

MACHINE

LEARNING

ZOOM CAM?

| INTRO

A, B, C



ML Zoomcamp 2023 – Introduction to Machine Learning – Part 1

👤 Peter ⏰ 9. September 2023 📁 Introduction, ML-Zoomcamp

🏷️ [Feature](#), [Label](#), [Machine Learning](#), [ML Zoomcamp](#), [Model](#), [Prediction](#), [Target](#), [Training](#)

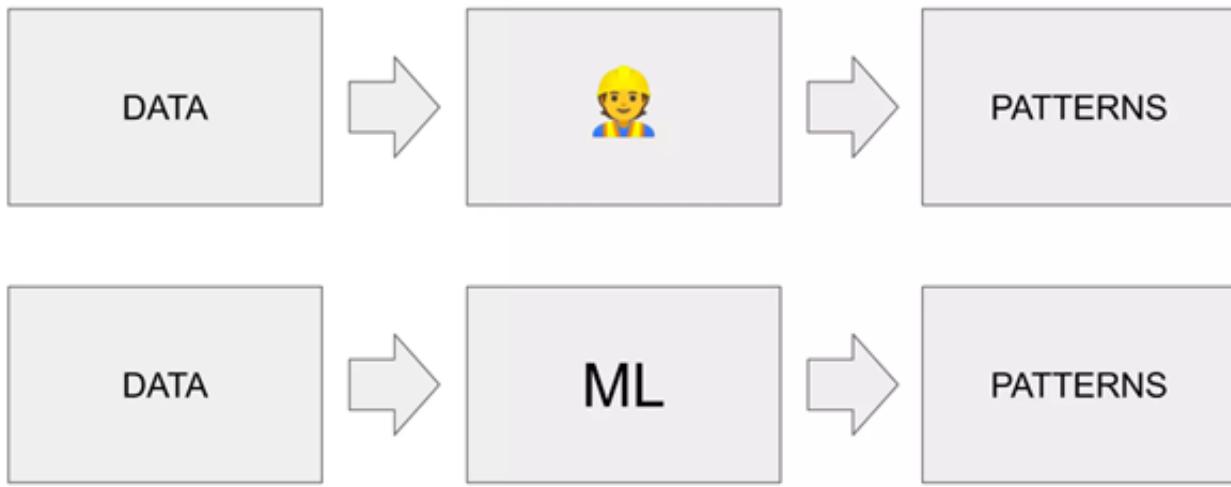


Overview:

1. [Train a model](#)
2. [Use a model](#)

This is a summary of what I've learned from the great ML course (<http://mlzoomcamp.com>) by Alexey Grigorev. All images from this post are from the course material. Images in other posts can also be copies of that material.

The introduction starts with an explanation about what ML really is. You can imagine a task that is normally done by an expert, like getting a good price for selling a car. The expert takes the data about the car and combines all the characteristics to get his opinion about the fair price. What he does is, he extracts patterns from the data. If a human is able to do this, so a model can do the same.



If an expert can, so can a model!

Following is an explanation of what ML is.

Machine Learning (ML) is about using **features** and the **target** information to train a **model** and use this model to predict unknown object targets. In other words it is a process of extracting patterns from data (features+target).

To understand this, you have to distinguish between the terms feature, target and model.

Features means what we know about an object. In this example what we know about the characteristics of a car. A feature is a characteristic of an object in form of a number, string, or other more complex form (e.g. location information, ...)

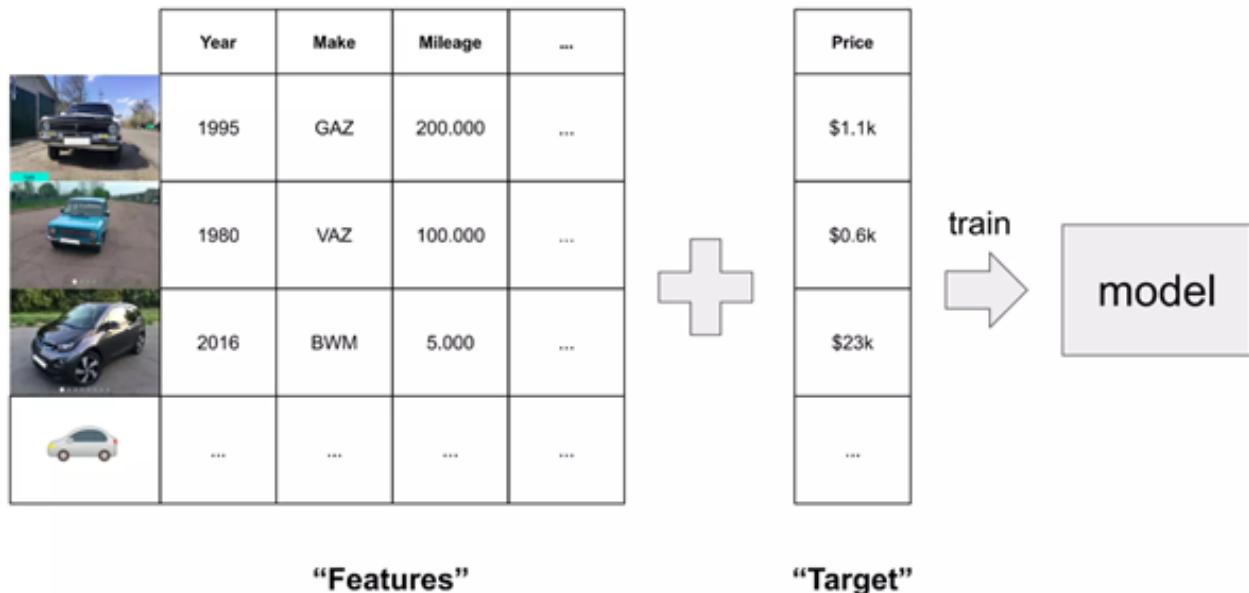
Target is what we want to predict. Other courses / sources also use the term label for this purpose. That means, in training, you talk about a labeled data set because you know the target. In this example, many labeled data sets of cars with prices are used to predict a label for an unknown data set of another car.

A **model** is the output artefact of a **training** that contains all the patterns learned from the trained examples. This output can be used later to make a **prediction** (try to output the target variable) based on features of an unknown object and the model itself.

Train a model

Model training is the process where the machine extracts the patterns from the given training data. In easy words the features are combined with the target – this leads to the model.

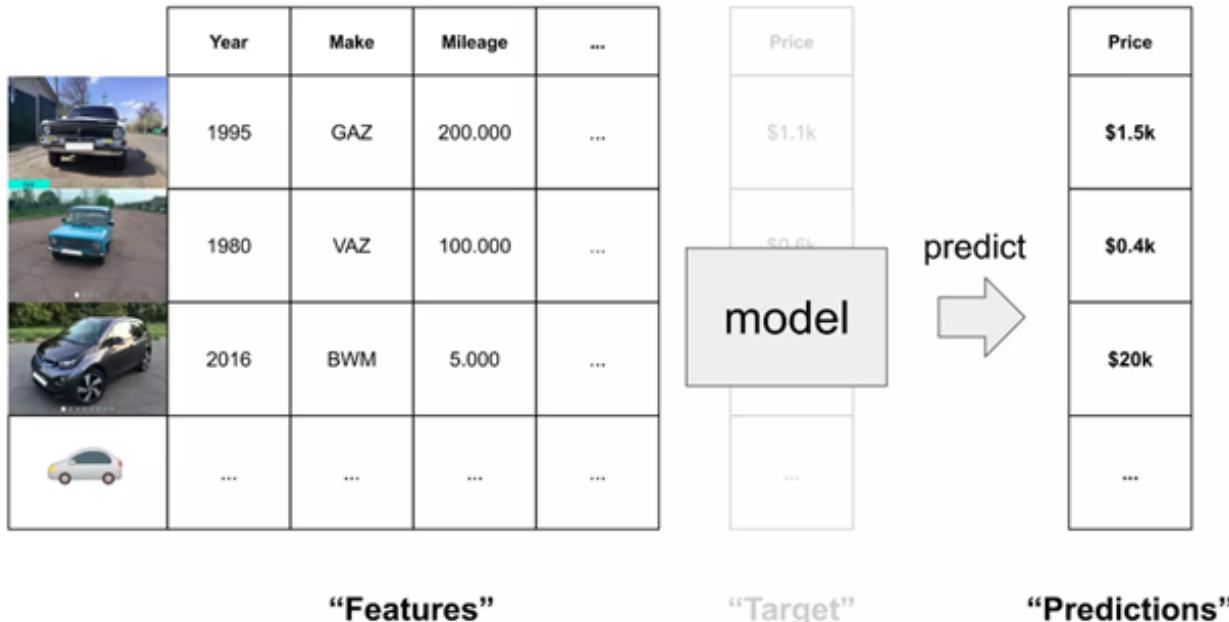
Machine Learning



Use a model

Mere training does not make a model useful. Only the application brings the benefit. Applying the trained model to an unknown data set (without target), you obtain a prediction for the missing information (here: the price).

Using a model



To summarize the difference between model training



and prediction



is that in training process you use features and the target to get the model. And in the prediction process you only use the features and apply the trained model to get a prediction for the target variable.

[Peter](#) [9. September 2023](#) [Introduction, ML-Zoomcamp](#)
[Feature](#), [Label](#), [Machine Learning](#), [ML Zoomcamp](#), [Model](#), [Prediction](#), [Target](#), [Training](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

knowMLedge.com

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 2

👤 Peter ⏰ 10. September 2023 📄 Introduction, ML-Zoomcamp
🏷️ [Machine Learning](#), [ML Zoomcamp](#), [Training](#)



Overview:

1. [Rule-based systems](#)
2. [Machine Learning](#)
 1. [Collect the data](#)
 2. [Define & calculate \(extract\) the features](#)
 3. [Training](#)
 4. [Apply the model](#)

The second part is about the distinction between **Machine Learning** (ML) and **rule-based systems**. The example of a spam filter is used to explain how the implementation would look like without ML.

Rule-based systems

What you need to do is to define some rules to distinguish between ham and spam. So you start defining the rules and for a while everything works fine. However, at some point you have to adjust the rule set and you end up on the hamster wheel because you can't handle the constant reconfiguration of the rules. Also, this system gets harder and harder to maintain.

Machine Learning

The second way to implement this Spam filter is to use ML instead of using hard-coded rules. That means you need to collect the data, define & calculate (extract) the features,

and then train and use the model to classify messages into spam and not spam.

Collect the data

Collecting the data while using the “SPAM” button of your mail system

Define & calculate (extract) the features

Creating the features -> start with the rules you would use in rule-based systems

Features:

- Length of title > 10? true/false
- Length of body > 10? true/false
- Sender “promotions@online.com”? true/false
- Sender “hpYOSKmL@test.com”? true/false
- Sender domain “test.com”? true/false
- Description contains “deposit”? true/false

All of the six features here are binary features, so you can encode each mail as binary code like [1, 1, 0, 0, 1, 1]. Besides this every email has a label¹ / target (spam = 1, no-spam = 0), which is the desired output.

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0
[1, 0, 1, 0, 1, 1]	0

Training

This data is used to train the model. This process is often called as fitting a model.

In training, something happens that is similar to solving a very complex system of equations with many parameters. Here, the features are offset against each other in such a way that the correct classification is obtained at the end. Correct in this example means 1 for spam 0 for no spam. More precisely, we get a probability for the correct label. The trained model contains exactly the information that best solves the equation, namely the weights with which the individual features must be offset to get the correct result.²

Apply the model

If the model is now applied to unknown data sets, the result is a probability. This probability indicates whether this is a spam mail or not. To finally decide how to categorize the mail, a threshold is used (e.g. 0.5). Thus, everything greater than or equal to 0.5 is declared as spam.

Apply	Features (data)	Predictions (output)	Final outcome (decision)
	→ [0, 0, 0, 1, 0, 1]	0.8	SPAM
	[0, 0, 0, 1, 1, 0]	0.6	S
	[1, 0, 1, 0, 1, 1]	0.1	GOOD
	[1, 1, 0, 1, 0, 0]	0.31	G
	[1, 0, 0, 0, 0, 1]	0.7	S
	[1, 1, 0, 0, 1, 1]	0.4	G

≥ 0.5

1. the term “label” is not used in this video ↵
2. this is not described in this video ↵

👤 Peter ⏰ 10. September 2023 📚 Introduction, ML-Zoomcamp
🏷️ Machine Learning, ML Zoomcamp, Training

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 3

👤 Peter ⏰ 11. September 2023 📁 [Introduction](#), [ML-Zoomcamp learning strategies](#), [Machine Learning](#), [ML Zoomcamp](#), [Supervised Machine Learning](#)



Overview:

1. [What is Supervised Machine Learning?](#)
2. [Types of Supervised Machine Learning](#)

As I mentioned before (in [part 2 of this Introduction to ML](#)) there are several approaches to get a software solution for a problem. To give an overview there is the classical approach where everything is hard-coded. In contrast to this there are AI-approaches.

On the one hand there are knowledge-based systems, that are divided into rule-based systems and case-based Reasoning. Rule-based systems were mentioned before in this blog.

On the other hand, there is machine learning as another AI approach. “*Machine Learning [...] provides systems with the ability to learn from experience without being programmed explicitly. Machine Learning is concerned with the development of [...] applications that can access data and learn from it on themselves.*”¹

There are different kind of problems ML is trying to solve.

- **Regression** (predict continuous values, e.g. prices)
- **Classification** (predict labels to distinguish between different classes)
- **Clustering** (predict groups or the data without having any group labels or group characteristics)

Depending on these problem types you can use several different **learning strategies** to solve the problem.

- **Supervised Learning**
- **Unsupervised Learning**
- **Semi-supervised Learning**
- **Reinforcement Learning**
- **Active Learning**

This should give you a brief overview about where Supervised Learning fits in. For more information on all other approaches, please refer to the literature, e.g. [2](#)

What is Supervised Machine Learning?

Supervising the model means that we act as a teacher for the model while we showing different examples with its target value (e.g. price of the car). The machine is able to learn from this examples, while it's extracting the patterns and generalize to new examples.

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 0]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 0]	0
[0, 0, 0, 1, 1, 0]	0
[1, 0, 1, 0, 1, 0]	0

DECODED

Feature matrix X

y

Figure 1.3.1

From figure 1.3.1 you can extract a lot of important information:

- Rows are the observations or objects for which we want to predict something
- Columns are features of the dedicated observation/object
- X is defined as the whole set of features that is called feature matrix (two-dimensional array, array of arrays)
- y is defined as vector with the target variable (one-dimensional array)

From this you can derive the formal definition of supervised machine learning: $\mathbf{g}(\mathbf{X}) \sim \mathbf{y}$ where:

- X : feature matrix
- y : target variable
- g : model that takes X and produces sth. that is approximately close to y

The aim of a training is to get the function g. Mostly this model (g) won't be able to predict always the correct target variable, but we try to be as close as possible.

Features (data)	Predictions (output)	
[0, 0, 0, 1, 0, 1]	0.93	↔ 1
[0, 0, 0, 1, 1, 0]	0.48	0
[1, 0, 1, 0, 1, 1]	0.19	0
[1, 1, 1, 0, 1, 0]	0.32	1
[1, 0, 0, 0, 0, 1]	0.01	0
[1, 1, 0, 0, 1, 1]	0.94	1

$X \rightarrow g(X)$

Figure 1.3.2 – Using the model for prediction

Figure 1.3.2 shows sample predictions for a few objects. You see the output of the function $g(X)$ as a likelihood. Depending on the threshold this value is evaluated to 0 or 1.

Types of Supervised Machine Learning

Regression:

- e.g. price of a car/house/...
- g predicts a number between $-\infty..+\infty$

Classification:

- e.g. identify a picture as a car, identify mail as spam
- g predicts a category
- Input is a picture (or characteristics of an object/observation) and the output is the label/class

Subtypes of classification:

- Multiclass classification problem (distinguishes between several classes (e.g. cat, dog, car))
- Binary classification problem (distinguishes between two classes (e.g. spam vs. not spam))

Ranking:

- Usually used when you want to rank something (e.g. recommender system) -> giving a score to each item in an e-commerce shop and show the top values, because the algorithm calculates that these items have the highest potential to be bought by this customer
- Google's search engine works in a similar way

1. [S. Burns (2019): Python Machine Learning – Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow, 4 ed., Amazon Kindle Publishing] ↵
2. [M. Deru, A. Ndiaye (2019): Deep Learning mit TensorFlow, Keras und TensorFlow.js, 1. Aufl., Bonn, Deutschland: Rheinwerk Computing] ↵

👤 [Peter](#) ⏰ [11. September 2023](#) 📁 [Introduction](#), [ML-Zoomcamp](#)
🏷️ [learning strategies](#), [Machine Learning](#), [ML Zoomcamp](#), [Supervised Machine Learning](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 4

• Peter • 12. September 2023 ■ Introduction, ML-Zoomcamp
🏷️ CRISP-DM, Feature Engineering, ML Zoomcamp



Overview:

1. [CRISP-DM Machine Learning Process](#)
 1. [CRISP-DM is an iterative process with 6 steps](#)
 1. [Business Understanding](#)
 2. [Data Understanding](#)
 3. [Data Preparation \(= Feature Engineering\)](#)
 4. [Modeling](#)
 5. [Evaluation](#)
 6. [Evaluation + Deployment \(Often happens together\)](#)
 7. [Deployment \(=engineering practices\)](#)
 8. [Iterate!](#)
 9. [General note](#)

CRISP-DM Machine Learning Process

This part is about the CRISP-DM Machine Learning Process (Cross-industry standard process for data mining). Methodologies like CRISP-DM help us to organize the ML project in a way that is manageable (what needs to happen in which order).



Figure 1.4.1 – Cross-industry standard process for data mining

Figure 1.4.1 is from [Wikipedia](#). You can find more information on that topic especially in the reference section there is a link to “[CRISP-DM 1.0 Step-by-step data mining guide](#)” if you need more details on that.

CRISP-DM is an iterative process with 6 steps

1. Business Understanding (try to understand the problem)
2. Data Understanding
3. Data Preparation (often called as Feature Engineering)
4. Modeling (train the model)
5. Evaluation
6. Deployment (using the model)

In the following more detailed descriptions of the steps there are some *italic* lines that are not from the course videos but from a book¹.

Business Understanding

- Identify the business problem
- *Detect available data sources*
- *Specify requirements, premises, and conditions*
- *Clarify risks and uncertainties*
- Understand whether the problem is important
- Understand how we can solve it
- **Understand how we measure the success of our project (Cost-Benefit-Analysis)**
- Do we actually need ML here?

Data Understanding

- Analyze available data sources
- *Collect and analyse data*
- Analyze if something is missing and what is missing
- Decide if this data is good/reliable/large enough
- Decide if we need to get more data

Data Preparation (= Feature Engineering)

- Transform the data so it can be put into a ML algorithm
- Usually this means extracting different features
- Clean the data / remove all the noise
- Build the pipelines (that transform raw data into clean data)
- Convert data into tabular form (needed to put in machine learning model)

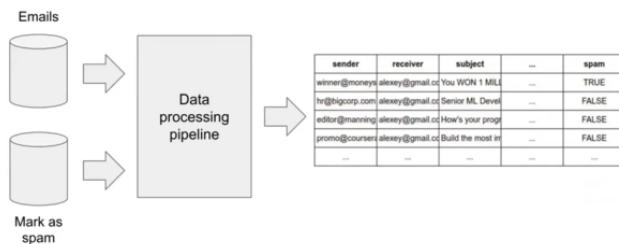


Figure 1.4.2

Figure 1.4.3

Feature Engineering is a key element of every ML project. There is a quote of Andrew Ng, Professor of the Standford University, about Feature Engineering: ***“Coming up with features is difficult, time-consuming, requires expert knowledge. ‘Applied Machine Learning’ is basically feature engineering.”*** I found this quote in a very good german book. This contains a chapter about the CRISP-DM model and Feature Engineering.²

In addition, I found on towardsdatascience.com an old but still interesting article on this subject. There, the importance of feature engineering is also highlighted.

Modeling

- Train the model (the actual ML happens here)
- Try different models
 - Logistic regression, Decision tree, Neural network, others
- *Select model parameters*
- *Try to improve model quality*
- Select the best one

- Sometimes, we may go back to data preparation
 - Add new features
 - Fix data issues
- General aspect that I've learned from practice: **model quality significantly depends on data quality -> keep in mind: Garbage in, Garbage out!**

Evaluation

- Measure how well the model solves the business problem
- Is the model good enough?
 - Have we reached the goal?
- Do our metrics improve?
- Goal: Reduce the amount of spam by 50%
 - Have we reduced it? By how much?
 - (Evaluate on the test group)
- Do a retrospective:
 - Was the goal achievable?
 - Did we solve/measure the right thing?
- After that, we may decide to:
 - Go back and adjust the goal
 - Roll out the model to more users/all users
 - Stop working on the project

Evaluation + Deployment (Often happens together)

- Online evaluation: evaluation of live users
 - It means: deploy the model, evaluate it

Deployment (=engineering practices)

- After online evaluation of some users -> deploy the model to production (all remaining users)
- Roll out the model to all users
- Proper monitoring
- Ensuring the quality and maintainability
- -> when we deploy model it has to work, it has to be reliable
- After that we care about scalability and other things
- *Like in project management this includes creating the final report*

Iterate!

- ML projects require many iterations!
- After deployment we come back to business understanding to check how can we improve the model or decide that it needs to be improved or not.

General note

- Start simple (e.g. with a simple model)

- Learn from feedback
 - Improve (e.g. come back to business understanding and make this model a bit more complex)
1. [R. Schwaiger, J. Steinwendner (2019): Neuronale Netze programmieren mit Python, 1. Aufl., Bonn, Deutschland: Rheinwerk Computing] ↗
 2. [R. Schwaiger, J. Steinwendner (2019): Neuronale Netze programmieren mit Python, 1. Aufl., Bonn, Deutschland: Rheinwerk Computing] ↗

👤 [Peter](#) 📅 [12. September 2023](#) 📄 [Introduction, ML-Zoomcamp](#)
🏷️ [CRISP-DM](#), [Feature Engineering](#), [ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 5

• [Peter](#) • [13. September 2023](#) ■ [Introduction, ML-Zoomcamp](#)
◆ [ML Zoomcamp, Model Selection Process](#)



Overview:

1. [**Model Selection Process**](#)
 1. [Steps to get model performance](#)
 2. [Multiple comparison problem](#)
 3. [Training – validation – test – split](#)
 4. [Summary](#)
 1. [the 6 steps](#)
 2. [Alternative Approach](#)

In my last post (Introduction to Machine Learning – Part 4) I wrote about the CRISP-DM ML Process with its six steps. This post is about the 4th step of that process – the modeling step. This means that the model selection process is also relevant here.

Model Selection Process

Imagine there is a model (g) and a dataset X with target values y . The model performs quite good on that data. But later there is new data and you would like to know how the model performs. What you can do now is to take all the data and use train-validation split. That means for example 80% used for training (old dataset) and 20% used for validation (new dataset).

Steps to get model performance

- Extract feature matrix X_{train} from train dataset

- You also have the target value y_{train} from train dataset
- Using X_{train} and y_{train} to train g
- From validation dataset you also have X_V and y_V
- Applying g to X_V to get predicted values: $g(X_V) = (\hat{y})_V$
- Comparing the predicted $(\hat{y})_V$ values with the actual y_V values to get information about model performance

$(\hat{y})_V$	$(Y-Hat)_V$	Y_V
0.8	1	1
0.7	1	0
0.6	1	1
0.1	0	0
0.9	1	1
0.6	1	0

4 of 6 predicted values are correct ~ 66% accuracy

- Trying different models and selecting the best one based on accuracy, e.g.

g_1	linear regression	66%
g_2	decision tree	60%
g_3	random forest	67%
g_4	neural network	80%

g_4 is the best model

Multiple comparison problem

The last table visualize a problem that could happen, when comparing different models on one validation dataset. The winning model could just get lucky (like a coin-flip) predicting the validation data. When testing the models on a totally different dataset, the winner could be another model.

Training – validation – test – split

To guard cases like this, use three datasets for training, validation, and testing

(60%-20%-20%). Hide the 20% for testing and do the same steps as before. To select the best model based on the training and validation set we use the same model selection process as described before. But now there is an additional step. To ensure that the winning model didn't get lucky on the validation dataset, we also apply this model to the test dataset $g(X_T) = y_T$

		accval	ACC _{Test}
g_1	linear regression	66%	
g_2	decision tree	60%	
g_3	random forest	67%	
g_4	neural network	80%	79%

g_4 performs similar on test set, so we can expect that model wasn't lucky

We can conclude that this model g_4 behaves quite well.

Summary

This process is called the model selection process and is one of the most important thing in Machine Learning.

the 6 steps

1. Split datasets (60%-20%-20%)
2. Train the model
3. Apply the model to validation dataset
Repeat 2 and 3 a few times
4. Select the best model
5. Apply the model to the test dataset
6. Check everything is good (compare accuracy of validation and test datasets)

Alternative Approach

To not waste the validation dataset you can reuse it. That means you train a model on the training dataset, apply the model on validation dataset, and choose the best model as before. But then combine train and validation datasets and train another model based on both datasets. Apply this new model on the test dataset.

The alternative approach mentioned above, where the validation dataset is not wasted, can be a practical solution in some cases. By combining the training and validation datasets, we can create a larger dataset for training a new model. This approach can help improve the performance and generalization of the selected model.

Here are the steps for the alternative approach:

1. Split the original dataset into training, validation, and test sets with a ratio of 60%-20%-20%.
2. Train the initial models using the training dataset.
3. Apply the initial models to the validation dataset and evaluate their performance.
4. Select the best-performing model based on the validation results.
5. Combine the training and validation datasets to create a new combined dataset.
6. Retrain the selected model using the new combined dataset.
7. Apply the newly trained model to the test dataset to assess its performance on unseen data.

By training the model on a larger combined dataset, we can potentially capture more patterns and improve the model's ability to generalize to new data. The final evaluation on the test dataset provides a more reliable measure of the model's performance and gives us confidence in its ability to make accurate predictions.

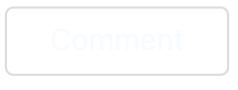
It's important to note that the alternative approach may not always yield better results compared to the original model selection process. The effectiveness of this approach depends on the specific characteristics of the dataset and the performance of the initial models. Experimentation and careful evaluation are key to determine the most suitable approach for your machine learning task.

In summary, the model selection process is a crucial step in machine learning, and it involves thoroughly assessing different models and selecting the one that performs the best on unseen data. The alternative approach of combining the training and validation datasets can be an effective strategy to enhance model performance and generalize better.

 [Peter](#)  [13. September 2023](#)  [Introduction](#), [ML-Zoomcamp](#)
 [ML Zoomcamp](#), [Model Selection Process](#)

Leave a comment

Write a comment...

 Comment

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 6

👤 Peter ⏰ 14. September 2023 📚 Introduction, ML-Zoomcamp
📍 ML Zoomcamp, NumPy



Overview:

1. [Introduction to NumPy part 1/3](#)
 1. [Creating \(one-dimensional\) arrays](#)
 1. [Converting a python list](#)
 2. [Accessing array elements](#)

Introduction to NumPy part 1/3

```
| import numpy as np
```

[NumPy](#) is a powerful Python library that provides support for large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to operate on these arrays. By importing the NumPy module as np, we gain access to all of its functionality and can easily manipulate arrays in our code.

One of the key features of NumPy is its ability to perform vectorized operations, which allows for faster and more efficient computations. Instead of looping over individual elements of an array, we can perform operations on the entire array at once. This not only simplifies our code but also improves its performance.

NumPy also provides numerous functions for creating arrays of different shapes and sizes. For example, we can use the np.array() function to create a new array from a list or a tuple. We can specify the data type of the array elements using the dtype parameter.

In addition to creating arrays, NumPy offers a wide range of functions for performing various mathematical operations. We can easily perform basic arithmetic operations, such as addition, subtraction, multiplication, and division, on NumPy arrays. NumPy also provides functions for calculating statistics, finding maximum and minimum values, and performing linear algebra operations.

Overall, NumPy is an essential library for any data scientist or programmer working with numerical data in Python. Its efficient array operations and wide range of mathematical functions make it a powerful tool for scientific computing and data analysis.

Creating (one-dimensional) arrays

In NumPy, creating one-dimensional arrays is a straightforward process. We can use the `np.array()` function to create an array from a list or a tuple. For example, to create an array of integers, we can do:

```
| import numpy as np  
| my_array = np.array([1, 2, 3, 4, 5])
```

In this case, `my_array` will be a one-dimensional numpy array with the values [1, 2, 3, 4, 5].

We can also create arrays of different data types by specifying the `dtype` parameter. For instance, if we want to create an array of floating-point numbers, we can do:

```
| my_float_array = np.array([1.1, 2.2, 3.3, 4.4, 5.5], dtype=float)
```

In this example, `my_float_array` will be a one-dimensional NumPy array with the values [1.1, 2.2, 3.3, 4.4, 5.5] and the data type set to float.

Another way to create one-dimensional arrays is by using NumPy's built-in functions. For instance, we can use the `np.arange()` function to generate a range of numbers. Here's an example:

```
| my_range_array = np.arange(1, 10, 2)
```

In this case, `my_range_array` will be a one-dimensional NumPy array with the values [1, 3, 5, 7, 9]. The `np.arange()` function takes three arguments: the start value, the stop value (exclusive), and the step value.

Apart from `np.arange()`, NumPy also provides functions like `np.zeros()`, `np.ones()`, and `np.linspace()` for creating arrays with specific values or spacing.

With these methods, you can easily create one-dimensional arrays in NumPy and start performing various mathematical operations on them. NumPy's efficient array operations make it a powerful tool for handling and manipulating large amounts of numerical data.

```
# argument is the size of the array
# result is an array with 5 zeros
np.zeros(5)
# Output:
# array([0., 0., 0., 0., 0.])

np.ones(10)
# Output:
# array([1., 1., 1., 1., 1., 1., 1., 1., 1.])

# first argument is the size of the array
# second argument is the element you want to fill the array with
np.full(10, 2.5)
# Output:
# array([2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5])

# create an array with the range from 0 to <argument>
np.arange(10)
# Output:
# array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.arange(3, 10)
# Output:
# array([3, 4, 5, 6, 7, 8, 9])

# linspace creates an array filled with numbers between first and last
np.linspace(0, 1, 11)
# Output:
# array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.])
```

Converting a python list

To convert a Python list into a NumPy array, we can use the `np.array()` function. This function takes the list as an argument and returns a NumPy array with the same elements.

Here's an example:

```
import numpy as np

my_list = [1, 2, 3, 5, 7, 12]
my_array = np.array(my_list)

# Output:
# array([1, 2, 3, 5, 7, 12])
```

In this case, `my_array` will be a one-dimensional NumPy array with the values `[1, 2, 3, 5, 7, 12]`, which is equivalent to the original Python list.

It's worth noting that NumPy arrays are more memory-efficient compared to regular Python lists, especially when dealing with large datasets. NumPy arrays are stored in a contiguous block of memory, allowing for faster access and manipulation of the data.

Once we have converted a list into a NumPy array, we can perform various mathematical operations on it. NumPy provides convenient functions for element-wise operations, such

as addition, subtraction, multiplication, and division. These operations are performed on each element of the array individually.

Accessing array elements

In NumPy, we can access individual elements of an array by using indexing. NumPy arrays are zero-indexed, which means the first element of the array has an index of 0, the second element has an index of 1, and so on.

To access a specific element of an array, we can use the indexing syntax `array_name[index]`. For example, let's say we have the following array:

```
import numpy as np  
my_array = np.array([1, 2, 3, 4, 5])
```

If we want to access the second element of `my_array`, we can do:

```
second_element = my_array[1]
```

In this case, `second_element` will have the value 2, since the second element of the array has an index of 1.

We can also use negative indexing to access elements from the end of the array. For example, to access the last element, we can use index -1:

```
last_element = my_array[-1]
```

In this case, `last_element` will have the value 5, since -1 refers to the last element of the array.

In addition to accessing individual elements, we can also access a range of elements using slicing. Slicing allows us to extract a subset of elements from an array. The syntax for slicing is `array_name[start_index:end_index]`.

For example, if we want to extract the elements from index 1 to index 3 (inclusive) from `my_array`, we can do:

```
subset_array = my_array[1:4]
```

In this case, `subset_array` will be a new array with the values [2, 3, 4].

If we omit the start or end index in the slicing syntax, it will default to the beginning or end of the array, respectively. For example, if we want to extract all elements from index 2 to the end of the array, we can do:

```
| subset_array = my_array[2:]
```

In this case, `subset_array` will be a new array with the values [3, 4, 5].

Overall, accessing array elements in NumPy is a powerful feature that allows us to extract specific values or subsets of data from arrays. This functionality is very useful when working with large datasets or performing calculations on specific elements of an array.

 [Peter](#)  [14. September 2023](#)  [Introduction](#), [ML-Zoomcamp](#)
 [ML Zoomcamp](#), [NumPy](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 7

👤 Peter ⏰ 14. September 2023 📄 Introduction, ML-Zoomcamp
📍 ML Zoomcamp, NumPy



Overview:

1. [Introduction to NumPy part 2/3](#)
 1. [Multi-dimensional arrays](#)
 1. [Accessing elements in a multi-dimensional array](#)
 2. [Accessing only rows](#)
 3. [Accessing only columns](#)

Introduction to NumPy part 2/3

Multi-dimensional arrays

One of the key advantages of [NumPy](#) is its ability to efficiently work with multi-dimensional arrays. These arrays can have any number of dimensions, from one-dimensional arrays (vectors) to two-dimensional arrays (matrices) to higher-dimensional arrays.

To create a multi-dimensional array in NumPy, you can use the `numpy.array()` function and pass in a nested list or tuple of values. For example, consider the following code snippet:

```
import numpy as np

# Create a 2-dimensional array
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

This will create a 2-dimensional array with two rows and three columns. The `print()` function will display the array as:

```
[[1 2 3]
 [4 5 6]]
```

NumPy provides many functions and methods for manipulating and accessing elements of multi-dimensional arrays. You can perform mathematical operations on arrays, such as addition, subtraction, multiplication, and division, using the standard arithmetic operators or the corresponding NumPy functions. For example:

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# Element-wise addition
result = arr1 + arr2
print(result)
# Output:
# [[ 6  8]
# [10 12]]

# Element-wise multiplication
result = arr1 * arr2
print(result)
# Output:
# [[ 5 12]
# [21 32]]
```

In addition to mathematical operations, you can also apply functions to arrays. NumPy provides a wide range of built-in functions that can be applied element-wise to multi-dimensional arrays. For example, you can calculate the sum, mean, maximum, minimum, or standard deviation of an array using the respective functions:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

# Calculate the sum of all elements
sum_val = np.sum(arr)
print(sum_val)
# Output:
# 21

# Calculate the mean of each column
mean_val = np.mean(arr, axis=0)
print(mean_val)
# Output:
# [2.5 3.5 4.5]
```

These are just a few examples of the operations and functions that can be performed on

multi-dimensional arrays in NumPy. By leveraging the power of NumPy, you can efficiently manipulate and process large amounts of numerical data. Stay tuned for the next section where we will explore more advanced features of NumPy.

```
# creates an array with 5 rows and 2 columns, filled with zero
np.zeros((5, 2))
# Output:
#array([[0.,  0.],
#       [0.,  0.],
#       [0.,  0.],
#       [0.,  0.],
#       [0.,  0.]])  
  
# creates an array from a python list
# here we have a list of lists
np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
# Output:
# array([[1, 2, 3],
#        [4, 5, 6],
#        [7, 8, 9]])
```

Accessing elements in a multi-dimensional array

To access elements in a multi-dimensional array, you can use indexing. In NumPy, indexing works similarly to regular Python lists or arrays. You can use square brackets [] and provide the index or indices for the desired element or elements.

For example, consider the following code snippet:

```
import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6]])  
  
# Accessing a single element
element = arr[0, 2]
print(element)
# Output: 3
```

In this example, we access a single element from the array `arr` using the indices 0 and 2. This will give us the value 3.

Accessing only rows

To access only rows of a multi-dimensional array in NumPy, you can use indexing with a colon : to specify all columns and provide the indices for the desired rows. For example:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

# Accessing a single row
row = arr[1, :]
print(row)
# Output: [4 5 6]
```

In this example, we access a single row from the array `arr` using the index `1` for the row and `:` to indicate all columns. This will give us the values `[4 5 6]` which represent the second row of the array.

You can also access multiple rows by providing a list of indices. For instance:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing multiple rows
rows = arr[[0, 2], :]
print(rows)
# Output:
# [[1 2 3]
# [7 8 9]]
```

In this example, we access the first and third rows of the array `arr` by providing a list of indices `[0, 2]`. The output will be `[[1 2 3] [7 8 9]]`, which represents the selected rows.

By using indexing with the colon `:` and providing the appropriate indices, you can easily access specific rows or subsets of rows from a multi-dimensional array in NumPy.

```
n = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# outputs the first row
n[0]
# Output:
# array([1, 2, 3])

n[0] = [12, 13, 14]
n
# Output:
# array([[12, 13, 14],
#        [4, 5, 6],
#        [7, 8, 9]])
```

Accessing only columns

To access only columns of a multi-dimensional array in NumPy, you can use indexing with a colon : to specify all rows and provide the indices for the desired columns. For example:

```
import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6]])  
  
# Accessing a single column  
column = arr[:, 1]  
print(column)  
# Output: [2 5]
```

In this example, we access a single column from the array `arr` using the index `1` for the column and `:` to indicate all rows. This will give us the values `[2 5]` which represent the second column of the array.

You can also access multiple columns by providing a list of indices. For instance:

```
import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
  
# Accessing multiple columns  
columns = arr[:, [0, 2]]  
print(columns)  
# Output:  
# [[1 3]  
# [4 6]  
# [7 9]]
```

In this example, we access the first and third columns of the array `arr` by providing a list of indices `[0, 2]`. The output will be `[[1 3] [4 6] [7 9]]`, which represents the selected columns.

By using indexing with the colon `:` and providing the appropriate indices, you can easily access specific columns or subsets of columns from a multi-dimensional array in NumPy.

```

n = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# outputs the first column
n[:, 0]
# Output:
# array([1, 4, 7])

n[:, 0] = [12, 13, 14]
n
# Output:
# array([[12, 2, 3],
#        [13, 5, 6],
#        [14, 8, 9]])

```

In this example, we access and modify the first column of the array `n`. The output will be `array([[12, 2, 3], [13, 5, 6], [14, 8, 9]])`, where the first column has been replaced with the values `[12, 13, 14]`.

Accessing specific columns or modifying columns in a multi-dimensional array can be done easily using the indexing capabilities provided by NumPy.

```

n = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# outputs the first column
n[:, 0]
# Output:
# array([1, 4, 7])

# outputs all the columns
n[:, :]
# Output:
# array([[1, 2, 3],
#        [4, 5, 6],
#        [7, 8, 9]])

n[:, 2] = [0, 1, 2]
n
# Output:
# array([[1, 2, 0],
#        [4, 5, 1],
#        [7, 8, 2]])

```

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 8

👤 Peter ⏰ 14. September 2023 📚 Introduction, ML-Zoomcamp
📍 ML Zoomcamp, NumPy



Overview:

1. [Introduction to NumPy part 3/3](#)
 1. [Randomly generated arrays](#)
 2. [Element-wise operations](#)
 3. [Comparison operations](#)
 4. [Summarizing operations](#)

Introduction to NumPy part 3/3

Randomly generated arrays

In addition to creating and manipulating multi-dimensional arrays, [NumPy](#) also provides the capability to generate arrays filled with random values. This can be useful in various scientific and mathematical applications, as well as for testing and simulation purposes.

To create a randomly generated array, you can use the `numpy.random` module, which provides a range of functions for generating random values. Here are a few examples:

```
import numpy as np

# Generate a 1-dimensional array of 5 random integers between
random_integers = np.random.randint(10, size=5)
print(random_integers)
# Output: [2 5 7 1 8]

# Generate a 2-dimensional array of shape (3, 4)
# with random floats between 0 and 1
random_floats = np.random.random((3, 4))
print(random_floats)
# Output:
# [[0.0863851  0.83574087  0.79192621  0.85248822]
#  [0.14040051  0.5714931   0.7586195   0.48544792]
#  [0.4304003   0.76688989  0.68447497  0.54361942]]

# Generate a 3-dimensional array of shape (2, 3, 2)
# with random values from a standard normal distribution
random_normal = np.random.normal(size=(2, 3, 2))
print(random_normal)
# Output:
# [[[ -0.27013393  0.54416022]
#   [ 0.02537238 -0.78380969]
#   [ -1.25909646 -1.04630766]]
#  [
#   [[ 0.34262622 -0.66770036]
#    [-0.35426751  0.00569635]
#    [ -0.05665257  0.02068191]]]
```

By using the appropriate functions from the `numpy.random` module, you can easily generate arrays with random values according to your specific requirements.

```

# generates a 2-dimensional array of size 5 rows and 2 columns
# with random numbers between 0 and 1
# rand samples from standard uniform distribution
np.random.rand(5, 2)
# Output:
# array([[0.83575882, 0.03277884],
#        [0.78785763, 0.34340225],
#        [0.79212789, 0.75564912],
#        [0.78937584, 0.4326158 ],
#        [0.90909093, 0.82098053]])

# when you set the random seed it's possible to reproduce the
# same "random" values
np.random.seed(2)
np.random.rand(5, 2)
# Output:
# array([[0.4359949 , 0.02592623],
#        [0.54966248, 0.43532239],
#        [0.4203678 , 0.33033482],
#        [0.20464863, 0.61927097],
#        [0.29965467, 0.26682728]])

# randn samples from standard normal distribution
np.random.seed(2)
np.random.randn(5, 2)
# Output:
# array([[-0.41675785, -0.05626683],
#        [-2.1361961 , 1.64027081],
#        [-1.79343559, -0.84174737],
#        [ 0.50288142, -1.24528809],
#        [-1.05795222, -0.90900761]])

# creates random numbers between 0 and 100
np.random.seed(2)
100 * np.random.rand(5, 2)
# Output:
# array([[43.59949021, 2.59262318],
#        [54.96624779, 43.53223926],
#        [42.03678021, 33.0334821 ],
#        [20.4648634 , 61.92709664],
#        [29.96546737, 26.68272751]])

# creates an array of random integer numbers
np.random.seed(2)
np.random.randint(low=0, high=100, size=(5, 2))
# Output:
# array([[40, 15],
#        [72, 22],
#        [43, 82],
#        [75, 7],
#        [34, 49]])

```

Element-wise operations

NumPy supports element-wise operations on arrays, which means you can perform mathematical operations or apply functions to each element of an array individually. This makes it easy to perform calculations on entire arrays without the need for explicit loops.

For example, you can add or subtract arrays element-wise using the `+` and `-` operators, respectively. Here's an example:

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
result = arr1 + arr2
print(result)
# Output: [5 7 9]

# Element-wise subtraction
result = arr1 - arr2
print(result)
# Output: [-3 -3 -3]
```

Similarly, you can perform element-wise multiplication or division using the `*` and `/` operators, respectively. Here's an example:

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise multiplication
result = arr1 * arr2
print(result)
# Output: [4 10 18]

# Element-wise division
result = arr1 / arr2
print(result)
# Output: [0.25 0.4 0.5]
```

In addition to basic arithmetic operations, you can also apply various mathematical functions to arrays element-wise. NumPy provides a comprehensive collection of built-in functions such as `np.sin()`, `np.cos()`, `np.exp()`, and `np.log()`, among others. Here's an example:

```
import numpy as np

arr = np.array([0, np.pi/2, np.pi])

# Calculate the sine of each element
result = np.sin(arr)
print(result)
# Output: [0.          1.          1.2246468e-16]

# Calculate the exponential of each element
result = np.exp(arr)
print(result)
# Output: [ 1.          4.81047738 23.14069263]
```

By leveraging element-wise operations and built-in functions, you can perform complex calculations on arrays efficiently and easily in NumPy.

```
a = np.arange(5)
a
# Output:
# array([0, 1, 2, 3, 4])

# adds 1 to every element in the array
# be careful, you cannot do this with a normal python list
a + 1
# Output:
# array([1, 2, 3, 4, 5])

a * 2
# Output:
# array([0, 2, 4, 6, 8])

a * 100
# Output:
# array([ 0, 100, 200, 300, 400])

a / 100
# Output:
# array([0. , 0.01, 0.02, 0.03, 0.04])

(10 + (a * 2)) ** 2
# Output:
# array([100, 144, 196, 256, 324])

b = (10 + (a * 2)) ** 2 / 100
b
# Output:
# array([1. , 1.44, 1.96, 2.56, 3.24])

# adds element-wise both arrays
a + b
# Output:
# array([1. , 2.44, 3.96, 5.56, 7.24])
```

Comparison operations

Comparison operations in NumPy allow you to compare elements of arrays and obtain boolean results. This is particularly useful for tasks such as filtering or conditional assignment.

For example, you can use the comparison operator `>` to determine which elements of an array are greater than a specified value. Here's an example:

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
# Compare elements with 3  
result = arr > 3  
print(result)  
# Output: [False False False True True]
```

In this example, the comparison `arr > 3` compares each element of the array `arr` with the value 3. The result is a boolean array where `True` represents elements that are greater than 3 and `False` represents elements that are lesser or equal to 3.

Comparison operations can also be used with multiple arrays of the same shape. The result is an element-wise comparison between corresponding elements of the arrays. Here's an example:

```
import numpy as np  
  
arr1 = np.array([1, 2, 3])  
arr2 = np.array([3, 2, 1])  
  
# Compare elements of arr1 and arr2  
result = arr1 == arr2  
print(result)  
# Output: [False True False]
```

In this example, the comparison `arr1 == arr2` compares each element of `arr1` with the corresponding element of `arr2`. The resulting boolean array has `True` values for elements that are equal in both arrays and `False` values otherwise.

Comparison operations can also be combined with logical operators such as `&` (logical AND) and `|` (logical OR) to perform more complex comparisons. Here's an example:

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
# Compare elements that are greater than 2 and less than 5  
result = (arr > 2) & (arr < 5)  
print(result)  
# Output: [False False True True False]
```

In this example, the comparison `(arr > 2) & (arr < 5)` combines two comparisons using the logical AND operator. The resulting boolean array contains `True` values only for elements that are both greater than 2 and less than 5.

By leveraging comparison operations in NumPy, you can easily perform element-wise comparisons and obtain boolean arrays representing the results. These boolean arrays can then be used for various purposes, such as filtering or conditional assignment.

```

a = np.arange(5)
b = (10 + (a * 2)) ** 2 / 100
# compare numbers element-wise
a >= 2
# Output:
# array([False, False, True, True, True])

a > b
# Output:
# array([False, False, True, True, True])

# checks which elements of a are greater than b
# a > b -> returns an boolean array
# a[a > b] returns the elements where the boolean array is true
# that are the elements 2, 3, and 4
a[a > b]
# Output:
# array([2, 3, 4])

a[2], a[3], a[4]
# Output:
(2, 3, 4)

```

Summarizing operations

```

# there are some operations that instead of returning a new array
# it returns a single number
# e.g. min() returns the smallest number
a.min()
# Output: 0

a.max()
# Output: 4

a.sum()
# Output: 10

a.mean()
# Output: 2

# standard deviation
a.std()
# Output: 1.4142135623730951

n = np.array([
    [12, 13, 0],
    [4, 5, 1],
    [7, 8, 2]
])

# this also works for 2-dimensional arrays
n.sum()
# Output: 52

n.min()
# Output: 0

```

For more information about NumPy functions check this links:

<https://www.datacamp.com/cheat-sheet/numpy-cheat-sheet-data-analysis-in-python>

<https://mlbookcamp.com/article/numpy>

<https://gist.github.com/ziritrion/9b8oe47956adcof20ecce209d494cdoa#numpy>

👤 Peter ⏰ 14. September 2023 📄 Introduction, ML-Zoomcamp
🏷️ ML Zoomcamp, NumPy

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 9

👤 Peter ⏰ 15. September 2023 📄 Introduction, ML-Zoomcamp
🏷️ [Linear algebra](#), [ML Zoomcamp](#)



Overview:

1. [Linear algebra refresher – Part 1/3](#)
 1. [Simple vector operations](#)
 1. [Scalar vector product](#)
 1. [Implementation in Python](#)
 2. [Vector vector addition](#)
 1. [Implementation in Python](#)

This part is a linear algebra refresher. There are different operations you need to understand for ML. Actually, you don't really need to know that if you know what you're doing in the code. However, a basic understanding can't do any harm at this point.

Because this article has become quite long, I decided to split it into three articles. The first part of the refresher covers simple vector operations. The second part is about vector vector multiplication, matrix vector multiplication and matrix matrix multiplication. The third part is about special matrices.

Linear algebra refresher – Part 1/3

The first part covers simple vector operations.

Simple vector operations

Scalar vector product

The scalar vector product, also known as scalar multiplication, is a fundamental operation in linear algebra. It involves multiplying a scalar (a single number) by each component of a vector.

To perform a scalar vector product, you simply multiply the scalar by each element of the vector separately. For example, if we have a scalar c and a vector v , the scalar vector product is denoted as $c * v$ and is calculated as:

$$c * v = (c * v_1, c * v_2, c * v_3, \dots, c * v_n)$$

where v_1, v_2, \dots, v_n are the individual components of the vector v .

$$2 \cdot \begin{bmatrix} 2 \\ 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \\ 10 \\ 12 \end{bmatrix}$$

This operation is useful in various mathematical and computational applications. It can be used to scale vectors, change their direction, or perform transformations in vector spaces. Additionally, scalar vector products play a significant role in linear combinations, linear transformations, and eigenvector calculations.

Understanding the scalar vector product is essential for grasping more advanced concepts in linear algebra and machine learning. It's a fundamental building block that forms the basis for more complex mathematical operations and algorithms.

Implementation in Python

```
import numpy as np  
  
u = np.array([2, 4, 5, 6])  
2 * u  
# Output: array([4, 8, 10, 12])
```

Vector vector addition

Vector vector addition is another important operation in linear algebra. It involves adding two vectors together to obtain a new vector.

To perform vector vector addition, you simply add corresponding components of the vectors. For example, if we have two vectors u and v , the vector vector addition is denoted as $u + v$ and is calculated as:

$$u + v = (u_1 + v_1, u_2 + v_2, u_3 + v_3, \dots, u_n + v_n)$$

where u_1, u_2, \dots, u_n and v_1, v_2, \dots, v_n are the individual components of the vectors u and v respectively.

$$\begin{bmatrix} 2 \\ 4 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 2+1 \\ 4+0 \\ 5+0 \\ 6+2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 8 \end{bmatrix}$$

Vector vector addition is a fundamental operation in various mathematical and computational applications. It can be used to combine the effects of multiple vectors, calculate displacement or velocity, and solve systems of linear equations.

Understanding vector vector addition is crucial for working with vector spaces, linear transformations, and solving problems in machine learning. It provides a basis for more complex operations and algorithms that involve manipulating vectors.

By comprehending both the scalar vector product and vector vector addition, you will have a solid foundation in linear algebra, enabling you to tackle more advanced topics in machine learning with confidence. These operations serve as the building blocks for many mathematical concepts and computational techniques used in the field.

Remember, even though a deep understanding of linear algebra may not be required for implementing machine learning algorithms directly, having a basic understanding can greatly enhance your ability to interpret and optimize these algorithms for better performance and results.

Implementation in Python

```
import numpy as np
u = np.array([2, 4, 5, 6])
v = np.array([1, 0, 0, 2])
u + v
# Output: array([3, 4, 5, 8])
```

👤 Peter ⏰ 15. September 2023 📚 Introduction, ML-Zoomcamp
🏷️ Linear algebra, ML Zoomcamp

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 10

• Peter ⏰ 15. September 2023 ■ [Introduction](#), [ML-Zoomcamp](#)
🏷️ [Dot product](#), [Linear algebra](#), [ML Zoomcamp](#), [Scalar product](#)



Overview:

1. [Linear algebra refresher – Part 2/3](#)
 1. [Vector vector multiplication \(dot product\)](#)
 1. [Implementation in Python](#)
 2. [Matrix vector multiplication](#)
 1. [Implementation in Python](#)
 3. [Matrix matrix multiplication](#)
 1. [Implementation in Python](#)

Linear algebra refresher – Part 2/3

This is the second part of the refresher and covers more sophisticated operations and its implementation in Python.

Vector vector multiplication (dot product)

The dot product, also known as the **scalar product**, is a key operation in linear algebra. It involves multiplying the corresponding components of two vectors and summing up the results.

To calculate the **dot product** of two vectors u and v , we multiply each component of u with the corresponding component of v and then add up the products. Mathematically, the dot product is denoted as $u \cdot v$ and is calculated as:

$$u \cdot v = u_1v_1 + u_2v_2 + u_3v_3 + \dots + u_nv_n$$

Here, u_1, u_2, \dots, u_n and v_1, v_2, \dots, v_n are the individual components of the vectors u and v , respectively.

$$\begin{bmatrix} 2 \\ 4 \\ 5 \\ 6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} = (2*1) + (4*0) + (5*0) + (6*2) = 2 + 12 = 14$$

An important use case is the dot product between a transposed vector and a vector, e.g. $v^T u$.

$$v^T = [2 \ 4 \ 5 \ 6] \quad u = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} \quad v^T u = \sum_{i=1}^n u_i v_i$$

The dot product yields a scalar value, hence the name “scalar product.” It provides information about the similarity or alignment of two vectors. For example, if the dot product of two vectors is zero, it indicates that the vectors are perpendicular or orthogonal to each other. Conversely, a positive dot product suggests that the vectors have a similar direction, while a negative dot product indicates opposite directions.

The dot product has various applications in mathematics and machine learning. It allows us to calculate the angle between vectors, determine vector projections, and perform vector comparisons. In machine learning, the dot product is particularly useful in computing similarity measures, such as cosine similarity, which is commonly used in recommendation systems and natural language processing tasks.

By understanding the dot product, you gain insights into the geometric relationship between vectors and acquire a powerful tool for analyzing and manipulating vector data. This knowledge forms the basis for advanced techniques in linear algebra and machine learning.

Expanding your understanding of vector operations beyond basic scalar vector products, vector vector addition, and vector vector multiplication sets a solid foundation for tackling more complex mathematical concepts and algorithms in the realm of machine learning. These operations serve as essential tools for data transformation, feature engineering, and model optimization.

Remember that while these operations are fundamental to the field, their implementation is often abstracted away by high-level libraries and frameworks. However, having a conceptual grasp of these operations can greatly enhance your ability to optimize and interpret machine learning models, as well as comprehend the underlying mathematical principles.

Implementation in Python

```
import numpy as np

def vector_vector_multiplication(u, v):
    assert u.shape[0] == v.shape[0]

    n = u.shape[0]
    result = 0.0

    for i in range(n):
        result = result + u[i] * v[i]

    return result

u = np.array([2, 4, 5, 6])
v = np.array([1, 0, 0, 2])
vector_vector_multiplication(u, v)
# Output: 14.0

# dot product is already implemented in numpy
u.dot(v)
# Output: 14.0
```

Matrix vector multiplication

Matrix vector multiplication is a fundamental operation in linear algebra that involves multiplying a matrix with a vector to produce a new vector.

To perform matrix vector multiplication, we multiply each row of the matrix by the corresponding element of the vector and sum up the results. This process results in a new vector that is a linear combination of the rows of the matrix.

Mathematically, if we have a matrix U and a vector v , the matrix vector multiplication is denoted as Uv and is calculated as:

$$Uv = (U_{11}v_1 + U_{12}v_2 + \dots + U_{1n}v_n, U_{21}v_1 + U_{22}v_2 + \dots + U_{2n}v_n)$$

Here, $U_{11}, U_{12}, \dots, U_{mn}$ are the individual elements of the matrix U , and v_1, v_2, \dots, v_n are the components of the vector v .

Matrix vector multiplication is a crucial operation in various computational and mathematical applications. It can be used to perform transformations, solve systems of linear equations, and represent linear mappings between vector spaces. In machine learning, matrix vector multiplication is fundamental in performing linear regression, applying weight matrices to input data, and transforming features in neural networks.

Understanding matrix vector multiplication is essential for working with matrices and applying linear transformations in machine learning algorithms. It provides a powerful tool for manipulating and analyzing data in high-dimensional spaces.

Beyond matrix vector multiplication, other important matrix operations in linear algebra include matrix addition, matrix multiplication, and matrix inversion. These operations enable more complex mathematical operations and algorithms, such as solving systems of linear equations, calculating determinants, and finding eigenvalues and eigenvectors.

By expanding your knowledge of matrix operations, you will have a solid foundation for tackling advanced concepts and techniques in machine learning. These operations form the backbone of many algorithms and models used in data analysis and pattern recognition.

Remember that while you may not need to manually perform matrix vector multiplication in your machine learning code, understanding its underlying principles can enable you to optimize your models, troubleshoot potential issues, and make informed decisions about your data and transformations.

Implementation in Python

```

import numpy as np

def matrix_vector_multiplication(U, v):
    assert U.shape[1] == v.shape[0]

    num_rows = U.shape[0]
    result = np.zeros(num_rows)

    for i in range(num_rows):
        result[i] = vector_vector_multiplication(U[i], v)

    return result

U = np.array([
    [2, 4, 5, 6],
    [1, 2, 1, 2],
    [3, 1, 2, 1]
])

v = np.array([1, 0, 0, 2])
matrix_vector_multiplication(U, v)
# Output: array([14.,  5.,  5.])

# dot product between vector and matrix is already implemented
U.dot(v)
# Output: array([14,  5,  5])

```

Matrix matrix multiplication

Matrix matrix multiplication is a fundamental operation in linear algebra that involves multiplying two matrices to produce a new matrix. This operation enables the transformation and manipulation of data in multidimensional spaces.

To perform matrix matrix multiplication, we multiply each row of the first matrix by each column of the second matrix, and then sum up the results. The resulting matrix has dimensions equal to the number of rows in the first matrix and the number of columns in the second matrix.

Mathematically, if we have two matrices, A and B, the matrix matrix multiplication is denoted as AB and is calculated as:

$$\begin{aligned}
 AB &= (A_{11}B_{11} + A_{12}B_{21} + \dots + A_{1n}B_{n1}, A_{11}B_{12} + A_{12}B_{22} + \dots + A_{1n}B_{n2}, \dots, A_{m1}B_{11} + \\
 &\quad A_{m2}B_{21} + \dots + A_{mn}B_{n1}; \\
 &\quad A_{11}B_{12} + A_{12}B_{22} + \dots + A_{1n}B_{n2}, A_{11}B_{12} + A_{12}B_{22} + \dots + A_{1n}B_{n2}, \dots, A_{m1}B_{12} \rightarrow A_{m2}B_{22} + \dots \\
 &\quad + A_{mn}B_{n2}; \\
 &\quad \dots \\
 &\quad A_{11}B_{1m} + A_{12}B_{2m} + \dots + A_{1n}B_{nm}, A_{11}B_{1m} + A_{12}B_{2m} + \dots + A_{1n}B_{nm}, \dots, A_{m1}B_{1m} + A_{m2}B_{2m} + \\
 &\quad \dots + A_{mn}B_{nm})
 \end{aligned}$$

Here, A_{11} , A_{12} , ..., A_{mn} and B_{11} , B_{21} , ..., B_{nm} are the individual elements of matrices A and B, respectively.

```

import numpy as np

def matrix_matrix_multiplication(U, V):
    assert U.shape[1] == V.shape[0]

    num_rows = U.shape[0]
    num_columns = V.shape[1]

    result = np.zeros((num_rows, num_columns))

    for i in range(num_columns):
        vi = V[:, i]
        Uvi = matrix_vector_multiplication(U, vi)
        result[:, i] = Uvi

    return result

U = np.array([
    [2, 4, 5, 6],
    [1, 2, 1, 2],
    [3, 1, 2, 1]
])

V = np.array([
    [1, 1, 2],
    [0, 0.5, 1],
    [0, 3, 1],
    [2, 1, 0]
])

matrix_matrix_multiplication(U, V)
# Output:
#array([[14. , 25. , 13. ],
#       [ 5. ,  7. ,  5. ],
#       [ 5. , 10.5,  9. ]])

# dot product between matrix and matrix is already implemented
U.dot(V)
# Output:
#array([[14. , 25. , 13. ],
#       [ 5. ,  7. ,  5. ],
#       [ 5. , 10.5,  9. ]])

```

In this example, U and V are two matrices that we want to multiply. The `np.dot()` function performs the matrix matrix multiplication, and the result is a new matrix.

Matrix matrix multiplication is a fundamental operation that opens the door to a wide range of mathematical techniques and algorithms. By grasping its principles and implementing it in your code, you can harness the power of linear algebra in your machine learning projects.

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 11

👤 Peter ⏰ 15. September 2023 📄 Introduction, ML-Zoomcamp
🏷️ [Linear algebra](#), [ML Zoomcamp](#)



Overview:

1. [Linear algebra refresher – Part 3/3](#)
 1. [Special matrix types](#)
 1. [Identity matrix](#)
 1. [Implementation in Python](#)
 2. [Inverse matrix](#)
 1. [Implementation in Python](#)
 2. [Eigenvalues and Eigenvectors](#)
 3. [Determinants](#)

Linear algebra refresher – Part 3/3

This is the third and last part of the refresher and covers special matrix types and its implementation in Python.

Special matrix types

Identity matrix

An identity matrix is a special type of square matrix in linear algebra. It is denoted as I and has ones along its main diagonal (from the top-left to the bottom-right) and zeros in all other positions.

The identity matrix is typically represented as follows:

```

I = [[1, 0, 0, ..., 0],
[0, 1, 0, ..., 0],
[0, 0, 1, ..., 0],
...
[0, 0, 0, ..., 1]]

```

Here, the matrix I is of size $n \times n$, where n represents the number of rows (or columns) in the matrix.

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Identity matrix of size 4×4

The identity matrix is unique because it behaves like the number one in matrix operations. When the identity matrix is multiplied with another matrix U, the result is U itself.

Mathematically, this can be expressed as:

$$I * U = U$$

Similarly, when matrix U is multiplied by the identity matrix, the result is also U:

$$U * I = U$$

The identity matrix has various applications in linear algebra and matrix operations. It serves as the neutral element for matrix multiplication, similar to how the number one acts as the neutral element for multiplication of real numbers. The identity matrix also plays a crucial role in defining matrix inverses, solving systems of linear equations, and performing transformations.

In machine learning, the identity matrix is often used as the initial value for weight matrices in neural networks. This ensures that the initial weights do not affect the input data during the first round of computations. The use of identity matrices in neural networks helps prevent over-fitting and facilitates better convergence during the training process.

Understanding the identity matrix and its properties is important for working with matrices, transformations, and solving systems of linear equations. It provides a solid foundation for more advanced topics in linear algebra and machine learning.

Implementation in Python

```

import numpy as np

np.eye(3)
# Output:
# array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])]

V = np.array([
    [1, 1, 2],
    [0, 0.5, 1],
    [0, 3, 1],
    [2, 1, 0]
])

V.dot(I)
# Output:
# array([[1. , 1. , 2. ],
#        [0. , 0.5, 1. ],
#        [0. , 3. , 1. ],
#        [2. , 1. , 0. ]])

V.dot(I) == V
# Output:
# array([[ True,  True,  True],
#        [ True,  True,  True],
#        [ True,  True,  True],
#        [ True,  True,  True]])

```

Inverse matrix

The inverse of a matrix is a fundamental concept in linear algebra. It is denoted as U^{-1} and represents the matrix that, when multiplied with the original matrix U , yields the identity matrix I .

To calculate the inverse of a matrix U , we need to ensure that U is a square matrix and that it is invertible (i.e., its determinant is non-zero). Here is the general formula for finding the inverse:

$$U^{-1} = (1/|U|) * \text{adj}(U)$$

In this formula, $|U|$ represents the determinant of matrix U , and $\text{adj}(U)$ denotes the adjugate of matrix U . The adjugate of a matrix is the transpose of its cofactor matrix.

Finding the inverse of a matrix is a crucial operation in many areas of mathematics and engineering. It allows us to solve systems of linear equations, perform geometric transformations, and analyze the properties and behavior of matrices. In machine learning, the inverse of a matrix is often used in optimization algorithms and data transformations.

It's important to note that not all matrices are invertible. If a matrix is not invertible (i.e., it has a determinant of zero), it is called a singular matrix. Singular matrices have zero as an eigenvalue and cannot be inverted.

Understanding the inverse of a matrix and how to calculate it is essential for working with linear systems, transformations, and solving problems in machine learning. It provides a powerful tool for data manipulation and model optimization.

Implementation in Python

```
import numpy as np

# only squared matrices has an inverse matrix
V = np.array([
    [1, 1, 2],
    [0, 0.5, 1],
    [0, 3, 1]
])

# there is a function in numpy available that returns the inv
V_inv = np.linalg.inv(V)
V_inv
# Output:
# array([[ 1. , -2. ,  0. ],
#        [ 0. , -0.4,  0.4],
#        [ 0. ,  1.2, -0.2]])

# just to check that V_inv.dot(V) == I
Vs_inv.dot(Vs)
# Output:
# array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])
```

Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are important concepts in linear algebra, particularly when analyzing the properties and behavior of matrices.

An eigenvector of a matrix U is a non-zero vector that, when multiplied by U , yields a scalar multiple of itself. In other words, the eigenvector remains in the same direction, although its magnitude may change.

Mathematically, for a square matrix U and its corresponding eigenvector v , the equation for eigenvalues and eigenvectors is given by:

$$U * v = \lambda * v$$

Here, λ represents the eigenvalue of matrix U associated with the eigenvector v .

Eigenvalues provide information about how stretching or compression occurs along eigenvectors when a matrix is applied. Eigenvectors, on the other hand, represent the directions that remain unchanged when a matrix transformation is applied.

Eigenvalues and eigenvectors have various applications in linear algebra and machine learning. They are used to analyze the behavior of matrices, perform matrix

decompositions (such as eigendecomposition), and understand the dynamics of linear systems. In machine learning, eigenvalues and eigenvectors are particularly relevant in dimensionality reduction techniques (such as Principal Component Analysis) and spectral clustering algorithms.

Understanding eigenvalues and eigenvectors is crucial for analyzing the properties of matrices, performing transformations, and applying advanced techniques in linear algebra and machine learning. They provide insights into the behavior and structure of data, leading to more effective models and algorithms.

Determinants

The determinant is a valuable quantity that carries important information about the properties and behavior of matrices. It is denoted as $|A|$ and is calculated for square matrices.

The determinant of a matrix A provides information about the scaling factor and orientation of a transformation represented by the matrix. It determines whether the matrix is invertible or singular, and it affects the behavior of the matrix in equations and calculations.

To calculate the determinant of a square matrix, we use various methods depending on the matrix's size. For small matrices (e.g., 2×2 or 3×3), we can use simple formulas. However, for larger matrices, we often employ more efficient algorithms, such as LU decomposition or Gaussian elimination.

The determinant is useful in various applications, such as solving systems of linear equations, determining matrix invertibility, finding eigenvalues, calculating volume or area scale factors, and analyzing the properties of transformations and matrices.

Understanding the determinant is essential for working with matrices, linear systems, and transformations. It provides insights into the behavior and properties of data, enabling more effective analysis and modeling in various fields, including machine learning.

By expanding your knowledge of linear algebra beyond the basic vector and matrix operations, you will have a solid foundation for understanding and applying more advanced mathematical concepts and algorithms. These concepts serve as essential tools for solving complex problems in machine learning and data analysis.

 [Peter](#)  [15. September 2023](#)  [Introduction, ML-Zoomcamp](#)
 [Linear algebra, ML Zoomcamp](#)

[Leave a comment](#)

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 12

👤 Peter ⏰ 16. September 2023 📄 Introduction, ML-Zoomcamp
🏷️ [DataFrame](#), [ML Zoomcamp](#), [Pandas](#), [Series](#)



Overview:

1. [Introduction to Pandas – Part 1/2](#)
 1. [Pandas DataFrame](#)
 2. [Pandas Series](#)
 3. [Adding columns to DataFrames](#)
 4. [Deleting columns from DataFrames](#)
 5. [Index](#)
 6. [Accessing elements](#)
 7. [Element-wise operations](#)

Introduction to Pandas – Part 1/2

The last part of the Introduction to ML covers the Python package [Pandas](#).

Pandas is a powerful and versatile data manipulation library in Python. It provides data structures and functions that are essential for data analysis and preprocessing tasks. With Pandas, you can easily load, manipulate, and analyze structured data.

One of the key data structures in Pandas is the **DataFrame**. It is a two-dimensional table-like structure that allows you to store and manipulate data in a row-column format. You can think of it as a spreadsheet or a SQL table. The DataFrame is designed to handle both homogeneous and heterogeneous data, making it suitable for a wide range of applications.

In addition to the DataFrame, Pandas also provides **Series**, which is a one-dimensional labeled array. It is similar to a column in a DataFrame and can be used to store and

manipulate a single variable. Series are particularly useful when you need to perform operations on a specific column or extract a subset of data from a DataFrame.

Pandas also offers a rich set of functions for data manipulation. You can perform operations such as filtering, sorting, transforming, and aggregating data with ease. Additionally, Pandas integrates well with other Python libraries such as NumPy and Matplotlib, allowing you to seamlessly combine data manipulation, numerical computation, and data visualization tasks.

Whether you are working on a small data analysis project or dealing with large datasets, Pandas provides efficient and intuitive tools to handle your data. Its extensive [documentation](#) and active community support make it a popular choice among data scientists and analysts.

Pandas DataFrame

A Pandas **DataFrame** is basically a table. In the following snippet **data** is a list of lists. Each sublist is a row in the table. Each row represents one car.

```
data = [
    ['Nissan', 'Stanza', 1991, 138, 4, 'MANUAL', 'sedan', 2000],
    ['Hyundai', 'Sonata', 2017, None, 4, 'AUTOMATIC', 'Sedan'],
    ['Lotus', 'Elise', 2010, 218, 4, 'MANUAL', 'convertible'],
    ['GMC', 'Acadia', 2017, 194, 4, 'AUTOMATIC', '4dr SUV'],
    ['Nissan', 'Frontier', 2017, 261, 6, 'MANUAL', 'Pickup'],
]

columns = [
    'Make', 'Model', 'Year', 'Engine HP', 'Engine Cylinders',
    'Transmission Type', 'Vehicle_Style', 'MSRP'
]

pd.DataFrame(data)
```

The last line outputs a Pandas Dataframe but without column names.

	0	1	2	3	4	5	6	7
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

DataFrame without any column names

To display the correct output with column names, the columns parameter must be set.

