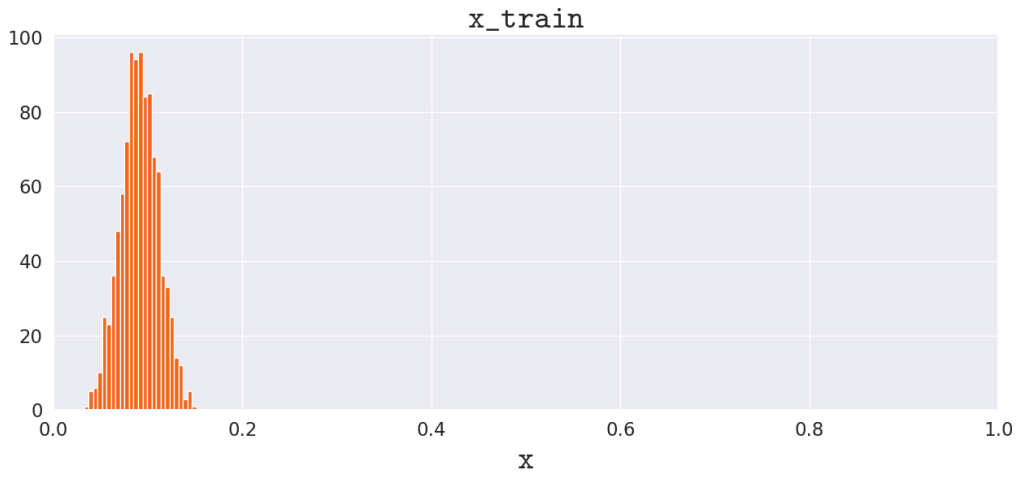


Figure 5.3: Future leakage: `x_train` after min-max scaling the whole dataset that has an outlier in `x_test`.

Future leakage can be caused for example by transformations, such as scaling or mean imputation. If the train, validation and test sets are all i.i.d. then mean imputation should not be too serious (but still should be avoided). Non-robust scaling can be a disaster due to outlier values. In Figure 5.3 we see the effect of the presence of a single outlier value in the test dataset on the training dataset if min-max scaling ($[0, 1]$) is performed on the whole dataset before the train-test split.

How can one avoid data leakage? By processing/cleaning each of our `X_train`, `X_validation`, and `X_test` datasets all in the same way, but each separately. For example, if we impute missing values in a feature using the mean in the `X_train` dataset, then missing values in the `X_validation`, and `X_test` datasets for that feature should be filled with the very same mean value calculated for the `X_train` data.

> **Remark:** The Scikit-learn `cross_val_score` routine does not process the train/test splits separately and thus has the potential to lead to overly optimistic results.

5.5 Covariate shift and Concept drift

Covariate shift $P(X)$: when the `X` in `X_train` are not similar to the `X` in `X_test`.
For example, we modeled houses having 70m^2 to 120m^2 but we are asked to predict for houses in the range 120m^2 to 170m^2

Concept drift $P(y|X)$: when the `y` in `y_train` are not similar to the `y` in `y_test`.
For example we fit our model to house prices that were collected in 1980. The houses themselves (the `X`) have not changed at all since then, but prices today are very different from those in 1980. The concepts that have changed could be for example inflation, market demand, or a singular event such as a currency change from Pesetas to Euros.

Remark: A fundamental assumption in tabular machine learning is that the data we are predicting for has the very same surface that we originally fitted to.

Jupyter Notebook: We can check for covariate shift using adversarial validation. See What is Adversarial Validation? which is based on the Kolmogorov-Smirnov test.

In Figure 5.4 we can see schematically the effect of shift and/or drift over time.

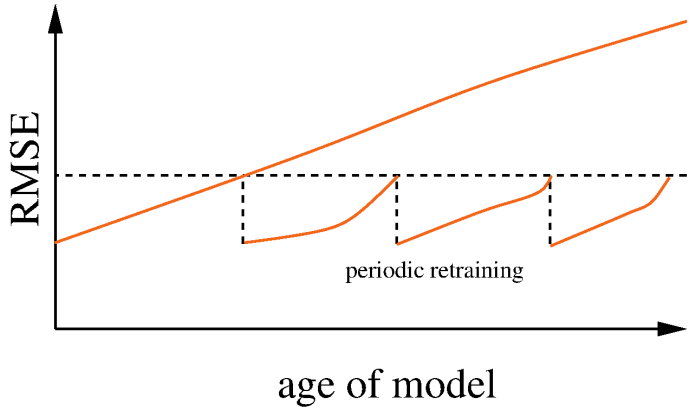


Figure 5.4: Incremental drift and periodic model re-training.

As well as incremental drift other types of drift are sudden drift, gradual drift, and reoccurring concepts[2]. The easiest solution to both shift and drift is refit our model with new data. This can be done either periodically when the model degradation passes an unacceptable limit, or can be done continuously via what is variously known as 'incremental', 'stream' or 'online' learning[3].

[2] Lu, Liu, Dong, Gu, Gama, Zhang "*Learning under Concept Drift: A Review*", IEEE Transactions on Knowledge and Data Engineering **31** pp. 2346-2363 (2018)

[3] Hoi, Sahoo, Lu, Zhao "*Online Learning: A Comprehensive Survey*" arXiv:1802.02871 (2018)

Recommended reading

Papers

- Stephen Bates, Trevor Hastie, Robert Tibshirani *"Cross-validation: what does it estimate and how well does it do it?"*, arXiv:2104.00673 (2022)
- Sudhir Varma, Richard Simon *"Bias in error estimation when using cross-validation for model selection"*, BMC Bioinformatics **7** 91 (2006)

Other

- Scikit-learn: *"Cross-validation: evaluating estimator performance"*

6. Regression

All models are wrong, but some are useful[1].

George E. P. Box

A regression problem in one in which the target values belong to the set of real numbers ($y \in \mathbb{R}$). From a machine learning standpoint our task is to produce point predictions (expectation values) and/or prediction intervals for previously unseen data.

In this chapter we shall look at the very simplest statistical (parametric) model, namely linear regression. We shall also look at the very simplest machine learning (non-parametric) estimator, namely the decision tree. We shall also see quantile regression (which includes median regression) and calculate prediction intervals using conformal prediction.

6.1 Regression baseline model

Our very first model should *always* be the baseline model. The baseline model is simply the expectation value. This is a model 'no-model' in that we make no use at all of the $\mathbf{X}$ values, only the target column, y.

Equation: Baseline model

$$\hat{y} = \mathbb{E}[f(x)] = \bar{y} \;\; \forall \, \hat{y} \tag{6.1}$$

What does the baseline model look like?

[1] George E. P. Box *"Science and Statistics"*, Journal of the American Statistical Association **71** pp. 791-799 (1974)

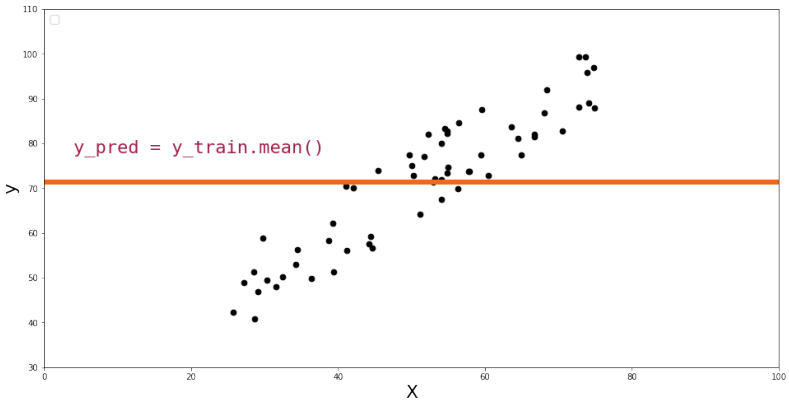


Figure 6.1: A regression baseline model.

This model embodies the notion that there is no learnable signal in any of the features in the data; it is all useless noise. There is no way to create a model that performs worse than this one. We then calculate our metric of choice for this model and use this to gauge our progress.

6.2 Univariate linear regression

Let us create a model that now has two degrees of freedom; positional and orientational. We can shift our straight line up or down by a a parameter that we shall call β_0, and we can rotate the line so as to have an inclination, β_1.

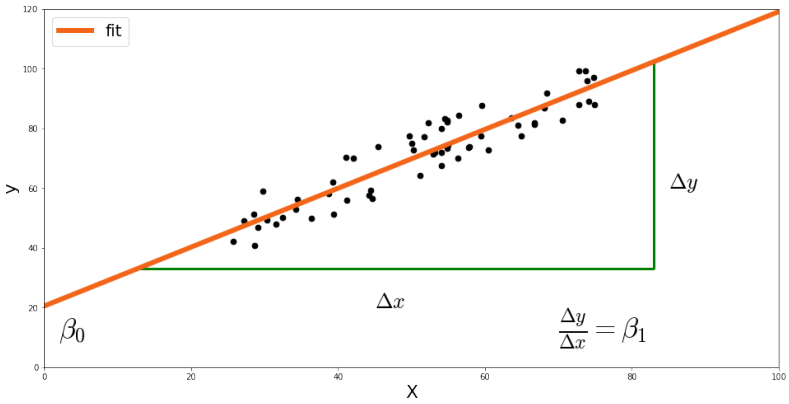


Figure 6.2: A straight line fit (orange) to the datapoints (black).

Equation: Linear regression model

$$\hat{y} = \mathbb{E}[y|\mathbf{X}] = \beta_1 x + \beta_0 \tag{6.2}$$

where β_1 is the *gradient*, and β_0 is the *bias* term.

For a dataset with multiple features the value of the bias (β_0) corresponds to the situation when the value of every feature is zero, and every feature, x_n, will have its own corresponding β_{1n} value.

Remark: This is known by statisticians as a *parametric* model: the functional form (here a straight line) is fixed, and we are only allowed to play with the parameters (β_1 and β_0). However, it is called parametric not because of β_1 and β_0, but because in the context of statistical modeling it is assumed that the underlying distribution of the data originates from a parametric distribution, for example the Gaussian distribution, which has the parameters μ and σ.

6.3 Calculating β_1 and β_0

6.3.1 Ordinary least squares

Equation: One can calculate β_1 and β_0 using ordinary least squares, for example:

$$\beta_1 = \frac{\sum((x_i - \bar{x})(y_i - \bar{y}))}{\sum(x_i - \bar{x})^2} \tag{6.3}$$

and then

$$\beta_0 = \bar{y} - \beta_1 \bar{x} \tag{6.4}$$

Sample code

```
import numpy as np

def OLS(x, y):
    b1 = (np.sum((x - np.mean(x))*(y - np.mean(y))))/ \
         (np.sum((x - np.mean(x))**2))
    b0 = np.mean(y) - (b1*np.mean(x))
    return b1, b0

b1, b0 = OLS(x, y)
```

6.3.2 Normal equation

We can envisage our `DataFrame` as if it were a system of linear equations where the variables are the β_{1n} for each feature n, and x_{mn} in our dataframe represent the coefficients.

$$\begin{cases} \beta_1.x_{0,1} + \beta_0.1 = y_0 \\ \beta_1.x_{1,1} + \beta_0.1 = y_1 \\ \beta_1.x_{2,1} + \beta_0.1 = y_2 \end{cases} \tag{6.5}$$

Our task is to find the β_0 (note the insertion of a constant column, which for example can be done using the statsmodels `add_constant`) and the β_1 associated with each column. In matrix notation:

$$\mathbf{X} = \begin{bmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{6.6}$$

Remark: $\mathbf{X}$ is sometimes known as the design matrix

Multiple linear regression, now with n features:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{01} & x_{02} & \cdots & x_{0n} \\ 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{6.7}$$

which can be written more compactly as

$$\mathbf{X}\beta = \mathbf{y} \tag{6.8}$$

In python this is written as `numpy.dot`:
`y = np.dot(X,beta)`
In linear algebra this is known as the *normal equation*, and can be re-written as

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{6.9}$$

where $(\mathbf{X}^T\mathbf{X})$ is known as the *Gram* or *normal* matrix.

Sample code

```
import numpy as np

X = np.array([x, np.ones(len(x))]).T
beta_1, beta_0 = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
```

6.3.3 Scikit-learn `LinearRegression`

Using the scikit-learn `LinearRegression` estimator

Sample code

```
from sklearn.linear_model import LinearRegression

regressor = LinearRegression()
regressor.fit(X_train, y_train)
y_pred = regressor.predict(X_validation)
```

Scikit-learn can output the parameters

```
beta_1 = regressor.coef_
beta_0 = regressor.intercept_
```

6.4 Assumptions of linear regression

No assumptions at all are required in order to perform linear regression just as long as there is more than one datapoint. However, suffice to say the linear model is most effectively used when there is a linear relationship between the independent variable (x) and the dependent variable (y), the target column. To get an idea as to whether is is indeed the case or not, when performing EDA one can use the `seaborn.lmplot` which will overlay a linear fit to the data, along with the confidence interval (`ci`) associated with the parameters.

```
sns.lmplot(x=X, y=y, data=data, ci=95)
```

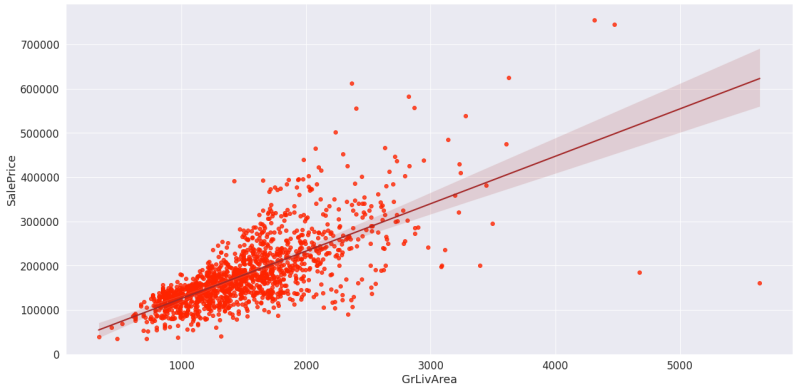


Figure 6.3: Example seaborn lmplot.

Furthermore, for the regression coefficients and confidence intervals to be reliable and comparable the errors should be:

- **i.i.d**: every error point is independent of the other errors, *i.e.* no autocorrelation (one can test this using the Durbin-Watson statistic)
- **have constant variance**: $\sigma^2 \neq f(x)$. This is known as homoskedasticity.
- **have a Gaussian distribution** *i.e.* $\forall\, i \in n, \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$.
- $\mathbb{E}(\varepsilon) = 0$

That said, the importance of the errors being Gaussian is sometimes over-exaggerated[2].

Remark: For small datasets that do not meet these conditions the `TheilSenRegressor` presents an interesting alternative.

[2] Knief, Forstmeier "*Violating the normality assumption may be the lesser of two evils*", Behavior Research Methods **53** pp. 2576-2590 (2021)

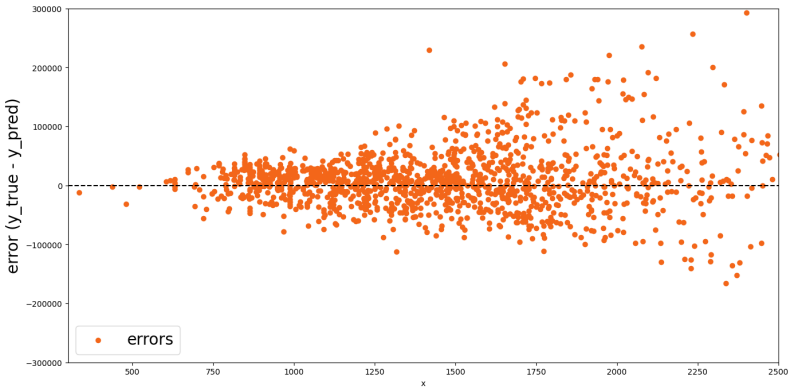


Figure 6.4: Error plot: An example of heteroscedasticity.

Sample code

One can create an error plot using

```
errors = y_true - y_pred

fig = plt.figure(figsize=(16, 8))
plt.scatter(x, errors, s=35, label='errors')
plt.axhline(y = 0, color = 'black', linestyle = 'dashed')
plt.legend(loc='lower left', fontsize=20)
plt.xlabel('x')
plt.ylabel('error (y_true - y_pred)', fontsize=20);
```

Remark: A possible solution for heteroscedasticity is to transform the dependent variable (y) by taking the logarithm or square root. However, it is not a good idea to transform the target values if there is more than one feature, which in machine learning is basically always the case.

6.5 Polynomial regression

It is not at all unusual to have data to which a curve would be a much better fit than a straight line (remember dataset B of Anscombe's quartet).

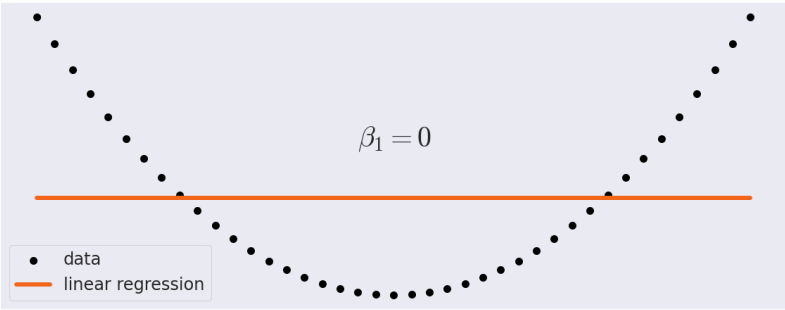


Figure 6.5: A linear fit to a parabola.

We can incorporate curvature by way of adding more terms to the polynomial:

$$\hat{y}(\mathbf{X}) = \beta_2 \mathbf{x}^2 + \beta_1 \mathbf{x} + \beta_0 \tag{6.10}$$

for example in Pandas via
`df["x_squared"] = df["x"]**2`
this new feature has allowed us to incorporate curvature into our model.

Remark: This is still a linear function in terms of the β parameters; to the estimator the x^2 term is simply another feature column.

One can use the scikit-learn `PolynomialFeatures` to automatically create the interaction features when one is performing multiple linear regression. For example if we have the features `a`, `b`, and `c` then for degree 2 this will also produce the new features a^2, b^2, and c^2 as well as the ab, ac, and bc interaction features.

6.6 Extrapolation

John von Neumann quipped *"...with four parameters I can fit an elephant, and with five I can make him wiggle his trunk."*. In Figure 6.6 we fit polynomials of increasing degree to a dataset. We can see that they all do rather well as long as there is data to fit to. However, as soon as there is no data their tails are pretty much free to do as they like. Such extrapolations make for extremely bad predictions so we wish to avoid extrapolation whenever we can.

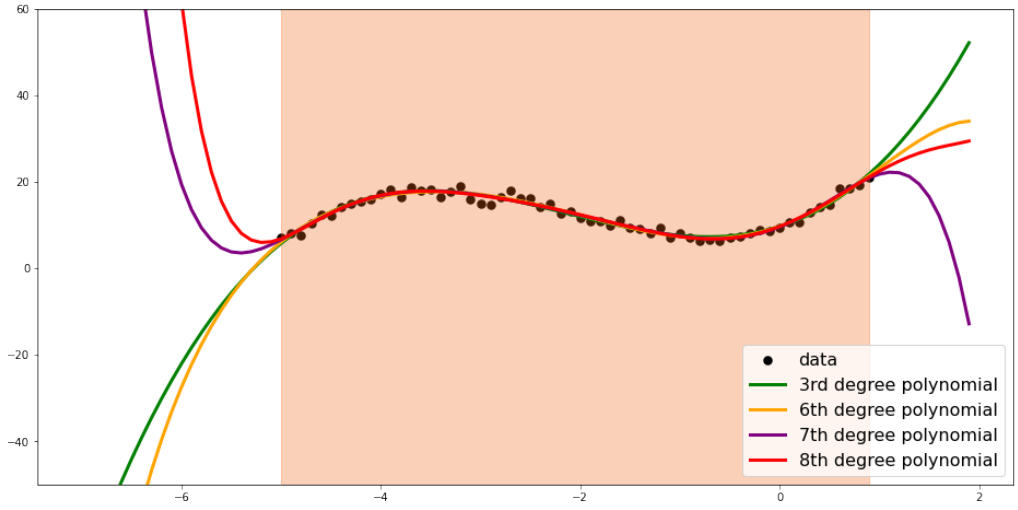


Figure 6.6: The danger of extrapolation

Remark: Broadly speaking interpolation is when we make predictions within the convex hull of our training data, and extrapolation occurs when we make predictions outside of the convex hull of our training data.

6.6.1 Convex hull

When it comes to making predictions one cannot filter out outliers; one does not simply say *'I am not going to produce a prediction for that particular point'*.

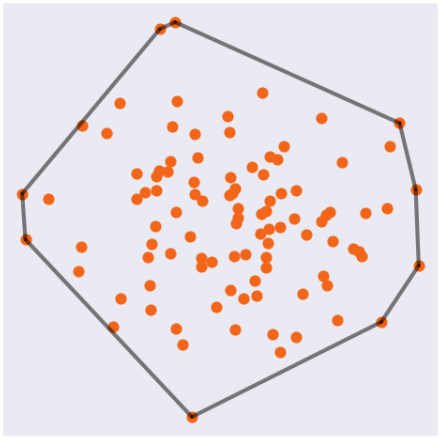


Figure 6.7: The 2D convex hull (black lines) for a set of points.

Thus in order to avoid having to extrapolate it is desirable that the convex hull of the training data at least spans the very same volume as the convex hull of the test data. For this reason it is convenient in machine learning to keep infrequent extreme values in the training dataset as they serve to expand the convex hull. This space will be very sparse, but this is still preferable to letting the estimator extrapolate if it encounters extreme values in the test data (see also Section 8.3.1).

6.7 Explainability

The regression coefficients β_1, that is to say the gradients, provide very valuable information regarding the relative importance (weight) of each feature in the model. Furthermore, if a feature has $\beta_1 \approx 0$ this feature has very little role in the model and basically no influence in predicting $\hat{y}$. Note that for a correct interpretation of the relative importance of the regression coefficients it is important that each of the features have the same scale and in multiple linear regression will only strictly hold of none of the variables are correlated. One can also calculate the confidence intervals (CI) associated with the coefficients with `statsmodels.regression.linear_model.OLS` using `conf_int`.

We can view the regression coefficients (weights) using ELI5:

```
import eli5

feature_names = X_train.columns.tolist()
eli5.show_weights(regressor, top=None, feature_names= feature_names)
```

which gives us

Weight?	Feature
+390135.335	m2_edificados
+378177.833	<BIAS>
+355886.998	precio_zona
+63058.702	exterior
+16381.696	orientación_norte
+4314.874	planta
+3809.496	año_construción
+1.025	precio_aparcamiento
-595.693	necesita_reforma
-4027.583	orientación_sur
-4077.580	n_baños
-6322.717	n_habitaciones
-7899.945	m2_útiles
-14058.596	orientación_este
-14204.718	orientación_oeste
-24590.137	aparcamiento
-26983.654	nueva_construcción
-80908.585	ascensor
-118913.261	precio_alquiler

Figure 6.8: The weights (β_1) for each feature and the bias (β_0) for a multiple linear regression.

where BIAS corresponds to β_0.

