The Engineering Guide to Machine Learning & Artificial Intelligence

Daniël Heres

Version 0.1.1
`https://github.com/real-ai/ml-guide`

May 29, 2019

Contents

Chapter 1

Introduction

What are the factors that makes Machine Learning projects successful? What should you focus on and what can be solved by tools? How do we bring models into production? Which skills do I need to develop to become productive? How do we tune models? How can we maintain models over time and understand the real time impact? How can we maintain and improve a model over time?

Machine Learning, being a relatively new field in industry, has many of these questions still open. Where more older profession in technology, such as Software Engineering, has developed lots of tools, patterns and ideas around it, in Machine Learning things are much more evolving and open. Also, as the field moves very fast, tools and ideas can become quickly outdated.

In this document I give an introduction into how to apply Machine Learning in practical settings. It is by no means complete or finished, and will need to be updated to stay actual.

In this guide we focus on the engineering side of machine learning, answering questions like these. I distill patterns that can be applied to the development cycle and the design and use of popular machine learning tools.

Chapter 2

Data Collection

2.1 Labeling

Currently, the collection of labeled data is an important part of the machine learning practice.

Today's algorithms often need both data of a certain quality (e.g. should be very similar to the) and a certain quantity (the best algorithms need > 1 million samples).

Even though in Machine Learning research approaches such as Transfer Learning and Self-supervised Learning reduce the need for labeled data, the need for large labeled data sets for accurate models is still big.

There are a couple of approaches towards collecting labeled data.

- **Automated**. This can be the case whenever there is existing historical data available, e.g. the number of page views at a certain day or the number of times a certain hashtag is used on Twitter. Although humans are part of the process, we don't need to manually collect the labels.
- **Semi-automated**. A human will provide feedback to a system's automatic suggestions. This example can then be used to improve the model's predictive performance and/or for measuring the performance. Sometimes labels can be collected without explicitly labeling
- **Manual**: Humans will completely label every example without the help of a computer.

2.2 Labeling Platforms

To collect labeled data, there are platforms and ecosystems. One of the biggest platform is Amazon Mechanical Turk `https://mturk.com`. This platform

2.3 Labeling by Customers

A common strategy used by companies is to use an existing product to collect labels from customers, instead of paying per label.

This way, we can vastly expand our labeled data, and improve our existing labeled data by finding consensus in labeling with minimal cost.

2.4 Bootstrapping

Chapter 3

Learning and modeling

Currently, the most successful and accurate models are using Supervised Learning. This means that a model is learned, from scratch, on a set of labeled data.

In this chapter we will focus on methods, strategies to apply and verify machine learning algorithms in the real world.

3.1 Train, validate & test

3.2 Cross-validation

3.3 Raw Feature Models

3.4 Image Models

3.5 Text Models

3.6 Sequence Models

3.7 Search & Ranking

3.8 Clustering

3.9 Anomaly Detection

Chapter 4

Feature Engineering

Although we are moving more and more towards the fully automatic learning. Especially with numerical data (such as financial, sales-numbers, etc) feature engineering is an crucial part of models.

4.1 Normalization

Normalization is an important part of making features. Lots of algorithms such as neural networks work better when features are normalized.

For neural networks, the most important property of features is that the range of different features should be similar.

One basic way for normalization is to transform the features to have unit zero mean:

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

Where μ is the mean and σ the standard deviation.

Depending on the distribution of data, we need different ways of transformation. E.g. for a variable that is more exponential in nature, using a logarithm to transform it back. to a more linear distribution helps to make it a more useful feature to learning algorithms.

Similarly, if the data you have is quadratic, then taking the square root of the variable helps optimization algorithms to learn in a more stable manner.

4.2 Cyclical Features

Cyclical features are very common in data involving time: we have often access to a timestamp or a day variable. Often there is a periodic pattern in this data. We want to use the fact that two times are similar when they are close to each other. E.g. 11:59 PM is very close to 00:00 AM. However, naively using the minutes since 00:00 AM as feature will have those features maximally apart!

blue	1,0,0,0
black	0,1,0,0
white	0,0,1,0
red	0,0,0,1

Table 4.1: One hot encoding

One good way of using the time as a feature is to project them on a circle using sine and cosine.