```
| df = pd.DataFrame(data, columns = columns)  
| df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

DataFrame with column names

There is another way to create a DataFrame. This time we use a list of dictionaries. In a dictionary we explicitly specify the value for each column. That means a dictionary is a key-value structure, where the keys are the column names and the values are the car specific value for that key.

```
data = [
    {
        "Make": "Nissan",
        "Model": "Stanza",
        "Year": 1991,
        "Engine HP": 138.0,
        "Engine Cylinders": 4,
        "Transmission Type": "MANUAL",
        "Vehicle_Style": "sedan",
        "MSRP": 2000
    },
    {
        "Make": "Hyundai",
        "Model": "Sonata",
        "Year": 2017,
        "Engine HP": None,
        "Engine Cylinders": 4,
        "Transmission Type": "AUTOMATIC",
        "Vehicle_Style": "Sedan",
        "MSRP": 27150
    },
    {
        "Make": "Lotus",
        "Model": "Elise",
        "Year": 2010,
        "Engine HP": 218.0,
        "Engine Cylinders": 4,
        "Transmission Type": "MANUAL",
        "Vehicle_Style": "convertible",
        "MSRP": 54990
    },
    {
        "Make": "GMC",
        "Model": "Acadia",
        "Year": 2017,
        "Engine HP": 194.0,
        "Engine Cylinders": 4,
        "Transmission Type": "AUTOMATIC",
        "Vehicle_Style": "4dr SUV",
        "MSRP": 34450
    },
    {
        "Make": "Nissan",
        "Model": "Frontier",
        "Year": 2017,
        "Engine HP": 261.0,
        "Engine Cylinders": 6,
        "Transmission Type": "MANUAL",
        "Vehicle_Style": "Pickup",
        "MSRP": 32340
    }
]

pd.DataFrame(data)
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

After loading a DataFrame from csv file or from sql query the first thing to do is to look at the first rows to get a fast overview about the data.

| df.head()

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

df.head() outputs the first rows of the DataFrame

| df.head(n=2)

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	Index
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	0
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	1

df.head(n=2) outputs only the first two rows of the DataFrame

Pandas Series

A Pandas **Series** is a one-dimensional labeled array that can hold data of any type. It is similar to a column in a DataFrame and can be thought of as a single variable. Series provide a powerful way to manipulate and analyze specific columns or subsets of data from a DataFrame.

You can create a Series by passing a list or array of values to the `pd.Series()` function. For example, let's create a Series of the "Make" column from the DataFrame:

```
| make_series = pd.Series(df['Make'])
```

This will create a Series where each element corresponds to the "Make" value of a row in the DataFrame. You can then perform various operations on the Series, such as filtering, sorting, or computing statistics.

For instance, to filter the Series and select only the rows where the make is "Nissan", you can use boolean indexing:

```
| nissan_cars = make_series[make_series == 'Nissan']
```

You can also perform mathematical operations on Series, such as adding or multiplying values. For example, let's say you have a Series representing the MSRP (Manufacturer's Suggested Retail Price) of the cars:

```
| msrp_series = pd.Series(df['MSRP'])
```

You can calculate the average MSRP using the `mean()` method:

```
| average_msrp = msrp_series.mean()
```

You can also apply a function to each element of the Series using the `apply()` method. This allows you to perform custom operations on the data. For example, let's say you want to convert the MSRP values from dollars to euros:

```
| def convert_to_euros(value):
|     return value * 0.85
|
| msrp_in_euros = msrp_series.apply(convert_to_euros)
```

This will create a new Series `msrp_in_euros` where each value is the MSRP converted to euros.

In addition to the above operations, Pandas Series also provide a wide range of methods for data manipulation and analysis. You can find more information and examples in the official [Pandas documentation](#).

By leveraging the power of Pandas Series, you can easily extract, transform, and analyze

specific columns or subsets of data from a DataFrame, making it a versatile tool for data manipulation and analysis in Python.

```
# To access one serie you can use the dot notation...
df.Make
# Output:
# 0      Nissan
# 1      Hyundai
# 2      Lotus
# 3      GMC
# 4      Nissan
# Name: Make, dtype: object

# ... or the bracket notation
df['Make']
# Output:
# 0      Nissan
# 1      Hyundai
# 2      Lotus
# 3      GMC
# 4      Nissan
# Name: Make, dtype: object
```

You can create a new subset of the DataFrame with a selection of columns using the bracket notation.

```
df[['Make', 'Model', 'MSRP']]
```

	Make	Model	MSRP
0	Nissan	Stanza	2000
1	Hyundai	Sonata	27150
2	Lotus	Elise	54990
3	GMC	Acadia	34450
4	Nissan	Frontier	32340

Output of selected columns ‘Make’, ‘Model’, and ‘MSRP’

Adding columns to DataFrames

Sometimes it's necessary to add one more column to the DataFrame. The next snippet shows how you can add a column ‘id’ with predefined values to the DataFrame.

```
df['id'] = [1, 2, 3, 4, 5]
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

Adds a new column 'id' to the DataFrame and sets values for this new column

```
df['id']
# Output:
# 0    1
# 1    2
# 2    3
# 3    4
# 4    5
# Name: id, dtype: int64

df['id'] = [10, 20, 30, 40, 50]
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

Output with changed values for the id column

Deleting columns from DataFrames

To delete a column you can use the **del** operator.

```
del df['id']
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

Output without id column

Index

The numbers in the first column refer to each row (called Index).

```
df.index
# Output: RangeIndex(start=0, stop=5, step=1)
```

Accessing elements

Using this index we can access the elements of the DataFrame. You can output only one element as shown in the first line of next snippet. But you can also return multiple rows as shown in the last line.

```
df.loc[1]
# Output:
# Make          Hyundai
# Model         Sonata
# Year          2017
# Engine HP     NaN
# Engine Cylinders    4
# Transmission Type  AUTOMATIC
# Vehicle Style   Sedan
# MSRP          27150
# Name: 1, dtype: object

df.loc[[1, 2]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	Index
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	1
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	2

We can also replace the index...

```
| df.index = ['a', 'b', 'c', 'd', 'e']
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
a	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
b	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
c	Lotus	Elise	2010	218.0	4	MANUAL	convertible
d	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
e	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

Changing index to a categorical one

```
| # usual index
df.loc[['b', 'c']]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	Index
b	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	4
c	Lotus	Elise	2010	218.0	4	MANUAL	convertible	5

Returning rows by accessing a categorical index

```
| # Still we can refer to a positional index (what we usually use)
df.iloc[[1, 2, 4]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
b	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
c	Lotus	Elise	2010	218.0	4	MANUAL	convertible
e	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

To reset the index we can use `reset_index` function. This keeps the previous index and creates a new column called `index`.

```
| df.reset_index()
```

	<code>index</code>	<code>Make</code>	<code>Model</code>	<code>Year</code>	<code>Engine HP</code>	<code>Engine Cylinders</code>	<code>Transmission Type</code>	<code>Vehicle Style</code>
0	a	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	b	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	c	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	d	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	e	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

DataFrame after resetting the index

If the old index is not needed it can simply be dropped.

```
| df = df.reset_index(drop=True)
df
```

	<code>Make</code>	<code>Model</code>	<code>Year</code>	<code>Engine HP</code>	<code>Engine Cylinders</code>	<code>Transmission Type</code>	<code>Vehicle Style</code>
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

DataFrame after removing the old index

Element-wise operations

When looking at data, `NaN` values can occur. `NaN` means “not a number”. The following example illustrates this.

```
# NaN denotes a missing number
df['Engine HP']
# Output:
# 0    138.0
# 1      NaN
# 2    218.0
# 3    194.0
# 4    261.0
# Name: Engine HP, dtype: float64
```

We can do everything we can do in NumPy, but here we operate on series from Pandas (instead of NumPy arrays). The main difference is, that you have here an index and a name. Under the hood Pandas actually uses NumPy. The next snippet shows the devision of every element by 100.

```
df['Engine HP'] / 100
# Output:
# 0    1.38
# 1      NaN
# 2    2.18
# 3    1.94
# 4    2.61
# Name: Engine HP, dtype: float64
```

But it's also possible to use logical operators.

```
df.Year
# Output:
# 0    1991
# 1    2017
# 2    2010
# 3    2017
# 4    2017
# Name: Year, dtype: int64

df.Year >= 2015
# Output:
# 0    False
# 1     True
# 2    False
# 3     True
# 4     True
# Name: Year, dtype: bool
```

ML Zoomcamp 2023 – Introduction to Machine Learning – Part 13

👤 Peter ⏰ 17. September 2023 📄 [Introduction, ML-Zoomcamp](#)
🏷️ [Filtering, Grouping, ML Zoomcamp](#)



Overview:

1. [Introduction to Pandas – Part 2/2](#)
 1. [Filtering](#)
 2. [String operations](#)
 3. [Summarizing operations](#)
 1. [Numerical Columns](#)
 1. [Describe function](#)
 2. [Categorical Columns](#)
 4. [Missing values](#)
 5. [Grouping](#)
 6. [Getting the NumPy arrays](#)
 7. [Getting a list of dictionaries](#)

Introduction to Pandas – Part 2/2

Filtering

Filtering refers to the process of selecting specific rows or columns from a DataFrame based on certain conditions. In Pandas, we can use various techniques to filter our data.

One common technique is to use boolean indexing. This involves creating a boolean mask that specifies the conditions we want to apply to our data. The mask is a DataFrame or Series of the same shape as the original data, where each element is either True or False depending on whether the corresponding element in the original data satisfies the

condition.

For example, if we have a DataFrame `df` with a column named `Year`, and we want to filter out all rows where the year is greater or equal to 2015, we can create our condition as follows:

```
| condition = df.Year >= 2015
```

We can then use this condition to select the desired rows from the DataFrame:

```
| filtered_df = df[condition]
```

```
| # or like this  
| df[df.Year >= 2015]
```

Another useful technique for filtering is using the `.query()` method. This method allows us to filter rows based on a string expression, similar to writing SQL queries. For example, to filter out all rows where the year is greater or equal to 2015 using `.query()`, we can do:

```
| filtered_df = df.query('year >= 2015')
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

Using filtering Year >= 2015

The next example is a filter on the make Nissan.

```
| df[  
|     df.Make == 'Nissan'  
| ]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	M
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	3

Using filtering Make == 'Nissan'

It is also possible to combine several conditions, for example, you want to have all Nissans after the year 2015.

```
df[  
    (df.Make == 'Nissan') & (df.Year > 2015)  
]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	N
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	3

Using chained filtering Make == 'Nissan' & Year > 2015

By using these filtering techniques, we can easily extract the data we need from our DataFrame and perform further analysis or computations on it. This can be particularly useful when working with large datasets or when dealing with complex conditions.

String operations

Next, let's explore string operations in Pandas. That is something NumPy doesn't have, because NumPy is mostly used for processing numbers. In Pandas we often have to deal with strings.

```
|
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup

Unchanged DataFrame

The next snippet shows the Vehicle_Style column. You can immediately see the different formatting. There are different spellings of the same words and there are spaces.

```
df['Vehicle_Style']
# Output:
# 0          sedan
# 1          Sedan
# 2    convertible
# 3      4dr SUV
# 4       Pickup
# Name: Vehicle_Style, dtype: object
```

There is a string operator to lower the string as you can see here:

```
'STRr'.lower()
# Output: 'strr'
```

Using this string operator, the output of Vehicle_Style looks more consistent, but still there are the spaces...

```
df['Vehicle_Style'].str.lower()
# Output:
# 0          sedan
# 1          sedan
# 2    convertible
# 3      4dr suv
# 4       pickup
# Name: Vehicle_Style, dtype: object
```

A typical pre-processing step when you work with text is to replace all spaces with underscores.

```
'machine learning zoomcamp'.replace(' ', '_')
# Output: 'machine_learning_zoomcamp'
```

Using this string operator, the output of Vehicle_Style looks perfect.

```
df['Vehicle_Style'].str.replace(' ', '_')
# Output:
# 0          sedan
# 1          Sedan
# 2    convertible
# 3      4dr_SUV
# 4       Pickup
# Name: Vehicle_Style, dtype: object

# Both operations can be chained
df['Vehicle_Style'] = df['Vehicle_Style'].str.lower().str.replace(' ', '_')
df
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	sedan
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr_suv
4	Nissan	Frontier	2017	261.0	6	MANUAL	pickup

Applying string operations to Vehicle_Style column

Summarizing operations

```
df.MSRP
# Output:
# 0    2000
# 1    27150
# 2    54990
# 3    34450
# 4    32340
# Name: MSRP, dtype: int64

df.MSRP.min()
# Output: 2000

df.MSRP.max()
# Output: 54990

df.MSRP.mean()
# Output: 30186.0
```

Numerical Columns

Describe function

It can be very helpful to get an overview with the describe function. This gives you a lot of information at once (count, mean, std, min, max, and the quantiles for 25%, 50% and 50%). There it can be used only for numerical columns. You can use it for one column...

```
df.MSRP.describe()  
# Output:  
# count      5.000000  
# mean     30186.000000  
# std      18985.044904  
# min      2000.000000  
# 25%      27150.000000  
# 50%      32340.000000  
# 75%      34450.000000  
# max      54990.000000  
# Name: MSRP, dtype: float64
```

... or apply it on the DataFrame itself. In this case it finds all numerical columns and computes the statistics.

| df.describe()

	Year	Engine HP	Engine Cylinders	MSRP
count	5.000000	4.00000	5.000000	5.000000
mean	2010.400000	202.75000	4.400000	30186.000000
std	11.260551	51.29896	0.894427	18985.044904
min	1991.000000	138.00000	4.000000	2000.000000
25%	2010.000000	180.00000	4.000000	27150.000000
50%	2017.000000	206.00000	4.000000	32340.000000
75%	2017.000000	228.75000	4.000000	34450.000000
max	2017.000000	261.00000	6.000000	54990.000000

Data have many digits after decimal point

This overview is a bit confusing because of the many decimal digits. With round(2) you can round all numbers to two decimal digits. The output looks much nicer now.

| df.describe().round(2)

	Year	Engine HP	Engine Cylinders	MSRP
count	5.00	4.00	5.00	5.00
mean	2010.40	202.75	4.40	30186.00
std	11.26	51.30	0.89	18985.04
min	1991.00	138.00	4.00	2000.00
25%	2010.00	180.00	4.00	27150.00
50%	2017.00	206.00	4.00	32340.00
75%	2017.00	228.75	4.00	34450.00
max	2017.00	261.00	6.00	54990.00

Data with rounded values

Categorical Columns

There are also some functions for string values. The next function can be used for numerical values, too. Nunique() returns the number of unique values. You can apply this on single columns or DataFrames.

```
# Returns the number of unique values of column Make
df.Make.nunique()
# Output: 4

# Returns the number of unique values for all columns of the
df.nunique()
# Output:
# Make          4
# Model         5
# Year          3
# Engine HP     4
# Engine Cylinders 2
# Transmission Type 2
# Vehicle_Style   4
# MSRP           5
# dtype: int64
```

However, if you are interested in the unique values themselves, you can use the unique() function.

```
df.Year.unique()
# Output
array([1991, 2017, 2010])
```

Missing values

Missing values can make our lives more difficult. That's why it makes sense to take a look at this problem. The function `isnull()` returns true for each value/cell that is missing.

```
| df.isnull()
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle Style	MS
0	False	False	False	False	False	False	False	Fal
1	False	False	False	True	False	False	False	Fal
2	False	False	False	False	False	False	False	Fal
3	False	False	False	False	False	False	False	Fal
4	False	False	False	False	False	False	False	Fal

Showing missing values for each row

The representation is very confusing therefor it's more useful to sum up the number of missing values for each column.

```
df.isnull().sum()  
# Output:  
# Make          0  
# Model         0  
# Year          0  
# Engine HP     1  
# Engine Cylinders 0  
# Transmission Type 0  
# Vehicle_Style 0  
# MSRP          0  
# dtype: int64
```

Grouping

Grouping is another option that is suitable to get an overview. In this query we are interested in the information from the transmission_type and MSRP columns, sorted by transmission_type. The grouping gives the average price for each transmission_type.

```
SELECT
    transmission_type,
    AVG(MSRP)
FROM
    cars
GROUP BY
    transmission_type
```

```
df.groupby('Transmission Type').MSRP.mean()
# Output:
# Transmission Type
# AUTOMATIC      30800.000000
# MANUAL         29776.666667
# Name: MSRP, dtype: float64
```

There are many other interesting applications of grouping, e.g. min() or max(). Even describe() can be used here.

```
df.groupby('Transmission Type').MSRP.min()
# Output:
# Transmission Type
# AUTOMATIC      27150
# MANUAL         2000
# Name: MSRP, dtype: int64

df.groupby('Transmission Type').MSRP.max()
# Output:
# Transmission Type
# AUTOMATIC      34450
# MANUAL         54990
# Name: MSRP, dtype: int64

df.groupby('Transmission Type').MSRP.describe()
```

Transmission Type	count	mean	std	min	25%	50%
AUTOMATIC	2.0	30800.000000	5161.879503	27150.0	28975.0	30800.0
MANUAL	3.0	29776.666667	26587.836191	2000.0	17170.0	32340.0

Using grouping of Transmission Type

Getting the NumPy arrays

Sometimes it is necessary to convert a Pandas DataFrame back to the underlying NumPy array.

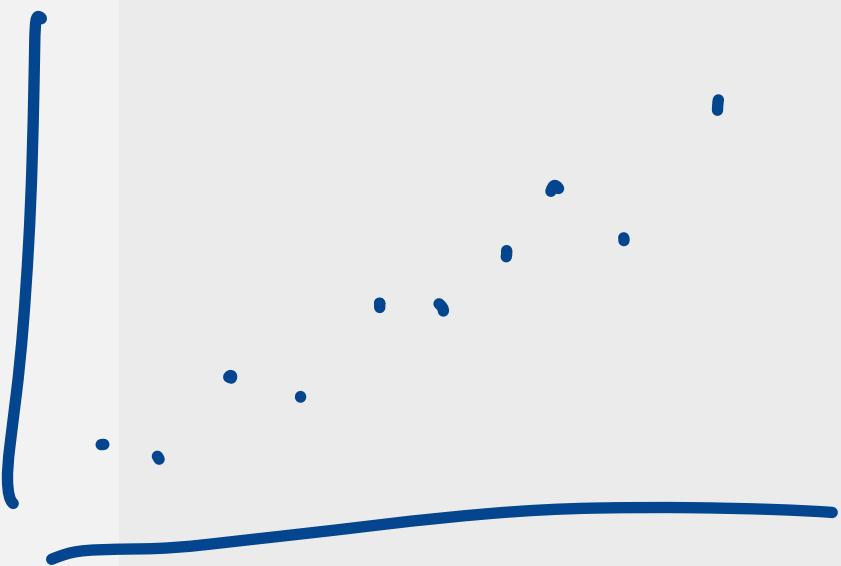
```
df.MSRP.values  
# Output:  
# array([ 2000, 27150, 54990, 34450, 32340])
```

Getting a list of dictionaries

To convert a Pandas DataFrame back to the form of a list of dictionaries. This form is used to save it to a file.

```
df.to_dict(orient='records')  
# Output:  
# [ {'Make': 'Nissan',  
#   'Model': 'Stanza',  
#   'Year': 1991,  
#   'Engine HP': 138.0,  
#   'Engine Cylinders': 4,  
#   'Transmission Type': 'MANUAL',  
#   'Vehicle Style': 'sedan',  
#   'MSRP': 2000},  
#   {'Make': 'Hyundai',  
#    'Model': 'Sonata',  
#    'Year': 2017,  
#    'Engine HP': nan,  
#    'Engine Cylinders': 4,  
#    'Transmission Type': 'AUTOMATIC',  
#    'Vehicle Style': 'sedan',  
#    'MSRP': 27150},  
#   {'Make': 'Lotus',  
#    'Model': 'Elise',  
#    'Year': 2010,  
#    'Engine HP': 218.0,  
#    'Engine Cylinders': 4,  
#    'Transmission Type': 'MANUAL',  
#    'Vehicle Style': 'convertible',  
#    'MSRP': 54990},  
#   {'Make': 'GMC',  
#    'Model': 'Acadia',  
#    'Year': 2017,  
#    'Engine HP': 194.0,  
#    'Engine Cylinders': 4,  
#    'Transmission Type': 'AUTOMATIC',  
#    'Vehicle Style': '4dr_suv',  
#    'MSRP': 34450},  
#   {'Make': 'Nissan',  
#    'Model': 'Frontier',  
#    'Year': 2017,  
#    'Engine HP': 261.0,  
#    'Engine Cylinders': 6,  
#    'Transmission Type': 'MANUAL',  
#    'Vehicle Style': 'pickup',  
#    'MSRP': 32340}]
```

LINEAR REGRESSION



SUMMARY

We have a table with features of a car and want to use it to predict car prices (using the prices of the cars listed under MSRP variable).

We first cleaned the dataset (capital cases, spaces between words, ...). We then did exploratory data analysis and found a long tail distribution of prices, which we solved using a logarithmic transformation of the data. We normalized it. We got rid of missing data with `isnull()`. We set up the validation framework and got a train set. We did linear regression to see how it works for a single example, we then saw how to use it in vector mode. The result of the regression is w . We trained the model and created a model, but the RMSE was large. We then validated the model and engineered $\|z\|^2$ to reduce RMSE. We checked how to include the categorical variables with binary columns, which increased RMSE, for which we used regularization after finding the best regularization parameter (r). Finally, we trained the model and used it to predict the price of a new car.

ML Zoomcamp 2023 – Machine Learning for Regression – Part 1

👤 Peter ⏰ 18. September 2023 📁 ML-Zoomcamp, Regression
🏷️ [Data Cleaning](#), [Data preparation](#), [ML Zoomcamp](#)



1. [Car price prediction project](#)
2. [Data preparation – General Information](#)
 1. [Data preparation – Car price prediction project](#)
 1. [Loading Data and get an overview](#)
 2. [Cleaning](#)

Car price prediction project

This chapter is about the implementation of an ML project and which steps have to be considered. It is about the prediction of car prices based on a [Kaggle dataset](#).

The steps will be described in individual blog posts and include:

1. Data preparation
2. EDA (Exploratory Data Analysis)
3. Use linear regression for price prediction
(MSRP – Manufacturer suggested retail price)
4. Understand the internals of linear regression
5. Evaluating the model with RMSE (root mean squared error)
6. Feature Engineering (creating new features)
7. Regularization
8. Using the model

Data preparation – General Information

In the data preparation phase, several important steps need to be followed to ensure the dataset is suitable for analysis and modeling. Here are some key considerations:

1. **Data Cleaning:** This involves handling missing values, dealing with outliers, and ensuring consistency in data formats. Missing values can be imputed using various techniques, such as mean/median imputation or using advanced methods like K-nearest neighbors. Outliers may need to be addressed by either removing them or transforming them to fall within a reasonable range.
2. **Data Integration:** If you have multiple datasets related to car prices, you may need to combine them into a single dataset. This can involve matching and merging records based on common identifiers or performing data joins based on shared attributes.
3. **Data Transformation:** Sometimes, the existing variables may not be in a suitable format for analysis. In such cases, feature engineering techniques can be applied to create new variables that may have a better relationship with the target variable, such as transforming categorical variables into numerical ones using one-hot encoding or label encoding.
4. **Feature Scaling:** It is crucial to make sure that the features are on a similar scale to avoid bias in the model. Common techniques for feature scaling include standardization (mean of 0 and standard deviation of 1) or normalization (scaling values between 0 and 1).
5. **Train-Validation Split:** Before building the predictive model, it is essential to split the dataset into training and validating subsets. Typically, the majority of the data is used for training, while a smaller portion is reserved for evaluating the model's performance. As I mentioned in past articles, a train-validate-test split might provide more reliable results.

By following these steps diligently, you can ensure that the data is well-prepared and ready for the subsequent stages of the car price prediction project.

Data preparation – Car price prediction project

This section covers a few topics that are mentioned before to get a better understanding about what happens in the preparation step.

Loading Data and get an overview

```
import pandas as pd
import numpy as np

# reading csv file after downloading...
df = pd.read_csv('data.csv')

# ... and getting a first overview about the data
df.head()

# What you can see here, there is some inconsistency
# in the way of naming columns
# -> sometimes the columns have underscores, sometimes not,
#      sometimes the columns have capital letters, sometimes
#
# df['Transmission Type']      is working
# df.Transmission Type         is not working because of space
```

Cleaning

To make the columns more consistent we might decide to make them all lowercased and we might replace spaces with underscores. The following code snippets show how to get this.

```
# Pandas DataFrame has a field called columns,  
# that contains the name of the columns  
# columns is an index, that is a special data structure  
# in Pandas (very similar to series)  
df.columns  
  
# like series it also has the str method for doing string  
# manipulation what we can do now is to apply the same  
# string function to all column names  
df.columns = df.columns.str.lower().str.replace(' ', '_')  
df.head()
```

Actually we have the same problems with the values. Before we can apply that, we need to detect all string columns, because the str function works only on strings.

```
# dtypes returns for all the columns what is the type of this  
# column and here we're interested in "objects"  
#  
# In case of csv files "objects" cannot be something  
# different than strings  
df.dtypes  
df.dtypes == 'object'  
  
# to select only the objects  
df.dtypes[df.dtypes == 'object']
```

The output of the last line of code in the last snippet are the values and the index of the series. We're not interested in values here, but we're interested in the names.

```
# Get access to the index of that series  
# Converting it to a python list with name strings  
strings = list(df.dtypes[df.dtypes == 'object'].index)  
strings
```

Similar to what we've done with the column names we want to apply to the specified columns.

```
df['make'].str.lower().str.replace(' ', '_')  
  
# Better way  
for col in strings:  
    df[col] = df[col].str.lower().str.replace(' ', '_')  
  
df.head()
```

ML Zoomcamp 2023 – Machine Learning for Regression – Part 2

👤 Peter 📅 19. September 2023 📁 ML-Zoomcamp, Regression
🏷️ Exploratory Data Analysis, Missing Values, ML Zoomcamp, Unique Values



1. [Exploratory data analysis \(EDA\) – General Information](#)
2. [Exploratory data analysis \(EDA\) – Car price prediction project](#)
 1. [Getting an overview](#)
 2. [Distribution of price](#)
 3. [Missing values](#)

Exploratory data analysis (EDA) – General Information

Exploratory data analysis (EDA) is an essential step in the data analysis process. It involves summarizing and visualizing the main characteristics of a dataset to gain insights and identify patterns or trends. By exploring the data, researchers can uncover hidden relationships between variables and make informed decisions.

One common technique in EDA is to calculate summary statistics like mean, median, and standard deviation to understand the distribution of the data. These statistics provide a general overview of the dataset and can help identify potential outliers or unusual patterns.

Visualizations also play a crucial role in EDA. Graphical representations such as histograms, scatter plots, and box plots help visualize the data distribution, identify clusters or groups, and detect any unusual patterns or trends. Visualizations can be particularly helpful in identifying relationships between variables or finding patterns that may not be immediately apparent.

Another important aspect of EDA is data cleaning. This involves handling missing values, outliers, and inconsistencies in the dataset. By carefully examining the data, researchers

can decide how to handle missing values (e.g., imputing or removing them) and identify and address outliers or errors.

EDA is not a one-time process but rather an iterative one. As researchers delve deeper into the data, they may uncover additional questions or areas of interest that require further exploration. Through this iterative process, researchers refine their understanding of the data and uncover valuable insights.

In conclusion, exploratory data analysis is a crucial step in the data analysis process. By summarizing, visualizing, and cleaning the data, researchers can uncover patterns, identify relationships, and make informed decisions. It provides the foundation for more advanced data analysis techniques and helps in the formation of hypotheses for further investigation.

Exploratory data analysis (EDA) – Car price prediction project

This section covers a few topics that are mentioned before to get a better understanding about what you can do in the EDA.

Getting an overview

First we want to understand how the data looks like just to get a feeling what values are there. That helps to learn more about the problem. What you can do is to look at each column and print some values.

```
for col in df.columns:  
    print(col)  
    print(df[col].head())  
    print()
```

The output is not very informative, but what about **unique values**?

```
for col in df.columns:  
    print(col)  
    # print only the first 5 values  
    # print(df[col].unique()[:5])  
    print(df[col].unique())  
    print("number of unique values: ",df[col].nunique())  
    print()
```

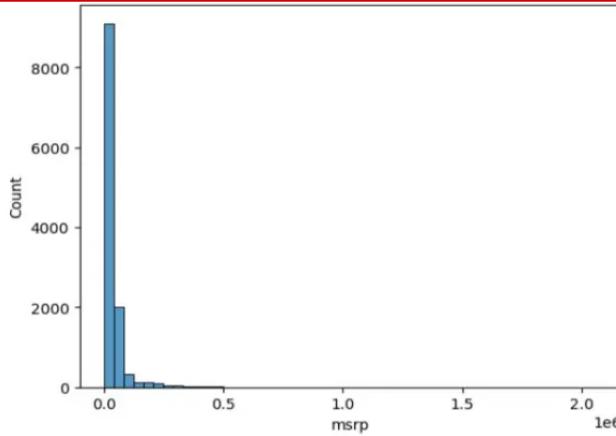
Distribution of price

Next we want to look at the price and visualize this column.

```
# For plotting we use two libraries
import matplotlib.pyplot as plt
import seaborn as sns

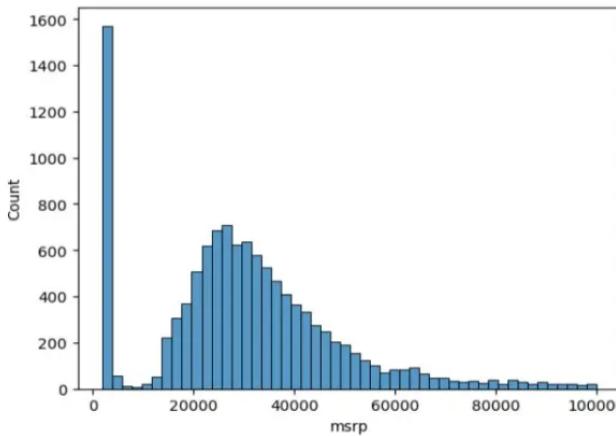
# this line is needed to display the plots in notebooks
%matplotlib inline

# bins = number of bars in the histogram
# in the diagram 1e6 means  $10^6 = 1,000,000$ 
sns.histplot(df.msrp, bins=50)
```



What you can see in the histogram, there are a lot of prices that are pretty cheap but only a few cars that are very expensive. That means this is a long-tail distribution (many prices in a small range, but a few prices in a wide range). We need to zoom in a bit to “ignore” the long tail with too less datapoints.

```
sns.histplot(df.msrp[df.msrp < 100000], bins=50)
```



This kind of distribution (long tail, and the peak) is not good for ML models, because this distribution will confuse them. There is a way to get rid of the long tail, by applying logarithm to the price. This results in more compact values.

```

#np.log([0, 1,10,1000,100000])
# problem with logarithm is when we have a 0, because log(0)
#np.log([0 + 1, 1 + 1, 10 + 1, 1000 + 1, 100000 + 1])
# Output: array([ 0. , 0.69314718, 2.39789527, 6.90
#
# to not always add 1 there is a NumPy function
#np.log1p([0, 1,10,1000,100000])
# Output: array([ 0. , 0.69314718, 2.39789527, 6.90

price_logs = np.log1p(df.msrp)
sns.histplot(price_logs, bins=50)

```

You can see the long tail is gone and you see a nice **bell curve** shape of a so called **normal distribution**, what is ideal for ML models. But still there is the strange peak. This could be the minimum price of \$1,000 of the platform.

Missing values

As the title suggests, this is about finding missing values (NaN values). We can use the function in the following snippet to find that values. The sum function sums across columns and shows for each column how much missing values are there. This information is important when training a model.

```

df.isnull().sum()
# Output:
# make          0
# model         0
# year          0
# engine_fuel_type 3
# engine_hp      69
# engine_cylinders 30
# transmission_type 0
# driven_wheels   0
# number_of_doors 6
# market_category 3742
# vehicle_size    0
# vehicle_style    0
# highway_mpg      0
# city_mpg        0
# popularity       0
# msrp            0
# dtype: int64

```

ML Zoomcamp 2023 – Machine Learning for Regression – Part 3

👤 Peter ⏰ 19. September 2023 📁 ML-Zoomcamp, Regression 🏷️ ML Zoomcamp



Setting up the validation framework

To validate the model, we take the dataset and split it into three parts (train-val-test / 60-20-20). The reason why this is useful was mentioned in an earlier blog post. This means that we train the model on the training dataset, check if it works fine on the validation dataset, and leave the test dataset for the end. We only use the test dataset very occasionally, and only to check if the model is performing well. For each of these three parts, we create the feature matrix X and the target variable y (X_{train} , y_{train} , X_{val} , y_{val} , X_{test} , y_{test}). So, what we need to do is calculate how much 20% is.

```
# Returns the number of records of the whole dataset
len(df)
# Output: 11914

# Calculate 20% of whole dataset
int(len(df) * 0.2)
# Output: 2382
```

With this preliminary work from the last code snippet, we can complete the calculation for splitting into the three datasets.

```

n = len(df)
n_val = n_test = int(n * 0.2)
n_train = n - n_val - n_test
n , n_val+n_test+n_train
# Output: (11914, 11914)

# sizes of our dataframes
n_val, n_test, n_train
# Output: (2382, 2382, 7150)

df_train = df.iloc[:n_train]
df_val = df.iloc[n_train:n_train + n_val]
df_test = df.iloc[n_train + n_val:]

```

You might think that this concludes the division, but there is one crucial problem. This approach brings us to the problem that it's sequential. That's a problem when there is an order in the dataset. That means we need to shuffle, otherwise, there are BMWs only in one dataset. Generally shuffling is always a good idea.

```

idx = np.arange(n)
idx
# Output: array([ 0, 1, 2, ..., 11911, 11912, 11913])

# to make it reproducible
#np.random.seed(2)
np.random.shuffle(idx)
idx
# Output: array([11545, 7488, 263, ..., 3119, 1696, 905])

```

Using this shuffled index we can create our shuffled datasets for training, validation and for testing.

```

# Create shuffled datasets with correct size
df_train = df.iloc[idx[:n_train]]
df_val = df.iloc[idx[n_train:n_train + n_val]]
df_test = df.iloc[idx[n_train + n_val:]]

```

Now there is no order in the index column so we can reset index and drop the old index column.

```

df_train = df_train.reset_index(drop=True)
df_val = df_val.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)

```

As I mentioned in the last blog article we should apply the log1p transformation to the price column to help the model perform well.

```

y_train = np.log1p(df_train.msrp.values)
y_val = np.log1p(df_val.msrp.values)
y_test = np.log1p(df_test.msrp.values)

```

There is one final but very important step. We should remove msrp values from dataframes (df_train, df_val, df_test) to make sure that we don't accidentally use it for training purposes.

```
del df_train['msrp']
del df_val['msrp']
del df_test['msrp']
```

 [Peter](#)  [19. September 2023](#)  [ML-Zoomcamp, Regression](#)  [ML Zoomcamp](#)

Leave a comment

Write a comment...

 Comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Machine Learning for Regression – Part 4

👤 Peter ⏰ 20. September 2023 📄 ML-Zoomcamp, Regression
🏷️ [Linear regression](#), [ML Zoomcamp](#)



1. [Linear regression](#)
 1. [Step back and focus on one observation](#)
 2. [Implementation of linear regression function](#)

Linear regression

Let's delve deeper into the topic of linear regression.

Linear regression is a fundamental statistical technique used in the field of machine learning for solving regression problems. In simple terms, regression analysis involves predicting a continuous outcome variable based on one or more input features. That means the output of the model is a number.

In the case of linear regression, the basic idea is to find the best-fitting linear relationship between the input features and the output variable. This relationship is represented by a linear equation of the form:

$$y = w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n$$

Here, "y" represents the output variable, and x_1, x_2, \dots, x_n represent the input features. The $w_0, w_1, w_2, \dots, w_n$ are the coefficients that determine the relationship between the input features and the output variable.

The goal of linear regression is to estimate the values of these coefficients in such a way that the sum of squared differences between the observed and predicted values is minimized. This minimization is typically achieved using a method called "ordinary least

squares,” which calculates the best-fitting line by minimizing the sum of the squared errors between the predicted and actual values.

Linear regression is widely used in various fields, including economics, finance, social sciences, and engineering, to name just a few. It provides a simple and interpretable way to understand the relationship between the input features and the continuous outcome variable.

In summary, linear regression is a powerful tool for predicting numerical values based on input features. By finding the best-fitting linear equation, it enables us to make accurate predictions and gain insights into the relationship between variables.

Step back and focus on one observation

Just to recap what we know from the articles before, there is the function $g(X) \sim y$ with: g as the model (in our example Linear regression), X as the feature matrix and y as the target (in our example price).

Now let's step back and look at only one observation. The corresponding function is $g(x_i) \sim y_i$ with: x_i as one car and y_i as its price. So x_i is one specific row of the feature matrix X . We can think of it as a vector with multiple elements (n different characteristics of this one car).

$$x_i = (x_{i1}, x_{i2}, x_{in}) \rightarrow g(x_{i1}, x_{i2}, x_{in}) \sim y_i$$

Let's look at one example and how this looks in code.

```
df_train.iloc[10]

# Output:
# make          chevrolet
# model         sonic
# year          2017
# engine_fuel_type regular_unleaded
# engine_hp     138.0
# engine_cylinders 4.0
# transmission_type automatic
# driven_wheels front_wheel_drive
# number_of_doors 4.0
# market_category NaN
# vehicle_size   compact
# vehicle_style  sedan
# highway_mpg    34
# city_mpg       24
# popularity      1385
# Name: 10, dtype: object
```

We take as an example the characteristic enging_hp, city_mpg, and popularity.

$$xi = [138, 24, 1385]$$

That's almost everything we need to implement $g(x_i) \sim y_i$:

- $x_i = (138, 24, 1385)$
- with $i = 10$
- need to implement the function $g(x_{i1}, x_{i2}, \dots, x_{in}) \sim y_i$

```
# in code this would look like --> this is what we want to imp
def g(xi):
    # do something and return the predicted price
    return 10000

g(xi)
# Output: 10000
```

However, this function g is still not very useful, because it always returns a fixed price. We need to implement the function

$$g(x_i) = w_0 + w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i3}$$

with w_0 as bias term and w_1 , w_2 , and w_3 as weights. This formula can be written as

$$g(x_i) = w_0 + \sum_{j=1}^3 w_j x_{ij}$$

Implementation of linear regression function

In general and because of array implementation in Python (indices of arrays start with 0 instead of 1), the formula for linear regression looks like this:

$$g(x_i) = w_0 + \sum_{j=0}^{n-1} w_j x_{ij}$$

The following snippet shows the implementation of the g -function (renamed as `linear_regression`).

```

def linear_regression(xi):
    n = len(xi)
    pred = w0

    for j in range(n):
        pred = pred + w[j] * xi[j]

    return pred

# sample values for w0 and w and the given xi
xi = [138, 24, 1385]
w0 = 0
w = [1, 1, 1]

linear_regression(xi)
# Output: 1547

# try some other values
w0 = 7.17
w = [0.01, 0.04, 0.002]
linear_regression(xi)
# Output: 12.280000000000001

```

What does this actually mean?

We've just implemented the formula as mentioned before with given values:

$$7.17 + 138 \cdot 0.01 + 24 \cdot 0.04 + 1385 \cdot 0.002 = 12.28$$

- $w_0 = 7.17$ bias term = the prediction of a car, if we don't know anything about this
- engine_hp: $138 \cdot 0.01$ that means in this case per 100 hp the price will increase by \$1
- city_mpg: $24 \cdot 0.04$ that means analog to hp, the more gallons the higher the price will be
- popularity: $1385 \cdot 0.002$ analog, but it doesn't seem that it's affecting the price too much, so for every extra mention on twitter the car becomes just a little bit more expensive

There is still one important step to do. Because we logarithmized ($\log(y+1)$) the price at the beginning, we now have to undo that. This gives us the predicted price in \$. Our car has a price of \$215,344.72.

```

# Get the real prediction for the price in $
# We do "-1" here to undo the "+1" inside the log
np.exp(12.280000000000001) - 1
# Output: 215344.7166272456

# Shortcut to not do -1 manually
np.expm1(12.280000000000001)
# Output: 215344.7166272456

# Just for checking only
np.log1p(215344.7166272456)
# Output: 12.280000000000001

```

ML Zoomcamp 2023 – Machine Learning for Regression – Part 5

👤 Peter ⏰ 20. September 2023 📄 ML-Zoomcamp, Regression
🏷️ [Linear regression](#), [ML Zoomcamp](#)



Linear regression vector form

This article covers the generalization to a vector form of what we did in the last article. That means coming back from only one observation x_i (of one car) to the whole feature matrix X .

$$g(x_i) = w_0 + \sum_{j=0}^{n-1} w_j x_{ij}$$

Looking at the last part of this formula we see the dot product (vector-vector multiplication). $g(x_i) = w_0 + x_i^T w$

Let's implement again the dot product (vector-vector multiplication)

```
def dot(xi, w):
    n = len(xi)

    res = 0.0
    for j in range(n):
        res = res + xi[j] * w[j]

    return res
```

Based on that the implementation of the linear_regression function could look like:

```
def linear_regression(xi):
    return w0 + dot(xi, w)
```

To make the last equation more simple, we can imagine there is one more feature x_{i0} , that is always equal to 1.

$$g(x_i) = w_0 + x_i^T w \rightarrow g(x_i) = w_0 x_{i0} + x_i^T w$$

That means vector w becomes a $n+1$ dimensional vector:

- $w = [w_0, w_1, w_2, \dots, w_n]$
- $x_i = [x_{i0}, x_{i1}, x_{i2}, \dots, x_{in}] = [1, x_{i1}, x_{i2}, \dots, x_{in}]$
- $w^T x_i = x_i^T w = w_0 + \dots$

That means we can use the dot product for the entire regression.

```
xi = [138, 24, 1385]
w0 = 7.17
w = [0.01, 0.04, 0.002]

# adding w0 to the vector w
w_new = [w0] + w
w_new
# Output: [7.17, 0.01, 0.04, 0.002]

xi
# Output: [138, 24, 1385]
```

The updated code for linear_regression function looks now like

```
def linear_regression(xi):
    xi = [1] + xi
    return dot(xi, w_new)

linear_regression(xi)
# Output: 12.280000000000001
```

Having done this, let's go back to thinking about all the examples together.
 X is a $m \times (n+1)$ dimensional matrix (with m rows and $n+1$ columns)

$$X = \begin{bmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & x_{21} & \dots & x_{2n} \\ 1 & x_{31} & \dots & x_{3n} \\ \dots & \dots & \dots & \dots \\ 1 & x_{m1} & \dots & x_{mn} \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \dots \\ w_n \end{bmatrix}$$

What we have to do here, for each row of X we multiply this row with the vector w . This vector contains our predictions, therefor we call it y_{pred} .

To sum up. What we need to do to get our model g is a matrix vector multiplication between X and w .

```
w0 = 7.17
w = [0.01, 0.04, 0.002]
w_new = [w0] + w

x1 = [1, 148, 24, 1385]
x2 = [1, 132, 25, 2031]
x10 = [1, 453, 11, 86]

# X becomes a list of lists
X = [x1, x2, x10]
X
# Output: [[1, 148, 24, 1385], [1, 132, 25, 2031], [1, 453, :]

# This turns the list of lists into a matrix
X = np.array(X)
X
# Output:
# array([[ 1, 148, 24, 1385],
#        [ 1, 132, 25, 2031],
#        [ 1, 453, 11, 86]])

# Now we have predictions, so for each car we have a price for it
y = X.dot(w_new)

# shortcut to not do -1 manually to get the real $ price
np.expm1(y)
# Output: array([237992.82334859, 768348.51018973, 222347.221])
```

The next snippet shows the implementation of the adapted linear_regression function

```
def linear_regression(X):
    return X.dot(w_new)

y = linear_regression(X)
np.expm1(y)
# Output: array([237992.82334859, 768348.51018973, 222347.221])
```

Maybe you wonder where the **w_new** vector comes from. That's the topic of the next article.

ML Zoomcamp 2023 – Machine Learning for Regression – Part 6

👤 Peter ⏰ 21. September 2023 📁 ML-Zoomcamp, Regression
🏷️ Gram matrix, Linear regression, ML Zoomcamp



Training a linear regression model

From the last article we know that we need to multiply the feature matrix X with weights vector w to get y (the prediction for price).

$$g(X) = Xw \sim y$$

Actually we want this Xw to be equal to y , but often it's not possible.
To achieve this, we need to find a way to compute w .

$$Xw = y \text{ can be written as } X^{-1}Xw = X^{-1}y \text{ that is } Iw = X^{-1}y$$

The last formula can be simplified to $w = X^{-1}y$ because we know that $Iw = wI = w$ for every vector w and Identity matrix I .

But there is a problem. X^{-1} exists only for **squared matrices** and X is of dimension $m \times (n+1)$ which is not squared in almost every case.

We need to find an approximation for this $X^T X w = X^T y$

$X^T X$ is called the **GRAM MATRIX** and for $X^T X$ the inverse exists, because this is squared $(n+1) \times (n+1)$

$$(X^T X)^{-1} X^T X w = (X^T X)^{-1} X^T y$$

$$(X^T X)^{-1} X^T X = Iw$$

$$Iw = (X^T X)^{-1} X^T y$$

That value is the closest possible solution

$$w = (X^T X)^{-1} X^T y$$

Now we know what we have to do now. We need to implement the function `train_linear_regression`, that takes the feature matrix X and the target variable y and returns the vector w .

```
def train_linear_regression(X, y):
    pass
```

To approach this implementation we first use a simplified example.

```
X = [
    [148, 24, 1385],
    [132, 25, 2031],
    [453, 11, 86],
    [158, 24, 185],
    [172, 25, 201],
    [413, 11, 83],
    [38, 54, 185],
    [142, 25, 431],
    [453, 31, 86],
]
X = np.array(X)
X
# Output:
# array([[ 148,   24, 1385],
#        [ 132,   25, 2031],
#        [ 453,   11,   86],
#        [ 158,   24, 185],
#        [ 172,   25, 201],
#        [ 413,   11,   83],
#        [ 38,   54, 185],
#        [ 142,   25, 431],
#        [ 453,   31,   86]])
```

From the last article we know that we need to add a new column with ones to the feature matrix X . That is for the multiplication with w_0 . We remember that we can use `np.ones()` to get a vector with ones at each position.

```
ones = np.ones(9)
ones
# Output: array([1., 1., 1., 1., 1., 1., 1., 1., 1.])

# X.shape[0] looks at the number of rows and creates the vector
ones = np.ones(X.shape[0])
ones
# Output: array([1., 1., 1.])
```

Now we need to stack this vector of ones with our feature matrix X. For this we can use the NumPy function `np.column_stack()` as shown in the next snippet.

```
np.column_stack([ones, ones])
# Output:
# array([[1., 1.],
#        [1., 1.],
#        [1., 1.],
#        [1., 1.],
#        [1., 1.],
#        [1., 1.],
#        [1., 1.],
#        [1., 1.]])
```

```
X = np.column_stack([ones, X])
y = [10000, 20000, 15000, 25000, 10000, 20000, 15000, 25000, :]

# GRAM MATRIX
XTX = X.T.dot(X)

# Inverse GRAM MATRIX
XTX_inv = np.linalg.inv(XTX)
```

In the following code snippet we test whether the multiplication of XTX with XTX_inv actually produces the Identity matrix I.

```

# Without round(1) it's not exactly identity matrix but the elements
# are very close to 0 --> we can treat them as 0 and take it
XTX.dot(XTX_inv)
# Output:
#array([[ 1.00000000e+00,  2.60208521e-18,  4.85722573e-17,
#        1.08420217e-18],
#       [ 4.54747351e-13,  1.00000000e+00,  1.50990331e-14,
#        2.22044605e-16],
#       [ 5.68434189e-14,  1.11022302e-16,  1.00000000e+00,
#        3.46944695e-17],
#       [ 9.09494702e-13,  0.00000000e+00, -2.13162821e-14,
#        1.00000000e+00]])
# This gives us the I matrix
XTX.dot(XTX_inv).round(1)
# Output:
# array([[ 1.,  0.,  0.,  0.],
#        [ 0.,  1.,  0.,  0.],
#        [ 0.,  0.,  1.,  0.],
#        [ 0.,  0., -0.,  1.]])

```

Now we can implement the formula to get the full w vector.

```

# w_full contains all the weights
w_full = XTX_inv.dot(X.T).dot(y)
w_full
# Output: array([ 3.00092529e+04, -2.27839691e+01, -2.57690874])

```

From that vector w_full we can extract w₀ and all the other weights.

```

w0 = w_full[0]
w = w_full[1:]
w0, w
# Output: (30009.25292276666, array([-22.78396914, -257.690874]))

```

Now we can implement the function train_linear_regression, that takes the feature matrix X and the target variable y and returns w₀ and the vector w.

```

def train_linear_regression(X, y):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    XTX = X.T.dot(X)
    XTX_inv = np.linalg.inv(XTX)
    w_full = XTX_inv.dot(X.T).dot(y)

    return w_full[0], w_full[1:]

```

Let's test this newly implemented function with some simple examples:

```
X = [
    [148, 24, 1385],
    [132, 25, 2031],
    [453, 11, 86],
    [158, 24, 185],
    [172, 25, 201],
    [413, 11, 83],
    [38, 54, 185],
    [142, 25, 431],
    [453, 31, 86],
]
X = np.array(X)
y = [10000, 20000, 15000, 25000, 10000, 20000, 15000, 25000,
train_linear_regression(X, y)
# Output: (30009.25292276666, array([-22.78396914, -257.690814]))
```



Peter



21. September 2023



ML-Zoomcamp, Regression



Gram matrix, Linear regression, ML Zoomcamp

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

knowMLedge.com,

ML Zoomcamp 2023 – Machine Learning for Regression – Part 7

👤 Peter ⏰ 21. September 2023 📁 ML-Zoomcamp, Regression 🏷️ ML Zoomcamp



1. [Car price baseline model](#)
 1. [Value extraction](#)
 2. [Missing values](#)
 3. [Plotting model performance](#)

Car price baseline model

This article is about building a baseline model for price prediction of a car. Here we'll use the implemented code from the last article to build the model. First we start with a simple model while we're using only **numerical columns**.

The next code snippet shows how to extract all numerical columns. By the way you can also use it to extract categorical columns.

```

df_train.dtypes

# Output:
# make          object
# model         object
# year          int64
# engine_fuel_type   object
# engine_hp      float64
# engine_cylinders  float64
# transmission_type  object
# driven_wheels   object
# number_of_doors  float64
# market_category  object
# vehicle_size    object
# vehicle_style    object
# highway_mpg      int64
# city_mpg        int64
# popularity       int64
# dtype: object

df_train.columns
# Output:
# Index(['make', 'model', 'year', 'engine_fuel_type', 'engine_hp',
#        'engine_cylinders', 'transmission_type', 'driven_wheels',
#        'number_of_doors', 'market_category', 'vehicle_size',
#        'highway_mpg', 'city_mpg', 'popularity'],
#       dtype='object')

```

We choose the columns **engine_hp**, **engine_cylinders**, **highway_mpg**, **city_mpg**, and **popularity** for our base model.

```

base = ['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_mpg']
df_train[base].head()

```

	engine_hp	engine_cylinders	highway_mpg	city_mpg	popularity
0	310.0	8.0	18	13	1851
1	170.0	4.0	32	24	640
2	165.0	6.0	15	13	549
3	150.0	4.0	39	28	873
4	510.0	8.0	19	13	258

7150 rows × 5 columns

Value extraction

We need to extract the values to use them in training.

```

X_train = df_train[base].values
X_train
# Output:
#array([[ 310.,    8.,   18.,   13., 1851.],
#       [ 170.,    4.,   32.,   24.,  640.],
#       [ 165.,    6.,   15.,   13.,  549.],
#       ...
#       [ 342.,    8.,   24.,   17.,  454.],
#       [ 170.,    4.,   28.,   23., 2009.],
#       [ 160.,    6.,   19.,   14.,  586.]])

```

Missing values

Missing values are generally not good for our model. Therefore, you should always check whether such values are present.

```

df_train[base].isnull().sum()
# Output:
# engine_hp      42
# engine_cylinders 17
# highway_mpg     0
# city_mpg        0
# popularity       0
# dtype: int64

```

As you can see there are two columns that have missing values. The easiest thing we can do is fill them with zeros. But notice filling it with 0 makes the model ignore this feature, because:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2$$

if $x_{i1} = 0$ then the last equation simplifies to

$$g(x_i) = w_0 + 0 + x_{i2}w_2$$

But 0 is not always the best way to deal with missing values, because that means there is an observation of a car with 0 cylinders or 0 horse powers. And a car without cylinders or 0 horse powers does not make much sense at this point. For the current example this procedure is sufficient for us.

```

df_train[base].fillna(0).isnull().sum()
# Output:
# engine_hp      0
# engine_cylinders 0
# highway_mpg     0
# city_mpg        0
# popularity       0
# dtype: int64

```

As you can see in the last snippet, the missing values have disappeared. However, now we need to apply this change in the DataFrame.

```

X_train = df_train[base].fillna(0).values
X_train
# Output:
# array([[ 310.,    8.,   18.,   13., 1851.],
#        [ 170.,    4.,   32.,   24.,  640.],
#        [ 165.,    6.,   15.,   13.,  549.],
#        ...
#        [ 342.,    8.,   24.,   17., 454.],
#        [ 170.,    4.,   28.,   23., 2009.],
#        [ 160.,    6.,   19.,   14.,  586.]])
y_train
# Output:
# array([10.40262514, 10.06032035,  7.60140233, ..., 10.9218,
#       9.91100919,  8.10892416])

```

Now we can train our model using the `train_linear_regression` function that we've implemented in the last article. The function return the value for w_0 and and array for vector w .

```

w0, w = train_linear_regression(X_train, y_train)
w0, w
# Output:
# (7.471835414587793,
#  array([ 9.30959186e-03, -1.19533938e-01,  4.68925224e-02, -1
#         -1.01631104e-05]))

```

We can use this two variables to apply the model to our training dataset to see how well the model has learned the training data. We do not want the model to simply memorize the data, but to recognize the correlations. Later, we'll also apply the model to unseen data so that we can eliminate memorization.

```

y_pred = w0 + X_train.dot(w)
y_pred
# Output:
# array([10.07931663,  9.79812647,  8.84104849, ..., 10.62739,
#       9.60798726,  8.97035041])

```

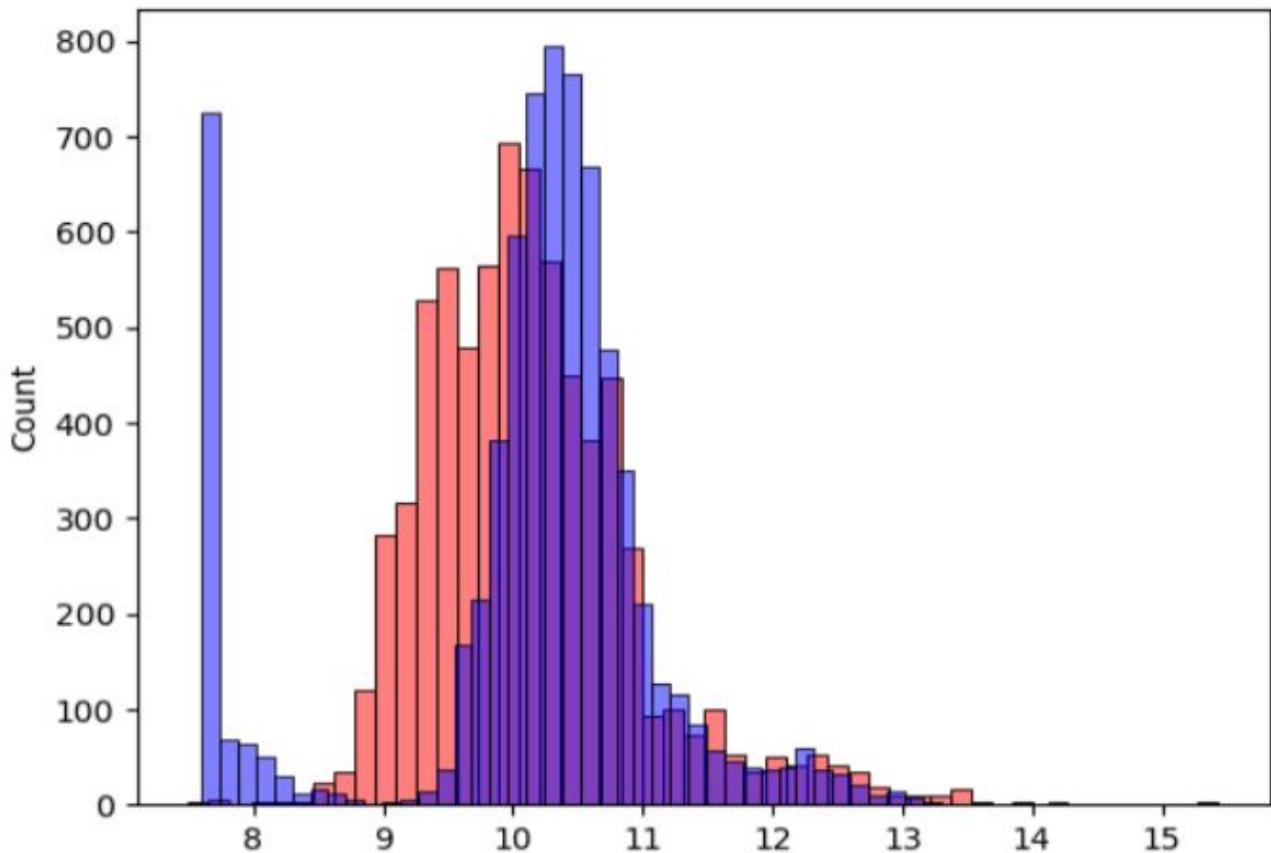
Plotting model performance

Three snippets before, you can see the actual values for y . The last snippet shows the predicted values for y . You could now go through these two lists manually and compare them. A better and also visual possibility is provided by Seaborn. The next snippet shows how to output these two lists accordingly.

```

# alpha changes the transparency of the bars
# bins specifies the number of bars
sns.histplot(y_pred, color='red', alpha=0.5, bins=50)
sns.histplot(y_train, color='blue', alpha=0.5, bins=50)

```



You see from this plot that the model is not ideal but it's better to have an objective way of saying that the model is good or not good. When we start improving the model, we also want to ensure that we really improving it. The next article is about an objective way to evaluate the performance of a regression model.

[Peter](#) [21. September 2023](#) [ML-Zoomcamp, Regression](#) [ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Machine Learning for Regression – Part 8

👤 Peter ⏰ 22. September 2023 📂 ML-Zoomcamp, Regression
🏷️ [ML Zoomcamp](#), [RMSE](#)



1. [Root Mean Squared Error – RMSE](#)
2. [Validating the model](#)

This part is about RMSE as an objective way to evaluate the model performance. In the first part of this article RMSE is introduced and in the second part RMSE is used to evaluate our model on unseen data.

Root Mean Squared Error – RMSE

We have the following variables, so we can calculate the RMSE.

- $g(x_i)$ – prediction for x_i (observation i)
- y_i – actual value
- m – number of different observations
- $\rightarrow g(x_i) - y_i$ is the difference between the prediction and the actual value

$$\text{RMSE} = \sqrt[m]{\frac{1}{m} \sum_{i=1}^m (g(x_i) - y_i)^2}$$

I t's the average of the squared difference between prediction and actual value

First let's look at this with an simplified example. First step is to calculate the difference between the prediction and the actual values.

y_pred	10	9	11	...	10
y_train	9	9	10.5	...	11.5
y_pred - y_train	1	0	0.5	...	-1.5

Difference between the prediction and the actual value

Then we need to square this difference to get the squared error.

square the difference: $(g(x_i) - y_i)^2$	1	0	0.25	...	2.25	SQUARED ERROR
Squared error						

Then we divide the squared error by number of observations to get the mean squared error. Lastly we can calculate the root mean squared error and we're done.

average	$(1 + 0 + 0.25 + 2.25) / 4 = 0.875$	MEAN SQUARED ERROR
root	$\sqrt{0.875} = 0.93$	ROOT MEAN SQUARED ERROR
RMSE		

We can implement the RMSE in code. This could look like:

```
def rmse(y, y_pred):
    se = (y - y_pred) ** 2
    mse = se.mean()
    return np.sqrt(mse)
```

In the last article we used Seaborn to visualize the performance but now we have an objective metric for the evaluation.

```
rmse(y_train, y_pred)
# Output: 0.7464137917148924
```

Validating the model

Evaluating the model performance on the training data does not really give a good indication of the real model performance. Since we don't know how well the model can apply the learned knowledge to unseen data. So what we want to do now after training the model g on our training dataset, we want to apply it on the validation dataset to see how it performs on unseen data. We use RMSE for validating the performance.

```

base = ['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_
X_train = df_train[base].fillna(0).values
w0, w = train_linear_regression(X_train, y_train)
y_pred = w0 + X_train.dot(w)

```

Next we implement the prepare_X function. The idea here to provide the same way of preparing the dataset regardless of whether it's train set, validation set, or test set.

```

def prepare_X(df):
    df_num = df[base]
    df_num = df_num.fillna(0)
    # extracting the Numpy array
    X = df_num.values
    return X

```

In the next section we change this function to improve it.

Now we can use this function when we prepare data for the training and for the validation as well. In the training part we only use training dataset to train the model. In the validation part we prepare the validation dataset the same way like before and apply the model. Lastly we compute the rmse.

```

# Training part:
X_train = prepare_X(df_train)
w0, w = train_linear_regression(X_train, y_train)

# Validation part:
X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)

# Evaluation part:
rmse(y_val, y_pred)
# Output: 0.7328022115111966

```

When we compare the RMSE from training with the value from validation (0.746 vs. 0.733) we see that the model performs similarly well on the seen and unseen data. That is what we have hoped for.

 [Peter](#)  [22. September 2023](#)  [ML-Zoomcamp, Regression](#)
 [ML Zoomcamp, RMSE](#)

Leave a comment

SIMPLE FEATURE ENGINEERING

Now that we have validated our model, let's see how to improve it.
In df_train we have the variable 'year' an important parameter. What we can do is create the variable 'age' to add it to X. Thus, we can change df_prepare_X(df) in the following way

```
def prepare_X(df):
    df = df.copy()           ← Not to change the
    df['age'] = 2017 - df.year
    features = base + ['age']
    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X
```

rmse (y-val, y-pred)

from 0.76 to 0.51, a big improvement

ML Zoomcamp 2023 – Machine Learning for Regression – Part 10

👤 Peter ⏰ 23. September 2023 📁 ML-Zoomcamp, Regression 🏷️ ML Zoomcamp



Categorical variables

Categorical variables are variables that are categories (typically strings)
Here: make, model, engine_fuel_type, transmission_type, driven_wheels,
market_category, vehicle_size, vehicle_style But, there is one value that looks like
numerical variable, but it isn't.
number_of_doors is not really a numerical number.

```
df_train.dtypes
# Output:
# make                         object
# model                        object
# year                          int64
# engine_fuel_type              object
# engine_hp                     float64
# engine_cylinders              float64
# transmission_type             object
# driven_wheels                 object
# number_of_doors                float64
# market_category               object
# vehicle_size                  object
# vehicle_style                 object
# highway_mpg                   int64
# city_mpg                      int64
# popularity                     int64
# dtype: object
```

```

df_train.number_of_doors
# Output:
# 0      4.0
# 1      4.0
# 2      3.0
# 3      4.0
# 4      4.0
#
# ...
# 7145   4.0
# 7146   2.0
# 7147   4.0
# 7148   4.0
# 7149   2.0
# Name: number_of_doors, Length: 7150, dtype: float64

df_train.number_of_doors == 2
# Output:
# 0      False
# 1      False
# 2      False
# 3      False
# 4      False
#
# ...
# 7145   False
# 7146   True
# 7147   False
# 7148   False
# 7149   True
# Name: number_of_doors, Length: 7150, dtype: bool

```

Typical way of encoding such categorical variables is that we represent it with a bunch of binary columns – so called one-hot encoding. For each value we have a different column.

Num of doors	num_doors_2	num_doors_3	num_doors_4
2	1	0	0
3	0	1	0
4	0	0	1
2	1	0	0

one-hot encoding for feature Num of Doors

We can imitate this encoding by turning the booleans from the last snippet into integers (1 and 0) and creating a new variable for each number of doors.

Because we are checking if condition is True / False

```

df_train['num_doors_2'] = (df_train.number_of_doors == 2).astype('int')
df_train['num_doors_3'] = (df_train.number_of_doors == 3).astype('int')
df_train['num_doors_4'] = (df_train.number_of_doors == 4).astype('int')

```

But we can do this easier with string replacement.

```

'num_doors_%s' % 4
# Output: 'num_doors_4'

# With that replacement we can write a loop
for v in [2, 3, 4]:
    df_train['num_doors_%s' % v] = (df_train.number_of_doors == v).astype('int')

# We delete this because we'll use another solution
for v in [2, 3, 4]:
    del df_train['num_doors_%s' % v]

```

This structure requires the %s snippet.
That's why after using %4 it becomes
asType('int')

Let's use this string replacement method in our prepare_X function.

```

def prepare_X(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age') ← Make copy and append to list
    # extracting the Numpy array
    X = df_num.values
    return X

prepare_X(df_train)
# Output:
# array([[310.,  8.,  18., ...,
#        [170.,  4.,  32., ...,
#        [165.,  6.,  15., ...,
#        [342.,  8.,  24., ...,
#        [170.,  4.,  28., ...,
#        [160.,  6.,  19., ...
#           0.,  0.,  1.],
#           0.,  0.,  1.],
#           0.,  1.,  0.],
#           0.,  0.,  1.],
#           0.,  0.,  1.],
#           1.,  0.,  0.]])

```

copy → To mess up the original

When you look at the output of the last snippet you see at the end of each list there are three new items – one for each number of doors (2, 3, 4). Now we can check if the model performance has improved with the new features.

```

X_train = prepare_X(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)

rmse(y_val, y_pred)
# Output: 0.5139733981046036

```

We see in contrast to the last training with rmse of 0.5153662333982238 there is only a slight improvement, almost negligible so the number of doors feature is not that useful.

Maybe the ‘Make’ information is more useful. (*The brand variable*)

```
df.make.unique()  
# Output: 48  
  
df.make  
# Output:  
# 0          bmw  
# 1          bmw  
# 2          bmw  
# 3          bmw  
# 4          bmw  
# ...  
# 11909      acura  
# 11910      acura  
# 11911      acura  
# 11912      acura  
# 11913      lincoln  
# Name: make, Length: 11914, dtype: object
```

There are 48 unique values in the ‘Make’ column. That could be too much. Let’s look at the most popular ones.

```
df.make.value_counts().head()  
# Output:  
# chevrolet    1123  
# ford         881  
# volkswagen   809  
# toyota        746  
# dodge         626  
# Name: make, dtype: int64  
  
# If we want to get the actual values, we use the index property  
df.make.value_counts().head().index  
# Wrap it in a usual Python list  
makes = list(df.make.value_counts().head().index)  
makes  
# Output: ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']
```

We can now adapt again our prepare_X function to add the new feature.

```

def prepare_X(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    for v in [2, 3, 4]:
        df['num_doors_%s' % v] = (df.number_of_doors == v).astype('int')
        features.append('num_doors_%s' % v)

    for v in makes:
        df['make_%s' % v] = (df.make == v).astype('int')
        features.append('make_%s' % v)

    df_num = df[features]
    df_num = df_num.fillna(0)
    # extracting the Numpy array
    X = df_num.values
    return X

```

Now we can use our new prepare_X function and train and validate again.

```

X_train = prepare_X(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)

rmse(y_val, y_pred)
# Output: 0.5058837299788781

```

The model performance has once again improved somewhat. How about adding all the other categorical variables now? This should improve the performance even more, right? Let's try.

```

categorical_variables = [
    'make', 'engine_fuel_type', 'transmission_type', 'driven_
    'market_category', 'vehicle_size', 'vehicle_style'
]

# The dictionary category will contain for each of the categories
# the top 5 most common ones
categories = {}

for c in categorical_variables:
    categories[c] = list(df[c].value_counts().head().index)

categories
# Output:
# {'make': ['chevrolet', 'ford', 'volkswagen', 'toyota', 'do
# 'engine_fuel_type': ['regular_unleaded',
# 'premium_unleaded_(required)'],
# 'premium_unleaded_(recommended)',
# 'flex-fuel_(unleaded/e85)',
# 'diesel'],
# 'transmission_type': ['automatic',
# 'manual',
# 'automated_manual',
# 'direct_drive',
# 'unknown'],
# 'driven_wheels': ['front_wheel_drive',
# 'rear_wheel_drive',
# 'all_wheel_drive',
# 'four_wheel_drive'],
# 'market_category': ['crossover',
# 'flex_fuel',
# 'luxury',
# 'luxury,performance',
# 'hatchback'],
# 'vehicle_size': ['compact', 'midsize', 'large'],
# 'vehicle_style': ['sedan',
# '4dr_suv',
# 'coupe',
# 'convertible',
# '4dr_hatchback']}

```

The next snippet shows how to implement the new features to our prepare_X function. This time we need two loops as described inline.

```

def prepare_X(df):
    # this is good way to do, otherwise while using df you'll
    # what is mostly not wanted
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    for v in [2, 3, 4]:
        df['num_doors_%s' % v] = (df.number_of_doors == v).a
        features.append('num_doors_%s' % v)

    # First loop is for each key of the dictionary categories:
    # Second loop is for each value inside the categories
    # For each of this values we create a new column.
    for c, values in categories.items():
        for v in values:
            df['%s_%s' % (c, v)] = (df[c] == v).astype('int')
            features.append('%s_%s' % (c, v))

    df_num = df[features]
    df_num = df_num.fillna(0)
    # extracting the Numpy array
    X = df_num.values
    return X

```

Annotations:

- Receiving a dictionary from the function.
- Pair of keys and values (values) for each element of the dictionary.
- Values or values per category.
- This means we have a dictionary (key: list of lists).
- For c, v in categories.items(): is like saying for ready to work with both the keys and lists/elements → for v in values operates on each element of the lists of values per key. This syntax is standard way to do that.