6.8 The loss and cost functions

How does a machine learning estimator find the optimal parameters? We shall introduce the *loss* (L) and *cost* (J) functions.

- *loss* is a measure of distance[3] between the two points $\hat{y}$ and y
- *cost* is the average of all of the losses: $\bar{L}$

Remark: Sometimes the loss function is known as the *error function* and the cost is known as the *objective function*.

[3] See the p-norm and Lebesgue spaces.

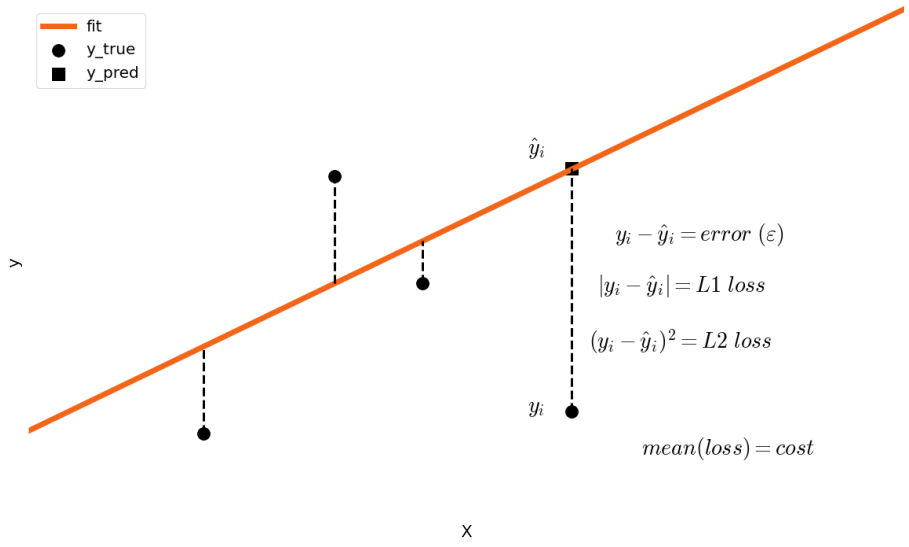


Figure 6.9: Fitting errors.

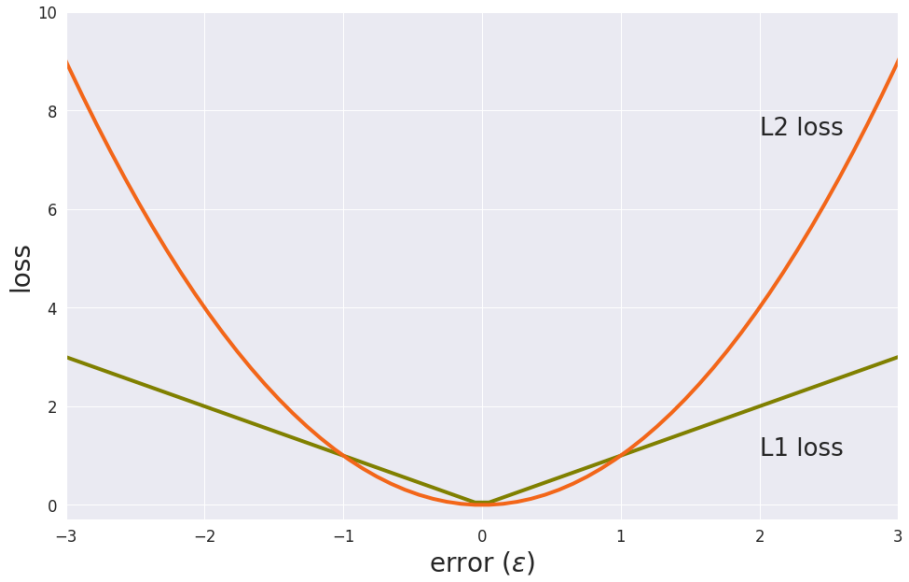


Figure 6.10: The $L1$ and $L2$ loss functions.

As code we have $y \rightarrow$ `y_true` and $\hat{y} \rightarrow$ `y_pred`.

Equation: ε (error) = `(y_true - y_pred)`

- $L1$ loss = AE (absolute error) = `abs(y_true - y_pred)`
- $L1$ cost = MAE (mean absolute error) = $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- $L2$ loss = SE (squared error) = `(y_true - y_pred)**2`
- $L2$ cost = MSE (mean squared error) = $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

Suffice to say the optimal solution is found when the cost function is at its minimum. For example, when performing linear regression using ordinary least squares, as the very name

suggests, we use the $L2$ loss function, so we wish to minimize the $L2$ cost function where:

$$\begin{aligned} J(\beta) &= MSE \\ &= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \\ J(\beta_1, \beta_0) &= \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_1 \mathbf{X}_i + \beta_0)^2 \end{aligned}$$

The optimal values of β_1 and β_0 are those associated with $\text{argmin}(J)$

Remark: If we use the $L1$ loss we are performing *median* regression, which is more robust than the standard mean regression (see also quantile regression).

What form does the $L1$ cost take for two points?

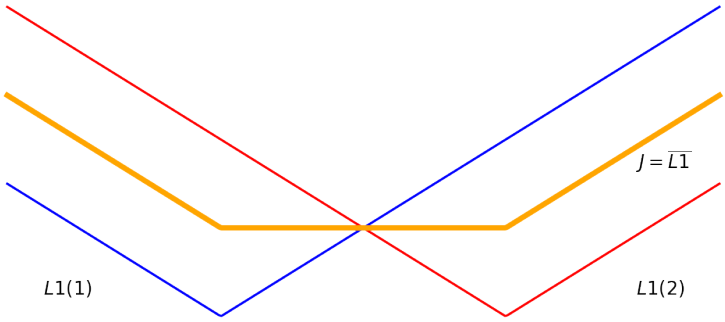


Figure 6.11: The $L1$ cost (orange) for two data points.

In Figure 6.11 we can see the $L1$ cost function (orange) for two datapoints (blue and red). We can see that the resulting cost is piecewise continuous, and may not have a unique minima, hence the optimal parameters are not unique.

Remark: The mean absolute error is a new member of the collection of objectives in XGBoost (October 2022)

What form does the $L2$ cost take for two points?

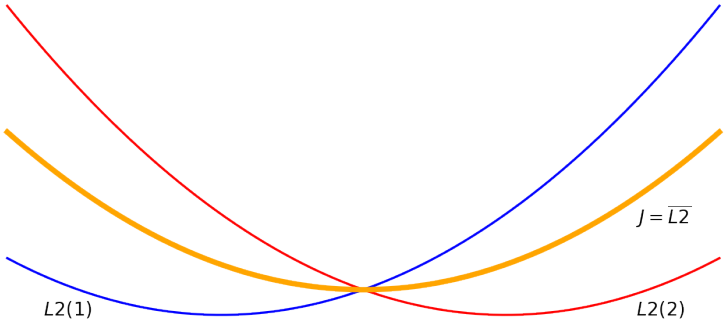


Figure 6.12: The $L2$ cost (orange) for two data points.

In Figure 6.12 we can see that for the $L2$ cost (orange) is a smooth continuous convex function, having a unique minima.

6.8.1 Gradient descent

What does the cost function for the $L2$ loss look like for two features? The cost function is now two-dimensional. For linear regression with one feature the function was a parabola. In two dimensions we now have a surface, known as a paraboloid.

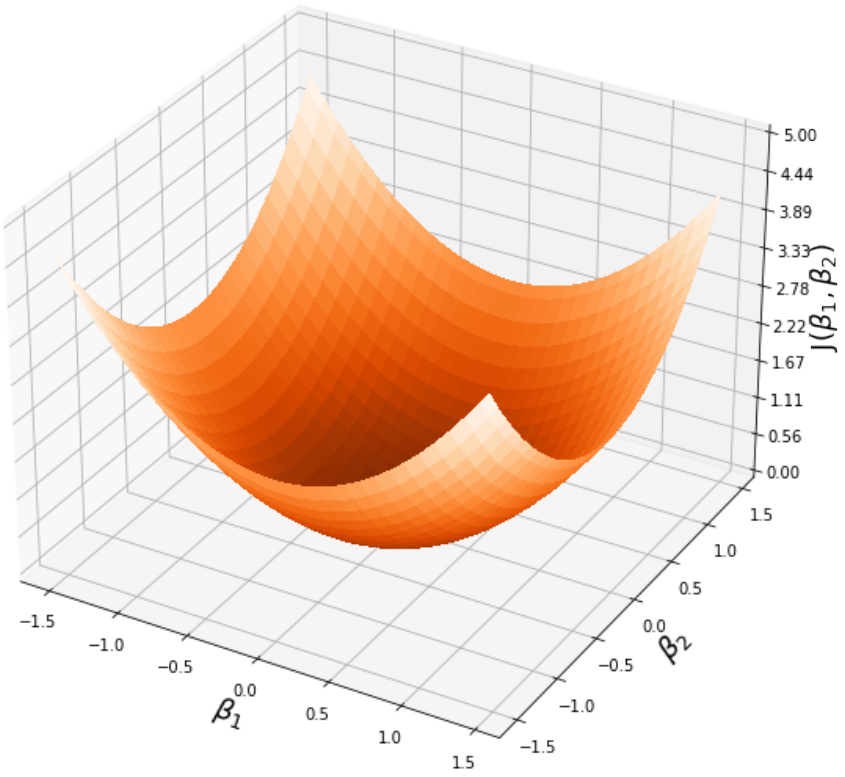


Figure 6.13: A 2-dimensional parabolic surface (paraboloid).

Here we introduce a very powerful and general numerical technique for finding local minima called gradient descent[4].

Remark: Any local minimum of a convex function is also a global minimum.

The gradient (G) of a surface (J) is given by

$$G = \frac{\partial J(\beta)}{\partial \beta} \tag{6.11}$$

where β are the parameters of our fit.

[4] Sebastian Ruder *"An overview of gradient descent optimization algorithms"*, arXiv:1609.04747 (2017)

Equation: Gradient descent *update rule*

$$\beta \leftarrow \beta - \alpha \frac{\partial J(\beta)}{\partial \beta} \tag{6.12}$$

where α is the *learning rate* parameter.

Our objective is to find the position if the global minimum of the cost surface (although there is no guarantee of being able to do this for a complicated surface, and we may end up in a local minimum).
Algorithm: First we choose a random point on the surface to start form, then:

1. calculate the gradient $(\partial J/\partial \beta)$
2. multiply by -1 to change the sign
3. scale by α
4. this becomes our new value for β
5. repeat process until β converges to within a specified tolerance (for example $\Delta J < 0.001$).

In Figure 6.14 we can see the effect of the learning rate hyperparameter. If the value of α is small we take many small steps towards the minima. We can speed up the process by choosing a larger value of α, but if it is too large than we can end up 'bouncing' from side to side of the surface.

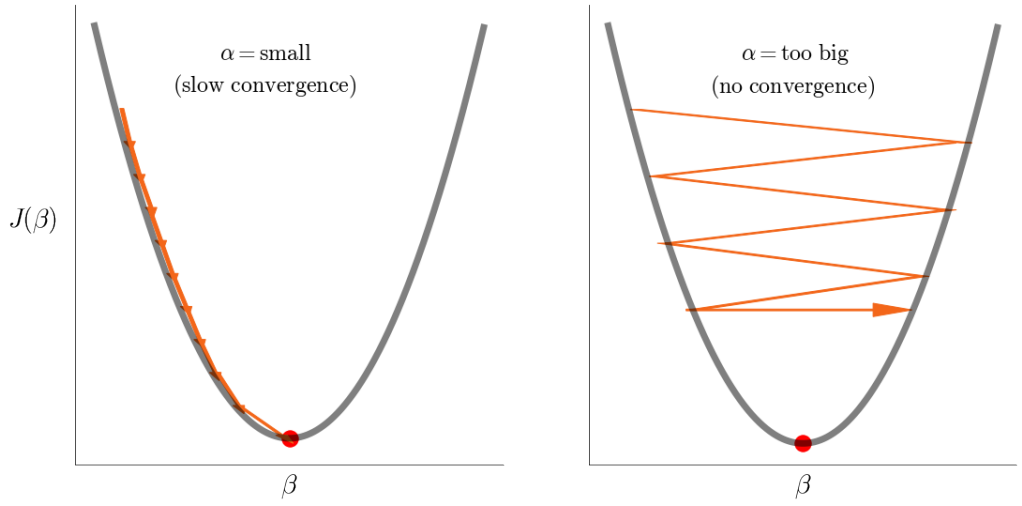


Figure 6.14: Effect of the learning rate hyperparameter

We can assess our choice of α by plotting the learning rate (Figure 6.15), observing the reduction in the cost function w.r.t. iteration. In the upper plot our choice of α was too small leading to slow convergence. In the lower figure we have a much better choice of α leading to faster convergence. Note however that if we become too enthusiastic and choose a very large learning rate the cost function can diverge or 'explode'; we overshoot the other side of the function and start climbing up until the computer has a numerical overflow.

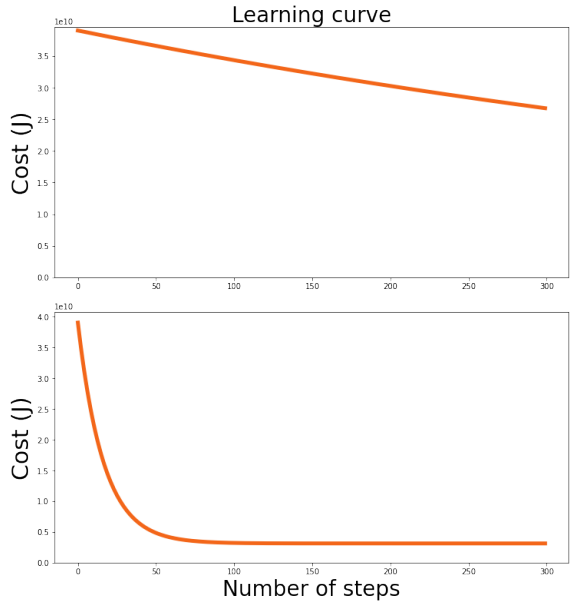


Figure 6.15: Different learning rates (α). In the upper figure the choice of learning rate is too conservative, in the lower figure we have a good learning rate.

We can use gradient descent for linear regression; the cost function for the MSE is

$$J(\beta) = \text{MSE} = \frac{1}{n}\left(\varepsilon^T \varepsilon\right) = \frac{1}{n}\left(\hat{\mathbf{y}} - \mathbf{y}\right)^T \left(\hat{\mathbf{y}} - \mathbf{y}\right) \tag{6.13}$$

$$= \frac{1}{n}\left(\beta\mathbf{X} - \mathbf{y}\right)^T \left(\beta\mathbf{X} - \mathbf{y}\right) \tag{6.14}$$

leading to

$$\frac{\partial J(\beta)}{\partial \beta} = \frac{1}{n}\left(\left(\beta\mathbf{X} - \mathbf{y}\right)\mathbf{X}\right) \tag{6.15}$$

giving

$$\beta \leftarrow \beta - \frac{\alpha}{n}\left(\varepsilon\mathbf{X}\right) \tag{6.16}$$

Sample code

```
X = np.array([z, np.ones(len(z))]).T
alpha        = 0.01 # size of step
n_iterations =  300 # number of steps
def gradient_descent(X, y_true, alpha, n_iterations):
    n = y_true.size
    beta = np.random.rand(2).T # random starting point for the beta
    historial = [beta] # save the betas
    costs     = []     # save the costs

    for i in range(n_iterations):
        y_pred = np.dot(X,beta)
```

```
        error = (y_pred - y_true)
        # cost = error cuadrático medio (MSE)
        cost  = np.sum(error**2)/n
        costs.append(cost)
        gradient = ((error).dot(X))/n
        # actualizar los beta
        beta = beta - alpha * gradient
        # guardar los beta
        historial.append(beta)
    return historial, costs
historial, costs = gradient_descent(X, y, alpha, n_iterations)
# los betas finales
beta = historial[-1]

dataset["y_pred"] = beta[0]*X[:,0] + beta[1]
# un-standardize x
x_sin_standardize  = (X[:,0]*x_std)+x_mean
```

In Figure 6.16 we have an example where the two features have very different scales. This makes life difficult for our gradient descent algorithm; a good value of alpha for the 'narrow' feature will will take a long time to explore the 'wide' feature. On the other hand a good alpha for the wide feature could be a disaster when exploring the narrow feature, leading to divergence. The solution to this is to scale the features (see the section Feature scaling: standardization and normalization in Chapter 4) beforehand. There are routines such as `AdaGrad` that can use an adaptive learning rate.

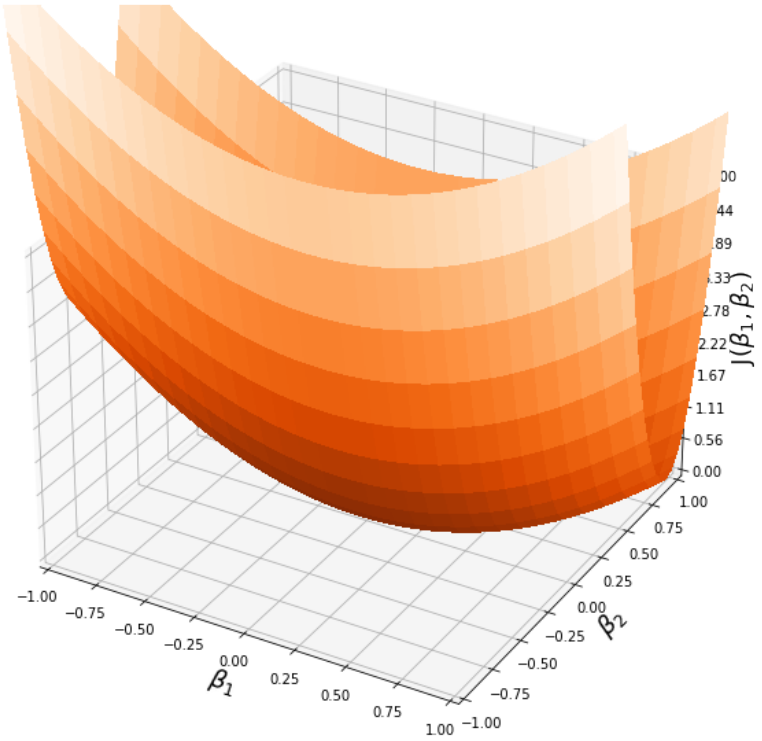


Figure 6.16: Two features with two very different scales.

Although there are analytical solutions for a simple surface such as that of the linear regression cost function, in Figure 6.17 we see an example loss surface of a neural network, where suffice to say no analytical solution exists and we have to use numerical methods.

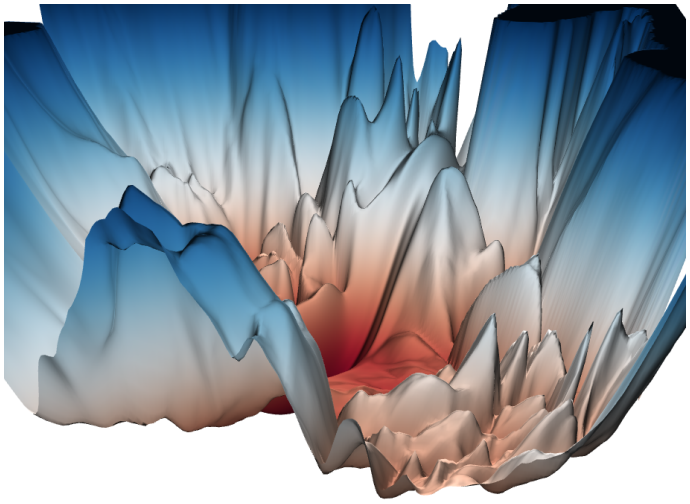


Figure 6.17: (Image credit: Li, Xu, Taylor, Studer, Goldstein *"Visualizing the Loss Landscape of Neural Nets"*, arXiv:1712.09913 (2018))

6.9 Metrics

How well did our model do? One way to visualize the performance of our model is to create a scatter plot the predicted values against the true values. If our predictions were perfect all of the points would lie on the $y = x$ line. Points above the line represent cases where we overestimate, and points below the line represent underestimates.

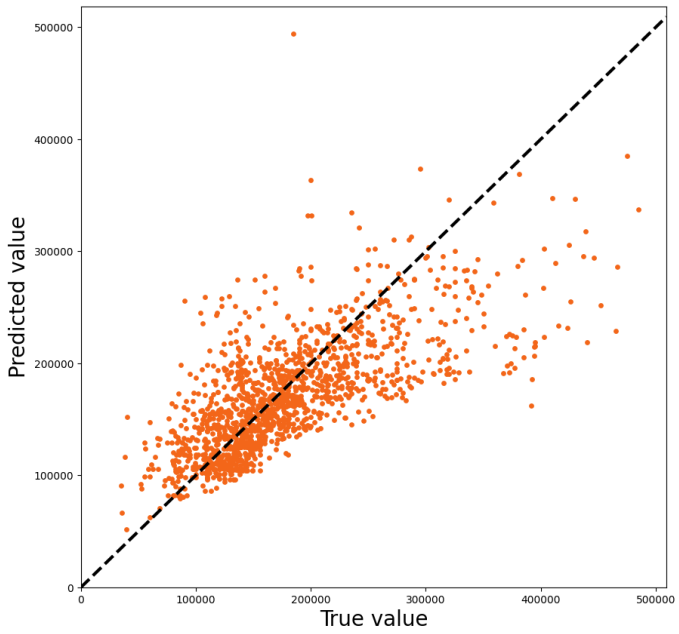


Figure 6.18: True value vs. predicted value plot.

For example, in Figure 6.18 we can see that for smaller predictions we have a similar dispersion of points either side of the $y = x$ line, which is to be expected. However, for larger values we see that in this example we always underestimate the true value, indicating that there is something missing in our model (and probably in our dataset) in this region.

Sample code

```
from sklearn.metrics import PredictionErrorDisplay

display = PredictionErrorDisplay(y_true=y_true, y_pred=y_pred)
display.plot()
```

Another interesting plot is a histogram of the prediction errors. Ideally the histogram should be symmetric, narrow and centred on our prediction (expectation value). The broader the histogram the greater the errors, and if it is strongly skewed this could indicate underfitting.

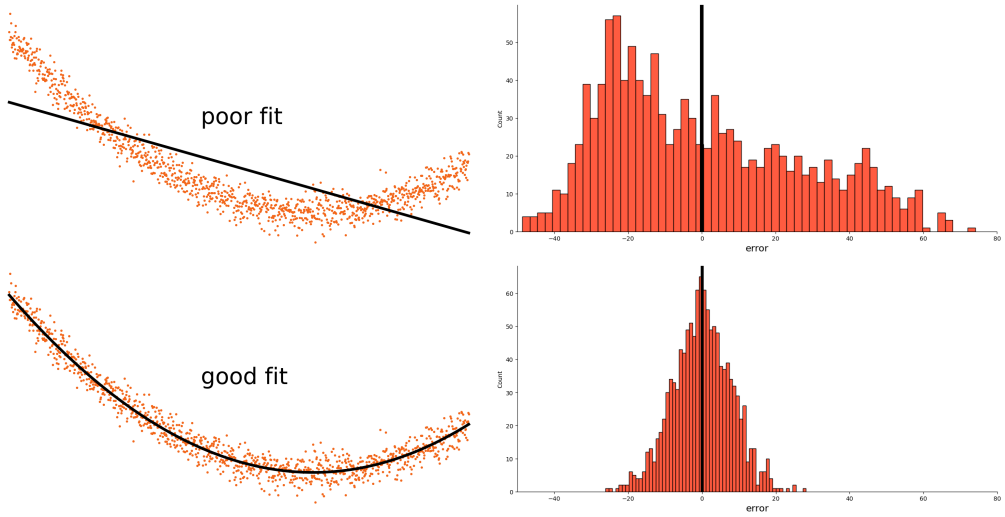


Figure 6.19: Histogram of our prediction errors.