If we have a (Unix) timestamp variable with the number of seconds, projecting it to a circle for minutes in an hour, hours in a day, and days in a week is easy:

$$\begin{aligned} min_{sin} &= \sin(\frac{t \cdot 2\pi}{3600}) \\ min_{cos} &= \cos(\frac{t \cdot 2\pi}{3600}) \\ hour_{sin} &= \sin(\frac{t \cdot 2\pi}{24 \cdot 3600}) \\ hour_{cos} &= \cos(\frac{t \cdot 2\pi}{24 \cdot 3600}) \\ day_{sin} &= \sin(\frac{t \cdot 2\pi}{24 \cdot 3600 \cdot 7}) \\ day_{cos} &= \cos(\frac{t \cdot 2\pi}{24 \cdot 3600} \cdot 7) \end{aligned} \tag{4.2}$$

We can do the same for yearly patterns, but the usefulness will depend on how many years of data you have.

4.3 Categorical Embeddings

If you have categorical variables. One basic way is to use One Hot Encoding.

Some examples of categories are

- Shoe color (blue, black, white, red, ...)
- A word (”apple”, ”banana”, ”fruit”, ”dog”)
- A country
- A user id
- A book author

A classic way of encoding them is using one-hot-encoding: a mapping from category to an n-dimensional vector where one of the values is unique (see table 4.1.

Figure 4.1: Thermometer 8.5/10

This however has a few downsides: because the vectors grow with the number of categories, if we have a large number of categories, this will use a lot of memory & wastes computation when used in a dense "way". Furthermore, every vector is as far as any other vector, so similar categories (e.g. similar colors, words, users) are not close together.

An efficient way of encoding is to use an Embedding: a lookup table with a n-dimensional vector for each category. This is efficient: we only need to retrieve the category or categories used for a certain training example.

Also, embeddings can be pre-trained on data other than that from the training task using Transfer Learning. Say if you for example want to train a text classification model, we fan

4.4 Thermometer Encoding

When we have a feature bounded between two numbers we can. However, for example in linear regression, we will fit a linear line. If we want to find any other relation. However: we can apply a non-linear function to the input data, to still be able to find non-linear relation ships in data.

The idea of thermometer encoding (also called unary encoding) is to transform one feature into n features, where each feature will be active at a certain threshold where each feature holds roughly the same amount of data points.

For example, when we have a variable from 0 to 10 and we want to transform it to a thermometer using stepsize of 1, the thresholds are at the values $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Between the buckets, we can interpolate the values to avoid removing information.

A value 8.5 can be visualized as this "thermometer":

An implementation in NumPy:

```python
def thermometer(x, start, end):
    thresholds = np.arange(start, end)
    thermo = (x > thresholds).astype(float)
    thermo[np.arange(len(x)),
        (np.floor((x - start))).astype(int).reshape(len(x))
        ] = np.fmod(x, 1.0).reshape(len(x))
    return thermo
```

In Figure 4.2 we can see how thermometer encoding and a linear regression model succeeds to accurately fit the quadratic line with 50 examples and 50 unseen examples, whereas using the one feature just fits a line (as expected)! The encoding, in combination with linear regression fits a piecewise linear curve.

This function gives the desired result:

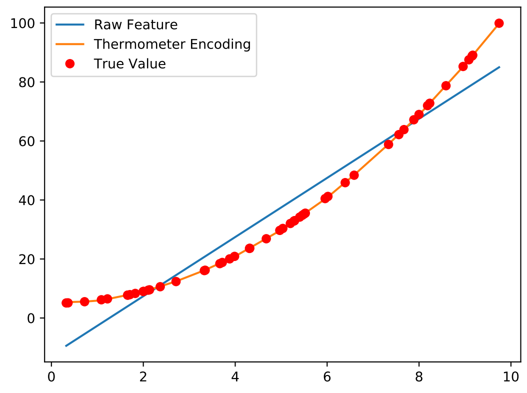

Figure 4.2: Thermometer Encoding versus raw feature

```
>>> thermometer(np.array([[8.5]]), 0, 10)
array([[1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 0.5, 0. ]])
```