Now we can train the model again and apply it to the validation data to see what is the model performance.

```

X_train = prepare_X(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)

rmse(y_val, y_pred)
# Output: 292.5054633101075

```

This time the model performance is very bad. As you can see the RMSE (292.505) is very large. So something went wrong. In the next article we'll see why that has happened and how to fix it.

 [Peter](#)  [23. September 2023](#)  [ML-Zoomcamp, Regression](#)  [ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Machine Learning for Regression – Part 11

👤 Peter ⏰ 23. September 2023 📁 ML-Zoomcamp, Regression
🏷️ ML Zoomcamp, Regularization



Regularization

The topic for this part is **regularization** as a way to solve the problem of duplicated columns in our data. Remember the formula for normal equation is:

$$w = (X^T X)^{-1} * X^T y$$

The problem what we have is connected with the first part $(X^T X)^{-1}$. We need to take an inverse of the GRAM matrix. Sometimes this inverse doesn't exist. This happens when there are duplicate features in X. Let's take an easy example.

Because there would be the linear combination of another one

Can be a data cleaning issue

Tiny differences makes instead w_0 and w_i values go huge

```

# You see here 2nd and 3rd columns are identical
X = [
    [4, 4, 4],
    [3, 5, 5],
    [5, 1, 1],
    [5, 4, 4],
    [7, 5, 5],
    [4, 5, 5]
]

X = np.array(X)
X
# Output:
# array([[4, 4, 4],
#        [3, 5, 5],
#        [5, 1, 1],
#        [5, 4, 4],
#        [7, 5, 5],
#        [4, 5, 5]])

XTX = X.T.dot(X)
XTX
# Output:
# array([[140, 111, 111],
#        [111, 108, 108],
#        [111, 108, 108]])

```

The output of the last snippet shows the XTX matrix. You see the 2nd and 3rd columns are the same. In this case the inverse doesn't exist.

Remember: In linear algebra we say that one column is a linear combination of other columns. That means it's possible to express the column number 3 with other columns of the matrix which is basically just a duplicate of column 2.

Therefore the next code snipped raises an error "Singular matrix".

```

np.linalg.inv(XTX)
# Output: LinAlgError: Singular matrix

```

The code from the last article didn't raise that error, so the inverse of that gram matrix exists. But the reason for the very big value for rmse is that our data is not very clean.

Let's go back to our last example but this time similar X as before with a few noise.

```

X = [
    [4, 4, 4],
    [3, 5, 5],
    [5, 1, 1],
    [5, 4, 4],
    [7, 5, 5],
    [4, 5, 5.0000001],
]

X = np.array(X)
y = [1, 2, 3, 1, 2, 3]

XTX = X.T.dot(X)
XTX
# Output:
# array([[140.        , 111.        , 111.0000004],
#        [111.        , 108.        , 108.0000005],
#        [111.0000004, 108.0000005, 108.000001 ]])

XTX_inv = np.linalg.inv(XTX)
XTX_inv
# Output:
# array([[ 3.93617174e-02, -1.76703046e+05,  1.76703004e+05],
#        [-1.76703046e+05,  4.02107113e+13, -4.02107110e+13],
#        [ 1.76703004e+05, -4.02107110e+13,  4.02107106e+13]])

```

As we can see from this example, a little noise is enough to make the two columns no longer identical. This leads to the fact that the calculation of the gram matrix no longer throws an error. Now we can calculate vector w again.

```

w = XTX_inv.dot(X.T).dot(y)
w
# Output: array([ 2.85838502e-01, -5.04106388e+06,  5.0410642e+06])

```

The first element (that's the unique feature) of w looks ok, but the second and the third elements (that are the duplicates with noise) are very big numbers. That's why we have duplicates in our feature matrix. The noise leads to the fact that no more error is thrown. Nevertheless, the model performs poorly due to the duplicates. What we can do to fix the problem is to add a small number (called **alpha**) to the diagonal of XTX. Let's demonstrate this on an easier example of XTX.

```

# Adding a small number to the diagonal
# helps to control. So the numbers of w become smaller
XTX = [
    [1.0001, 2, 2],
    [2, 1.0001, 1.0000001],
    [2, 1.0000001, 1.0001]
]

XTX = np.array(XTX)
np.linalg.inv(XTX)
# Output:
# array([[-3.33366691e-01,  3.33350007e-01,  3.33350007e-01],
#        [ 3.33350007e-01,  5.00492166e+03, -5.00508835e+03],
#        [ 3.33350007e-01, -5.00508835e+03,  5.00492166e+03]])

```

The larger the number alpha adding to the diagonal, the more we have these weights under control. The reason why this works this way is, that this decrease the likelihood that these two columns are just copies of each other.

```
XTX = [
    [1.01, 2, 2],
    [2, 1.01, 1.0000001],
    [2, 1.0000001, 1.01]
]

XTX = np.array(XTX)
np.linalg.inv(XTX)
# Output:
# array([[-0.33668908,  0.33501399,  0.33501399],
#        [ 0.33501399,  49.91590897, -50.08509104],
#        [ 0.33501399, -50.08509104,  49.91590897]])
```

Let's implement this.

```
XTX = [
    [1, 2, 2],
    [2, 1, 1.0000001],
    [2, 1.0000001, 1]
]

XTX = np.array(XTX)
XTX
# Output:
# array([[1.          , 2.          , 2.          ],
#        [2.          , 1.          , 1.0000001],
#        [2.          , 1.0000001, 1.          ]])
```

Remember there was the eye function to get an Identity matrix. Maybe we can use this...

```
np.eye(3)

# When adding XTX to this matrix, it adds one on the diagonal
XTX + np.eye(3)

# We can multiply this eye by a small number
XTX = XTX + 0.01*np.eye(3)
XTX
# Output:
# array([[2.13       , 2.          , 2.          ],
#        [2.          , 2.13       , 1.0000001],
#        [2.          , 1.0000001, 2.13       ]])

#XTX = XTX + 0.1*np.eye(3)
#XTX

#XTX = XTX + 1*np.eye(3)
#XTX
```

```
np.linalg.inv(X.TX)
# Output:
# array([[-2.34791133,  1.50026279,  1.50026279],
#        [ 1.50026279, -0.35641202, -1.24136785],
#        [ 1.50026279, -1.24136785, -0.35641202]])
```

Solving this problem is called regularization and means in this case controlling. We're controlling the weights that they don't grow too much. Alpha = 0.01 is a parameter, and the larger this parameter the larger the numbers on the diagonal. And the larger this numbers on the diagonal the smaller the values in the inverse XTX matrix.

This leads us to reimplementing the train_linear_regression function again.

```
# reg = regularized
# parameter r = short for regularization
def train_linear_regression_reg(X, y, r=0.001):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    X.TX = X.T.dot(X)
    X.TX = X.TX + r*np.eye(X.TX.shape[0])

    X.TX_inv = np.linalg.inv(X.TX)
    w_full = X.TX_inv.dot(X.T).dot(y)

    return w_full[0], w_full[1:]
```

To test the effect of regularization, we need to re-train our model and apply it to the validation data. Then we can calculate the rmse.

```
X_train = prepare_X(df_train)
w0, w = train_linear_regression_reg(X_train, y_train, r=0.01)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)

rmse(y_val, y_pred)
# Output: 0.45685446091134857
```

This is the best result we had before, but we don't know that there is no better one. To find a good value for r is the topic of the next article.

👤 Peter ⏰ 23. September 2023 📁 ML-Zoomcamp, Regression
🏷️ ML Zoomcamp, Regularization

Leave a comment

ML Zoomcamp 2023 – Machine Learning for Regression – Part 12

👤 Peter ⏰ 24. September 2023 📁 ML-Zoomcamp, Regression 🏷️ ML Zoomcamp



1. [Tuning the model](#)
2. [Using the model](#)
 1. [Combining datasets](#)
 2. [Resetting index](#)
 3. [Getting feature matrix X](#)
 4. [Train the final model](#)
 5. [Applying model to test data](#)
 6. [Using the model](#)
 1. [Feature Extraction](#)
 2. [Predicting the price](#)

Tuning the model

The topic for this article is finding the best regularization parameter for our linear regression model. We realized that the parameter r affects the quality of our model and now we try to find the best value for this r .

```

for r in [0.0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10]:
    X_train = prepare_X(df_train)
    w0, w = train_linear_regression_reg(X_train, y_train, r=r)

    X_val = prepare_X(df_val)
    y_pred = w0 + X_val.dot(w)

    score = rmse(y_val, y_pred)

    print("reg parameter: ",r, "bias term: ",w0, "rmse: ",score)

# Output:
# reg parameter: 0.0 bias term: 2.6643718859809136e+16 rmse: 266.599
# reg parameter: 1e-05 bias term: 6.099552653959844 rmse: 0.460
# reg parameter: 0.0001 bias term: 6.8929434420779865 rmse: 0.4108
# reg parameter: 0.001 bias term: 6.900647539490208 rmse: 0.4608
# reg parameter: 0.01 bias term: 6.885494975398419 rmse: 0.4608
# reg parameter: 0.1 bias term: 6.7419125296313265 rmse: 0.45
# reg parameter: 1 bias term: 5.908895080537622 rmse: 0.45
# reg parameter: 10 bias term: 4.234139685166065 rmse: 0.4

```

What you see here is using r=0 makes the bias term huge and the rmse score aswell.
0.001 could be a good parameter for r.

```

r = 0.001
X_train = prepare_X(df_train)
w0, w = train_linear_regression_reg(X_train, y_train, r=r)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)

score = rmse(y_val, y_pred)

print("rmse: ",score)
# Output: rmse: 0.4568807317131709

```

Using the model

In the last article we found the best parameter for the linear regression and in this lesson we'll train the model again and use it. What we did so far is, we trained our model on training dataset and applied the best model on validation dataset. To check the model performance we calculated the RMSE.

What we want to do now is to train our final model on both training dataset and validation dataset. We call this FULL TRAIN. After that we make the final evaluation on the test dataset to make sure that our model works fine and check what is the value for RMSE. It shouldn't be too different from what we saw on the validation dataset.

Combining datasets

First step to do is getting our data. So we need to combine df_train and df_val into one dataset. We can use Pandas concat() function that takes a list of dataframes and

concatenates them together.

```
df_full_train = pd.concat([df_train, df_val])
```

We also need to concatenate y_train and y_val to get y_full_train. This time we use the concatenate function of NumPy library.

```
y_full_train = np.concatenate([y_train, y_val])
y_full_train
# Output: array([10.40262514, 10.06032035, 7.60140233, ...,
   10.3663092 , 10.37101938])
```

Resetting index

When combining two dataframes it can happen that the index is not sequential. Here you can use an already known function and reset the index.

```
df_full_train = df_full_train.reset_index(drop=True)
```

Getting feature matrix X

Now we have again a coherent dataset for training and we can prepare it for the usage as we did before. The prepare_X() function still works fine.

```
X_full_train = prepare_X(df_full_train)
X_full_train
# Output:
# array([[310.,    8.,   18., ...,    0.,    0.,    0.],
       [170.,    4.,   32., ...,    0.,    0.,    0.],
       [165.,    6.,   15., ...,    0.,    0.,    0.],
       ...,
       [295.,    8.,   19., ...,    0.,    0.,    0.],
       [283.,    6.,   25., ...,    0.,    0.,    0.],
       [182.,    4.,   32., ...,    0.,    0.,    0.]])
```

Train the final model

Next step is to train the final model on the combined dataset. We're using the new train_linear_regression_reg() function to get the value for w_0 and the vector w.

```
w0, w = train_linear_regression_reg(X_full_train, y_full_train,
w0, w
# Output:
# (6.78312259616272,
# array([ 1.46535912e-03,  1.06314995e-01, -3.46567859e-02,
       -5.29907921e-05, -1.00251712e-01, -1.12652502e+00, -
       -9.91483092e-01, -2.98403708e-02,  1.71081845e-01,
       -1.21180790e-01, -1.07844220e-01, -4.76163384e-01,
       -3.21665624e-01, -5.42738768e-01,  4.29611924e-02,
       9.88804373e-01,  1.20687968e+00,  2.79900016e+00,
       1.78916282e+00,  1.63554122e+00,  1.73947001e+00,
       -8.10459522e-02,  3.06406210e-02, -3.41386920e-02, -
       3.75251434e-02,  2.33450124e+00,  2.22007067e+00,
       4.14224325e-02,  4.99735446e-02,  2.45833450e-01,
       -1.16344690e-01]))
```

Applying model to test data

Now is the great moment for the final model. It must pass the final test. For this purpose we use test data, which are again prepared with the prepare_X() function. Then the model is applied to the test data and the RMSE can be calculated.

```
X_test = prepare_X(df_test)
y_pred = w0 + X_test.dot(w)

score = rmse(y_test, y_pred)

print("rmse: ", score)
# Output: rmse:  0.5094518818513973
```

RMSE_test = 0.5094518818513973 is not so far away from RMSE_val = 0.4568807317131709. That means the model generalizes quite well and it didn't get this score by chance. Now we have our final model and we can use it. The way we want to use it is to predict the price of an (unseen) car – unseen means here that the model hasn't seen this car during training.

Using the model

Using the model means:

1. Extracting all the features (getting feature vector of the car)
2. Applying our final model to this feature vector & predicting the price

Feature Extraction

For this step we can take any car from our test dataset and pretend it's a new car. Let's just take one car.

```

df_test.iloc[20]
# Output:
# make          saab
# model         9-3_griffin
# year          2012
# engine_fuel_type premium_unleaded_(recommended)
# engine_hp      220.0
# engine_cylinders   4.0
# transmission_type manual
# driven_wheels    all_wheel_drive
# number_of_doors   4.0
# market_category   luxury
# vehicle_size      compact
# vehicle_style      wagon
# highway_mpg        30
# city_mpg           20
# popularity          376
# Name: 20, dtype: object

```

Usually the way we do it is that we don't get a dataframe here. But it could be a Python dictionary with all the information about the car. In real life you can imagine a website or an app, where people enter all the values. Then the website sends the request with all the information (as dictionary) to the model. The model replies back with the predicted price.

For this example we turn this data of our car into a dictionary.

```

car = df_test.iloc[20].to_dict()
car
# Output:
# {'make': 'saab',
#  'model': '9-3_griffin',
#  'year': 2012,
#  'engine_fuel_type': 'premium_unleaded_(recommended)',
#  'engine_hp': 220.0,
#  'engine_cylinders': 4.0,
#  'transmission_type': 'manual',
#  'driven_wheels': 'all_wheel_drive',
#  'number_of_doors': 4.0,
#  'market_category': 'luxury',
#  'vehicle_size': 'compact',
#  'vehicle_style': 'wagon',
#  'highway_mpg': 30,
#  'city_mpg': 20,
#  'popularity': 376}

```

The car is our request and now remember the prepare_X function expects a dataframe, so we need to create a dataframe with a single row for our request.

```

df_small = pd.DataFrame([car])
df_small

```

	make	model	year	engine_fuel_type	engine_hp	engi
0	saab	9-3_griffin	2012	premium_unleaded_(recommended)	220	

DataFrame of our requested car

We can use this single row DataFrame as input for the prepare_X() function to get the feature matrix. In this case our feature matrix is a feature vector.

```
X_small = prepare_X(df_small)
X_small
# Output:
# array([[220.,  4., 30., 20., 376.,  5.,  0.,  0.,  1.,
#        0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,
#        0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,
#        1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

Predicting the price

The final step is to apply the final model to our requested car (feature vector) and predict the price.

```
y_pred = w0 + X_small.dot(w)
# Don't need an array but it's first (and only) item
y_pred = y_pred[0]
y_pred
# Output: 9.954435569951846
```

9.95 is still not the price in \$. To get the real price we need to undo the logarithm.

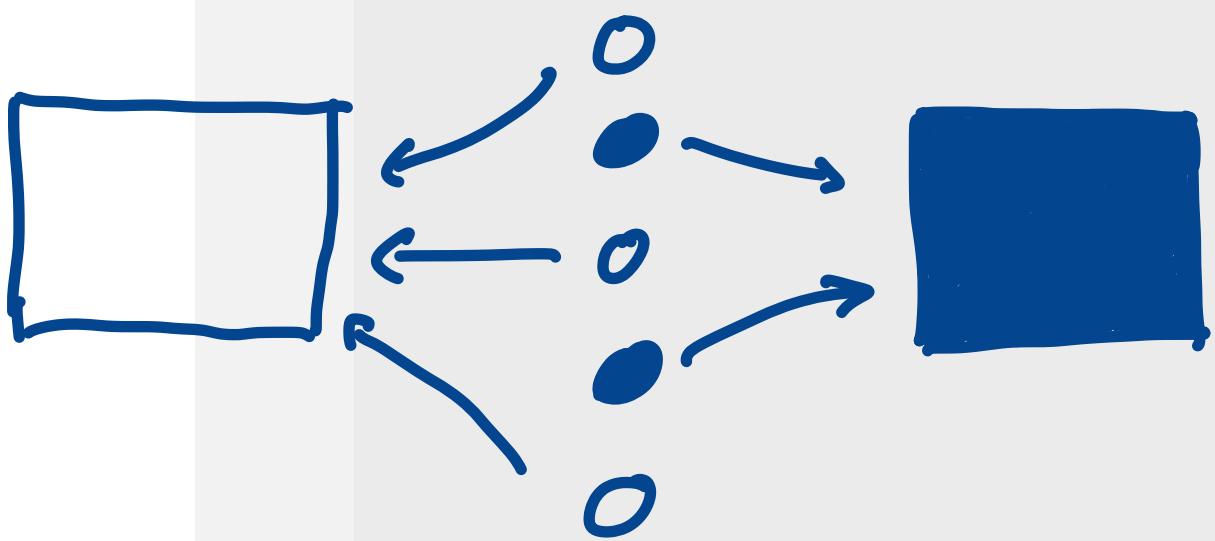
```
np.expm1(y_pred)
# Output: 21044.363844829495
```

After undoing the logarithm we get the price in \$. So we think that a car with these characteristics should cost \$21,044.36.

Lastly to get an evaluation about model performance let's compare the predicted price to the actual price of this requested car.

```
np.expm1(y_test[20])
# Output: 34975.0
```

CHURN CLASSIFICATION



SUMMARY

We want to build a model to predict whether a customer will churn (abandon your company). We want to identify customers about to churn to send them a promotional email. We have their info in a dataset, which we cleaned and split for validation framework using sklearn.

We did EDA using risk ratio. We use mutual information to understand the relevance of different variables when it comes to churn. We also use correlation for numerical values. Then, one-hot encoding allows us to encode categorical features, i.e., to turn variables such as "color" into binary data.

We then discuss logistic regression, used for binary classification and saw that it's just like linear regression, but adding a sigmoid variable, a function that turns a score into probability

We create a model that we train with scikitLearn. We also discuss how we can interpret the model using a small training set as example with less features. We finally train the full model

There's a nice summary of the code and the steps in the first "chapter" of the next session

ML Zoomcamp 2023 – Machine Learning for Classification– Part 1

👤 Peter ⏰ 25. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, ML Zoomcamp



Churn prediction project

In this project, we're focusing on churn prediction for a telecommunications company. Consider a telecommunications company with a diverse customer base. Some of these customers are satisfied with the services they receive, while others are not. Those who are dissatisfied are contemplating terminating their contracts and switching to another service provider.

What we want to do?

We want to identify that clients that are willing to churn (= stop using this services) How could we do this? We give every customer a score between 0 and 1 that tells how likely this customer is going to leave. Then we can send promotional e-mails to this customers – giving some discount to keep this customer.

The way we approach this with Machine Learning is **Binary Classification**.

Remember the formula for one single observation: $g(x_i) \sim y_i$

In this particular case we're dealing with our target variable y_i (tells us whether customer x_i left the company or not) and x_i that is a feature vector that is describing the i th customer.

Output of our model is y_i that is a value between {0, 1} which is the likelihood that this particular customer number i is going to churn.

1 means **positive** example, customer left the company, 0 means **negative** example,

customer didn't leave the company. So in general **1** means that there is something **present** (the effect we want to predict is present) and **0** means that it's **not present**.

The way we do this is as follows: Let's say we take the customers from last month, and for each customer, we know who actually left. For those customers, we can set the target label to 1; for the rest, we can set it to 0 because they stayed. All the target labels together become 'y.' The information we have about the customers, including demographics, where they live, how much they pay, what kind of services they have, and the type of contract, becomes 'X.'

So, we want to gather information to determine which factors lead to churn. That's the main idea of this project. We aim to build a model using historical data, including existing customers that we can score.

For that, we are using a dataset from Kaggle titled '[**Telco Customer Churn – Focused Customer Retention Programs**](#)' The column we intend to predict is the 'Churn' column.

👤 [Peter](#) ⏰ [25. September 2023](#) 📁 [Classification, ML-Zoomcamp](#)
🏷️ [Classification, ML Zoomcamp](#)

Leave a comment

Write a comment...

[Comment](#)

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Machine Learning for Classification– Part 2

👤 Peter ⏰ 26. September 2023 📄 Classification, ML-Zoomcamp
🏷️ Classification, Data preparation, ML Zoomcamp



Data preparation

The topics that we cover in this section are:

1. [Data preparation](#)
 1. [Downloading the data](#)
 2. [Reading the data](#)
 3. [Making column names and values look uniform](#)
 4. [Verify that all the columns have been read correctly](#)
 5. [Checking if the churn variable needs any preparation](#)

Downloading the data

First, we import all the necessary packages. Then, we can download our CSV file using the ‘wget’ command. When using Jupyter Notebook, it’s important to note that ‘!’ indicates the execution of a shell command, and the ‘\$’ symbol, as seen in ‘\$data,’ is the way to reference data within this shell command.

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt

data = "https://..."
!wget $data -O data-week-3.csv
```

Reading the data

When reading the data this time, we can see that there are a lot of columns – 21 in total. The three dots ‘...’ in the header row indicate that not all columns are shown, making it a bit more challenging to get a complete overview.

```
| df = pd.read_csv('data-week-3.csv')  
df.head()
```

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	...
0	7590-VHVEG	Female	0	Yes	No	1	N	...
1	5575-GNVDE	Male	0	No	No	34	Y	...
2	3668-QPYBK	Male	0	No	No	2	Y	...
3	7795-CFOCW	Male	0	No	No	45	N	...
4	9237-HQITU	Female	0	No	No	2	Y	...

5 rows × 21 columns

To display all of them simultaneously, we can use the transpose function. This will switch the rows to become columns and the columns to become rows.

```
| df.head().T
```

	0	1	2	3	4
customerID	7590-VHVEG	5575-GNVDE	3668-QPYBK	7795-CFOCW	9237-HQITU
gender	Female	Male	Male	Male	Female
SeniorCitizen	0	0	0	0	0
Partner	Yes	No	No	No	No
Dependents	No	No	No	No	No
tenure	1	34	2	45	2
PhoneService	No	Yes	Yes	No	Yes
MultipleLines	No phone service	No	No	No phone service	No
InternetService	DSL	DSL	DSL	DSL	Fiber optic
OnlineSecurity	No	Yes	Yes	Yes	No
OnlineBackup	Yes	No	Yes	No	No
DeviceProtection	No	Yes	No	Yes	No
TechSupport	No	No	No	Yes	No
StreamingTV	No	No	No	No	No
StreamingMovies	No	No	No	No	No
Contract	Month-to-month	One year	Month-to-month	One year	Month-to-month
PaperlessBilling	Yes	No	Yes	No	Yes
PaymentMethod	Electronic check	Mailed check	Mailed check	Bank transfer (automatic)	Electronic check
MonthlyCharges	29.85	56.95	53.85	42.3	70.7
TotalCharges	29.85	1889.5	108.15	1840.75	151.65
Churn	No	No	Yes	No	Yes

Making column names and values look uniform

What we can observe here is that the data is not consistent or uniform. So, we'll follow a similar approach as we did before in the car price prediction project.

```
df.columns = df.columns.str.lower().str.replace(' ', '_')
categorical_columns = list(df.dtypes[df.dtypes == 'object'].index)
for c in categorical_columns:
    df[c] = df[c].str.lower().str.replace(' ', '_')
df.head().T
```

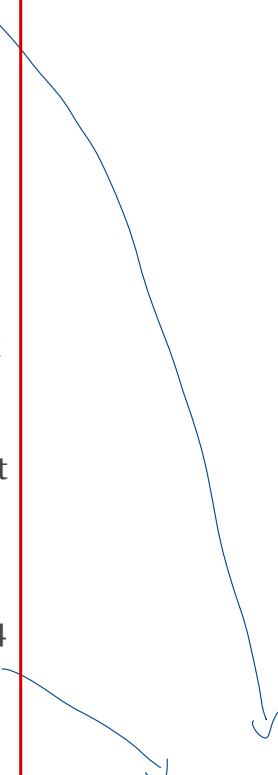
	0	1	2	3
customerid	7590-vhveg	5575-gnvde	3668-qpybk	7795-ct
gender	female	male	male	ma
seniorcitizen	0	0	0	c
partner	yes	no	no	n
dependents	no	no	no	n
tenure	1	34	2	4
phoneservice	no	yes	yes	n
multiplelines	no_phone_service	no	no	no_phone
internetservice	dsl	dsl	dsl	d
onlinesecurity	no	yes	yes	ye
onlinebackup	yes	no	yes	n
deviceprotection	no	yes	no	ye
techsupport	no	no	no	ye
streamingtv	no	no	no	n
streamingmovies	no	no	no	n
contract	month-to-month	one_year	month-to-month	one_
paperlessbilling	yes	no	yes	n
paymentmethod	electronic_check	mailed_check	mailed_check	bank_transfer
monthlycharges	29.85	56.95	53.85	42
totalcharges	29.85	1889.5	108.15	1840
churn	no	no	yes	n

Verify that all the columns have been read correctly

Now, all the column names are uniform, and all spaces have been replaced with underscores. Next, let's examine the data types we have.

df.dtypes

```
# Output:  
# customerid object  
# gender object  
# seniorcitizen int64  
# partner object  
# dependents object  
# tenure int64  
# phoneservice object  
# multiplelines object  
# internetservice object  
# onlinesecurity object  
# onlinebackup object  
# deviceprotection object  
# techsupport object  
# streamingtv object  
# streamingmovies object  
# contract object  
# paperlessbilling object  
# paymentmethod object  
# monthlycharges float64  
# totalcharges object  
# churn object  
# dtype: object
```



```
# Output:  
# 0 29.85  
# 1 1889.5  
# 2 108.15  
# 3 1840.75  
# 4 151.65  
# ...  
# 7038 1990.5  
# 7039 7362.9  
# 7040 346.45  
# 7041 306.6  
# 7042 6844.5  
# Name: totalcharges, Length: 7043, dtype: object
```

Indeed, ‘totalcharges’ appears to be numeric in nature. It seems that some of the values are not in numeric format. Let’s attempt to convert them into numbers...

```
pd.to_numeric(df.totalcharges)
```



```
# Output: ValueError: Unable to parse string “_” at position 488
```

This means that the column doesn’t only contain numbers but also includes characters like

'_,' which are not numeric. The reason for this is that in this dataset, a space represents 'not available,' indicating missing data. We have replaced these spaces with underscores. We can utilize Pandas for this task. If Pandas encounters a string that it cannot parse as a number, we can instruct it to ignore the string and replace it with 'NaN' (Not a number). This can be achieved by using a parameter called 'errors,' and setting it to 'coerce,' which means Pandas will disregard such errors.

```
# tc means total charges  
tc = pd.to_numeric(df.totalcharges, errors='coerce')  
tc  
# Output:  
# 0 29.85  
# 1 1889.50  
# 2 108.15  
# 3 1840.75  
# 4 151.65  
# ...  
# 7038 1990.50  
# 7039 7362.90  
# 7040 346.45  
# 7041 306.60  
# 7042 6844.50  
# Name: totalcharges, Length: 7043, dtype: float64
```

```
tc.isnull().sum()  
# Output: 11  
  
df[tc.isnull()]  
# it's not easy to see so we need a different one  
df[tc.isnull()][['customerid', 'totalcharges']]
```

	customerid	totalcharges
488	4472-lvygi	—
753	3115-czmzd	—
936	5709-lvoeq	—
1082	4367-nuyao	—
1340	1371-dwpaz	—
3331	7644-omvmy	—
3826	3213-vvolg	—
4380	2520-sgtta	—
5218	2923-arzlg	—
6670	4075-wkniu	—
6754	2775-sefee	—

Now let's take off those values. We can fill those values with 0. However, it's important to keep in mind that 0 is not always the ideal choice, but in many practical scenarios, it's an acceptable one.

```
df.totalcharges = pd.to_numeric(df.totalcharges, errors='coerce')
df.totalcharges = df.totalcharges.fillna(0)
df.totalcharges.isnull().sum()
# Output: 0
```

Checking if the churn variable needs any preparation

The last thing we wanted to do is look at the churn variable. We see that the values are either "yes" or "no." In machine learning, for classification, we are interested in numerical values. In this case, we can represent "churn" as 1 and "not churn" as 0.

```
df.churn.head()
# Output:
# 0 no
# 1 no
# 2 yes
# 3 no
# 4 yes
# Name: churn, dtype: object
```

To obtain these numerical values, we can replace “yes” with 1 and “no” with 0, as demonstrated in the following snippet.

```
(df.churn == 'yes').astype(int).head()  
# Output:  
# 0 0  
# 1 0  
# 2 1  
# 3 0  
# 4 1  
# Name: churn, dtype: int64
```

Now, let's apply this transformation to all values in this column and update it in the `df.churn` column.

```
df.churn = (df.churn == 'yes').astype(int)  
df.churn  
# Output:  
# 0 0  
# 1 0  
# 2 1  
# 3 0  
# 4 1  
# ..  
# 7038 0  
# 7039 0  
# 7040 0  
# 7041 1  
# 7042 0  
# Name: churn, Length: 7043, dtype: int64
```

👤 [Peter](#) ⏰ [26. September 2023](#) 📁 [Classification, ML-Zoomcamp](#)
🏷️ [Classification, Data preparation, ML Zoomcamp](#)

Leave a comment

Write a comment...

ML Zoomcamp 2023 – Machine Learning for Classification– Part 3

👤 Peter ⏰ 27. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, ML Zoomcamp, Train Test Split



Setting up the validation framework

Perform the train/validation/test split with Scikit-Learn

60	20	20
----	----	----

You can utilize the `train_test_split` function from the `sklearn.model_selection` package to automate the splitting of your data into training, validation, and test sets. Before you can use it, make sure to import it first as follows:

```
from sklearn.model_selection import train_test_split  
  
# to see the documentation  
train_test_split?
```

The `train_test_split` function divides the dataframe into two parts, with 80% for the full train set and 20% for the test set. We use `random_state=1` to ensure that the results are reproducible.

```
df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=1)  
len(df_full_train), len(df_test)  
  
# Output: (5634, 1409)
```

To obtain three sets (train, validation, and test), we should perform the split again with the

full train set. However, this time, we want to allocate 60% for the train set and 20% for the validation set. To calculate the validation set size, we can't use `test_size=0.2` as before because we are dealing with 80% of the data. Instead, we need to determine 20% of 80%, which is equivalent to 25%. Therefore, for the validation set, we should use `test_size=0.25`.

```
df_train, df_val = train_test_split(df_full_train, test_size=0.25,  
len(df_train), len(df_val))  
# Output: (4225, 1409)  
random_state = 1)
```

We see that validation set and test set have the same size now.

```
len(df_train), len(df_val), len(df_test)  
# Output: (4225, 1409, 1409)
```

Two code snippets earlier, we used the `train_test_split` function to create shuffled datasets. However, this resulted in the indexes within the records being shuffled rather than continuous. To reset the indices, you can use the `reset_index` function with the `drop=True` parameter to drop the old index column. Here's how you can do it:

```
df_full_train = df_full_train.reset_index(drop=True)  
df_train = df_train.reset_index(drop=True)  
df_val = df_val.reset_index(drop=True)  
df_test = df_test.reset_index(drop=True)
```

Now, we need to extract our target variable, which is 'y' (churn). Here's how it looks for the four datasets:

```
y_full_train = df_full_train.churn.values  
y_train = df_train.churn.values  
y_val = df_val.churn.values  
y_test = df_test.churn.values
```

Certainly, to prevent accidental use of the "churn" variable when building a model, we should remove it from our dataframes. Here's how you can remove the "churn" column from each of the four datasets:

```
del df_full_train['churn']  
del df_train['churn']  
del df_val['churn']  
del df_test['churn']
```

After performing these operations, the "churn" variable will be removed from your datasets, and you can proceed with building your model without the risk of accidentally using it.

ML Zoomcamp 2023 – Machine Learning for Classification– Part 4

👤 Peter ⏰ 27. September 2023 📁 Classification, ML-Zoomcamp

🏷️ [Classification](#), [EDA](#), [Exploratory Data Analysis](#), [Missing Values](#), [ML Zoomcamp](#)



EDA – Exploratory Data Analysis

The topics that we cover in this section are:

1. [EDA – Exploratory Data Analysis](#)
 1. [Checking missing values](#)
 2. [Looking at the target variable \(churn\)](#)
 3. [Looking at numerical and categorical variables](#)

Checking missing values

The following snippet indicates that the dataset ‘df_full_train’ contains no missing values:

$$df_full_Train = df_full_Train.reset_index (drop=True)$$

```
df_full_train.isnull().sum()  
  
# Output:  
# customerid      0  
# gender          0  
# seniorcitizen   0  
# partner         0  
# dependents      0  
# tenure          0  
# phoneservice    0  
# multiplelines   0  
# internetservice 0  
# onlinesecurity  0  
# onlinebackup    0  
# deviceprotection 0  
# techsupport     0  
# streamingtv     0  
# streamingmovies 0  
# contract        0  
# paperlessbilling 0  
# paymentmethod   0  
# monthlycharges  0  
# totalcharges    0  
# churn           0  
# dtype: int64
```

Looking at the target variable (churn)

```
df_full_train.churn  
  
# Output:  
# 0      0  
# 1      1  
# 2      0  
# 3      0  
# 4      0  
#  
# 5629   ..  
# 5630   0  
# 5631   1  
# 5632   1  
# 5633   0  
# Name: churn, Length: 5634, dtype: int64
```

First what we can check is the distribution of our target variable ‘churn’. How many customers are churning and how many are not-churning.

```
df_full_train.churn.value_counts()  
  
# Output:  
# 0    4113  
# 1    1521  
# Name: churn, dtype: int64
```

There is information about a total of 5634 customers. Among these, 1521 are dissatisfied

customers (churning), while the remaining 4113 are satisfied customers (not churning). Understanding the distribution of your target variable is an essential step in any data analysis or modeling task, as it provides valuable insights into the data's class balance, which can influence modeling decisions and evaluation metrics.

Using the `value_counts` function with the `normalize=True` parameter provides the churn rate, which represents the proportion of churning customers relative to the total number of customers. In our case, we've calculated that the churn rate is almost 27%.

```
df_full_train.churn.value_counts(normalize=True)  
# Output:  
# 0    0.730032  
# 1    0.269968  
# Name: churn, dtype: float64
```

There is another way to calculate the global churn rate; we can simply use the `mean()` function, as shown in the next snippet.

```
global_churn_rate = df_full_train.churn.mean()  
round(global_churn_rate, 2)  
# Output: 0.27
```

We realize that it's the same value as the churn rate. Let's explore why this works here.

The formula for the mean is given by:

$$\text{mean} = \frac{1}{n} \sum x$$

In this case, where $x \in \{0,1\}$, it simplifies to:

$$\text{mean} = (\text{number of ones}) / n$$

And that is indeed the churn rate. This principle holds true for all binary datasets, because the mean of binary values corresponds directly to the proportion of ones in the dataset, which is essentially the churn rate in this context.

Looking at numerical and categorical variables

To identify the categorical and numerical variables in your dataset, you can use the `dtypes` function as mentioned earlier. Here's how you can use it in general:

```

numerical_vars = df.select_dtypes(include=['int64', 'float64'])
categorical_vars = df.select_dtypes(include=['object'])

print("Numerical Variables:")
print(numerical_vars.columns)

print("\nCategorical Variables:")
print(categorical_vars.columns)

```

This code will help you separate and display the numerical and categorical variables in your dataset, making it easier to understand the data's structure and plan your data analysis accordingly. Let's look at our dataframe.

`df_full_train.dtypes`

```

# Output:
# customerid          object
# gender               object
# seniorcitizen        int64
# partner              object
# dependents           object
# tenure               int64
# phoneservice         object
# multiplelines        object
# internetservice      object
# onlinesecurity       object
# onlinebackup          object
# deviceprotection     object
# techsupport           object
# streamingtv          object
# streamingmovies       object
# contract              object
# paperlessbilling     object
# paymentmethod         object
# monthlycharges        float64
# totalcharges          float64
# churn                int64
# dtype: object

```

Notice that this one
isn't really an integer,
if it is a categorical
value.
↓
Only three variables are
numerical

As we know, there are three numerical variables: tenure, monthly charges, and total charges. Let's define numerical and categorical columns.

```
df_full_train.columns

# Output:
# Index(['customerid', 'gender', 'seniorcitizen', 'partner', 'dependents',
#        'tenure', 'phoneservice', 'multiplelines', 'internetservice',
#        'onlinesecurity', 'onlinebackup', 'deviceprotection', 'techsupport',
#        'streamingtv', 'streamingmovies', 'contract', 'paperlessbill',
#        'paymentmethod', 'monthlycharges', 'totalcharges', 'churn'],
#       dtype='object')

numerical = ['tenure', 'monthlycharges', 'totalcharges']

# Removing 'customerid', 'tenure', 'monthlycharges', 'totalcharges'
categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
               'phoneservice', 'multiplelines', 'internetservice',
               'onlinesecurity', 'onlinebackup', 'deviceprotection', 'techsupport',
               'streamingtv', 'streamingmovies', 'contract', 'paperlessbill',
               'paymentmethod']
```

To determine the number of unique values for all the categorical variables, we can use the `nunique()` function.

```
df_full_train[categorical].nunique()
```

```
# Output:  
# gender  
# seniorcitizen  
# partner  
# dependents  
# phoneservice  
# multiplelines  
# internetservice  
# onlinesecurity  
# onlinebackup  
# deviceprotection  
# techsupport  
# streamingtv  
# streamingmovies  
# contract  
# paperlessbilling  
# paymentmethod  
# dtype: int64
```

2
2
2
2
2
3
3
3
3

Means
are just
 \bar{x}
Types of
values

👤 Peter 🕒 27. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, EDA, Exploratory Data Analysis, Missing Values, ML Zoomcamp

Leave a comment

ML Zoomcamp 2023 – Machine Learning for Classification– Part 5

👤 Peter ⏰ 28. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, Feature Importance, ML Zoomcamp, Risk Ratio



Feature importance: Churn rate and risk ratio

Feature importance analysis is a part of exploratory data analysis (EDA) and involves identifying which features affect our target variable.

- Churn rate
- Risk ratio
- Mutual information – later

Churn rate

Last time, we examined the global churn rate. Now, we are focusing on the churn rate within different groups. For example, we are interested in determining the churn rate for the gender group.

```
| # Selecting the subset of female customers
| df_full_train[df_full_train.gender == 'female']
```

	customerid	gender	seniorcitizen	partner	dependants	tenure	phoi
1	6261-rcvns	female	0	no	no	42	
5	4765-oxppd	female	0	yes	yes	9	
9	1732-vhubq	female	1	yes	yes	47	
11	7017-vfuly	female	0	yes	no	2	
13	1374-dmzui	female	1	no	no	4	
...
5618	8065-ykxkd	female	0	no	no	10	
5619	5627-tvbpp	female	0	no	yes	35	
5626	3262-eidhv	female	0	yes	yes	72	
5627	7446-sfaoa	female	0	yes	no	37	
5633	5840-nvdeg	female	0	yes	yes	16	

2796 rows × 21 columns

The following snippet displays the value of the global churn rate. In comparison to that value, we can also calculate the churn rates for the female and male groups. We observe that the female churn rate is slightly higher than the global rate, while the male churn rate is slightly lower than the global rate. This suggests that women are somewhat more likely to churn.

```

global_churn = df_full_train.churn.mean()
global_churn
# Output: 0.26996805111821087

churn_female = df_full_train[df_full_train.gender == 'female'].churn.
churn_female
# Output: 0.27682403433476394      ≈ 27.7%  
mean()

global_churn - churn_female
# Output: -0.006855983216553063

churn_male = df_full_train[df_full_train.gender == 'male'].churn.mean()
churn_male
# Output: 0.2632135306553911

global_churn - churn_male
# Output: 0.006754520462819769

```

Let's check the churn rate of another group (with partner vs. without partner).

```

df_full_train.partner.value_counts()

# Output:
# no    2932
# yes   2702
# Name: partner, dtype: int64

```

When examining this group, we notice that customers with partners are significantly less likely to churn. The churn rate for this group is approximately 20.5%, contrasting with the global churn rate of almost 27%. On the other hand, customers without partners have a much higher churn rate compared to the global rate, standing at 33% as opposed to 27%.

```

global_churn = df_full_train.churn.mean()
global_churn
# Output: 0.26996805111821087

churn_partner = df_full_train[df_full_train.partner == 'yes'].churn.mean()
churn_partner
# Output: 0.20503330866025166

global_churn - churn_partner
# Output: 0.06493474245795922

churn_no_partner = df_full_train[df_full_train.partner == 'no'].churn.
churn_no_partner
# Output: 0.3298090040927694  
mean()

global_churn - churn_no_partner
# Output: -0.05984095297455855

```

This observation suggests that the partner variable may be more influential for predicting churn than the gender variable. *since gender results were similar to global churn, while those from partner weren't*

To assess feature importance, if $\text{global_group} > 0$, That group is less likely to churn, and vice versa. This can also be determined by The risk ratio (see below), where if $\text{group} > 1$, Then That group is more likely to churn. The ratio is nice because it tells you how much higher or lower the risk is. If the result is, e.g., 1.22, Then 22% higher chance to churn.

Risk ratio

In the context of machine learning and classification, the “risk ratio” typically refers to a statistical measure used to assess the likelihood or probability of a certain event occurring in one group compared to another. It’s a useful concept in various fields, including healthcare, finance, and customer churn analysis.

In the specific context of churn rate, the risk ratio can help you understand the relative risk of churn (i.e., customers leaving) for different groups or segments within your dataset. It can provide insights into which features or factors are associated with a higher or lower risk of churn.

Here’s a simplified explanation of how risk ratio works in the context of churn rate:

1. **Definition of Risk Ratio:** The risk ratio (also known as the relative risk) is defined as the probability of an event occurring in one group divided by the probability of the same event occurring in another group. In the case of churn rate, you’re typically comparing two groups: one group that exhibits a certain characteristic or behavior (e.g., customer has churned) and another group that does not exhibit that characteristic (e.g., customer hasn’t churned).
2. **Interpretation:** A risk ratio greater than 1 suggests that the event (churn in this case) is more likely in the first group compared to the second group. A risk ratio less than 1 suggests the event is less likely in the first group. A risk ratio equal to 1 means there is no difference in risk between the two groups.
3. **Application:** We can use risk ratios to assess the impact of different features or interventions on churn rate. For example, we might calculate the risk ratio of churn for customers who received a promotional offer versus those who did not. If the risk ratio is significantly greater than 1, it indicates that the promotional offer had a positive impact on reducing churn.
4. **Statistical Significance:** It’s important to also consider statistical significance when interpreting risk ratios. Statistical tests such as chi-squared tests or confidence intervals can help determine if the observed differences in churn rates are statistically significant.

So the risk ratio is a valuable tool for assessing the impact of different factors or features on churn rate in classification tasks. It helps you quantify and compare the relative risk of churn between different groups, providing insights that can inform decision-making and strategies for reducing churn.

Let’s compare the risk ratio for churning between people with partners and those without partners.

```
churn_no_partner / global_churn  
# Output: 1.2216593879412643  
  
churn_partner / global_churn  
# Output: 0.7594724924338315
```

This demonstrates that the churn rate for people without partners is 22% higher, whereas for people with partners, it is 24% lower than the global churn rate.

Let's take the data and group it by gender, and for each variable within the gender group, let's calculate the average churn rate within that group and calculate the difference and risk. We can perform this analysis for all the variables, not just the gender variable.

The SQL query would look like:

```
SELECT
    gender,
    AVG(churn),
    AVG(churn) - global_churn AS diff,
    AVG(churn) / global_churn AS risk
FROM
    date
GROUP BY
    gender;
```

This in pandas looks like:

```
df_full_train.groupby('gender').churn.mean()

# Output:
# gender
# female    0.276824
# male      0.263214
# Name: churn, dtype: float64

# agg takes a list of different aggregations
df_full_train.groupby('gender').churn.agg(['mean', 'count'])
```

gender	mean	count
female	0.276824	2796
male	0.263214	2838

```
df_group = df_full_train.groupby('gender').churn.agg(['mean', 'count'])
df_group['diff'] = df_group['mean'] - global_churn
df_group['risk'] = df_group['mean'] / global_churn
df_group
```

gender	mean	count	diff	risk
female	0.276824	2796	0.006856	1.025396
male	0.263214	2838	-0.006755	0.974980

mean, count, diff, and risk for gender column

This table is interesting, but it only displays information for the gender groups. Now, let's extend this analysis to include all the categorical columns.

```

from IPython.display import display → I think it's to display
                                Tables, but not sure
                                double-check
for c in categorical:
    #print(c)
    df_group = df_full_train.groupby(c).churn.agg(['mean', 'count'])
    df_group['diff'] = df_group['mean'] - global_churn
    df_group['risk'] = df_group['mean'] / global_churn
    display(df_group)
    print()
    print()

```

GENDER	MEAN	COUNT	DIFF	RISK
FEMALE	0.276824	2796	0.006856	1.025396
MALE	0.263214	2838	-0.006755	0.974980
mean, count, diff, and risk for gender column				

seniorcitizen	mean	count	diff	risk
0	0.242270	4722	-0.027698	0.897403
1	0.413377	912	0.143409	1.531208
mean, count, diff, and risk for senior citizen column				

partner	mean	count	diff	risk
no	0.329809	2932	0.059841	1.221659
yes	0.205033	2702	-0.064935	0.759472
mean, count, diff, and risk for partner column				

dependents	mean	count	diff	risk
no	0.313760	3968	0.043792	1.162212
yes	0.165666	1666	-0.104302	0.613651
mean, count, diff, and risk for dependents column				

phoneservice	mean	count	diff	risk
no	0.241316	547	-0.028652	0.893870
yes	0.273049	5087	0.003081	1.011412
mean, count, diff, and risk for phone service column				

multiplelines	mean	count	diff	risk
no	0.257407	2700	-0.012561	0.953474
no_phone_service	0.241316	547	-0.028652	0.893870
yes	0.290742	2387	0.020773	1.076948
mean, count, diff, and risk for multiple lines column				

internetservice	mean	count	diff	risk
dsl	0.192347	1934	-0.077621	0.712482
fiber_optic	0.425171	2479	0.155203	1.574895
no	0.077805	1221	-0.192163	0.288201
mean, count, diff, and risk for internet service column				

onlinesecurity	mean	count	diff	risk
no	0.420921	2801	0.150953	1.559152
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.153226	1612	-0.116742	0.567570
mean, count, diff, and risk for online security column				

onlinebackup	mean	count	diff	risk
no	0.404323	2498	0.134355	1.497672
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.217232	1915	-0.052736	0.804660
mean, count, diff, and risk for onlinebackup column				

deviceprotection	mean	count	diff	risk
no	0.395875	2473	0.125907	1.466379
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.230412	1940	-0.039556	0.853480

mean, count, diff, and risk for device protection column

techsupport	mean	count	diff	risk
no	0.418914	2781	0.148946	1.551717
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.159926	1632	-0.110042	0.592390

mean, count, diff, and risk for tech support column

streamingtv	mean	count	diff	risk
no	0.342832	2246	0.072864	1.269897
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.302723	2167	0.032755	1.121328

mean, count, diff, and risk for streaming tv column

streamingmovies	mean	count	diff	risk
no	0.338906	2213	0.068938	1.255358
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.307273	2200	0.037305	1.138182

mean, count, diff, and risk for streaming movies column

contract	mean	count	diff	risk
month-to-month	0.431701	3104	0.161733	1.599082
one_year	0.120573	1186	-0.149395	0.446621
two_year	0.028274	1344	-0.241694	0.104730

mean, count, diff, and risk for contract column

paperlessbilling	mean	count	diff	risk
no	0.172071	2313	-0.097897	0.637375
yes	0.338151	3321	0.068183	1.252560

mean, count, diff, and risk for paperless billing column

paymentmethod	mean	count	diff	risk
bank_transfer_(automatic)	0.168171	1219	-0.101797	0.622928
credit_card_(automatic)	0.164339	1217	-0.105630	0.608733
electronic_check	0.455890	1893	0.185922	1.688682
mailed_check	0.193870	1305	-0.076098	0.718121

mean, count, diff, and risk for payment method column

Summary

This article has covered the difference and the risk ratio as two important tools for assessing feature importance.

Concerning the difference, we calculate it by subtracting the group's churn rate from the global churn rate. Here, we are primarily interested in significant differences, unlike in the gender case. Values for this difference smaller than 0 indicate a higher likelihood to churn, while values larger than 0 indicate a lower likelihood to churn.

As for the risk ratio, it is obtained by dividing the group's churn rate by the global churn rate. Values greater than 1 suggest a higher likelihood to churn, whereas values less than 1 suggest a lower likelihood to churn.

In essence, both difference and risk ratio convey similar information but in different ways, providing insights into the importance of features with respect to churn prediction.

We observe certain categories in which people tend to churn more or less frequently compared to the global average. These are the types of variables we are interested in and want to use in machine learning algorithms. While it's informative to see this for individual variables in each table, it would be valuable to have a measure that quantifies the overall importance of each variable.

To determine how we can assess whether the “contract” variable is less or more important than “streamingmovies,” we will proceed with the following steps.

• [Peter](#) • [28. September 2023](#) ■ [Classification, ML-Zoomcamp](#)
🏷️ [Classification, Feature Importance, ML Zoomcamp, Risk Ratio](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Machine Learning for Classification– Part 6

👤 Peter ⏰ 28. September 2023 📂 Classification, ML-Zoomcamp
🏷️ Classification, Feature Importance, ML Zoomcamp, Mutual Information



Feature importance: Mutual information

Indeed, the risk ratio provides valuable insights into the importance of different categorical variables, particularly when examining the likelihood of churn for each value within a variable. For example, when analyzing the “contract” variable with values like “month-to-month,” “one_year,” and “two_years,” we can observe that customers with a “month-to-month” contract are more likely to churn compared to those with a “two_years” contract. This suggests that the “contract” variable is likely to be an important factor in predicting churn. However, without a way to compare this importance with other variables, we may not have a clear understanding of its relative significance.

Mutual information, a concept from information theory, addresses this issue by quantifying how much we can learn about one variable when we know the value of another.

The higher the mutual information, the more information we gain about churn by observing the value of another variable.) In essence, it provides a means to measure the importance of categorical variables and their values in predicting churn, allowing us to compare their significance relative to one another.

$$I(X, Y) = D_{KL}(P_{(X,Y)} || P_X \otimes P_Y)$$

$$I(X, Y) = \sum_{j \in Y} \sum_{x \in X} p(x, y) p(x, y) \log \left(\frac{p(x, y)}{p_x(x) p_y(y)} \right)$$

Too complex, we just need the intuition - what this means

```

from sklearn.metrics import mutual_info_score

mutual_info_score(df_full_train.churn, df_full_train.contract)
# order is not important
#mutual_info_score(df_full_train.contract, df_full_train.churn)
# Output: 0.0983203874041556

mutual_info_score(df_full_train.churn, df_full_train.gender)
# Output: 0.0001174846211139946 Gender doesn't tell us much about churn

mutual_info_score(df_full_train.churn, df_full_train.partner)
# Output: 0.009967689095399745 Partner is more informative than gender but less than contract

```

Once more: The intuition here is how much we learn about churn by observing the value of the contract variable or any other variable and vice versa. We observe, for example, that the gender variable is not particularly informative.

At all we can learn about the relative importance of the features. What we can do now, we can apply this metric to all the categorical variables and see which one has the highest mutual information.

The `apply` function takes a function with one argument, but `mutual_info_score` requires two arguments. That's why we need to implement the `mutual_info_churn_score` function, which can be applied to the dataframe to compute mutual information scores column-wise.

mutual info score

```

def mutual_info_churn_score(series):
    return mutual_info_score(series, df_full_train.churn)

mi = df_full_train[categorical].apply(mutual_info_churn_score)
mi

# Output:
# gender          0.000117
# seniorcitizen  0.009410
# partner         0.009968
# dependents     0.012346
# phoneservice   0.000229
# multiplelines  0.000857
# internetservice 0.055868
# onlinesecurity 0.063085
# onlinebackup   0.046923
# deviceprotection 0.043453
# techsupport    0.061032
# streamingtv    0.031853
# streamingmovies 0.031581
# contract        0.098320
# paperlessbilling 0.017589
# paymentmethod   0.043210
# dtype: float64

```

To arrange this list in such a way that the most important variables come first, we can sort the variables based on their mutual information scores in descending order. This way, the most important variables will appear at the top of the list.

More relevant for churn

Less impact

Higher values above

```
mi.sort_values(ascending=False)
```

Variable	Mutual Information Value
# contract	0.098320
# onlinesecurity	0.063085
# techsupport	0.061032
# internetservice	0.055868
# onlinebackup	0.046923
# deviceprotection	0.043453
# paymentmethod	0.043210
# streamingtv	0.031853
# streamingmovies	0.031581
# paperlessbilling	0.017589
# dependents	0.012346
# partner	0.009968
# seniorcitizen	0.009410
# multiplelines	0.000857
# phoneservice	0.000229
# gender	0.000117
# dtype: float64	

Using this approach, we can gain a better understanding of which variables are highly informative for our analysis and which are less so.

• [Peter](#) • [28. September 2023](#) └ [Classification, ML-Zoomcamp](#)
 # [Classification, Feature Importance, ML Zoomcamp, Mutual Information](#)

Leave a comment

Write a comment...

[Comment](#)

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Machine Learning for Classification– Part 7

👤 Peter ⏰ 29. September 2023 📂 Classification, ML-Zoomcamp
🏷️ Classification, Correlation Coefficient, Feature Importance, ML Zoomcamp



Feature importance: Correlation

For measuring feature importance for numerical variables, one common approach is to use the correlation coefficient, specifically [Pearson's correlation coefficient](#). The Pearson correlation coefficient quantifies the degree of linear dependency between two numerical variables.

The correlation coefficient (often denoted as "r") has a range of -1 to 1:

- A negative correlation ($r = -1$) indicates a strong inverse relationship, where one variable increases as the other decreases.
- A positive correlation ($r = 1$) indicates a strong positive relationship, where both variables increase together.
- An r value close to 0 suggests a weak or no linear relationship between the variables.

The strength of the correlation is indicated by the absolute value of r:

- $0.0 < |r| < 0.2$: Low correlation
- $0.2 < |r| < 0.5$: Moderate correlation
- $0.6 < |r| < 1.0$: Strong correlation

You can calculate the Pearson correlation coefficient between your numerical variables and the target variable (churn) to assess their importance. Higher absolute values of r indicate a stronger linear relationship, which can be interpreted as higher feature importance.

To calculate the Pearson correlation coefficient in Python, you can use the `corr()`

function from pandas:

```
correlation = df[numerical_vars].corr()['churn']
```

This will give you a series of correlation values between each numerical variable and the churn variable, and you can sort them in descending order to identify the most important numerical features.

In the context of churn prediction:

- Positive Correlation: A positive correlation between a numerical variable (e.g., tenure) and churn means that as the numerical variable increases (e.g., longer tenure), the likelihood of churn (1) increases.
- Negative Correlation: A negative correlation between a numerical variable (e.g., tenure) and churn means that as the numerical variable increases (e.g., longer tenure), the likelihood of churn (1) decreases.
- Zero Correlation: A correlation close to zero suggests that there is no significant linear relationship between the numerical variable and churn.

You can calculate the Pearson correlation coefficient (r) between each numerical variable and the target variable (churn) to assess the feature importance of numerical variables. Higher absolute values of r indicate a stronger linear relationship, which can be interpreted as higher feature importance. This analysis helps you identify which numerical variables have the most impact on churn prediction.

```
| df_full_train[numerical]
```

	tenure	monthlycharges	totalcharges
0	12	19.70	258.35
1	42	73.90	3160.55
2	71	65.15	4681.75
3	71	85.45	6300.85
4	30	70.40	2044.75
...
5629	9	100.50	918.60
5630	60	19.95	1189.90
5631	28	105.70	2979.50
5632	2	54.40	114.10
5633	16	68.25	1114.85

5634 rows × 3 columns

```
df_full_train[numerical].corrwith(df_full_train.churn)

# Outlook:
# tenure           -0.351885
# monthlycharges   0.196805
# totalcharges     -0.196353
# dtype: float64
```

If you're primarily interested in the importance of numerical variables without considering the direction of the correlation, you can focus on the absolute values of the correlation coefficients ($|r|$). This approach allows you to rank numerical variables based on their overall impact on the target variable (churn) regardless of whether the relationship is positive or negative.

By sorting the absolute values of the correlation coefficients in descending order, you can identify which numerical variables have the greatest magnitude of correlation with churn and therefore contribute most significantly to predicting churn. This approach simplifies the interpretation by treating both positive and negative correlations as having the same importance.

```

df_full_train[numerical].corrwith(df_full_train.churn).abs()

# Output:
# tenure           0.351885
# monthlycharges  0.196805
# totalcharges    0.196353
# dtype: float64

```

It seems that tenure is the most important numerical variable. Let's look at some examples:

(Stayed in The company less than 2 months)

```

df_full_train[df_full_train.tenure <= 2].churn.mean()
# Output: 0.5953420669577875

```

```

df_full_train[df_full_train.tenure > 2].churn.mean()
# Output: 0.22478269658378816

```

```

df_full_train[(df_full_train.tenure > 2) & (df_full_train.tenure <=
# Output: 0.3994413407821229

```

```

df_full_train[df_full_train.tenure > 12].churn.mean()
# Output: 0.17634908339788277

```

Regarding the variable “tenure,” we can observe that the group of customers with the highest likelihood to churn consists of those with a tenure of less than or equal to 2. It has a churn rate of almost 60%. We also see that the longer people stay with the company, the less they churn.

Let's look at another example:

```

if_full_train[df_full_train.monthlycharges <= 20].churn.mean()
# Output: 0.08795411089866156

```

```

if_full_train[(df_full_train.monthlycharges > 20) & (df_full_train.m
# Output: 0.18340943683409436

```

```

if_full_train[df_full_train.monthlycharges > 50].churn.mean()
# Output: 0.32499341585462205

```

Concerning the variable “monthlycharges,” we can observe that the group of customers with the highest likelihood to churn belongs to the group with monthly charges greater than \$50. It has a churn rate of 32.5%.

 [Peter](#)  [29. September 2023](#)  [Classification, ML-Zoomcamp](#)
 [Classification, Correlation Coefficient, Feature Importance, ML Zoomcamp](#)

Leave a comment

ML Zoomcamp 2023 – Machine Learning for Classification– Part 8

👤 Peter

⌚ 29. September 2023

💻 Classification, ML-Zoomcamp

🏷️ Classification, ML Zoomcamp, One-hot Encoding



One-hot encoding

One-hot encoding is a technique used in machine learning to convert categorical (non-numeric) data into a numeric format that can be used by machine learning algorithms. It's particularly useful when working with algorithms that require numerical input, such as many classification and regression models. Scikit-Learn, a popular machine learning library in Python, provides convenient tools for performing one-hot encoding.

- **Problem:** Categorical data, such as “color” with categories like “red,” “green,” and “blue,” cannot be directly used as input for most machine learning algorithms because they require numerical data. One-hot encoding solves this problem by converting categorical data into binary vectors.
- **How It Works:** For each categorical feature, one-hot encoding creates a new binary (0 or 1) feature for each category within that feature. Each binary feature represents the presence or absence of a specific category. For example, in the “color” example, you'd create three binary features: “IsRed,” “IsGreen,” and “IsBlue.” If an observation belongs to the “red” category, the “IsRed” feature is set to 1, while “IsGreen” and “IsBlue” are set to 0.

Use Scikit-Learn to encode categorical features

Using the Pandas `to_dict` function with the `orient` parameter set to ‘records’ transforms the dataframe into a collection of dictionaries. In this format, each row or record is converted into a separate dictionary.

```
: turns each column into a dictionary --> but the result is not what  
df_train[['gender', 'contract']].iloc[:100].to_dict()
```

```
dicts = df_train[['gender', 'contract']].iloc[:100].to_dict(orient='records')  
dicts
```

Output:

```
[{'gender': 'female', 'contract': 'two_year'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'two_year'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'two_year'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'two_year'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'two_year'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'one_year'},  
 {'gender': 'male', 'contract': 'two_year'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'one_year'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'female', 'contract': 'two_year'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 . . .  
 {'gender': 'male', 'contract': 'one_year'},  
 {'gender': 'female', 'contract': 'month-to-month'},  
 {'gender': 'male', 'contract': 'month-to-month'},  
 {'gender': 'male', 'contract': 'one_year'},  
 {'gender': 'male', 'contract': 'month-to-month'}]
```

dictionary created ↗

Using DictVectorizer

First, we need to create a new instance of this class. Then, we train our DictVectorizer instance. This involves presenting the data to the DictVectorizer so that it can infer the column names and their corresponding values. Based on this information, it creates the one-hot encoding feature matrix. It's worth noting that the DictVectorizer is intelligent enough to detect numerical variables and exclude them from one-hot encoding, as they don't require such encoding.

```
| from sklearn.feature_extraction import DictVectorizer  
| dv = DictVectorizer()  
| dv.fit(dicts)
```

Then we need to transform the dictionaries.

```

dv.transform(dicts)
# Output:
# <10x4 sparse matrix of type '<class 'numpy.float64'>' 
#   with 20 stored elements in Compressed Sparse Row format>

```

When using the `transform` method as shown in the last snippet, it creates a sparse matrix by default. A sparse matrix is a memory-efficient way of encoding data when there are many zeros in the dataset. But we'll not use sparse matrix here.

In the code of the next snippet, the `DictVectorizer` returns a regular NumPy array where the first three columns represent the “contract” variable, and the last two columns represent the “gender” variable.

```

v = DictVectorizer(sparse=False)
v.fit(dicts)

v.get_feature_names_out()
Output:
array(['contract=month-to-month', 'contract=one_year',
       'contract=two_year', 'gender=female', 'gender=male'],

v.transform(dicts)
Output:
array([[0., 0., 1., 1., 0.],
       [1., 0., 0., 0., 1.],
       [1., 0., 0., 1., 0.],
       [1., 0., 0., 1., 0.],
       [0., 0., 1., 1., 0.],
       [1., 0., 0., 0., 1.],
       [1., 0., 0., 0., 1.],
       [1., 0., 0., 1., 0.],
       [0., 0., 1., 1., 0.],
       [1., 0., 0., 1., 0.],
       [0., 0., 1., 1., 0.],
       [1., 0., 0., 0., 1.],
       [0., 0., 1., 1., 0.],
       [1., 0., 0., 1., 0.],
       [1., 0., 0., 1., 0.],
       [1., 0., 0., 0., 1.],
       [0., 0., 1., 1., 0.],
       [1., 0., 0., 1., 0.],
       [1., 0., 0., 1., 0.],
       [0., 1., 0., 0., 1.],
       [0., 0., 1., 0., 1.],
       [1., 0., 0., 0., 1.],
       [0., 1., 0., 1., 0.],
       [1., 0., 0., 1., 0.],
       [0., 0., 1., 1., 0.],
       [1., 0., 0., 0., 1.]])

```

Note: If you apply DictVectorizer to a numerical variable, it leaves it as a numerical value without transforming it into binary.

Contract *Gender*

Let's bring it all together

```
train_dicts = df_train[categorical + numerical].to_dict(orient='record')
train_dicts[0]
Output:
{'gender': 'female',
 'seniorcitizen': 0,
 'partner': 'yes',
 'dependents': 'yes',
 'phoneservice': 'yes',
 'multiplelines': 'yes',
 'internetservice': 'fiber_optic',
 'onlinesecurity': 'yes',
 'onlinebackup': 'yes',
 'deviceprotection': 'yes',
 'techsupport': 'yes',
 'streamingtv': 'yes',
 'streamingmovies': 'yes',
 'contract': 'two_year',
 'paperlessbilling': 'yes',
 'paymentmethod': 'electronic_check',
 'tenure': 72,
 'monthlycharges': 115.5,
 'totalcharges': 8425.15}
```

↳ Take both categorical and numerical variables

```
dv = DictVectorizer(sparse=False)
dv.fit(train_dicts)
```

```
dv.get_feature_names_out()
```

Output:

```
array(['contract=month-to-month', 'contract=one_year',
       'contract=two_year', 'dependents=no', 'dependents=yes',
       'deviceprotection=no', 'deviceprotection=no_internet_service',
       'deviceprotection=yes', 'gender=female', 'gender=male',
       'internetservice=dsl', 'internetservice=fiber_optic',
       'internetservice=no', 'monthlycharges', 'multiplelines=no',
       'multiplelines=no_phone_service', 'multiplelines=yes',
       'onlinebackup=no', 'onlinebackup=no_internet_service',
       'onlinebackup=yes', 'onlinesecurity=no',
       'onlinesecurity=no_internet_service', 'onlinesecurity=yes',
       'paperlessbilling=no', 'paperlessbilling=yes', 'partner=no',
       'partner=yes', 'paymentmethod=bank_transfer_(automatic)',
       'paymentmethod=credit_card_(automatic)',
       'paymentmethod=electronic_check', 'paymentmethod=mailed_check',
       'phoneservice=no', 'phoneservice=yes', 'seniorcitizen',
       'streamingmovies=no', 'streamingmovies=no_internet_service',
       'streamingmovies=yes', 'streamingtv=no',
       'streamingtv=no_internet_service', 'streamingtv=yes',
       'techsupport=no', 'techsupport=no_internet_service',
       'techsupport=yes', 'tenure', 'totalcharges'], dtype=object)
```

Short version without long outputs:

```
from sklearn.feature_extraction import DictVectorizer
train_dicts = df_train[categorical + numerical].to_dict(orient='records')
dv = DictVectorizer(sparse=False)

dv.fit(train_dicts)
X_train = dv.transform(train_dicts)
instead of last two lines, you can also use
X_train = dv.fit_transform(train_dicts)

X_train.shape
Output: (4225, 45)
```

When dealing with validation data, we can reuse the same `DictVectorizer` instance that we created before. Instead of using the `fit` function followed by the `transform` function, we only need to apply the `transform` function to the validation data. This ensures that the transformation process applied to the validation data is consistent with the encoding used for the training data.

```
val_dicts = df_val[categorical + numerical].to_dict(orient='records')
X_val = dv.transform(val_dicts)
```

👤 Peter ⏰ 29. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, ML Zoomcamp, One-hot Encoding

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Machine Learning for Classification– Part 9

👤 Peter ⏰ 30. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, Linear regression, Logistic Regression, ML Zoomcamp, Sigmoid



Logistic Regression

Model $\rightarrow g(x_i) = y_i$ ← Target
Regression → Classification → Binary: $y_i \in \{0, 1\}$
(2 Types) → Multi-class

As mentioned earlier, classification problems can be categorized into binary problems and multi-class problems. Binary problems are the types of problems that logistic regression is typically used to solve.

In binary classification, the target variable y_i belongs to one of two classes: 0 or 1. These classes are often referred to as “negative” and “positive,” and they represent two mutually exclusive outcomes. In the context of churn prediction, “no churn” and “churn” are examples of binary classes. Similarly, in email classification, “no spam” and “spam” are also binary classes.

That means $g(x_i)$ outputs a number from 0 to 1 that we can treat as the probability of x_i belonging to the positive class.

Formula:

Linear regression: $g(x_i) = w_0 + w^T x_i \rightarrow$ outputs a number $-\infty..+\infty \in \mathbb{R}$

- x_0 – bias term
- w^T – weights vector
- x_i – features

Logistic regression does something similar as linear regression, but the output is between 0 and 1, instead $-\infty..+\infty$
Logistic regression: $g(x_i) = \text{SIGMOID}(w_0 + w^T x_i) \rightarrow$ outputs a number $0..1 \in \mathbb{R}$

$\text{sigmoid}(z) = 1 / (1 + \exp(-z))$ ↗ Function used for logistic regression
Probability output ↘ Score ↗ Basically you turn a score into probability

This function maps any real number z to the range of 0 to 1, making it suitable for modeling probabilities in logistic regression. We'll use this function to convert a score into a probability.