That said, what we would really like is a single numerical value that indicates our overall performance.

6.9.1 Root mean square error (RMSE)

The most natural and logical metric to follow if we are using the $L2$ loss function is the root mean squared error (RMSE).

Equation: Root mean square error

$$\text{RMSE} := \sqrt{\overline{\varepsilon^2}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{6.17}$$

Observe that the MSE looks similar to the formula for the variance. However, here we are measuring distances w.r.t. the true value (`y_true`) whereas the variance measures

distances w.r.t. the mean of the values themselves, $\overline{y_i}$. This leads to what is known as the bias-variance decomposition

$$\text{MSE} = \text{variance} + \text{bias}^2 \tag{6.18}$$

Remark: A machine learning metric is not the same thing as a business key performance indicator (KPI) for example customer churn rate or growth in revenue, and optimizing the machine learning metric is not the same thing as optimizing a KPI. Often it is not easy at all to connect the two.

Sample code

As of scikit-learn version 1.4 there is now:

```
from sklearn.metrics import root_mean_squared_error

RMSE = root_mean_squared_error(y_true, y_pred)
```

and prior to version 1.4:

```
from sklearn.metrics import mean_squared_error

RMSE = mean_squared_error(y_true, y_pred, squared=False)
```

6.9.2 Mean absolute error (MAE)

If we are using the $L1$ loss function the natural metric is the mean absolute error (MAE).

Equation: The mean absolute error (MAE) is given by:

$$\text{MAE} := \frac{1}{n}\sum|\varepsilon| = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{6.19}$$

Sample code

```
from sklearn.metrics import mean_absolute_error

MAE = mean_absolute_error(y_true, y_pred)
```

6.9.3 The R^2 metric

A popular regression metric amongst statisticians for comparing explanatory models, although not so much amongst machine learning practitioners, is the R^2 metric, also known as the coefficient of determination.

Equation: The R^2 score is given by:

$$R^2 := 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y}_i)^2} \tag{6.20}$$

The mean baseline model has an R^2 score of zero, and the perfect fit (perfect overfitting) has a score of 1.

Sample code

```
from sklearn.metrics import r2_score

R2 = r2_score(y_true, y_pred)
```

6.10 Decision tree regressor

The linear regression model has the virtue of being very easy to interpret by way of the β parameters associated with each feature. However, it only works well if the relationship between each feature (x) and y is linear, which is frequently not the case. Let us take for example the following toy dataset in Table 6.1:

X	y_true
1	1
2	1
4	2
5	2

Table 6.1: Toy dataset.

In Figure 6.20 the uppermost plot represents the mean baseline model.

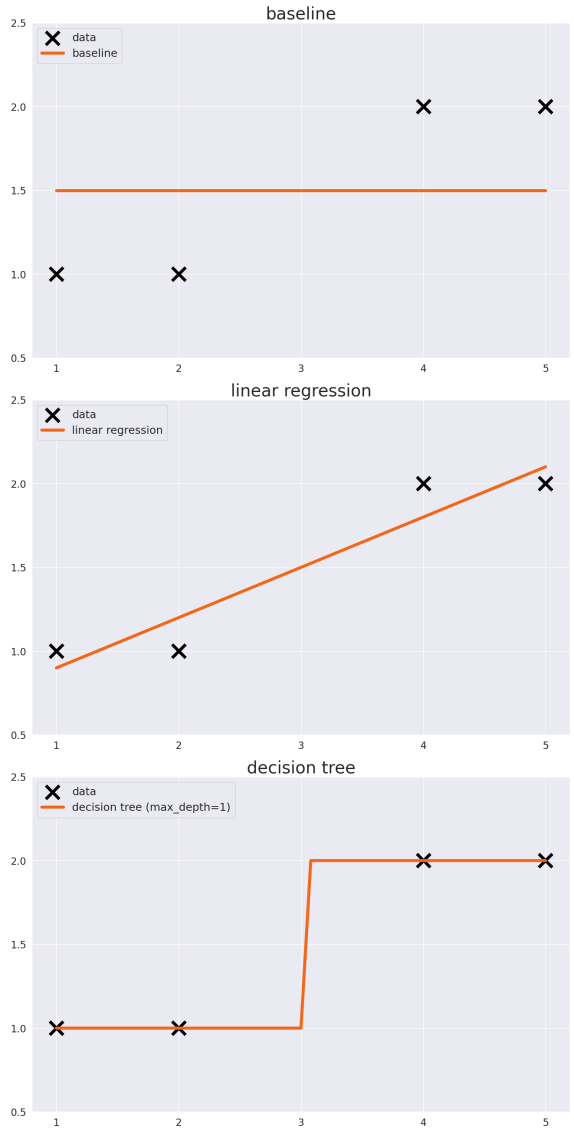


Figure 6.20: Baseline, linear regression, and decision tree predictions

In the middle plot we can easily see that it is not possible for a straight line to pass through all four points simultaneously; thus such a fit will have a non-zero error.

model	**RMSE**
Baseline	0.5
Linear regression	0.158
Decision tree	0

Table 6.2: Toy dataset RMSE results.

However, if we judiciously split the dataset into two sections, and fit a mean baseline model to either side, we can obtain a perfect fit. This is the essence of the Decision Tree regressor. Decision Trees use recursive binary splitting to divide up the training data into *leaves*. The value of each leaf is the mean of the data points within it. The decision tree stops splitting when a specified depth (`max_depth`) or specified number of data points

per leaf has been (or will be) reached (`min_samples_leaf`). Here is a graph of the final decision tree model for the aforementioned dataset:

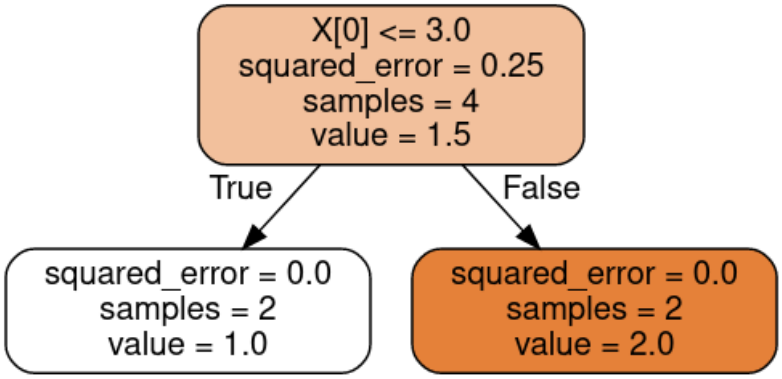


Figure 6.21: graphviz of our decision tree regressor

Sample code for the graph

```
import graphviz
from sklearn.tree import export_graphviz

dot_data = export_graphviz(regressor,
                           filled=True,
                           rounded=True,
                           proportion=False)
graphviz.Source(dot_data)
```

where we can see that the initial squared error (*i.e.* the $L2$ loss) is 0.25, containing 4 data points, with a mean value of 1.5. Upon splitting at $X \leq 3$ two leaves are produced, both having zero error, containing two data points, the left hand leaf having a mean of 1, and the right hand leaf a mean value of 2. Indeed we can even describe our model in two sentences:

- if x is less than 3, $\hat{y}$ is 1
- if x is greater than 3, $\hat{y}$ is 2

Remark: A decision tree of `max_depth=1` is also known as a 'decision stump'

The position of the binary split is chosen so as to minimize the cost, *i.e.* the total of the squared error ($L2$ loss). **Example**: For our toy dataset making the split between the first and second datapoints (*i.e.* at $X = 1.5$) would lead to the LHS split having a squared error of 0 and for the RHS we first create the mean baseline $(1+2+2)/3 = 1.66$, then calculate the total error for the points w.r.t. this baseline $(1-1.66)^2 + (2-1.66)^2 + (2-1.66)^2 = 0.66$. If we now split between the second and third datapoint (*i.e.* at $X = 3$) both sides have a squared error of zero, and finally If we split between the second and third datapoint (*i.e.* at $X = 4.5$) we have a LHS baseline of $(1+1+2)/3 = 1.33$ and squared error of 0.66. Thus we find the optimal split is located at $X = 3$.
Say we have a two feature dataset. In this case first we split on feature 1 and calculate the sum of the squared errors (ΣSE) where ΣSE = SE(LHS) + SE(RHS), then we split on feature 2 and also calculate the corresponding ΣSE. Whichever feature has the lowest ΣSE is the is the split that is kept in the tree.

Feature scaling

From this example it is clear that for this non-parametric estimator the solver (binary splitting) does not require that the features be scaled.

Hyperparameters

Unlike the linear regressor we now have what are known as hyperparameters. Linear regression has no hyper-parameters:

```
regressor = LinearRegression()
```

but now our decision tree model has two very important hyperparameters

```
regressor = DecisionTreeRegressor(max_depth=5, min_samples_leaf=1)
```

6.10.1 Hyperparameter: `max_depth`

By far the most influential hyperparameter for a decision tree is its depth – although height would perhaps be a more apt name for a tree... In Figure 6.22 we see fits of a decision tree of `max_depth=1` and `max_depth=2` to the same dataset

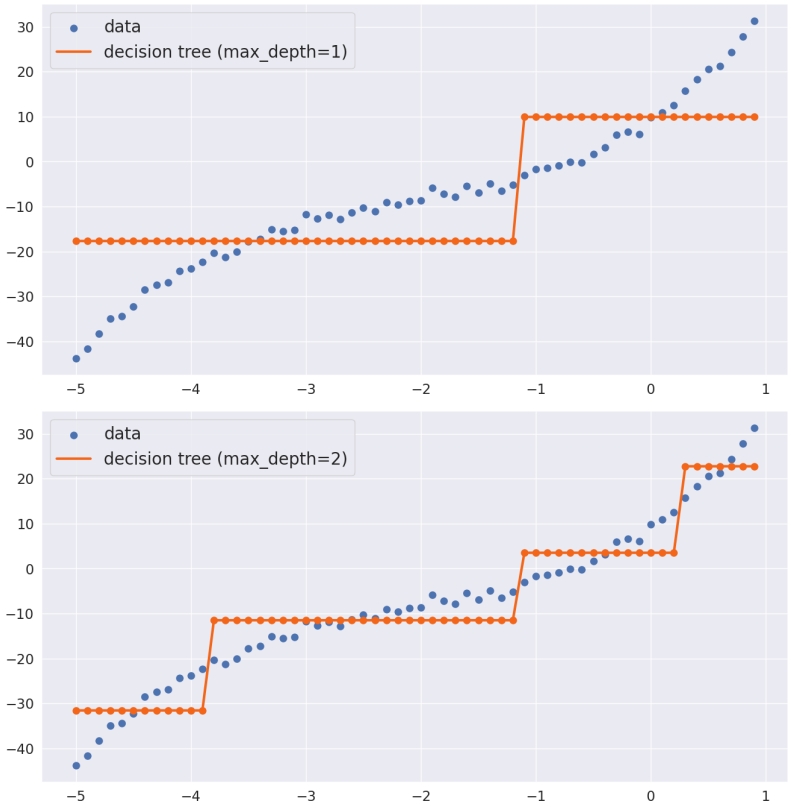


Figure 6.22: Decision trees (orange) for `max_depth=1` and `max_depth=2`.

Here we can see the tree in action; at depth 1 the tree splits the data into two plateaux then, for depth 2 the the position of the initial split is maintained and the tree then independently splits the left hand side plateau again into two, and the right hand side plateau into two, creating a total of four plateaux which correspond to the four leaves of the tree.

Remark: Although the depth=1 split may be optimal, there may be a more optimal split for depth=2 by not maintaining the position of the first split. The decision tree is an example of a greedy algorithm.

As we increase the depth the number of leaves has the potential to grow exponentially. Eventually we will have the capacity to assign a leaf to each and every data-point, thus fitting any dataset perfectly with zero error:

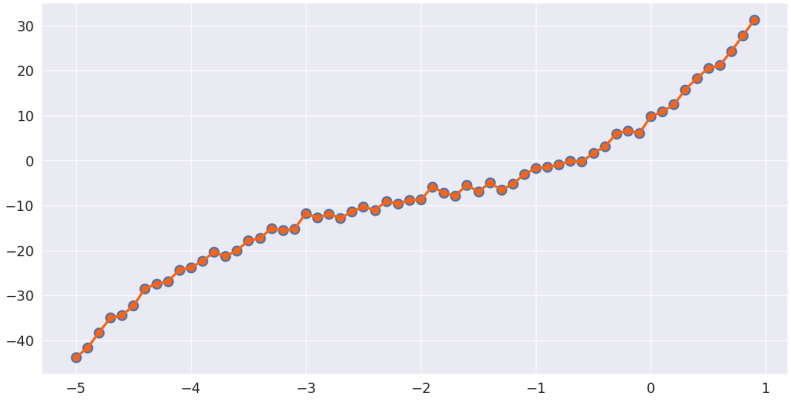


Figure 6.23: A perfect fit to the training data.

This seems fantastic, but is actually a complete disaster! Let us plot the RMSE metric with respect to the tree depth for our fit to the training data, and for a test dataset having the same underlying signal, but having different random noise.

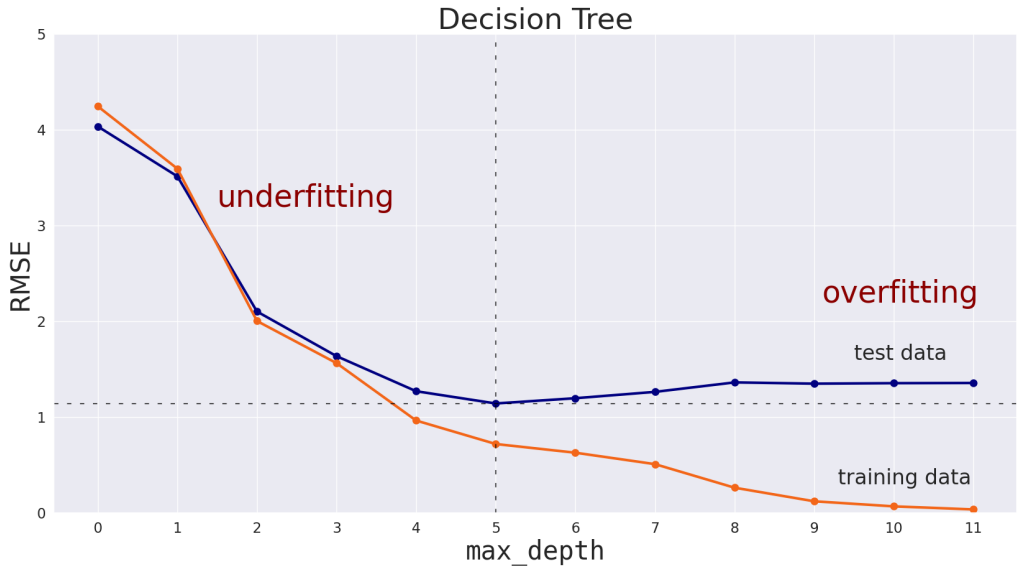


Figure 6.24: Learning curve as a function of `max_depth`

The important things in Figure 6.24 are that, with sufficient depth, a decision tree can perfectly fit any training dataset; eventually every single datapoint will have its very own leaf with each leaf having zero error. However, on the unseen test data (in blue) we see that, in the above example, the best performance (lowest test RMSE) was achieved at `max_depth=5`, and indeed the results even become very slightly worse for deeper trees

as the models fitted to more and more of the noise. The greater the variance of the noise component, the worse the fit will become. In this example beyond `max_depth=5` we start what is known as overfitting.

6.11 Overfitting

Overfitting can be understood as when we have fitted to all of the signal, and we now start fitting to the noise component. Our model has too much complexity or flexibility.

Remark: If it were not for overfitting then machine learning would be trivial; simply fit a deep decision tree to everything!

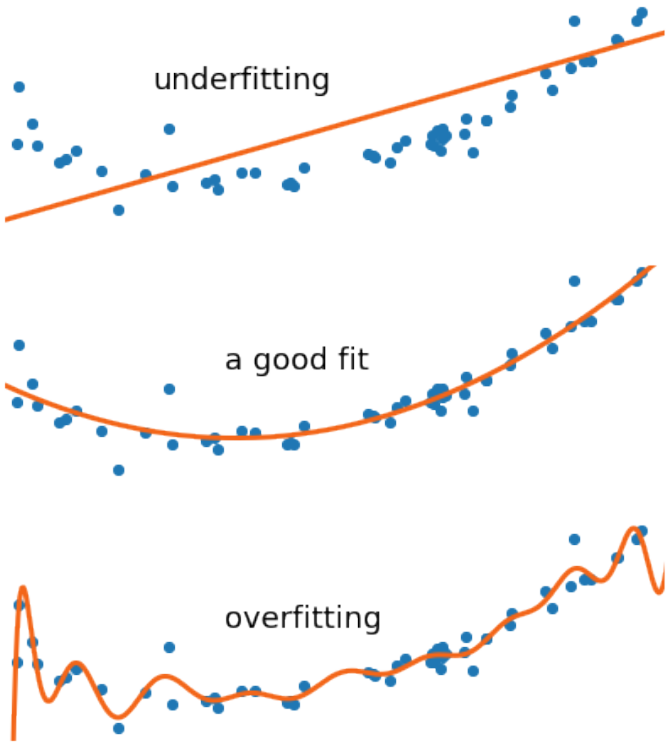


Figure 6.25: Overfitting vs. good fit vs. overfitting.

6.11.1 Parametric models: regularization

How do we reduce overfitting in parametric models? *shrinkage* via regularization. We have seen that linear regression tends to under-fit a single feature, but if we have a great many features we can actually overfit the dataset. To remove features that are of little importance (basically noise) we can use regularization. (Remember, if we have $\beta = 0$ this feature forms no part of the model).

Equation: Regularization = cost function (J) + penalty term

The $L1$ penalty (also known as LASSO) is $|\beta|$

$$J \leftarrow J + \lambda \sum |\beta| \tag{6.21}$$

which can be written as $\|\beta\|_1$, known as the Manhattan norm.
The $L2$ penalty (Ridge) is β^2

$$J \leftarrow J + \lambda \sum \beta^2 \tag{6.22}$$

which can be written as $\|\beta\|_2^2$, the Euclidean norm.

λ is the regularization hyper-parameter; the larger the value of λ the stronger the penalty.

In scikit-learn these penalties available via the following estimators:

- LASSO regression – Linear least squares with $L1$ regularization
- Ridge regression – Linear least squares with $L2$ regularization
- ElasticNet regression – Linear least squares with both $L1$ and $L2$ regularization

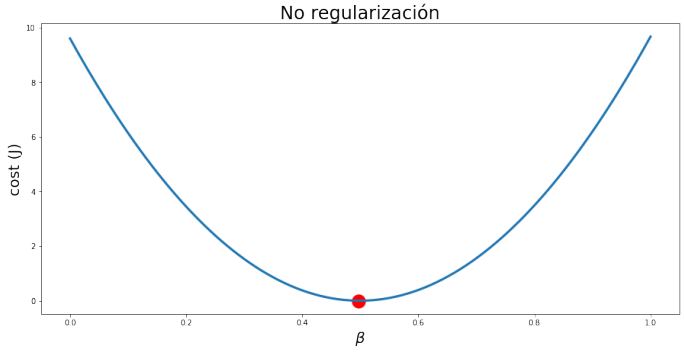


Figure 6.26: No regularization

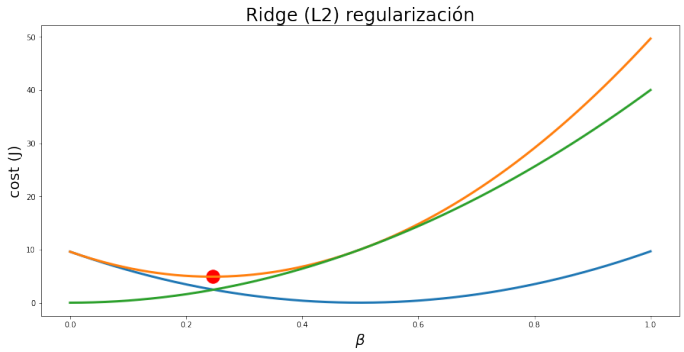


Figure 6.27: The effect of L2 (ridge) regularization

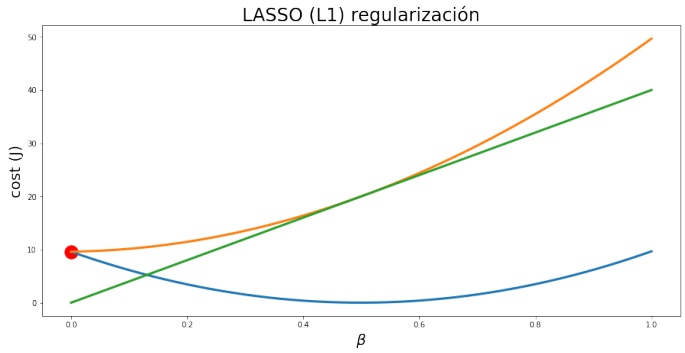


Figure 6.28: The effect of L1 (LASSO) regularization

In Figure 6.26 we have no regularization, corresponding to standard linear regression. The optimal value of β is found at the minima of the cost function (in blue), and has been marked with a red dot. In Figure 6.27 we apply the $L2$ penalty (in green), which is quadratic. When we add the $L2$ penalty to the original cost function (blue) we find that the minima of our new cost function (orange) has been shifted to a lower value. In Figure 6.28 this time we apply the $L1$ penalty (in green) which is linear. We can now see that it is possible to shrink the value of β all the way down to zero. Remember that when a feature has a β of zero, this feature no longer forms part of the model.

6.11.2 Tree models: `min_samples_leaf`

How do we reduce overfitting in decision tree models? by gardening! In Figure 6.23 we saw a tree that was over-fitted so as to have one single datapoint in each leaf. This is where the `min_samples_leaf` hyperparameter comes into play. We stop the splitting when reaching the specified number of samples per leaf. We can also make shallower trees by limiting the `max_depth` hyperparameter.

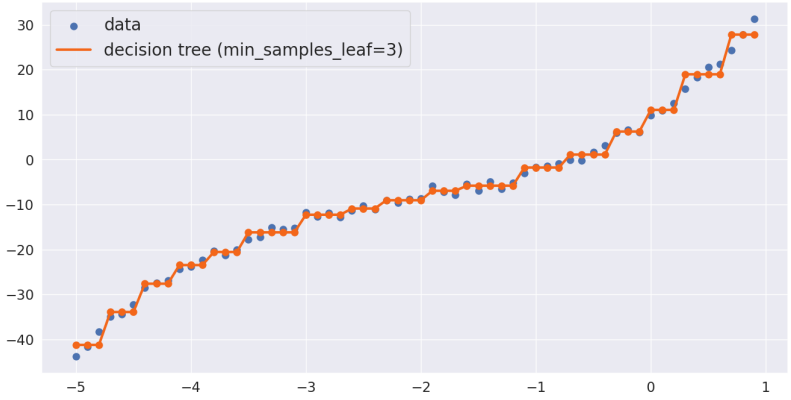


Figure 6.29: A decision tree with `min_samples_leaf = 3`.