4.5 Positional Encoding

Chapter 5

Version Control for Data, Models and Experiments

Chapter 6

Data Augmentation

Chapter 7

Feature Databases

Chapter 8

Algorithms & Data Structures

Chapter 9

Metrics

9.1 Mean Opinion Score

Chapter 10

Optimization

10.1 Stochastic Gradient Descent

10.1.1 Stochastic Gradient Descent with momentum

10.1.2 Adam

10.2 Evolutionary Algorithms

10.3 Bayesian Optimization

Chapter 11

Hyperparameter Optimization

Usually, models contain lots of parameters that are chosen before training a model. Some examples are the number of layers in a neural network and the number of convolutional filters, the features that are used in the model or the learning rate that is used for the optimization algorithm.

We usually do hyperparameter on a metric which we directly want to optimize, such as top-1 accuracy. This is in contrast with gradient-based optimization were we often use a surrogate metric such as cross-entropy loss.

11.1 Neural Architecture Search

Chapter 12

Complexity and Maintainability in Machine Learning

12.1 Feature Selection

12.2 Ablation Studies

Many machine learning models contain a lot of complexity that seem to make them successful.

The question is whether all this complexity is needed, or can we do with less?

One of the ways to answer this is to perform an ablation study: we want to empirically test whether we can *remove* a part of a model or algorithm and see if it doesn't degrade performance.

Doing this can help in a couple of ways:

- Removing input variables or complexities from a model might improve runtime performance.
- Reducing the number of variables helps to remove dependencies from other systems and databases, making integrating and maintaining the model easier.
- Seeing what works and what doesn't gives a clear direction in your project.

Many of today's popular machine learning models went through such a cycle: researchers try constantly to come up with better models by adding novel methods while other researchers are researching how to develop more efficient models that can run for example on mobile devices with less computation and memory available.

12.3 Expressiveness

12.4 Don't Repeat Yourself

12.5 Feature Store

In a larger organisation, multiple data science teams often use the same datasets for learning predictive models. Without any tooling or discussion, they will start to perform feature engineering and . This has a couple of downsides:

- Teams can not quickly experiment and try to learn a model based on features, they have to start to build features first.
- Every team performs the same steps in feature engineering, with large differences in quality of the features

12.6 Data Recency

Chapter 13

Numerical Libraries and Frameworks

To define and train Machine Learning models efficiently, we need libraries and frameworks.

13.1 TensorFlow

13.2 PyTorch

13.3 MxNet

13.4 Scikit-learn

13.5 NumPy

13.6 Keras

13.7 XGBoost

13.8 LightGBM

13.9 FastAI

Chapter 14

Machine Learning on Big Data

14.1 Multi machine Learning

14.2 Distributed Machine Learning

14.3 Data Formats

Chapter 15

Productionizing Models

15.1 Model Formats

After learning a ML model such as a neural network or a random forest, we have to store the *weights* of the model and the *structure* of the model.

- ONNX
- TensorFlow SavedModel
- TensorFlow Lite
- Keras File. A binary file format from Keras that can be used to save / share. It uses the HDF5 format to save the model weights and
- Pickle. This is the built-in object serialization functionality of Python. It is meant to temporarily write an (Python) object to disk, to load it again later within the same environment.

15.2 Serving Models

- TensorFlow Serving
- MXNet Model Server

Chapter 16

Testing

16.1 Numerical Stability: Epsilon

16.2 Randomness in Testing

Chapter 17

Hardware for Machine Learning

17.1 GPU

17.2 TPU

17.3 CPU

17.4 Mobile Devices

17.5 Future Hardware

Chapter 18

Automated Machine Learning

Chapter 19

Reinforcement Learning