Let's see how to implement the sigmoid function and use it. We can create an array with 51 values between -7 and 7 using `np.linspace(-7, 7, 51)`. This is our z in the next snippet.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

z = np.linspace(-7, 7, 51) → Create a random set for example
```

Output:

```
array([-7.000000e+00, -6.720000e+00, -6.440000e+00, -6.160000e-
      -5.880000e+00, -5.600000e+00, -5.320000e+00, -5.040000e+00,
      -4.760000e+00, -4.480000e+00, -4.200000e+00, -3.920000e+00,
      -3.640000e+00, -3.360000e+00, -3.080000e+00, -2.800000e+00,
      -2.520000e+00, -2.240000e+00, -1.960000e+00, -1.680000e+00,
      -1.400000e+00, -1.120000e+00, -8.400000e-01, -5.600000e-01,
      -2.800000e-01, 8.8817842e-16, 2.800000e-01, 5.600000e-01,
      8.400000e-01, 1.120000e+00, 1.400000e+00, 1.680000e+00,
      1.960000e+00, 2.240000e+00, 2.520000e+00, 2.800000e+00,
      3.080000e+00, 3.360000e+00, 3.640000e+00, 3.920000e+00,
      4.200000e+00, 4.480000e+00, 4.760000e+00, 5.040000e+00,
      5.320000e+00, 5.600000e+00, 5.880000e+00, 6.160000e+00,
      6.440000e+00, 6.720000e+00, 7.000000e+00])
```

We can apply this sigmoid function to our array z ...

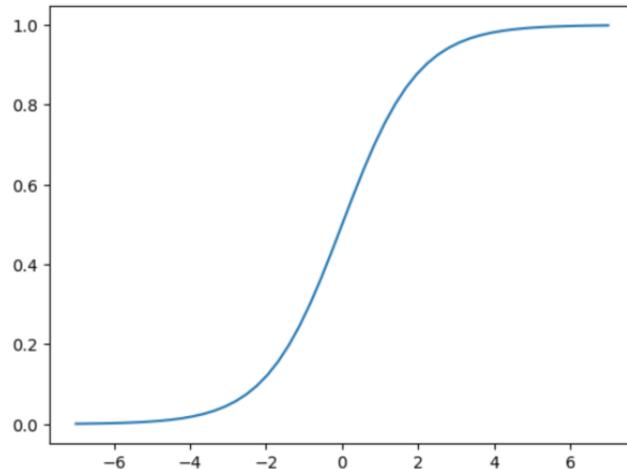
```
sigmoid(z)
```

Output:

```
array([9.11051194e-04, 1.20508423e-03, 1.59386223e-03, 2.10780106e-
      2.78699622e-03, 3.68423990e-03, 4.86893124e-03, 6.43210e-
      8.49286285e-03, 1.12064063e-02, 1.47740317e-02, 1.94550e-
      2.55807883e-02, 3.35692233e-02, 4.39398154e-02, 5.73240e-
      7.44679452e-02, 9.62155417e-02, 1.23467048e-01, 1.57090e-
      1.97816111e-01, 2.46011284e-01, 3.01534784e-01, 3.63540e-
      4.30453776e-01, 5.00000000e-01, 5.69546224e-01, 6.36450e-
      6.98465216e-01, 7.53988716e-01, 8.02183889e-01, 8.42900e-
      8.76532952e-01, 9.03784458e-01, 9.25532055e-01, 9.42670e-
      9.56060185e-01, 9.66430777e-01, 9.74419212e-01, 9.80540e-
      9.85225968e-01, 9.88793594e-01, 9.91507137e-01, 9.93560e-
      9.95131069e-01, 9.96315760e-01, 9.97213004e-01, 9.97890e-
      9.98406138e-01, 9.98794916e-01, 9.99088949e-01])
```

... but let's visualize how the graph of the sigmoid function looks.

```
| plt.plot(z, sigmoid(z))
```



At the end of this article, both implementations are presented for comparison. The first snippet demonstrates the familiar linear regression, while the second snippet illustrates logistic regression. It's evident that there is essentially only one difference between the two: in logistic regression, the sigmoid function is applied to the result of the linear regression to transform it into a probability value between 0 and 1.

```
def linear_regression(xi):
    result = w0

    for j in range(len(w)):
        result = result + xi[j] * w[j]

    return result
```

```
def logistic_regression(xi):
    score = w0

    for j in range(len(w)):
        score = score + xi[j] * w[j]

    result = sigmoid(score)
    return result
```

Linear regression and logistic regression are called **linear models**, because dot product in linear algebra is a linear operator. Linear models are fast to use, fast to train.

• [Peter](#)
 ⌚ [30. September 2023](#)
 📁 [Classification, ML-Zoomcamp](#)
🏷️ [Classification](#), [Linear regression](#), [Logistic Regression](#), [ML Zoomcamp](#), [Sigmoid](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Machine Learning for Classification – Part 10

👤 Peter ⏰ 30. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Accuracy, Classification, Logistic Regression, ML Zoomcamp



Training logistic regression with Scikit-Learn

1. [Training logistic regression with Scikit-Learn](#)

1. [Train a model with Scikit-Learn](#)
2. [Apply model to the validation dataset](#)
3. [Calculate the accuracy](#)

Train a model with Scikit-Learn

When you want to train a logistic regression model, the process is quite similar to training a linear regression model.

```
from sklearn.linear_model import LogisticRegression
```

```
model = LogisticRegression()
```

```
model.fit(X_train, y_train) → To Train the model
```

We have it from when we did
DiVectorizer

You can use the 'coef_' attribute to display the weights (coefficients) in a logistic regression model.

```

model.coef_ → Easier To see using model.coef_.round(3)
# Output:
# array([[ 4.74725393e-01, -1.74869739e-01, -4.07533674e-01,
#         -2.96832307e-02, -7.79947901e-02,  6.26830488e-02,
#         -8.89697670e-02, -8.13913026e-02, -3.43104989e-02,
#         -7.33675219e-02, -3.35206588e-01,  3.16498334e-01,
#         -8.89697670e-02,  3.67393252e-03, -2.58133752e-01,
#         1.41436648e-01,  9.01908316e-03,  6.25300062e-02,
#         -8.89697670e-02, -8.12382600e-02,  2.65582755e-01,
#         -8.89697670e-02, -2.84291008e-01, -2.31202837e-01,
#         1.23524816e-01, -1.66018462e-01,  5.83404413e-02,
#         -8.70075565e-02, -3.20578701e-02,  7.04875625e-02,
#         -5.91001566e-02,  1.41436648e-01, -2.49114669e-01,
#         2.15471208e-01, -1.20363620e-01, -8.89697670e-02,
#         1.01655367e-01, -7.08936452e-02, -8.89697670e-02,
#         5.21853914e-02,  2.13378878e-01, -8.89697670e-02,
#         -2.32087131e-01, -7.04067163e-02,  3.82395921e-04]]]

```

The ‘coef_’ attribute in logistic regression returns a 2-dimensional array, but if you’re interested in the weight vector ‘w,’ you can access it by indexing the first row. In most cases, you’ll find the weight vector ‘w’ you’re interested in by accessing ‘coef_[0].’

```
model.coef_[0].round(3)
```

```

# Output:
# array([ 0.475, -0.175, -0.408, -0.03 , -0.078,  0.063, -0.089, -0
#        -0.034, -0.073, -0.335,  0.316, -0.089,  0.004, -0.25
#        0.009,  0.063, -0.089, -0.081,  0.266, -0.089, -0.28
#        0.124, -0.166,  0.058, -0.087, -0.032,  0.07 , -0.05
#        -0.249,  0.215, -0.12 , -0.089,  0.102, -0.071, -0.089
#        0.213, -0.089, -0.232, -0.07 ,  0.   ])

```

You can use the ‘intercept_’ attribute to display the bias term (intercept) in a logistic regression model.

```

model.intercept_
# Output: array([-0.10903301])

# actually it's an array with one element
model.intercept_[0]
# Output: -0.10903300803603666

```

Now we have our trained logistic regression model, we can apply it to a dataset. Let’s begin by testing it on the training data.

```

model.predict(X_train)
# Output: array([0, 1, 1, ..., 1, 0, 1])

```

We observe that the model provides **hard predictions**, meaning it assigns either zeros (representing “not churn”) or ones (representing “churn”). These hard predictions are called such because we already have the exact labels in the training data.

Instead of hard predictions, we can generate **soft predictions** by using the predict_proba function, as demonstrated in the following snippet.

```
model.predict_proba(X_train)
# Output:
# array([[0.90451975, 0.09548025],
#        [0.32068109, 0.67931891],
#        [0.36632967, 0.63367033],
#        ...,
#        [0.46839952, 0.53160048],
#        [0.95745572, 0.04254428],
#        [0.30127894, 0.69872106]])
```

↳ Stands for probability

*Reminder, This means
63,3%*

Indeed, when using the predict_proba function in logistic regression, the output contains two columns. The first column represents the probability of belonging to the negative class (0), while the second column represents the probability of belonging to the positive class (1). In the context of churn prediction, we are typically interested in the second column, which represents the probability of churn.

Hence, you can simply extract the second column to obtain the probabilities of churn. Then, to make the final decision about whether to classify individuals as churned or not, you can choose a threshold. People with probabilities above this threshold are classified as churned, while those below it are classified as not churned. The choice of threshold can affect the model's precision, recall, and other performance metrics, so it's an important consideration when making predictions with logistic regression.

Apply model to the validation dataset

```
y_pred = model.predict_proba(X_val)[:, 1]
y_pred
# Output:
# array([0.00899701, 0.20452226, 0.21222307, ..., 0.13638772,
#        0.83739781])
```

The result is a binary array with predictions. To proceed, you can define your chosen threshold and use it to select all customers for whom you believe the model predicts churn. This process allows you to identify the customers whom the model suggests are likely to churn based on the chosen threshold.

```
churn_decision = y_pred > 0.5
churn_decision
# Output: array([False, False, False, ..., False, True, True,
df_val[churn_decision]
```

	customerid	gender	seniorcitizen	partner	dependents	tenure	phon
3	8433-wxgna	male	0	no	no	2	
8	3440-jpscl	female	0	no	no	6	
11	2637-fkfsy	female	0	yes	no	3	
12	7228-omtpn	male	0	no	no	4	
19	6711-flfdb	female	0	no	no	7	
...
1397	5976-jcjrh	male	0	yes	no	10	
1398	2034-cgrhz	male	1	no	no	24	
1399	5276-kqwhg	female	1	no	no	2	
1407	6521-ytyti	male	0	no	yes	1	
1408	3049-solay	female	0	yes	no	3	

311 rows × 20 columns

These are the individuals who will receive a promotional email with a discount. The process involves selecting all the rows for which the `churn_decision` is true, indicating that the model predicts them as likely to churn based on the chosen threshold.

```
df_val[churn_decision].customerid  
  
# Output:  
# 3      8433-wxgna  
# 8      3440-jpscl  
# 11     2637-fkfsy  
# 12     7228-omtpn  
# 19     6711-flfb  
#  
#       ..  
# 1397    5976-jcjrh  
# 1398    2034-cgrhz  
# 1399    5276-kqwhg  
# 1407    6521-yytyi  
# 1408    3049-solay  
# Name: customerid, Length: 311, dtype: object
```

Calculate the accuracy

Let's assess the accuracy of our predictions. This time, we'll use the accuracy metric instead of root mean squared error (RMSE). Accuracy is a common metric for evaluating classification models like logistic regression.

To calculate the accuracy, you can use the actual values `y_val` and your predicted values as integers. You can obtain integer values from your predicted probabilities by using the `astype(int)` function, which will convert the probabilities to either 0 or 1 based on your chosen threshold. Then, you can compare these integer predictions with the actual values to calculate the accuracy.

```
y_val  
# Output: array([0, 0, 0, ..., 0, 1, 1])  
  
churn_decision.astype(int)  
# Output: array([0, 0, 0, ..., 0, 1, 1])
```

You can check how many of your predictions match the actual `y_val` values to calculate accuracy. This is essentially a shortcut for calculating the fraction of True or 1 values in the array of comparisons between predictions and actual values.

```
(y_val == churn_decision).mean()  
# Output: 0.8034066713981547
```

Remember that for binary data, the mean = the % of values that are one

80% of our predictions match

Let's examine how the last line of the last snippet works internally.

```
df_pred = pd.DataFrame()  
df_pred['probability'] = y_pred  
df_pred['prediction'] = churn_decision.astype(int)  
df_pred['actual'] = y_val  
df_pred['correct'] = df_pred.prediction == df_pred.actual  
df_pred
```

	probability	prediction	actual	correct
0	0.008997	0	0	True
1	0.204522	0	0	True
2	0.212223	0	0	True
3	0.543039	1	1	True
4	0.213786	0	0	True
...
1404	0.313668	0	0	True
1405	0.039359	0	1	False
1406	0.136388	0	0	True
1407	0.799759	1	1	True
1408	0.837398	1	1	True

1409 rows × 4 columns

```
df_pred.correct.mean()
```

```
# Output: 0.8034066713981547
```

Certainly, in this context, the `mean()` function calculates the fraction of ones in the binary array. Since it's a boolean array, `True` values are automatically converted to 1, and `False` values are converted to 0 when calculating the mean. This automatic conversion simplifies the process of calculating accuracy.

You've observed that the model has an accuracy of 80%, which means it is correct in predicting the outcome in 80% of the cases. This indicates that the model is performing reasonably well in classifying whether customers will churn or not based on the chosen threshold and the evaluation on the validation dataset.

👤 Peter ⏰ 30. September 2023 📁 Classification, ML-Zoomcamp
🏷️ Accuracy, Classification, Logistic Regression, ML Zoomcamp

Leave a comment

ML Zoomcamp 2023 – Machine Learning for Classification – Part 11

👤 Peter ⏰ 1. October 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, ML Zoomcamp



Model interpretation

1. [Model interpretation](#)
 1. [Look at the coefficients](#)
 2. [Train a smaller model with fewer features](#)
 3. [Model interpretation](#)

Look at the coefficients

Now, we want to combine each feature with its corresponding coefficient. This involves associating each feature with the weight (coefficient) assigned to it by the logistic regression model.

```
v.get_feature_names_out()
Output:
array(['contract=month-to-month', 'contract=one_year',
       'contract=two_year', 'dependents=no', 'dependents=yes',
       'deviceprotection=no', 'deviceprotection=no_internet_service',
       'deviceprotection=yes', 'gender=female', 'gender=male',
       'internetservice=dsl', 'internetservice=fiber_optic',
       'internetservice=no', 'monthlycharges', 'multiplelines=no',
       'multiplelines=no_phone_service', 'multiplelines=yes',
       'onlinebackup=no', 'onlinebackup=no_internet_service',
       'onlinebackup=yes', 'onlinesecurity=no',
       'onlinesecurity=no_internet_service', 'onlinesecurity=yes',
       'paperlessbilling=no', 'paperlessbilling=yes', 'partner=no',
       'partner=yes', 'paymentmethod=bank_transfer_(automatic)',
       'paymentmethod=credit_card_(automatic)',
       'paymentmethod=electronic_check', 'paymentmethod=mailed_chec',
       'phoneservice=no', 'phoneservice=yes', 'seniorcitizen',
       'streamingmovies=no', 'streamingmovies=no_internet_service',
       'streamingmovies=yes', 'streamingtvtv=no',
       'streamingtvtv=no_internet_service', 'streamingtvtv=yes',
       'techsupport=no', 'techsupport=no_internet_service',
       'techsupport=yes', 'tenure', 'totalcharges'], dtype=object)
```

```
m odel.coef_[0].round(3)
Output:
array([ 0.475, -0.175, -0.408, -0.03 , -0.078,  0.063, -0.089, -0.
       -0.034, -0.073, -0.335,  0.316, -0.089,  0.004, -0.258,  0.1
       0.009,  0.063, -0.089, -0.081,  0.266, -0.089, -0.284, -0.2
       0.124, -0.166,  0.058, -0.087, -0.032,  0.07 , -0.059,  0.1
       -0.249,  0.215, -0.12 , -0.089,  0.102, -0.071, -0.089,  0.0
       0.213, -0.089, -0.232, -0.07 ,  0. ])
```

To combine both sets of information, you can use the `zip` function. This function allows you to pair each feature with its corresponding coefficient.

✓ Can also turn it into a dictionary with dict instead of list which is what he actually does in the version of the video I watch

```
# Output:
# [('contract=month-to-month', 0.475),
# ('contract=one_year', -0.175),
# ('contract=two_year', -0.408),
# ('dependents=no', -0.03),
# ('dependents=yes', -0.078),
# ('deviceprotection=no', 0.063),
# ('deviceprotection=no_internet_service', -0.089),
# ('deviceprotection=yes', -0.081),
# ('gender=female', -0.034),
# ('gender=male', -0.073),
# ('internetservice=dsl', -0.335),
# ('internetservice=fiber_optic', 0.316),
# ('internetservice=no', -0.089),
# ('monthlycharges', 0.004),
# ('multiplelines=no', -0.258),
# ('multiplelines=no_phone_service', 0.141),
# ('multiplelines=yes', 0.009),
# ('onlinebackup=no', 0.063),
# ('onlinebackup=no_internet_service', -0.089),
# ('onlinebackup=yes', -0.081),
# ('onlinesecurity=no', 0.266),
# ('onlinesecurity=no_internet_service', -0.089),
# ('onlinesecurity=yes', -0.284),
# ('paperlessbilling=no', -0.231),
# ('paperlessbilling=yes', 0.124),
# ...
# ('techsupport=no', 0.213),
# ('techsupport=no_internet_service', -0.089),
# ('techsupport=yes', -0.232),
# ('tenure', -0.07),
# ('totalcharges', 0.0)]
```

Train a smaller model with fewer features

Let's take a subset of features to train a model with fewer features.

```
small = ['contract', 'tenure', 'monthlycharges']
df_train[small].iloc[:10]
```

	contract	tenure	monthlycharge
0	two_year	72	115.50
1	month-to-month	10	95.25
2	month-to-month	5	75.55
3	month-to-month	5	80.85
4	two_year	18	20.10
5	month-to-month	4	30.50
6	month-to-month	1	75.10
7	month-to-month	1	70.30
8	two_year	72	19.75
9	month-to-month	6	109.90

```

df_train[small].iloc[:10].to_dict(orient='records')
Output:
[{'contract': 'two_year', 'tenure': 72, 'monthlycharges': 115.5},
 {'contract': 'month-to-month', 'tenure': 10, 'monthlycharges': 95.25},
 {'contract': 'month-to-month', 'tenure': 5, 'monthlycharges': 75.55},
 {'contract': 'month-to-month', 'tenure': 5, 'monthlycharges': 80.85},
 {'contract': 'two_year', 'tenure': 18, 'monthlycharges': 20.1},
 {'contract': 'month-to-month', 'tenure': 4, 'monthlycharges': 30.5},
 {'contract': 'month-to-month', 'tenure': 1, 'monthlycharges': 75.1},
 {'contract': 'month-to-month', 'tenure': 1, 'monthlycharges': 70.3},
 {'contract': 'two_year', 'tenure': 72, 'monthlycharges': 19.75},
 {'contract': 'month-to-month', 'tenure': 6, 'monthlycharges': 109.9}.

```

```

dicts_train_small = df_train[small].to_dict(orient='records')
dicts_val_small = df_val[small].to_dict(orient='records')

dv_small = DictVectorizer(sparse=False)
dv_small.fit(dicts_train_small)

# three binary features for the contract variable and two numerical
# monthlycharges and tenure
dv_small.get_feature_names_out()
Output:
array(['contract=month-to-month', 'contract=one_year',
       'contract=two_year', 'monthlycharges', 'tenure'], dtype='|S22')

```

For now, we want to look at the coefficients w and w_0

```

X_train_small = dv_small.transform(dicts_train_small)
model_small = LogisticRegression()
model_small.fit(X_train_small, y_train)

w0 = model_small.intercept_[0]
# Output: -2.476775661122344
w0
w = model_small.coef_[0]
w.round(3)
# Output: array([ 0.97 , -0.025, -0.949,  0.027, -0.036])

```

We can link them together now

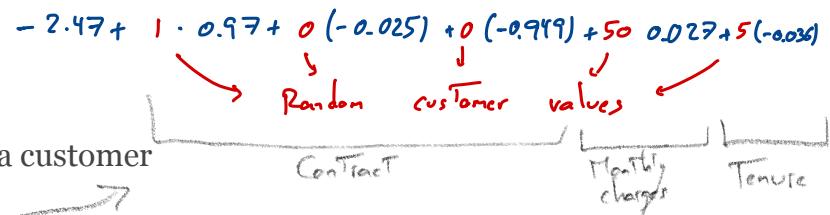
```

dict(zip(dv_small.get_feature_names_out(), w.round(3)))

```

Output:
{'contract=month-to-month': 0.97,
'contract=one-year': -0.025,
'contract=two_year': -0.949,
'monthlycharges': 0.027,
'tenure': -0.036}

Model interpretation



Now let's use the coefficients and score a customer

M 1Y 2Y
~~-2.47 + (1*0.97 + 0*(-0.025) + 0*(-0.949))~~ CONTRACT (customer has monthly contract)
~~+ 50*0.027~~ MONTHLYCHARGES (customer pays \$50 per month)
~~+ 5*(-0.036)~~ TENURE (tenure is 5 months)
~~=-0.33~~

```

sigmoid(-2.47)
Output: 0.07798823512936635

```

```

sigmoid(-2.47+0.97)
Output: 0.18242552380635632

```

```

sigmoid(-2.47 + 0.97 + 50*0.027)
Output: 0.46257015465625034

```

```

sigmoid(-2.47 + 0.97 + 50*0.027 + 5*(-0.036))
Output: 0.41824062315816374 → Probability of churning

```

~~2.47 + 0.97 + 50*0.027 + 5*(-0.036)~~
~~Output: -0.3300000000000001~~ ← Not the same

'_' is a magic variable in Jupyter and means take the output of the previous cell is

```

sigmoid(_)
Output: 0.41824062315816374

```

We see for this customer the probability of churning is 41.8%.

Let's calculate the score for another example where the result before applying the sigmoid

function is greater than 0, indicating that this customer is more likely to churn. As mentioned, a score greater than 0 implies a higher likelihood of churning, and sigmoid(0) corresponds to a 50% likelihood of churning.

```
-2.47 + 0.97 + 60*0.027 + 1*(-0.036)
# Output: 0.08399999999999966

sigmoid(_)
# Output: 0.5209876607065322
```

Let's calculate the score for one last example.

```
-2.47 + (-0.949) + 30*0.027 + 24*(-0.036)
# Output: -3.473

sigmoid(_)
# Output: 0.030090303318277657
```

The actual probability of this customer churning is very low, only 3%.

• [Peter](#) • [1. October 2023](#) ■ [Classification, ML-Zoomcamp](#)
🏷️ [Classification, ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Machine Learning for Classification – Part 12

👤 Peter ⏰ 1. October 2023 📁 Classification, ML-Zoomcamp
🏷️ Classification, ML Zoomcamp



Using the model

- train a model on full_train dataset

```
| df_full_train
```

	customerid	gender	seniorcitizen	partner	dependents	tenure	pho1
0	5442-pptjy	male	0	yes	yes	12	
1	6261-rcvns	female	0	no	no	42	
2	2176-osjuv	male	0	yes	no	71	
3	6161-erdgd	male	0	yes	yes	71	
4	2364-ufrom	male	0	no	no	30	
...
5629	0781-lkxbr	male	1	no	no	9	
5630	3507-gasnp	male	0	no	yes	60	
5631	8868-wozgu	male	0	no	no	28	
5632	1251-krreg	male	0	no	no	2	
5633	5840-nvdcg	female	0	yes	yes	16	

5634 rows × 21 columns

First we need to get the dictionaries.

```

dicts_full_train = df_full_train[categorical + numerical].to_dict(orient='records')
dicts_full_train[:3]

```

Output:

```

[{'gender': 'male',
 'seniorcitizen': 0,
 'partner': 'yes',
 'dependents': 'yes',
 'phoneservice': 'yes',
 'multiplelines': 'no',
 'internetservice': 'no',
 'onlinesecurity': 'no_internet_service',
 'onlinebackup': 'no_internet_service',
 'deviceprotection': 'no_internet_service',
 'techsupport': 'no_internet_service',
 'streamingtv': 'no_internet_service',
 'streamingmovies': 'no_internet_service',
 'contract': 'two_year',
 'paperlessbilling': 'no',
 'paymentmethod': 'mailed_check',
 'tenure': 12,
 'monthlycharges': 19.7,
 'totalcharges': 258.35},
 {'gender': 'female',
 'seniorcitizen': 0,
 'partner': 'no',
 'dependents': 'no',
 'phoneservice': 'yes',
 'multiplelines': 'no',
 ...
 'paperlessbilling': 'no',
 'paymentmethod': 'bank_transfer_(automatic)',
 'tenure': 71,
 'monthlycharges': 65.15,
 'totalcharges': 4681.75}]

```

```

dv = DictVectorizer(sparse=False)

```

from this dictionaries we get the feature matrix

```

X_full_train = dv.fit_transform(dicts_full_train)

```

then we train a model on this feature matrix

```

y_full_train = df_full_train.churn.values
model = LogisticRegression()
model.fit(X_full_train, y_full_train) → This is our final model, now we can

```

do the same things for test dataset

```

dicts_test = df_test[categorical + numerical].to_dict(orient='records')
X_test = dv.transform(dicts_test) ← Apply

```

do the predictions

```

y_pred = model.predict_proba(X_test)[:, 1]

```

compute accuracy

```

churn_decision = (y_pred >= 0.5) → We want to say that if probability > 50%, customer will churn
(churn_decision == y_test).mean()

```

Output: 0.815471965933286 ↳ We check the accuracy

An accuracy of 81.5% on the test data is slightly more accurate than what we had in the validation data. Minor differences in performance are acceptable, but significant differences between training and validation/test data can indeed indicate potential issues with the model, such as overfitting. Ensuring that the model's performance is consistent across different datasets is an important aspect of model evaluation and generalization.

Let's imagine that we want to deploy the logistic regression model on a website where we can use it to predict whether a customer is likely to leave (churn) or not. When a customer visits the website and provides their information, this data is transferred as a dictionary over the network to the server hosting the model. The server then uses the model to compute a probability, which is returned to determine whether the customer is likely to churn. This approach allows to make real-time predictions about customer churn and take appropriate actions, such as sending promotional offers to customers who are likely to leave.

Let's take a sample customer from our test set: *(an example of whether a customer would leave)*

```
customer = dict_test[10]           ↪ random person chosen
customer
# Output:
# {'gender': 'male',
#  'seniorcitizen': 1,
#  'partner': 'yes',
#  'dependents': 'yes',
#  'phoneservice': 'yes',
#  'multiplelines': 'no',
#  'internetservice': 'fiber_optic',
#  'onlinesecurity': 'no',
#  'onlinebackup': 'yes',
#  'deviceprotection': 'no',
#  'techsupport': 'no',
#  'streamingtvt': 'yes',
#  'streamingmovies': 'yes',
#  'contract': 'month-to-month',
#  'paperlessbilling': 'yes',
#  'paymentmethod': 'mailed_check',
#  'tenure': 32,
#  'monthlycharges': 93.95, → They pay a lot
#  'totalcharges': 2861.45}
```

To get the feature matrix for the requested customer as a dictionary, we create a list containing just that customer's dictionary.

`X_small = dv.transform([customer])` → We want to create the features matrix of the customer

: Output:

```
array([[1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 9.39500e+01, 1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00, 0.00000e+00, 3.20000e+01, 2.86145]]))
```

`X_small.shape`

: Output: (1, 45)

: one customer with 45 features

```
model.predict_proba(X_small)[0,1]
```

```
# Output: 0.4056810977975889
```

We see this customer has a probability of only 40% of churning. We assume this customer is not going to churn.

```
# Let's check the actual value...
```

```
y_test[10]
```

```
# Output: 0
```

Our decision not sending an email to this customer was correct. Let's test one customer that is going to churn.

```
customer = dicts_test[-1]
```

```
X_small = dv.transform([customer])
```

```
model.predict_proba(X_small)[0,1]
```

```
# Output: 0.5968852088398422
```

We see this customer has a probability of almost 60% of churning. We assume this customer is going to churn.

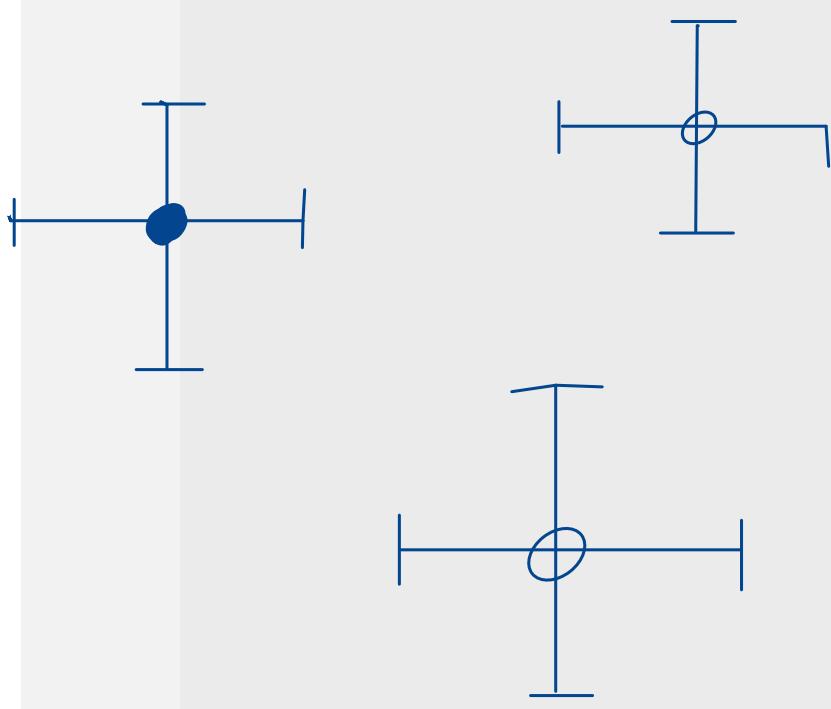
```
# Let's check the actual value...
```

```
y_test[-1]
```

```
# Output: 1
```

The prediction is correct.

EVALUATION METRICS



SUMMARY

Here we discuss the methods for the validation of binary classification models. To do so, we use metrics, which are numbers that describe the performance of a model. They include:

- Accuracy: fraction of correct answers, sometimes misleading
- Precision and recall: less misleading
- ROC curve: evaluates the performance at all thresholds, okay to use with imbalance
- K-fold CV: more reliable estimate for performance (mean + std)

With accuracy at different thresholds here we see that the best threshold is 0.5, but it is not the best metric here, 72% of customers are predicted not to churn. We then talk about confusion tables to visualize errors and we discuss precision and recall metrics. We see that for our model recall is not the best metric either. We then go for ROC curves, which evaluates at different thresholds different TPR and FPR. It tells you the possibility that a random selected positive example has a higher score than a randomly selected negative example. It tells you how well you can separate positive and negative scores. A perfect separation is if all positive samples have higher scores than negative ones.

Finally, cross-validation is discussed

ML Zoomcamp 2023 – Evaluation metrics for classification– Part 1

👤 Peter ⏰ 2. October 2023 📄 Evaluation Metrics, ML-Zoomcamp
🏷️ Classification, Logistic Regression, ML Zoomcamp



Overview

Today's post recaps all the important lines of code that are crucial for the rest of this chapter. This includes the necessary imports, data preparation, data splitting for training, validation, and testing, separating the target variable 'churn', training the logistic regression model, and finally, validating the model on the validation data and outputting the accuracy at the end.

In the first code snippet, we observe the necessary imports: Pandas, NumPy, and Matplotlib, as well as the three imports from the Scikit-Learn package. These imports are used once for the train-test-split, once for the DictVectorizer, and once for the linear model for LogisticRegression.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import LogisticRegression
```

First, the dataset is read and stored in the 'df' dataframe. In the second line, the column names are standardized by converting them to lowercase and replacing spaces with underscores. Then, all categorical columns are assigned to the variable 'categorical_columns' using the condition 'dtypes == object'. However, it's worth noting that the 'totalcharges' column is mistakenly considered categorical, but it is, in fact,

numerical. To correct this, the ‘totalcharges’ column is converted to a numerical format using the Pandas function ‘to_numeric’, with the ‘errors=’coerce” parameter set to ignore any errors. After this conversion, missing values are filled with ‘0’. Finally, the ‘churn’ column is converted to integer values, where ‘churn==yes’ becomes ‘1’ and ‘churn==no’ becomes ‘0’.

```
df = pd.read_csv('data-week-3.csv')

df.columns = df.columns.str.lower().str.replace(' ', '_')

categorical_columns = list(df.dtypes[df.dtypes == 'object'].

for c in categorical_columns:
    df[c] = df[c].str.lower().str.replace(' ', '_')

df.totalcharges = pd.to_numeric(df.totalcharges, errors='co
df.totalcharges = df.totalcharges.fillna(0)

df.churn = (df.churn == 'yes').astype(int)
```

Next, we use the ‘train_test_split’ function to split the datasets into ‘full_train’ (80%) and ‘test’ in the first step. In the second step, ‘full_train’ is further divided into two datasets for training and validation. Ultimately, we achieve a split of the initial data into a 60%-20%-20% ratio, where 60% is used for training, and 20% each is allocated for validation and testing. The parameter ‘random_state=1’ ensures that the random split is reproducible.

Following this, the next three lines reset the indices in each of the three datasets. Since the random split may result in non-continuous indices, setting ‘drop=True’ removes the old index.

Then, for each record, the target variable ‘y’ is set, which in this case is the ‘churn’ column. Finally, the target column is removed from the records to prevent accidental usage during training.

```
df_full_train, df_test = train_test_split(df, test_size=0.2,
df_train, df_val = train_test_split(df_full_train, test_size=0.5)

df_train = df_train.reset_index(drop=True)
df_val = df_val.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)

y_train = df_train.churn.values
y_val = df_val.churn.values
y_test = df_test.churn.values

del df_train['churn']
del df_val['churn']
del df_test['churn']
```

In the next snippet, we define two variables, ‘numerical’ and ‘categorical’, which contain the relevant column names. The ‘numerical’ array contains the names of all numerical columns, while the ‘categorical’ array contains the names of all categorical columns.

```
numerical = ['tenure', 'monthlycharges', 'totalcharges']

categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
               'phoneservice', 'multiplelines', 'internetservice',
               'onlinesecurity', 'onlinebackup', 'deviceprotection',
               'streamingtv', 'streamingmovies', 'contract', 'paperlessbilling',
               'paymentmethod']
```

Next, we create a DictVectorizer instance. We then transform the dataframe into dictionaries, and using the ‘fit_transform(train_dict)’ function, we train the DictVectorizer. This step involves showing the DictVectorizer how the data is structured, allowing it to distinguish column names and values and perform one-hot encoding based on this information. Importantly, the DictVectorizer is smart enough to distinguish between categorical values and numeric values, so numeric values are ignored during one-hot encoding. The ‘transform’ part of this process converts the dictionary into a vector or matrix suitable for machine learning.

After preparing the data, we move on to model creation. In this case, a Logistic Regression model is used. The ‘model.fit’ function is then employed to train the model on the training data.

```
dv = DictVectorizer(sparse=False)

train_dict = df_train[categorical + numerical].to_dict(orient='records')
X_train = dv.fit_transform(train_dict)

model = LogisticRegression()
model.fit(X_train, y_train)
```

We can then proceed to validate the trained model using the validation data. To do this, we need to prepare the validation DataFrame in the same way as shown for the training DataFrame. This involves transforming it into dictionaries and applying the ‘transform’ function of the DictVectorizer. However, during validation, we only need to use the ‘transform’ function of the DictVectorizer since it already knows the data structure. In the case of validation, we are primarily interested in the transformed output, which serves as input for prediction.

For prediction, we use the ‘predict_proba’ function of the model, which provides us with probabilities in two columns. Here, we are interested in the second column. We evaluate the model’s performance using a threshold of ‘ $>=0.5$ ’. The ‘churn_decision’ variable contains ‘True’ for any value in the prediction greater than or equal to the threshold, and ‘False’ otherwise. We calculate the accuracy using the ‘mean’ function, and in this case, it is approximately 80%.

```
val_dict = df_val[categorical + numerical].to_dict(orient='records')
X_val = dv.transform(val_dict)

y_pred = model.predict_proba(X_val)[:, 1]
churn_decision = (y_pred >= 0.5)
(y_val == churn_decision).mean()

# Output: 0.8034066713981547
```

ML Zoomcamp 2023 – Evaluation metrics for classification– Part 2

👤 Peter ⏰ 3. October 2023 📁 Evaluation Metrics, ML-Zoomcamp
🏷️ Accuracy, Classification, ML Zoomcamp



1. [Accuracy and Dummy Model](#)
 1. [Evaluate the model on different thresholds](#)
 2. [Check Accuracy of Dummy Baseline](#)

Accuracy and Dummy Model

In the last article, we calculated that our model achieved an accuracy of 80% on the validation data. Now, let's determine whether this is a good value or not.

Accuracy measures the fraction of correct predictions made by the model.

In our evaluation, we checked each customer in the validation dataset to determine whether the model's churn prediction was correct or incorrect. This decision was based on our threshold of 0.5, meaning a customer with a predicted value greater than or equal to 0.5 was considered a churning customer, while values below the threshold were considered non-churning customers.

Out of the 1409 customers in the validation dataset, we made 1132 correct predictions. Therefore, the accuracy is calculated as $1132/1409 = 0.80$, which corresponds to 80%. This accuracy indicates that our model correctly predicted the churn status for 80% of the customers in the validation dataset. Whether this is considered good or not depends on the specific context and requirements of the problem.

```
len(y_val)
# Output: 1409

(y_val == churn_decision).sum()
# Output: 1132

1132 / 1409
# Output: 0.8034

(y_val == churn_decision).mean()
# Output: 0.8034
```

Evaluate the model on different thresholds

The question now is whether we have chosen a good value for the threshold. To evaluate this, we can adjust the threshold and perform validation again. By systematically varying the threshold, we can observe whether it improves the accuracy or not. To do this, we can use the ‘linspace’ function from NumPy to generate an array with multiple threshold values (e.g., 21 values evenly spaced between 0 and 1). For each threshold value, we can calculate the accuracy and then determine the best threshold value based on the validation results. This process allows us to fine-tune the threshold to optimize the model’s performance.

```
thresholds = np.linspace(0, 1, 21)
thresholds

# Output:
# array([0.          , 0.05, 0.1       , 0.15, 0.2       , 0.25, 0.3       ,
#        0.35, 0.4       , 0.45, 0.5       , 0.55, 0.6       , 0.65, 0.7       ,
#        0.75, 0.8       , 0.85, 0.9       , 0.95])
```

```

scores = []

for t in thresholds:
    churn_decision = (y_pred >= t)
    score = (y_val == churn_decision).mean()
    print('%.2f %.3f' % (t, score))
    scores.append(score)

# Output:
# 0.00 0.274
# 0.05 0.509
# 0.10 0.591
# 0.15 0.666
# 0.20 0.710
# 0.25 0.739
# 0.30 0.760
# 0.35 0.772
# 0.40 0.785
# 0.45 0.793
# 0.50 0.803
# 0.55 0.801
# 0.60 0.795
# 0.65 0.786
# 0.70 0.766
# 0.75 0.744
# 0.80 0.735
# 0.85 0.726
# 0.90 0.726
# 0.95 0.726
# 1.00 0.726

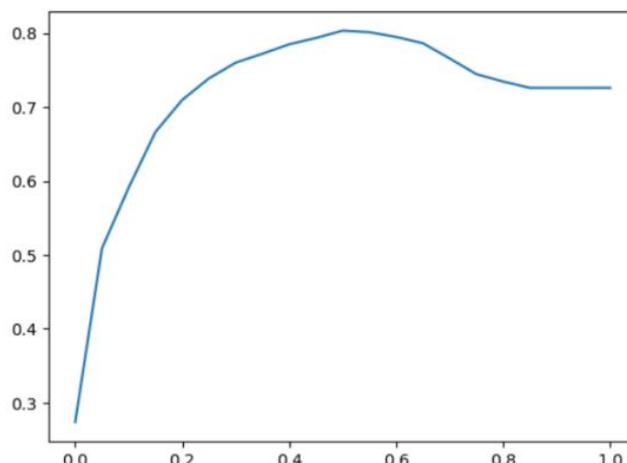
```

Basically, if the y-pred is greater than the threshold I set To determine that a customer will churn...

*'%.2f %.3f' means that
The first value has 2 decimals and the second one has 3 decimals*

It appears that 0.5 is indeed the best threshold based on the validation set. This suggests that the default threshold of 0.5 is an appropriate choice for our model in this context. To visually represent this threshold optimization process, we can create a plot. The x-axis will represent the threshold values, while the y-axis will represent the corresponding scores (in this case, accuracy or another relevant metric). This plot will provide a clear visualization of how the model's performance varies with different threshold values, helping us identify the threshold that maximizes the desired metric.

```
plt.plot(thresholds, scores)
```



While we used our custom function to calculate accuracy, it's worth noting that Scikit-

Learn provides built-in functions for common evaluation metrics, including accuracy. These built-in functions can simplify the process of evaluating your model's performance, making it more convenient and efficient.

```
from sklearn.metrics import accuracy_score

thresholds = np.linspace(0, 1, 21)
scores = []

for t in thresholds:
    score = accuracy_score(y_val, y_pred >= t)
    print('{:.2f} {:.3f}'.format(t, score))
    scores.append(score)

# Output:
# 0.00 0.274
# 0.05 0.509
# 0.10 0.591
# 0.15 0.666
# 0.20 0.710
# 0.25 0.739
# 0.30 0.760
# 0.35 0.772
# 0.40 0.785
# 0.45 0.793
# 0.50 0.803
# 0.55 0.801
# 0.60 0.795
# 0.65 0.786
# 0.70 0.766
# 0.75 0.744
# 0.80 0.735
# 0.85 0.726
# 0.90 0.726
# 0.95 0.726
# 1.00 0.726
```

Check Accuracy of Dummy Baseline

There is an important point about the limitations of accuracy as an evaluation metric. While we may achieve a certain level of accuracy, it doesn't always provide the full picture of a model's performance, especially in cases with imbalanced datasets or when specific types of errors are more critical than others.

In this example, I've mentioned that the accuracy of the model is 80%, but a dummy model that predicts all customers as not churning achieves an accuracy of 73%. This highlights the issue with accuracy, as it doesn't differentiate between different types of errors. In churn prediction, false negatives (predicting a customer won't churn when they actually do) can be more costly than false positives (predicting a customer will churn when they won't).

Choosing the most appropriate evaluation metric depends on the specific goals and requirements of the problem. For example, in cases where minimizing false negatives is crucial (e.g., in medical diagnoses or fraud detection), recall may be a more relevant metric

than accuracy.

```
from collections import Counter

Counter(y_pred >= 1.0)
# Output: Counter({False: 1409})

# Distribution of y_val
Counter(y_val)

# Output: Counter({0: 1023, 1: 386})

1023 / 1409
# Output: 0.7260468417317246

y_val.mean()
# Output: 0.2739531582682754

1 - y_val.mean()
# Output: 0.7260468417317246
```

We can observe that there are significantly more non-churning customers than churning ones, with only 27% being churning customers and 73% being non-churning customers. This situation highlights a common challenge known as class imbalance, where one class has far more samples than the other.

In cases of class imbalance, the traditional accuracy metric can be misleading. For example, a dummy model that predicts the majority class for all samples can achieve a high accuracy simply by getting most of the samples right for the majority class. However, it will perform poorly in identifying the minority class (in this case, the churning customers), which is often more crucial to predict accurately.

To effectively address class imbalance and evaluate our model, we should consider alternative metrics such as:

1. **Precision:** This metric measures the proportion of true positive predictions among all positive predictions. It is particularly useful when the cost of false positives is high.
2. **Recall:** Recall measures the proportion of true positive predictions among all actual positive instances. It is valuable when the cost of false negatives is significant.
3. **F1-Score:** The F1-Score is the harmonic mean of precision and recall, providing a balanced measure that considers both false positives and false negatives.
4. **Area Under the Receiver Operating Characteristic Curve (AUC-ROC):** The ROC curve plots the true positive rate against the false positive rate at various threshold settings. The AUC-ROC score assesses the classifier's ability to distinguish between the positive and negative classes, making it particularly useful for imbalanced datasets.

Selecting the appropriate evaluation metric depends on the specific goals and requirements of the problem. In cases of class imbalance, accurate identification of the minority class (churning customers) is crucial.

ML Zoomcamp 2023 – Evaluation metrics for classification– Part 3

👤 Peter ⏰ 4. October 2023 📄 Evaluation Metrics, ML-Zoomcamp
🏷️ [Confusion Matrix, ML Zoomcamp](#)



1. [Confusion table / matrix](#)
 1. [Different types of errors and correct decisions](#)
 2. [Arranging them in a table](#)

Confusion table / matrix → *Allows us to arrange the errors we made in a Table*

Different types of errors and correct decisions

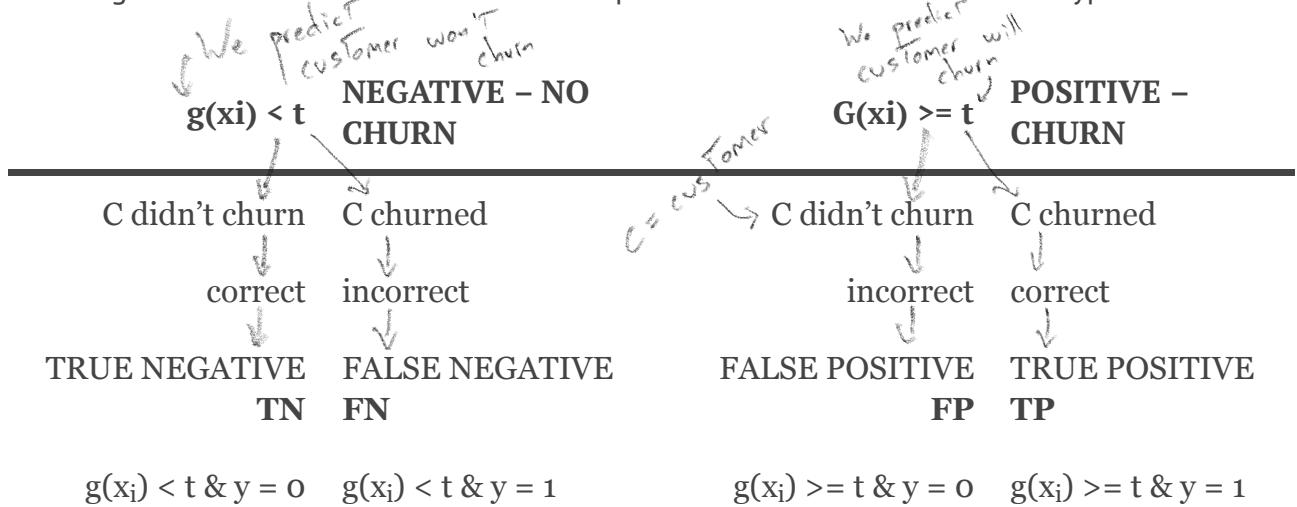
In this section, we'll discuss the confusion matrix, a vital tool for evaluating the performance of binary classification models. The confusion matrix allows us to examine the various errors and correct decisions made by our model.

As we've previously discussed, class imbalance can significantly impact the accuracy metric. To address this issue, we need alternative evaluation methods that provide a more comprehensive view of our model's performance.

The confusion matrix breaks down the model's predictions into four categories:

1. **True Positives (TP)**: These are cases where the model correctly predicted the positive class (churning customers).
2. **True Negatives (TN)**: These are cases where the model correctly predicted the negative class (non-churning customers).
3. **False Positives (FP)**: These are cases where the model incorrectly predicted the positive class when the true class was negative. This is also known as a Type I error.
4. **False Negatives (FN)**: These are cases where the model incorrectly predicted the

negative class when the true class was positive. This is also known as a Type II error.



TN, FN, FP, TP for churn project

To check this, we first check whether people churned

```
# people who are going to churn
actual_positive = (y_val == 1)
# people who are not going to churn
actual_negative = (y_val == 0)
```

```
t = 0.5
predict_positive = (y_pred >= t)
predict_negative = (y_pred < t)
```

We examine the cases where both “predict_positive” and “actual_positive” are true. This is precisely what the “**&**” operator represents, indicating a logical AND operation.

```
predict_positive & actual_positive
# Output: array([False, False, False, ..., False, True, True])
tp = (predict_positive & actual_positive).sum()
tp
# Output: 210
tn = (predict_negative & actual_negative).sum()
tn
# Output: 922
fp = (predict_positive & actual_negative).sum()
fp
# Output: 101
fn = (predict_negative & actual_positive).sum()
fn
# Output: 176
```

Arranging them in a table

That was preparation for understanding the confusion matrix. The confusion matrix is a way to consolidate all these values (tp, tn, fp, fn) into a single table. This table comprises 4 cells, forming a 2×2 matrix. (See next page to see the table)

- In the columns of this table, we have the predictions (NEGATIVE: $g(x_i) < t$ and POSITIVE: $g(x_i) \geq t$).
- In the rows, we have the actual values (NEGATIVE: $y=0$ and POSITIVE: $y=1$).

Now, let's proceed to implement this confusion matrix in NumPy.

```
confusion_matrix = np.array([
    [tn, fp],
    [fn, tp]
])

confusion_matrix
# Output:
# array([[922, 101],
#        [176, 210]])
```

		NO CHURN $g(x_i) < t$ NEGATIVE	CHURN $g(x_i) \geq t$ POSITIVE
NO CHURN $y=0$ NEGATIVE	True Negative TN	101 8%	
	False Negative FN	210 15%	
CHURN $y=1$ POSITIVE	922 65%	176 12%	

x-axis = Prediction, y-axis = Actual values / Accuracy = $65+15 = 80\%$

We observe that we have more false negatives than false positives. False positives represent customers who receive the email even though they are not likely to churn, resulting in a loss of money due to unnecessary discounts. False negatives are customers who do not receive the email and end up leaving, causing financial losses as well. Both situations are undesirable.

Instead of using absolute numbers, we can also express these values in relative terms to gain a better perspective on the model's performance.

```
(confusion_matrix / confusion_matrix.sum()).round(2)
# Output:
# array([[0.65, 0.07],
#        [0.12, 0.15]])
```

👤 Peter ⏰ 4. October 2023 📚 Evaluation Metrics, ML-Zoomcamp
⭐ Confusion Matrix, ML Zoomcamp

From the values in this Table we can later on do Precision and Recall (next chapter) and ROC Curves (next after that)

ML Zoomcamp 2023 – Evaluation metrics for classification– Part 4

👤 Peter ⏰ 5. October 2023 📁 Evaluation Metrics, ML-Zoomcamp
🏷️ Accuracy, Confusion Matrix, ML Zoomcamp, Precision, Recall



Precision & Recall

Precision and Recall are essential metrics for evaluating binary classification models.

Precision measures the fraction of positive predictions that were correct. In other words, it quantifies how accurately the model predicts customers who are likely to churn.

Precision = True Positives / (# Positive Predictions) = True Positives / (True Positives + False Positives)

Recall, on the other hand, quantifies the fraction of actual positive cases that were correctly identified by the model. It assesses how effectively the model captures all customers who are actually churning.

Recall = True Positives / (# Positive Observations) = True Positives / (True Positives + False Negatives)

In summary, precision focuses on the accuracy of positive predictions, while recall emphasizes the model's ability to capture all positive cases. These metrics are crucial for understanding the trade-offs between correctly identifying churning customers and minimizing false positives.

Actual Values	Negative Predictions	positive Predictions
	$g(x_i) < t$	$g(x_i) \geq t$
Negative Example $y=0$	TN	FP
Positive Example $y=1$	FN	TP
Confusion matrix Recall=TP/(TP+FN) Precision=TP/(TP+FP)		

```

accuracy = (tp + tn) / (tp + tn + fp + fn)
accuracy
# Output: 0.8034066713981547

precision = tp / (tp + fp)
precision
# Output: 0.6752411575562701

# --> promotional email goes to 311 people, but 210 are actual
tp + fp
# Output: 210

recall = tp / (tp + fn)
recall
# Output: 0.5440414507772021

# --> For 46% of people who are churning we failed to identify
tp + fn
# Output: 386

```

While accuracy can give a misleading impression of a model's performance, metrics like precision and recall are much more informative, especially in situations with class imbalance. Precision and recall provide a more detailed understanding of how well the model is performing in identifying positive cases (in this case, churning customers).

In scenarios where correctly identifying specific cases is critical, such as identifying churning customers to prevent loss, precision and recall help us make more informed decisions and assess the trade-offs between correctly identifying positives and minimizing false positives or false negatives. So, relying solely on accuracy may not provide a complete picture of a model's effectiveness for a particular task.

 [Peter](#)  [5. October 2023](#)  [Evaluation Metrics](#), [ML-Zoomcamp](#)
 [Accuracy](#), [Confusion Matrix](#), [ML Zoomcamp](#), [Precision](#), [Recall](#)

Leave a comment

ML Zoomcamp 2023 – Evaluation metrics for classification– Part 5

👤 Peter ⏰ 6. October 2023 📄 Evaluation Metrics, ML-Zoomcamp
🏷️ [False Positive Rate](#), [FPR](#), [ML Zoomcamp](#), [ROC](#), [TPR](#), [True Positive Rate](#)



1. [ROC Curve \(Receiver Operating Characteristics\)](#)

1. [Random model](#)
2. [Ideal model](#)
3. [Putting everything together](#)
4. [What kind of information do we get from ROC curve?](#)

ROC Curve (Receiver Operating Characteristics) → Shows you TPR vs FPR

ROC (Receiver Operating Characteristic) curves are a valuable tool for evaluating binary classification models, especially in scenarios where you want to assess the trade-off between false positives and true positives at different decision thresholds.

The ROC curve visually represents the performance of a model by plotting the TPR (True Positive Rate or Sensitivity) against the FPR (False Positive Rate or 1 – Specificity) at various threshold settings. The area under the ROC curve (AUC-ROC) is a summary measure of a model's overall performance, with a higher AUC indicating better discrimination between positive and negative cases.

ROC curves help you make informed decisions about the choice of threshold that balances your priorities between minimizing false positives (FPR) and maximizing true positives (TPR) based on the specific context and requirements of your problem.

Actual Values	Predictions		$\text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}}$
	Negative Predictions	positive Predictions	
Negative Example $y=0$	$g(x_i) < t$	$g(x_i) \geq t$	TN
Positive Example $y=1$	FN	TP	FP

Confusion matrix FPR – False Positive Rate TPR – True Positive Rate

```
tpr = tp / (tp + fn)
tpr
# Output: 0.5440414507772021
recall
# Output: 0.5440414507772021
# --> tpr = recall

fpr = fp / (fp + tn)
fpr
# Output: 0.09872922776148582
```

The ROC curve is a useful visualization tool that allows you to assess the performance of a binary classification model across a range of decision thresholds.

scores = [] \longrightarrow To store the values generated
thresholds = np.linspace(0, 1, 101)

```
for t in thresholds:  
    actual_positive = (y_val == 1)  
    actual_negative = (y_val == 0)  
  
    predict_positive = (y_pred >= t)  
    predict_negative = (y_pred < t)  
  
    tp = (predict_positive & actual_positive).sum()  
    tn = (predict_negative & actual_negative).sum()  
  
    fp = (predict_positive & actual_negative).sum()  
    fn = (predict_negative & actual_positive).sum()  
  
    scores.append((t, tp, tn, fp, fn))
```

scores

```
# Output:  
# [(0.0, 386, 0, 1023, 0),  
# (0.01, 385, 110, 913, 1),  
# (0.02, 384, 193, 830, 2),  
# (0.03, 383, 257, 766, 3),  
# (0.04, 381, 308, 715, 5),  
# (0.05, 379, 338, 685, 7),  
# (0.06, 377, 362, 661, 9),  
# (0.07, 372, 382, 641, 14),  
# (0.08, 371, 410, 613, 15),  
# (0.09, 369, 443, 580, 17),  
# (0.1, 366, 467, 556, 20),  
# (0.11, 365, 495, 528, 21),  
# (0.12, 365, 514, 509, 21),  
# (0.13, 360, 546, 477, 26),  
# (0.14, 355, 570, 453, 31),  
# (0.15, 351, 588, 435, 35),  
# (0.16, 347, 604, 419, 39),  
# (0.17, 346, 622, 401, 40),  
# (0.18, 344, 639, 384, 42),  
# (0.19, 338, 654, 369, 48),  
# (0.2, 333, 667, 356, 53),  
# (0.21, 330, 682, 341, 56),  
# (0.22, 323, 701, 322, 63),  
# (0.23, 320, 710, 313, 66),  
# (0.24, 316, 719, 304, 70),  
# ...  
# (0.96, 0, 1023, 0, 386),  
# (0.97, 0, 1023, 0, 386),  
# (0.98, 0, 1023, 0, 386),  
# (0.99, 0, 1023, 0, 386),  
# (1.0, 0, 1023, 0, 386)]
```

We end up with 101 confusion matrices evaluated for different thresholds. Let's turn that into a dataframe.

```
columns = ['threshold', 'tp', 'tn', 'fp', 'fn']
df_scores = pd.DataFrame(scores, columns=columns)
df_scores
```

	threshold	TP	TN	FP	FN
0	0.00	386	0	1023	0
1	0.01	385	110	913	1
2	0.02	384	193	830	2
3	0.03	383	257	766	3
4	0.04	381	308	715	5
...
96	0.96	0	1023	0	386
97	0.97	0	1023	0	386
98	0.98	0	1023	0	386
99	0.99	0	1023	0	386
100	1.00	0	1023	0	386

101 rows × 5 columns

We can look at each tenth record by using this column `:10` operator. This works by printing every record starting from the first record and moving forward with increments of 10.

```
df_scores[::10]
```

	threshold	tp	tn	fp	fn
0	0.0	386	0	1023	0
10	0.1	366	467	556	20
20	0.2	333	667	356	53
30	0.3	284	787	236	102
40	0.4	249	857	166	137
50	0.5	210	922	101	176
60	0.6	150	970	53	236
70	0.7	76	1003	20	310
80	0.8	13	1022	1	373
90	0.9	0	1023	0	386
100	1.0	0	1023	0	386

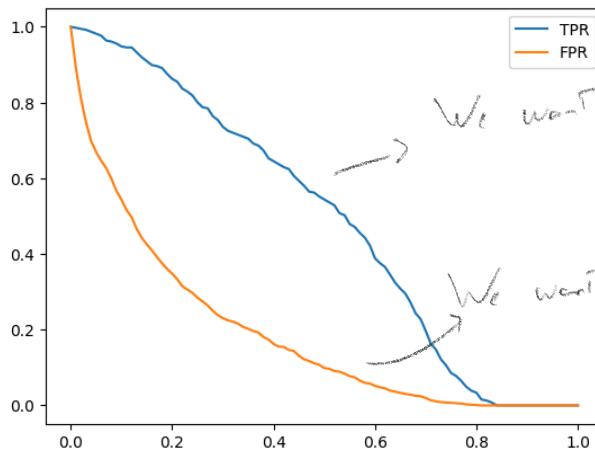
```

df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fp)
df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)
df_scores[:10]

```

	threshold	tp	TN	FP	FN	tpr	fpr
0	0.0	386	0	1023	0	1.000000	1.000000
10	0.1	366	467	556	20	0.948187	0.543500
20	0.2	333	667	356	53	0.862694	0.347996
30	0.3	284	787	236	102	0.735751	0.230694
40	0.4	249	857	166	137	0.645078	0.162268
50	0.5	210	922	101	176	0.544041	0.098729
60	0.6	150	970	53	236	0.388601	0.051808
70	0.7	76	1003	20	310	0.196891	0.019550
80	0.8	13	1022	1	373	0.033679	0.000978
90	0.9	0	1023	0	386	0.000000	0.000000
100	1.0	0	1023	0	386	0.000000	0.000000

```
plt.plot(df_scores.threshold, df_scores['tpr'], label='TPR')
plt.plot(df_scores.threshold, df_scores['fpr'], label='FPR')
plt.legend()
```



if To be as close to one as possible

if To be as low as possible

Random model

Let's write the case where the prediction of whether a customer churns is random

```
np.random.seed(1)
y_rand = np.random.uniform(0, 1, size=len(y_val))
y_rand.round(3)
# Output: array([0.417, 0.72 , 0. , ..., 0.774, 0.334, 0.085])
```

```
# Accuracy for our random model is around 50%
((y_rand >= 0.5) == y_val).mean()

# Output: 0.5017743080198722
```

Let's put the previously used code into a function.

```

def tpr_fpr_dataframe(y_val, y_pred):
    scores = []
    thresholds = np.linspace(0, 1, 101)

    for t in thresholds:
        actual_positive = (y_val == 1)
        actual_negative = (y_val == 0)

        predict_positive = (y_pred >= t)
        predict_negative = (y_pred < t)

        tp = (predict_positive & actual_positive).sum()
        tn = (predict_negative & actual_negative).sum()

        fp = (predict_positive & actual_negative).sum()
        fn = (predict_negative & actual_positive).sum()

        scores.append((t, tp, tn, fp, fn))

    columns = ['threshold', 'tp', 'tn', 'fp', 'fn']
    df_scores = pd.DataFrame(scores, columns=columns)

    df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
    df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)

    return df_scores

```

```

df_rand = tpr_fpr_dataframe(y_val, y_rand)
df_rand[::10]

```

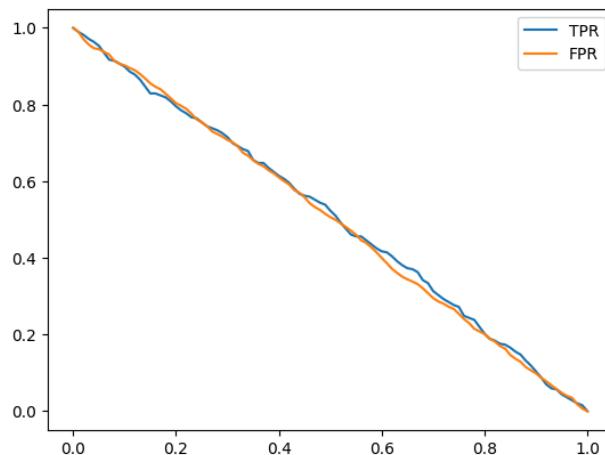
	threshold	tp	tn	fp	fn	tpr	fpr
0	0.0	386	0	1023	0	1.000000	1.000000
10	0.1	347	100	923	39	0.898964	0.902248
20	0.2	307	201	822	79	0.795337	0.803519
30	0.3	276	299	724	110	0.715026	0.707722
40	0.4	237	399	624	149	0.613990	0.609971
50	0.5	202	505	518	184	0.523316	0.506354
60	0.6	161	614	409	225	0.417098	0.399804
70	0.7	121	721	302	265	0.313472	0.295210
80	0.8	78	817	206	308	0.202073	0.201369
90	0.9	40	922	101	346	0.103627	0.098729
100	1.0	0	1023	0	386	0.000000	0.000000

Values decrease as T increases

```

plt.plot(df_rand.threshold, df_rand['tpr'], label='TPR')
plt.plot(df_rand.threshold, df_rand['fpr'], label='FPR')
plt.legend()

```



Let's examine an example using a threshold of 0.6. On the x-axis, we have our thresholds, and when we set the threshold to 0.6, we obtain a True Positive Rate (TPR) of 0.4 and a False Positive Rate (FPR) of 0.4.

The reason behind these values is that our model's predictions are nearly equivalent to tossing a coin. In 60% of cases, the model predicts that a customer is non-churning, and in 40% of cases, it predicts that the customer is churning. In other words, these rates indicate that the model predicts a customer as churning with a 40% probability and as non-churning with a 60% probability. Consequently, the model is incorrect for non-churning customers in 40% of cases.

Ideal model

I. The ideal model, we sort the scores of each customer from lower value to higher value and we find the line that separates those who churn and those who don't

Now, let's discuss the concept of an ideal model that makes correct predictions for every example. To implement this, we need to determine the number of negative examples, which corresponds to the number of people who are not churning in our dataset.

```

num_neg = (y_val == 0).sum()
num_pos = (y_val == 1).sum()
num_neg, num_pos

# Output: (1023, 386)

```

To create the ideal model's predictions for our validation set, we first create a `y_ideal` array that contains only negative observations (0s) followed by positive observations (1s). We use the `np.repeat()` function to achieve this, creating an array with 1023 zeros and then 386 ones.

This function repeats the zeros as many times as you have num_neg and same for the ones

```

y_ideal = np.repeat([0, 1], [num_neg, num_pos])
y_ideal

# Output: array([0, 0, 0, ..., 1, 1, 1])

```

To create our predictions for the ideal model, which are numbers between 0 and 1, we can use the `np.linspace()` function to generate an array of evenly spaced values between 0 and 1. This array should have the same length as `y_ideal`, which is 1409 in this case.

```
y_ideal_pred = np.linspace(0, 1, len(y_ideal))
y_ideal_pred
# Output:
# array([0.0000000e+00, 7.10227273e-04, 1.42045455e-03, ...,
#        9.98579545e-01, 9.99289773e-01, 1.0000000e+00])

1 - y_val.mean()
# Output: 0.7260468417317246

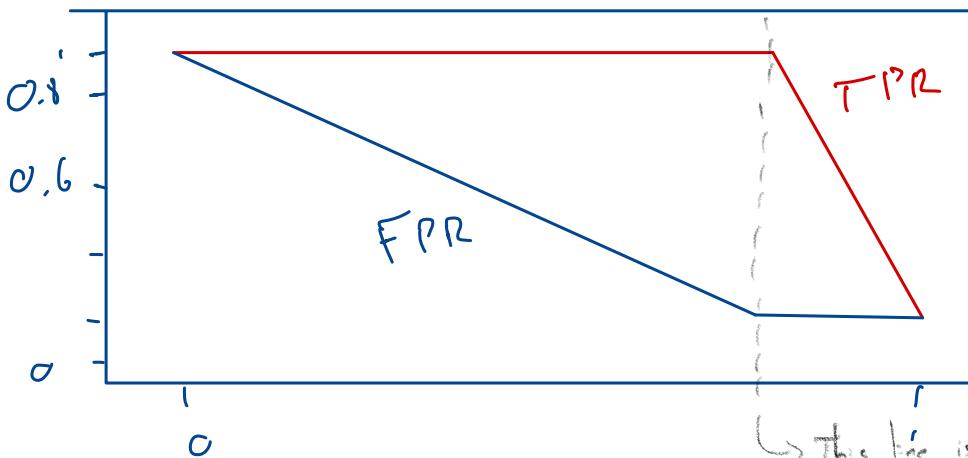
accuracy_ideal = ((y_ideal_pred >= 0.726) == y_ideal).mean()
accuracy_ideal
# Output: 1.0
```

The ideal model, which makes perfect predictions, doesn't exist in reality, but it serves as a benchmark to understand how well our actual model is performing. By comparing our model's performance to that of the ideal model, we can assess how much room for improvement there is.

```
df_ideal = tpr_fpr_dataframe(y_ideal, y_ideal_pred)
df_ideal[:10]
```

	threshold	tp	tn	fp	fn	tpr	fpr
0	0.0	386	0	1023	0	1.000000	1.000000
10	0.1	386	141	882	0	1.000000	0.862170
20	0.2	386	282	741	0	1.000000	0.724340
30	0.3	386	423	600	0	1.000000	0.586510
40	0.4	386	564	459	0	1.000000	0.448680
50	0.5	386	704	319	0	1.000000	0.311828
60	0.6	386	845	178	0	1.000000	0.173998
70	0.7	386	986	37	0	1.000000	0.036168
80	0.8	282	1023	0	104	0.730570	0.000000
90	0.9	141	1023	0	245	0.365285	0.000000
100	1.0	1	1023	0	385	0.002591	0.000000

```
plt.plot(df_ideal.threshold, df_ideal['tpr'], label='TPR')
plt.plot(df_ideal.threshold, df_ideal['fpr'], label='FPR')
plt.legend()
```



↳ This line is the limit that we mentioned at the beginning of ideal Model section.

What we see here is that TPR almost always stays around 1 and starts to go down after the threshold of 0.726. So, this model can correctly identify churning customers up to that threshold. For people who are not churning but are classified as churning by the model when the threshold is below 0.726, the model is not always correct. However, the detection becomes always true after the threshold of 0.726.

Let's take another example with a threshold of 0.4. The FPR is around 45%, and the model makes some mistakes. So, for around 32% of people who are predicted as non-churning when the threshold is set to 0.726 but are below that threshold, we predict them as churning even though they are not.