(We can also perform 'pruning' using the `ccp_alpha` hyperparameter).

Remark: **Warning**: The default scikit-learn `DecisionTreeRegressor()` has no `max_depth` and `min_samples_leaf` is 1 so it creates "fully grown unpruned trees", which is basically an elegant way of saying it is **guaranteed to overfit**.

6.12 Quantile regression

Thus far we have produced what are known as point predictions, *i.e.* for a new feature vector we use our model to tell us $\hat{y}$. However, it is arguably much more useful, especially when it comes to decision making, to calculate prediction intervals. For example, say our model for house prices gave us a point prediction of 121,516€ for a given property. If we saw a house with the very same characteristics on sale for 120,000€ ¿Is that a bargain? If we put an asking price of 125,000€ on the house ¿Will it sell, or is it overpriced? This is where prediction intervals can really help us to make a decision. A good technique for producing prediction intervals is quantile regression[5,6]. We can specify a coverage (say 80%) and then produce a model. That model could return a lower price of say 110,000€ and an upper price of 130,000€ for said house. With this data we are now confident that if we see a house with the same characteristics on sale for 100,000€ then this property is below market value, and could be a very good investment opportunity. Or on the other hand we could confidently try to sell the property at 130,000€ without leaving money on the table.

6.12.1 Pinball loss function

In ordinary least squares (OLS) one uses the $L2$ loss: $(y_i - \hat{y}_i)^2$. For quantile regression one uses the pinball loss, which is given by

$$\mathscr{L}_p = \begin{cases} \tau\,(y_i - \hat{y}_i)\,, & \text{if } \hat{y}_i \leq y_i \\ (\tau - 1)\,(y_i - \hat{y}_i)\,, & \text{if } \hat{y}_i > y_i \end{cases} \tag{6.23}$$

where τ is the quantile, $\tau \in (0,1)$.

Remark: If $\tau = 0.5$ the pinball loss reduces to the half the $L1$ loss and we perform median regression.

Some estimators that can be used to perform quantile regression in python:

- `sklearn.linear_model.QuantileRegressor`
- `sklearn.ensemble.GradientBoostingRegressor`
- `sklearn.ensemble.HistGradientBoostingRegressor`
- `XGBoost` / `LightGBM` / `CatBoost`

There is also the `quantile-forest` package that implements quantile regression forests. What does quantile regression look like? For a linear model:

[5] Bassett, Koenker "*Regression Quantiles*", Econometrica **vol 46** pp. 33-50 (1978)
[6] Koenker, Hallock "*Quantile Regression*", Journal of Economic Perspectives **vol 15** pp. 143-156 (2001)

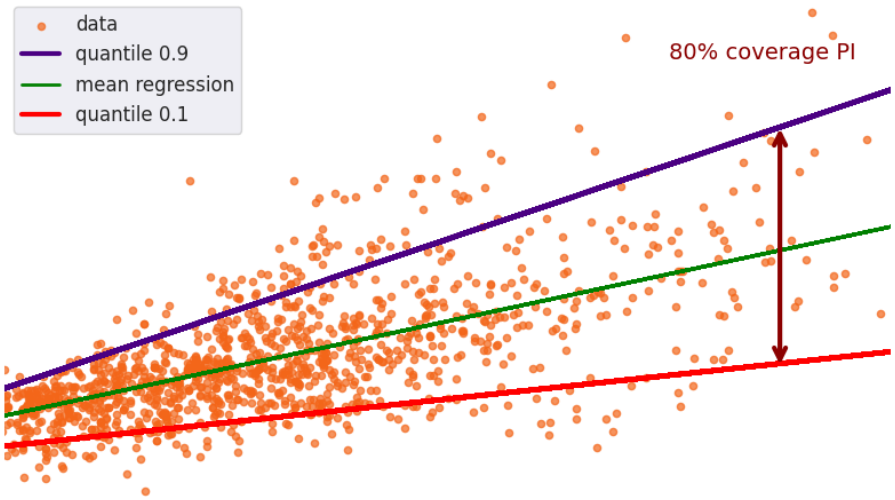


Figure 6.30: 80% prediction interval from a linear model

and for a non-parametric tree model

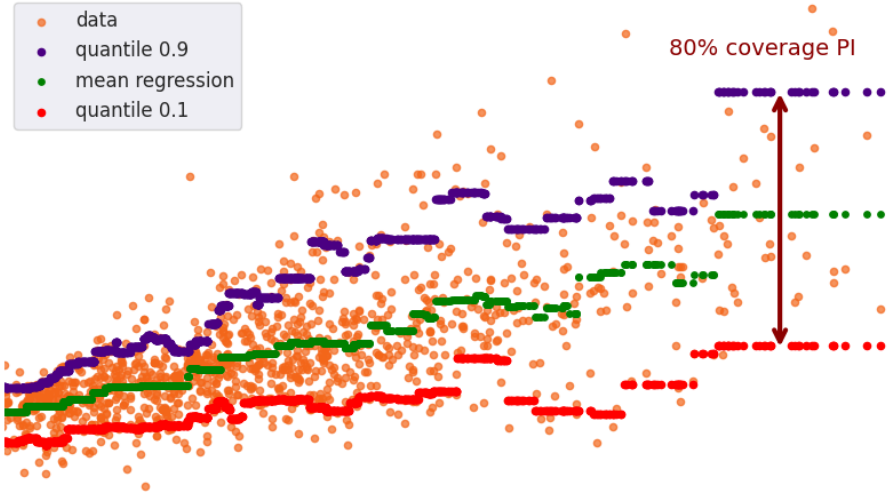


Figure 6.31: 80% prediction interval from a tree model

N **Jupyter Notebook:** PI competition I: Quantile regression

6.13 Conformal prediction intervals

Conformal prediction is a framework designed to provide valid uncertainty quantification and for an excellent introduction to this burgeoning technique see the book "Practical Guide to Applied Conformal Prediction in Python" by Valeriy Manokhin.

Remark:

- prediction intervals $\not\equiv$ error bars

- prediction intervals $\not\equiv$ confidence intervals

What is the problem? The intervals from our base quantile regression estimator may be wrong. For example, we may ask for 80% coverage, but we may not get 80% coverage. However, conformal prediction will guarantee our specified coverage. There are two types of coverage: *marginal* and *conditional*. For, say 80% coverage:

- **marginal**: 80% of the intervals will contain `y_true`
- **conditional**: a given interval has an 80% chance of containing `y_true`

It is not possible to guarantee distribution-free conditional coverage[7]. However, conformal prediction does guarantee marginal coverage - this is what is meant by validity. We want our prediction intervals to have three things:

- **validity**: guaranteed marginal coverage
- **efficiency**: intervals to be as narrow as possible
- **adaptability**: in cases of heteroscedasticity have variable lengths

conformal prediction provides the validity, the base estimator provides the efficiency (validity can also help the efficiency; overcoverage is inefficient), and the quantile regression provides the adaptability[8]. Conformal prediction comes in two 'flavours':

- **transductive** (*full* or *online*): no calibration set needed, but computationally expensive: multiple fits
- **inductive** (*split* or `prefit`): needs hold-out data for a calibration set, but is computationally efficient: just one fit

Requirements: for machine learning to work correctly our splits should be i.i.d. samples. For conformal prediction to work we only need *exchangeability*[9] ¿What is exchangeability? It is when one can `shuffle` or `sort` the rows of a dataset and it will make no difference to the predictions. Exchangeability is a weaker condition than i.i.d., so if we are i.i.d. we are good to go.

Remark: Exchangeability becomes problematic for time series data, where the order of the rows is indeed important.

6.13.1 Conformalized quantile regression (CQR)

We shall describe the conformalized quantile regression (CQR) technique by Emmanuel Candès and co-workers[10,11] which adds conformal prediction to standard quantile regression. Conformalized quantile regression is essentially a six step process:
1) train your two quantile regressors, Q_{lower} and Q_{upper}, on `X_train`, `y_train`
2) predict the two quantiles for the `X_calibrate` data

```
# calculate lower and upper quantile values of the calibration set
y_pred_lower_upper = qrf.predict(X_calib,
                                 quantiles=[alpha/2, 1-alpha/2])
```

[7] Barber, Candès, Ramdas, Tibshirani *"The limits of distribution-free conditional predictive inference"*, arXiv:1903.04684 (2020)
[8] Zecchin, Park, Simeone, Hellström *"Generalization and Informativeness of Conformal Prediction"*, arXiv: 2401.11810 (2024)
[9] Kuchibhotla *"Exchangeability, Conformal Prediction, and Rank Tests"*, arXiv:2005.06095 (2021)
[10] Romano, Patterson, Candès *"Conformalized Quantile Regression"*, NeurIPS **vol 32** (2019)
[11] Sesia, Candès *"A comparison of some conformal quantile regression methods"*, Stat **vol 9** e261 (2020)

3) for each `y_true` in `y_calibrate` calculate a conformity score E_i

$$E_i := \max(Q_{lower} - \texttt{y\_true},\ \texttt{y\_true} - Q_{upper})$$

```
calibration_df = pd.DataFrame(y_calib.values, columns = ['y_true'])
calibration_df["lower"] = y_pred_lower_upper[:,0]
calibration_df["upper"] = y_pred_lower_upper[:,1]
a  = (calibration_df["lower"]-calibration_df["y_true"]).values
b  = (calibration_df["y_true"]-calibration_df["upper"]).values
calibration_df["Ei"] = (np.vstack((a, b)).T).max(axis=1)
```

4) calculate `s` being the value of the $(1-\alpha)(1+1/n_{calib})$ quantile of this set of conformity scores $\{E\}$ where $\alpha :=$ (100-coverage%)/100, and n_{calib} is the number of calibration rows

```
s = calibration_df["Ei"].quantile((1-alpha)*(1+(1/(len(calibration_df)))))
```

5) predict the two quantiles for `X_test`
6) obtain validity by subtracting `s` from Q_{lower} and adding `s` to Q_{upper}

```
predictions["conformal_lower"] = predictions["lower"] - s
predictions["conformal_upper"] = predictions["upper"] + s
```

Remark: For the CQR calibration dataset (`X_calibrate`, `y_calibrate`) I would suggest using between 1000 and 2000 rows of data.

Jupyter Notebook: For a worked example see: Prediction intervals: Quantile Regression Forests

6.13.2 Locally-weighted conformal regression

In locally-weighted conformal regression[12] we use two estimators, one fitted to the training data ($\hat{\mu}$), and the other estimator ($\hat{\rho}$) is fitted to the errors, *i.e.* the difference between the predictions output by the model fitted to the training data (`y_pred_train`), and the `y_true` (Y) values. We denote our estimator as $\hat{\mu}$ as generally most machine learning regressors return a mean expectation value since the default loss function is usually $L2$, *i.e.* the squared error.

$$\mu(x) = \mathbb{E}(Y|X = x) \tag{6.24}$$

The locally-weighted residuals R_i are given by

$$R_i = \frac{|Y_i - \hat{\mu}(X_i)|}{\hat{\rho}(X_i)} \tag{6.25}$$

[12] Jing Lei, Max G'Sell, Alessandro Rinaldo, Ryan J. Tibshirani, and Larry Wasserman *"Distribution-Free Predictive Inference for Regression"*, Journal of the American Statistical Association, **113** pp. 1094-1111 (2018)

```
model_means  = regressor_mean.fit(X_train, y_train)
y_pred_train = model_means.predict(X_train)

# calculate the absolute deviations
AD = np.abs(y_pred_train - y_train)
# fit a model to these deviations
model_residuals = regressor_residual_mean.fit(X_train, AD)
```

now we apply these two models to the calibration data and calculate the conformity of each row (i)

```
y_pred_calib = model_means.predict(X_calib)
MAD_calib    = model_residuals.predict(X_calib)

R_i = np.abs(y_calib - y_pred_calib) / MAD_calib
```

and calculate the final conformity score (d), here for 90% coverage (*i.e.* $\alpha = 0.1$) (note the small finite sample correction)

$$d = \text{quantile}((1-\alpha)(1+1/n_{calib})) \tag{6.26}$$

where n_{calib} is the number of rows in the calibration set.

```
alpha = 0.1 # i.e. 90% coverage
n = len(X_calib)
d = R_i.quantile( (1-alpha)*(1+(1/n)) )
```

now for the test data

```
y_pred   = model_means.predict(X_test)
MAD_pred = model_residuals.predict(X_test)
```

our lower and upper conformal prediction intervals are calculated as

$$[\hat{\mu}(X_i) - \hat{\rho}(X_i)d,\ \hat{\mu}(X_i) + \hat{\rho}(X_i)d] \tag{6.27}$$

```
predictions = y_test.to_frame()
predictions.columns = ['y_true']
predictions["point prediction"] = y_pred

# create our conformal predictions
predictions["lower"] = y_pred - (MAD_pred * d)
predictions["upper"] = y_pred + (MAD_pred * d)
```

N **Jupyter Notebook:** Example notebook of performing locally-weighted conformal regression.

6.13.3 Prediction interval metric: Winkler interval score

We can evaluate the performance of our prediction interval estimates using the Winkler interval score[13, 14]:

Equation: Prediction interval performance metric: the mean Winkler interval score

$$W_\alpha = \begin{cases} (u-l) + \frac{2}{\alpha}(l-y), & \text{if } y < l \\ (u-l), & \text{if } l \leq y \leq u \\ (u-l) + \frac{2}{\alpha}(y-u), & \text{if } y > u \end{cases} \tag{6.28}$$

where y is the true value, u is the upper prediction, l is the lower prediction, and α is (100-coverage%)/100.

6.14 Summary

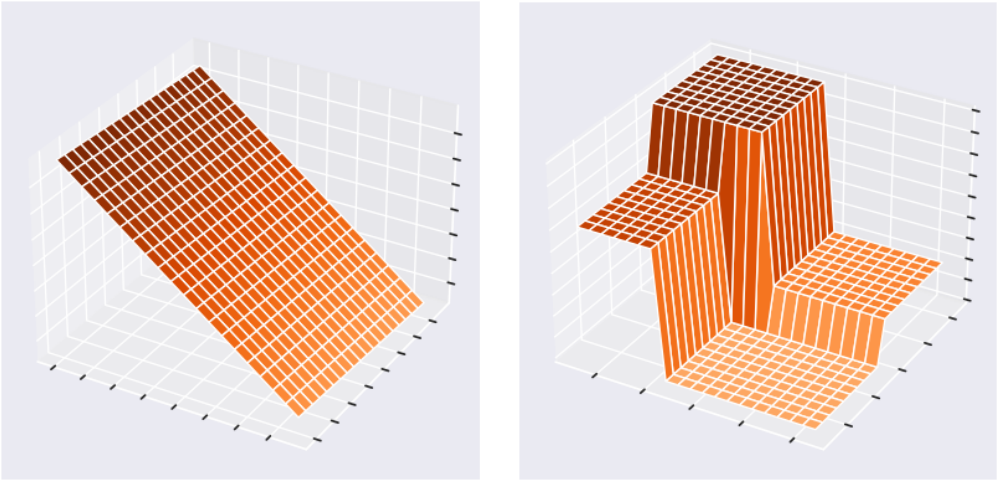


Figure 6.32: With two features linear regression fits a plane to the data, and a decision tree creates plateaux.

In this chapter we have seen the parametric linear regression model, the non-parametric decision tree model, and have also looked at median regression with quantile regression. We have looked at prediction intervals as a more informative option to calculating expectation values *i.e.* point predictions.

[13] Winkler *"A Decision-Theoretic Approach to Interval Estimation"*, Journal of the American Statistical Association **vol 67**, pp. 187-191 (1972)

[14] Brehmer, Gneiting *"Scoring interval forecasts: Equal-tailed, shortest, and modal interval"*, Bernoulli **vol 27**, pp. 1993-2010 (2021)

Recommended reading

Papers

- Cawley, Talbot "*On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation*", The Journal of Machine Learning Research **11** pp. 2079-2107 (2010)
- Mikhail Belkin, Daniel Hsu, Siyuan Ma, Soumik Mandal *"Reconciling modern machine-learning practice and the classical bias–variance trade-off"*, PNAS **116** pp. 15849-15854 (2019)
- Sebastian Raschka "*Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*", arXiv:1811.12808 (2020)
- Angelopoulos, Bates "*A Gentle Introduction to Conformal Prediction and Distribution-Free Uncertainty Quantification*", arXiv:2107.07511 (2022) (§2.2)

Books

- Cosma Rohilla Shalizi "*The Truth about Linear Regression*", (2024)
- Wessel N. van Wieringen *"Lecture notes on ridge regression"*, arXiv:1509.09169 (2023)
- Valery Manokhin "*Practical Guide to Applied Conformal Prediction in Python*", Packt Publishing (2023) (Chapter 7)

Packages

- quantile-forest by Reid Johnson
- MAPIE - Model Agnostic Prediction Interval Estimator
- Puncc
- crepes by Henrik Boström

7. Classification

> The only useful function of a statistician is to make predictions, and thus to provide a basis for action.
>
> *W. Edwards Deming*

A classification problem in one in which the target values (often called *labels*) belong to a set of classes. From a machine learning standpoint our task is to produce the probability of new data belonging to each of these classes. The most frequent problem is known as binary classification, where the labels are either `0` or `1`, *i.e.* $y \in \{0, 1\}$

Remark: Class 1 is often called the *positive* class and correspondingly is not unusual in older texts to see the class 0 to logically be called the *negative* class and written as -1. I would suggest that a 0 is now used for the negative class as it is less expensive on a computer to store an unsigned 0 than it is a -1.

In regression our challenge was to find a line that was as close as possible to all of the datapoints at the same time

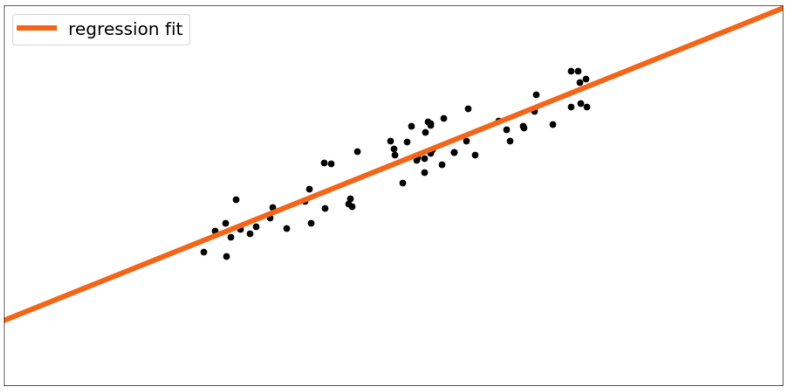


Figure 7.1: Linear regression

However, now in classification we would like to find the line that best divides the two classes, called the *decision boundary*.

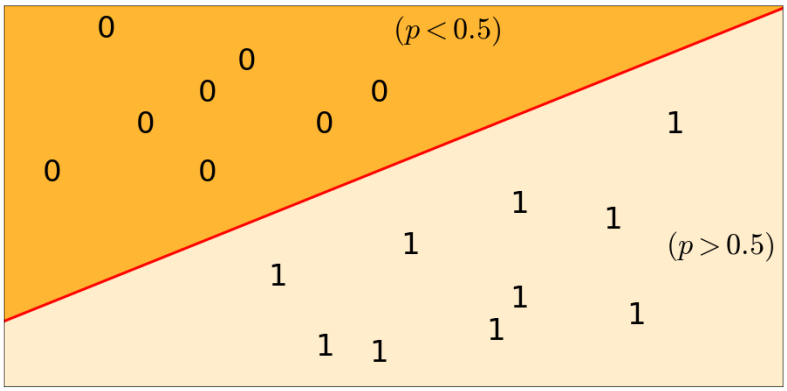


Figure 7.2: Classification with the decision boundary in red for the $p = 0.5$ decision threshold.

7.1 Logistic regression

Given our data-points in Figure 7.3 the first thing we could try doing is to fit a linear regression to the data:

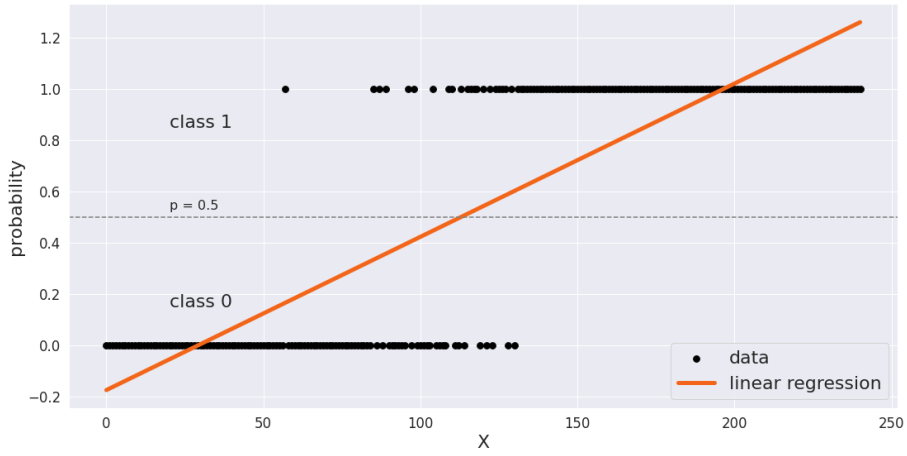


Figure 7.3: Linear regression fit to binary data.

However, there are some clear drawbacks to doing this: for example unless the straight line we have fitted is horizontal ($\beta_1 = 0$) which due to the nature of the data cannot be the case, then there is no upper nor lower bound to our fit. If our fit is to represent a probability, clearly probabilities greater than one, and less than zero are meaningless. We could rectify this by Winsorizing the predictions by clipping our data on $[0, 1]$ with something like `df["y_pred"] = df["y_pred"].clip(lower=0,upper=1)` but closer inspection reveals other problems. Here we are using our fit to represent the probability that for a given value of X our `y_pred` represents a probability of seeing the class 1. For example in Figure 7.3 at $X = 150$ our line indicated a probability of roughly 0.8 of seeing class 1. However we can see from the datapoints that there are no events. Our straight line fit is evidently overly pessimistic.

This linear fit to binary data can clearly be improved upon. A much better model was discovered long ago when fitting curves to data that saturates. A classic example can be found from chemistry in the form of an autocatalytic reaction in which the concentration of the product slows until all of the reactant is used up.