Putting everything together

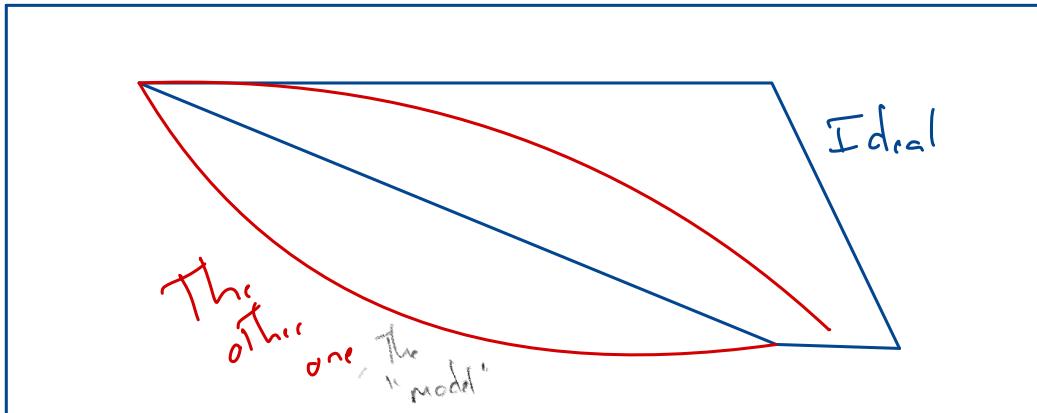
Now let's try to plot all the models together so we can hold the benchmarks together.

```
plt.plot(df_scores.threshold, df_scores['tpr'], label='TPR')
plt.plot(df_scores.threshold, df_scores['fpr'], label='FPR')

# plt.plot(df_rand.threshold, df_rand['tpr'], label='TPR')
# plt.plot(df_rand.threshold, df_rand['fpr'], label='FPR')

plt.plot(df_ideal.threshold, df_ideal['tpr'], label='TPR', color='red')
plt.plot(df_ideal.threshold, df_ideal['fpr'], label='FPR', color='blue')

plt.legend()
```



We see that our TPR is far from the ideal model. We want it to be as close as possible to 1. We also notice that our FPR is significantly different from that of the ideal model. Plotting against the threshold is not always intuitive. For example, in our model, the best threshold is 0.5, as we know from accuracy. However, for the ideal model, as we saw earlier, the best threshold is 0.726. So they have different thresholds. What we can do to better visualize this is to plot FPR against TPR. On the x-axis, we'll have FPR, and on the y-axis, we'll have TPR. To make it easier to understand, we can also add the benchmark lines.

```
plt.figure(figsize=(5,5))

plt.plot(df_scores.fpr, df_scores.tpr, label='model')
plt.plot([0,1], [0,1], label='random')
# plt.plot(df_rand.fpr, df_rand.tpr, label='random')
# plt.plot(df_ideal.fpr, df_ideal.tpr, label='ideal')

plt.xlabel('FPR')
plt.ylabel('TPR')

plt.legend()
```

In the curve of the ideal model, there is one crucial point, often referred to as the ‘north star’ or ideal spot, located in the upper-left corner where TPR is 100% and FPR is 0%. This point represents the optimal performance we aim to achieve with our model. A ROC curve visualizes this by plotting TPR against FPR, and we usually add a diagonal random baseline. Our goal is to make our model’s curve as close as possible to this ideal spot, which means simultaneously being as far away as possible from the random baseline. In essence, if our model closely resembles the random baseline model, it is not performing well.

```
# We can also use the ROC functionality of scikit learn package
from sklearn.metrics import roc_curve

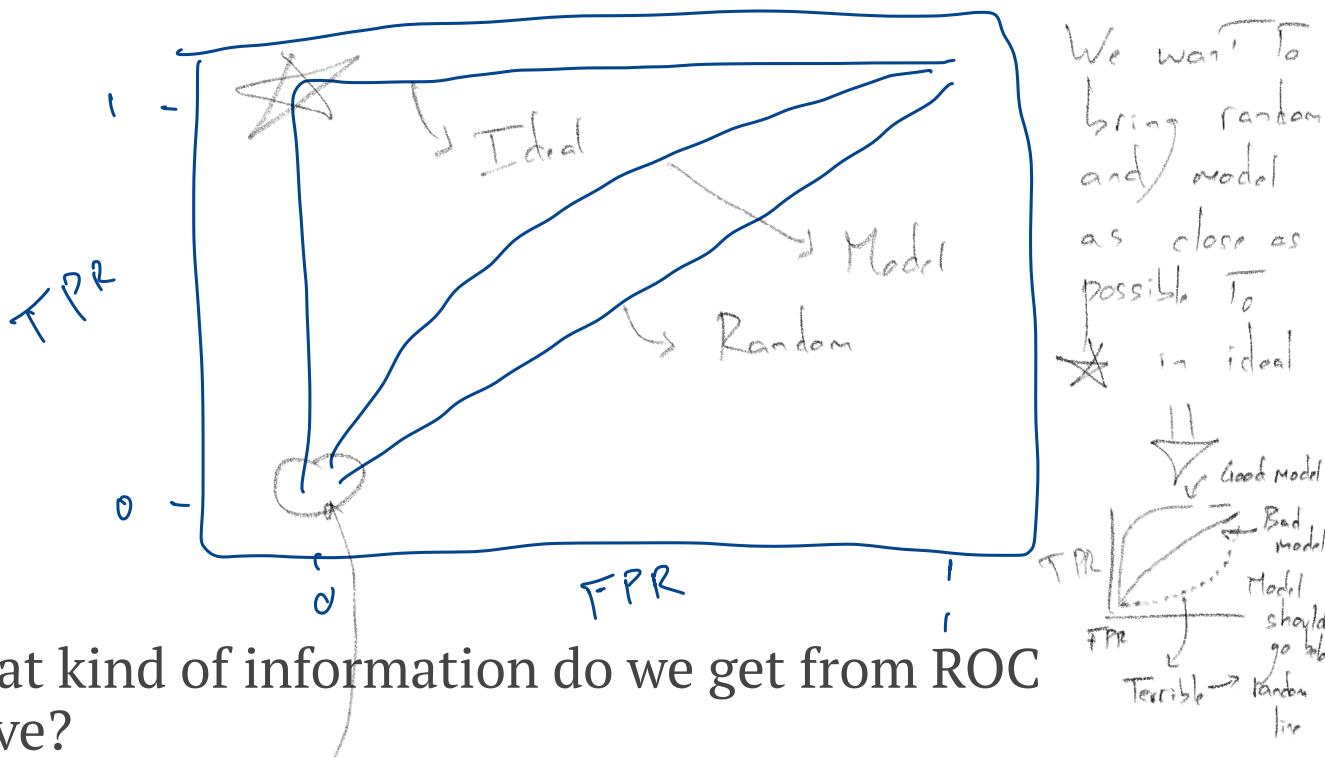
fpr, tpr, thresholds = roc_curve(y_val, y_pred)

plt.figure(figsize=(5,5))

plt.plot(fpr, tpr, label='Model')
plt.plot([0,1], [0,1], label='Random', linestyle='--')

plt.xlabel('FPR')
plt.ylabel('TPR')

plt.legend()
```



What kind of information do we get from ROC curve?

Let's begin in the lower-left corner, where both TPR and FPR are 0. This occurs at higher thresholds like 1.0. In this scenario, we predict that every customer is non-churning, resulting in TPR being 0 since we don't predict anyone as churning. FPR is also 0 because there are no false positives; we only have true negatives (TN).

As we move from the lower left corner, where the threshold starts at 1.0, we eventually reach the upper-right corner with a threshold of 0.0. Here, our model achieves 100% TPR because we predict everyone as churning, enabling us to identify all churning customers. However, we also make many mistakes, incorrectly identifying non-churning customers. Thus, we have $\text{TPR} = \text{FPR} = 100\%$.

When we adjust the threshold, we predict more customers as churning, causing our TPR to increase, but the FPR also increases concurrently.

The ROC curve allows us to observe how the model behaves at different thresholds. Each point on the ROC curve represents TPR and FPR evaluated at a specific threshold. By plotting this curve, we can assess how far the model is from the ideal spot and how far it is from the random baseline. Additionally, the ROC curve is useful for comparing different models, as it's easy to determine which one is superior (a model closer to the ideal spot is better, while one closer to the random baseline is worse).

There is an interesting metric derived from the ROC curve known as AUC, which stands for the area under the curve.

ML Zoomcamp 2023 – Evaluation metrics for classification – Part 6

👤 Peter ⏰ 7. October 2023 📄 Evaluation Metrics, ML-Zoomcamp
🏷️ [AUC](#), [ML Zoomcamp](#)



1. [ROC AUC – Area under the ROC curve](#)

1. [Useful metric](#)
2. [AUC interpretation](#)

ROC AUC – Area under the ROC curve

Useful metric

One way to quantify how close we are to the ideal point is by measuring the area under the ROC curve (AUC). AUC equals 0.5 for a random baseline and 1.0 for an ideal curve.

Therefore, our model's AUC should fall between 0.5 and 1.0. When AUC is less than 0.5, we've made a mistake. AUC = 0.8 is considered good, while 0.9 is great, but 0.6 is considered poor. We can calculate AUC using the scikit-learn package. This package is not specifically for roc curves, this is for any curve. It can calculate area under any curve.

```
from sklearn.metrics import auc
# auc needs values for x-axis and y-axis
auc(fpr, tpr)
# Output: 0.843850505725819
```

```
auc(df_scores.fpr, df_scores.tpr)
# Output: 0.8438732975754537
```

```
auc(df_ideal.fpr, df_ideal.tpr)
# Output: 0.9999430203759136
```

```
fpr, tpr, thresholds = roc_curve(y_val, y_pred)
auc(fpr, tpr)

# Output: 0.843850505725819
```

There is a shortcut in scikit-learn package

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_val, y_pred)

# Output: 0.843850505725819
```

AUC interpretation

AUC tells us the probability that a randomly selected positive example has a score that is higher than a randomly selected negative example.

```
neg = y_pred[y_val == 0]
pos = y_pred[y_val == 1]
```

```
import random
pos_ind = random.randint(0, len(pos) - 1)
neg_ind = random.randint(0, len(neg) - 1)
```

We want to compare the score of this positive example with the score of the negative example.

```
pos[pos_ind] > neg[neg_ind]
# Output: True
```

So, for this random example, this is true. We can do this 100,000 times and evaluate the performance.

```
n = 100000
success = 0

for i in range(n):
    pos_ind = random.randint(0, len(pos) - 1)
    neg_ind = random.randint(0, len(neg) - 1)

    if pos[pos_ind] > neg[neg_ind]:
        success += 1

success / n

# Output: 0.84389
```

That result is quite close to `roc_auc_score(y_val, y_pred) = 0.843850505725819`.

Instead of implementing this manually, we can use NumPy. Be aware that in `np.random.randint(low, high, size, dtype)`, ‘low’ is inclusive, and ‘high’ is exclusive.

```
n = 50000

np.random.seed(1)
pos_ind = np.random.randint(0, len(pos), size=n)
neg_ind = np.random.randint(0, len(neg), size=n)
pos[pos_ind] > neg[neg_ind]
# Output: array([False, True, True, ..., True, True, True])

(pos[pos_ind] > neg[neg_ind]).mean()
# Output: 0.84646
```

Because of this interpretation, AUC is quite popular as a way of measuring the performance of binary classification models. It’s quite intuitive, and we can use it to assess how well our model ranks positive and negative examples and separates positive examples from negative ones.

👤 Peter ⏰ 7. October 2023 📁 Evaluation Metrics, ML-Zoomcamp
🏷️ AUC, ML Zoomcamp

Leave a comment

Write a comment...

Comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Evaluation metrics for classification – Part 7

👤 Peter ⏰ 8. October 2023 📄 Evaluation Metrics, ML-Zoomcamp
🏷️ Cross-Validation, K-Fold, ML Zoomcamp, Parameter Tuning



1. Cross-Validation

1. [Evaluating the same model on different subsets of data](#)
2. [Getting the average prediction and the spread within predictions](#)
3. [Parameter Tuning](#)

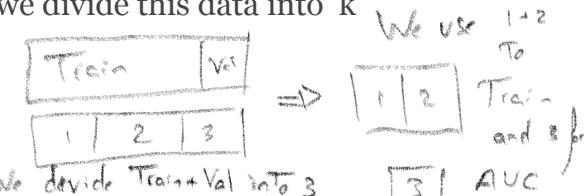
Cross-Validation

Evaluating the same model on different subsets of data

In this article, I'll discuss parameter tuning, which involves selecting the optimal parameter. Typically, we start by splitting our entire dataset into three parts: training, validation, and testing. We utilize the validation dataset to determine the best parameter for the formula $g(x_i)$, essentially finding the optimal parameters for training our model.

For the time being, we set aside the test set and continue working with our combined training and validation dataset – so called full_train. Next, we divide this data into 'k' parts, with 'k' equal to 3.

FULL TRAIN
1 2 3



We can train our model using datasets 1 and 2, using dataset 3 for validation. Subsequently, we calculate the AUC on the validation dataset (3).

Next step is to train another model based on 1 and 3 and validate this model on dataset 2.

We redo this with 1+3 and with 2+3. That's k-fold validation

Again compute the AUC on validation data (2).

TRAIN VAL

1 3 2

Next step is to train another model based on 2 and 3 and validate this model on dataset 1. Again compute the AUC on validation data (1).

TRAIN VAL

2 3 1

After obtaining three AUC values, we calculate their mean and standard deviation. The standard deviation reflects the model's stability and how scores vary across different folds.

K-Fold Cross-Validation is a method for assessing the same model on various subsets of our dataset.

```
def train(df_train, y_train):
    dicts = df_train[categorical + numerical].to_dict(orient='records')
    dv = DictVectorizer(sparse=False)
    X_train = dv.fit_transform(dicts)

    model = LogisticRegression()
    model.fit(X_train, y_train)

    return dv, model
```

We can use This model To Train

```
| dv, model = train(df_train, y_train)
```

Now we create a model for predictions

```
def predict(df, dv, model):
    dicts = df[categorical + numerical].to_dict(orient='records')

    X = dv.fit_transform(dicts)
    y_pred = model.predict_proba(X)[:, 1]

    return y_pred
```

```
y_pred = predict(df_val, dv, model)
y_pred
```

```
# Output: array([0.00899722, 0.20451861, 0.2122173 , ..., 0.136391
```

We now have the 'train' and 'predict' functions in place. Let's proceed to implement K-Fold Cross-Validation.

```

from sklearn.model_selection import KFold

kfold = KFold(n_splits=10, shuffle=True, random_state=1)

kfold.split(df_full_train)
# Output: <generator object _BaseKFold.split at 0x2838baf20>

train_idx, val_idx = next(kfold.split(df_full_train))
len(train_idx), len(val_idx)
# Output: (5070, 564)

len(df_full_train)
# Output: 5634

# We can use iloc to select a part of this dataframe
df_train = df_full_train.iloc[train_idx]
df_val = df_full_train.iloc[val_idx]

```

The following code snippet demonstrates the implementation for 10 folds. Finally, we use the 'roc_auc_score' function to calculate and output the corresponding score for each fold.

```

from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score

kfold = KFold(n_splits=10, shuffle=True, random_state=1)
scores = []

for train_idx, val_idx in kfold.split(df_full_train):
    df_train = df_full_train.iloc[train_idx]
    df_val = df_full_train.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    scores.append(auc)

scores
# Output:
# [0.8479398247539081,
# 0.8410581683168317,
# 0.8557214756739697,
# 0.8333552794008724,
# 0.8262717121588089,
# 0.8342657342657342,
# 0.8412569195701727,
# 0.8186669829222013,
# 0.8452349192233585,
# 0.8621054754462034]

```

Same implementation but this time with tqdm package.

↳ Which Tells you how long each iteration takes and which one you are running

```

from sklearn.model_selection import KFold
!pip3 install tqdm
from tqdm.auto import tqdm

kfold = KFold(n_splits=10, shuffle=True, random_state=1)
scores = []

for train_idx, val_idx in tqdm(kfold.split(df_full_train)):
    df_train = df_full_train.iloc[train_idx]
    df_val = df_full_train.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    scores.append(auc)

scores
# Output:
# [0.8479398247539081,
# 0.8410581683168317,
# 0.8557214756739697,
# 0.8333552794008724,
# 0.8262717121588089,
# 0.8342657342657342,
# 0.8412569195701727,
# 0.8186669829222013,
# 0.8452349192233585,
# 0.8621054754462034]

```

Getting the average prediction and the spread within predictions

We can utilize the scores generated to compute the average score across the 10 folds, which is 84.1%, with a standard deviation of 0.012.

```

print('%.3f +- %.3f' % (np.mean(scores), np.std(scores)))
# Output: 0.841 +- 0.012

```

Parameter Tuning

We discussed parameter tuning, particularly the ‘C’ parameter in our LogisticRegression model, which serves as the regularization parameter with a default value of 1.0. We can include this ‘C’ parameter in our ‘train’ function. If ‘C’ is set to a very small value, it implies strong regularization. Additionally, we can address an annoying message by setting the ‘max_iter’ value to 1000.

```

def train(df_train, y_train, C=1.0):
    dicts = df_train[categorical + numerical].to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    X_train = dv.fit_transform(dicts)

    model = LogisticRegression(C=C, max_iter=1000)
    model.fit(X_train, y_train)

    return dv, model

```

```
| dv, model = train(df_train, y_train, C=0.001)
```

We can iterate over various values for ‘C,’ keeping in mind that ‘C’ cannot be set to 0.0, as it would result in an ‘InvalidParameterError.’ The ‘C’ parameter for LogisticRegression must be a float within the range (0.0, inf], so we need to avoid using 0.0.

```

from sklearn.model_selection import KFold

kfold = KFold(n_splits=10, shuffle=True, random_state=1)

for C in [0.001, 0.01, 0.1, 0.5, 1, 5, 10]:
    scores = []

    for train_idx, val_idx in kfold.split(df_full_train):
        df_train = df_full_train.iloc[train_idx]
        df_val = df_full_train.iloc[val_idx]

        y_train = df_train.churn.values
        y_val = df_val.churn.values

        dv, model = train(df_train, y_train, C=C)
        y_pred = predict(df_val, dv, model)

        auc = roc_auc_score(y_val, y_pred)
        scores.append(auc)

    print('C=%s %.3f +- %.3f' % (C, np.mean(scores), np.std(score)))

# Output:
# C=0.001 0.826 +- 0.012
# C=0.01 0.840 +- 0.012
# C=0.1 0.841 +- 0.011
# C=0.5 0.841 +- 0.011
# C=1 0.840 +- 0.012
# C=5 0.841 +- 0.012
# C=10 0.841 +- 0.012

```

We can implement the same procedure using the ‘tqdm’ package, which provides a more visually appealing output.

```

from sklearn.model_selection import KFold
n_splits = 5

for C in tqdm([0.001, 0.01, 0.1, 0.5, 1, 5, 10]):
    scores = []

    kfold = KFold(n_splits=n_splits, shuffle=True, random_state=1)

    for train_idx, val_idx in kfold.split(df_full_train):
        df_train = df_full_train.iloc[train_idx]
        df_val = df_full_train.iloc[val_idx]

        y_train = df_train.churn.values
        y_val = df_val.churn.values

        dv, model = train(df_train, y_train, C=C)
        y_pred = predict(df_val, dv, model)

        auc = roc_auc_score(y_val, y_pred)
        scores.append(auc)

    print('C=%s %.3f +- %.3f' % (C, np.mean(scores), np.std(scores)))

# Output:
# 14%|██████| 1/7 [00:01<00:06, 1.03s/it]
# C=0.001 0.825 +- 0.009
# 29%|█████| 2/7 [00:02<00:05, 1.07s/it]
# C=0.01 0.840 +- 0.009
# 43%|█████| 3/7 [00:03<00:04, 1.06s/it]
# C=0.1 0.840 +- 0.008
# 57%|█████| 4/7 [00:04<00:03, 1.08s/it]
# C=0.5 0.841 +- 0.006
# 71%|█████| 5/7 [00:05<00:02, 1.13s/it]
# C=1 0.841 +- 0.008
# 86%|█████| 6/7 [00:06<00:01, 1.15s/it]
# C=5 0.841 +- 0.007
# 100%|████████| 7/7 [00:07<00:00, 1.10s/it]
# C=10 0.841 +- 0.008

```

Afterward, we aim to train our final model using the entire training dataset (df_full_train) and then validate it using the test dataset.

```

dv, model = train(df_full_train, df_full_train.churn.values, C=1.)
y_pred = predict(df_test, dv, model)

auc = roc_auc_score(y_test, y_pred)
auc
# Output: 0.8572386167896259

```

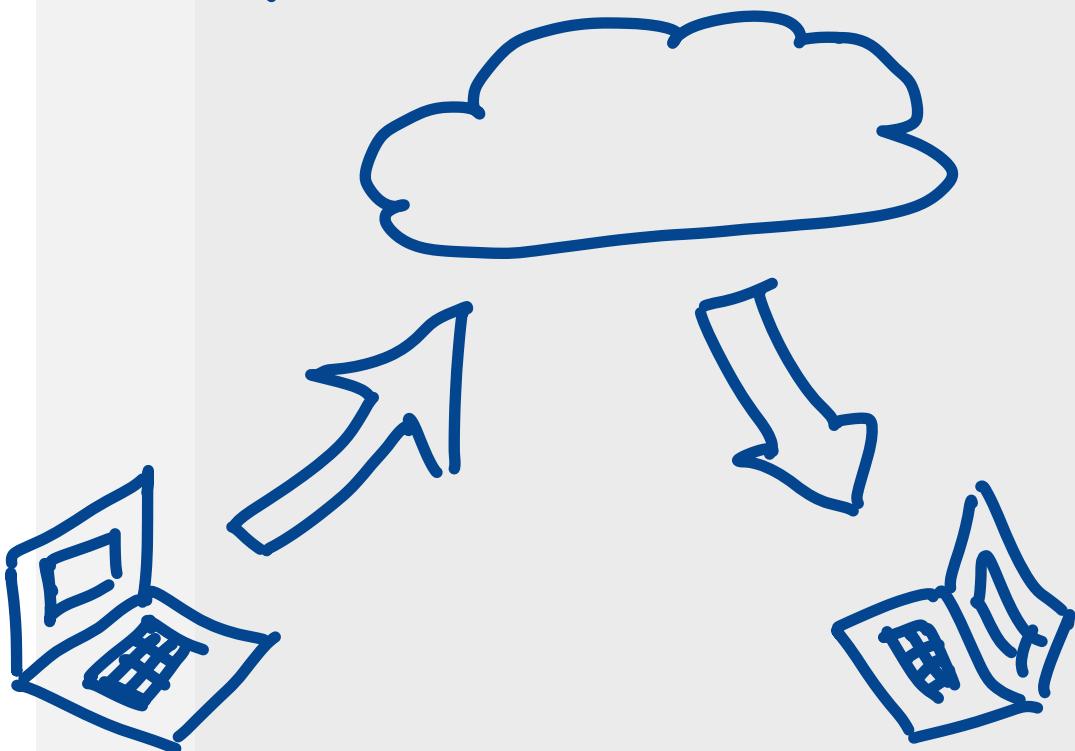
We observe that the AUC is slightly better than what we observed during k-fold cross-validation, though not significantly higher. This is expected when the difference is small.

In terms of when to use cross-validation versus traditional hold-out validation, for larger datasets, standard hold-out validation is often sufficient. However, if your dataset is smaller or you require insight into the model's stability and variation across folds, then

DEPLOYING

ML

MODELS



SUMMARY

This chapter is about how to put your model into the cloud
so that someone can use it

ML Zoomcamp 2023 – Deploying Machine Learning Models– Part 1

👤 Peter ⏰ 9. October 2023 📁 ML-Zoomcamp, Model Deployment
🏷️ Deployment, ML Zoomcamp



Overview

In this chapter, we will delve into the deployment process of our machine learning model. We'll start by taking the Jupyter notebook where our model resides and save it to a file, which we'll call 'model.bin.' The next step is to load this model from a different process, a web service aptly named the 'churn service,' which houses this model.

The primary purpose of this service is to identify churning customers. To put it into context, imagine we also have another service called the 'marketing service,' which holds comprehensive customer information. For a specific customer, we want to determine their churning probability. The marketing service sends a request to the churn service, providing the necessary customer information. The churn service processes this request and returns the prediction. Subsequently, based on this prediction, the marketing service can decide whether to send a promotional email with a discount offer.

To transform the model into a web service, we will leverage Flask, a powerful framework for creating web services. Moreover, we aim to isolate the dependencies for this service to prevent interference with other services on our machine. To achieve this isolation, we will create a dedicated environment for Python dependencies using pipenv. Following that, we'll add another layer, which will comprise system dependencies, and for this purpose, we'll employ Docker.

Once the local setup is complete, the final step is deploying our service to the cloud. We will containerize the application and deploy it to AWS Elastic Beanstalk, taking advantage of the cloud's scalability and reliability.

ML Zoomcamp 2023 – Deploying Machine Learning Models– Part 2

👤 Peter ⏰ 10. October 2023 📁 ML-Zoomcamp, Model Deployment
🏷️ K-Fold, ML Zoomcamp, Pickle



1. [Saving and loading the model](#)
 1. [Saving the model to pickle](#)
 2. [Loading the model with Pickle](#)
 3. [Turning our notebook into a Python script](#)

Saving and loading the model

Let's take a moment to recap what we've accomplished so far. Before we can save a model, the crucial first step is training it. I've extensively covered model training in previous articles, where we explored various techniques, including K-Fold cross-validation. Below, I've included all the code necessary for model training. While there's nothing new here, it serves as a useful recap of our progress.

The first code snippet contains all the necessary imports.

```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold

from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
```

The next snippet is about data preparation, where we need to read the csv file, make the column names more homogenous, and deal with categorical and numerical values.

```

# Data preparation

df = pd.read_csv('data-week-3.csv')

df.columns = df.columns.str.lower().str.replace(' ', '_')

categorical_columns = list(df.dtypes[df.dtypes == 'object'].

for c in categorical_columns:
    df[c] = df[c].str.lower().str.replace(' ', '_')

df.totalcharges = pd.to_numeric(df.totalcharges, errors='coerce')
df.totalcharges = df.totalcharges.fillna(0)

df.churn = (df.churn == 'yes').astype(int)

```

The next snippet is about data splitting. Again we use the `train_test_split` function to divide the dataset in full_train and test data.

```

# Data splitting

df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=1)

```

In the next snippet you see what are the numerical column names and what are the categorical column names.

```

numerical = ['tenure', 'monthlycharges', 'totalcharges']

categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
               'phoneservice', 'multiplelines', 'internetservice',
               'onlinesecurity', 'onlinebackup', 'deviceprotection',
               'streamingtv', 'streamingmovies', 'contract', 'paperlessbilling',
               'paymentmethod']

```

The next snippet is about the `train` function. It has three arguments – the training dataframe and the target values `y_train`, and the third argument is `C` which is a `LogisticRegression` parameter for our model. First step here is to create dictionaries from the categorical columns, remember the numerical columns are ignored here. Next we create a `DictVectorizer` instance which we need to use `fit_transform` function on the dictionaries. So we get the `X_train`. Then we create our model which is a logistic regression model, that we can use for training (`fit` function) based on the training data (`X_train` and `y_train`). To apply the model later we need to return the `DictVectorizer` and the model as well.

```

def train(df_train, y_train, C=1.0):
    dicts = df_train[categorical + numerical].to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    X_train = dv.fit_transform(dicts)

    model = LogisticRegression(C=C, max_iter=1000)
    model.fit(X_train, y_train)

    return dv, model

```

As I just mentioned in the paragraph before to use the model we need also the DictVectorizer. Both are arguments for the predict function which is show in the next snippet. Besides both arguments you also need a dataframe where we can provide a prediction for. First step here is the same like in training function, we need to get the dictionaries. This can be transformed by the DictVectorizer so we get the X, what we need to make a prediction on. What we return here is the predicted probability for churning.

```

def predict(df, dv, model):
    dicts = df[categorical + numerical].to_dict(orient='records')

    X = dv.transform(dicts)
    y_pred = model.predict_proba(X)[:, 1]

    return y_pred

```

Next snippet is to setup two parameters. The first one is the C value for the Logistic Regression model, and the 'n_splits' parameter tells us how many splits we're going to use in K-Fold cross-validation. Here, we're using 5 splits.

```

C = 1.0
n_splits = 5

```

Next snippet shows the implemented K-Fold cross validation, where we use the parameters from the last snippet. The for loop loops over all folds and does a training for each. After that we calculate the roc_auc_score and collect the values for each fold. At the end the mean score and the standard deviation for all folds are printed.

```

kfold = KFold(n_splits=n_splits, shuffle=True, random_state=1)

scores = []

for train_idx, val_idx in kfold.split(df_full_train):
    df_train = df_full_train.iloc[train_idx]
    df_val = df_full_train.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train, C=C)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    scores.append(auc)

print('C=%s %.3f +- %.3f' % (C, np.mean(scores), np.std(scores)))
# Output: C=1.0 0.841 +- 0.008

```

The last snippet doesn't show the score for each fold separately but here you can see each value.

```

scores

# Output:
# [0.8438508214866044,
# 0.8450763971659383,
# 0.8327513546056594,
# 0.8301724275756219,
# 0.8521461516739357]

```

Last step is to train the final model based on the full_train data. The steps here are similar to the steps mentioned before. First is model training, then predicting the test data, and lastly calculate the roc_auc_score. We see a value of 85.7% which is a bit higher than the average of the k-folds. But there is not a big difference.

```

dv, model = train(df_full_train, df_full_train.churn.values, C=1.0)
y_pred = predict(df_test, dv, model)
y_test = df_test.churn.values

auc = roc_auc_score(y_test, y_pred)
auc

# Output: 0.8572386167896259

```

Until now the model still lives in our Jupyter notebook. So we cannot just take this model and put it in a web service. Remember we want to put this model in a web service, that the marketing service can use it to score the customers. That means now we need to save this model in order to be able to load it later.

Saving the model to pickle

For saving the model we'll use pickle, what is a built in library for saving Python objects.

| **import** pickle

First, we need to name our model file before we can write it to a file. The following snippet demonstrates two ways of naming the file.

```
output_file = 'model_C=%s.bin' % C  
output_file  
# Output: 'model_C=1.0.bin'  
  
output_file = f'model_C={C}.bin'  
output_file  
# Output: 'model_C=1.0.bin'
```

Now we want to create a file with that file name. 'wb' means Write Binary. We need to save DictVectorizer and the model as well, because with just the model we'll not be able to translate a customer into a feature matrix. Closing the file is crucial. Otherwise, we cannot be certain whether this file truly contains the content.

```
f_out = open(output_file, 'wb')  
pickle.dump((dv, model), f_out)  
f_out.close()
```

Saves
The
model

Here we specify what we want to do with this file. We use wb because we won't write text there, we will write binary data there.

To avoid accidentally forgetting to close the file, we can use the 'with' statement, which ensures that the file is closed automatically. Everything we do inside the 'with' statement keeps the file open. However, once we exit this statement, the file is automatically closed.

```
with open(output_file, 'wb') as f_out:  
    pickle.dump((dv, model), f_out)
```

Loading the model with Pickle

For loading the model we'll also use pickle.

Notice that even if you don't need to import scikit-learn you still need to have it in your computer

| **import** pickle

```
model_file = 'model_C=1.0.bin'
```

We also utilize the 'with' statement for loading the model. Here, 'rb' denotes Read Binary. We employ the 'load' function from pickle, which returns both the DictVectorizer and the model.

```
with open(model_file, 'rb') as f_in:  
    dv, model = pickle.load(f_in)  
  
dv, model  
# Output: (DictVectorizer(sparse=False), LogisticRegression(m
```

After loading the model, let's use it to score one sample customer.

```
customer = {  
    'gender': 'female',  
    'seniorcitizen': 0,  
    'partner': 'yes',  
    'dependents': 'no',  
    'phoneservice': 'no',  
    'multiplelines': 'no_phone_service',  
    'internetservice': 'dsl',  
    'onlinesecurity': 'no',  
    'onlinebackup': 'yes',  
    'deviceprotection': 'no',  
    'techsupport': 'no',  
    'streamingtv': 'no',  
    'streamingmovies': 'no',  
    'contract': 'month-to-month',  
    'paperlessbilling': 'yes',  
    'paymentmethod': 'electronic_check',  
    'tenure': 1,  
    'monthlycharges': 29.85,  
    'totalcharges': 29.85  
}
```

Before we can apply the predict function to this customer we need to turn it into a feature matrix. The DictVectorizer expects a list of dictionaries, that's why we create a list with one customer.

```
X = dv.transform([customer])  
X  
  
# Output:  
# array([[ 1. ,  0. ,  0. ,  1. ,  0. ,  1. ,  0. ,  0. ,  0.  
#        ,  0. ,  1. ,  0. ,  0. ,  29.85,  0. ,  1. ,  0.  
#        ,  0. ,  1. ,  1. ,  0. ,  0. ,  0. ,  1. ,  0.  
#        ,  0. ,  0. ,  1. ,  0. ,  0. ,  1. ,  0. ,  0. ,  1.  
#        ,  0. ,  1. ,  0. ,  0. ,  1. ,  0. ,  0. ,  0. ,  1.
```

We use predict_proba function to get the probability that this particular customer is going to churn. We're interested in the second element, so we need to set the row=0 and column=1.

```
model.predict_proba(X)  
# Output: array([[0.36364158, 0.63635842]])  
  
model.predict_proba(X)[0,1]  
# Output: 0.6363584152758612
```

Turning our notebook into a Python script

We can turn the Jupyter Notebook code into a Python file. One easy way of doing this is click on “File” -> “Download as” and then “Python (.py)”

👤 [Peter](#) ⏰ [10. October 2023](#) 📁 [ML-Zoomcamp, Model Deployment](#)
◆ [K-Fold](#), [ML Zoomcamp](#), [Pickle](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Deploying Machine Learning Models– Part 3

👤 Peter

⌚ 11. October 2023

📁 ML-Zoomcamp, Model Deployment

🏷️ [Flask](#), [ML Zoomcamp](#)



1. [Introduction to Flask](#)

1. [Writing a simple ping/pong function](#)
2. [Installing Flask](#)
3. [Writing a simple ping/pong app](#)
4. [Querying the simple app with 'curl' and browser](#)

Introduction to Flask

A **web service** is a method for communicating between two devices over a network. So let's say we have our web service and a user who wants to make a request. So the user sends the request with some information. The request has some parameters, then the user gets back result with the answer of this request.

We use **Flask** for implementing the web service and it takes care of all the internals.

Writing a simple ping/pong function

In a simple sample implementation, we want to create a ping-pong service. This means we send a “ping” request to a web service, and it responds with “pong.” To achieve this, we create a file named ping.py with the content of the next snippet:

```
| def ping():  
|     return "PONG"
```

To easily test whether it is working or not, you can follow these steps:

1. Open a console.
2. Launch an interactive Python session with `ipython`.
3. Import the `ping` module by typing `import ping`.
4. Call the `ping.ping()` function.

When you execute these steps, it should return “PONG.”

Installing Flask

The next step is to turn this simple function into a web service. To achieve this, you need to install Flask, a Python web framework. You can install Flask using pip. Here are the steps to install Flask:

1. Open your command prompt or terminal.
2. Run the following command to install Flask:

```
pip install flask
```

This will download and install Flask along with its dependencies.

Writing a simple ping/pong app

Once Flask is installed, we can proceed to create our web service using Flask and integrate our “ping-pong” functionality into it.

We employ a **decorator** for our definition. A decorator is a mechanism for adding additional functionality to our functions. This added functionality will enable us to transform this function into a web service.

In the snippet below, we begin by importing Flask and creating an app with the name ‘ping’. We then use the `@app.route('/ping')` decorator to designate the `ping` function as a web service accessible at the “/ping” route. In this context, ‘route’ specifies the address at which the function will be accessible.

The ‘method’ specifies how we can access this address. When we visit the website in a browser, the browser sends a GET request. In this case, we want to access the function using the GET method, and it will be located at the ‘/ping’ address.

Finally, we start the Flask application with `app.run()`. It’s important to note that `app.run()` should be placed inside the main block (`if __name__ == '__main__'`) to ensure it is executed only when we run the script using “`python ping.py`” in the top-level script environment.

```
from flask import Flask

app = Flask('ping')

@app.route('/ping', methods=['GET'])
def ping():
    return "PONG"

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Querying the simple app with ‘curl’ and browser

Now, when you run the ‘ping.py’ script that starts the web service and open a web browser or use a tool like ‘curl’ to access <http://localhost:9696/ping>, you should receive “PONG” as the response. This effectively transforms your ‘ping’ function into a simple web service.

‘Curl’ is a specialized command-line utility for communicating with web services. To use it, simply open a new console and type the following command:

```
curl http://0.0.0.0:9696/ping
```

Alternatively, you can access it via a web browser by opening a browser and typing the following URL:

```
http://localhost:9696/ping
```

In both cases, the web service will respond with “PONG”.

👤 Peter ⏰ 11. October 2023 📁 ML-Zoomcamp, Model Deployment
🏷️ [Flask](#), [ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Deploying Machine Learning Models– Part 4

👤 Peter ⏰ 12. October 2023 📁 ML-Zoomcamp, Model Deployment
🏷️ Flask, gunicorn, ML Zoomcamp, waitress



1. [Serving the churn model with Flask](#)

1. [Wrapping the predict script into a Flask app](#)
2. [Querying it with 'requests'](#)
3. [Preparing for production: gunicorn](#)
4. [Running it on Windows with waitress](#)

Serving the churn model with Flask

Now we want to serve the churn model with Flask.

Wrapping the predict script into a Flask app

First, we need to convert all the code from Jupyter Notebook into .py scripts. The following snippet displays the code for “train.py”. The prediction section is separated from the training part since that will be the responsibility of the web service.

```
import pickle
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score

# Setting model parameters
C = 1.0
```

```

n_splits = 5
output_file = f'model_C={C}.bin'

# Data preparation
df = pd.read_csv('data-week-3.csv')
df.columns = df.columns.str.lower().str.replace(' ', '_')
categorical_columns = list(df.dtypes[df.dtypes == 'object'])

for c in categorical_columns:
    df[c] = df[c].str.lower().str.replace(' ', '_')

df.totalcharges = pd.to_numeric(df.totalcharges, errors='coerce')
df.totalcharges = df.totalcharges.fillna(0)

df.churn = (df.churn == 'yes').astype(int)

# Data splitting
df_full_train, df_test = train_test_split(df, test_size=0.2)
numerical = ['tenure', 'monthlycharges', 'totalcharges']
categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
               'phoneservice', 'multiplelines', 'internetservice',
               'onlinesecurity', 'onlinebackup', 'deviceprotection',
               'streamingtv', 'streamingmovies', 'contract', 'paperlessbilling',
               'paymentmethod']

# Training
def train(df_train, y_train, C=1.0):
    dicts = df_train[categorical + numerical].to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    X_train = dv.fit_transform(dicts)

    model = LogisticRegression(C=C, max_iter=1000)
    model.fit(X_train, y_train)

    return dv, model

def predict(df, dv, model):
    dicts = df[categorical + numerical].to_dict(orient='records')

    X = dv.fit_transform(dicts)
    y_pred = model.predict_proba(X)[:, 1]

    return y_pred

# Validation
print(f'doing validation with C={C}')

kfold = KFold(n_splits=n_splits, shuffle=True, random_state=42)
scores = []
fold = 0

for train_idx, val_idx in kfold.split(df_full_train):

```

```

df_train = df_full_train.iloc[train_idx]
df_val = df_full_train.iloc[val_idx]

y_train = df_train.churn.values
y_val = df_val.churn.values

dv, model = train(df_train, y_train, C=C)
y_pred = predict(df_val, dv, model)

auc = roc_auc_score(y_val, y_pred)
scores.append(auc)

print(f'auc on fold {fold} is {auc}')
fold += 1

print('validation result:')
print(f'C={C:.3f} +- {std:.3f}' % (C, np.mean(scores), np.std(scores)))
# Output: C=1.0 0.841 +- 0.008

# Train the final model
print('train the final model')

dv, model = train(df_full_train, df_full_train.churn.values)
y_pred = predict(df_test, dv, model)
y_test = df_test.churn.values

auc = roc_auc_score(y_test, y_pred)

print(f'auc={auc}')
# Output: 0.8572386167896259

# Saving the model with Pickle
with open(output_file, 'wb') as f_out:
    pickle.dump((dv, model), f_out)

print(f'the model is saved to {output_file}')

```

The following snippet covers to the prediction part (predict.py). There's a key distinction this time: we won't be using the GET method as we did in our previous simple sample from the last article. Instead, we'll use the POST method since we need to send information to the web service.

```

import pickle
from flask import Flask
from flask import request
from flask import jsonify

model_file = 'model_C=1.0.bin'

with open(model_file, 'rb') as f_in:
    dv, model = pickle.load(f_in)

app = Flask('churn')

@app.route('/predict', methods=['POST'])
def predict():
    # json = Python dictionary → To expand on This: you get the prediction in JSON format when a JSON is like a dictionary, just like a Python
    customer = request.get_json() → Transforms the JSON into a Python dict

    X = dv.transform([customer])
    model.predict_proba(X)
    y_pred = model.predict_proba(X)[0, 1]
    churn = y_pred >= 0.5

    result = { → This is what you give to the prediction
        # the next line raises an error so we need to change
        #'churn_probability': y_pred, by making it a float
        'churn_probability': float(y_pred),
        # the next line raises an error so we need to change
        #'churn': churn
        'churn': bool(churn)
    }

    return jsonify(result) → Returns json to format
if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=9696)

```

This time we cannot test it in the browser, because browser sends an GET request.

Otherwise we'll get an error:

127.0.0.1 -- [06/Oct/2023 17:23:04] "GET /predict HTTP/1.1" 405 -

Querying it with 'requests'

To test our implementation, we use “predict-test.py,” which is implemented in the following snippet. In this script, we use the `requests` library to send our request to the web service. As previously mentioned, we use the POST method. The POST method of `requests` takes two arguments. ‘url’ points to the web service, and ‘json’ represents our customer information, which needs to be in JSON format. Depending on the result, the script will determine whether to send a promotion email or not.

```

import requests

url = 'http://localhost:9696/predict'

customer_id = 'xyz-123'
customer = {
    "gender": "female",
    "seniorcitizen": 0,
    "partner": "yes",
    "dependents": "no",
    "phoneservice": "no",
    "multiplelines": "no_phone_service",
    "internetservice": "dsl",
    "onlinesecurity": "no",
    "onlinebackup": "yes",
    "deviceprotection": "no",
    "techsupport": "no",
    "streamingtv": "no",
    "streamingmovies": "no",
    "contract": "month-to-month",
    "paperlessbilling": "yes",
    "paymentmethod": "electronic_check",
    "tenure": 1,
    "monthlycharges": 29.85,
    "totalcharges": 29.85
}

response = requests.post(url, json=customer).json()
print(response)

if response['churn'] == True:
    print('sending promo email to %s' % customer_id)
else:
    print('not sending promo email to %s' % customer_id)

```

Preparing for production: gunicorn

Flask is not recommended for use in production, but there are several alternatives, such as Gunicorn, which is suitable for Linux, Unix, and MacOS. You can also use Gunicorn in the Windows Subsystem for Linux (WSL). To use Gunicorn on the mentioned operating systems, it needs to be installed first with the command ‘`pip install gunicorn`’. However, please note that Gunicorn is not compatible with Windows.

After installing Gunicorn, you can start the web service by running the command ‘`gunicorn --bind 0.0.0.0:9696 predict:app`’. This command binds the web service to port 9696 on localhost and uses the ‘predict’ function from our app which we’ve implemented in ‘`predict.py`’.

Now, we can send a request to our web service using the command ‘`python predict-test.py`’. The web service responds to this request, and the evaluation result looks like:

```
{'churn': True, 'churn_probability': 0.6363584152721401}
sending promo email to xyz-123
```

Running it on Windows with waitress

As mentioned before we cannot use gunicorn on Windows. But there is an alternative called **waitress** which is similar to gunicorn. Before the first use it also needs to be installed with ‘`pip install waitress`’.

After installing Waitress, you can start the web service by running the command ‘`waitress --listen=0.0.0.0:9696 predict:app`’. This command binds the web service to port 9696 on localhost and uses the ‘predict’ function from our app which we’ve implemented in ‘`predict.py`’.

Now, we can send the request to the web service using the command ‘`python predict-test.py`’. The web service responds to this request, and the evaluation result looks like:

```
{'churn': True, 'churn_probability': 0.6363584152721401}  
sending promo email to xyz-123
```

👤 [Peter](#) ⏰ [12. October 2023](#) 📁 [ML-Zoomcamp, Model Deployment](#)
🏷️ [Flask](#), [gunicorn](#), [ML Zoomcamp](#), [waitress](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Deploying Machine Learning Models– Part 5

👤 Peter

⌚ 13. October 2023

📁 ML-Zoomcamp, Model Deployment

🏷️ [ML Zoomcamp, pipenv](#)



1. [Python virtual environment: Pipenv](#)

1. [Dependency and environment management](#)

1. [Idea of virtual environments](#)
2. [Virtual Environments](#)

2. [Installing Pipenv & Installing libraries with Pipenv](#)

3. [Running things with Pipenv](#)

Python virtual environment: Pipenv

Dependency and environment management

When you run ‘pip install scikit-learn,’ it searches in the directories listed in the \$PATH variable, such as ~/anaconda3/bin/ in this case. Inside this folder, you have ‘pip,’ ‘python,’ and other packages. The ‘pip’ from this folder is used, and it connects to the Python Package Index (PyPI) at pypi.org. It then installs the latest version of the package into the directory specified in the \$PATH variable.

Now, let’s consider a scenario where you have two applications:

1. Churn service, which relies on scikit-learn==0.24.2.
2. Lead scoring service, which depends on scikit-learn==1.0.

In this case, we have two different versions of scikit-learn in use, and they are isolated from each other. To manage this situation effectively, you can use different virtual environments.

Idea of virtual environments

We have two separate environments, each with its own Python installation for the services:

1. Churn service (Python resides in ~/venv/churn/bin/python)
2. Lead Scoring Service (Python resides in ~/venv/lead/bin/python)

Now, when you execute ‘pip install scikit-learn,’ it installs the package into its respective Python location. This approach ensures that you won’t encounter conflicts when using different versions of scikit-learn for the two services.

Virtual Environments

1. **Virtual Environments:** Virtual environments, often referred to as “venv” or “virtualenv,” allow you to create isolated Python environments for your projects. This separation helps manage dependencies and avoids conflicts between packages used in different projects.
2. **Conda:** Conda is an open-source package management and environment management system that can handle both Python and non-Python packages. It’s commonly used in data science and scientific computing for managing complex environments.
3. **Poetry:** Poetry is a dependency management and packaging tool for Python. It simplifies the process of managing project dependencies, packaging, and publishing Python packages. Poetry aims to provide a consistent and user-friendly approach to Python development.
4. **Pipenv:** Pipenv is another tool for managing dependencies and virtual environments. It combines pip (Python’s package installer) and virtualenv into one tool, making it easier to create and manage virtual environments and package dependencies for your projects.

Each of these tools has its strengths and may be more suitable for specific use cases. The choice of which one to use depends on your project’s requirements and your personal preferences.

Installing Pipenv & Installing libraries with Pipenv

To install Pipenv and libraries using Pipenv, follow these steps:

1. Install Pipenv by running the command:
`pip install pipenv`
2. Install the desired libraries and specify their versions using the `pipenv install` command. For example:
`pipenv install numpy scikit-learn==0.24.2 flask gunicorn`
This will create two files, `Pipfile` and `Pipfile.lock`, in the local directory where this command is executed.
3. To use this environment on a different computer, navigate to the folder containing the `Pipfile` and `Pipfile.lock` files and run the following command:
`pipenv install`
This will recreate the virtual environment and install the specified libraries along with their versions, ensuring consistency across different machines.

Running things with Pipenv

To run things using Pipenv, you can follow these steps:

1. To activate the virtual environment for your project, use the command:

```
pipenv shell
```

Running “pipenv shell” allows you to use all the packages within the environment until you exit the environment with Ctrl+C.

Alternatively, you can run a specific command within the virtual environment using:

```
pipenv run <your_command_here>
```

When you launch `pipenv shell`, it will display the folder where the virtual environment is stored.

2. You can confirm the location of a specific command within the virtual environment by running:

```
which gunicorn
```

This will show you the path to the version of the command installed in the virtual environment.

3. To retrieve the path to the virtual environment itself, you can use:

```
echo $PATH
```

4. For running a specific command within the environment, like starting your web service with Gunicorn, you can use:

```
pipenv run gunicorn --bind 0.0.0.0:9696 predict:app
```

This allows you to use packages and commands within the virtual environment.

While Pipenv helps manage different Python package versions, it doesn’t address differences in system library versions installed via apt-get. For that level of isolation, you might need additional tools or configurations.

 [Peter](#)  [13. October 2023](#)  [ML-Zoomcamp, Model Deployment](#)
 [ML Zoomcamp, pipenv](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Deploying Machine Learning Models– Part 6

👤 Peter ⏰ 14. October 2023 📁 ML-Zoomcamp, Model Deployment
🏷️ Container, Docker, ML Zoomcamp



1. [Environment management: Docker](#)

1. [Why do we need Docker?](#)
2. [Running a Python image with Docker](#)
3. [Dockerfile – general information](#)
4. [Dockerfile for churn app](#)
5. [Building a Docker image](#)
6. [Running a Docker image](#)

Environment management: Docker

Why do we need Docker?

Docker is a powerful tool that addresses many challenges in modern software development and deployment. It allows us to isolate entire applications from the rest of the system's processes and dependencies, providing a level of encapsulation and portability that is invaluable in today's computing landscape.

One of Docker's core advantages is its ability to create **containers**, which are lightweight, portable environments. These containers can include not only the application itself but also all the necessary libraries, dependencies, and even different versions of operating systems. This isolation ensures that a single system can host multiple containers, each running its own set of services and applications.

By placing different services in separate containers, we achieve complete **isolation** between them. These services are unaware of each other's presence, as each one operates

in its own container, believing it is the sole process running on the system. This isolation greatly simplifies software management, maintenance, and troubleshooting.

Moreover, Docker allows us to define specific environments with precision. For example, we can create a container with Ubuntu 18.04, configure it with different system libraries, and specify particular Python versions. Docker's flexibility in environment definition ensures consistency across development, testing, and production stages.

Perhaps the most compelling feature of Docker is its portability. Once we've encapsulated a service within a Docker container, we can easily move it between different environments, from a local development machine to the Cloud or any other deployment target. This consistent packaging and deployment process save time and reduce potential deployment issues.

In summary, Docker revolutionizes the way we manage and deploy applications by providing isolation, flexibility, and portability. It has become an essential tool in modern software development, empowering developers to build, test, and deploy software reliably across various platforms and environments.

Running a Python image with Docker

To get started with running a Python image in Docker, you can search for available Python images on [Docker Hub](#) by navigating to 'hub.docker.com/_/python'.

Once there, you'll find various tags for Python images that you can choose from. Let's select a specific version, such as '3.8.12-slim'.

To run the chosen Python image, you can use the following command:

```
docker run -it --rm python:3.8.12-slim
```

Here's what each part of the command does:

- **-it**: This option allows you to interact with the container's terminal.
- **--rm**: This flag indicates that you want to remove the container once you're done with it.

If the specified Python version isn't already present on your system, Docker will automatically download it from the internet. Running the command will grant you access to the Python terminal within the container.

Additionally, you can access the container's terminal directly by overwriting the entry point. The entry point is the default command that gets executed when you run the container. Here's how you can access the container's terminal:

```
docker run -it --rm --entrypoint=bash python:3.8.12-slim
```

Running this command will open a terminal within the container, allowing you to interact

with it directly.

Dockerfile – general information

A Dockerfile is a script-like text file used in Docker to build container images. It contains a series of instructions that Docker follows to create a reproducible and self-contained environment for running applications.

Here are some key aspects of a Dockerfile:

1. **Base Image:** You start by specifying a base image, which serves as the foundation for your container. Base images are typically lightweight Linux distributions or specific application images.
2. **Instructions:** Dockerfiles consist of various instructions, each responsible for a specific task. Common instructions include `FROM` (to set the base image), `RUN` (to execute commands inside the container during build), `COPY` (to copy files into the container), and `CMD` (to specify the default command to run when the container starts).
3. **Layering:** Docker images are built in layers. Each instruction in the Dockerfile creates a new layer. Layers are cached, and if nothing changes in a layer, Docker can reuse it from cache, making builds faster.
4. **Environment Configuration:** You can set environment variables, configure ports, define working directories, and perform other setup tasks to configure the container environment.
5. **File Copying:** You can copy files from the host system into the container image using the `COPY` instruction. This is often used to add application code, configuration files, or other assets.
6. **Container Execution Command:** The `CMD` instruction specifies the default command to run when the container is started. It's often used to launch the primary application within the container.
7. **Best Practices:** Following [**best practices**](#) in Dockerfile design can help create efficient and [**secure images**](#). This includes minimizing the number of layers, avoiding installing unnecessary packages, and keeping images small.

Dockerfiles are a crucial component of Docker's philosophy of containerization. They allow developers to define the entire environment needed to run an application, ensuring consistency across different environments and simplifying deployment and scaling.

Dockerfile for churn app

Let's examine the following Dockerfile for the churn app, along with explanations of the steps involved:

```
WORKDIR /app
```

This instruction sets the working directory to '/app' within the Docker container. If the directory doesn't exist, it creates it and navigates to it, similar to the 'cd' command.

```
COPY ["Pipfile", "Pipfile.lock", "./"]
```

Here, we copy the ‘Pipfile’ and ‘Pipfile.lock’ from the local directory into the ‘/app’ directory within the Docker container.

```
RUN pipenv install --system --deploy
```

Instead of creating a virtual environment within the Docker container (using `RUN pipenv install`), this instruction installs the required dependencies directly into the system environment. This is often preferred to keep the container smaller and simpler.

However, there are two important tasks missing. First, we need to specify which port should be exposed by the container to the host machine. We achieve this with the ‘EXPOSE’ instruction, which tells Docker to open port 9696.

```
EXPOSE 9696
```

Finally, when running the container, we need to map the port between the container and the host machine. This is done using the ‘-p’ option, where ‘9696:9696’ (container port : host port) indicates that port 9696 on the host machine should be mapped to port 9696 within the container.

```
FROM python:3.8.12-slim

RUN pip install pipenv

WORKDIR /app
COPY["Pipfile", "Pipfile.lock", "./"]

RUN pipenv install --system --deploy
COPY["predict.py", "model_C=1.0.bin", "./"]

EXPOSE 9696

ENTRYPOINT["gunicorn", "--bind=0.0.0.0:9696", "predict:app"]
```

Building a Docker image

To build the Docker container, use the following command:

```
docker build -t zoomcamp-test .
```

This command instructs Docker to build an image tagged as ‘zoomcamp-test’ using the current directory as the build context.

Running a Docker image

To run the container, use the following command:

```
docker run -it --rm -p 9696:9696 zoomcamp-test
```

The port mapping established by `-p 9696:9696` enables our ‘test.py’ script running on the host machine to communicate with the churn service running inside the Docker container. This mapping allows the script to utilize the local machine’s port, which is mapped to the container’s port used by the churn service.

👤 [Peter](#) ⏰ [14. October 2023](#) 📁 [ML-Zoomcamp, Model Deployment](#)
🏷️ [Container](#), [Docker](#), [ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Deploying Machine Learning Models– Part 7

👤 Peter ⏰ 15. October 2023 📁 ML-Zoomcamp, Model Deployment
🏷️ AWS, Cloud, Deployment, Elastic Beanstalk, ML Zoomcamp



1. [Deployment to the cloud: AWS Elastic Beanstalk](#)

1. [Introduction to AWS Elastic Beanstalk](#)
 1. [What is AWS Elastic Beanstalk?](#)
 2. [Who Should Use AWS Elastic Beanstalk?](#)
2. [How AWS Elastic Beanstalk Enhances the Churn Service](#)
3. [Installing the EB CLI](#)
4. [Running EB locally](#)
5. [EB best practices](#)
6. [Deploying the model to the cloud](#)

Deployment to the cloud: AWS Elastic Beanstalk

This article covers the deployment of the Docker container to the cloud. In the previous article we built a Docker image, build the container, put our churn prediction service there and also the model. Then we build this image and learn how to run it locally. Now we want to take this image and deploy it to the cloud.

[Amazon Elastic Beanstalk](#) is one of the services in Amazon AWS. It's an easy way to deploy your services, also including dockerized containers.

Introduction to AWS Elastic Beanstalk

In the world of cloud computing, deploying and managing web applications can be a

complex and time-consuming task. Enter AWS Elastic Beanstalk, a service provided by Amazon Web Services (AWS) that simplifies the process of deploying and scaling web applications.

What is AWS Elastic Beanstalk?

[**AWS Elastic Beanstalk**](#) is a Platform as a Service (PaaS) offering that abstracts away the underlying infrastructure and streamlines the deployment, scaling, and management of web applications. With Elastic Beanstalk, developers can focus on writing code, while AWS takes care of provisioning resources, load balancing, auto-scaling, and monitoring.

Key Features of AWS Elastic Beanstalk:

1. **Easy Deployment:** Elastic Beanstalk supports multiple programming languages and web frameworks, making it suitable for a wide range of applications. You can deploy your code with a few simple commands or use integrated development environments (IDEs) to seamlessly deploy from your development environment.
2. **Automatic Scaling:** Elastic Beanstalk can automatically adjust the number of application instances based on traffic load. This ensures that your application remains responsive even during traffic spikes, without manual intervention.
3. **Managed Infrastructure:** AWS handles all the infrastructure management tasks, including server provisioning, operating system updates, and security patches. This allows developers to focus on writing code and delivering features.
4. **Monitoring and Analytics:** Elastic Beanstalk provides built-in integration with AWS CloudWatch, enabling real-time monitoring of application performance and resource utilization. You can set up alarms and triggers to respond to issues proactively.
5. **Easy Environment Management:** Developers can create multiple environments for different stages of the development lifecycle, such as development, testing, and production. Each environment is isolated, making it easy to test changes before deploying them to production.

Who Should Use AWS Elastic Beanstalk?

AWS Elastic Beanstalk is an excellent choice for startups, small businesses, and development teams looking to streamline the deployment and management of web applications. It's also suitable for experienced AWS users who want to reduce the operational overhead of managing infrastructure.

How AWS Elastic Beanstalk Enhances the Churn Service

Here's how it works: AWS Elastic Beanstalk (EB) serves as the orchestrator in the cloud. Our churn service is running inside a container within EB. To enable external communication, we've already exposed the port within this container.

Now, when the marketing service sends a request to EB, Elastic Beanstalk acts as a bridge. It forwards the incoming request to the container hosting our churn service. The container processes the request and sends back the prediction. This way, the marketing service

obtains the necessary prediction data.

One of the remarkable features of Elastic Beanstalk is its ability to dynamically adjust to varying traffic loads. When a surge of requests arrives, EB can automatically scale up, utilizing load balancing to distribute traffic efficiently. This scaling process involves adding more containers and instances of our service to meet the increased demand.

Conversely, when traffic subsides, Elastic Beanstalk can scale down gracefully, reducing the number of containers and instances. This automatic scaling ensures that resources are optimized according to the actual workload, helping us achieve cost-efficiency and responsiveness in our application deployment.

In essence, AWS Elastic Beanstalk acts as a flexible and responsive manager for our containerized churn service, allowing it to seamlessly adapt to changing traffic patterns while maintaining performance and reliability.

Installing the EB CLI

To install the command line interface for [AWS Elastic Beanstalk](#) (awsebcli) as a development dependency, you can use the following commands:

```
pipenv install awsebcli --dev
```

After installing, you can activate the virtual environment with:

```
pipenv shell
```

Then, initialize an Elastic Beanstalk application with the specified options using:

```
eb init -p docker -r eu-west-1 churn-serving
```

This command will create an Elastic Beanstalk application named ‘churn-serving’ and generate a ‘.elasticbeanstalk’ folder containing a ‘config.yml’ file, which contains configuration settings for your Elastic Beanstalk environment.

Running EB locally

Before deploying your application to the cloud, you can use AWS Elastic Beanstalk to test it locally. Elastic Beanstalk provides a convenient way to replicate the deployment environment on your local machine.

To run your application locally with Elastic Beanstalk, you can use the following command:

```
eb local run --port 9696
```

This command sets up a local environment that mimics the configuration of your Elastic Beanstalk environment, including port mappings.

Now, to test our locally running application, we open another terminal window and use the following command:

```
python predict-test.py
```

This command will send a request to our locally hosted service, allowing us to validate that everything is working as expected before deploying to the cloud. It's a crucial step in ensuring a smooth deployment process.

EB best practices

When installing and using AWS Elastic Beanstalk (EB), there are a few additional considerations and best practices to keep in mind:

1. **AWS Credentials:** Ensure that you have configured AWS credentials properly. You can use the AWS Command Line Interface (CLI) or IAM roles if you're running EB on an AWS resource like an EC2 instance. Make sure the credentials have the necessary permissions for creating and managing Elastic Beanstalk resources.
2. **Region Selection:** Specify the AWS region where you want to deploy your Elastic Beanstalk application. The '-r' flag in the 'eb init' command specifies the region. Choose a region that is geographically close to your target audience for lower latency.
3. **Platform:** Select the appropriate platform for your application. In your example, you're using the Docker platform. Elastic Beanstalk supports various platforms, including Node.js, Python, Ruby, Java, and more. Choose the one that matches your application's technology stack.
4. **Environment Configuration:** After initializing your application, you'll typically create one or more environments within that application. Environments are isolated instances of your application with their own settings. You can define environment-specific configuration, such as environment variables and scaling options.
5. **Deployment:** Use the 'eb deploy' command to package and deploy your application code to Elastic Beanstalk environments. This command uploads your code and dependencies to the environment, making it available for execution.
6. **Logs and Monitoring:** AWS Elastic Beanstalk integrates with AWS CloudWatch for monitoring and logging. You can access logs and performance metrics to monitor the health and performance of your environments. Set up alarms and triggers to be notified of any issues.
7. **Scaling:** Elastic Beanstalk provides auto-scaling options to automatically adjust the number of instances based on traffic. You can configure scaling policies to ensure that your application can handle fluctuations in demand.
8. **Security:** Implement security best practices by configuring security groups, using AWS Identity and Access Management (IAM) for access control, and applying encryption where necessary.
9. **Cost Management:** Keep an eye on the cost of running Elastic Beanstalk environments, especially when using auto-scaling. Use AWS Budgets and Cost Explorer to manage and optimize your AWS expenses.
10. **Documentation and Support:** AWS Elastic Beanstalk has extensive documentation and a support community. If you encounter issues or need assistance, consult the official documentation and consider leveraging AWS support options.

By following these considerations and best practices, you can effectively install, configure, and manage AWS Elastic Beanstalk for your applications while ensuring reliability and cost-efficiency.

Deploying the model to the cloud

To create an Elastic Beanstalk environment for your application, you can use the following command:

```
eb create churn-serving-env
```

This command initiates the creation of an Elastic Beanstalk environment named ‘churn-serving-env.’ Please note that the environment creation is not instantaneous, so it may take a moment to complete. Once the process finishes, you’ll receive information indicating the specific address where your application is available.

An essential point to highlight here is that creating an Elastic Beanstalk environment this way makes it accessible from the internet by default. Therefore, it’s crucial to implement proper security measures and ensure that only authorized services and users have access.

Now, to test our running application, open another terminal window and execute the following command:

```
python predict-test.py
```

While using EB we must do a few changes in predict-test.py.

```

import requests

host = 'churn-serving-env....eu-west-1.elasticbeanstalk.com'
url = f'http://{host}/predict'

customer_id = 'xyz-123'
customer = {
    "gender": "female",
    "seniorcitizen": 0,
    "partner": "yes",
    "dependents": "no",
    "phoneservice": "no",
    "multiplelines": "no_phone_service",
    "internetservice": "dsl",
    "onlinesecurity": "no",
    "onlinebackup": "yes",
    "deviceprotection": "no",
    "techsupport": "no",
    "streamingtv": "no",
    "streamingmovies": "no",
    "contract": "month-to-month",
    "paperlessbilling": "yes",
    "paymentmethod": "electronic_check",
    "tenure": 1,
    "monthlycharges": 29.85,
    "totalcharges": 29.85
}

response = requests.post(url, json=customer).json()
print(response)

if response['churn'] == True:
    print('sending promo email to %s' % customer_id)
else:
    print('not sending promo email to %s' % customer_id)

```

To terminate the Elastic Beanstalk environment when you're done with it, you can use the following command:

```
eb terminate churn-serving-env
```

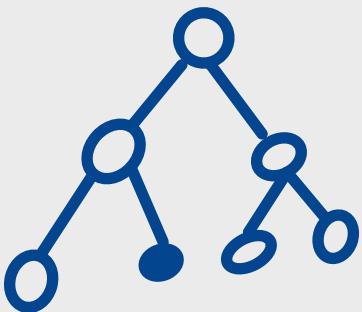
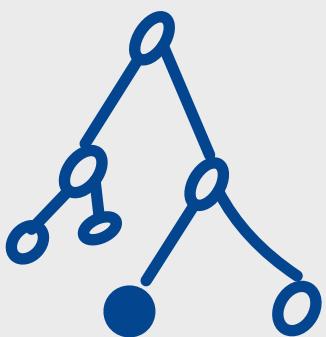
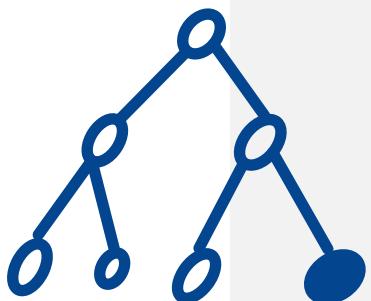
This command will gracefully shut down and remove the Elastic Beanstalk environment, helping you manage your resources efficiently.

 [Peter](#)  [15. October 2023](#)  [ML-Zoomcamp](#), [Model Deployment](#)
 [AWS](#), [Cloud](#), [Deployment](#), [Elastic Beanstalk](#), [ML Zoomcamp](#)

Leave a comment

DECISION TREES

+
ENSEMBLE
LEARNING



SUMMARY

Here, we see decision Trees. Then, random forest and we finish with XGBoosting, which usually gives the best performance. This XGBoost consists on, instead of training decision Trees in parallel (as in random forests), you train them sequentially.

Note that even though here we use it on classification problems, these methods can be used for linear regression.

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 1

👤 Peter ⏰ 16. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ Binary Classification, Classification, ML Zoomcamp



Credit Risk Scoring Project

The project for this week involves credit risk scoring. Imagine you want to buy a mobile phone, so you visit your bank to apply for a loan. You fill out an application form that requests various details, such as your income, the price of the phone, and the loan amount you need. The bank evaluates your application and assigns a score, ultimately deciding whether to approve or decline your request with a ‘yes’ or ‘no’ response.

In this chapter, our goal is to build a model that the bank can utilize to make informed decisions about lending money to customers. The bank can provide the model with customer information, and in return, the model will generate a risk score, indicating the likelihood of a customer defaulting on the loan. This risk score enables the bank to make well-informed lending decisions.

Our approach involves analyzing historical data from various customers and their loan applications. For each case, we have information about the requested loan amount and whether the customer successfully repaid the loan or defaulted.

For instance:

- Customer A -> OK
- Customer B -> OK
- Customer C -> DEFAULT
- Customer D -> DEFAULT
- Customer E -> OK

This problem can be framed as **binary classification**, where ‘y’ represents the target variable, and it can take on two values: 0 (OK) or 1 (DEFAULT). Our objective is to train a model to predict, for each new customer, the probability that they will default:

$g(x_i) \rightarrow \text{PROBABILITY OF DEFAULT}$

We have ‘X,’ which encompasses all the customer information, and the target variable ‘y,’ which indicates the default probability.

Dataset: You can find the dataset at this [link](#). The ‘Status’ variable in the dataset denotes whether the customer defaulted or not.

 [Peter](#)  [16. October 2023](#)  [Decision Trees, ML-Zoomcamp](#)
 [Binary Classification, Classification, ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 2

👤 Peter ⏰ 17. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ Data preparation, Decision Tree, ML Zoomcamp



1. [Preparation Steps – Part 1/2](#)

1. [Imports for this project](#)
2. [Downloading the dataset](#)
3. [Previewing the CSV File](#)
4. [Adapting Column Format](#)

In part 1 of this chapter, “Decision Trees and Ensemble Learning,” we introduced the project, which is a **binary classification** problem aimed at predicting the probability of a client defaulting on a loan. Part 2 of this chapter is divided into two main sections.

Preparation Steps

In the first part, we focus on necessary preparation steps. This includes importing essential libraries, downloading the dataset, previewing the the CSV File, and performing an initial column format adaptation to ensure uniformity in our data.

Data Transformation and Splitting

The second part is dedicated to re-encoding categorical variables and performing the train/validation/test split, a crucial step in preparing our data for modeling and evaluation.

Preparation Steps – Part 1/2

Imports for this project

For this project, we'll need to import several essential libraries that we're already familiar with. These libraries provide the foundation for our data analysis and machine learning tasks. The necessary libraries include:

1. **NumPy**: NumPy is a fundamental library for numerical and array operations in Python.
2. **Pandas**: Pandas is used for data manipulation and analysis, allowing us to work with structured data efficiently.
3. **Scikit-Learn**: Scikit-Learn is a powerful machine learning library that provides a wide range of tools and algorithms for our classification task. This library we'll import at a later point.
4. **Matplotlib**: Matplotlib is essential for data visualization, enabling us to create informative plots and charts.
5. **Seaborn**: Seaborn complements Matplotlib and simplifies the creation of aesthetically pleasing statistical visualizations.

By ensuring that we have these libraries at our disposal, we'll be well-equipped to tackle the various tasks involved in our credit risk scoring project.

```
import pandas as pd
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt
%matplotlib inline
```

Downloading the dataset

To begin our analysis, we must first obtain the dataset. We achieve this using the ‘wget’ command. The following console command, which starts with ‘!’, allows us to download the CSV file from the web. We access the data by referencing the URL stored in the ‘data’ variable.

```
data = 'https://github.com/gastonstat/CreditScoring/blob/master/
!wget $data

# Output
# --2023-10-08 19:13:49-- https://github.com/gastonstat/CreditS...
# Resolving github.com (github.com)... 140.82.121.4
# Connecting to github.com (github.com)|140.82.121.4|:443... con...
# HTTP request sent, awaiting response... 200 OK
# Length: 321064 (314K) [text/plain]
# Saving to: 'CreditScoring.csv'

# CreditScoring.csv   100%[=====] 313,54K  --.-KB/s
# 2023-10-08 19:13:50 (3,24 MB/s) - 'CreditScoring.csv' saved [3]
```

Previewing the CSV File

Similarly to using the ‘wget’ console command, we can gain an initial overview of the CSV

file by using the ‘head’ function. This function works with text files and provides a quick look at the file’s content, just as we’ve seen with Pandas dataframes.

```
!head CreditScoring.csv  
#df = pd.read_csv(data)  
df = pd.read_csv('CreditScoring.csv')  
df.head()
```

	Status	Seniority	Home	Time	Age	Marital	Records	Job	Expenses
0	1	9	1	60	30	2	1	3	73
1	1	17	1	60	58	3	1	1	48
2	2	10	2	36	46	2	2	3	90
3	1	0	1	60	24	1	1	1	63
4	1	0	1	36	26	1	1	1	46

Adapting Column Format

We’ve observed that some of the categorical variables, such as ‘status,’ ‘home,’ ‘marital,’ ‘records,’ and ‘job,’ are currently encoded as numerical values, which can be less intuitive. To make the data more understandable, we’ll convert these columns into text format.

As a first step, we’ll lowercase all the column names for consistency:

```
df.columns = df.columns.str.lower()  
df.head()
```

This change ensures uniformity in our data and improves readability.

	Status	Seniority	Home	Time	Age	Marital	Records	Job	Expenses
0	1	9	1	60	30	2	1	3	73
1	1	17	1	60	58	3	1	1	48
2	2	10	2	36	46	2	2	3	90
3	1	0	1	60	24	1	1	1	63
4	1	0	1	36	26	1	1	1	46

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 3

👤 Peter ⏰ 18. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ [Decision Tree](#), [Missing Values](#), [ML Zoomcamp](#)



1. [Data cleaning and preparation – Data Transformation and Splitting – Part 2/2](#)
 1. [Re-encoding the categorical variables](#)
 1. [Missing values](#)
 2. [Performing the train/validation/test split](#)

Building upon the necessary preparation steps outlined in the previous article, this section is dedicated to two critical processes: re-encoding categorical variables and performing the train/validation/test split, a crucial step in preparing our data for modeling and evaluation.

Data cleaning and preparation – Data Transformation and Splitting – Part 2/2

Re-encoding the categorical variables

To handle the categorical variables, we need to address a few important considerations. The R file in the repository provides insights into preprocessing this data.

Here are the key points:

1. **Missing Values:** The missing values are encoded as a series of nines (99999999). We'll need to address how to handle these missing values.
2. **Categorical Variable Information:** The R file also offers information about the categorical variables. For example:

- ‘Status’ is encoded as ‘good’ (1) and ‘bad’ (2).
- ‘Home’ includes categories like ‘rent,’ ‘owner,’ ‘priv,’ ‘ignore,’ ‘parents,’ and ‘other.’
- ‘Marital’ encompasses ‘single,’ ‘married,’ ‘widow,’ ‘separated,’ and ‘divorced.’
- ‘Records’ has ‘yes’ and ‘no.’
- ‘Job’ includes ‘fixed,’ ‘partime,’ ‘freelance,’ and ‘other.’

To proceed, we’ll need to translate these numerical values back into their respective categorical strings. This ensures our data is more interpretable and ready for analysis.

To address the ‘status’ variable, we examine the possible values: 1, 2, and 0. As previously mentioned, 1 corresponds to ‘good,’ and 2 corresponds to ‘bad.’ However, we also have one record with the value 0, which we’ll designate as ‘unknown.’

To map these values accordingly, we can use the ‘map’ method. This method takes a dictionary that maps each original dataframe value to a new value.

```
df.status.value_counts()
# Output:
# 1    3200
# 2    1254
# 0     1
# Name: status, dtype: int64

status_values = {
    1: 'ok',
    2: 'default',
    0: 'unk'
}

df.status = df.status.map(status_values)
df.head()
```

	status	seniority	home	time	age	marital	records	job	expenses	in
0	ok	9	1	60	30	2	1	3	73	
1	ok	17	1	60	58	3	1	1	48	
2	default	10	2	36	46	2	2	3	90	
3	ok	0	1	60	24	1	1	1	63	
4	ok	0	1	36	26	1	1	1	46	

Dataframe with mapped values for status column

The same re-encoding process applied to the ‘status’ column should also be carried out for the remaining categorical columns: ‘home_values,’ ‘marital_values,’ ‘records_values,’ and ‘job_values’.

```

home_values = {
    1: 'rent',
    2: 'owner',
    3: 'private',
    4: 'ignore',
    5: 'parents',
    6: 'other',
    0: 'unk'
}
df.home = df.home.map(home_values)

marital_values = {
    1: 'single',
    2: 'married',
    3: 'widow',
    4: 'separated',
    5: 'divorced',
    0: 'unk'
}
df.marital = df.marital.map(marital_values)

records_values = {
    1: 'no',
    2: 'yes',
    0: 'unk'
}
df.records = df.records.map(records_values)

job_values = {
    1: 'fixed',
    2: 'partime',
    3: 'freelance',
    4: 'others',
    0: 'unk'
}
df.job = df.job.map(job_values)

df.head()

```

	status	seniority	home	time	age	marital	records	job	expense
0	ok	9	rent	60	30	married	no	freelance	73
1	ok	17	rent	60	58	widow	no	fixed	48
2	default	10	owner	36	46	married	yes	freelance	90
3	ok	0	rent	60	24	single	no	fixed	63
4	ok	0	rent	36	26	single	no	fixed	46

Dataframe with mapped values for all columns

Missing values

With all categorical variables decoded back to strings, the next step is to address the missing values.

```
| df.describe().round()
```

	seniority	time	age	expenses	income	assets	deb
count	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455
mean	8.0	46.0	37.0	56.0	763317.0	1060341.0	40438
std	8.0	15.0	11.0	20.0	8703625.0	10217569.0	634425
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0
50%	5.0	48.0	36.0	51.0	120.0	3500.0	0.0
75%	12.0	60.0	45.0	72.0	166.0	6000.0	0.0
max	48.0	72.0	68.0	180.0	99999999.0	99999999.0	999999

3 columns with missing values

It's apparent that 'income,' 'assets,' and 'debt' columns contain extremely large values (e.g., 99999999.0) as maximum values. To address this issue, we need to replace these outlier values. Let's explore the replacement process.

```
| df.income.max()  
# Output: 99999999  
  
| df.income.replace(to_replace=99999999, value=np.nan)  
  
| df.income.replace(to_replace=99999999, value=np.nan).max()  
# Output: 959.0
```

We'll address the outlier values in a loop for all three columns: 'income,' 'assets,' and 'debt'.

```
| for c in ['income', 'assets', 'debt']:  
|     df[c] = df[c].replace(to_replace=99999999, value=np.nan)  
  
| df.describe().round()
```

	seniority	time	age	expenses	income	assets	debt	an
count	4455.0	4455.0	4455.0	4455.0	4421.0	4408.0	4437.0	44
mean	8.0	46.0	37.0	56.0	131.0	5403.0	343.0	10
std	8.0	15.0	11.0	20.0	86.0	11573.0	1246.0	4
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	1
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	7
50%	5.0	48.0	36.0	51.0	120.0	3000.0	0.0	10
75%	12.0	60.0	45.0	72.0	165.0	6000.0	0.0	13
max	48.0	72.0	68.0	180.0	959.0	300000.0	30000.0	50

Columns ‘income,’ ‘assets,’ and ‘debt’ without 99999999 values

While analyzing the ‘status’ column, we discovered a single record with the value ‘unk,’ representing a missing or unknown status. Since our focus is solely on ‘ok’ and ‘default’ values, we can safely remove this record from the dataframe.

```
df.status.value_counts()

# Output:
# ok      3200
# default 1254
# unk     1
# Name: status, dtype: int64

df = df[df.status != 'unk'].reset_index(drop=True)
df
```

	status	seniority	home	time	age	marital	records	job	expe
0	ok	9	rent	60	30	married	no	freelance	7
1	ok	17	rent	60	58	widow	no	fixed	4
2	default	10	owner	36	46	married	yes	freelance	9
3	ok	0	rent	60	24	single	no	fixed	6
4	ok	0	rent	36	26	single	no	fixed	4
...
4449	default	1	rent	60	39	married	no	fixed	6
4450	ok	22	owner	60	46	married	no	fixed	6
4451	default	0	owner	24	37	married	no	partime	6
4452	ok	0	rent	48	23	single	no	freelance	4
4453	ok	5	owner	60	32	married	no	freelance	6

4454 rows × 14 columns

Performing the train/validation/test split

The final step in our data preparation is to split the dataset into training, validation, and test sets. We achieve this using the ‘train_test_split’ function from scikit-learn.

Here’s the code for the split:

```

from sklearn.model_selection import train_test_split

df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=11)
df_train, df_val = train_test_split(df_full_train, test_size=0.25,
                                    random_state=11)

df_train = df_train.reset_index(drop=True)
df_val = df_val.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)

df_train.status
# Output:
# 0      default
# 1      default
# 2          ok
# 3      default
# 4          ok
#
#       ...
# 2667      ok
# 2668      ok
# 2669      ok
# 2670      ok
# 2671      ok
# Name: status, Length: 2672, dtype: object

```

To predict a probability, we need to convert our target variable ‘status’ into a numerical format.

```

(df_train.status == 'default').astype('int')

# Output:
# 0      1
# 1      1
# 2      0
# 3      1
# 4      0
#
#       ..
# 2667      0
# 2668      0
# 2669      0
# 2670      0
# 2671      0
# Name: status, Length: 2672, dtype: int64

```

To complete our data preparation, we need to assign target variables for the training, validation, and test sets. Additionally, to prevent accidental use of the target variable during training, we should remove it from ‘df_train,’ ‘df_val,’ and ‘df_test.’

```

y_train = (df_train.status == 'default').astype('int').values
y_val = (df_val.status == 'default').astype('int').values
y_test = (df_test.status == 'default').astype('int').values

del df_train['status']
del df_val['status']
del df_test['status']

df_train

```

	seniority	home	time	age	marital	records	job	expenses	i
0	10	owner	36	36	married	no	freelance	75	
1	6	parents	48	32	single	yes	fixed	35	
2	1	parents	48	40	married	no	fixed	75	
3	1	parents	48	23	single	no	partime	35	
4	5	owner	36	46	married	no	freelance	60	
...
2667	18	private	36	45	married	no	fixed	45	
2668	7	private	60	29	married	no	fixed	60	
2669	1	parents	24	19	single	no	fixed	35	
2670	15	owner	48	43	married	no	freelance	60	
2671	12	owner	48	27	married	yes	fixed	45	

2672 rows × 13 columns

 [Peter](#)
  [18. October 2023](#)
  [Decision Trees, ML-Zoomcamp](#)
 [Decision Tree, Missing Values, ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 4

👤 Peter ⏰ 19. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ [Decision Tree, ML Zoomcamp](#)



1. [Decision Trees – Part 1/2](#)
 1. [Introduction to Decision Trees](#)
 2. [How a decision tree looks like](#)
 3. [Training a decision tree](#)

The next part is also divided into two parts. First I give a brief introduction to decision trees, how a decision tree look like. The last section is about how to train a decision tree. The second part will be about overfitting a decision tree and how to control the size of a tree.

Decision Trees – Part 1/2

This time we want to use the ready-to-use data set from the last article to predict if customers are going to default or not. We want to use decision trees for that.

Introduction to Decision Trees

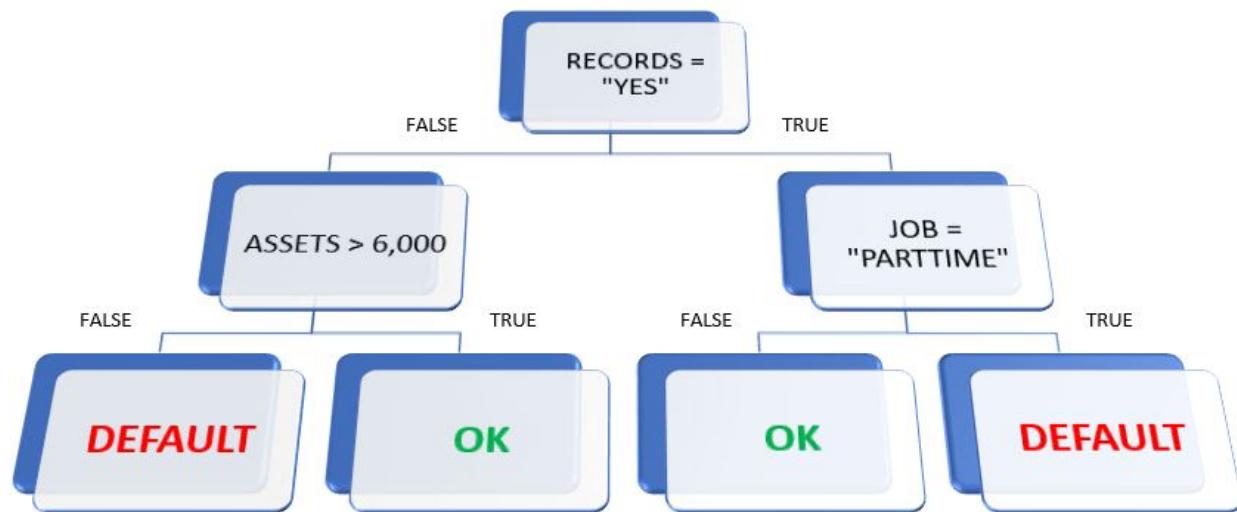
Decision trees are powerful tools in the field of machine learning and data analysis. They are a versatile and interpretable way to make decisions and predictions based on a set of input features. Imagine a tree-like structure where each internal node represents a feature or attribute, each branch signifies a decision or outcome, and each leaf node provides a final prediction or classification.

Decision trees are widely used for tasks such as classification and regression. They are known for their simplicity and ease of interpretation, making them a valuable resource for

understanding and solving complex problems. In this blog post, we'll delve into the world of decision trees, exploring how they work, and how to build them.

How a decision tree looks like

A **decision tree** is a data structure where we have a node which is the condition. And from this node there is one arrow to the left (condition = false) and one to the right (condition=true). Then there is the next condition which can be true or false. ... until there is the final decision 'OK' or 'DEFAULT'



Let's write a function that implements this rule set, and test it with one sample client record.

Training a decision tree translates into the model
Creating its own tree with its own path choices based
on the data

```

def assess_risk(client):
    if client['records'] == 'yes':
        if client['job'] == 'parttime':
            return 'default'
        else:
            return 'ok'
    else:
        if client['assets'] > 6000:
            return 'ok'
        else:
            return 'default'

# just take one record to test
xi = df_train.iloc[0].to_dict()
xi
# Output:
# {'seniority': 10,
# 'home': 'owner',
# 'time': 36,
# 'age': 36,
# 'marital': 'married',
# 'records': 'no',
# 'job': 'freelance',
# 'expenses': 75,
# 'income': 0.0,
# 'assets': 10000.0,
# 'debt': 0.0,
# 'amount': 1000,
# 'price': 1400}

```

When we look at the decision tree, what would be the result for this client? First condition is “RECORDS = YES.” Our client has no records, so we go to the left. The second condition is “ASSETS > 6000.” The assets of our client are 10,000, so we go to the right. Now we reach the decision node, which in this case is “OK.” Let’s use the implemented function shown in the previous snippet and compare the output.

```

assess_risk(xi)
# Output: 'ok'

```

Indeed the output is also “OK”. We don’t want to always implement the whole rule set. These rules can be learned from the data using the decision tree algorithm.

Training a decision tree

Before we can train a decision tree, we first need to import necessary packages. From Scikit-Learn, we import DecisionTreeClassifier. Because we have categorical variables, we also need to import DictVectorizer as seen before.

```

from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction import DictVectorizer
from sklearn.metrics import roc_auc_score

```

Now we need to turn our training dataframe into a list of dictionaries then turn this list of dictionaries into the feature matrix. After that we train the model.

```
# This will lead us later to an error
#train_dicts = df_train.to_dict(orient='records')
# ... so we use instead:

train_dicts = df_train.fillna(0).to_dict(orient='records')
train_dicts[:5]

# Output:
#[{'seniority': 10,
# 'home': 'owner',
# 'time': 36,
# 'age': 36,
# 'marital': 'married',
# 'records': 'no',
# 'job': 'freelance',
# 'expenses': 75,
# 'income': 0.0,
# 'assets': 10000.0,
# 'debt': 0.0,
# 'amount': 1000,
# 'price': 1400},
# {'seniority': 6,
# 'home': 'parents',
# 'time': 48,
# 'age': 32,
# 'marital': 'single',
# 'records': 'yes',
# 'job': 'fixed',
# 'expenses': 35,
# 'income': 85.0,
# 'assets': 0.0,
# 'debt': 0.0,
# 'amount': 1100,
# 'price': 1330},
# {'seniority': 1,
# 'home': 'parents',
# 'time': 48,
# 'age': 40,
# 'marital': 'married',
# 'records': 'no',
# 'job': 'fixed',
# 'expenses': 75,
# 'income': 121.0,
# 'assets': 0.0,
# 'debt': 0.0,
# 'amount': 1320,
# 'price': 1600},
# {'seniority': 1,
# 'home': 'parents',
# 'time': 48,
# 'age': 23,
# 'marital': 'single',
# 'records': 'no',
# 'job': 'partime',
# 'expenses': 35,
# 'income': 72.0},
```

```

# 'assets': 0.0,
# 'debt': 0.0,
# 'amount': 1078,
# 'price': 1079},
# {'seniority': 5,
# 'home': 'owner',
# 'time': 36,
# 'age': 46,
# 'marital': 'married',
# 'records': 'no',
# 'job': 'freelance',
# 'expenses': 60,
# 'income': 100.0,
# 'assets': 4000.0,
# 'debt': 0.0,
# 'amount': 1100,
# 'price': 1897}]

```

```

dv = DictVectorizer(sparse=False)
X_train = dv.fit_transform(train_dicts)
X_train

# Output:
# array([[3.60e+01, 1.00e+03, 1.00e+04, ..., 0.00e+00, 1.00e-
#          [3.20e+01, 1.10e+03, 0.00e+00, ..., 1.00e+00, 0.00e+00],
#          [4.00e+01, 1.32e+03, 0.00e+00, ..., 0.00e+00, 0.00e+00],
#          ...,
#          [1.90e+01, 4.00e+02, 0.00e+00, ..., 0.00e+00, 0.00e+00],
#          [4.30e+01, 2.50e+03, 1.80e+04, ..., 0.00e+00, 0.00e+00],
#          [2.70e+01, 4.50e+02, 5.00e+03, ..., 1.00e+00, 0.00e+00]])

```

All the numerical features remain unchanged, but we have encoding for categorical features.

```

dv.get_feature_names_out()

# Ouput:
# array(['age', 'amount', 'assets', 'debt', 'expenses', 'home',
#        'home=other', 'home=owner', 'home=parents', 'home=rent',
#        'home=unk', 'income', 'job=fixed', 'job=others',
#        'job=partime', 'job=unk', 'marital=married',
#        'marital=separated', 'marital=unk', 'marital=widow',
#        'price', 'records=yes', 'seniority', 'time'], dtype=object)

```

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning – Part 5

👤 Peter ⏰ 20. October 2023 📄 Decision Trees, ML-Zoomcamp
🏷️ Decision Stump, Decision Tree, ML Zoomcamp



1. [Decision Trees – Part 2/2](#)

1. [Overfitting](#)
2. [Decision Stump](#)
 1. [Visualizing Decision Stump](#)
3. [Decision tree with depth of 2](#)
 1. [Visualizing Decision tree](#)

This is part 2 of Decision Trees. While part 1 introduces the concept of a Decision Tree briefly, this section is about overfitting a decision tree and how to control the size of a tree.

Decision Trees – Part 2/2

Let's look back at the performance of our trained Decision Tree from part 1.

```
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)

val_dicts = df_val.fillna(0).to_dict(orient='records')
X_val = dv.transform(val_dicts)
y_pred = dt.predict_proba(X_val)[:, 1]

roc_auc_score(y_val, y_pred)
# Output: 0.6547098641350415
```

We talk that column because we need it to be 1D but y-pred has two columns. More on that later

Overfitting

0.65 is not really a great value, let's look at training data and calculate auc score.

```
y_pred = dt.predict_proba(X_train)[:, 1]
roc_auc_score(y_train, y_pred)
# Output: 1.0
```

This is called overfitting. **Overfitting** is when our model simply memorizes the data, but it memorizes in such a way that when it sees a new example it doesn't know what to do with this example. So it memorizes the training data but it fails to generalize. The reason why this happens to decision trees is that the model creates a specific rule for each example. That works fine for training data, but it doesn't work for any unseen example. The reason why this can happen is, that we let the tree grow too deep. If we restrict the tree to only grow up to three levels deep, the tree will learn rules that are less specific.

```
dt = DecisionTreeClassifier(max_depth=3)
dt.fit(X_train, y_train)

y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train', auc)
# Output: train 0.7761016984958594

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('val', auc)
# Output: val 0.7389079944782155
```

Decision Stump

If we restrict the depth to 3, the model performance on validation is significantly better. It's now 74% compared to 65%. By the way a decision tree with a depth of 1 is called **Decision Stump**. It's not really a tree, because this is only one condition.

```
dt = DecisionTreeClassifier(max_depth=1)
dt.fit(X_train, y_train)

y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train', auc)
# Output: train 0.6282660131823559

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('val', auc)
# Output: val 0.6058644740984719
```

The auc score of this decision stump is only a bit worse than the overfitted one.

Visualizing Decision Stump

Let's examine this tree to understand the rules it has learned. To do that, we can use a

specialized function in Scikit-Learn for visualizing trees.

```
from sklearn.tree import export_text
print(export_text(dt))

# Output:
# |--- feature_25 <= 0.50
# |   |--- class: 1
# |--- feature_25 >  0.50
# |   |--- class: 0
```

This is like saying if feature 25 is less than 0.5, then the output is 1. Else, ...

To understand the meaning of ‘feature_25,’ we need to consult the DictVectorizer feature names dictionary.

```
# in the video Alexey uses
# print(export_text(dt, feature_names=dv.get_feature_names()))

names = dv.get_feature_names_out().tolist()
print(export_text(dt, feature_names=names))

# Output:
# |--- records=no <= 0.50
# |   |--- class: 1
# |--- records=no >  0.50
# |   |--- class: 0
```

This means that if there are no records (records=no <= 0.50), it’s labeled as ‘DEFAULT.’ If there are records (records=no > 0.50), it’s labeled as ‘OK.’ It’s important to note that one-hot encoding is applied here, and there is a column named ‘records=NO,’ which is encoded as 0 when it’s not ‘no’ and 1 when it’s ‘no’.

Decision tree with depth of 2

```
dt = DecisionTreeClassifier(max_depth=2)
dt.fit(X_train, y_train)

y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train', auc)
# Output: train 0.7054989859726213

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('val', auc)
# Output: val 0.6685264343319367
```

Even two levels in a decision tree already performs better than the overfitted one.

Visualizing Decision tree

```
print(export_text(dt))

# Output:
# |--- feature_26 <= 0.50
# |   |--- feature_16 <= 0.50
# |   |   |--- class: 0
# |   |   |--- feature_16 >  0.50
# |   |   |   |--- class: 1
# |--- feature_26 >  0.50
# |   |--- feature_27 <= 6.50
# |   |   |--- class: 1
# |   |   |--- feature_27 >  6.50
# |   |       |--- class: 0
```

```
names = dv.get_feature_names_out().tolist()
print(export_text(dt, feature_names=names))

# Output:
# |--- records=yes <= 0.50
# |   |--- job=partime <= 0.50
# |   |   |--- class: 0
# |   |   |--- job=partime >  0.50
# |   |   |   |--- class: 1
# |--- records=yes >  0.50
# |   |--- seniority <= 6.50
# |   |   |--- class: 1
# |   |   |--- seniority >  6.50
# |       |--- class: 0
```

• Peter ⏰ 20. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ Decision Stump, Decision Tree, ML Zoomcamp

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 6

👤 Peter ⏰ 21. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ [Decision Stump](#), [Decision Tree](#), [Impurity](#), [Misclassification Rate](#), [ML Zoomcamp](#)



1. [Decision Tree Learning Algorithm – Part 1/2](#)
 1. [Finding the best split for one column](#)
 1. [Misclassification Rate](#)
 2. [Impurity](#)

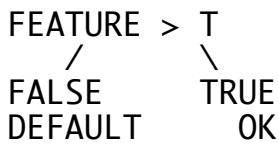
Decision Tree Learning Algorithm – Part 1/2

Before we dive deeper into parameter tuning, which is the topic of Part 8 of this chapter (Decision Trees and Ensemble Learning), let's take a step back for a moment and explore how a decision tree can generate rules, as we've seen in the last article.

In this article, we simplify the problem by using a much smaller dataset with just one feature. Additionally, we employ a very small tree, known as a “decision stump.” This approach offers a clearer understanding of how a decision tree learns from data.

So, let's begin with a **decision stump**, which is essentially a decision tree with only one level. This means we have a **condition node** with the condition inside. Conditions are in the form of “feature > T”, where T represents a threshold. The condition “feature > T” is referred to as a “split”. This split divides the data into records that satisfy the condition (TRUE) and those that do not (FALSE). At the top of the tree, we have the condition node, and at the bottom, there are the **leaves** of the tree. These leaves are called **decision nodes**. Decision nodes are where the tree doesn't go any deeper, and this is where decisions are made. In this context, “decision making” means that we choose the most frequently occurring status label within each group and use it as the final decision for that

group.



To illustrate how the learning algorithm works, let's use a simple dataset. This dataset contains 8 records with one value for 'assets' and one for 'status,' which is our target variable.

```
data = [
    [8000, 'default'],
    [2000, 'default'],
    [0, 'default'],
    [5000, 'ok'],
    [5000, 'ok'],
    [4000, 'ok'],
    [9000, 'ok'],
    [3000, 'default'],
]
df_example = pd.DataFrame(data, columns=['assets', 'status'])
```

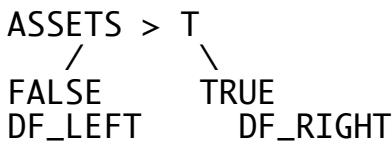
	assets	status
0	8000	default
1	2000	default
2	0	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok
7	3000	default

Simple dataset with 8 records

Finding the best split for one column

What we want to do now is, we want to use this numerical column 'assets' in our training. We want to train a decision tree using this column. So we want to train a model based on the condition ASSETS > T The question is: What is the best T? So what is the best split?

We want to split the dataset into two parts. One part where the condition holds true, and the other part where the condition does not hold true.



DF_RIGHT is where this condition is true, so for all records where ‘assets’ is greater than T. DF_LEFT is where this condition is false, so for all records where ‘assets’ is not greater than T.

This is called **splitting**, so we’re splitting the data set into two parts where it’s true and where it’s false.

What we can do now is sorting the data by the ‘assets’ column.

```
df_example.sort_values('assets')
```

	Assets	Status
2	0	default
1	2000	default
7	3000	default
5	4000	ok
3	5000	ok
4	5000	ok
0	8000	default
6	9000	ok

Simple dataset sorted by ‘assets’ column

We are considering different threshold values “T” for the condition “ASSETS > T.” The possibilities for the threshold “T” are:

- T = 2000 (ASSETS<=2,000 becomes LEFT, ASSETS>2,000 becomes RIGHT)
- T = 3000 (ASSETS<=3,000 becomes LEFT, ASSETS>3,000 becomes RIGHT)
- T = 4000 (ASSETS<=4,000 becomes LEFT, ASSETS>4,000 becomes RIGHT)
- T = 5000 (ASSETS<=5,000 becomes LEFT, ASSETS>5,000 becomes RIGHT)
- T = 8000 (ASSETS<=8,000 becomes LEFT, ASSETS>8,000 becomes RIGHT)

It’s important to note that values like T = 9000 and T = 0 don’t make sense for splitting

because there would be nothing on the right or left side, respectively.

```
| Ts = [0, 2000, 3000, 4000, 5000, 8000]
```

We want to evaluate each of these threshold values “T” by splitting our dataset into the left (LEFT) and right (RIGHT) subsets and determine which split provides the best results.

```
from IPython.display import display  
  
for T in Ts:  
    print(T)  
    df_left = df_example[df_example.assets <= T]  
    df_right = df_example[df_example.assets > T]  
  
    display(df_left)  
    display(df_right)  
  
    print()
```

0

	Assets	status
2	0	default
LEFT for T=0		

	Assets	Status
0	8000	default
1	2000	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok
7	3000	default
RIGHT for T=0		

2000

	Assets	Status
1	2000	default
2	0	default
LEFT for T=2000		

	Assets	status
0	8000	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok
7	3000	default
RIGHT for T=2000		

3000

	Assets	Status
1	2000	default
2	0	default
7	3000	default
LEFT for T=3000		

	assets	status
0	8000	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok

RIGHT for T=3000

4000

	assets	status
1	2000	default
2	0	default
5	4000	ok
7	3000	default

LEFT for T=4000

	assets	status
0	8000	default
3	5000	ok
4	5000	ok
6	9000	ok

RIGHT for T=4000

5000

	assets	status
1	2000	default
2	0	default
3	5000	ok
4	5000	ok
5	4000	ok
7	3000	default

LEFT for T=5000

	assets	status
0	8000	default
6	9000	ok
RIGHT for T=5000		

8000

	assets	status
0	8000	default
1	2000	default
2	0	default
3	5000	ok
4	5000	ok
5	4000	ok
7	3000	default

LEFT for T=8000

	ASSETS	STATUS
6	9000	ok
RIGHT for T=8000		

Now that we have generated many splits using different threshold values, we need to determine which split is the best. To evaluate this, we can use various split evaluation criteria. Let's take the example where $T=4000$. In this case, on the LEFT side, we have 3 cases labeled as DEFAULT and 1 case labeled as OK. On the RIGHT side, we have 3 cases labeled as OK and 1 case labeled as DEFAULT. As mentioned earlier, we select the most frequently occurring status within each group and use it as the final decision. In the case of the LEFT group, the most frequent status is "default," and for the RIGHT group, it's "ok."

```

T = 4000
df_left = df_example[df_example.assets <= T]
df_right = df_example[df_example.assets > T]

display(df_left)
display(df_right)

```

	Assets	Status
1	2000	default
2	0	default
5	4000	ok
7	3000	default
LEFT for T=4000		

	ASSETS	STATUS
0	8000	default
3	5000	ok
4	5000	ok
6	9000	ok
RIGHT for T=4000		

ASSETS > 4000 / \ FALSE TRUE DEFAULT OK
--

Misclassification Rate

To evaluate the accuracy of our predictions, we can calculate the misclassification rate. The misclassification rate measures the fraction of errors when we predict that everyone in the LEFT group is a DEFAULT, and similarly for the RIGHT group. It gives us insight into how well our model is performing.

For the LEFT group (3xDEFAULT, 1xOK), where we predict everything as DEFAULT, the misclassification rate is 1/4, which equals 25%.

For the RIGHT group (3xOK, 1xDEFAULT), where we predict everything as OK, the misclassification rate is also 1/4, or 25%.

This is how we evaluate the quality of our split. For T=4000, the average misclassification rate is 25%. We can also use a weighted average, which is particularly useful when there are significantly more examples on one side than on the other. You can print the normalized value counts as shown in the next snippet.

```

T = 4000
df_left = df_example[df_example.assets <= T]
df_right = df_example[df_example.assets > T]

display(df_left)
print(df_left.status.value_counts(normalize=True))
display(df_right)
print(df_right.status.value_counts(normalize=True))

```

	ASSETS	Status
1	2000	default
2	0	default
5	4000	ok
7	3000	default

LEFT for T=4000

```

default    0.75
ok        0.25
Name: status, dtype: float64

```

	ASSETS	STATUS
0	8000	default
3	5000	ok
4	5000	ok
6	9000	ok

RIGHT for T=4000

```

default    0.75
ok        0.25
Name: status, dtype: float64

```

Let's print out the average misclassification rate for each split. You can use the code implemented in the snippet below to calculate and display this information.

```

for T in Ts:
    print(T)
    df_left = df_example[df_example.assets <= T]
    df_right = df_example[df_example.assets > T]

    display(df_left)
    print(df_left.status.value_counts(normalize=True))
    display(df_right)
    print(df_right.status.value_counts(normalize=True))

    print()

```

0

ASSETS	STATUS
2	0

LEFT for T=0

```

default 1.0
Name: status, dtype: float64

```

ASSETS	STATUS
0	8000

1	2000	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok
7	3000	default

RIGHT for T=0

```

ok 0.571429
default 0.428571
Name: status, dtype: float64

```

2000

	ASSETS	STATUS
1	2000	default
2	0	default

LEFT for T=2000

```
| default 1.0
| Name: status, dtype: float64
```

	ASSETS	STATUS
0	8000	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok
7	3000	default

RIGHT for T=2000

```
| ok 0.666667
| default 0.333333
| Name: status, dtype: float64
```

	ASSETS	STATUS
1	2000	default
2	0	default
7	3000	default

LEFT for T=3000

```
| default 1.0
| Name: status, dtype: float64
```

	ASSETS	STATUS
0	8000	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok

RIGHT for T=3000

```
| ok      0.8
| default 0.2
Name: status, dtype: float64
```

4000

	ASSETS	STATUS
1	2000	default
2	0	default
5	4000	ok
7	3000	default

LEFT for T=4000

```
| default 0.75
| ok      0.25
Name: status, dtype: float64
```

	ASSETS	STATUS
0	8000	default
3	5000	ok
4	5000	ok
6	9000	ok

RIGHT for T=4000

```
| ok      0.75  
| default 0.25  
| Name: status, dtype: float64
```

5000

	ASSETS	STATUS
1	2000	default
2	0	default
3	5000	ok
4	5000	ok
5	4000	ok
7	3000	default

LEFT for T=5000

```
| default 0.5  
| ok      0.5  
| Name: status, dtype: float64
```

	ASSETS	STATUS
0	8000	default
6	9000	ok

RIGHT for T=5000

```
| default 0.5  
| ok      0.5  
| Name: status, dtype: float64
```

8000

	ASSETS	STATUS
0	8000	default
1	2000	default
2	0	default
3	5000	ok
4	5000	ok
5	4000	ok
7	3000	default

LEFT for T=8000

```
| default    0.571429
| ok        0.428571
| Name: status, dtype: float64
```

	ASSETS	STATUS
6	9000	ok

RIGHT for T=8000

```
| ok      1.0
| Name: status, dtype: float64
```

Impurity

The misclassification rate is just one way of measuring **impurity**. We aim for our leaves to be as pure as possible, meaning that the groups resulting from the split should ideally contain only observations of one class, making them **pure**. The misclassification rate informs us of how impure these groups are by quantifying the error rate in classifying observations.

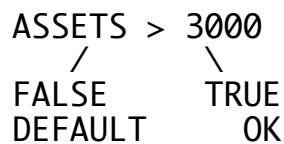
There are also alternative methods to measure impurity. Scikit-Learn offers more advanced criteria for splitting, such as entropy and Gini impurity. These criteria provide different ways to evaluate the quality of splits, with the goal of creating decision trees with more homogeneous leaves.

T	Decision LEFT	Impurity LEFT	Decision RIGHT	Impurity RIGHT	AVG
0	DEFAULT	0%	OK	43%	21%
2000	DEFAULT	0%	OK	33%	16%
3000	DEFAULT	0%	OK	20%	10%
4000	DEFAULT	25%	OK	25%	25%
5000	DEFAULT	50%	OK	50%	50%
8000	DEFAULT	43%	OK	0%	21%
Average Impurity					

We observe that the split with $T = 3000$ yields the lowest average misclassification rate, resulting in the best impurity of 10%. This is how we determine the best split when dealing with just one column. To summarize our approach:

- We sort the dataset.
- We identify all possible thresholds.
- For each of the thresholds, we split the dataset.
- For each split, we calculate the impurity on the left and the right.
- We then compute the average impurity.
- Among all the splits considered, we select the one with the lowest average impurity.

The decision tree we've learned based on this split is as follows:



ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 7

👤 Peter ⏰ 22. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ [Decision Tree](#), [Learning Algorithm](#), [ML Zoomcamp](#)



1. [Decision Tree Learning Algorithm – Part 2/2](#)
 1. [Finding the best split for two columns](#)
 2. [Finding the best split algorithm for multiple features](#)
 1. [Stopping Criteria](#)
 3. [Decision Tree Learning Algorithm](#)

Decision Tree Learning Algorithm – Part 2/2

In part 1/2 of “Decision Tree Learning Algorithm,” we delved into a simple example to understand how a decision tree learns rules. We used the misclassification rate as a means to evaluate the accuracy of our predictions. Additionally, we discussed that the misclassification rate is just one way to measure impurity. In the second part, we will explore how to find the best split when we have two feature columns. This will lead us to a more general algorithm for finding the best split when working with multiple features. Finally, we will conclude this section with a comprehensive Decision Tree Learning Algorithm.

Finding the best split for two columns

In the last article, we began with a simple example that involved just one feature. Now, let's explore what happens when we introduce another feature, ‘debt.’ This feature represents the amount of debt that clients have.

```

data = [
    [8000, 3000, 'default'],
    [2000, 1000, 'default'],
    [0, 1000, 'default'],
    [5000, 1000, 'ok'],
    [5000, 1000, 'ok'],
    [4000, 1000, 'ok'],
    [9000, 500, 'ok'],
    [3000, 2000, 'default'],
]
df_example = pd.DataFrame(data, columns=['assets', 'debt', 'status']
df_example

```

	ASSETS	Debt	Status
0	8000	3000	default
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
6	9000	500	ok
7	3000	2000	default

Dataframe with three features

We've previously attempted splitting the data based on the 'assets' feature. Now, let's investigate the results of attempting a split based on the 'debt' feature. To do this, we should first sort the data by the 'debt' column.

```
| df_example.sort_values('debt')
```

	ASSETS	DEBT	STATUS
6	9000	500	ok
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
7	3000	2000	default
0	8000	3000	default

Dataframe sorted by debt column

Possible thresholds:

- T = 500 (DEBT<=500 becomes LEFT, DEBT>500 becomes RIGHT)
- T = 1000 (DEBT<=1,000 becomes LEFT, DEBT>1,000 becomes RIGHT)
- T = 2000 (DEBT<=2,000 becomes LEFT, DEBT>2,000 becomes RIGHT)

So we've seen how it works for only one feature. Let's generalize a bit and put both threshold variables in a dictionary. If there are more features, we can put them here as well.

```

thresholds = {
    'assets': [0, 2000, 3000, 4000, 5000, 8000],
    'debt': [500, 1000, 2000]
}

for feature, Ts in thresholds.items():
    print('#####')
    print(feature)
    for T in Ts:
        print(T)
        df_left = df_example[df_example[feature] <= T]
        df_right = df_example[df_example[feature] > T]

        display(df_left)
        print(df_left.status.value_counts(normalize=True))
        display(df_right)
        print(df_right.status.value_counts(normalize=True))

    print()
print('#####')

```

#####

assets
0

	ASSETS	DEBT	STATUS
2	0	1000	default

LEFT for T=0

default 1.0
Name: status, dtype: float64

	ASSETS	DEBT	STATUS
0	8000	3000	default
1	2000	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
6	9000	500	ok
7	3000	2000	default

RIGHT for T=0

ok 0.571429
default 0.428571
Name: status, dtype: float64

2000

	ASSETS	DEBT	STATUS
1	2000	1000	default
2	0	1000	default

LEFT for T=2000

default 1.0
Name: status, dtype: float64

	ASSETS	DEBT	STATUS
0	8000	3000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
6	9000	500	ok
7	3000	2000	default

RIGHT for T=2000

```
| ok      0.666667
| default  0.333333
| Name: status, dtype: float64
```

3000

	ASSETS	DEBT	STATUS
1	2000	1000	default
2	0	1000	default
7	3000	2000	default

LEFT for T=3000

```
| default  1.0
| Name: status, dtype: float64
```

	ASSETS	DEBT	STATUS
0	8000	3000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
6	9000	500	ok

RIGHT for T=3000

```
ok      0.8
default 0.2
Name: status, dtype: float64
```

4000

	ASSETS	DEBT	STATUS
1	2000	1000	default
2	0	1000	default
5	4000	1000	ok
7	3000	2000	default

LEFT for T=4000

```
default 0.75
ok     0.25
Name: status, dtype: float64
```

	ASSETS	DEBT	STATUS
0	8000	3000	default
3	5000	1000	ok
4	5000	1000	ok
6	9000	500	ok

RIGHT for T=4000

```
ok      0.75
default 0.25
Name: status, dtype: float64
```

5000

	ASSETS	DEBT	STATUS
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
7	3000	2000	default

LEFT for T=5000

```
| default    0.5  
| ok        0.5  
| Name: status, dtype: float64
```

	ASSETS	DEBT	STATUS
0	8000	3000	default
6	9000	500	ok

RIGHT for T=5000

```
| default    0.5  
| ok        0.5  
| Name: status, dtype: float64
```

8000

	ASSETS	DEBT	STATUS
0	8000	3000	default
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
7	3000	2000	default

LEFT for T=8000

```
| default 0.571429
| ok 0.428571
| Name: status, dtype: float64
```

	ASSETS	DEBT	STATUS
6	9000	500	ok

RIGHT for T=8000

```
| ok 1.0
| Name: status, dtype: float64
```

```
#####
#####
debt
500
```

	Assets	Debt	Status
6	9000	500	ok

LEFT for T=500

```
| ok 1.0
| Name: status, dtype: float64
```

	ASSETS	DEBT	STATUS
0	8000	3000	default
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
7	3000	2000	default

RIGHT for T=500

```
| default 0.571429
| ok      0.428571
| Name: status, dtype: float64
```

1000

	ASSETS	DEBT	STATUS
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
6	9000	500	ok

LEFT for T=1000

```
| ok      0.666667
| default 0.333333
| Name: status, dtype: float64
```

	ASSETS	DEBT	STATUS
0	8000	3000	default
7	3000	2000	default
RIGHT for T=1000			
default 1.0 Name: status, dtype: float64			
2000			

	ASSETS	DEBT	STATUS
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
6	9000	500	ok
7	3000	2000	default
LEFT for T=2000			

	ASSETS	DEBT	STATUS
0	8000	3000	default
RIGHT for T=2000			
default 1.0 Name: status, dtype: float64			

We calculate the table for assets already, let's do this also for debt.

T	Decision LEFT	Impurity LEFT	Decision RIGHT	Impurity RIGHT	AVG
0	DEFAULT	0%	OK	43%	21%
2000	DEFAULT	0%	OK	33%	16%
-----	-----	-----	-----	-----	-----
3000	DEFAULT	0%	OK	20%	10%
-----	-----	-----	-----	-----	-----
4000	DEFAULT	25%	OK	25%	25%
5000	DEFAULT	50%	OK	50%	50%
8000	DEFAULT	43%	OK	0%	21%
Average Impurity for Assets column					

T	Decision LEFT	Impurity LEFT	Decision RIGHT	Impurity RIGHT	AVG
500	OK	0%	DEFAULT	43%	21%
1000	OK	33%	DEFAULT	0%	16%
2000	OK	43%	DEFAULT	0%	21%
Average Impurity for Debt column					

We can observe that the “debt” feature is not as useful for making the split as the “assets” feature. The best split overall remains the same as before: “ASSETS > 3000.” If we were working with three features, we would have another group in the table with another set of rows and another variable to consider.

Finding the best split algorithm for multiple features

```

FOR each feature in FEATURES:
  FIND all thresholds for the feature
  FOR each threshold in thresholds:
    SPLIT the dataset using "feature > threshold" condition
    COMPUTE the impurity of this split
  SELECT the condition with the LOWEST IMPURITY
  
```

We may need to establish some stopping criteria. This is because we apply recursive

splitting, but how do we determine when to stop?

Recursive splitting = keep splitting
because you didn't reach pure leaves

Stopping Criteria

To determine when to halt the recursive splitting process, it is essential to define stopping criteria. These criteria are vital for preventing overfitting and ensuring that the tree-building process concludes when specific conditions are satisfied. In the next article, when we discuss parameter tuning, we will delve into common stopping criteria for decision trees. Selecting the right stopping criteria is crucial as it depends on the dataset and the problem we are addressing. By adhering to these criteria, we can achieve a balance between model complexity and generalization. These criteria can be integrated into the decision tree algorithm to determine when to cease recursive splitting.

The most common stopping criteria are:

1. The group is already pure:

If a node is pure, meaning all samples in that node belong to the same class, further splitting is unnecessary as it won't provide any additional information.

2. The tree reached depth limit (controlled by `max_depth`):

Limiting the depth of the tree helps control overfitting and ensures that the tree doesn't become too complex.

3. The group is too small for further splitting (controlled by `min_samples_leaf`):

Restricting splitting when the number of samples in a node falls below a certain threshold helps prevent overfitting and results in smaller, more interpretable trees.

When we use these criteria, we enforce simplicity in our model, thereby preventing overfitting. Let's summarize the decision tree learning algorithm.

Decision Tree Learning Algorithm

1. Find the Best Split:

For each feature evaluate all splits based on all possible thresholds and select this one that has the lowest impurity.

2. Stop if Max Depth is Reached:

Stop the splitting process if the maximum allowable depth of the tree is reached.

3. If LEFT is Sufficiently Large and Not Pure Repeat for LEFT:

If the left subset of data is both sufficiently large and not pure (contains more than one class), repeat the splitting process for the left subset.

4. If RIGHT is Sufficiently Large and Not Pure Repeat for RIGHT:

Similarly, if the right subset of data is both sufficiently large and not pure, repeat the splitting process for the right subset.

For more information on decision trees, you can visit the [scikit-learn website](#). In the 'Classification criteria' section, you'll find various methods for measuring impurity, including 'Misclassification,' as well as others like 'Entropy' and 'Gini.' It's important to note that decision trees can also be employed to address regression problems.

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 8

👤 Peter ⏰ 23. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ [Decision Tree](#), [Heatmap](#), [ML Zoomcamp](#), [Parameter Tuning](#)



1. [Decision trees parameter tuning](#)
 1. [Selecting max_depth](#)
 2. [Selecting min_samples_leaf](#)

Decision trees parameter tuning

Part 8 of the ‘Decision Trees and Ensemble Learning’ section is dedicated to tuning decision tree parameters. Parameter tuning involves selecting the best parameters for training. In this context, ‘tuning’ means choosing parameters in a way that maximizes or minimizes a chosen performance metric (such as AUC or RMSE) on the validation set. In the case of AUC, our goal is to maximize it on the validation set by finding the parameter values that yield the highest score.

Let’s take a closer look at the parameters available for the `DecisionTreeClassifier`. For a more comprehensive list of parameters, you can refer to the [official website](#). Some of the key parameters include:

- **criterion:** This parameter determines the impurity measure used for splitting. You can choose between ‘gini’ for Gini impurity and ‘entropy’ for information gain. The choice of criterion can significantly impact the quality of the splits in the decision tree.
- **max_depth:** This parameter controls the maximum depth of the decision tree. It plays a crucial role in preventing overfitting by limiting the complexity of the tree. Selecting an appropriate value for `max_depth` helps strike a balance between model simplicity and complexity.
- **min_samples_leaf:** This parameter specifies the minimum number of samples required in a leaf node. It influences the granularity of the splits. Smaller values can result in finer

splits and a more complex tree, while larger values lead to coarser splits and a simpler tree.

By carefully tuning these parameters, you can find the right configuration that optimizes your model's performance, ensuring it's well-suited for your specific machine learning task.

Selecting max_depth

To start the parameter tuning process, our initial focus will be on the 'max_depth' parameter. Our goal is to identify the optimal 'max_depth' value before proceeding to fine-tune other parameters. 'max_depth' governs the maximum depth of the decision tree. When set to 'None,' it imposes no restrictions, allowing the tree to grow as deeply as possible, potentially resulting in numerous layers.

We will conduct experiments using various values for 'max_depth,' including the 'None' setting, which serves as a baseline for comparison and enables us to understand the consequences of not constraining the tree's depth.

```
depths = [1, 2, 3, 4, 5, 6, 10, 15, 20, None]  
for depth in depths:  
    dt = DecisionTreeClassifier(max_depth=depth)  
    dt.fit(X_train, y_train)  
  
    # remember we need the column with negative scores  
    y_pred = dt.predict_proba(X_val)[:, 1]  
    auc = roc_auc_score(y_val, y_pred)  
  
    print('%4s -> %.3f' % (depth, auc))  
  
# Output:  
# 1 -> 0.606  
# 2 -> 0.669  
# 3 -> 0.739  
# 4 -> 0.761  
# 5 -> 0.767  
# 6 -> 0.760  
# 10 -> 0.706  
# 15 -> 0.663  
# 20 -> 0.654  
# None -> 0.657
```

Handwritten annotations on the code output:

- A blue bracket groups the first five lines of output (#1 to #5), labeled "Best ones".
- A blue bracket groups the last three lines of output (#10 to #20), labeled "# Output:".
- An arrow points from the "%4s" placeholder in the print statement to the "Show 4 characters max" note.
- An arrow points from the ".3f" placeholder in the print statement to the "Show up to 3 characters after the dot (3 decimals)" note.

What we can observe here is that the optimal values appear to be around 76% for 'max_depth' values of 4, 5, and 6. This indicates that our best-performing tree should have between 4 to 6 layers. If there were no other parameters to consider, we could choose a depth of 4 to make the tree simpler, with only 4 layers instead of 5. A simpler tree is generally easier to read and understand, making it more transparent in terms of what's happening.

Selecting min_samples_leaf

But ‘max_depth’ is not the only parameter; there is another one called ‘min_samples_leaf.’ We have already determined that the optimal depth falls between 4 and 6. For each of these values, we can experiment with different ‘min_samples_leaf’ values to observe their effects.

```
scores = []

for d in [4, 5, 6]:
    for s in [1, 2, 5, 10, 15, 20, 100, 200, 500]:
        dt = DecisionTreeClassifier(max_depth=depth, min_samples_leaf=s)
        dt.fit(X_train, y_train)

        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        scores.append((d, s, auc))

columns = ['max_depth', 'min_samples_leaf', 'auc']
df_scores = pd.DataFrame(scores, columns=columns)
df_scores.head()
```

	max_depth	min_samples_leaf	auc
0	4	1	0.655327
1	4	2	0.697389
2	4	5	0.712749
3	4	10	0.762578
4	4	15	0.786521

| df_scores.sort_values(by='auc', ascending=False).head()

	max_depth	min_samples_leaf	Auc
Even though we said that max_depth was good enough and is simpler, the best AUC is for more max_depth, setting the limit instead on min_samples_leaf	22	6	0.786997
4	4	15	0.786521
13	5	15	0.785398
14	5	20	0.782159
23	6	20	0.782120

Dataframe sorted by AUC column

This information can be presented differently by transforming it into a DataFrame. In this

DataFrame, ‘min_samples_leaf’ will be on the rows, ‘max_depth’ will be on the columns, and the cell values will represent the ‘auc’ scores. To achieve this, we can utilize the ‘pivot’ function. This tabular format is more user-friendly, and it’s evident that 0.787 is the highest value.

```
# index - rows
df_scores_pivot = df_scores.pivot(index='min_samples_leaf', columns=['max_depth'], values=['auc'])
df_scores_pivot.round(3)
```

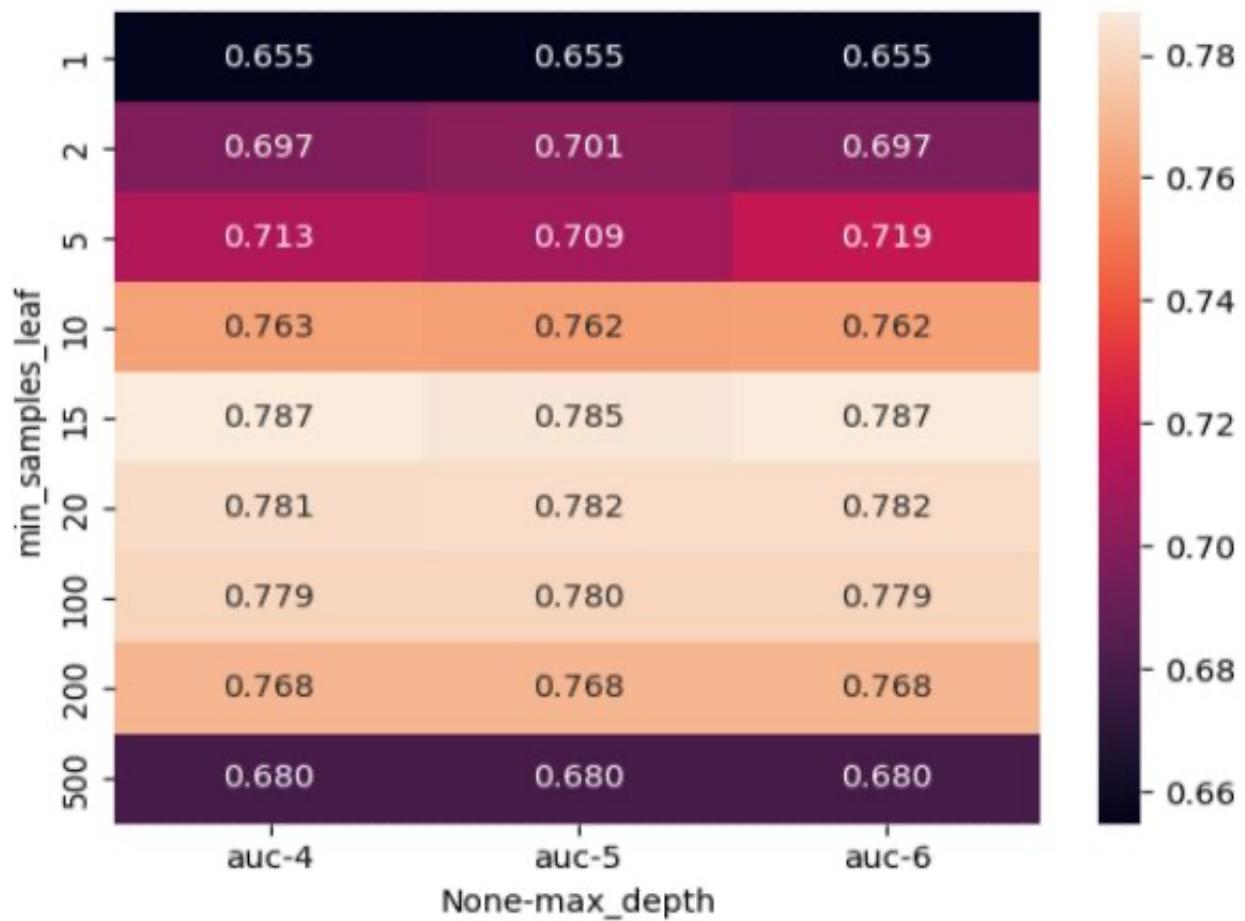
min_samples_leaf	4	5	6
1	0.655	0.655	0.655
2	0.697	0.701	0.697
5	0.713	0.709	0.719
10	0.763	0.762	0.762
15	0.787	0.785	0.787
20	0.781	0.782	0.782
100	0.779	0.780	0.779
200	0.768	0.768	0.768
500	0.680	0.680	0.680

AUC score for different values for max_depth and min_samples_leaf

Another visualization option is to create a heatmap.

```
sns.heatmap(df_scores_pivot, annot=True, fmt=".3f")
```

```
<Axes: xlabel='None-max_depth', ylabel='min_samples_leaf'>
```



In this heatmap, it's easy to identify the highest value as it appears the lightest, while the darkest shade represents the lowest or poorest value. However, it's important to note that this method of selecting the best parameter might not always be optimal. There's a possibility that a 'max_depth' of 7, 10, or another value works better, but we haven't explored those possibilities. This is because we initially tuned the 'max_depth' parameter and then selected the best 'min_samples_leaf.' For small datasets, it's feasible to try a variety of values, but with larger datasets, we need to constrain our search space to be more efficient. Therefore, it's often a good practice to first optimize the 'max_depth' parameter and then fine-tune the other parameters.

Nevertheless, given the small size of this dataset, training is fast, allowing us to experiment with different combinations. Let's explore a few more combinations.

```

scores = []

for d in [4, 5, 6, 7, 10, 15, 20, None]:
    for s in [1, 2, 5, 10, 15, 20, 100, 200, 500]:
        dt = DecisionTreeClassifier(max_depth=depth, min_samples_leaf=s)
        dt.fit(X_train, y_train)

        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        scores.append((d, s, auc))

columns = ['max_depth', 'min_samples_leaf', 'auc']
df_scores = pd.DataFrame(scores, columns=columns)

df_scores.sort_values(by='auc', ascending=False).head()

```

	max_depth	min_samples_leaf	auc
22	6.0	15	0.787878
31	7.0	15	0.787762
58	20.0	15	0.787711
49	15.0	15	0.787293
13	5.0	15	0.786939

AUC score for different values for max_depth and min_samples_leaf sorted by AUC column

The preceding table demonstrates that the AUC scores of the top five options are quite close. Interestingly, in each of these top five cases, ‘min_samples_leaf’ is set to 15. Let’s explore alternative visualizations for deeper insights.

```

df_scores_pivot = df_scores.pivot(index='min_samples_leaf', c
df_scores_pivot.round(3)

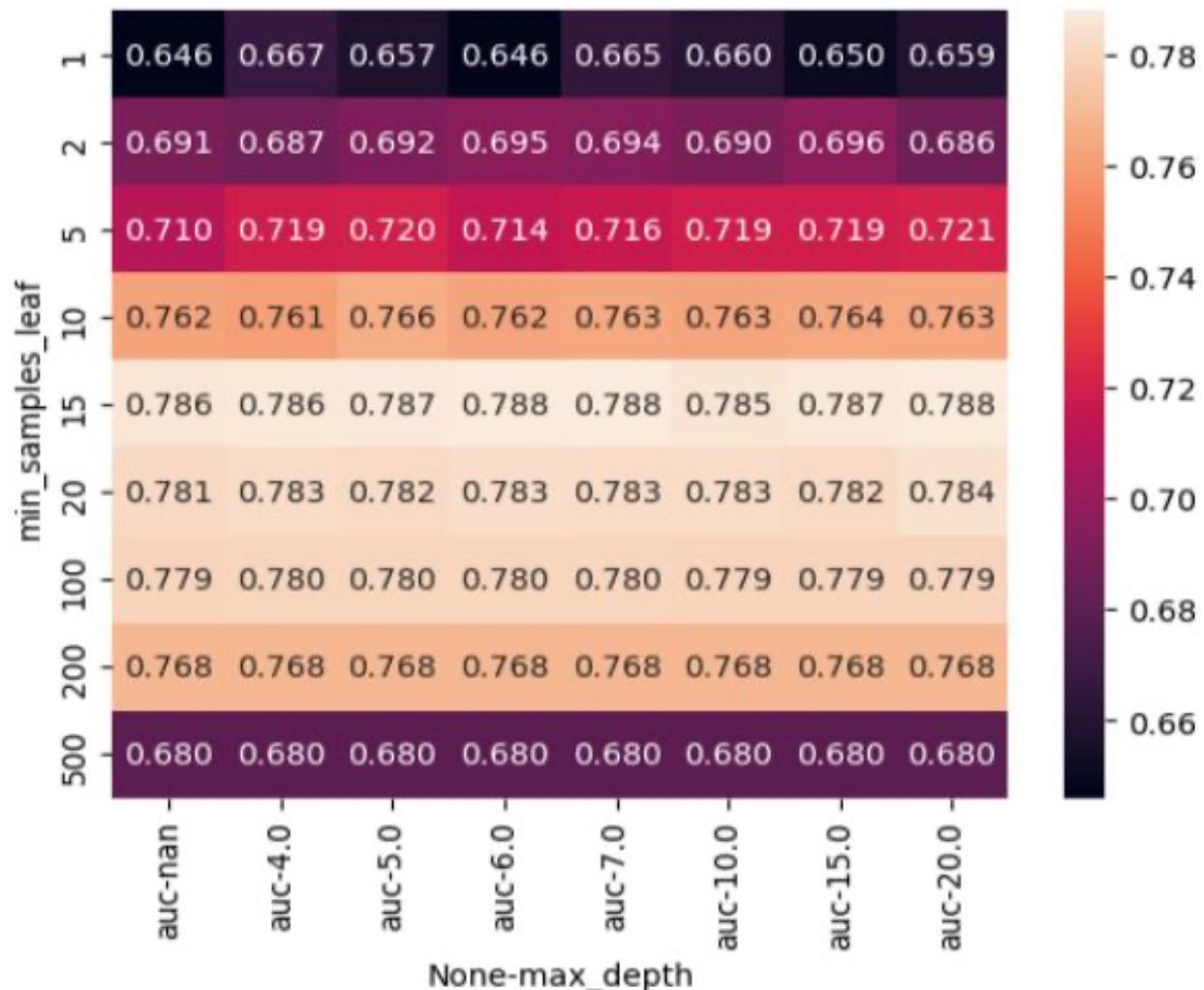
```

max_depth min_samples_leaf	NaN	4.0	5.0	6.0	7.0	10.0	15.0	20.0
1	0.646	0.667	0.657	0.646	0.665	0.660	0.650	0.659
2	0.691	0.687	0.692	0.695	0.694	0.690	0.696	0.686
5	0.710	0.719	0.720	0.714	0.716	0.719	0.719	0.721
10	0.762	0.761	0.766	0.762	0.763	0.763	0.764	0.763
15	0.786	0.786	0.787	0.788	0.788	0.785	0.787	0.788
20	0.781	0.783	0.782	0.783	0.783	0.783	0.782	0.784
100	0.779	0.780	0.780	0.780	0.780	0.779	0.779	0.779
200	0.768	0.768	0.768	0.768	0.768	0.768	0.768	0.768
500	0.680	0.680	0.680	0.680	0.680	0.680	0.680	0.680

AUC score for different values for max_depth and min_samples_leaf

```
| sns.heatmap(df_scores_pivot, annot=True, fmt=".3f")
```

```
<Axes: xlabel='None-max_depth', ylabel='min_samples_leaf'>
```



In the last snippet, we train our `DecisionTreeClassifier` with the final tuned parameters:

'max_depth' set to 6 and 'min_samples_leaf' set to 15. *Avoid auc-nan, can be dangerous not to have limits*

```
dt = DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
dt.fit(X_train, y_train)

# Output: DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
#print(export_text(dt, feature_names=list(dv.get_feature_names)))
```

👤 Peter ⏰ 23. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ Decision Tree, Heatmap, ML Zoomcamp, Parameter Tuning

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 9

👤 Peter ⏰ 24. October 2023 📄 Decision Trees, ML-Zoomcamp
🏷️ [Decision Tree](#), [ML Zoomcamp](#), [Parameter Tuning](#), [Random Forest](#)



1. [Ensemble and random forest](#)
 1. [Board of experts](#)
 2. [Ensembling models](#)
 3. [Random forest – ensembling decision trees](#)
 4. [Tuning random forest](#)

Ensemble and random forest

This article discusses the concept of Random Forest as a technique for combining multiple decision trees. Before diving into Random Forest, we'll explore the concept of ensemble modeling, where multiple models act as a 'board of experts.' The final part of the article will cover the process of tuning a Random Forest model.

Board of experts

So far, we've been discussing the process when a client approaches a bank for a loan. They submit an application with basic information, and features are extracted from this application. These features are then fed into a decision tree, which provides a score representing the probability of the customer defaulting on the loan. Based on this score, the bank makes a lending decision.

Now, let's imagine an alternative scenario where we don't use a decision tree but rather rely on a 'board of experts,' consisting of five experts. When a customer submits an application, it's distributed to each of these experts. Each expert independently evaluates the application and decides whether to approve or reject it. The final decision is

determined by a majority vote—if the majority of experts say ‘yes,’ the bank approves the loan; if the majority says ‘no,’ the application is rejected.

The underlying concept of the ‘board of experts’ is based on the belief that the collective wisdom of five experts is more reliable than relying solely on one expert’s judgment. By aggregating multiple expert opinions, we aim to make better decisions.

This same concept can be applied to models. Instead of a ‘board of five experts,’ we can have five models (g_1, g_2, \dots, g_5), each of which returns a probability of default. We can then aggregate these model predictions by calculating the average: $(1/n) * \Sigma(p_i)$.

Ensembling models

This method of aggregating multiple models is applicable to any type of model. However, in this case, we specifically focus on using decision trees. When we ensemble decision trees, we create what is known as a ‘random forest.’

But you might wonder, why do we call it a ‘random forest’ and not just a ‘forest’? The reason is that if we take the same application (with the same set of features) and build the same group of trees with identical parameters, the resulting trees would also be identical. These identical trees would produce exactly the same probability of default, and therefore, the average would be the same as well. Essentially, we would be training the same model five times, which is not what we want.

To avoid this redundancy and bring the ‘random’ element into play, a ‘random forest’ introduces variability during the training process by using bootstrapped samples and random subsets of features, leading to a more diverse and robust ensemble of decision trees.

Random forest – ensembling decision trees

So, what exactly happens in a random forest?

In a random forest, each of the applications or features that the trees receive is slightly different. For instance, if we have a total of 10 features, each tree might receive 7 out of the 10 features, creating a distinct set for each tree.

To illustrate this with a smaller example, let’s consider 3 features: assets, debt, and price. If we train only 3 models, we might have the following feature sets for each:

- Decision Tree #1: Features – assets, debt
- Decision Tree #2: Features – assets, price
- Decision Tree #3: Features – debt, price

This way, we have 3 different models. To obtain the final prediction, we calculate the average score as $(1/3) * (p_1 + p_2 + p_3)$. While in this simplified example, the feature selection is not entirely random due to the limited number of features, the real idea is to select features randomly from a larger set.

In a random forest, each model gets a random subset of features, and the response generated by the ensemble will be a probability of default. This is the fundamental concept behind a random forest. To use it in scikit-learn, you need to import it from the ensemble package.

```
from sklearn.ensemble import RandomForestClassifier  
  
# n_estimators - number of models we want to use  
rf = RandomForestClassifier(n_estimators=10)  
rf.fit(X_train, y_train)  
  
y_pred = rf.predict_proba(X_val)[:, 1]  
roc_auc_score(y_val, y_pred)  
# Output: 0.781835024581628  
  
rf.predict_proba(X_val[[0]])  
# Output: array([[1., 0.]])
```

This model achieves an AUC score of 78.2%, which is relatively good. Notably, this performance is on par with the best decision tree model without any specific tuning. In this instance, we used the default hyperparameters and only reduced the ‘n_estimators’ value from the default of 100 to 10.

However, it’s important to recognize that a random forest introduces an element of randomness during training. Consequently, when we retrain the model and make predictions again, we may obtain slightly different results due to this randomization. To ensure consistent and reproducible results, we can use the ‘random_state’ parameter. By setting a fixed ‘random_state,’ regardless of how many times we run the model, the results will remain the same.

```
rf = RandomForestClassifier(n_estimators=10, random_state=1)  
rf.fit(X_train, y_train)  
  
y_pred = rf.predict_proba(X_val)[:, 1]  
roc_auc_score(y_val, y_pred)  
# Output: 0.7744726453706618  
  
rf.predict_proba(X_val[[0]])  
# Output: array([[0.9, 0.1]])
```

Tuning random forest

Let’s delve into the possibilities of fine-tuning our random forest model. To start, we’ll explore how the model’s performance evolves when we increase the number of estimators or models it employs. Our approach will involve iterating over a range of values to observe how the model’s performance improves or changes with an increasing number of trees.

```
scores = []

for n in range(10, 201, 10):
    rf = RandomForestClassifier(n_estimators=n, random_state=1)
    rf.fit(X_train, y_train)

    y_pred = rf.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)

    scores.append((n, auc))

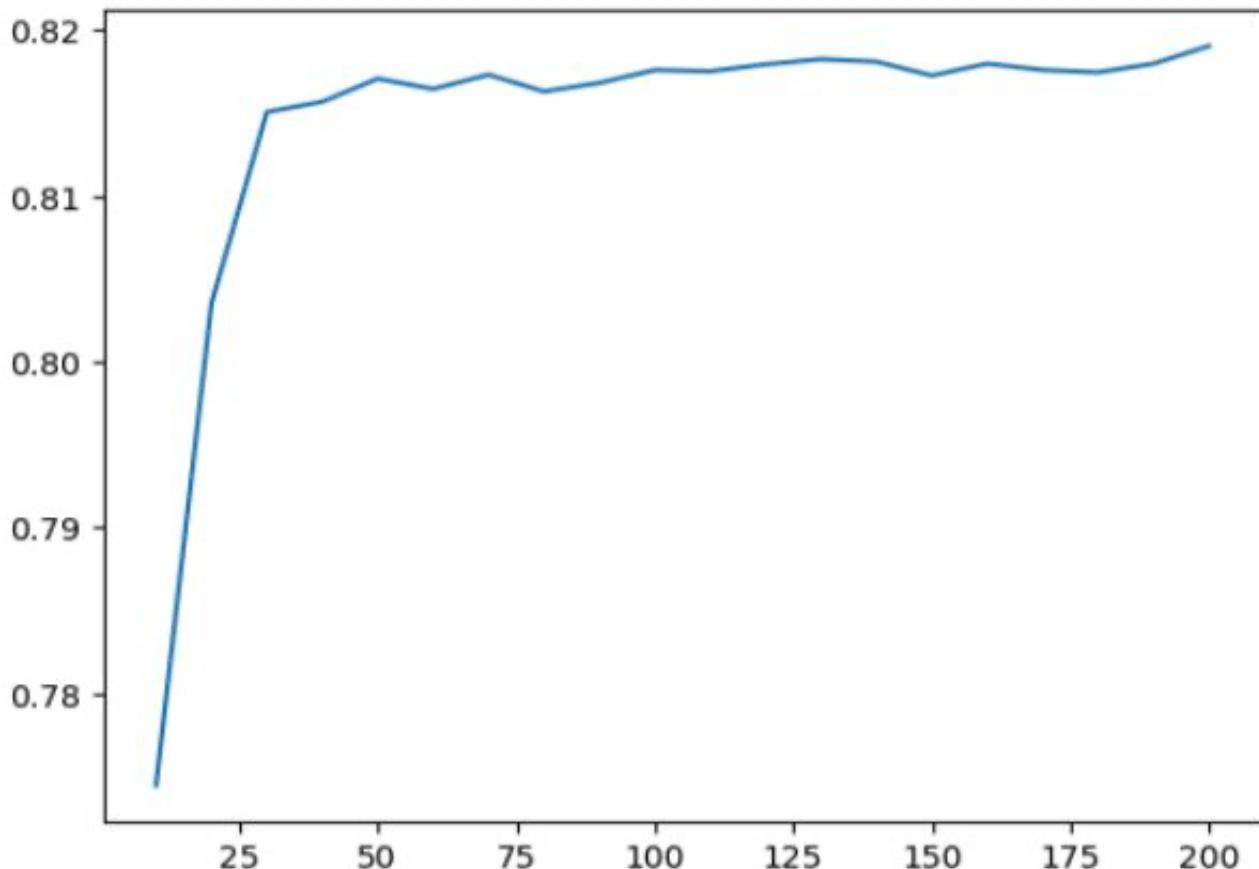
df_scores = pd.DataFrame(scores, columns=['n_estimators', 'auc'])
```

	n_estimators	auc
0	10	0.774473
1	20	0.803532
2	30	0.815075
3	40	0.815686
4	50	0.817082
5	60	0.816458
6	70	0.817321
7	80	0.816307
8	90	0.816824
9	100	0.817599
10	110	0.817527
11	120	0.817939
12	130	0.818253
13	140	0.818102
14	150	0.817270
15	160	0.817981
16	170	0.817606
17	180	0.817463
18	190	0.817981
19	200	0.819050

AUC score for different numbers of n_estimators

```
# x-axis - n_estimators  
# y-axis - auc score  
plt.plot(df_scores.n_estimators, df_scores.auc)
```

```
[<matplotlib.lines.Line2D at 0x7f2343e9f110>]
```



We observe that the model's performance improves as we increase the number of estimators up to 50, but beyond that point, it reaches a plateau. Additional trees don't significantly enhance the performance. Hence, training more than 50 trees doesn't appear to be beneficial.