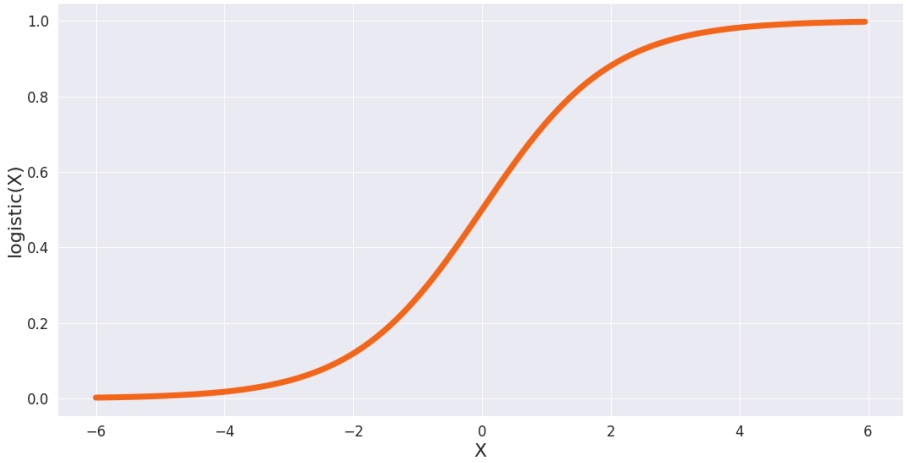


Figure 7.4: The logistic function

Equation: The logistic function, which can be used to map $\mathbb{R} \to (0,1)$

$$\text{logistic}(x) := \frac{1}{1+e^{-(\kappa_1 x+\kappa_0)}} \tag{7.1}$$

Observe that the term within the exp() function is the same as our linear regression model. Essentially we are simply squashing a linear regression to now fit between 0 and 1.

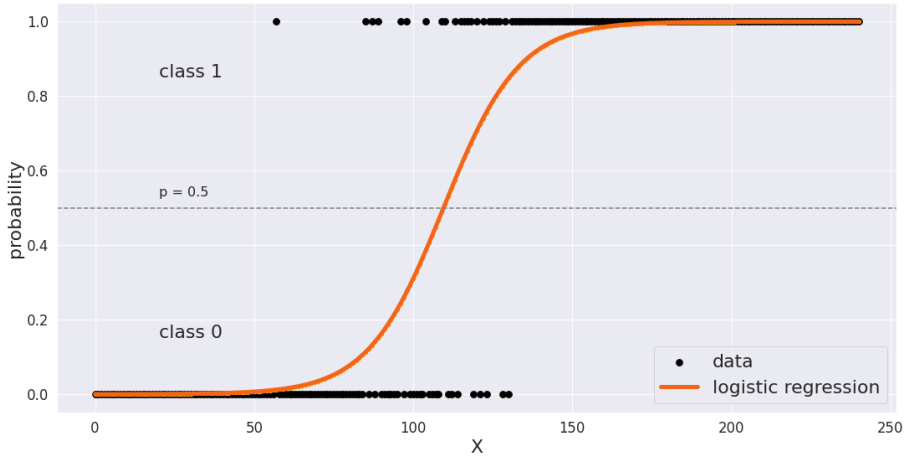


Figure 7.5: Fitting the data with a logistic function

Thus, in linear regression our task is to find the parameters β_1 and β_0 of the straight line that best fits our data.

$$\hat{y}(x) = \beta_1 x + \beta_0$$

In logistic regression similarly our task is to find the parameters κ_1 and κ_0 of the logistic function that best fits our data

$$\ln\left(\frac{\hat{y}}{1-\hat{y}}\right) = \kappa_1 x + \kappa_0 \tag{7.2}$$

Remark: The LHS of this equation is known as the *log-odds* or the logit link function (g). Beyond the scope of this book are generalized linear models (GLM).

Sample code

```
from sklearn.linear_model import LogisticRegression

classifier = LogisticRegression(penalty='none')
classifier.fit(X.reshape(-1, 1), y)
y_pred  = classifier.predict_proba(X.reshape(-1, 1))[:,1]

k1 = classifier.coef_[0]
k0 = classifier.intercept_
```

7.1.1 Explainability

Logistic regression, just like its homologue linear regression, has the virtue that its predictions, although not necessarily the best, are relatively easy to explain.

N **Jupyter Notebook:** Titanic explainability: Why me? asks Miss Doyle

7.2 Log-loss function

It is often computationally convenient to use log probability; it is worth reminding ourselves that the probability (of independent events) is multiplicative

$$p(A \cap B) = p(A) \cdot p(B) \tag{7.3}$$

which we can make additive by using logarithms since

$$\ln(p(A) \cdot p(B)) = \ln(p(A)) + \ln(p(B)) = \sum_i \ln(p_i) \tag{7.4}$$

Equation: The loss function in classification is the log loss or *cross-entropy* loss

$$\mathscr{L}_{log} = -y \ln(p(\hat{y})) - (1-y)\ln(1-p(\hat{y})) \tag{7.5}$$

and the cost function is the sum of the log losses

$$J_{log} = -\frac{1}{N}\sum (y \ln(p(\hat{y})) + (1-y)\ln(1-p(\hat{y}))) \tag{7.6}$$

where $p(\hat{y})$ is the probability of belonging to class 1, and thus $(1-p(\hat{y}))$ is the probability of belonging to class 0.

Remark: Just as in linear regression this cost function is convex, so we can always find the global minimum.

What does this loss function $\mathscr{L}$ look like? If the class of y is 0 then Equation 7.5 reduces to $\mathscr{L}_{log} = -\ln(1-p(\hat{y}))$

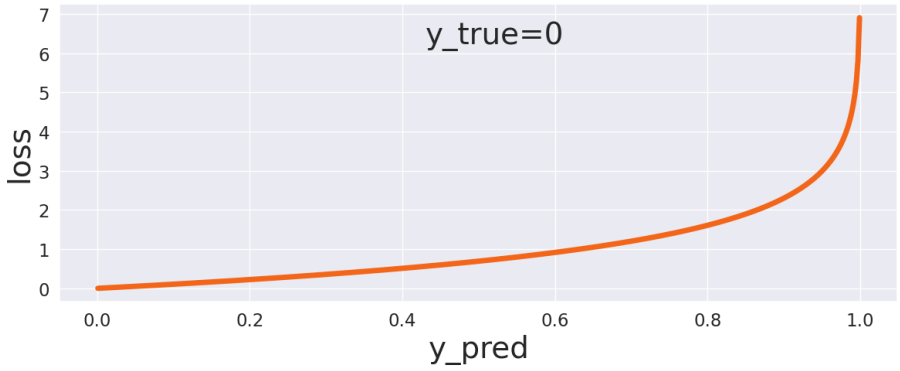


Figure 7.6: The loss for a given predicted probability when the true class is 0.

and if the class y is 1 then we have $\mathscr{L}_{log} = -\ln(p(\hat{y}))$

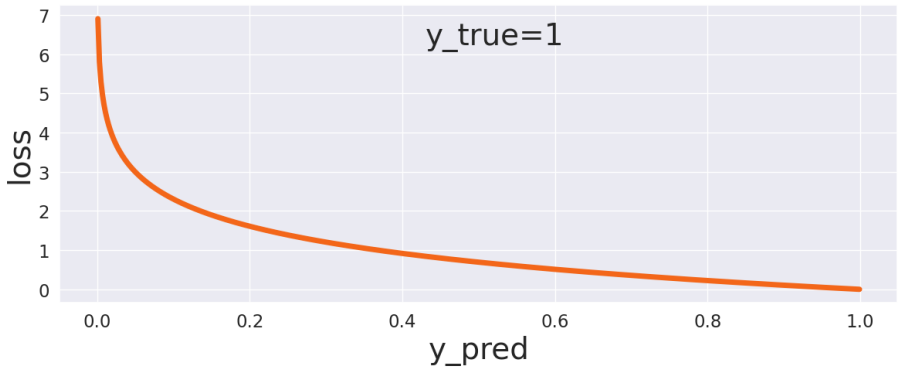


Figure 7.7: The loss for a given predicted probability when the true class is 1.

Remark: Note that with logistic regression, as the very name suggests, we are still performing regression, this time in probability space rather than in real space as per linear regression. This only becomes classification upon the application of a decision threshold to assign class labels to our predicted probabilities.

7.3 Decision tree classifier

Just as with regression, we can also perform classification using decision trees. Again we shall use a toy dataset

X	y_true
1	0
2	0
4	1
5	1

Table 7.1: Toy classification dataset.

and here is our decision tree fit to the data

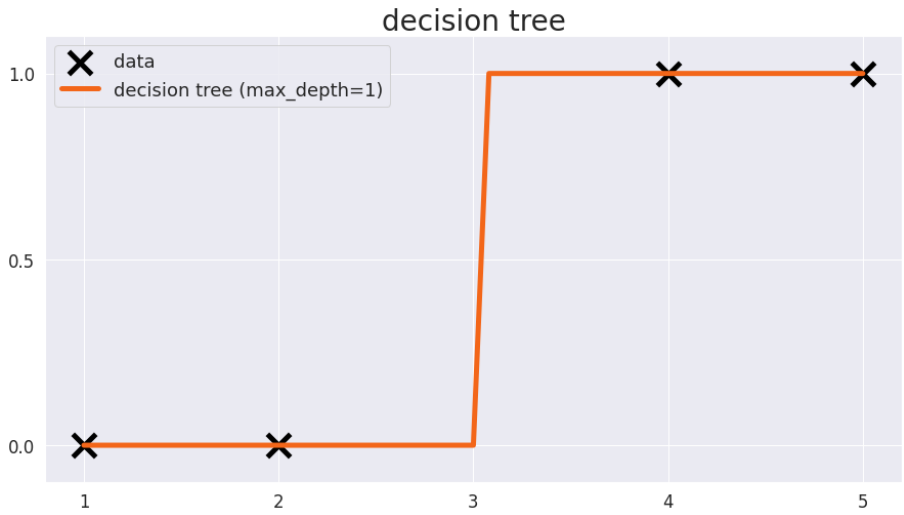


Figure 7.8: Decision tree classifier results for the toy dataset

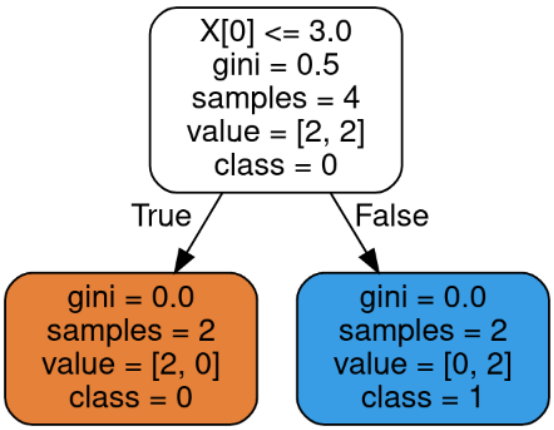


Figure 7.9: graphviz of our decision tree classifier (having `max_depth=1`)

Sample code

```
import graphviz
from sklearn.tree import export_graphviz

dot_data = export_graphviz(classifier,
```

```
                                filled=True,
                                rounded=True,
                                proportion=False,
                                class_names=['0', '1'])
graphviz.Source(dot_data)
```

Indeed we can even describe our model in two sentences:

- if x is less than 3, $\hat{y}$ is 0
- if x is greater than 3, $\hat{y}$ is 1

This time instead of the squared error loss function used in regression we now use the *Gini impurity* (I_G) as the cost function for splitting.

Equation: The Gini impurity (I_G), named after Corrado Gini, is given by

$$I_G(p) = 1 - \sum_{i=1}^{J} p_i^2 \tag{7.7}$$

where p_i is the fraction of samples in class i. For binary classification this becomes

$$I_G = 1 - p(\text{zero})^2 - p(\text{one})^2 \tag{7.8}$$

The more pure the leaf node, the smaller the Gini score (range $[0, 0.5]$)

mixture	**Gini score**
0 0 0 1 1 1	$I_G = 1 - \left(\frac{3}{6}\right)^2 - \left(\frac{3}{6}\right)^2 = 0.5$
0 0 0 1 1	$I_G = 1 - \left(\frac{3}{5}\right)^2 - \left(\frac{2}{5}\right)^2 = 0.48$
0 0 0 1	$I_G = 1 - \left(\frac{3}{4}\right)^2 - \left(\frac{1}{4}\right)^2 = 0.375$
0 0 0	$I_G = 1 - \left(\frac{3}{3}\right)^2 - \left(\frac{0}{3}\right)^2 = 0$
1 1 1	$I_G = 1 - \left(\frac{0}{3}\right)^2 - \left(\frac{3}{3}\right)^2 = 0$

Table 7.2: Example Gini calculations

An alternative impurity criteria is the Shannon information entropy (I_E),

Equation: Shannon information entropy (I_E)

$$I_E(p) = -\sum_{i=1}^{J} p_i \log_2(p_i) \tag{7.9}$$

In Figure 7.10 we plot the Gini and Shannon functions.

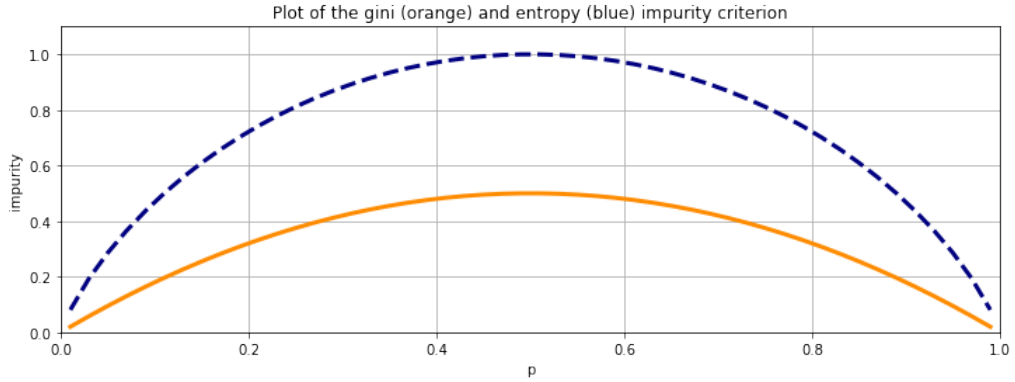


Figure 7.10: Plot of the Gini (orange) and Shannon (blue) functions.

However, it has been found that empirically there is little to choose between these two criteria[1].

N **Jupyter Notebook:** Titanic: some sex, a bit of class, and a tree...

7.4 Classification baseline model

Equation: Classification baseline model

$$\hat{y} = 0 \ \ \forall \, \hat{y} \tag{7.10}$$

We always predict the majority class label (0 by convention). This is also known as the ZeroR model.

7.5 Classification metrics

Unlike the situation in regression a plethora of metrics have been invented for classification.

7.5.1 Strictly proper scoring rules

Just as in regression, the most reasonable metrics are those that are aligned with the loss function and this leads to a set of strictly proper scoring rules[2]. Of particular note are the logarithmic score and the Brier score. Classifier calibration will lead to an improvement in these metrics.

7.5.2 Accuracy score

The accuracy score is the most popular (and perhaps the least advisable) classification metric. It is simply the fraction of correctly classified examples. The baseline accuracy score for a binary classification problem corresponds to the fraction of class 0 in the dataset. We can see that the accuracy score is only a reasonable metric if one is working with a balanced dataset. For imbalanced data one encounters the *accuracy paradox* in which our baseline model performs increasingly well the more imbalanced the data. Be very wary of

[1] Laura Elena Raileanu, Kilian Stoffel *"Theoretical Comparison between the Gini Index and Information Gain Criteria"*, Ann. Math. Art. Int. **41** pp. 77-93 (2004)

[2] Gneiting, Raftery *"Strictly Proper Scoring Rules, Prediction, and Estimation"*, Journal of the American Statistical Association **102** pp. 359-378 (2007)

any study that reports the accuracy score and check the imbalance of the dataset to get an idea of the baseline accuracy. If one really does have to use the accuracy score, then best use the balanced variant: `from sklearn.metrics import balanced_accuracy_score`

7.5.3 Confusion matrix

For binary labels there are four possible classifications

false positive (FP)	$y = 0$ and $\hat{y} = 1$
false negative (FN)	$y = 1$ and $\hat{y} = 0$
true positive (TP)	$y = 1$ and $\hat{y} = 1$
true negative (TN)	$y = 0$ and $\hat{y} = 0$

Table 7.3: The four possible binary classification assignments.

which are often presented in what is known as a *confusion matrix* or *contingency table*:

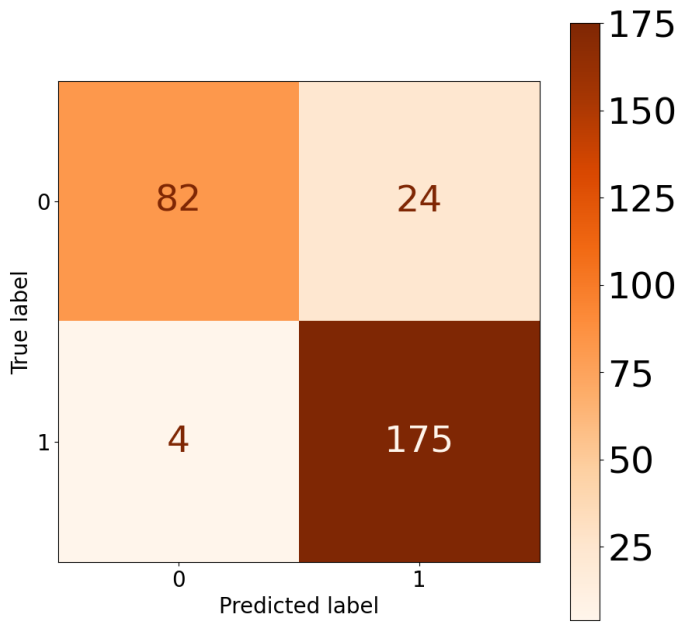


Figure 7.11: A example of a confusion matrix.

In Figure 7.11 we have an example of a confusion matrix. The true values are along the main diagonal, here TN=82, TP=175, and the off-diagonal has the false values; FN=4 and FP=24. If the classification were perfect the off-diagonal elements would all be zeros.

N **Jupyter Notebook:** "Titanic: In all the confusion..."

Sample code

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_true, y_pred, labels=classifier.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
```