Now, let's proceed with tuning our random forest model. It's important to note that a random forest comprises multiple decision trees, so the parameters we tune within the random forest are essentially the same—specifically, we are interested in the `max_depth` and `min_samples_leaf` parameters. Starting with the `max_depth` parameter, we aim to train a random forest model with different depth values to assess its impact on performance.

```
scores = []

for d in [5, 10, 15]:
    for n in range(10, 201, 10):
        rf = RandomForestClassifier(n_estimators=n,
                                   max_depth=d,
                                   random_state=1)
        rf.fit(X_train, y_train)

        y_pred = rf.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        scores.append((d, n, auc))

columns = ['max_depth', 'n_estimators', 'auc']
df_scores = pd.DataFrame(scores, columns=columns)
df_scores.head()
```

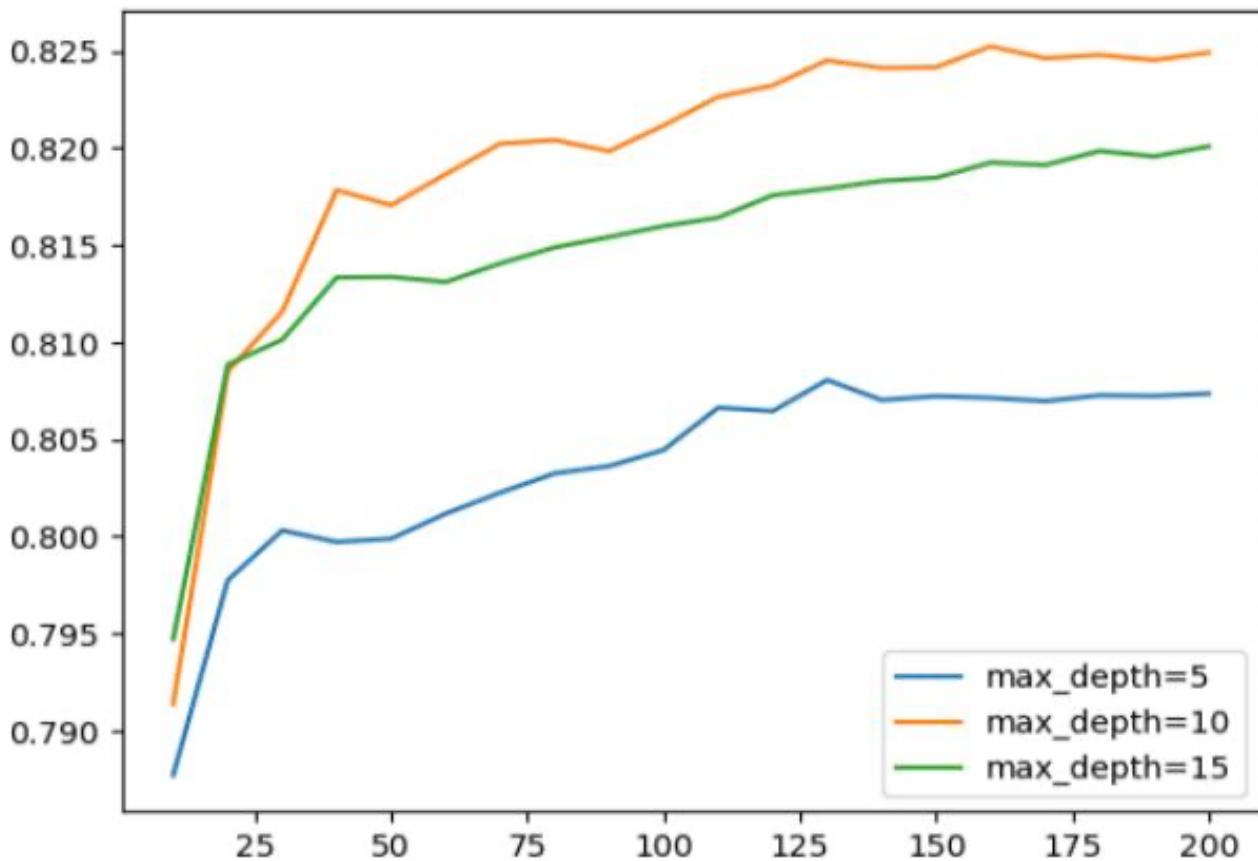
	max_depth	n_estimators	auc
0	5	10	0.787699
1	5	20	0.797731
2	5	30	0.800305
3	5	40	0.799708
4	5	50	0.799878

```
# Let's plot it
for d in [5, 10, 15]:
    df_subset = df_scores[df_scores.max_depth == d]

    plt.plot(df_subset.n_estimators, df_subset.auc,
              label='max_depth=%d' % d)

plt.legend()
```

<matplotlib.legend.Legend at 0x7f2343f26c50>



In the plot, we can observe that the AUC scores for 'max_depth' values of 10 and 15 are initially quite close, but after a certain point, the score for 'max_depth' 15 begins to level off, showing only marginal improvement. In contrast, 'max_depth' 10 continues to perform significantly better, peaking at around 125. This clearly illustrates that the choice of 'max_depth' indeed matters. We can confidently select a value of 10 as the best choice, as the difference between 10 and 15, and between 10 and 5, is notably significant.

```
# Let's select 10 as the best value  
max_depth = 10
```

Now, we'll proceed to find the optimal value for the 'min_samples_leaf' parameter using a similar method as before.

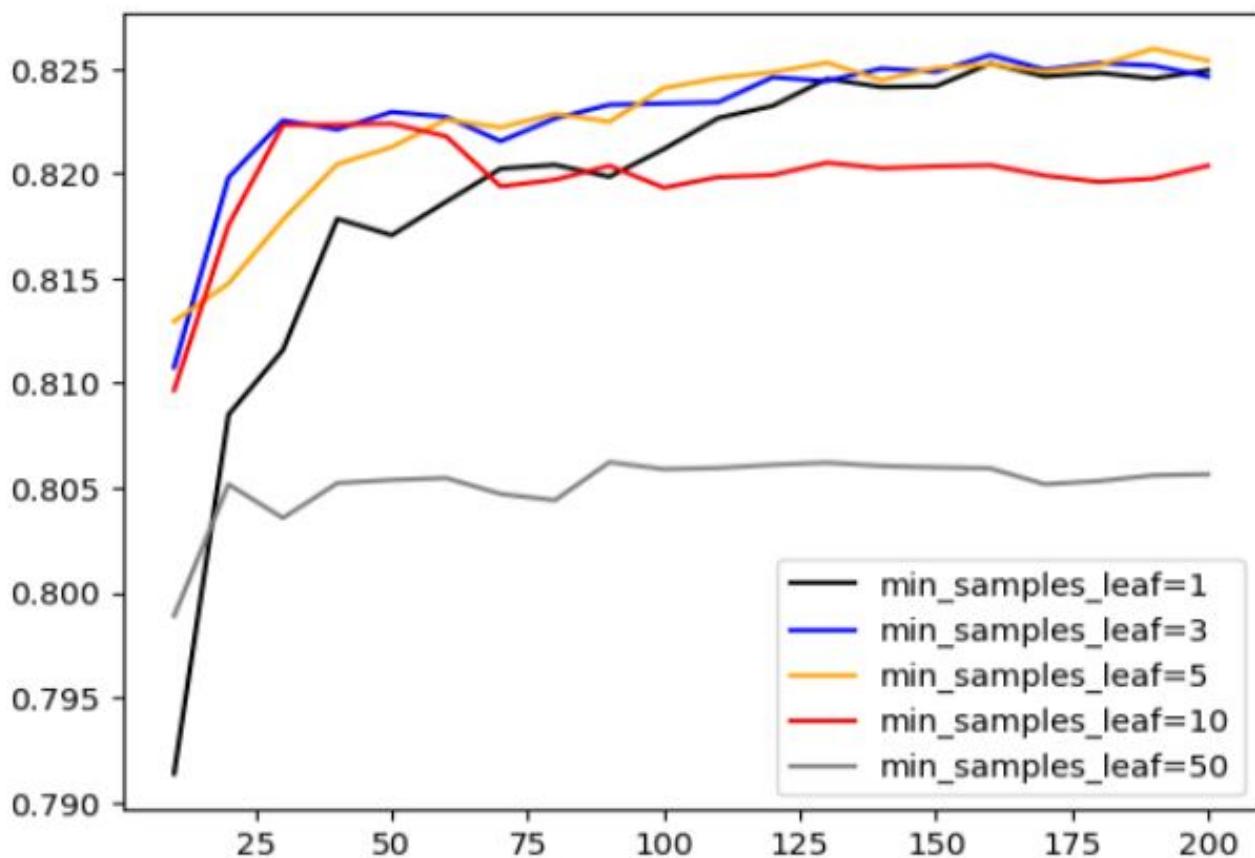
```
scores = []  
  
for s in [1, 3, 5, 10, 50]:  
    for n in range(10, 201, 10):  
        rf = RandomForestClassifier(n_estimators=n,  
                                    max_depth=max_depth,  
                                    min_samples_leaf=s,  
                                    random_state=1)  
        rf.fit(X_train, y_train)  
  
        y_pred = rf.predict_proba(X_val)[:, 1]  
        auc = roc_auc_score(y_val, y_pred)  
  
        scores.append((s, n, auc))  
  
columns = ['min_samples_leaf', 'n_estimators', 'auc']  
df_scores = pd.DataFrame(scores, columns=columns)
```

For a better distinction between the graphs in the following plot we can change the colors as you see in the next two snippets.

```
colors = ['black', 'blue', 'orange', 'red', 'grey']  
min_samples_leaf_values = [1, 3, 5, 10, 50]  
list(zip(min_samples_leaf_values, colors))  
  
# Output: [(1, 'black'), (3, 'blue'), (5, 'orange'), (10, 'red'), (50, 'grey')]
```

```
colors = ['black', 'blue', 'orange', 'red', 'grey']  
min_samples_leaf_values = [1, 3, 5, 10, 50]  
  
for s, col in zip(min_samples_leaf_values, colors):  
    df_subset = df_scores[df_scores.min_samples_leaf == s]  
  
    plt.plot(df_subset.n_estimators, df_subset.auc,  
             color=col,  
             label='min_samples_leaf=%d' % s)  
  
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f2306946c50>
```



We can observe that ‘`min_samples_leaf`’ set to 50 performs the worst, while the three most favorable options are 1, 3, and 5. Among these, ‘`min_samples_leaf`’ of 3 stands out a good choice since it achieves good performance earlier than the others.

```
# Let's select 3 as the best value  
min_samples_leaf = 3
```

Now, we’re ready to retrain the model using these selected values.

```
rf = RandomForestClassifier(n_estimators=100,  
                           max_depth=max_depth,  
                           min_samples_leaf=min_samples_leaf,  
                           random_state=1,  
                           n_jobs=-1)  
rf.fit(X_train, y_train)  
  
# Output:  
# RandomForestClassifier(max_depth=10, min_samples_leaf=3, n_jobs=  
# random_state=1)
```

These are not the only two parameters we can tune in a random forest model. There are several other useful parameters to consider:

- **max_features**: This parameter determines how many features each decision tree receives during training. It’s essential to remember that random forests work by selecting only a subset of features for each tree.
- **bootstrap**: Bootstrap introduces another form of randomization, but at the row level. This

randomization ensures that the decision trees are as diverse as possible.

- **n_jobs:** The training of decision trees can be parallelized because all the models are independent of each other. The ‘n_jobs’ parameter specifies how many trees can be trained in parallel. The default is ‘None,’ which means no parallelization. Using ‘-1’ means utilizing all available processors, which can significantly speed up the training process.”

These additional parameters offer opportunities for further fine-tuning and customization of your random forest model. For additional details and documentation on these parameters, you can refer to the [official Scikit-Learn website](#).

 [Peter](#)  [24. October 2023](#)  [Decision Trees](#), [ML-Zoomcamp](#)
 [Decision Tree](#), [ML Zoomcamp](#), [Parameter Tuning](#), [Random Forest](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

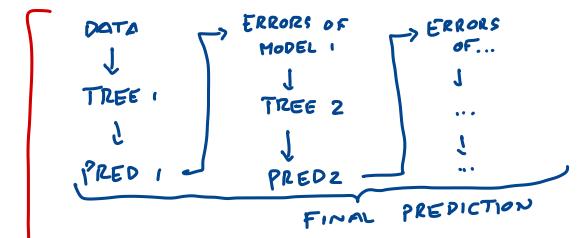
[knowMLedge.com](#), 

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning – Part 10

👤 Peter ⏰ 25. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ Decision Tree, Gradient Boosting, ML Zoomcamp, XGBoost



1. [Gradient boosting and XGBoost – Part 1/2](#)
 1. [Gradient boosting vs. random forest](#)
 2. [Installing XGBoost](#)
 3. [Training the first model](#)



Gradient boosting and XGBoost – Part 1/2

This time, we delve into a different approach for combining decision trees, where models are trained sequentially, with each new model correcting the errors of the previous one. This method of model combination is known as boosting. We will specifically explore gradient boosting and utilize the XGBoost library, which is designed for implementing the gradient boosted tree algorithm.

Gradient boosting vs. random forest

In a random forest, multiple independent decision trees are trained on the same dataset. The final prediction is achieved by aggregating the results of these individual trees, typically by taking an average: $((1/n) * \Sigma(p_i))$.

On the other hand, boosting employs a different strategy for combining multiple models into one ensemble. In boosting, we begin with the dataset and train the first model. The first model makes predictions, and we evaluate the errors made. Based on these errors, we train a second model, which generates its own predictions and, in the process, introduces its own errors. We then train a third model, which aims to correct the errors made by the second model. This process can be repeated for multiple iterations. At the end of these

iterations, we combine the predictions from these multiple models into the final prediction.

The core idea behind boosting is the sequential training of multiple models, where each subsequent model corrects the mistakes of the previous one.

Installing XGBoost

XGBoost is a library known for its highly effective implementation of gradient boosting.

```
!pip install xgboost  
import xgboost as xgb
```

Training the first model

The first step in the process is to structure the training data into a specialized data format known as ‘DMatrix.’ This format is optimized for training XGBoost models, allowing for faster training.

XGBoost Parameters – Some of the most crucial parameters include:

- **eta:** This parameter represents the learning rate, determining how quickly the model learns.
- **max_depth:** Similar to random forests and decision trees, ‘max_depth’ controls the size of the trees.
- **min_child_weight:** This parameter controls the minimum number of observations that should be present in a leaf node, similar to the ‘min_samples_leaf’ in decision trees.
- **objective:** Since we have a binary classification task, where we aim to classify clients into ‘defaulting’ or ‘non-defaulting,’ we need to specify the ‘objective.’ There are various objectives available for different types of problems, including regression and classification.
- **nthread:** XGBoost has the capability to parallelize training, and here, we specify how many threads to utilize.
- **seed:** This parameter controls the randomization used in the model.
- **verbosity:** It allows us to control the level of detail in the warnings and messages generated during training.

```
features = dv.get_feature_names()  
dtrain = xgb.DMatrix(X_train, label = y_train, feature_names = features)  
dval = xgb.DMatrix(X_val, label = y_val, feature_names = features)
```

```

xgb_params = {
    'eta': 0.3, → Default value
    'max_depth': 6, → Size Tree, default value
    'min_child_weight': 1, → Default value
    'objective': 'binary:logistic', → GBoost could be used for linear reg,
    'nthread': 8, → Sth about bin having 8 cores but here we want to do some sort of
    'seed': 1, → Show one warning. Zero shows none, binary classification, so we use that logistic
    'verbosity': 1, → Two shows many
}
model = xgb.train(xgb_params, dtrain, num_boost_round= 10)

```

Number of
Trees ↑

Now, we're ready to test the model. To do this, we can simply use the `predict` function of the XGBoost model. It returns a one-dimensional array with the model's predictions.

```
y_pred = model.predict(dval)
```

We can proceed to calculate the AUC and observe that this model achieves a value of nearly 81%. This is a commendable performance, considering that we haven't performed any specific parameter tuning; we've used the default settings. However, it's essential to exercise caution regarding the number of trees we train and the tree sizes, as XGBoost models can also be prone to overfitting, a topic we'll explore in more depth later on. Notably, in this case, the performance with `num_boost_round=10` is quite comparable to `num_boost_round=200`.

```

roc_auc_score(y_val, y_pred)
# Output: 0.8065256351262986 → 80.6% accuracy

```

• Peter ⏰ 25. October 2023 📁 Decision Trees, ML-Zoomcamp
 🔍 Decision Tree, Gradient Boosting, ML Zoomcamp, XGBoost

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 11

👤 Peter ⏰ 26. October 2023 📄 Decision Trees, ML-Zoomcamp
🏷️ [Decision Tree](#), [ML Zoomcamp](#), [Overfitting](#), [Performance Monitoring](#), [XGBoost](#)



1. [Gradient boosting and XGBoost – Part 2/2](#)
 1. [Performance Monitoring](#)
 2. [Parsing xgboost's monitoring output](#)

Gradient boosting and XGBoost – Part 2/2

This is part 2 of ‘Gradient boosting and XGBoost.’ In the first part, we compared random forests and gradient boosting, followed by the installation of XGBoost and training our first XGBoost model. In this chapter, we delve into performance monitoring.

Performance Monitoring

In **XGBoost**, it’s feasible to monitor the performance of the training process, allowing us to closely observe each stage of the training procedure. To achieve this, after each iteration where a new tree is trained, we can promptly evaluate its performance on our validation data to assess the results. For this purpose, we can establish a `watchlist` that comprises the datasets we intend to use for evaluation.

```
| watchlist = [(dtrain, 'train'), (dval, 'val')]
```

By default, XGBoost displays the error rate (logloss), a metric commonly used for parameter tuning. However, given the technical nature of this metric, we’ll opt for another more accessible metric for our analysis.

```

xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    evals=watchlist)

# Output:
# [0]  train-logloss:0.49703  val-logloss:0.54305
# [1]  train-logloss:0.44463  val-logloss:0.51462
# [2]  train-logloss:0.40707  val-logloss:0.49896
# [3]  train-logloss:0.37760  val-logloss:0.48654
# [4]  train-logloss:0.35990  val-logloss:0.48007
# [5]  train-logloss:0.33931  val-logloss:0.47563
# [6]  train-logloss:0.32586  val-logloss:0.47413
# [7]  train-logloss:0.31409  val-logloss:0.47702
# [8]  train-logloss:0.29962  val-logloss:0.48205
# [9]  train-logloss:0.29216  val-logloss:0.47996
# [10] train-logloss:0.28407  val-logloss:0.47969
#
# ...
# [195] train-logloss:0.02736  val-logloss:0.67492
# [196] train-logloss:0.02728  val-logloss:0.67518
# [197] train-logloss:0.02693  val-logloss:0.67791
# [198] train-logloss:0.02665  val-logloss:0.67965
# [199] train-logloss:0.02642  val-logloss:0.68022

```

For our monitoring purposes, we choose to use AUC as the metric, which we've previously employed. To do this, we set the `eval_metric` parameter to `auc`.

```

xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                   evals=watchlist)

# Output:
# [0]  train-auc:0.86730  val-auc:0.77938
# [1]  train-auc:0.89140  val-auc:0.78964
# [2]  train-auc:0.90699  val-auc:0.79010
# [3]  train-auc:0.91677  val-auc:0.79967
# [4]  train-auc:0.92246  val-auc:0.80443
# [5]  train-auc:0.93086  val-auc:0.80858
# [6]  train-auc:0.93675  val-auc:0.80981
# [7]  train-auc:0.94108  val-auc:0.80872
# [8]  train-auc:0.94809  val-auc:0.80456
# [9]  train-auc:0.95100  val-auc:0.80653
# [10] train-auc:0.95447  val-auc:0.80851
#
# ...
# [195] train-auc:1.00000  val-auc:0.80708
# [196] train-auc:1.00000  val-auc:0.80759
# [197] train-auc:1.00000  val-auc:0.80718
# [198] train-auc:1.00000  val-auc:0.80719
# [199] train-auc:1.00000  val-auc:0.80725

```

The AUC on the training data reaches perfect accuracy (equal to one), but on the validation data, the performance remains relatively stable at around 80%. This suggests that the model may be overfitting.

To make this output more user-friendly, it would be beneficial to visualize it. Instead of printing output for every epoch, we can use `verbose_eval=5` to display results only for every 5th epoch, making the monitoring process more manageable.

```

xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                   verbose_eval=5,
                   evals=watchlist)

# Output:
# [0]      train-auc:0.86730  val-auc:0.77938
# [5]      train-auc:0.93086  val-auc:0.80858
# [10]     train-auc:0.95447  val-auc:0.80851
# [15]     train-auc:0.96554  val-auc:0.81334
# [20]     train-auc:0.97464  val-auc:0.81729
# [25]     train-auc:0.97953  val-auc:0.81686
# [30]     train-auc:0.98579  val-auc:0.81543
# [35]     train-auc:0.99011  val-auc:0.81206
# [40]     train-auc:0.99421  val-auc:0.80922
# [45]     train-auc:0.99548  val-auc:0.80842
# [50]     train-auc:0.99653  val-auc:0.80918
# [55]     train-auc:0.99765  val-auc:0.81114
# [60]     train-auc:0.99817  val-auc:0.81172
# [65]     train-auc:0.99887  val-auc:0.80798
# [70]     train-auc:0.99934  val-auc:0.80870
# [75]     train-auc:0.99965  val-auc:0.80555
# [80]     train-auc:0.99979  val-auc:0.80549
# [85]     train-auc:0.99988  val-auc:0.80374
# [90]     train-auc:0.99993  val-auc:0.80409
# [95]     train-auc:0.99996  val-auc:0.80548
# [100]    train-auc:0.99998  val-auc:0.80509
# [105]    train-auc:0.99999  val-auc:0.80629
# [110]    train-auc:1.00000  val-auc:0.80637
# [115]    train-auc:1.00000  val-auc:0.80494
# [120]    train-auc:1.00000  val-auc:0.80574
# [125]    train-auc:1.00000  val-auc:0.80727
# [130]    train-auc:1.00000  val-auc:0.80746
# [135]    train-auc:1.00000  val-auc:0.80753
# [140]    train-auc:1.00000  val-auc:0.80899
# [145]    train-auc:1.00000  val-auc:0.80733
# [150]    train-auc:1.00000  val-auc:0.80841
# [155]    train-auc:1.00000  val-auc:0.80734
# [160]    train-auc:1.00000  val-auc:0.80711
# [165]    train-auc:1.00000  val-auc:0.80707
# [170]    train-auc:1.00000  val-auc:0.80734
# [175]    train-auc:1.00000  val-auc:0.80704
# [180]    train-auc:1.00000  val-auc:0.80723
# [185]    train-auc:1.00000  val-auc:0.80678
# [190]    train-auc:1.00000  val-auc:0.80672
# [195]    train-auc:1.00000  val-auc:0.80708
# [199]    train-auc:1.00000  val-auc:0.80725

```

Parsing xgboost's monitoring output

When you're interested in visualizing this information on a plot, one of the challenges with XGBoost is that it doesn't provide an easy way to extract this information since it's printed to standard output. However, in Jupyter Notebook, there's a method to capture whatever is printed to standard output and manipulate it. You can use the command `%%capture output` to achieve this. It captures all the content that the code outputs into a special object, which you can then use to extract the information. It's worth noting that although something is happening in the code, we won't see any output because it's being captured.

```
%%capture output

xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                   verbose_eval=5,
                   evals=watchlist)
```

```

s = output.stdout

print(s)
# Output:
# [0]      train-auc:0.86730    val-auc:0.77938
# [5]      train-auc:0.93086    val-auc:0.80858
# [10]     train-auc:0.95447    val-auc:0.80851
# [15]     train-auc:0.96554    val-auc:0.81334
# [20]     train-auc:0.97464    val-auc:0.81729
# [25]     train-auc:0.97953    val-auc:0.81686
# [30]     train-auc:0.98579    val-auc:0.81543
# [35]     train-auc:0.99011    val-auc:0.81206
# [40]     train-auc:0.99421    val-auc:0.80922
# [45]     train-auc:0.99548    val-auc:0.80842
# [50]     train-auc:0.99653    val-auc:0.80918
# [55]     train-auc:0.99765    val-auc:0.81114
# [60]     train-auc:0.99817    val-auc:0.81172
# [65]     train-auc:0.99887    val-auc:0.80798
# [70]     train-auc:0.99934    val-auc:0.80870
# [75]     train-auc:0.99965    val-auc:0.80555
# [80]     train-auc:0.99979    val-auc:0.80549
# [85]     train-auc:0.99988    val-auc:0.80374
# [90]     train-auc:0.99993    val-auc:0.80409
# [95]     train-auc:0.99996    val-auc:0.80548
# [100]    train-auc:0.99998    val-auc:0.80509
# [105]    train-auc:0.99999    val-auc:0.80629
# [110]    train-auc:1.00000    val-auc:0.80637
# [115]    train-auc:1.00000    val-auc:0.80494
# [120]    train-auc:1.00000    val-auc:0.80574
# [125]    train-auc:1.00000    val-auc:0.80727
# [130]    train-auc:1.00000    val-auc:0.80746
# [135]    train-auc:1.00000    val-auc:0.80753
# [140]    train-auc:1.00000    val-auc:0.80899
# [145]    train-auc:1.00000    val-auc:0.80733
# [150]    train-auc:1.00000    val-auc:0.80841
# [155]    train-auc:1.00000    val-auc:0.80734
# [160]    train-auc:1.00000    val-auc:0.80711
# [165]    train-auc:1.00000    val-auc:0.80707
# [170]    train-auc:1.00000    val-auc:0.80734
# [175]    train-auc:1.00000    val-auc:0.80704
# [180]    train-auc:1.00000    val-auc:0.80723
# [185]    train-auc:1.00000    val-auc:0.80678
# [190]    train-auc:1.00000    val-auc:0.80672
# [195]    train-auc:1.00000    val-auc:0.80708
# [199]    train-auc:1.00000    val-auc:0.80725

```

Now that we have the captured output in a string, the first step is to split it into individual lines by using the new line operator `\n`. The result is a string for each line of the output.

```

s.split('\n')

# Output:
# ['[0]\ttrain-auc:0.86730\tval-auc:0.77938',
#  '[5]\ttrain-auc:0.93086\tval-auc:0.80858',
#  '[10]\ttrain-auc:0.95447\tval-auc:0.80851',
#  '[15]\ttrain-auc:0.96554\tval-auc:0.81334',
#  '[20]\ttrain-auc:0.97464\tval-auc:0.81729',
#  '[25]\ttrain-auc:0.97953\tval-auc:0.81686',
#  '[30]\ttrain-auc:0.98579\tval-auc:0.81543',
#  '[35]\ttrain-auc:0.99011\tval-auc:0.81206',
#  '[40]\ttrain-auc:0.99421\tval-auc:0.80922',
#  '[45]\ttrain-auc:0.99548\tval-auc:0.80842',
#  '[50]\ttrain-auc:0.99653\tval-auc:0.80918',
#  '[55]\ttrain-auc:0.99765\tval-auc:0.81114',
#  '[60]\ttrain-auc:0.99817\tval-auc:0.81172',
#  '[65]\ttrain-auc:0.99887\tval-auc:0.80798',
#  '[70]\ttrain-auc:0.99934\tval-auc:0.80870',
#  '[75]\ttrain-auc:0.99965\tval-auc:0.80555',
#  '[80]\ttrain-auc:0.99979\tval-auc:0.80549',
#  '[85]\ttrain-auc:0.99988\tval-auc:0.80374',
#  '[90]\ttrain-auc:0.99993\tval-auc:0.80409',
#  '[95]\ttrain-auc:0.99996\tval-auc:0.80548',
#  '[100]\ttrain-auc:0.99998\tval-auc:0.80509',
#  '[105]\ttrain-auc:0.99999\tval-auc:0.80629',
#  '[110]\ttrain-auc:1.00000\tval-auc:0.80637',
#  '[115]\ttrain-auc:1.00000\tval-auc:0.80494',
#  '[120]\ttrain-auc:1.00000\tval-auc:0.80574',
#  '[125]\ttrain-auc:1.00000\tval-auc:0.80727',
#  '[130]\ttrain-auc:1.00000\tval-auc:0.80746',
#  '[135]\ttrain-auc:1.00000\tval-auc:0.80753',
#  '[140]\ttrain-auc:1.00000\tval-auc:0.80899',
#  '[145]\ttrain-auc:1.00000\tval-auc:0.80733',
#  '[150]\ttrain-auc:1.00000\tval-auc:0.80841',
#  '[155]\ttrain-auc:1.00000\tval-auc:0.80734',
#  '[160]\ttrain-auc:1.00000\tval-auc:0.80711',
#  '[165]\ttrain-auc:1.00000\tval-auc:0.80707',
#  '[170]\ttrain-auc:1.00000\tval-auc:0.80734',
#  '[175]\ttrain-auc:1.00000\tval-auc:0.80704',
#  '[180]\ttrain-auc:1.00000\tval-auc:0.80723',
#  '[185]\ttrain-auc:1.00000\tval-auc:0.80678',
#  '[190]\ttrain-auc:1.00000\tval-auc:0.80672',
#  '[195]\ttrain-auc:1.00000\tval-auc:0.80708',
#  '[199]\ttrain-auc:1.00000\tval-auc:0.80725',
#  '']

```

Each line consists of three components: the number of iterations, the evaluation on the training dataset, and the evaluation on the validation dataset. We can split these components using the tabulator operator \t, resulting in three separate components. To ensure the correct format (integer, float, float), we utilize the `strip` method and perform the necessary string-to-integer and string-to-float conversions. The following snippet demonstrates these steps.

```

line = s.split('\n')[0]
line
# Output: '[0]\ttrain-auc:0.86730\tval-auc:0.77938'

line.split('\t') ↪ Easier To read
# Output: ['[0]', 'train-auc:0.86730', 'val-auc:0.77938']

num_iter, train_auc, val_auc = line.split('\t')
num_iter, train_auc, val_auc
# Output: ('[0]', 'train-auc:0.86730', 'val-auc:0.77938')

int(num_iter.strip('[]'))
# Output: 0
float(train_auc.split(':')[1])
# Output: 0.8673
float(val_auc.split(':')[1])
# Output: 0.77938

```

Even easier To
read

We can combine all these steps to transform the information (number of iterations, AUC on the training data, and AUC on the validation data) from the output into a dataframe. The following snippet encapsulates all these steps within a single function for ease of use. This allows us to plot the data and perform further analysis.

```

def parse_xgb_output(output):
    results = []

    for line in output.stdout.strip().split('\n'):
        it_line, train_line, val_line = line.split('\t')

        it = int(it_line.strip('[]'))
        train = float(train_line.split(':')[1])
        val = float(val_line.split(':')[1])

        results.append((it, train, val))

    columns = ['num_iter', 'train_auc', 'val_auc']
    df_results = pd.DataFrame(results, columns=columns)
    return df_results

```

Now, let's see how the function works in action.

```

df_score = parse_xgb_output(output)
df_score

```

	num_iter	train_auc	val_auc
0	0	0.86730	0.77938
1	5	0.93086	0.80858
2	10	0.95447	0.80851
3	15	0.96554	0.81334

4	20	0.97464	0.81729
5	25	0.97953	0.81686
6	30	0.98579	0.81543
7	35	0.99011	0.81206
8	40	0.99421	0.80922
9	45	0.99548	0.80842
10	50	0.99653	0.80918
11	55	0.99765	0.81114
12	60	0.99817	0.81172
13	65	0.99887	0.80798
14	70	0.99934	0.80870
15	75	0.99965	0.80555
16	80	0.99979	0.80549
17	85	0.99988	0.80374
18	90	0.99993	0.80409
19	95	0.99996	0.80548
20	100	0.99998	0.80509
21	105	0.99999	0.80629
22	110	1.00000	0.80637
23	115	1.00000	0.80494
24	120	1.00000	0.80574
25	125	1.00000	0.80727
26	130	1.00000	0.80746
27	135	1.00000	0.80753
28	140	1.00000	0.80899
29	145	1.00000	0.80733
30	150	1.00000	0.80841

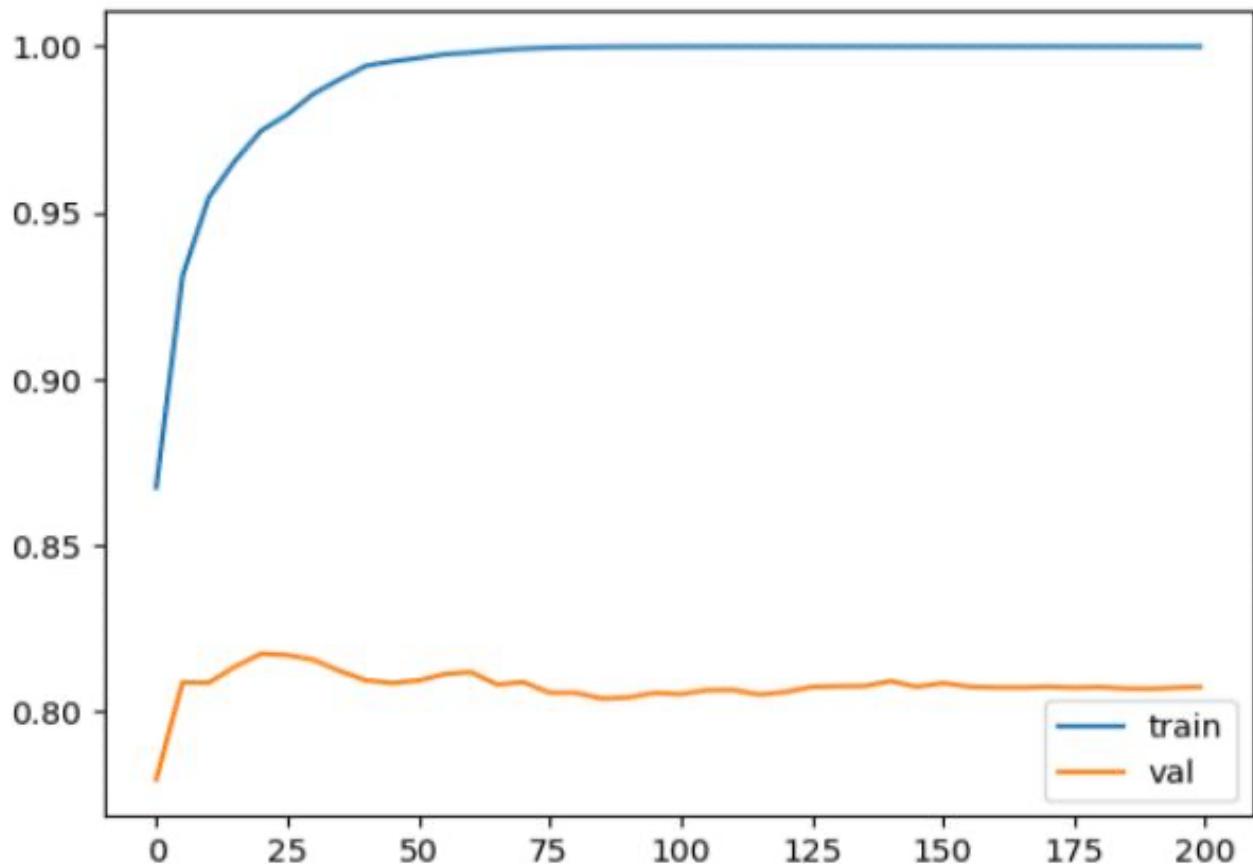
31	155	1.00000	0.80734
32	160	1.00000	0.80711
33	165	1.00000	0.80707
34	170	1.00000	0.80734
35	175	1.00000	0.80704
36	180	1.00000	0.80723
37	185	1.00000	0.80678
38	190	1.00000	0.80672
39	195	1.00000	0.80708
40	199	1.00000	0.80725

Output of parse_xgb_output function

The result of the `parse_xgb_output` function is a dataframe, enabling us to utilize the `plot` function for graph visualization.

```
# x-axis - number of iterations
# y-axis - auc
plt.plot(df_score.num_iter, df_score.train_auc, label='train')
plt.plot(df_score.num_iter, df_score.val_auc, label='val')
plt.legend()
```

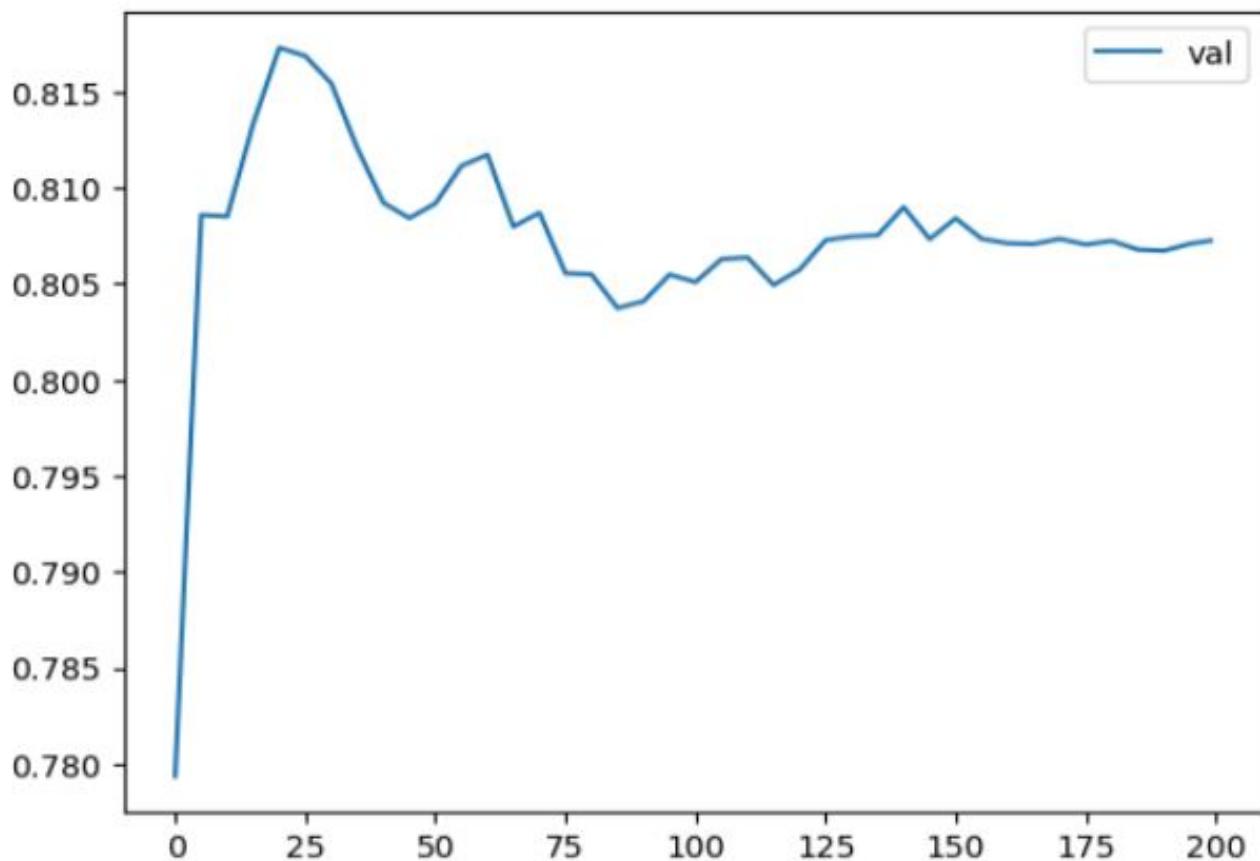
```
<matplotlib.legend.Legend at 0x7f23068c8e50>
```



We can observe that the AUC on the training dataset consistently improves. However, the picture is different for the validation dataset. The curve reaches its peak earlier and then starts to decline and stagnate, indicating the onset of **overfitting**. This decline in performance on the validation dataset is more apparent when plotting only the AUC on validation, while the AUC on the training dataset remains consistently high. The decline in performance is more evident when we exclusively plot the validation graph.

```
| plt.plot(df_score.num_iter, df_score.val_auc, label='val')  
| plt.legend()
```

<matplotlib.legend.Legend at 0x7f2333f58150>



• Peter ⏰ 26. October 2023 📄 Decision Trees, ML-Zoomcamp
🏷️ Decision Tree, ML Zoomcamp, Overfitting, Performance Monitoring, XGBoost

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

[knowMLedge.com](#), 

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 12

👤 Peter ⏰ 27. October 2023 📁 Decision Trees, ML-Zoomcamp
🏷️ Decision Tree, Learning Rate, ML Zoomcamp, Parameter Tuning, XGBoost



1. [XGBoost parameter tuning – Part 1/2](#)

1. [Tuning Eta](#)

1. [Eta = 0.3](#)
2. [Eta = 1.0](#)
3. [Eta = 0.1](#)
4. [Eta = 0.05](#)
5. [Eta = 0.01](#)

2. [Plotting Eta](#)

XGBoost parameter tuning – Part 1/2

This part is about **XGBoost** parameter tuning. It's the first part of a two-part series, where we begin by tuning the initial parameter – ‘eta’. The subsequent article will explore parameter tuning for ‘max_depth’ and ‘min_child_weight’. In the final phase, we'll train the final model. Let's start tuning the first parameter.

Tuning Eta

Eta, also known as the learning rate, determines the influence of the following model when correcting the results of the previous model. If the weight is set to 1.0, all new predictions are used to correct the previous ones. However, when the weight is 0.3, only 30% of the new predictions are considered. In essence, eta governs the size of the steps taken during the learning process.

Now, let's explore how different values of eta impact model performance. To facilitate this,

we'll create a dictionary called 'scores' to store the performance scores for each value of eta.

Eta = 0.3

```
| scores = {}
```

```
%%capture output

xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    verbose_eval=5,
                    evals=watchlist)
```

We aim to structure keys in the format 'eta=0.3' to serve as identifiers in the scores dictionary.

```
| 'eta=%s' % (xgb_params['eta'])
# Output: 'eta=0.3'
```

In the next snippet, we once again employ the 'parse_xgb_output' function, which we defined in the previous article. This function returns a dataframe that contains train_auc and val_auc values for different num_iter.

```
key = 'eta=%s' % (xgb_params['eta'])
scores[key] = parse_xgb_output(output)
key

# Output: 'eta=0.3'
```

Now the dictionary scores contains the dataframe for this eta.

```
scores
```

```
# Output:  
# {'eta=0.3':      num_iter  train_auc  val_auc  
#  0              0    0.86730  0.77938  
#  1              5    0.93086  0.80858  
#  2             10   0.95447  0.80851  
#  3             15   0.96554  0.81334  
#  4             20   0.97464  0.81729  
#  5             25   0.97953  0.81686  
#  
#  ...  
#  36            180  1.00000  0.80723  
#  37            185  1.00000  0.80678  
#  38            190  1.00000  0.80672  
#  39            195  1.00000  0.80708  
#  40            199  1.00000  0.80725}
```

```
| scores['eta=0.3']
```

	num_iter	train_auc	val_auc
0	0	0.86730	0.77938
1	5	0.93086	0.80858
2	10	0.95447	0.80851
3	15	0.96554	0.81334
4	20	0.97464	0.81729
5	25	0.97953	0.81686
6	30	0.98579	0.81543
7	35	0.99011	0.81206
8	40	0.99421	0.80922
9	45	0.99548	0.80842
10	50	0.99653	0.80918
11	55	0.99765	0.81114
12	60	0.99817	0.81172
13	65	0.99887	0.80798
14	70	0.99934	0.80870
15	75	0.99965	0.80555

16	80	0.99979	0.80549
17	85	0.99988	0.80374
18	90	0.99993	0.80409
19	95	0.99996	0.80548
20	100	0.99998	0.80509
21	105	0.99999	0.80629
22	110	1.00000	0.80637
23	115	1.00000	0.80494
24	120	1.00000	0.80574
25	125	1.00000	0.80727
26	130	1.00000	0.80746
27	135	1.00000	0.80753
28	140	1.00000	0.80899
29	145	1.00000	0.80733
30	150	1.00000	0.80841
31	155	1.00000	0.80734
32	160	1.00000	0.80711
33	165	1.00000	0.80707
34	170	1.00000	0.80734
35	175	1.00000	0.80704
36	180	1.00000	0.80723
37	185	1.00000	0.80678
38	190	1.00000	0.80672
39	195	1.00000	0.80708
40	199	1.00000	0.80725

Dataframe for 'eta=0.3'

Eta = 1.0

Now, let's set the eta value to its maximum, which is 1.0.

```
%%capture output
xgb_params = {
    'eta': 1.0,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}
model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    verbose_eval=5,
                    evals=watchlist)
```

```
key = 'eta=%s' % (xgb_params['eta'])
scores[key] = parse_xgb_output(output)
key
# Output: 'eta=1.0'
```

With the updated eta value, the scores dictionary should now encompass two values and two corresponding keys.

```
scores['eta=1.0']
```

	num_iter	train_auc	val_auc
0	0	0.86730	0.77938
1	5	0.95857	0.79136
2	10	0.98061	0.78355
3	15	0.99549	0.78050
4	20	0.99894	0.78591
5	25	0.99989	0.78401
6	30	1.00000	0.78371
7	35	1.00000	0.78234
8	40	1.00000	0.78184

9	45	1.00000	0.77963
10	50	1.00000	0.78645
11	55	1.00000	0.78644
12	60	1.00000	0.78545
13	65	1.00000	0.78612
14	70	1.00000	0.78515
15	75	1.00000	0.78516
16	80	1.00000	0.78420
17	85	1.00000	0.78570
18	90	1.00000	0.78793
19	95	1.00000	0.78865
20	100	1.00000	0.79075
21	105	1.00000	0.79107
22	110	1.00000	0.79022
23	115	1.00000	0.79036
24	120	1.00000	0.79021
25	125	1.00000	0.79025
26	130	1.00000	0.78994
27	135	1.00000	0.79084
28	140	1.00000	0.79048
29	145	1.00000	0.78967
30	150	1.00000	0.78969
31	155	1.00000	0.78992
32	160	1.00000	0.79064
33	165	1.00000	0.79067
34	170	1.00000	0.79115
35	175	1.00000	0.79126

36	180	1.00000	0.79199
37	185	1.00000	0.79179
38	190	1.00000	0.79165
39	195	1.00000	0.79201
40	199	1.00000	0.79199

Dataframe for 'eta=1.0'

Eta = 0.1

Let's go through the process once more for 'eta=0.1' and subsequently print out the dataframe.

```
%capture output
xgb_params = {
    'eta': 0.1,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}
model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    verbose_eval=5,
                    evals=watchlist)
```

```
key = 'eta=%s' % (xgb_params['eta'])
scores[key] = parse_xgb_output(output)
key
# Output: 'eta=0.1'
```

```
scores['eta=0.1']
```

	num_iter	train_auc	val_auc
0	0	0.86730	0.77938
1	5	0.90325	0.79290

2	10	0.91874	0.80510
3	15	0.93126	0.81380
4	20	0.93873	0.81804
5	25	0.94638	0.82065
6	30	0.95338	0.82063
7	35	0.95874	0.82404
8	40	0.96325	0.82644
9	45	0.96694	0.82602
10	50	0.97195	0.82549
11	55	0.97475	0.82648
12	60	0.97708	0.82781
13	65	0.97937	0.82775
14	70	0.98214	0.82681
15	75	0.98315	0.82728
16	80	0.98517	0.82560
17	85	0.98721	0.82503
18	90	0.98840	0.82443
19	95	0.98972	0.82389
20	100	0.99061	0.82456
21	105	0.99157	0.82359
22	110	0.99224	0.82274
23	115	0.99288	0.82147
24	120	0.99378	0.82154
25	125	0.99481	0.82195
26	130	0.99541	0.82252
27	135	0.99564	0.82190
28	140	0.99630	0.82219

29	145	0.99673	0.82177
30	150	0.99711	0.82136
31	155	0.99750	0.82154
32	160	0.99774	0.82102
33	165	0.99821	0.82060
34	170	0.99838	0.82060
35	175	0.99861	0.82012
36	180	0.99882	0.82053
37	185	0.99898	0.82028
38	190	0.99904	0.81973
39	195	0.99920	0.81909
40	199	0.99927	0.81864

Dataframe for 'eta=0.1'

Eta = 0.05

Let's do it again for 'eta=0.05' and print out the dataframe.

```
%%capture output
xgb_params = {
    'eta': 0.05,
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}
model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                   verbose_eval=5,
                   evals=watchlist)
```

```
key = 'eta=%s' % (xgb_params['eta'])
scores[key] = parse_xgb_output(output)
key
# Output: 'eta=0.05'
```

```
scores['eta=0.05']
```

	num_iter	train_auc	val_auc
0	0	0.86730	0.77938
1	5	0.88650	0.79584
2	10	0.90368	0.79623
3	15	0.91072	0.79938
4	20	0.91774	0.80510
5	25	0.92385	0.80895
6	30	0.92987	0.81175
7	35	0.93379	0.81480
8	40	0.93856	0.81547
9	45	0.94316	0.81807
10	50	0.94753	0.81793
11	55	0.95028	0.81926
12	60	0.95324	0.81998
13	65	0.95581	0.82159
14	70	0.95762	0.82299
15	75	0.95944	0.82368
16	80	0.96100	0.82524
17	85	0.96308	0.82604
18	90	0.96572	0.82666
19	95	0.96798	0.82667
20	100	0.96955	0.82719

21	105	0.97133	0.82745
22	110	0.97288	0.82819
23	115	0.97426	0.82822
24	120	0.97578	0.82768
25	125	0.97702	0.82790
26	130	0.97788	0.82760
27	135	0.97923	0.82764
28	140	0.98012	0.82725
29	145	0.98113	0.82665
30	150	0.98190	0.82575
31	155	0.98285	0.82581
32	160	0.98381	0.82560
33	165	0.98457	0.82576
34	170	0.98541	0.82591
35	175	0.98652	0.82581
36	180	0.98711	0.82526
37	185	0.98789	0.82525
38	190	0.98876	0.82535
39	195	0.98932	0.82538
40	199	0.98977	0.82522

Dataframe for ‘eta=0.05’

Eta = 0.01

Once more, let’s assess the performance for ‘eta=0.01’.

%%capture output

```
xgb_params = {  
    'eta': 0.01,  
    'max_depth': 6,  
    'min_child_weight': 1,  
  
    'objective': 'binary:logistic',  
    'eval_metric': 'auc',  
  
    'nthread': 8,  
    'seed': 1,  
    'verbosity': 1,  
}  
  
model = xgb.train(xgb_params, dtrain, num_boost_round=200,  
                   verbose_eval=5,  
                   evals=watchlist)
```

```
key = 'eta=%s' %% (xgb_params['eta'])  
scores[key] = parse_xgb_output(output)  
key  
  
# Output: 'eta=0.01'
```

```
scores['eta=0.01']
```

	num_iter	train_auc	val_auc
0	0	0.86730	0.77938
1	5	0.87157	0.77925
2	10	0.87247	0.78051
3	15	0.87541	0.78302
4	20	0.87584	0.78707
5	25	0.88406	0.79331
6	30	0.89027	0.79763
7	35	0.89559	0.79914
8	40	0.89782	0.79883
9	45	0.89983	0.79845
10	50	0.90182	0.79697
11	55	0.90394	0.79775

12	60	0.90531	0.79684
13	65	0.90630	0.79616
14	70	0.90796	0.79672
15	75	0.90955	0.79807
16	80	0.91116	0.79976
17	85	0.91227	0.80130
18	90	0.91368	0.80285
19	95	0.91515	0.80390
20	100	0.91654	0.80499
21	105	0.91791	0.80534
22	110	0.91902	0.80523
23	115	0.92032	0.80515
24	120	0.92135	0.80497
25	125	0.92262	0.80520
26	130	0.92405	0.80605
27	135	0.92547	0.80692
28	140	0.92667	0.80730
29	145	0.92771	0.80818
30	150	0.92919	0.80928
31	155	0.92990	0.81039
32	160	0.93077	0.81125
33	165	0.93155	0.81160
34	170	0.93246	0.81189
35	175	0.93348	0.81257
36	180	0.93466	0.81340
37	185	0.93585	0.81429
38	190	0.93685	0.81481

39	195	0.93777	0.81554
40	199	0.93862	0.81575

Dataframe for 'eta=0.01'

Plotting Eta

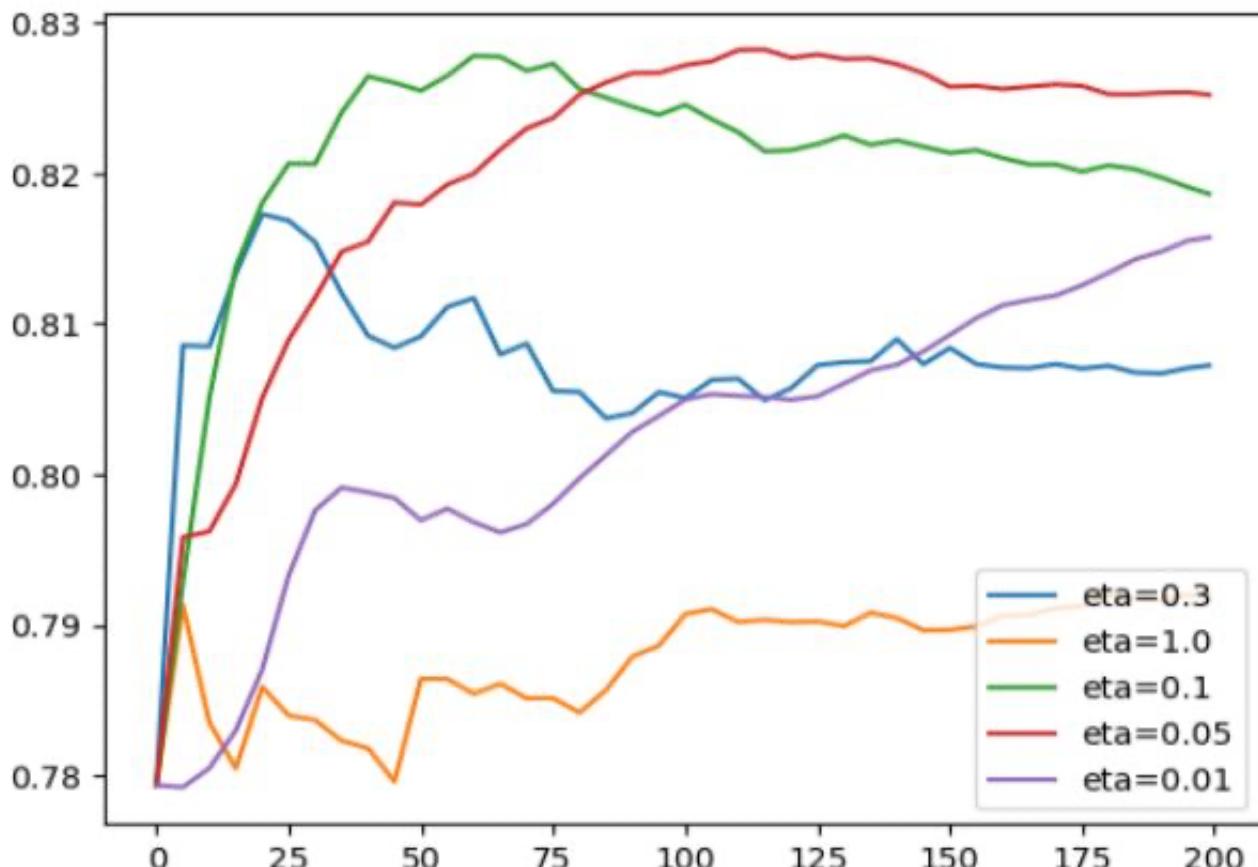
Now that we've inserted key-value pairs and gathered information from different runs, we can examine the keys in the dictionary. Next we can compare all runs of 'eta=0.3', 'eta=1.0', 'eta=0.1', 'eta=0.05', and 'eta=0.01'.

```
scores.keys()
# Output: dict_keys(['eta=0.3', 'eta=1.0', 'eta=0.1', 'eta=0.05', 'eta=0.01'])
```

Let's plot the information of our runs.

```
for key, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=key)
plt.legend()
```

<matplotlib.legend.Legend at 0x7f2333e05f50>



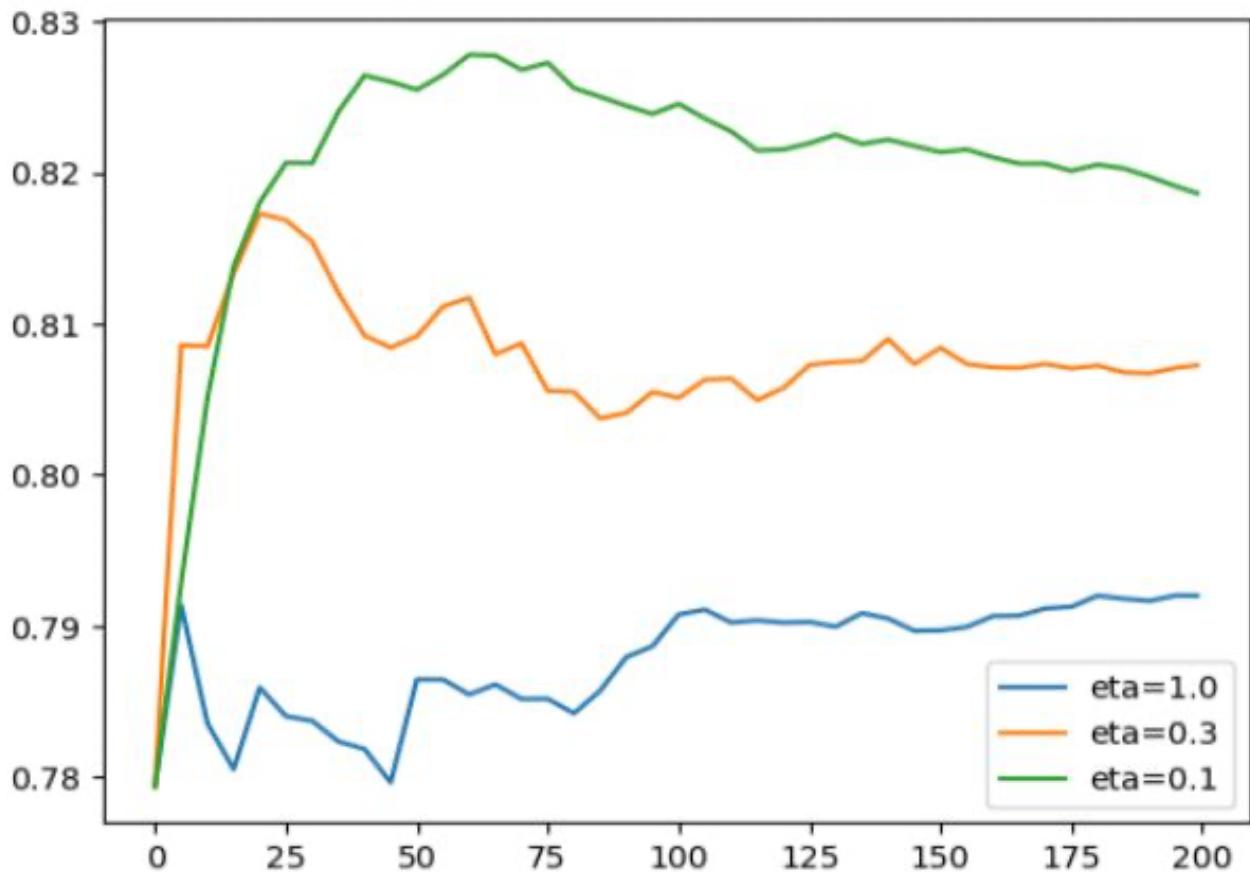
Let's concentrate on a few graphs for the initial analysis. We will plot three of them: 'eta=1.0', 'eta=0.3', and 'eta=0.1'.

```

etas = ['eta=1.0', 'eta=0.3', 'eta=0.1']
for eta in etas:
    df_score = scores[eta]
    plt.plot(df_score.num_iter, df_score.val_auc, label=eta)
plt.legend()

```

<matplotlib.legend.Legend at 0x7f23068eead0>



This plot provides a clearer view of the results. Notably, ‘eta=1.0’ exhibits the worst performance. It quickly reaches peak performance but then experiences a sharp decline, maintaining a consistently poor level. ‘eta=0.3’ performs reasonably well until around iteration 25, after which it steadily deteriorates. On the other hand, ‘eta=0.1’ demonstrates a slower growth rate, reaching its peak at a later stage before descending. This pattern is a direct reflection of the learning rate’s influence.

The learning rate controls both the speed at which the model learns and the size of the steps it takes during each iteration. If the steps are too large, the model learns rapidly but eventually starts to degrade due to the excessive step size, resulting in **overfitting**. Conversely, a smaller learning rate signifies slower but more stable learning. Such models tend to degrade more gradually, and their overfitting tendencies are less pronounced compared to models with higher learning rates.

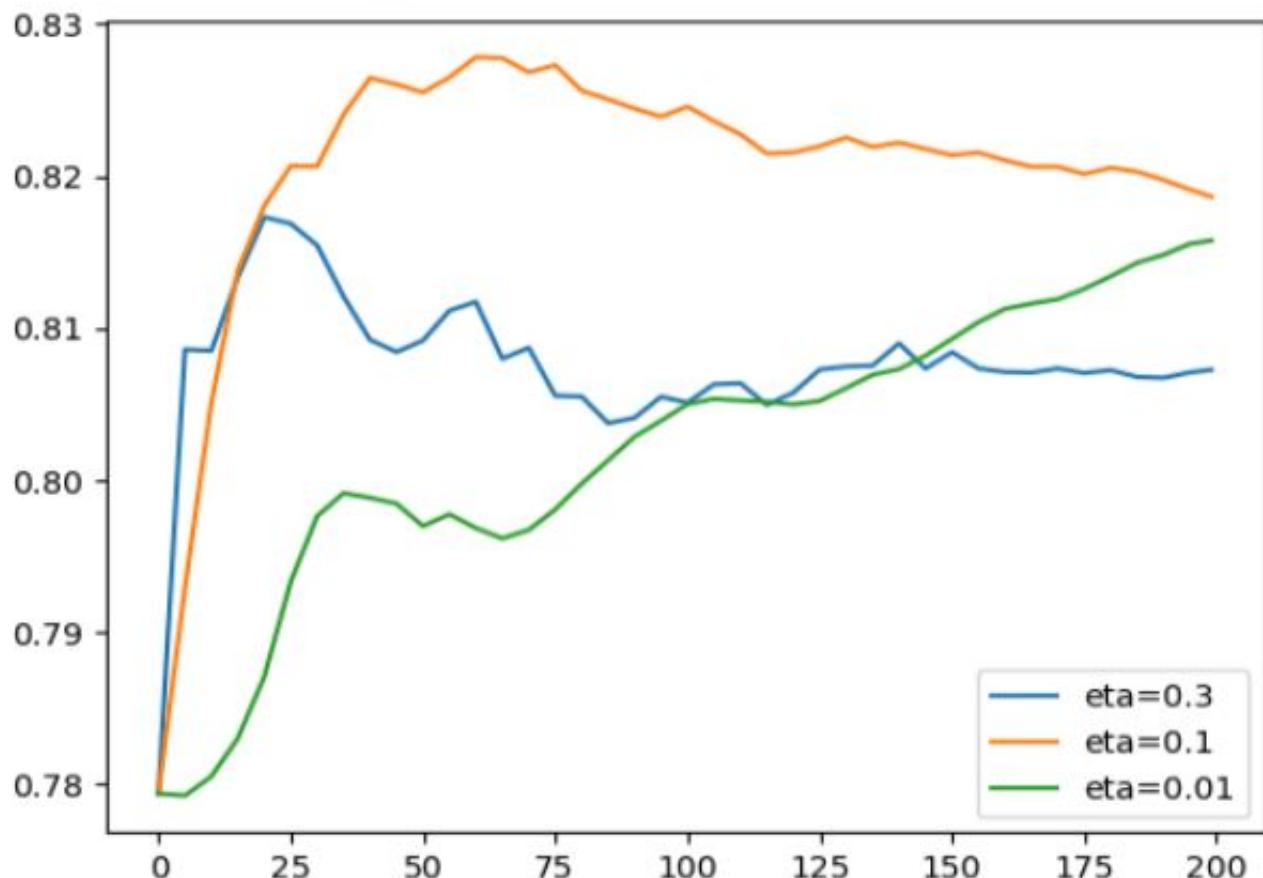
Next let's look at eta=0.3, eta=0.1, and eta=0.01

```

etas = ['eta=0.3', 'eta=0.1', 'eta=0.01']
for eta in etas:
    df_score = scores[eta]
    plt.plot(df_score.num_iter, df_score.val_auc, label=eta)
plt.legend()

```

```
<matplotlib.legend.Legend at 0x7f2343cd0090>
```



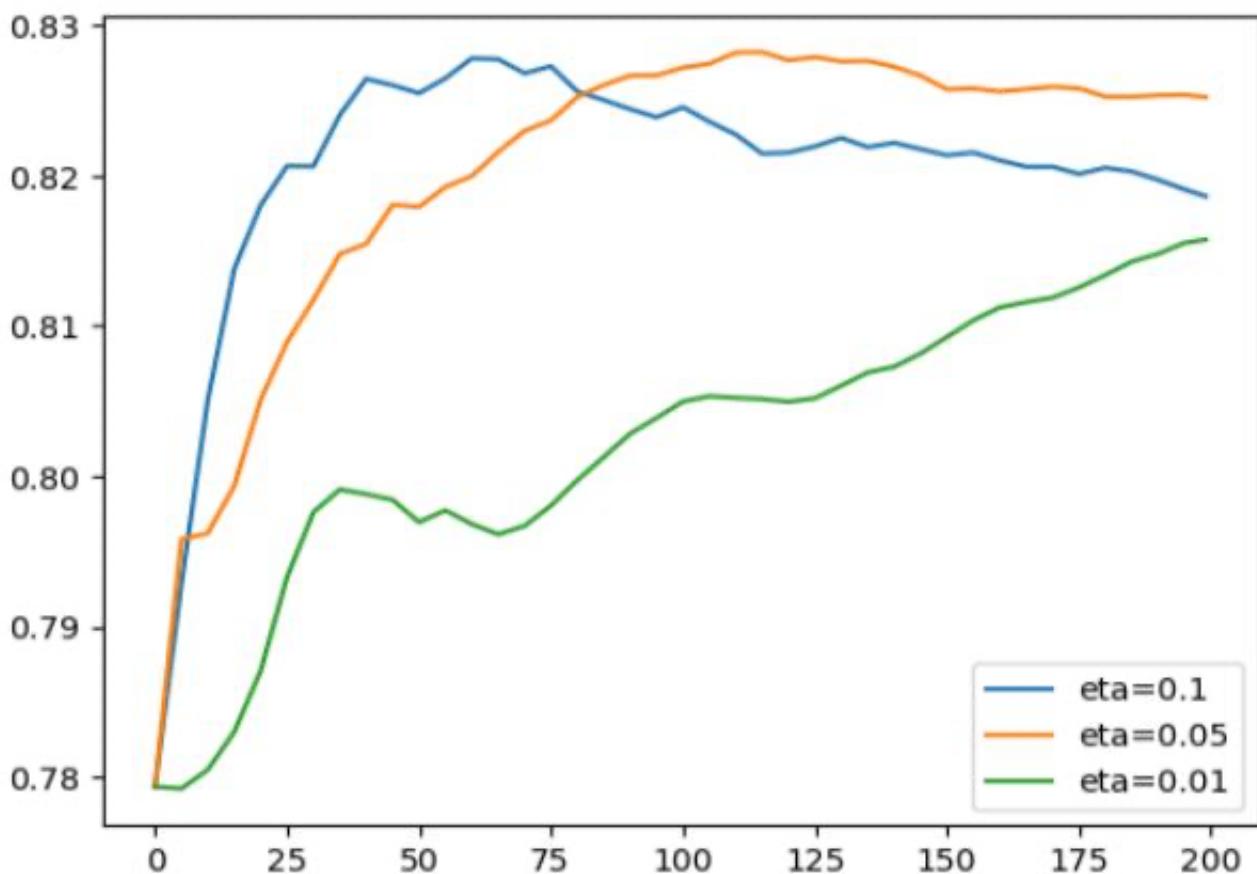
'eta=0.01' displays an extremely slow learning rate, making it challenging to estimate how long it might take to outperform the other model (represented by the orange curve). This model's progress is painstakingly slow, as the steps it takes are exceedingly tiny.

On the other hand, 'eta=0.3' takes a few significant steps initially but succumbs to overfitting more rapidly. In this plot, 'eta=0.1' seems to strike the ideal balance, particularly between 50 and 75 iterations. It may take a bit longer to reach its peak performance, but the resulting performance improvement justifies the wait.

There was also eta=0.05 let's finally look also at this plot.

```
etas = ['eta=0.1', 'eta=0.05', 'eta=0.01']
for eta in etas:
    df_score = scores[eta]
    plt.plot(df_score.num_iter, df_score.val_auc, label=eta)
plt.legend()
```

<matplotlib.legend.Legend at 0x7f2333d90150>



The ‘eta=0.05’ model requires approximately twice as many iterations to converge when compared to the blue model (‘eta=0.1’). Although it takes smaller steps and requires more time, the end result is still inferior to the blue model. Thus, it’s evident that the ‘eta=0.1’ model stands out as the best option, as it achieves better performance with fewer steps.

Now that we’ve found the best value for ‘eta,’ in the second part of XGBoost parameter tuning, we’ll focus on tuning two more parameters: ‘max_depth’ and ‘min_child_weight’ before proceeding to train the final model.

👤 Peter ⏰ 27. October 2023 📁 Decision Trees, ML-Zoomcamp
🔖 Decision Tree, Learning Rate, ML Zoomcamp, Parameter Tuning, XGBoost

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning– Part 13

👤 Peter ⏰ 28. October 2023 📄 Decision Trees, ML-Zoomcamp
🏷️ Decision Tree, ML Zoomcamp, Parameter Tuning, XGBoost



1. [XGBoost parameter tuning – Part 2/2](#)

1. [Tuning max_depth](#)
 1. [max_depth=6](#)
 2. [max_depth=3](#)
 3. [max_depth=4](#)
 4. [max_depth=10](#)
2. [Plotting max_depth](#)
3. [Tuning min_child_weight](#)
 1. [min_child_weight=1](#)
 2. [min_child_weight=10](#)
 3. [min_child_weight=30](#)
4. [Plotting min_child_weight](#)
5. [Train final model](#)

XGBoost parameter tuning – Part 2/2

This is the second part about **XGBoost** parameter tuning. In the first part, we tuned the first parameter – ‘eta’. Now we will explore parameter tuning for ‘max_depth’ and ‘min_child_weight’. Finally, we’ll train the final model.