```
                                    display_labels=classifier.classes_)
disp.plot();
```

7.5.4 Precision and recall

The precision metric favours few false positives. A use case for precision could be in classifying whether a student text is a case of plagiarism or not; we would rather not falsely accuse someone, so we would like as few false positives as possible.

$$\text{precision} = \frac{TP}{TP+FP} \tag{7.11}$$

The recall metric favours few false negatives

$$\text{recall} = \frac{TP}{TP+FN} \tag{7.12}$$

A use case of recall could be in initial cancer screening tests; we would rather err on the side of caution and try to catch all of the subjects that may have cancer, thus we would rather have false positives than let one false negative slip through.

However, both of these metrics can be trivially gamed to achieve perfect scores; the precision by predicting `0` for everything and a `1` for the single most probable example, and the recall simply by always predicting `1`, thus these metrics are of little use when reported alone. A combination of the two can be found in the F_1 score, which assigns equal importance to precision and recall by taking the harmonic mean:

$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{7.13}$$

This is available in scikit-learn via `from sklearn.metrics import f1_score` or as part of a `classification_report`. (Note: The F_1 score is the F_β score where $\beta = 1$).

7.5.5 Decision threshold

The accuracy score, precision, recall and F_1 score all work with class labels rather than the underlying probabilities (see the wonderful article *"Classification vs. Prediction"* by Frank E. Harrell Jr.). By default the decision threshold for converting probabilities into class labels is set at 0.5. However, optimizing the location of this threshold can lead to improvements in the aforementioned metrics. For the F_1 score it has been suggested that *"...if the classifier outputs are well-calibrated conditional probabilities, then the optimal threshold is half the optimal F_1 score."*[3].

Remark: As of version 1.5 Scikit-learn now includes the `TunedThresholdClassifierCV` which post-tunes the decision threshold using cross-validation.

Jupyter Notebook: False positives, false negatives and the discrimination threshold.

[3] Chase Lipton, Elkan, Narayanaswamy *"Thresholding Classifiers to Maximize F1 Score"*, arXiv:1402.1892 (2014)

7.5.6 AUC ROC

The AUC ROC is shorthand for the area under the receiver operating characteristic curve. and has it origins in assessing radar operators ability to discriminate between noise signals (say a flock of seagulls) and aircraft signals. In Figure 7.12 we schematically envisage a histogram of our probabilistic predictions (for example from the scikit `predict_proba` method) along with a vertical dashed line representing the *decision threshold*. Predictions below the decision threshold are classed as `0` and predictions above the decision threshold are classed as `1`. We can see that negative class examples that lie above the decision threshold become false positives, and positive class examples that lie below the decision threshold become false negatives.

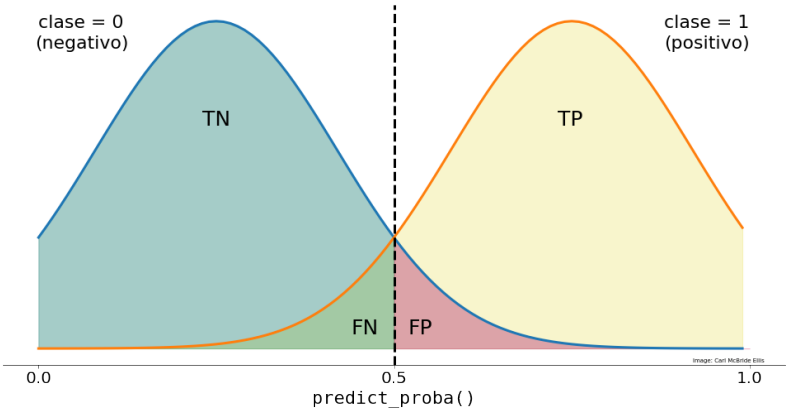


Figure 7.12: The classes we wish to distinguish between.

The better the classifier the more it is able to discriminate the two classes; the distance between the means of the distributions in units of σ is called the discriminability (or sensitivity) index (d') and a distance of 0 indicates that the classifier cannot distinguish between the two classes, whereas larger discriminability index values indicates better separation. This is shown on the LHS of Figure 7.13. On the RHS is the corresponding plot of the true positive rate against the false positive rate and it is the area under this curve that becomes the AUC ROC metric[4].

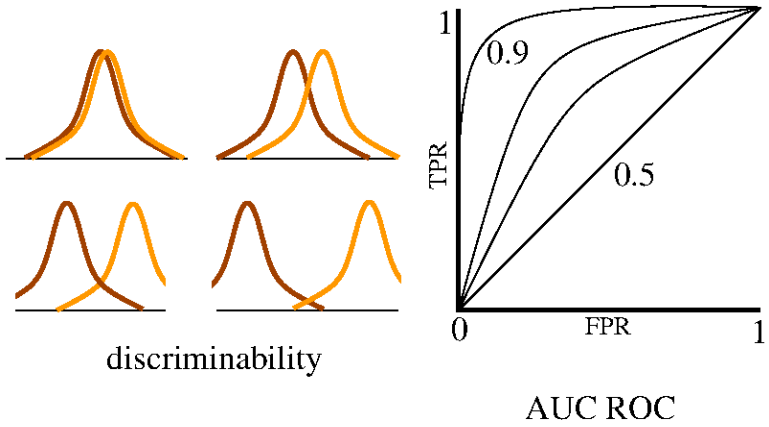


Figure 7.13: Discriminability index and the ROC curve.

[4] Tom Fawcett *"An introduction to ROC analysis"*, Pattern Recognition Letters **27** pp. 861-874 (2006)

An AUC ROC score of 0.5 is no better than random guessing, whilst a value of 1 represents perfect classification (`from sklearn.metrics import roc_auc_score`).

Equation:

$$\text{gini} = 2 \times \text{AUC} - 1 \tag{7.14}$$

7.6 Imbalanced classification

Imbalanced data is, in the binary case, when the number of examples of the majority class far outweighs the minority class, as represented in Figure 7.14.

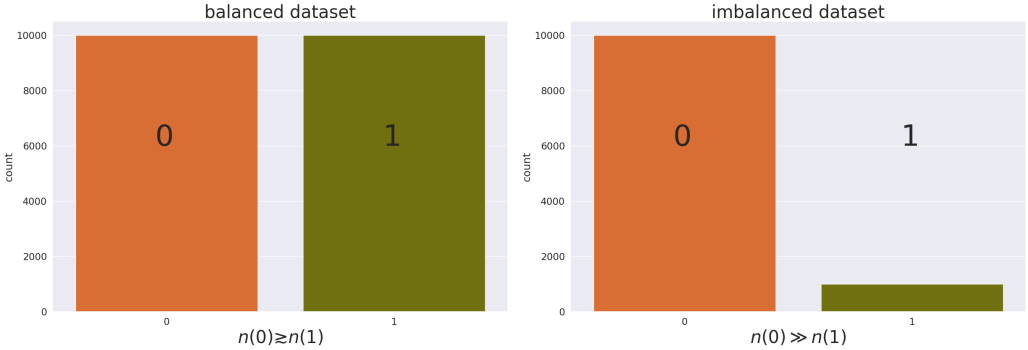


Figure 7.14: Countplots for a balanced and an imbalanced dataset

There have been rivers of ink dedicated to the so-called problem of imbalanced classification, and I would suggest this is due to the poor behavior of many classification metrics once the imbalance is greater than ≈ 0.9 as seen in Figure 7.15. I say so-called because quite frankly modern estimators do not have any particular problem with performing imbalanced classification at all.

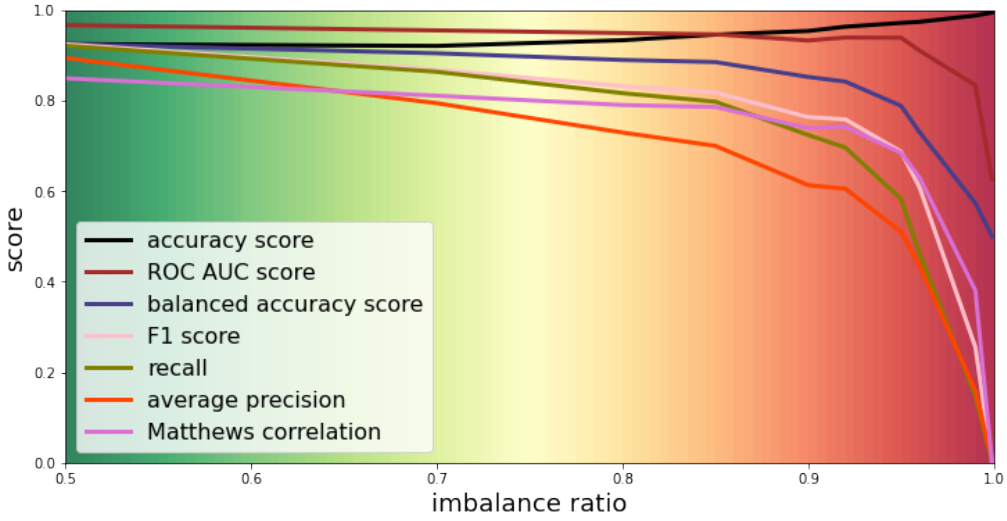


Figure 7.15: Metrics applied to imbalanced data

Jupyter Notebook: Classification: How imbalanced is "imbalanced"?

7.6.1 What to do about imbalanced data?

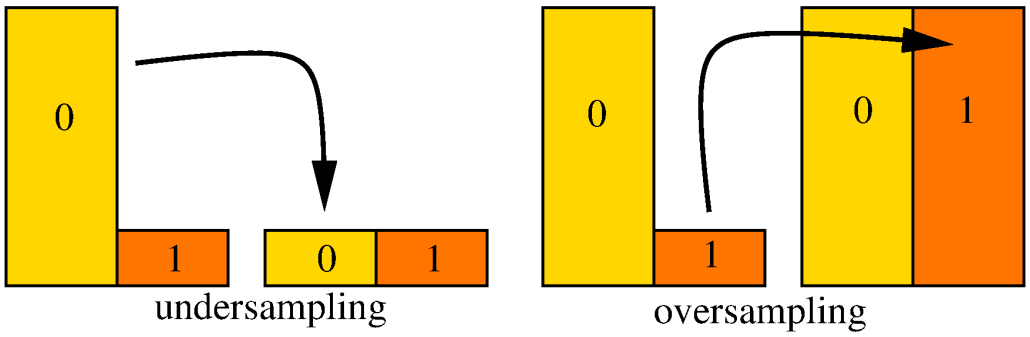


Figure 7.16: Undersampling and oversampling to rebalance data.

When it comes the imbalanced data one often encounters several potential approaches:

- Re-balance (50%:50%) by:
  - undersampling: delete a load of good data from the majority class (0)
  - oversampling: create a load of new synthetic data for the minority class (1)
- weight the loss function (*i.e.* `class_weight="balanced"`)
- do absolutely nothing

Regarding oversampling: the popular method of bulking up the minority class with synthetic data has a fundamental flaw; if one can readily identify the minority class examples that one wishes to synthesize more of, then there is actually no need to create new data in the first place. And if one cannot readily identify the minority class, then one simply produces yet more noisy points. Either way it is a pointless exercise. Indeed studies have found that rebalancing data has no advantages whatsoever[5].

Remark: Whatever you do, do not use SMOTE or variants thereof to create synthetic data (see the excellent paper: "To SMOTE, or not to SMOTE?").

7.7 Overfitting

Visual examples of under and overfitting for a parametric classifier

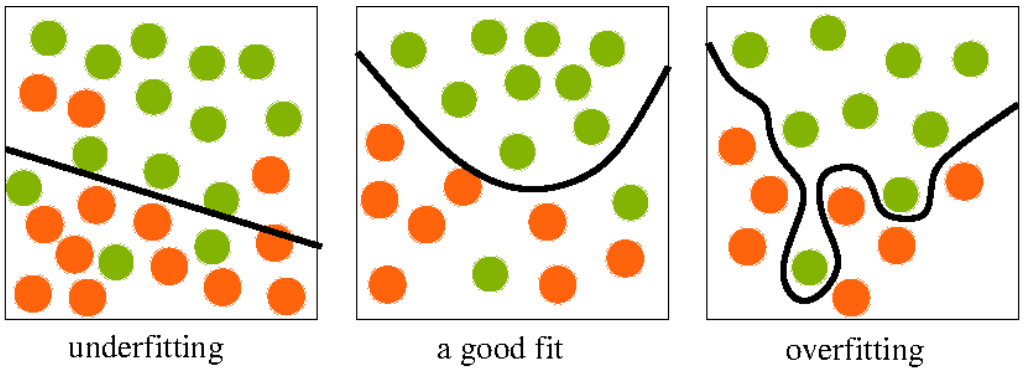


Figure 7.17: Underfitting and overfitting for a parametric classifier.

[5] van den Goorbergh, van Smeden, Timmerman, Van Calster *"The harm of class imbalance corrections for risk prediction models: illustration and simulation using logistic regression"*, Journal of the American Medical Informatics Association **29** pp. 1525-1534 (2022)

and for a non-parametric classifier

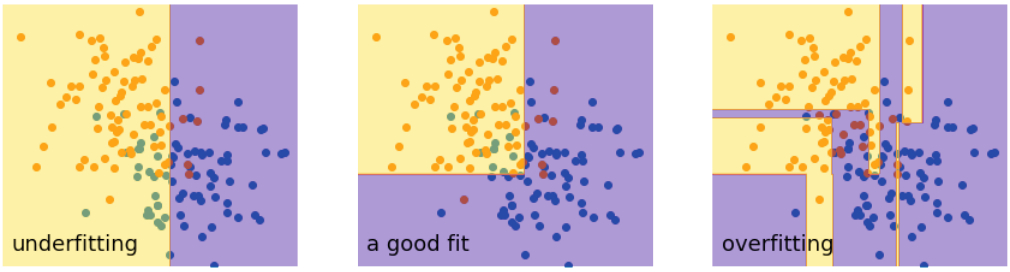


Figure 7.18: Underfitting and overfitting for a Decision Tree classifier

In classification, just as in regression, one can reduce overfitting in parametric models (such as logistic regression) by using regularization ($L1$, $L2$, $L1+L2$), and in tree based estimators by increasing the `min_samples_leaf` hyperparameter.

Remark: Note that for some reason by default the $L2$ term is non-zero ($C = 1$) for the `sklearn LogisticRegression` estimator.

7.8 No free lunch theorem

Which is the best estimator? Well that depends on the dataset: *a priori* one does not know which estimator is the best for your dataset. For some datasets maybe a parametric linear model will work best, and for others maybe a Decision Tree will work best. Suffice to say that a model trained on a specific dataset is not transferable to a different dataset. This goes by the rather fanciful name of the '*No free lunch theorem*'[6].

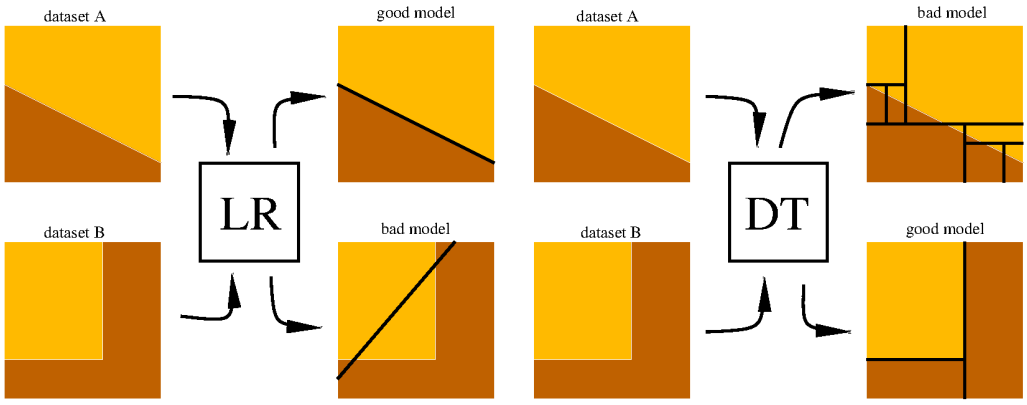


Figure 7.19: Different estimators for different datasets; there is no one best model.

7.9 Classifier calibration

For many classifiers `predict_proba` returns a relative score[7] For cost-sensitive decision making it is imperative to work with good probabilities. Motivation: ¿Which fraud case is

[6] David Wolpert "*The Lack of A Priori Distinctions Between Learning Algorithms*", Neural Computation **8** pp. 1341-1390. (1996)

[7] Sweidan, Johansson *"Probabilistic Prediction in scikit-learn"*, diva2:1603345 (2021)

most worth investigating first?
uncalibrated:

- 500K€ fraud with $p = 0.61 \rightarrow$ expected cost = 305K
- 1M€ fraud with $p = 0.27 \rightarrow$ expected cost = 270K

calibrated:

- 500K€ fraud with $p = 0.59 \rightarrow$ expected cost = 295K
- 1M€ fraud with $p = 0.34 \rightarrow$ expected cost = 340K

We can see that incorrect probabilities will lead to incorrect decisions. Classifier probabilities are calibrated if they are unbiased conditional on their own predictions $\mathbb{E}[Y|p] = p$ Predicted probabilities can be calibrated via a correction function fitted to empirical data so as to reflect the true probabilities[8].

7.9.1 Reliability diagrams

Visually one can inspect calibration via a reliability diagram as shown in Figure 7.20

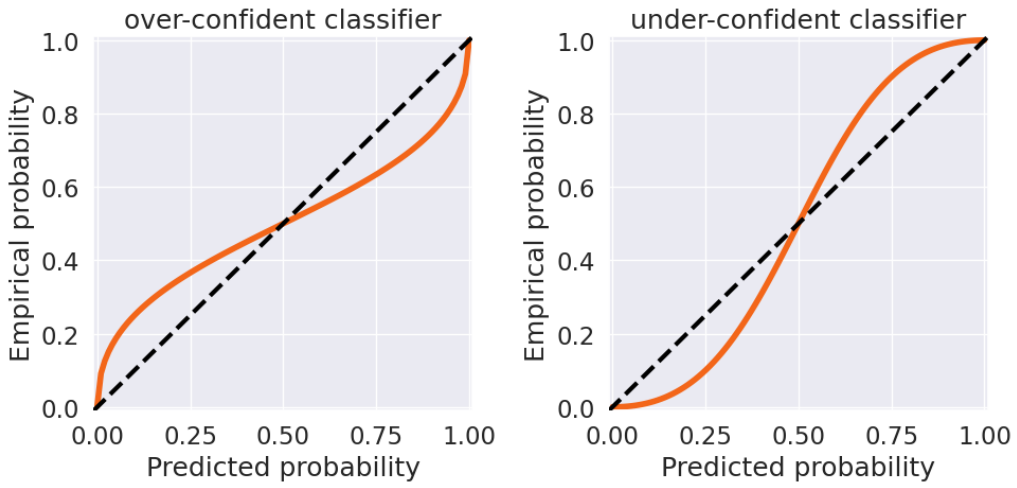


Figure 7.20: Miscalibration: over- and under-confident classifiers.

Some of the many probability calibration techniques are:

- **Platt scaling** - this is a two-parameter sigmoid correction, originally designed for the SVM classifier
- **Isotonic scaling** - non-parametric monotonic step-wise regression fitted to class 1 for $P(1)$ values
- **Venn-ABERS** method - double isotonic; fitted to both class 1 and class 0, then merge the two via

$$p = \frac{\text{fit}_{(1)}}{(1 - \text{fit}_{(0)}) + \text{fit}_{(1)}} \tag{7.15}$$

[8] Niculescu-Mizil, Caruana *"Predicting good probabilities with supervised learning"*, ICML '05, pp. 625-632 (2005)

7.9.2 Venn-ABERS calibration

For the Venn-ABERS method [9, 10, 11] what do the two fits $p1$ and $p0$ look like?

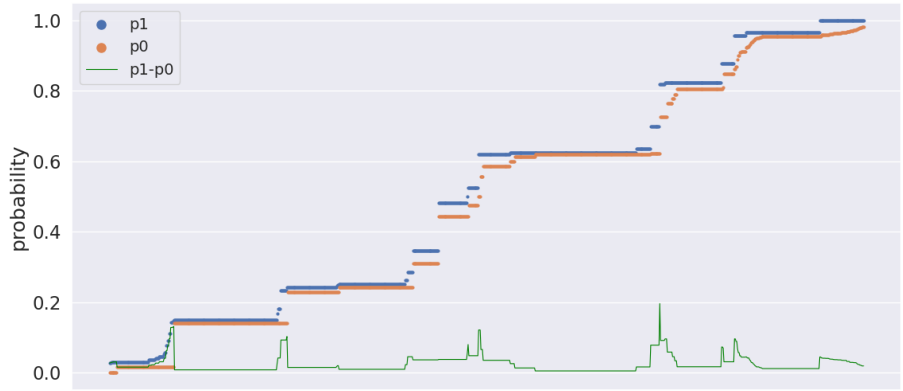


Figure 7.21: Venn-ABERS: plots for the $p0$ and $p1$ isotonic regressions.

There are two implementations of Venn-ABERS predictors

- **IVAP**: inductive (`"prefit"`) which requires a hold-out calibration set
- **CVAP**: cross-conformal: fit and calibrate k times via `StratifiedKFold`

First we install the `venn-abers` routine, written by Ivan Petej

```
!pip install -q venn-abers
from venn_abers import VennAbersCalibrator
```

IVAP example: split/`"prefit"` Venn-ABERS

```
p_cal  = classifier.predict_proba(X_calib)
p_test = classifier.predict_proba(X_test)

VAC = VennAbersCalibrator()
predictions = VAC.predict_proba(p_cal = p_cal,
                                y_cal = y_calib.values,
                                p_test = p_test)[:,1]
```

Remark: For the IVAP calibration dataset (`X_calib`, `y_calib`) I would suggest for good results setting aside between 1000 and 2000 rows of data from the training data data.

CVAP example: cross-conformal Venn-ABERS (no hold-out calibration set required)

```
VAC = VennAbersCalibrator(estimator=classifier,
                          inductive=False,
```

[9] Vovk, Shafer, Nouretdinov "*Self-calibrating Probability Forecasting*", NIPS'03 pp. 1133-1140 (2003)
[10] Vovk, Petej "*Venn-Abers Predictors*", arXiv:1211.0025 (2014)
[11] Vovk, Petej, Fedorova "*Large-scale probabilistic prediction with and without validity guarantees*", arXiv:1511.00213 (2015)

```
                              n_splits=5)

VAC.fit(X_train, y_train)
predictions = VAC.predict_proba(X_test)[:,1]
```

In Figure 7.22 we have the reliability diagram for an XGBoost calculation. We can see that it is clearly over-confident in its predictions (metric results: log loss 0.801, Brier score 0.188).

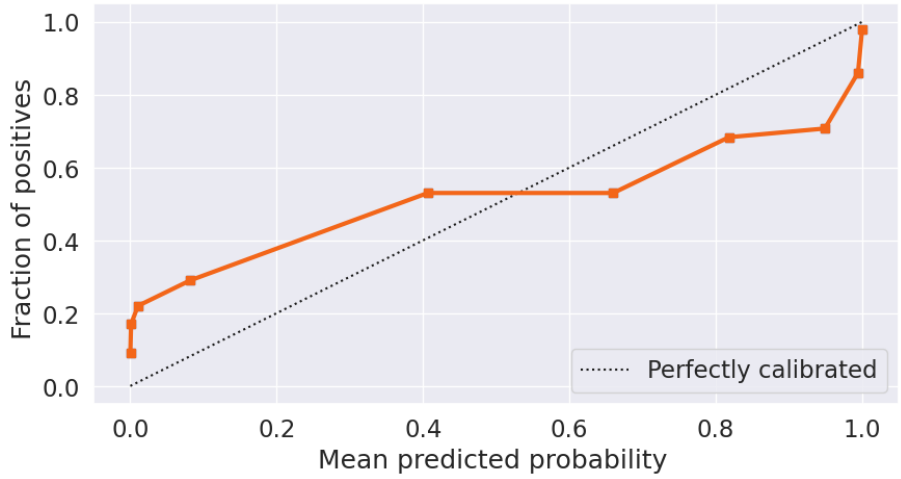


Figure 7.22: The reliability diagram for an XGBoost fit without calibration.

We now perform calibrated cross-conformal Venn-ABERS with the same estimator and the results show a marked improvement; log loss 0.497 and Brier score 0.164, as well as can be seen visually in Figure 7.23.

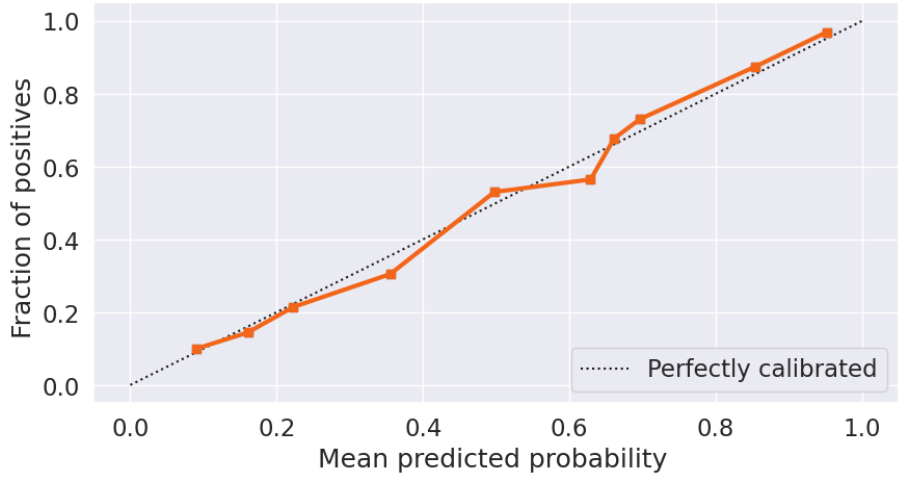


Figure 7.23: XGBoost fit with cross Venn-ABERS calibration.

N **Jupyter Notebook:** Classifier calibration using Venn-ABERS

7.10 Multiclass classification

Say instead of binary classification ($\{0,1\}$) we now have three classes: $\{0,1,2\}$. The *one-vs-rest* (or *one-vs-all*) technique involves training as many binary classifiers as there are classes, where we decompose the data as follows:

- classifier 1: treat class 0 now as class 1, and both classes 1 and 2 as class 0
- classifier 2: treat class 1 as class 1, and both classes 0 and 2 as class 0
- classifier 3: treat class 2 now as class 1, and both classes 0 and 1 as class 0

The final classification result is given by that of the classifier with the highest probability score.

For example (if the kind reader will permit the anthropomorphization), we have a row of test data whose ground truth value corresponds to class 2. We show this row to classifier 1, which was specifically trained to detect class 0 as the positive class, and it says "My prediction for this row is 0.325 thus it is not an example of class 0". We then show this row to classifier 2, which was trained specifically to detect class 1, and it says "My prediction for this row is 0.221 thus it is not an example of class 1". Finally, we show this row to classifier 3, which was specifically trained to detect class 2, and it says "My prediction for this row is 0.827, I really think what we are seeing here is an example of class 2".

7.10.1 Multiclass metrics

Binary metrics can be easily extended for multiclass classification[12]. For example for n classes:

$$\text{precision} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} (TP_i + FP_i)} \tag{7.16}$$

$$\text{recall} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} (TP_i + FN_i)} \tag{7.17}$$

[12] Marina Sokolova, Guy Lapalme "*A systematic analysis of performance measures for classification tasks*", Information Processing & Management **45** pp. 427-437 (2009)

Recommended reading

Papers

- J. S. Cramer *"The Origins of Logistic Regression"*, Tinbergen Institute Working Paper No. 2002-119/4 (2003)
- Tom Fawcett *"An introduction to ROC analysis"*, Pattern Recognition Letters **27** pp. 861-874 (2006)
- Yotam Elor, Hadar Averbuch-Elor *"To SMOTE, or not to SMOTE?"*, arXiv:2201.08528 (2022)
- Telmo Silva Filho, Hao Song, Miquel Perello-Nieto, Raul Santos-Rodriguez, Meelis Kull, Peter Flach *"Classifier calibration: a survey on how to assess and improve predicted class probabilities"*, Machine Learning **112** pp. 3211-3260 (2023)
- Dimitriadis, Gneiting, Jordan, Vogel "*Evaluating probabilistic classifiers: The triptych*", International Journal of Forecasting **40** pp. 1101-1122 (2024)

Books

- Kumar Abhishek, Mounir Abdelaziz "*Machine Learning for Imbalanced Data*", Packt Publishing Limited (2023)
- Valery Manokhin "*Practical Guide to Applied Conformal Prediction in Python*", Packt Publishing (2023) (Chapter 6)

Packages

- imbalanced-learn
- Venn-ABERS calibration by Ivan Petej

8. Ensemble estimators

> ...while the individual man is an insoluble puzzle, in the aggregate he becomes a mathematical certainty.
>
> *Arthur Conan Doyle*

Thus far we have seen individual estimators in action, namely linear regression, the decision tree regressor, logistic regression and the decision tree classifier. However, by carefully aggregating individual estimators into an ensemble much better results can be achieved.

8.1 Random Forest

Random Forest is an example of a "bagging" (bootstrap + aggregating) ensemble estimator. In order to reduce the amount of noise that we fit to (*i.e.* overfitting) as seen with the Decision Tree estimator we average the results of a collection of trees *i.e.* a forest!

- new hyperparameter: `n_estimators` (default = 100)

Suffice to say simply creating 100 trees, each exactly the same, is pointless; we have to mix things up a bit.

8.1.1 Bootstrapping: row subsampling with replacement

The first step is bootstrap sampling to create `n_estimators` datasets, each of the datasets having exactly the same number of rows as the original training dataset. This is done by random sampling *with replacement*, where rows can be sampled multiple times.

```
sample = df.sample(frac=1, replace=True)
```

On average each of these bootstrap datasets will contain $\approx 63\%$ of the rows form the original dataset.

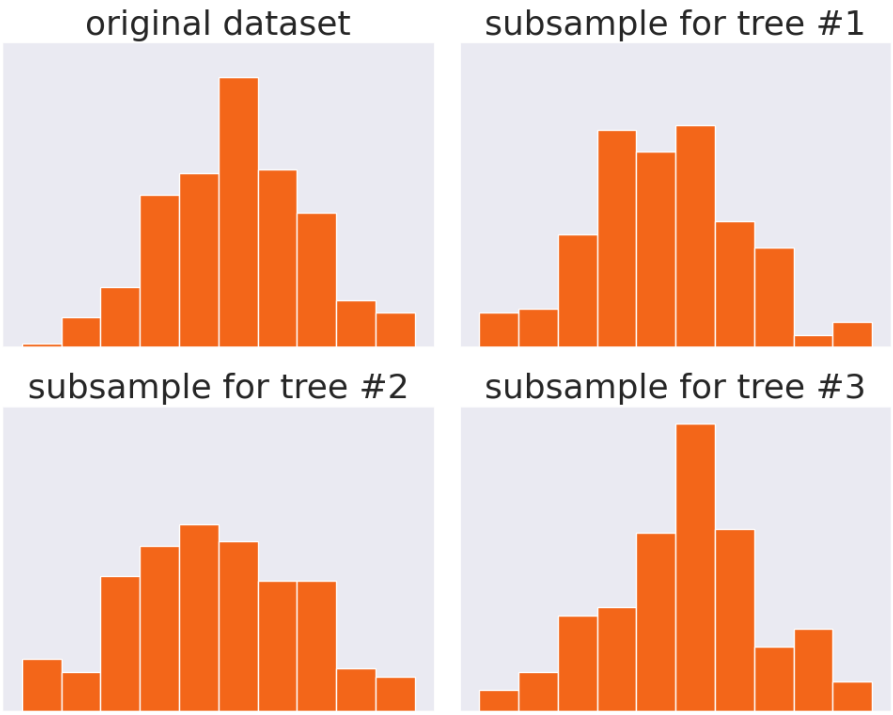


Figure 8.1: Bootstrap sampling with replacement.

out-of-bag samples

Clever use can be made of the unchosen 37% of the rows for each tree; these can be used as a validation set to monitor the generalization performance by setting the parameter `oob_score=True`. Each row of the training data is fed to each tree that did not use said row of data, and the score is recorded and the average result becomes the `oob_score_`. By default `regressor.oob_score_` returns the R^2 metric and the `classifier.oob_score_` returns the accuracy score (both of which I would suggest are unfortunate choices).

8.1.2 Feature subsampling

One can further decorrelate trees by randomly using different subsets of the features in each tree when splitting. In this way each tree is slightly different.

> **Remark:** By default the scikit-learn `RandomForestRegressor` uses all of the features, and the `RandomForestClassifier` uses $\sqrt{\texttt{n\_features}}$. This can be changed using the `max_features` hyperparameter.

Figure 8.2: Each tree in the Random Forest is slightly different.

8.1.2.1 Extremely Randomized Trees

Extremely Randomized Trees goes one step further than Random Forests and, when it comes to choosing the best split, the split for each feature is performed completely randomly. Amazingly this chaotic approach to tree building sometimes performs even better than the more contemplative Random Forest.
(scikit-learn: `ExtraTreesRegressor`/ `ExtraTreesClassifier`).

8.1.3 Results

When making a prediction for a new feature vector the data is given to each of the trees, and the final result is the average of the prediction returned for each tree. In classification the final result can be obtained via the majority vote for each of the trees, or (as in scikit-learn) by averaging the probabilities.

Why does the Random Forest work so well? Each individual decision tree fits to both the signal and to the random noise. Averaged over many trees the random noise component becomes cancelled out, leaving just the signal.

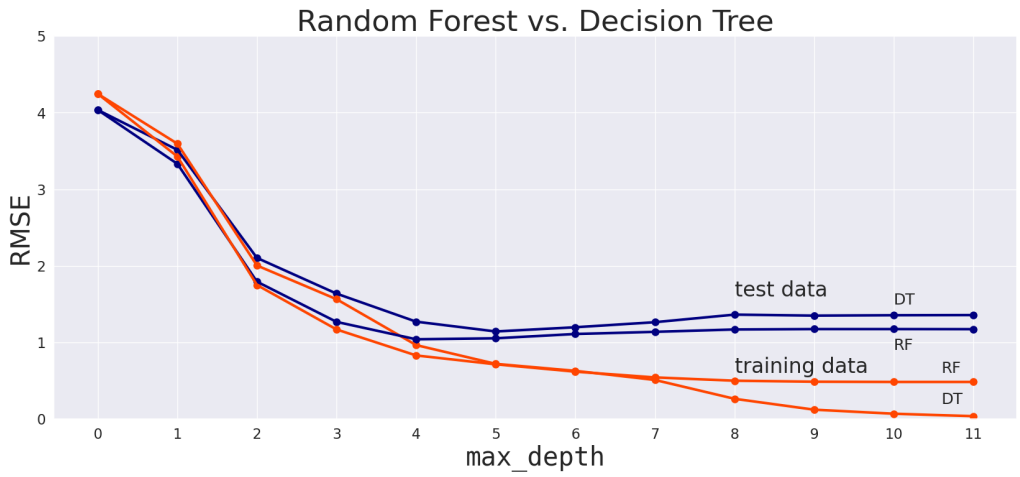


Figure 8.3: Random Forest vs. the decision tree

Remark: With the `RandomForest` one cannot overfit by increasing the number of trees (`n_estimators`) as seen empirically in Figure 8.4.

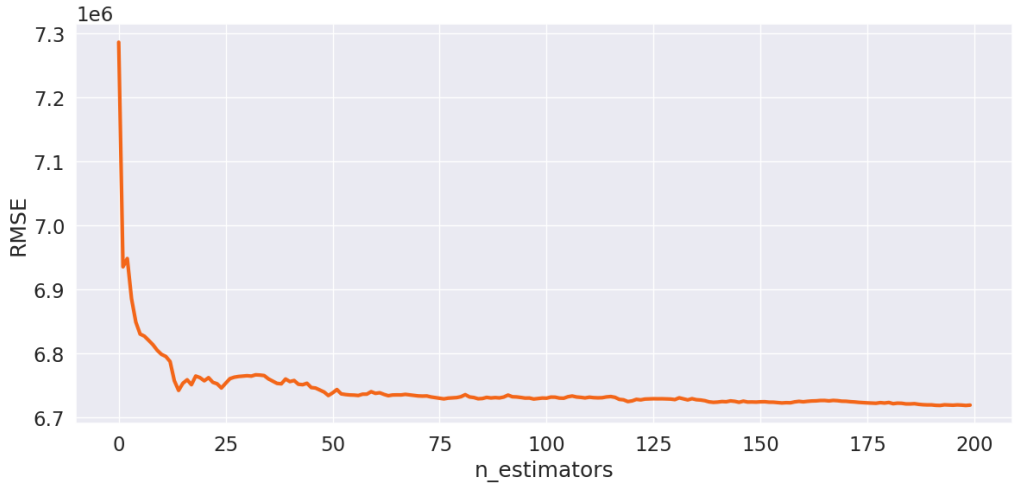


Figure 8.4: Random Forest does not overfit w.r.t. `n_estimators`.

Explainability

With the `RandomForest` we gain in performance, but now we cannot draw our prediction process as a single tree as there are now many of them. We have sacrificed our ability to easily explain the output of our model.

Sample code

```
from sklearn.ensemble import RandomForestRegressor

regressor = RandomForestRegressor(max_depth= 4,
                                  min_samples_leaf= 2,
                                  n_estimators= 100)

regressor.fit(X_train, y_train)
predictions = regressor.predict(X_validation)
```

Remark: XGBoost has Random Forest and is significantly faster than the scikit-learn implementation.

8.2 Weak learners and boosting

In 1989 the question was asked whether a collection of weak estimators, which do little better than random guessing, could constitute a strong estimator and a year later Robert Schapire in the paper *"The strength of weak learnability"* showed that this was indeed the case leading to AdaBoost.

8.2.1 AdaBoost (Adaptive Boosting)

Freund and Schapire invented AdaBoost[1] in 1995 and it led to them winning the prestigious Gödel Prize for 2003. Bagging works in parallel, creating many independent trees. On the other hand boosting is sequential, performing various iterations. Boosting algorithms introduce a new hyperparameter, namely the `learning_rate`. In AdaBoost the base estimator is, just like in Random Forest, a decision tree. However in this case the tree is only of depth 1 (also known as a *decision stump*) (The scikit-learn `AdaBoostRegressor` uses trees of depth 3 as the base estimator). AdaBoost proceeds as follows:

1. fit an initial base estimator to the data
2. fit the next estimator to the errors made by the previous prediction, also applying a weight (the `learning_rate`)
3. repeat until either the requested number of iterations have been performed, or all the errors have become zero (a perfect fit).

When performing classification one works with probabilities (not the class labels).

> **Remark:** Unlike bagging, increasing the number of iterations in boosting can eventually lead to overfitting. Each of the *big four* include the possibility of *early stopping*.

8.3 Gradient boosted decision trees (GBDT)

In gradient boosting each iteration fits a tree to the negative gradient of the loss function (see gradient descent) this time w.r.t the predictions rather than the parameters (this is a non-parametric estimator so we are interested in the predictions and not in finding the values of the best parameters). This is somewhat akin to a predictor-corrector method. Rather than fitting directly to the target GBDT fit to a new target column which is the error w.r.t. the baseline model. Gradient boosted decision trees constitute the *state of the art* (SoTA) in tabular machine learning, and the *big four* implementations are:

- XGBoost - “eXtreme Gradient Boosting" (2014)
- LightGBM - “Light Gradient Boosting Machine" (2016)
- CatBoost - “Categorical Boosting" (2017)
- `HistGradientBoostingRegressor` or the `HistGradientBoostingClassifier` from scikit-learn

The workings of each of these these estimators comes with many bells and whistles, and each takes its own particular approach, and is beyond the scope of this edition of the book. We shall limit ourselves to providing the most basic examples of performing regression (classification is very similar) with each of the four. The hyperparameters provided in each of these examples represent the default values.

```
from sklearn.ensemble import HistGradientBoostingRegressor

regressor = HistGradientBoostingRegressor(max_depth= None,
                                          min_samples_leaf= 20,
                                          max_iter= 100,
                                          learning_rate= 0.1,
                                         )
```

[1] Yoav Freund, Robert E. Schapire *“A decision-theoretic generalization of on-line learning and an application to boosting”*, Computational Learning Theory pp 23-37 (1995)

```
regressor.fit(X_train, y_train)
```

XGBoost has a version that uses the Scikit-Learn API:

```
import xgboost as xgb

regressor = xgb.XGBRegressor(max_depth= 6,
                             min_child_weight= 1,
                             n_estimators= 100,
                             learning_rate= 0.3,
                            )
regressor.fit(X_train, y_train)
```

```
from lightgbm import LGBMRegressor

regressor = LGBMRegressor(max_depth= -1, # no limit
                          min_child_samples= 20,
                          n_estimators= 100,
                          learning_rate= 0.1,
                          verbose= -1
                         )
regressor.fit(X_train, y_train)
```

```
from catboost import CatBoostRegressor

regressor = CatBoostRegressor(depth= 6,
                              min_data_in_leaf= 1,
                              iterations= 1000,
                              #learning_rate is automatic
                              silent= True
                             )
regressor.fit(X_train, y_train)
```

Remark: If you increase the number of iterations, you should decrease the learning rate.

Suffice to say each of these estimators has much more functionality and many more parameters to tune, and can be used with GPU's to speed up training, especially if one has a very large dataset. Much has been written as to which is the *best of the best*. Personally I would suggest simply trying all four out on ones particular dataset, and choosing based of performance, as per the 'No free lunch' theorem.

Jupyter Notebook: An introduction to XGBoost regression

8.3.1 Extrapolation

As we have seen earlier in Section 6.6 extrapolation is something we wish to avoid. It is worth mentioning that even the mighty SoTA estimators have problems with extrapolation. In Figure 8.5 we fit to the function $y = x$. In this case, given the linear nature of the data unsurprisingly a linear regressor fits the data perfectly, and intuitively we see that it also extrapolates as expected. However, XGBoost does dot fare so well.

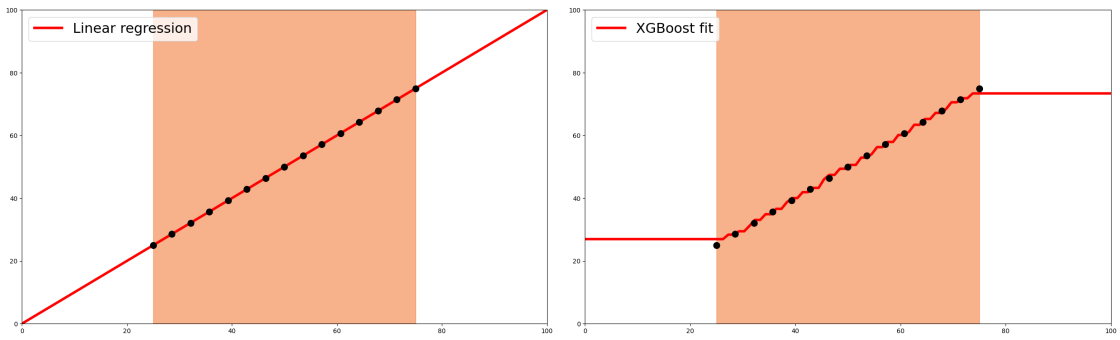


Figure 8.5: Extrapolation with XGBoost

This can be understood in terms of our very first experience with the Decision Tree Regressor (the building blocks of gradient boosting decision trees); the extremal plateaux remain constant regardless of how far we travel either side from the region they were fitted on.

N **Jupyter Notebook:** Extrapolation: Do not stray out of the forest!

Although not part of this book this is particularly pertinent to time series problems. In Figure 8.6 we fit to a sinusoidal function, and predict for a similar function that has drifted. We can see that the predictions are clipped to the maximum value that was seen in the training data.

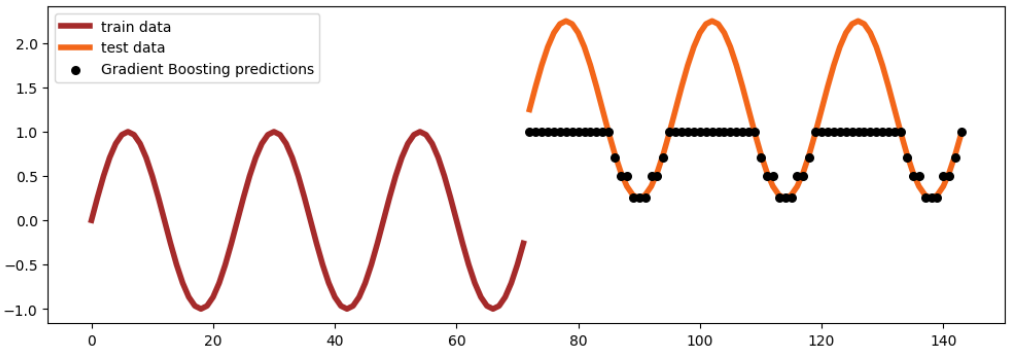


Figure 8.6: Drift in a time series with clipped predictions

So-called out-of-distribution generalization[2] is a complex issue and beyond the scope of this book.

[2] Liu, Shen, He, Zhang, Xu, Yu, Cui "*Towards Out-Of-Distribution Generalization: A Survey*", arXiv:2108.13624 (2023)

8.4 Convex combination of model predictions (CCMP)

A trick that sometimes results in an, albeit often small, performance boost is to ensemble diverse[3] estimators, such as a parametric model with a non-parametric model. A simple method for regression is to create a new set of predictions from a convex combination of the predictions of the constituent models. In Figure 8.7 and Table 8.1 we can see the results of combining 50% of the output value of a linear regression with 50% of the decision tree regression.

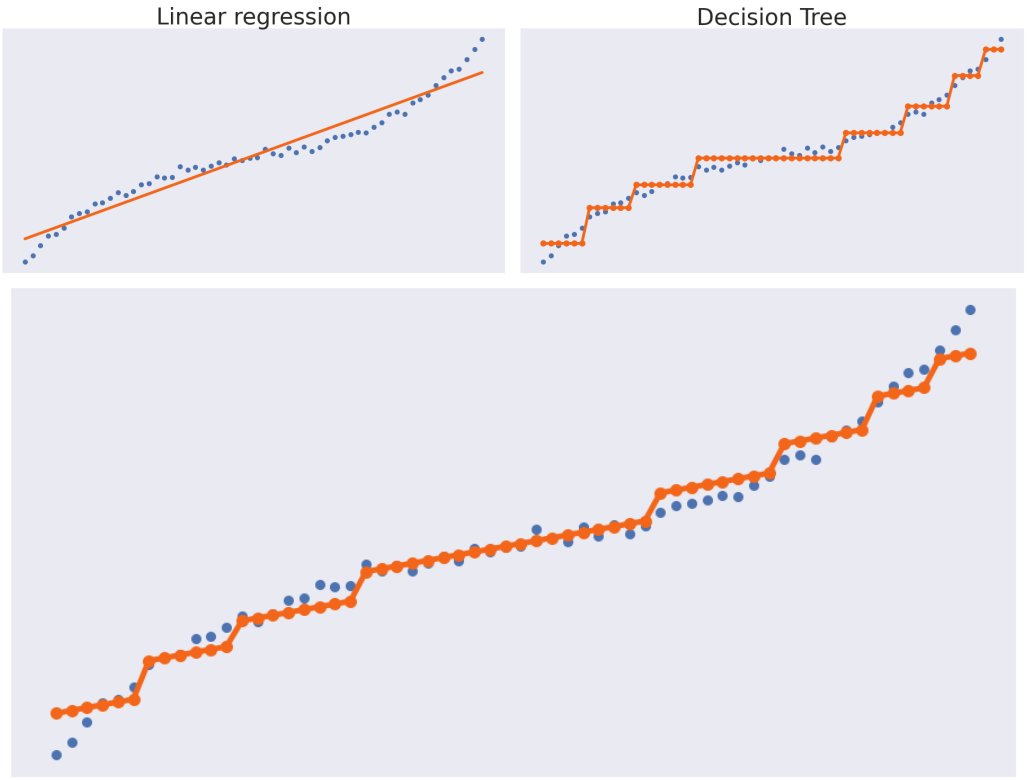


Figure 8.7: Ensembling of a linear regression and a decision tree

Model	RMSE
Linear regression	3.96
Decision tree	2.59
CCMP ensemble	2.36

Table 8.1: Ensemble results

Visually we can see where the performance boost came from; the linear regressor added inclination to the horizontal plateaus of the decision tree, leading to a better overall fit to the data.

[3] Wood, Mu, Webb, Reeve, Luján, Brown "*A unified theory of diversity in ensemble learning*", Journal of Machine Learning Research **24** pp. 17302-17350 (2023)

Sample code

```
df["y_pred"] = 0.5*df["y_pred_LR"] + 0.5*df["y_pred_DT"]
```

The optimal weights can be found by fitting either to a hold-out dataset or to the out-of-fold (OOF) predictions obtained from cross validation by using the hill climbing optimization technique (see the `hillclimbers` package).

N **Jupyter Notebook:** Ensembling: Convex combination of model predictions (CCMP) and the hillclimbers package

For classification one can combine the the calibrated probabilities for each classifier by minimizing the log loss w.r.t. the class labels.

Remark: For classification if working directly with class labels, rather than CCMP one can combine predictions by taking the mode (majority vote) of an odd number of classifiers.

8.5 Stacking

In stacking[4] firstly one saves each of the set of out-of-fold predictions for the estimator and then combine them all into a complete column of predictions and save this column of predictions as a feature in a new dataset. We then repeat this for another model, leading to another column. We can use as many models as we like. Next we fit a simple 'meta' estimator (say linear regression or logistic regression so as not to overfit) to this new n column dataset where n is the number of models that we used. When it comes to prediction time we calculate the predictions for each of the models, and again save these predictions as features in a new dataset. Finally or final results is the prediction of our simple estimator for this new n column dataset.

[4] David H. Wolpert "*Stacked generalization*", Neural Networks **5** pp. 241-259 (1992)

Recommended reading

Papers

- Leo Breiman "*Random Forests*", Machine Learning **45** pp. 5-32 (2001)
- Robert E. Schapire "*The Boosting Approach to Machine Learning: An Overview*", Nonlinear Estimation and Classification **171** pp. 149-171 (2003)
- Jerome H. Friedman *"Greedy function approximation: A gradient boosting machine"*, The Annals of Statistics **29** pp. 1189-1232 (2001)

Books

- Gautam Kunapuli *"Ensemble Methods for Machine Learning"*, Manning Publications (2023)

Packages

- XGBoost
- LightGBM
- CatBoost
- linear-tree by Marco Cerliani
- hillclimbers by Matt Hill

Other

- MLWave *"Kaggle Ensembling Guide"*

9. Hyperparameter optimization

"knobs to twiddle"

Brad Efron

So far we have seen

- λ for Ridge and LASSO regularization
- the depth of a tree: `max_depth`
- the number of trees: `n_estimators`
- the learning rate
- `min_samples_leaf`
- ...

which are all examples of hyperparameters. The parameters are the values found by the model whilst fitting to the data, for example the β in linear regression. On the other hand hyperparameters help to define the model one will fit, and are chosen before the fitting process begins. How does one find the optimal set of hyperparameters? One can perform a manual search, but that can be rather tedious. Scikit-learn provides the following three routines:

- `sklearn.model_selection.GridSearchCV`
- `sklearn.model_selection.RandomizedSearchCV`
- `sklearn.model_selection.HalvingGridSearchCV`

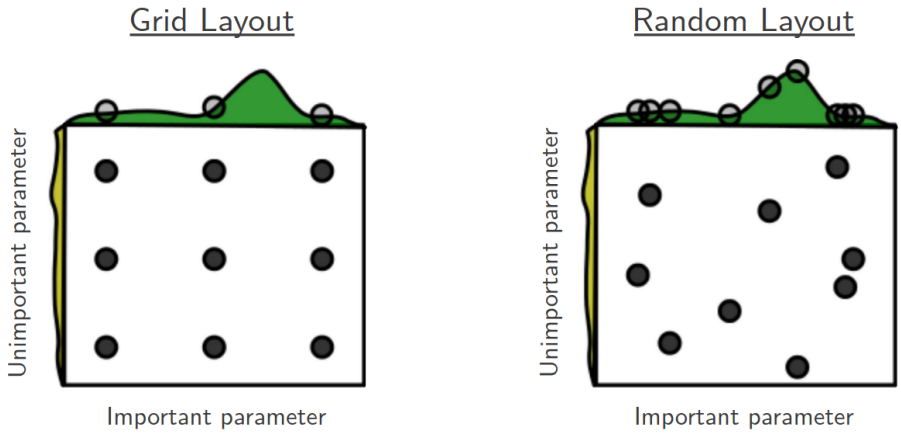


Figure 9.1: Grid search vs. random search for nine trials (Image credit: James A. Bergstra, Yoshua Bengio *"Random search for hyper-parameter optimization"*, The Journal of Machine Learning Research **13** pp. 281-305 (2012))

Perhaps the best suited of the three routines for a tree estimator, many of whose hyperparameters are integers, is the exhaustive grid search. Meanwhile the `RandomizedSearch` randomly selects only `n_iter` of the possible hyperparameter combinations. For a neural network, whose hyperparameters can have continuous values, a random (perhaps Bayesian) search is better. The halving grid search initially evaluates all of the parameter combinations, just as with the exhaustive grid search, however, using only a very small subset of the training data, thus is much faster. Based on the results, it then selects the, say, top 30% best performing parameter sets, and recalculates the results, now using more rows from the training data, and so on.

Sample code

```
from sklearn.model_selection import GridSearchCV

param_grid = {"max_depth": [3, 4, 5],
              "min_samples_leaf": [2, 3, 4],
              "n_estimators": [100, 300, 1000],}

search = GridSearchCV(regressor, param_grid, cv=5).fit(X_train,y_train)

print("The best hyperparameters: ",search.best_params_)
```

Remark: The above grid search will run a total of $(3 \times 3 \times 3) \times 5 = 135$ fits, so if for example a single calculation takes just 1 minute to run be aware that this grid search would take over 2 hours to complete!
Tip: Set `n_jobs=-1` to run in parallel using all available processors.

Recommended reading

Papers

- James A. Bergstra, Yoshua Bengio *"Random search for hyper-parameter optimization"*, The Journal of Machine Learning Research **13** pp. 281-305 (2012)
- Tong Yu, Hong Zhu *"Hyper-Parameter Optimization: A Review of Algorithms and Applications"*, arXiv:2003.05689 (2020)

Packages

- Hyperopt: Distributed Hyperparameter Optimization
- Optuna: A hyperparameter optimization framework
- scikit-optimize `BayesSearchCV`

Other

- scikit-learn "Tuning the hyper-parameters of an estimator"

10. Feature engineering and selection

> The greatest value of a picture is when it forces us to notice what we never expected to see.
>
> *John Tukey*

If one has ever had the pleasure of seeing an armillary sphere based on the geocentric model (there is a magnificent example in the library of the Real Monasterio de San Lorenzo de El Escorial I would be happy to show if you are passing through Madrid) and compares it to an orrery based on the heliocentric model one will appreciate the relative simplicity of calculations in the Copernican system. When it comes to feature engineering we need be like Copernicus, we need to create features that help the estimator to find simplicity in the data. Feature engineering is about creating new, more predictive features out of either the features we have been given, or by adding features from an external or secondary data source.

On the other hand feature selection is about deleting either useless, noisy, or low value features, leading to a simpler more robust model. Deleting features also leads to more memory efficient model with faster predictions. This can also result in cost savings, as some features may be very expensive to obtain.

10.1 Feature engineering

Feature engineering can work wonders for model performance, and is perhaps the aspect of machine learning that requires the most craftsmanship, with 'domain knowledge' often playing a large role. We have already seen a magnificent example of feature engineering in Section 6.5 where we created the x^2 term form the x term, which immediately led to a much better fit.

10.1.1 Interaction and cross features

Particularly good are creating cross or interaction features

- non-linear features *i.e.* x^2, $x^{1/n}$,...

- interactions between features *i.e.* $(x_1 + x_2)$, $(x_1 \times x_2)$,...

For example

```
df["area"] = df["length"] * df["breadth"]
df["disposable_income"] = df["income"] - df["taxes"]
```

One can use the scikit-learn `PolynomialFeatures` to help with this process but if one has many features in ones dataset this could easily get out of hand, and I would suggest creating the interaction features by hand, based on common sense and domain knowledge.

10.1.2 Bucketing of continuous features

One can perform discretization or *bucketing* of continuous features prior to modeling, for example via the scikit-learn `sklearn.preprocessing.KBinsDiscretizer`, which has options such as 'uniform', 'quantile', or 'kmeans'. However, I would strongly suggest this process is actually best left in the hands of ones tree model.

10.1.3 Power transforms: Yeo-Johnson

Variance-stabilizing transformations can potentially be useful for parametric models. For example converting a feature having the positive skewed Galton distribution into a Gaussian distribution: `df["log_x1"] = np.log(df["x1"])`. (Note that $\ln(0) = -\infty$, so watch out for zero-inflated features, where one could perhaps take the liberty of converting the zeros into ones before transforming). A Yeo-Johnson transform can be used to correct skewness or kurtosis (scikit `power_transform`). Such a transform may help in interpreting linear models that assume a Gaussian distribution of errors. However, feature transformations probably will not enhance the predictive performance of a linear model, and certainly will not help tree based estimators.

10.1.4 User defined transform

If one wishes to apply a bespoke transformation an option is to do so using a function

```
def fn(x):
    # y=f(x) here
    return y

df["new_feature"] = df["old_feature"].apply(fn)
```

For example, one may have origin-destination data in terms of latitude and longitude features, but in some cases converting these into relative distances may be more informative (see the `vincenty` package).

10.1.5 External secondary features

Adding external data can also lead to a significant improvement in model performance. An example could be in modeling taxi usage; maybe adding a feature based on the weather could be informative as people may be more inclined to make use of a taxi service if it is raining, and less inclined if it is sunny. This data could be scraped from a meteorological service. Another useful feature could be to add a categorical column that indicates if the day was a weekday, and weekend, or a local or national holiday (one could use `dateutil.easter` for adding the Easter holidays to ones feature).

10.2 Feature selection

Feature selection is the removal of weak features from the model, leading to a more robust model with less potential for overfitting and faster to train. The very first thing to do is to drop features that have zero variance, and also perfectly (or very highly) correlated features.

10.2.1 Correlation

Remark: Some statistical texts recommend dropping features that are uncorrelated with the target variable based on the linear parametric Pearson correlation coefficient. However, if one is using a non-linear/non-parametric model I would advise against doing this as we have seen that seemingly uncorrelated features can indeed contain mutually useful information.

10.2.2 Permutation importance

Permutation importance is performed as follows:

1. calculate model metric score which becomes our point of reference
2. randomly reorder the values of a single column (feature)
   (*i.e.* `df['feature_n'] = df['feature_n'].sample(frac=1).values`)
3. re-calculate the metric score for the new `df`
4. the importance of the feature is proportional to how much the metric score degrades
   (*i.e.* if the feature was unimportant the metric score would not degrade at all)
5. reset the column and repeat for each of the columns in the dataset
6. produce an importance ranking based of relative score degradation

	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	Target
0	0	0	9	0	0	0	0	0	0	0
1	1	1	5	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2
3	3	3	0	3	3	3	3	3	3	3
4	4	4	3	4	4	4	4	4	4	4
5	5	5	6	5	5	5	5	5	5	5
6	6	6	4	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8	8
9	9	9	1	9	9	9	9	9	9	9

Figure 10.1: Permutation importance: a `df` with a randomly shuffled column.

This is an efficient technique, only requiring as many calculations as there are features as well as the baseline calculation. There is a caveat that should be mentioned in that, especially if there are highly correlated features, the act of random permutation on such features will create a dataset that has points outside of the convex hull of data that the model was trained on, *i.e.* we end up extrapolating which leads to unreliable results especially for tree models[1].

[1] Giles Hooker, Lucas Mentch, Siyu Zhou *"Unrestricted Permutation forces Extrapolation: Variable Importance Requires at least One More Model"*, arXiv:1905.03151 (2021)

Jupyter Notebook: House Prices: Permutation Importance example.

Sample code

Example of performing permutation importance using ELI5

```
import eli5
from eli5.sklearn import PermutationImportance

feature_names = X_train.columns.tolist()
perm_import = PermutationImportance(regressor,
                random_state= 42).fit(X_train, y_train)
eli5.show_weights(perm_import, top=None, feature_names)
```

Weight	Feature
1.1113 ± 0.0215	m2_edificados
0.3970 ± 0.0121	precio_zona
0.0194 ± 0.0013	precio_alquiler
0.0065 ± 0.0006	ascensor
0.0019 ± 0.0004	exterior
0.0007 ± 0.0001	aparcamiento
0.0004 ± 0.0001	n_habitaciones
0.0003 ± 0.0002	m2_útiles
0.0002 ± 0.0001	orientación_este
0.0002 ± 0.0002	precio_aparcamiento
0.0002 ± 0.0001	nueva_construcción
0.0001 ± 0.0001	orientación_oeste
0.0001 ± 0.0000	orientación_norte
0.0001 ± 0.0000	planta
0.0001 ± 0.0001	n_baños
0.0000 ± 0.0000	orientación_sur
0.0000 ± 0.0000	año_construción
0.0000 ± 0.0000	necesita_reforma

Figure 10.2: Permutation importance output from ELI5.

Remark: The relative importance of the features will vary between estimators.

10.2.3 Stepwise regression

Stepwise regression techniques either go forward by starting from no features and accreting new features until no improvement is seen. Or they can go backward, deleting features whose removal do not degrade the model. Choosing the features based on statistical significance has long been known to be problematic[2] and stepwise regression should be based on out-of-fold or out-of-sample results, *i.e.* by monitoring the effect on the validation set. However, one should still be wary, due to the potentially large number of test performed, of overfitting to the validation set.

[2] Gary Smith *"Step away from stepwise"*, Journal of Big Data **5** 32 (2018)

Recursive feature elimination (RFE)

Choose the number of features one wishes to select (n) then:

1. calculate the importance of each feature, for example via the RandomForest `feature_importances_`
2. eliminate the weakest feature
3. repeat until n features remain

I would suggest setting `n_features_to_select` parameter to be 1 to produce a ranking, then remove the weakest features one by one by hand, all the while monitoring validation performance. Scikit-learn also has the `RFECV` routine that performs RFE in conjunction with cross-validation.

Jupyter Notebook: "Recursive Feature Elimination (RFE) example"

10.2.4 LASSO

The LASSO regularization can be used to eliminate weak features. Unlike Ridge regression, the LASSO can effectively reduce coefficients to become zero, at which point they no longer form part of the model.

Jupyter Notebook: Feature importance using the LASSO

10.2.5 Boruta trick

The 'Boruta' trick[3] consists of (temporarily) inserting a number of random noise features into ones training dataset. Any feature whose importance is lower than any one of these nose features is, by extension, also essentially noise, and can be dropped.

Sample code

```
df['RANDOM_1'] = np.random.normal(size=len(df))
df['RANDOM_2'] = np.random.normal(size=len(df))
df['RANDOM_n'] = np.random.normal(size=len(df))
```

Indeed one can be surprised by how spuriously important a random feature can sometimes be in a small dataset.

Jupyter Notebook: "Feature selection using the Boruta-SHAP package"

10.2.6 Native feature importance plots

RandomForest `feature_importances_`

The scikit-learn RandomForest estimators upon fitting can provide `feature_importances_` based on the mean decrease in impurity.

[3] Stoppiglia, Dreyfus, Dubois, Oussar *"Ranking a random feature for variable and feature selection"*, Journal of Machine Learning Research **3** pp. 1399-1414 (2003)

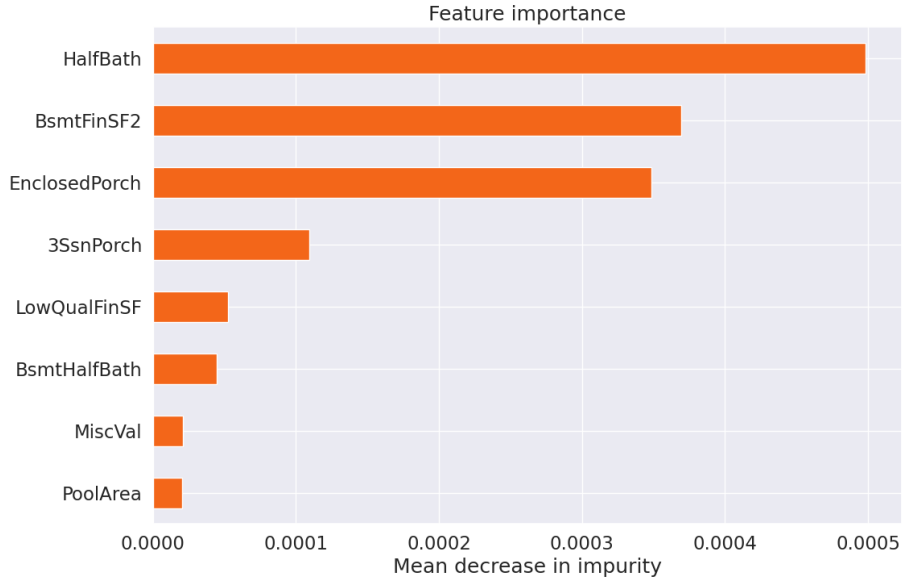


Figure 10.3: RandomForest feature importance plot

Sample code

Here we plot the 8 most important features for our `RF_regressor`:

```
feature_importances = RF_regressor.feature_importances_
feature_names = X_train.columns.tolist()
importances = pd.Series(feature_importances, index=feature_names)
Top_8 = importances.sort_values(ascending=True).head(8)

fig, ax = plt.subplots()
Top_8.plot.barh()
ax.set_title("Feature importance")
ax.set_xlabel("Mean decrease in impurity")
plt.show()
```

XGBoost feature importance

XGBoost provides a similar plot via `plot_importance` method, this time reporting something called the 'F score' which is *"...based on the number of times that variable was selected for splitting in the tree weighted by the squared improvement to the model as a result of each of those splits*[4]". The great thing about both of these plots is that they require no extra computation, although they are not as trustworthy as out-of-sample methods.

Sample code

Here we plot the 8 most important features for our `XGB_regressor`:

```
from xgboost import plot_importance
```

[4] Jerome H. Friedman, Jacqueline J. Meulman "*Multiple additive regression trees with application in epidemiology*", Statistics in Medicine **22** pp. 1365-1381 (2003)

```
fig, ax = plt.subplots(figsize=(12,8))
plot_importance(XGB_regressor,
                max_num_features=8, # Top 8
                importance_type='weight',
                ax=ax)
plt.show();
```

N **Jupyter Notebook:** "An introduction to XGBoost regression"

10.3 Principal component analysis (PCA)

Figure 10.4: PCA using SVD in action.

Principal component analysis (PCA) was invented by Karl Pearson in 1901. It is interesting in that it is a technique that both creates new features, and at the same time can reduce the number of features (dimensionality reduction) that we use in our model. For example, one could have an n-dimensional dataset, having n features, and reduce the dataset all the way down to 1-dimension by creating a feature consisting of just the first principal component. However, when it comes to making predictions one still need access to all of the original features so as to re-create the same principal component(s) to feed into the trained model. So, although PCA reduces the number of dimensions, it is not to be considered a feature elimination technique.

It should be mentioned that PCA has notable limitations; the feature-mixtures in the components created by PCA become uninterpretable (see Figure 10.4). Furthermore a recent study has found[5]:

> *. . . we carried out extensive analyses on twelve PCA applications, using model- and real-populations to evaluate the reliability, robustness, and reproducibility*

[5] Eran Elhaik "*Principal Component Analyses (PCA)-based findings in population genetic studies are highly biased and must be reevaluated*", Scientific Reports **12** 14683 (2022)

> *of PCA. We found that PCA failed in all criteria and showed how easily it could generate erroneous, contradictory, and absurd results.*

so use PCA with caution.

Recommended reading

Papers

- Zhao, Anand, Wang "*Maximum Relevance and Minimum Redundancy Feature Selection Methods for a Marketing Machine Learning Platform*", arXiv:1908.05376 (2019)

Books

- Alice Zheng, Amanda Casari "*Feature Engineering for Machine Learning*", O'Reilly Media, Inc. (2018)
- Max Kuhn, Kjell Johnson "*Feature Engineering and Selection: A Practical Approach for Predictive Models*", CRC Press (2019)
- Soledad Galli "*Python Feature Engineering Cookbook*", Packt Publishing Limited (2022)

Packages

- Boruta-Shap
- Featuretools
- Feature Engine
- LOFO (Leave One Feature Out) Importance by Ahmet Erdem
- mRMR - minimum Redundancy Maximum Relevance

11. Why no neural networks/deep learning?

> There are two kinds of fools: one says, "This is old, therefore it is good"; the other says, "This is new, therefore it is better."
>
> *William Ralph Inge (1931)*

When it comes to tabular data it has been found that artificial neural networks (ANN) are, quite literally, a waste of time.

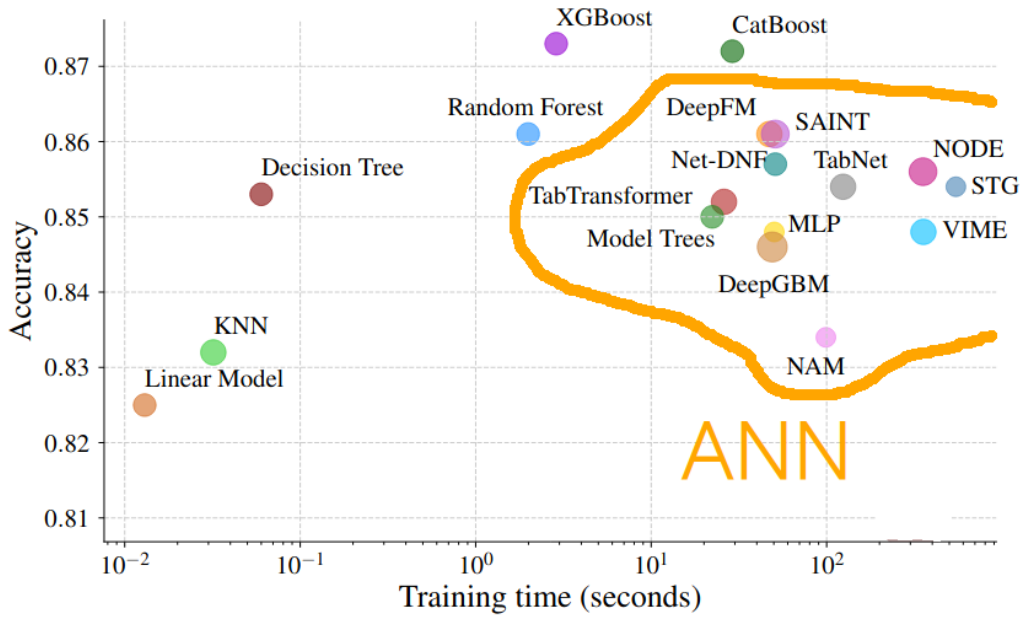


Figure 11.1: (Image credit: adapted from Borisov, Leemann, Seßler, Haug, Pawelczyk, Kasneci, *"Deep Neural Networks and Tabular Data: A Survey"*, IEEE Transactions on Neural Networks and Learning Systems **35** pp. 7499-7519 (2024)).

In Figure 11.1, which plots performance against time (on a logarithmic scale), to the left we can see our linear and decision tree models and on the RHS are neural network estimators which at best on par with the Random Forest, and none achieving the SoTA performance of XGBoost or CatBoost seen at the very top of the plot. This is potentially because tabular data does not form a smooth continuous hypersurface, due to noisy data, missing values, categorical features, features with disparate ranges (for example `salary` and `age`, *etc.*) all working against the manifold hypothesis which neural networks make use of. In view of these performance and computational cost issues, unless one has a very good reason for wanting to use neural networks for tabular data I would be very much inclined to suggest sticking to the classical techniques we have seen earlier. That said, if CPU/GPU time is no object one could try either CCMP or stacking of a gradient boosted decision tree with a neural network to squeeze out a little bit more performance as neural network estimators can almost always form a very valuable component of an ensemble.

On the other hand for non-i.i.d. data the ability of neural networks to extrapolate plays in their favour when it comes to forecasting time series data, where each row of data is highly correlated with the previous row of data, outperforming GBDT. For the reader interested in both time series and neural networks the text by Manu Joseph "Modern Time Series Forecasting with Python" is an excellent read.

Just for fun the following are code snippets to perform univariate linear regression and logistic regression both using a single neuron. Just as with the statistical linear and logistic regression routines the data should be scaled beforehand.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(x.values.reshape(-1, 1))
```

11.0.1 Single neuron regressor

```
import tensorflow
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers

# keras model
model = Sequential()
# input layer
n_neurons = 1
model.add(Dense(n_neurons, input_dim=1, activation=None))
optimizer = tensorflow.keras.optimizers.SGD(learning_rate=0.3)
model.compile(loss='mean_squared_error', optimizer=optimizer)

# train the model and predict
model.fit(X_scaled.reshape(-1, 1), y, epochs=25, verbose=0)
y_pred  = model.predict(X_scaled.reshape(-1, 1))
```

11.0.2 Single neuron classifier

```
import tensorflow
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers

# keras model
model = Sequential()
# input layer
n_neurons = 1
model.add(Dense(n_neurons, input_dim=1, activation='sigmoid'))
optimizer = tensorflow.keras.optimizers.SGD(learning_rate=0.3)
model.compile(loss='binary_crossentropy', optimizer=optimizer)

# train the model and predict
model.fit(X_scaled.reshape(-1, 1), y, epochs=2000, verbose=0)
y_pred = model.predict(X_scaled.reshape(-1, 1))
```

Recommended reading

Papers

- Ravid Shwartz-Ziv, Amitai Armon *"Tabular Data: Deep Learning is Not All You Need"*, arXiv:2106.03253 (2021)
- Grinsztajn, Oyallon, Varoquaux *"Why do tree-based models still outperform deep learning on typical tabular data?"*, arXiv:2207.08815 (2022)
- Borisov, Leemann, Seßler, Haug, Pawelczyk, Kasneci, *"Deep Neural Networks and Tabular Data: A Survey"*, IEEE Transactions on Neural Networks and Learning Systems **35** pp. 7499-7519 (2024)

Packages

- Keras
- TabNet by Sebastien Fischman
- PyTorch Tabular by Manu Joseph
- TabPFN

Essential reading

Books

- Trevor Hastie, Robert Tibshirani, Jerome Friedman *"The Elements of Statistical Learning"*, Springer (2017)
- Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, Jonathan Taylor *"An Introduction to Statistical Learning"*, Springer (2023)
- Christopher M. Bishop *"Pattern Recognition and Machine Learning"*, Springer (2006)

Articles

- Sebastian Raschka *"Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning"*, arXiv:1811.12808 (2020)

For many more excellent texts visit the compendium of free ML reading resources.

See also

- Carl McBride Ellis *"A selection of my Kaggle notebooks"*