Tuning max_depth

max_depth=6

Now that we’ve set ‘eta’ to 0.1, which we determined to be the best value, we’re going to

focus on tuning the ‘max_depth’ parameter. To do that, we’ll reset our scores dictionary to keep track of the new experiments. Initially, we’ll train a model with the same parameters as before, using it as a baseline for comparing different ‘max_depth’ values.

```
| scores = {}
```

```
%%capture output
xgb_params = {
    'eta': 0.1, ← Set in previous chapter
    'max_depth': 6,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    verbose_eval=5,
                    evals=watchlist)
```

```
key = 'max_depth=%s' % (xgb_params['max_depth'])
scores[key] = parse_xgb_output(output)

# Output: 'max_depth=6'
```

max_depth=3

Now, let’s set the ‘max_depth’ value to 3.

```
%%capture output
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    verbose_eval=5,
                    evals=watchlist)
```

```
key = 'max_depth=%s' % (xgb_params['max_depth'])
scores[key] = parse_xgb_output(output)
key
# Output: 'max_depth=3'
```

max_depth=4

Let's go through the process once more for 'max_depth=4'.

```
%%capture output
xgb_params = {
    'eta': 0.1,
    'max_depth': 4,
    'min_child_weight': 1,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}
model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    verbose_eval=5,
                    evals=watchlist)
```

```
key = 'max_depth=%s' % (xgb_params['max_depth'])
scores[key] = parse_xgb_output(output)
key
# Output: 'max_depth=4'
```

max_depth=10

Let's repeat the process for 'max_depth=10' once more.

XXcapture output

```
xgb_params = {  
    'eta': 0.1,  
    'max_depth': 10,  
    'min_child_weight': 1,  
  
    'objective': 'binary:logistic',  
    'eval_metric': 'auc',  
  
    'nthread': 8,  
    'seed': 1,  
    'verbosity': 1,  
}  
  
model = xgb.train(xgb_params, dtrain, num_boost_round=200,  
                   verbose_eval=5,  
                   evals=watchlist)
```

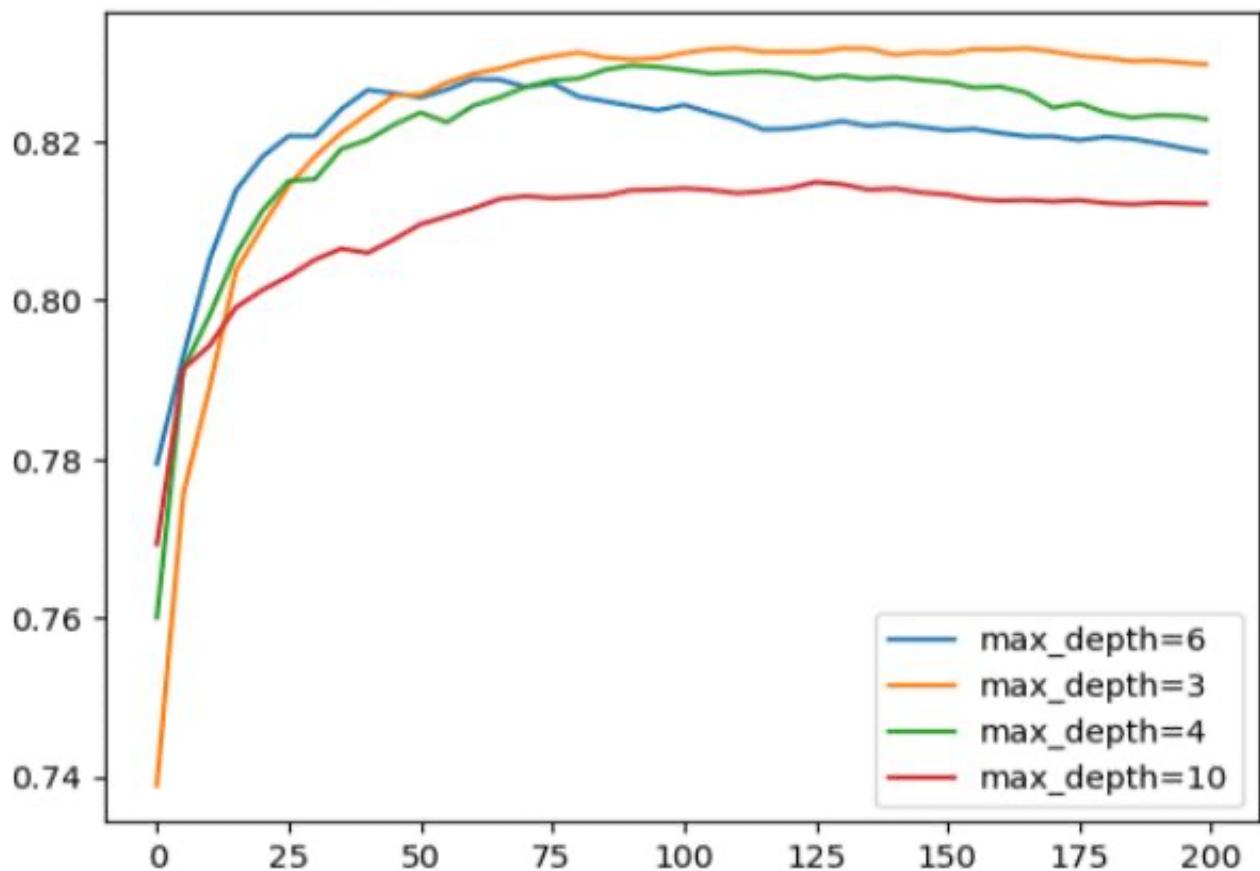
```
key = 'max_depth=%s' % (xgb_params['max_depth'])  
scores[key] = parse_xgb_output(output)  
key  
  
# Output: 'max_depth=10'
```

Plotting max_depth

Now that we've collected data from four runs, let's plot this information and determine which model performs the best.

```
for max_depth, df_score in scores.items():  
    plt.plot(df_score.num_iter, df_score.val_auc, label=max_depth)  
  
plt.ylim(0.8, 0.84)  
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f2333ff8150>
```



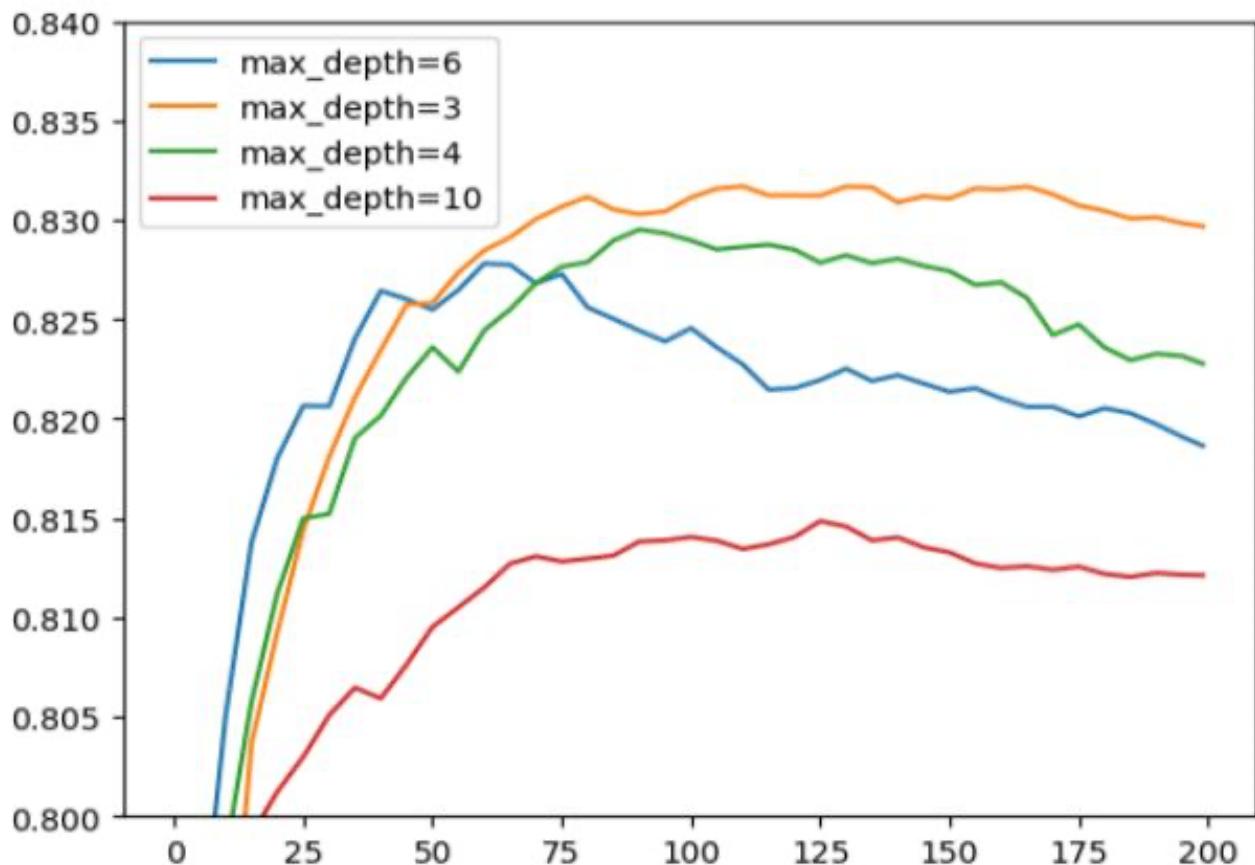
We see the depth of 10 is worst. So actually we can delete it by
del scores ['max_depth=10']

To get a better view on the y-area between 0.8 and 0.84 we can limit the plot as shown in the next snippet.

```
for max_depth, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=max_depth)

plt.ylim(0.8, 0.84)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f2333c38090>
```



'max_depth = 6' is the second worst. We conclude that 'max_depth' of 3 is the best depth for us.

Tuning min_child_weight

min_child_weight=1

Now, we'll set 'eta' to 0.1 and 'max_depth' to 3. We're ready to start tuning the last parameter, which is 'min_child_weight.' To do this, we need to reset our scores dictionary once again to track the new experiments. Initially, we'll train a model with the same parameters as before, but we'll set 'min_child_weight' to 1. This model will serve as our baseline for comparing different 'min_child_weight' values.

```
| scores = {}
```

%%capture output

```
xgb_params = {  
    'eta': 0.1,  
    'max_depth': 3, ← Set .. previous section  
    'min_child_weight': 1,  
  
    'objective': 'binary:logistic',  
    'eval_metric': 'auc',  
  
    'nthread': 8,  
    'seed': 1,  
    'verbosity': 1,  
}  
  
model = xgb.train(xgb_params, dtrain, num_boost_round=200,  
                   verbose_eval=5,  
                   evals=watchlist)
```

```
key = 'min_child_weight=%s' % (xgb_params['min_child_weight'])  
scores[key] = parse_xgb_output(output)  
key  
  
# Output: 'min_child_weight=1'
```

min_child_weight=10

Now, let's set the 'min_child_weight' value to 10.

%%capture output

```
xgb_params = {  
    'eta': 0.1,  
    'max_depth': 3,  
    'min_child_weight': 10,  
  
    'objective': 'binary:logistic',  
    'eval_metric': 'auc',  
  
    'nthread': 8,  
    'seed': 1,  
    'verbosity': 1,  
}  
  
model = xgb.train(xgb_params, dtrain, num_boost_round=200,  
                   verbose_eval=5,  
                   evals=watchlist)
```

```
key = 'min_child_weight=%s' % (xgb_params['min_child_weight'])  
scores[key] = parse_xgb_output(output)  
key  
  
# Output: 'min_child_weight=10'
```

min_child_weight=30

Let's go through the process once more for 'min_child_weight=30'.

```
%%capture output

xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 30,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                    verbose_eval=5,
                    evals=watchlist)
```

```
key = 'min_child_weight=%s' % (xgb_params['min_child_weight'])
scores[key] = parse_xgb_output(output)
key

# Output: 'min_child_weight=30'
```

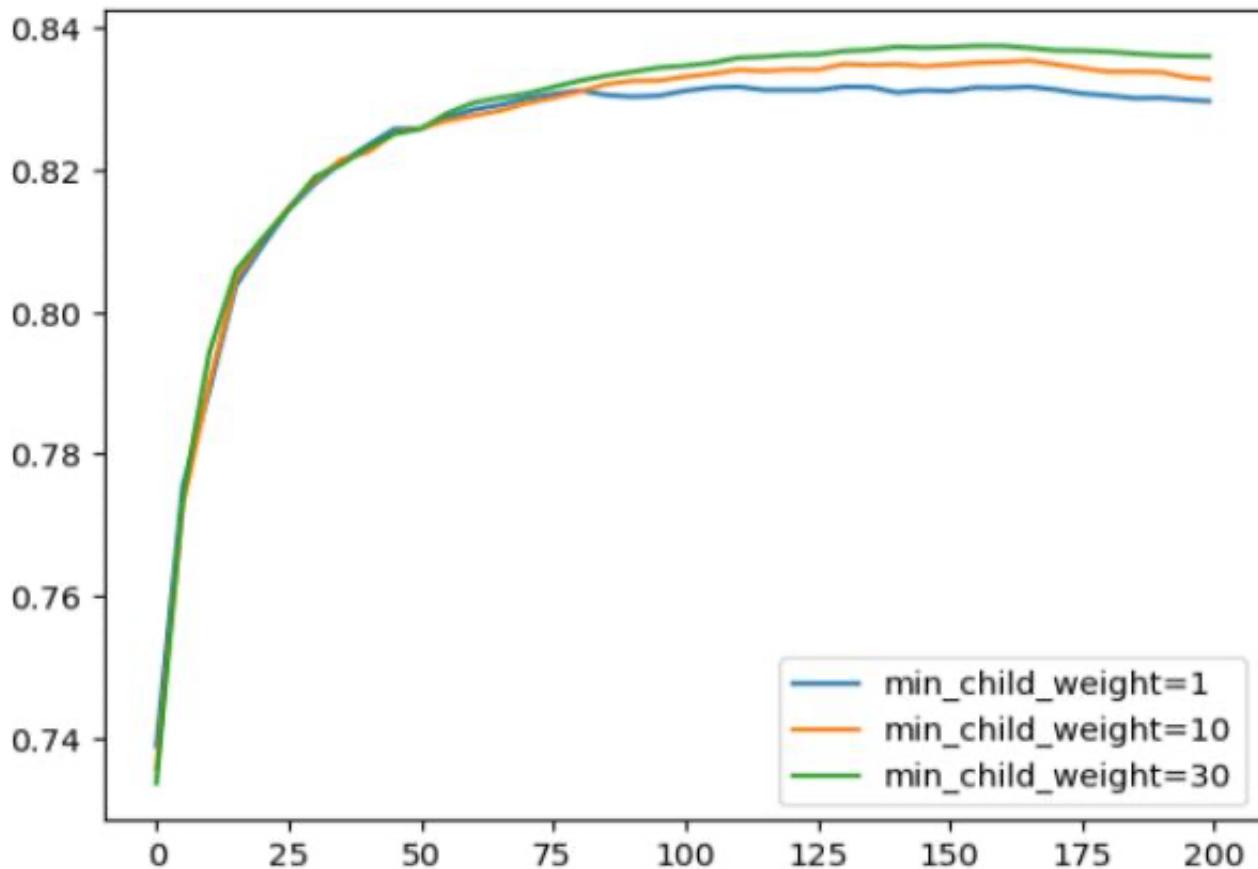
Plotting min_child_weight

This should give us an idea if we actually need to increase this value or not. Now we can compare all runs of 'min_child_weight=1', 'min_child_weight=10', and 'min_child_weight=30'.

```
for min_child_weight, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=min_cl)

plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f2333bdddd0>
```

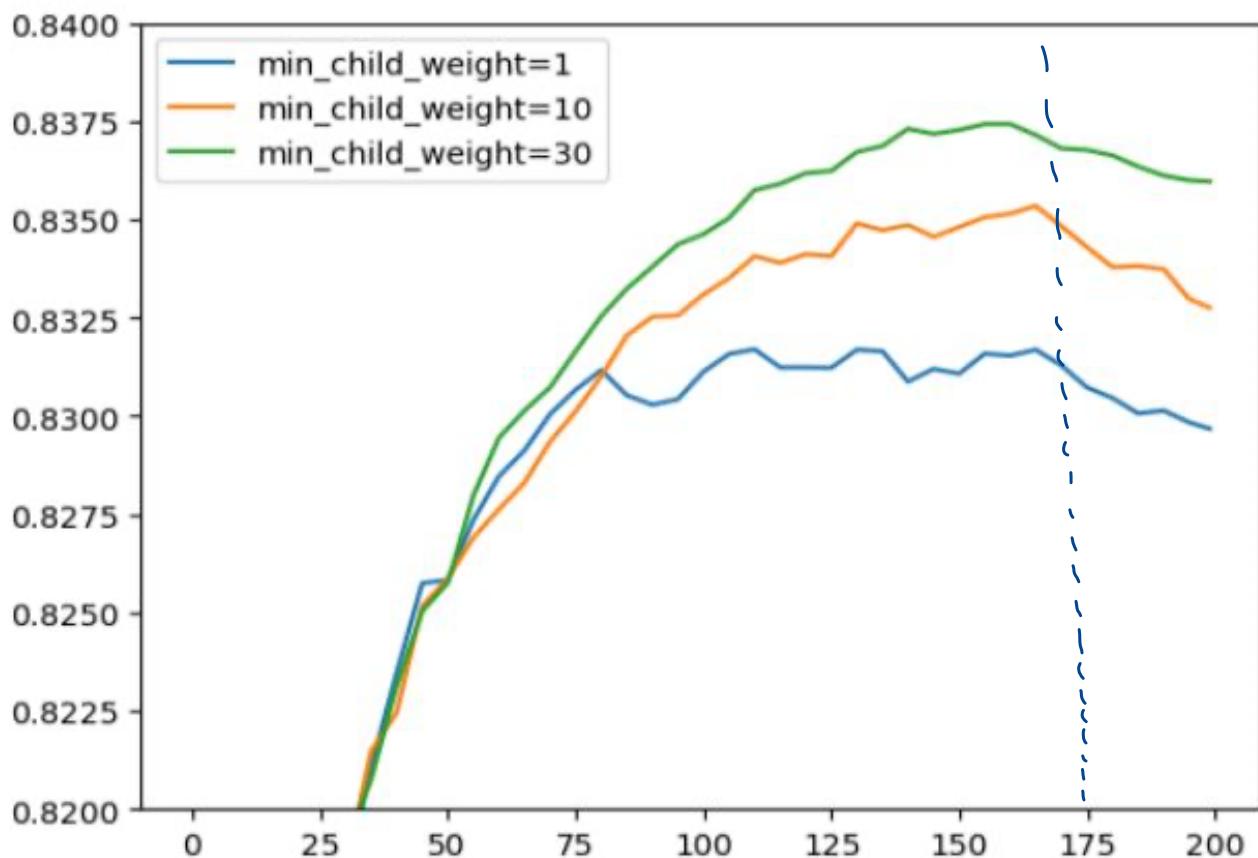


Here it's not so easy to see which one is the best. We should also enlarge it a bit.

```
for min_child_weight, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=min_cl)

plt.ylim(0.82, 0.84)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f233393a2d0>
```



This plot shows some differences compared to the video by Alexey. In this plot, a ‘min_child_weight’ of 30 appears to be the best-performing value, whereas in the video, the choice is ‘min_child_weight’ of 1. To maintain consistency with the video’s results, I’ve opted to select a ‘min_child_weight’ of 1. It’s worth noting that parameter tuning can be influenced by various factors, and flexibility in choosing the optimal values is important.

Train final model

To train the final model, we need to determine the number of iterations for training. In the video, Alexey chose to train for 175 iterations, *based on the last plot*

```
xgb_params = {  
    'eta': 0.1,  
    'max_depth': 3,  
    'min_child_weight': 1,  
  
    'objective': 'binary:logistic',  
    'eval_metric': 'auc',  
  
    'nthread': 8,  
    'seed': 1,  
    'verbosity': 1,  
}  
  
model = xgb.train(xgb_params, dtrain, num_boost_round=175)
```

```

xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 30,
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=175)

```

Always creating all these plots may not be necessary. You can also examine the raw output and use tools like pen and paper or an Excel spreadsheet when experimenting with parameter tuning. Finding the best approach depends on your preferences and needs. Eta, max_depth, and min_child_weight are indeed important parameters, but there are other valuable ones to consider.

For reference, you can explore the complete list of XGBoost parameters here: [XGBoost Parameters](#).

Two parameters that can be particularly useful are ‘subsample’ and ‘colsample_bytree.’ They have some similarities:

- ‘colsample_bytree’:
Similar to what we observe in Random Forest, this parameter controls how many features each tree gets to see at each iteration. The maximum value is 1.0. You can experiment with values like 0.3 and 0.6, and then fine-tune around those values.
- ‘subsample’:
Instead of sampling columns, this parameter allows you to sample rows. It means you can choose to provide only a percentage of the training data. For example, setting it to 0.5 means you randomly select 50% of the training data.

In addition to these, there’s a wealth of information available on rules of thumb for tuning XGBoost parameters. Kaggle is a valuable resource for learning more, and you can find many tutorials on XGBoost parameter tuning.

 [Peter](#)  [28. October 2023](#)  [Decision Trees](#), [ML-Zoomcamp](#)
 [Decision Tree](#), [ML Zoomcamp](#), [Parameter Tuning](#), [XGBoost](#)

Leave a comment

Write a comment...

ML Zoomcamp 2023 – Decision Trees and Ensemble Learning – Part 14

👤 Peter ⏰ 29. October 2023 📄 Decision Trees, ML-Zoomcamp
⭐ Decision Tree, ML Zoomcamp, Random Forest, XGBoost



1. [Selecting the final model](#)

1. [Choosing between XGBoost, random forest and decision tree](#)
 1. [Retrain the best model of each type](#)
 2. [Evaluate all the best of models on validation data](#)
2. [Training the final model](#)
3. [Evaluate the final model!!](#)

Selecting the final model

This is the final part of the module ‘Decision Trees and Ensemble Learning – Part 14.’ This time, we revisit the best model of each type and evaluate their performance on the validation data. Based on these evaluations, we will select the overall best model and train it on the full training dataset. The final model will then be evaluated on the test set.

Choosing between XGBoost, random forest and decision tree

Retrain the best model of each type

Let's retrain the best **Decision Tree** model we had.

```
dt = DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
dt.fit(X_train, y_train)

# Output:
# DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
```

Let's retrain the best **Random Forest** model we had.

```
rf = RandomForestClassifier(n_estimators=200,
                           max_depth=10,
                           min_samples_leaf=3,
                           random_state=1)
rf.fit(X_train, y_train)

# Output:
# RandomForestClassifier(max_depth=10, min_samples_leaf=3, n_estimators=200,
# random_state=1)
```

Let's retrain the best **XGBoost** model we had.

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=175)
```

Evaluate all the best of models on validation data

```
# Decision Tree
y_pred = dt.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)

# Output: 0.7850954203095104
```

```
# Random Forest
y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)

# Output: 0.8246258264512848
```

```
# XGBoost Model  
y_pred = model.predict(dval)  
roc_auc_score(y_val, y_pred)  
# Output: 0.8309347073212081
```

We see that the **XGBoost** model has the best auc score. We'll use this to train the final model.

Training the final model

To train the final model, we will use the entire dataset. Following the training, we will evaluate the final model on our test dataset.

```
| df_full_train
```

	status	seniority	home	time	age	marital	records	job	exp
3668	ok	22	owner	48	48	married	no	fixed	
2540	default	8	other	60	41	married	no	freelance	
279	ok	2	parents	36	19	married	no	fixed	
3536	ok	1	owner	12	61	married	no	others	
3866	ok	13	owner	60	27	married	no	fixed	
...
332	default	4	owner	60	47	married	no	freelance	
1293	ok	2	rent	60	28	single	no	fixed	
4023	ok	2	parents	36	25	single	no	fixed	
3775	ok	4	other	60	25	single	no	fixed	
1945	default	1	parents	48	25	single	no	freelance	

3563 rows × 14 columns

Upon reviewing the previous output, we can observe that the index is not ordered. To address this, we will start by resetting the index.

```
| df_full_train = df_full_train.reset_index(drop=True)
```

The next steps involve setting the 'y' value and removing the 'status' column from the

training dataframe to prevent accidental use of this column during training.

```
y_full_train = (df_full_train.status == 'default').astype(int).values  
y_full_train  
# Output: array([0, 1, 0, ..., 0, 0, 1])
```

```
del df_full_train['status']
```

We can create dictionaries for the DictVectorizer and then use the `fit_transform` method to obtain `X_full_train`. For `X_test`, we only need to call the `transform` method since the vectorizer has already been fitted.

```
dicts_full_train = df_full_train.to_dict(orient='records')  
  
dv = DictVectorizer(sparse=False)  
X_full_train = dv.fit_transform(dicts_full_train)  
  
dicts_test = df_test.to_dict(orient='records')  
X_test = dv.transform(dicts_test)
```

Inspecting the ‘feature_names’ reveals the one-hot-encoded columns and confirms that there is no ‘status’ column, indicating that our data preparation is complete and we are ready to train the final model.

```
feature_names = list(dv.get_feature_names_out())
feature_names

# Output:
# ['age',
#  'amount',
#  'assets',
#  'debt',
#  'expenses',
#  'home=ignore',
#  'home=other',
#  'home=owner',
#  'home=parents',
#  'home=private',
#  'home=rent',
#  'home=unk',
#  'income',
#  'job=fixed',
#  'job=freelance',
#  'job=others',
#  'job=partime',
#  'job=unk',
#  'marital=divorced',
#  'marital=married',
#  'marital=separated',
#  'marital=single',
#  'marital=unk',
#  'marital=widow',
#  'price',
#  'records=no',
#  'records=yes',
#  'seniority',
#  'time']
```

XGBoost models require data in the form of **DMatrix** for training. We also prepare the test data, which doesn't require labels, as we'll evaluate it using Scikit-Learn.

```
feature_names = list(dv.get_feature_names_out())
dfulltrain = xgb.DMatrix(X_full_train, label=y_full_train,
                         feature_names=feature_names)

dtest = xgb.DMatrix(X_test, feature_names=feature_names)
```

Now let's set the parameters and train the final model.

```
xgb_params = {  
    'eta': 0.1,  
    'max_depth': 3,  
    'min_child_weight': 1,  
  
    'objective': 'binary:logistic',  
    'eval_metric': 'auc',  
  
    'nthread': 8,  
    'seed': 1,  
    'verbosity': 1,  
}  
  
model = xgb.train(xgb_params, dfulltrain, num_boost_round=175)
```

Evaluate the final model

```
y_pred = model.predict(dtest)  
roc_auc_score(y_test, y_pred)  
  
# Output: 0.8289367577342261
```

The performance of the final model is a little bit worse than the best **XGBoost** model (0.831), but it's only like a fraction of one percent. So this is fine. We can conclude that our model didn't overfit. The final model generalizes quite well on unseen data. **XGBoost** models are often one of the best models at least for tabular data (dataframe with features). The downside of this is that **XGBoost** models are more complex, it's more difficult to tune, it has more parameters, and it's easier to overfit with **XGBoost**. But you can get a better performance out of this.

• Peter • 29. October 2023 ■ Decision Trees, ML-Zoomcamp
🏷️ Decision Tree, ML Zoomcamp, Random Forest, XGBoost

Leave a comment

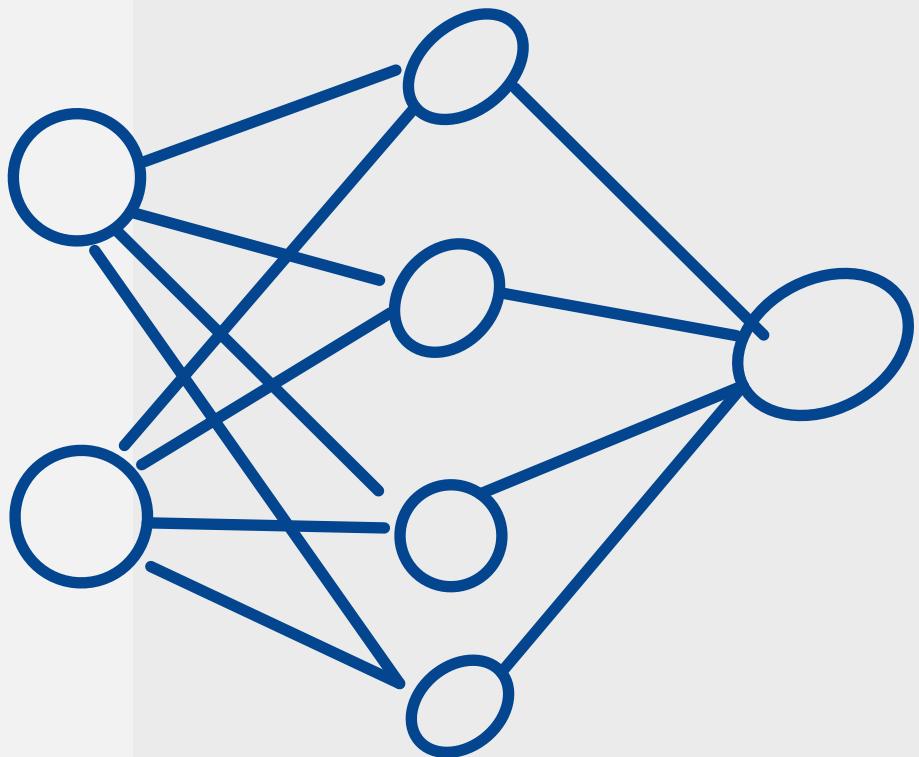
Write a comment...

Comment

NEURAL NETWORKS

+

DEEP
LEARNING



SUMMARY

ML Zoomcamp 2023 – Deep Learning – Part 1

👤 Peter ⏰ 18. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, ML Zoomcamp



What is Deep Learning and why it is important for this chapter?

Deep learning is a subfield of machine learning that involves neural networks with multiple layers (deep neural networks). It's particularly powerful in handling complex tasks such as image processing, speech recognition, natural language processing, and more. Handling images is exactly what we will do here.

We'll look at images – instead of tabular data like in the past chapters. This time we have images with clothes and want to predict a label what is on this image. The project what we'll do is a classification project (here: multi-class classification). We want to predict if an image belongs to one of ten different clothing categories.

Use Case: We have a website and the user wants to create a listing in the fashion category, so he wants to sell a t-shirt for example. That means he uses this website and uploads a picture and there is a fashion classification service. This service will get this picture and it will reply with a suggested category (here: t-shirt).

This classification service will contain a neural network which will look at the image and predict a category for this image. The dataset is a clothing dataset (<https://github.com/alexeygrigorev/clothing-dataset>) with 5.000 images of 20 different classes. But we'll use a subset of this dataset which contains 10 most popular classes (<https://github.com/alexeygrigorev/clothing-dataset-small>).

This time there already train, validate, and test folders with images of this 10 classes we're interested in, so we don't have to do train-test split by our own. We just need to clone the repo.

Theory of neural networks on CS 231n (Stanford course) online

| !git clone git@github.com:alexeygrigorev/clothing-dataset-sma

You can find the theory behind that neural networks here: <https://cs231n.github.io>

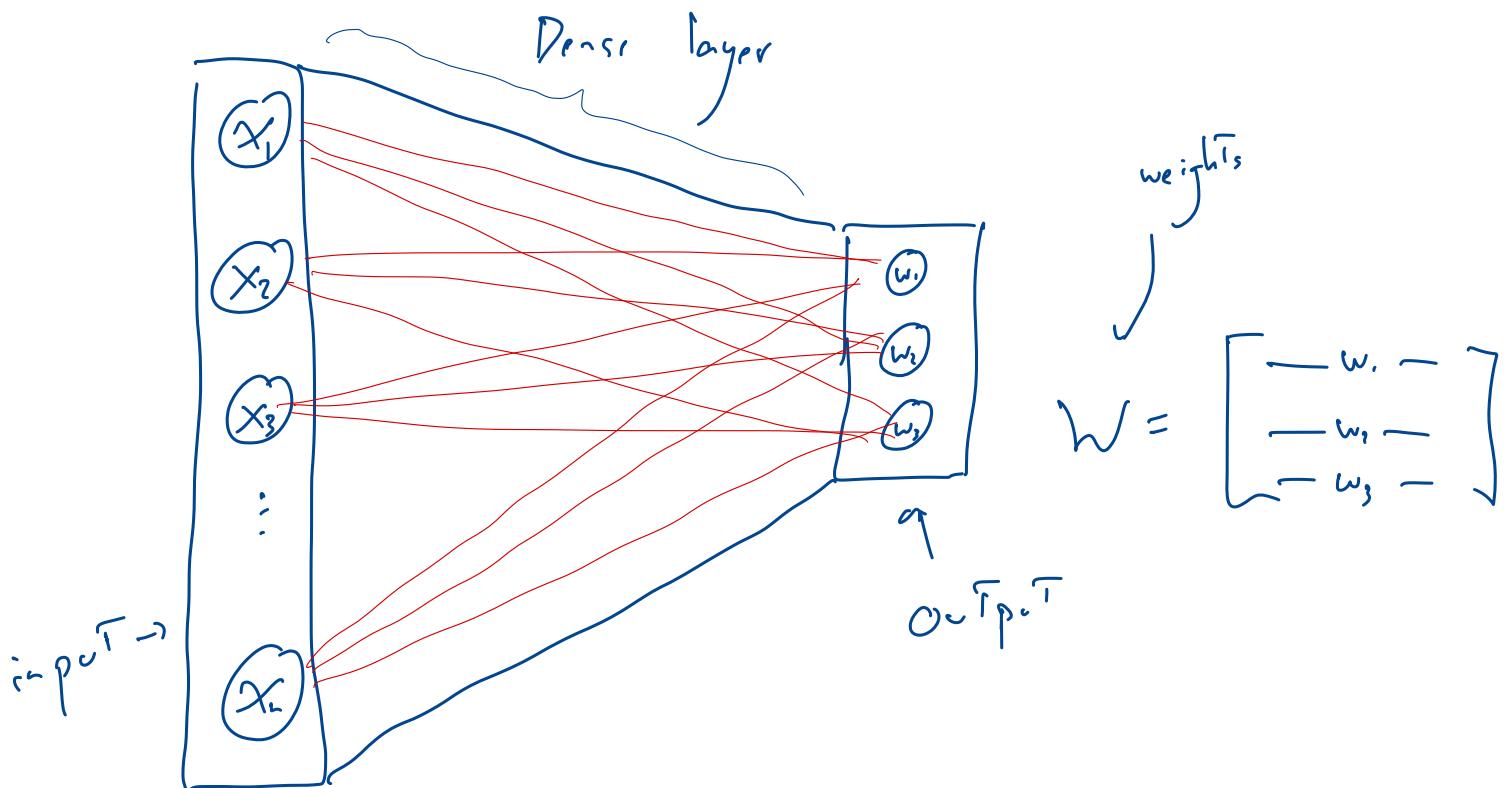
• Peter • 18. November 2023 ■ Deep Learning, ML-Zoomcamp
Deep Learning, ML Zoomcamp

Leave a comment

Write a comment...

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

[knowMLedge.com](#), W



ML Zoomcamp 2023 – Deep Learning – Part 2

👤 Peter ⏰ 18. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ CUDA, cuDNN, Deep Learning, GPU, ML Zoomcamp, TensorFlow, Windows



1. [How to find the right Cuda version?](#)
2. [Step 1: Installing CUDA Toolkit](#)
3. [Step 2: Installing cuDNN](#)
4. [Step 3: Installing TensorFlow and Requirements](#)
5. [Step 4: Testing](#)

This part is not discussed in the course ML Zoomcamp, but it could be useful how to setup GPU support for your local machine.

How to find the right Cuda version?

First you need to know which **CUDA Toolkit** works with your NVIDIA graphics adapter. The easiest way to find out the supported version is to search your adapter in the list on wikipedia (<https://de.wikipedia.org/wiki/CUDA>). In my case I need Cuda version 10.0-11.3. You can find the version 11.3 for Linux and Windows on NVIDIAs developer page (<https://developer.nvidia.com/cuda-11.3.0-download-archive>). In this article I want to describe the Windows part.

CUDA Toolkit 11.3 Downloads

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown. By downloading and using the software, you agree to fully comply with the terms and conditions of the [CUDA EULA](#).

Operating System	Linux	Windows	
Architecture	x86_64		
Version	10	Server 2016	Server 2019
Installer Type	exe (local)	exe (network)	

Download Installer for Windows 10 x86_64

The base installer is available for download below.

> Base Installer

Download (2.7 GB)

Installation Instructions:

1. Double click cuda_11.3.0_465.89_win10.exe
2. Follow on-screen prompts

The checksums for the installer and patches can be found in [Installer Checksums](#).
For further information, see the [Installation Guide for Microsoft Windows](#) and the [CUDA Quick Start Guide](#).

Step 1: Installing CUDA Toolkit

When installing be careful only to update/install the CUDA software. It can happen that you've installed already a newer version of drivers and the other software packages, but the CUDA version has to match with your graphics adapter. The installation can take some minutes. After the installation has finished you can find it here "C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA".

Step 2: Installing cuDNN

To really use the full potential of your NVIDIA card you need another software library from NVIDIA – the NVIDIA CUDA Deep Neural Network Library (cuDNN). This library provides optimized implementations for operations (specifically for GPUs) that are normally used in neural networks (pooling, convolution, and matrix operations).

But before you can download this package from <https://developer.nvidia.com/cudnn> you need a free account at NVIDIA's Developer Program. When having already an account at Facebook or Google, you can use that account (didn't test it).

When download is finished you can find a zip package in your download folder. Now you need to copy some files to your file system.

- **cudnn64_8.dll**
 - from zip folder bin\
 - to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.3\bin
- **cudnn.h**
 - from zip folder include
 - to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.3\bin
- **cudnn.lib**
 - from zip folder lib\x64\
 - to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.3\lib\x64

The last step is to check the system variables for CUDA. If there are more CUDA versions installed you have different variables of kind CUDA_PATH_Vxx_y. Check if all point to the same directory (**CUDA_HOME**, **CUDA_PATH**, **CUDA_PATH_V11_3**).

Systemvariablen	
Variable	Wert
CUDA_HOME	C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.3
CUDA_PATH	C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.3
CUDA_PATH_V10_1	C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.1
CUDA_PATH_V11_3	C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.3

Step 3: Installing TensorFlow and Requirements

Now your Windows machine ready to use TensorFlow with GPU-Support. The installation of TensorFlow is now quite easy and works similar to CPU version. Before we can install tensorflow we need to ensure having installed *Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017, and 2019* (<https://learn.microsoft.com/de-DE/cpp/windows/latest-supported-vc-redist?view=msvc-170>).

Now we can try `pip install` or `pip3 install`. But there is an important information from <https://www.tensorflow.org/install/pip#windows-native>, use tensorflow<2.11 because anything above 2.10 is not supported on the GPU on Windows Native.

```
| pip install "tensorflow<2.11"
```

Step 4: Testing

Now just open a new Jupyter notebook an check that everything works fine.

```
In [1]: !python --version
```

Python 3.9.18

```
In [2]: import tensorflow as tf
```

Verify the CPU setup:

```
In [3]: print(tf.reduce_sum(tf.random.normal([1000, 1000])))
```

tf.Tensor(949.1373, shape=(), dtype=float32)

Verify the GPU setup:

```
In [4]: print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

Num GPUs Available: 1

👤 Peter ⏰ 18. November 2023 📄 Deep Learning, ML-Zoomcamp
🏷️ CUDA, cuDNN, Deep Learning, GPU, ML Zoomcamp, TensorFlow, Windows

Leave a comment

Write a comment...

[Comment](#)

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

knowMLedge.com, 

ML Zoomcamp 2023 – Deep Learning – Part 3

👤 Peter 🕒 19. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, Keras, ML Zoomcamp, PIL, Pillow, TensorFlow



1. [TensorFlow and Keras](#)
2. [Installing TensorFlow](#)
3. [Loading images](#)
4. [PIL and Pillow](#)

TensorFlow and Keras

TensorFlow and **Keras** are powerful tools in the field of deep learning. **TensorFlow**, an open-source machine learning library developed by Google, provides a comprehensive platform for building and deploying machine learning models. It facilitates the creation of complex neural network architectures and supports both training and inference across various devices.

Keras, on the other hand, is a high-level neural networks API that runs on top of TensorFlow. It simplifies the process of building and experimenting with neural networks, making it user-friendly and accessible. **Keras** provides a modular and extensible interface, allowing developers to quickly prototype and experiment with different architectures.

Together, **TensorFlow** and **Keras** form a robust ecosystem for deep learning, enabling practitioners to seamlessly transition from model development to deployment while benefiting from a rich set of features, extensive documentation, and a vibrant community.

Installing TensorFlow

To install TensorFlow, you can use either `conda` or `pip`. Here are the commands for both:

Using `conda`:

```
| conda install tensorflow -y
```

Using **pip**:

```
| pip install tensorflow
```

There is no need to install Keras separately because it comes preinstalled with TensorFlow.

Loading images

As usual, we first need some imports. We know already numpy and matplotlib but now we also import tensorflow and keras. From Keras we also import a special function for loading images called `load_img`.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import load_img
```

Let's see this function in action. We take one special image from the training dataset.

```
path = './clothing-dataset-small/train/t-shirt'
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'
fullname = f'{path}/{name}'
load_img(fullname)
```



When loading an image, you can specify the size. The reason we need to do this is that a neural network expects an image of a certain size, usually 299×299 , 224×224 , or 150×150 . If the original size of our image is larger, we need to resize it to one of the mentioned formats. Resizing is quite easy; you only need to use the `target_size` parameter. Here's an example:

```
| img = load_img(fullname, target_size=(299, 299))
```

Make sure to adjust the `target_size` parameter based on the input size expected by your neural network model.

PIL and Pillow

The library for processing images is called **PIL** (Python Imaging Library), and it's used by many image processing libraries. **PIL** is developed by Fredrik Lundh and contributors. There is also **Pillow**, which is a fork of **PIL** by [Jeffrey A. Clark \(Alex\) and contributors](#). You can find the documentation [here](#).

```
| print(img)
| # Output: <PIL.Image.Image image mode=RGB size=299x299 at 0x7f
```

RGB

The way an image is represented internally is with three channels, and for each channel, we have an array. These arrays contain numbers between 0 and 255. Each pixel is a combination of the three values for the red, green, and blue channels. The shape of that image should be $(299, 299, 3)$ with (height, width, #channels). We can easily translate this Pillow image into a numpy array.

```

x = np.array(img)
x
# Output: 
# array([[[179, 171, 99],
#          [179, 171, 99],
#          [181, 173, 101],
#          ...
#          [[188, 179, 112],
#           [187, 178, 111],
#           [186, 177, 108],
#           ...
#           [[251, 252, 247],
#            [251, 252, 247],
#            [251, 252, 246]],
#           ...
#           [[199, 189, 127],
#            [200, 190, 128],
#            [200, 191, 126],
#            ...
#            [[250, 251, 245],
#             [250, 251, 245],
#             [250, 251, 245]],
#             ...
#             ...,
#             ...
#             [171, 157, 82],
#             ...
#             [[181, 133, 22],
#              [179, 131, 20],
#              [182, 134, 23]]], dtype=uint8)

```

The dtype of this array is uint8, where “u” means unsigned, so it doesn’t have any sign, ranging from 0 to 255. The “8” signifies that it’s an integer taking 8 bits (=1 byte).

```

x.shape
# Output: (299, 299, 3)

```

[Peter](#)
 [19. November 2023](#)
 [Deep Learning, ML-Zoomcamp](#)
[Deep Learning, Keras, ML Zoomcamp, PIL, Pillow, TensorFlow](#)

Leave a comment

Write a comment...

ML Zoomcamp 2023 – Deep Learning – Part 4

👤 Peter ⏰ 19. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, Image Classification, ImageNet, Keras, ML Zoomcamp, Neural Network, TensorFlow



In this part we'll use a pre-trained convolutional neural network to understand what is on the image we load previously.

Pre-trained convolutional neural networks

This time we want to take an image and an off-the-shelf neural network that was already trained by somebody, so we can use it. Now we want to use a special model called "Xception" from Keras which was trained on [ImageNet](#). You can find more pre-trained models on [Keras](#). Before defining the model we need some imports first.

```
from tensorflow.keras.applications.xception import Xception
from tensorflow.keras.applications.xception import preprocess_input
from tensorflow.keras.applications.xception import decode_predictions

# weights = "imagenet" means we want to use pre-trained network

model = Xception(
    weights="imagenet",
    input_shape=(299, 299, 3)
)
```

Now we want to use this model to classify the image, we used before. But this time the `model.predict` function expects a bunch of images. So let's create an array with possibly multiple images. In this case it is just one.

```
X = np.array([x])
X.shape

# Output: (1, 299, 299, 3)
```

To do the prediction, we need some preprocessing before. This model expects inputs in a certain way using the `preprocess_input` function.

```
X = preprocess_input(X)
X[0]

# Output:
# array([[[ 0.4039216 ,  0.3411765 , -0.2235294 ],
#          [ 0.4039216 ,  0.3411765 , -0.2235294 ],
#          [ 0.41960788,  0.35686278, -0.20784312],
#          ...
#          [ 0.96862745,  0.9843137 ,  0.94509804],
#          [ 0.96862745,  0.9843137 ,  0.94509804],
#          [ 0.96862745,  0.99215686,  0.9372549 ]],
#         [
#          [[ 0.47450984,  0.4039216 , -0.12156862],
#          [ 0.46666667 ,  0.39607847, -0.12941176],
#          [ 0.45882356,  0.38823533, -0.15294117],
#          ...
#          [ 0.96862745,  0.9764706 ,  0.9372549 ],
#          [ 0.96862745,  0.9764706 ,  0.9372549 ],
#          [ 0.96862745,  0.9764706 ,  0.92941177]],
#         [
#          [[ 0.56078434,  0.48235297, -0.00392157],
#          [ 0.5686275 ,  0.4901961 ,  0.00392163],
#          [ 0.5686275 ,  0.49803925, -0.01176471],
#          ...
#          [ 0.9607843 ,  0.96862745,  0.92156863],
#          [ 0.9607843 ,  0.96862745,  0.92156863],
#          [ 0.9607843 ,  0.96862745,  0.92156863]],
#         ...
#         ...
#          [ 0.3411765 ,  0.2313726 , -0.35686272],
#          ...
#          [ 0.41960788,  0.04313731, -0.827451 ],
#          [ 0.4039216 ,  0.02745104, -0.84313726],
#          [ 0.427451 ,  0.05098045, -0.81960785]]], dtype=
```

```
pred = model.predict(X)

# 1/1 [=====] - 2s 2s/step
```

```
pred.shape

# (1, 1000)
```

This 1000 means that there are 1000 different classes and 1 means there is one image.

```

pred

# Output:
# array([[3.23712389e-04, 1.57383955e-04, 2.13493346e-04, 1.1
#           2.47626507e-04, 3.05036228e-04, 3.20592342e-04,
#           ...
#           2.07101941e-04, 2.05870383e-04, 4.28847765e-04,
#           1.12896028e-04, 1.57900504e-04, 1.94431108e-04,
#           3.20827705e-04, 2.70084536e-04, 3.43746680e-04,
#           2.78319319e-04, 3.25885747e-04, 1.71753796e-04,
#           dtype=float32)

```

Each value is the probability that this image belongs to some class. To be able to make sense from this output, we need to know what are the classes. Therefor we need another function called `decode_predictions` to make the prediction human readable.

```

decode_predictions(pred)

# Output:
# [(['n03595614', 'jersey', 0.6819631),
#  ('n02916936', 'bulletproof_vest', 0.038140077),
#  ('n04370456', 'sweatshirt', 0.034324776),
#  ('n03710637', 'maillot', 0.011354236),
#  ('n04525038', 'velvet', 0.0018453619)]]

```

In real this image is a t-shirt, but ImageNet is not very good when it comes to clothes detection. That means it doesn't really work for our purpose here. That means we need to train a different model with the classes we need for our case. Good point here, we don't have to retrain the model from scratch. We can reuse this model. That means we can build on top of what big companies or universities have provided and adapt to our specific use case.

 [Peter](#)  [19. November 2023](#)  [Deep Learning](#), [ML-Zoomcamp](#)
 [Deep Learning](#), [Image Classification](#), [ImageNet](#), [Keras](#), [ML Zoomcamp](#), [Neural Network](#), [TensorFlow](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Deep Learning – Part 5

👤 Peter ⏰ 20. November 2023 📁 Deep Learning, ML-Zoomcamp

🔖 [Binary Classification](#), [Convolution Layer](#), [Convolutional Neural Network](#), [Dense Layer](#),
[Feature Map](#), [Filters](#), [ML Zoomcamp](#), [Multiple Classification](#)



1. [Convolutional neural networks](#)
2. [Convolutional layers and filters](#)
 1. [What filters do?](#)
 2. [What happens when we take an image and pass it through a set of convolutional layers?](#)
3. [Dense layers](#)
 1. [Binary classification problem](#)
 2. [Multiple classification problem](#)
4. [Summary](#)

This part is about Convolutional Neural Networks (CNN). In the last part we already touched this topic while we were using a pre-trained neural network. This part is a bit more about the theory.

Convolutional neural networks

Convolutional neural networks are mostly used for images and they consists of different kind of layers. Let's imagine a CNN as a black box that gets an image and outputs a prediction. What happens inside? There are different kinds of layers. We'll look at two main types of layers which are convolutional layers and dense layers. There are much more layers. You can find good information [here](#).

Convolutional layers and filters

Convolutional layers consist of filters which are kind of small images. They're usually quite small images like 5×5 , could be even smaller. These image filters contain simple shapes, like

[^] [-] [/] [u] [|] [\] This is not from a real network it's just to give an understanding.

Then we take the image and take one filter and slide this filter across the image. Every time we apply the filter to an image we see how similar this filter is to the part of the image. A table is created with a lot of cells in which each cells value corresponds to the similarity of this filter to the corresponding part of the image which is a number between 0 (no similarity) and 9 (very similar). This table is called a **feature maps**. So a **feature maps** is the result of applying a filter to an image.

So we slide a filter across the image and every time we calculate the similarity between this particular filter and the part of the image, we record the result and get this **feature map**. High values mean higher degree of similarity. We do this steps for each filter. We have six filters here that means we'll get six feature maps.

So the output of the first convolutional layer is a set of **feature maps**. We can take this output and treat it as a new image that we made from the original one. Then we can have another convolutional layer that has its own set of filters it applies. Then this layer produces its own feature map. Let's say in the first convolution we used 6 filters and now in the second convolution we have 12 filters. So the second convolution produces 12 **feature maps**. We can easily imagine to go on with other convolutional layers with its own filters. Because of this chaining each layer learns more and more complex filters.

image	->	CONV LAYER 1	->	CONV LAYER 2	->	CONV LAYER 3	->	...
		[^] [-] [/]		[<] [o] [>]		[() [U] [)]		
		[u] [] [\]		[L] [x] [+]		[?] [Q] [~]		
		Low-Level		Mid-Level		High-Level		

The second layer learns the new filters by combining the filters from the previous layer. That means with every layer you can learn more complex shapes (= more high-level features of the image). This representation is not exactly how filters look like, but we can think of them in this way.

What filters do?

We have a filter that we apply to some area, then it looks at that region across all the values of all the feature maps in this particular region. That means it "goes in depth" here and looks at all the different feature maps. So let's assume that in one place there is a 6 in case of [/] and a 6 in case of [\] then there is a high possibility that there is an "X" on the image in this area.

What happens when we take an image and pass it through a set of convolutional layers?

The result is a vector representation of the image. Let's say the image is 299x299x3, then the vector representation could be something like 2,048, which is a one-dimensional array.

This way we turn an image into a vector. This vector captures all this information about the image. So all the filters of the convolutional layers extract features from the image that contain all the information that the neural network was able to extract from the image.

Dense layers

What we can do with this vector representation is, we can build some dense layers. These layers turn the vector representation into final prediction. The final prediction could be a label like “t-shirt”. While the role of convolutional layers is extracting a vector representation the role of a dense layer is using the vector representation to make the final prediction.

How to get the final prediction?

The vector representation consists of many numbers (vector dimensionality could be 1,024 or 2,048, so it's always something to the power of 2 for some reason).

Binary classification problem

Using this vector we want to build a model that makes a prediction. Let's start with a **binary classification problem**. “Is this image a t-shirt or not?”. The vector is x and $y = \{0, 1\}$ with 0 means not a t-shirt, and 1 means a t-shirt. Here we use logistic regression. $g(x) = \text{sigmoid}(xTw)$. x is the feature vector and we have to train a regression model to get the weights (w). The output of this would be the probability that this x is a t-shirt.

With this vector w we can make a prediction by multiplying $x_1 * w_1 + x_2 * w_2 + \dots + x_n * w_n$. Then we take this sum and turn it into probability by using sigmoid. The output then is the probability for being a t-shirt.

Multiple classification problem

What to do in case of multiple classes? We can build one model for each class – one for shirt, one for t-shirt, and one for dress. So we get another w vector for shirts and we can do the same for dresses. So we end up with three different w vectors each for one class. But in case of three sigmoid functions we use something different. A sigmoid for multiple classes is called softmax. The output of softmax will have three numbers (in this case here). The first number will contain the probability of x being a shirt, then the second will be the probability of x being a t-shirt and the last one for the probability of x being a dress.

So what happens here is, we put multiple logistic regressions together and as a result we got a neural network.

DENSE	Layer
Input	Output
$ x_1 $	
$ x_2 $	
$ x_3 $	$ o $
$ \cdot $	$ o $
$ \cdot $	$ o $
$ \cdot $	
$ x_n $	

This layer is called a dense layer. It's dense because it connects each element of the input with all the elements of its output. For this reason, these layers are sometimes called "fully connected".

Each of the output elements has its own w (w_1, w_2, w_3), so there is a W:

$$W = \begin{vmatrix} -w_1- \\ -w_2- \\ -w_3- \end{vmatrix}$$

All we need to do to transform the column vector x to the output is W^*x . That means the dense layer is a matrix multiplication.

We can put multiple dense layers together. Then we have an inner dense layer and an outer dense layer.

Summary

To summarize we have seen the following steps from image to prediction.

image \rightarrow conv layers \rightarrow vector representation \rightarrow dense layers \rightarrow prediction

This gives just a high-level overview about the internals of convolutional neural networks. There are many other internals and other layers as well. For more in-depth knowledge check the notes of the course [**CS231n Convolutional Neural Networks for Visual Recognition**](#).

There is one important layer that hasn't covered here, the "pooling layer". The purpose of this layer is that we can reduce the size of convolutional layers. The reason for doing this is reducing the size leads to having fewer parameters. You can for example reduce a 200×200 into a 100×100 image. Especially this layer is described very good in the course

ML Zoomcamp 2023 – Deep Learning – Part 6

👤 Peter ⏰ 21. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, ML Zoomcamp, Transfer Learning



This part is divided into 2 sections. This article is about the first part where we want to read image data using the `ImageDataGenerator`. The second article covers the second part where we want to train an Xception model.

1. [Transfer Learning – Part 1/2](#)
 1. [Reading data with `ImageDataGenerator`](#)

Transfer Learning – Part 1/2

This time we want to use an already trained network that was trained on ImageNet. The filters that were learned are quite generic so they can be used for many purposes. The model learned to take an image and convert it into a vector representation. This part (without dense layers) is quite generic and we don't really change it for our task. To train these filters in the convolution layers is very difficult because a lots of images is needed. Then there is a bunch of dense layers after the conversion into vector representation for making the final prediction. This second part of dense layers this is specific to the dataset that is used. In this case its specific to ImageNet and that dataset is trained on 1,000 different classes, which we don't need for our task. While the vector representation is very useful the dense layers are not. So we don't need the second part. That means we keep the convolutional layers but we want to train new dense layers. This is the idea behind transfer learning.

Reading data with `ImageDataGenerator`

Use image size of 150×150 to experiment faster, because to train 299×299 needs four times more. That means to train model faster we will use smaller images and then at the end we will retrain bigger images. `batch_size` means how many images are needed at once. The

shape will be (32, 150, 150, 3). Then the batch will go through convolutional layers and then we will have 32 vectors and also 32 predictions at the end.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_gen = ImageDataGenerator(preprocessing_function=preprocess_input)
train_ds = train_gen.flow_from_directory(
    './clothing-dataset-small/train',
    target_size=(150, 150),
    batch_size=32
)
# Output: Found 3068 images belonging to 10 classes.
```

For now we see that 3,068 were found belonging to 10 classes. To see which classes are there we can use:

```
train_ds.class_indices
# Output:
# {'dress': 0,
#  'hat': 1,
#  'longsleeve': 2,
#  'outwear': 3,
#  'pants': 4,
#  'shirt': 5,
#  'shoes': 6,
#  'shorts': 7,
#  'skirt': 8,
#  't-shirt': 9}
```

The names are inferred from the folder structure. That means everything that is inside the folder t-shirt is put under the t-shirt class.

```
# !ls -l clothing-dataset-small/train
!ls clothing-dataset-small/train
# Output: dress  hat  longsleeve  outwear  pants  shirt  shc
```

Then we can look at what this dataset generates. We use an iterator to be able to get the next batch. (A for-loop does this internally).

```

next(train_ds)

# Output:
# (array([[[[ 0.03529418, -0.09803921, -0.30196077],
#           [ 0.05098045, -0.08235294, -0.2862745 ],
#           [ 0.06666672, -0.06666666, -0.27058822],
#           [
#             ...,
#             [ 0.07450986, -0.05882353, -0.26274508],
#             [ 0.02745104, -0.10588235, -0.3098039 ],
#             [ 0.04313731, -0.09019607, -0.29411763]],
#           [
#             [[ 0.04313731, -0.09019607, -0.29411763],
#               [ 0.06666672, -0.06666666, -0.27058822],
#               [ 0.082353 , -0.05098039, -0.25490195],
#               ...
#               [ 0.09803927, -0.03529412, -0.23921567],
#               [ 0.14509809,  0.01176476, -0.19215685],
#               [ 0.09803927, -0.03529412, -0.23921567]],
#             [
#               ...,
#               [
#                 [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
#                 [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
#                 [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
#                 [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
#                 [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]]],
```

This returns two things. It returns our features the images and then y which are the labels.

```
X, y = next(train_ds)
```

The output of X is nothing new. What happens here is a bunch of images is loaded and then the function “preprocess_input” function is applied. To look at the shape of X, we see that is exactly what we expect here.

```

X

# Output:
# array([[[[ 0.30980396,  0.20784318,  0.13725495],
#           [ 0.30980396,  0.16078436,  0.20784318],
#           [ 0.22352946,  0.04313731,  0.10588241],
#           ...
#           ...,
#           [ 0.30980396,  0.06666672, -0.64705884],
#           ...
#           ...,
#           [ 0.32549024,  0.05882359, -0.70980394],
#           [ 0.3176471 ,  0.07450986, -0.6392157 ],
#           [ 0.3176471 ,  0.09019613, -0.6313726 ]]]], dtype=
```

```

X.shape

# Output: (32, 150, 150, 3)
```

For the labels one-hot encoding is used. As we have seen before the last column with index 9 is for t-shirts. So we have 3 t-shirts, 1 pents, and 1 shoes examples. This is how we do

multi-class classification. You can also think of this as 10 different binary variables and then fitting 10 different models. In real it's only one but you can conceptually think like this.

```
train_ds.class_indices  
  
# Output:  
# {'dress': 0,  
#  'hat': 1,  
#  'longsleeve': 2,  
#  'outwear': 3,  
#  'pants': 4,  
#  'shirt': 5,  
#  'shoes': 6,  
#  'shorts': 7,  
#  'skirt': 8,  
#  't-shirt': 9}  
  
y[:5]  
  
# Output:  
# array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
#        0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],  
#       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
#        0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],  
#       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
#        0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],  
#       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
#        0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],  
#       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
#        0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])  
# Rows are samples. This row is a shirt
```

Annotations on the code:

- "Columns are classes"
- "Rows are samples. This row is a shirt"

Let's do the same for validation data.

```
val_gen = ImageDataGenerator(preprocessing_function=preprocess)  
  
val_ds = val_gen.flow_from_directory(  
    './clothing-dataset-small/validation',  
    target_size=(150, 150),  
    batch_size=32,  
    shuffle=False  
)  
  
# Output: Found 341 images belonging to 10 classes.
```

The output is very similar as seen before. It finds 341 images belonging to the same 10 known classes.

👤 Peter ⏰ 21. November 2023 📄 Deep Learning, ML-Zoomcamp
⭐ Deep Learning, ML Zoomcamp, Transfer Learning

Leave a comment

ML Zoomcamp 2023 – Deep Learning – Part 7

👤 Peter ⏰ 22. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, ML Zoomcamp, Training, Transfer Learning



This is the second part of Transfer Learning section. While in the last article we looked at reading image data this article covers the training part.

1. [Transfer Learning – Part 2/2](#)
 1. [Train Xception on smaller images \(150×150\) \(Better to run it with a GPU\)](#)

Transfer Learning – Part 2/2

Train Xception on smaller images (150×150) (Better to run it with a GPU)

So far for reading the data, now let's train a model. `base_model` here means that we take the convolution part of the Xception model that was trained on ImageNet and then train our custom model on top of that. This will have 10 classes.

To only keep the convolutional layers there is a parameter `include_top` that we need to set to false. “top” could be a bit confusing, but Keras stacks layers conceptionally from bottom to top. That means on top there are the dense layers. (*See diagram below*)

Next important point is we don't want to train this model, we only want to use it for extracting the vector representation. With “`base_model.trainable = False`” we can define the model as not trainable. That means when we train our model, we don't want to change the convolutional layers (=freezing the convolutional layers).

```

base_model = Xception(
    weights='imagenet',
    include_top=False,
    input_shape=(150, 150, 3))

```

base_model.trainable = False

Output: To Train The model we want to extract the vector representation, so we're freezing the convolutional layers in the diagram

Downloading data from <https://storage.googleapis.com/tensorflow/83683744/83683744> [=====] - 30s 0us

Next thing to do is creating a new top. First thing is specifying the input. Input is the part of the model that receive the images. This input then goes to the base model which we use to extract the vector representation.

```

inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs)
outputs = base
model = keras.Model(inputs, outputs)
preds = model.predict(X)

# Output: 1/1 [=====] - 3s 3s/step

preds.shape
# Output: (32, 5, 5, 2048)

```

What we get here is a 4-dimensional shape. 32 is the batch size. So for each image we've got a $5 \times 5 \times 2,048$ representation which doesn't look like a vector representation yet. So what we need to do now is turning this to a bunch of vectors. We can do this while chunking the $5 \times 5 \times 2048$ representation in slices of size 5×5 and put the average of this 25 values to a one-dimensional vector. This reduction of dimensionality is called pooling. In this case here we use average pooling – more concrete we do a 2D average pooling. In Keras we can do this pooling as shown in the next snippet.

```

inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs)

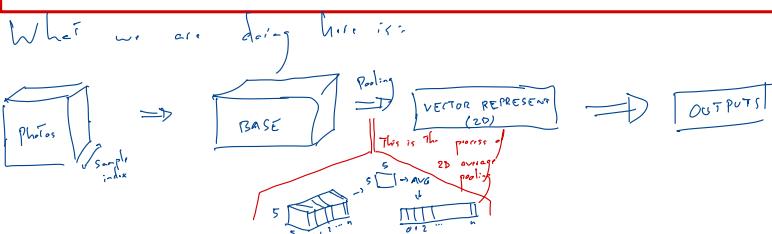
pooling = keras.layers.GlobalAveragePooling2D()
vectors = pooling(base)
outputs = vectors

model = keras.Model(inputs, outputs)
preds = model.predict(X)

# Output:
# WARNING:tensorflow:5 out of the last 6 calls to <function Model
# 1/1 [=====] - 3s 3s/step

preds.shape
# Output: (32, 2048)

```



```

# Shorter form in functional style. That's the way how we view th
inputs = keras.Input(shape=(150, 150, 3))

base = base_model(inputs)
vectors = keras.layers.GlobalAveragePooling2D()(base)
outputs = vectors
model = keras.Model(inputs, outputs)

preds = model.predict(X)
preds.shape

# Output:
# 1/1 [=====] - 3s 3s/step
# (32, 2048)

```

We're still not done yet. We have our vectors but now we need to put the dense layer on top of that to turn the vectors into predictions. What we want to have at the end is an array 32×10 with their predictions. That is what we call output. For turning vectors into outputs we want to create a dense layer.

	applying base model		pooling			
INPUTS	\Rightarrow	BASE	\Rightarrow	VECTORS	\Rightarrow	OUTPUTS
$32 \times 150 \times 150$		$32 \times 5 \times 5 \times 2048$		32×2048		32×10

```

inputs = keras.Input(shape=(150, 150, 3))

base = base_model(inputs)
vectors = keras.layers.GlobalAveragePooling2D()(base)
outputs = keras.layers.Dense(10)(vectors)
model = keras.Model(inputs, outputs)

preds = model.predict(X)
preds.shape

# Output:
# 1/1 [=====] - 3s 3s/step
# (32, 10)

```

To summarize what we've done so far (for one image). We have our t-shirt ($150 \times 150 \times 3$) as input. Applying the `base_model` on inputs gives us the "base" ($5 \times 5 \times 2048$). Applying pooling on "base" we turn this into a one-dimensional vector (2048). On top of this we added the dense layer, which turns the vector representation into predictions. The dimensionality of this is 10 because we have 10 classes. This is what goes to the outputs variable. "outputs" is what we will have when we invoke predict and the input is the X.

```

base_model = Xception(
    weights='imagenet',
    include_top=False,
    input_shape=(150, 150, 3)
)

base_model.trainable = False
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vectors = keras.layers.GlobalAveragePooling2D()(base)
outputs = keras.layers.Dense(10)(vectors)
model = keras.Model(inputs, outputs)

preds = model.predict(X)
preds.shape

# Output:
# 1/1 [=====] - 2s 2s/step
# (32, 10)

```

```

preds[0]

# Output:
# array([-0.03203598,  1.0506403 ,  1.5579934 , -0.47385398,
#        0.09166865, -1.2453439 , -0.7762855 ,  0.753671,
#        dtype=float32)

```

What's important the model outputs here just random numbers because we haven't trained the model yet. The reason for this is, when creating a dense layer it's initialized with random numbers. That means we now have to train the model.

To train a model we need to have some things.

First we need an optimizer which finds the best weights for the model. For more information on optimizers look at the [documentation](#). But also the [CS231n course](#) is a great reference for more information on that.

The learning rate here is similar to eta in case of XGBoost.

Then the optimizer needs to evaluate the changes are, therefor we need a concept that is called loss. Keras has some different kind of losses. Because we have a multi-class classification problem we use "CategoricalCrossentropy". For binary classification problem we would use BinaryCrossentropy, and for regression problems we would use MeanSquaredError. CategoricalCrossentropy outputs just a number – the lower the better. The optimizer is trying to optimize this number to make it as low as possible. How the optimizer is doing this? The optimizer can change the parameters of the dense layers.

There is an important parameter "from_logits" which is set to "True". The documentation says "Whether 'y_pred' is expected to be a logits tensor. By default, we assume that 'y_pred' encodes a probability distribution. Note: Using from_digits=True may be more numerically stable."

Let's try to explain what's happening here.

When we talked about dense layers, then there was an input and an output layer. We applied softmax to this output. This softmax is called "activation". It takes the output from the output layer as input and turns it into a probability. And exactly this input is called "logits" which is the raw output of the dense layer before applying softmax. If we have softmax then we have probabilities, if we don't have softmax we have raw score. Here we want to use raw score so we set that value to True. That means we don't use activation here. In case we need probabilities we can set that value to False. But then we must change the dense layer code from

```
outputs = keras.layers.Dense(10)(vectors) --> outputs = keras.layers.Dense(10,  
activation='softmax')(vectors)
```

But we stay with true and left the other code unchanged.

Now we need to compile the model before we can start training it. For compiling we need the optimizer and the loss that we defined before. We're also interested in monitoring a special metric which is accuracy. At each step of training it will show the progress.

```
learning_rate = 0.01  
optimizer = keras.optimizers.Adam(learning_rate=learning_rate)  
loss = keras.losses.CategoricalCrossentropy(from_logits=True)  
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
```

Good for multiclass classification problem. For binary classification, use "BinaryCrossEntropy" and "Sigmoid" activation function. For regression, use "MeanSquaredError".

More stable than False. It uses score-rows rather than softmax. If you want to use softmax, set from_logits=False. To activate softmax in outputs, keras.layers.Dim(1).activation='softmax' (curves)

Now everything is ready for training the model. Here we use the `model.fit` method which need the training data, the epoch count for how many epochs the model should be trained and the validation data. One epoch means that we go over the whole training dataset once, not image by image, but in batches of size 32 as we have defined before. Last batch can be less than this predefined size of 32 images. When training a model we apply to one batch at a time and when it's done for all batches we call this one epoch. 10 epoch for example means go over the data 10 times.

The output shows the current loss which is CategoricalCrossentropy and the accuracy for each epoch.

```

model.fit(train_ds, epochs=10, validation_data=val_ds)

# Output:
# Epoch 1/10
# 96/96 [=====] - 152s 2s/step - loss: 1
# Epoch 2/10
# 96/96 [=====] - 163s 2s/step - loss: 0
# Epoch 3/10
# 96/96 [=====] - 125s 1s/step - loss: 0
# Epoch 4/10
# 96/96 [=====] - 121s 1s/step - loss: 0
# Epoch 5/10
# 96/96 [=====] - 148s 2s/step - loss: 0
# Epoch 6/10
# 96/96 [=====] - 147s 2s/step - loss: 0
# Epoch 7/10
# 96/96 [=====] - 141s 1s/step - loss: 0
# Epoch 8/10
# 96/96 [=====] - 151s 2s/step - loss: 0
# Epoch 9/10
# 96/96 [=====] - 137s 1s/step - loss: 0
# Epoch 10/10
# 96/96 [=====] - 137s 1s/step - loss: 0
# <keras.src.callbacks.History at 0x7fa9012e9880>

```

This time we want to access the loss and accuracy values. In case of XGBoost we needed to capture the output but here we don't need to do this because the model.fit method returns an history object which contains all this information.

```

history = model.fit(train_ds, epochs=10, validation_data=val_ds)

# Output:
# Epoch 1/10
# 96/96 [=====] - 140s 1s/step - loss: 1
# Epoch 2/10
# 96/96 [=====] - 137s 1s/step - loss: 0
# Epoch 3/10
# 96/96 [=====] - 124s 1s/step - loss: 0
# Epoch 4/10
# 96/96 [=====] - 134s 1s/step - loss: 0
# Epoch 5/10
# 96/96 [=====] - 135s 1s/step - loss: 0
# Epoch 6/10
# 96/96 [=====] - 144s 1s/step - loss: 0
# Epoch 7/10
# 96/96 [=====] - 146s 2s/step - loss: 0
# Epoch 8/10
# 96/96 [=====] - 131s 1s/step - loss: 0
# Epoch 9/10
# 96/96 [=====] - 132s 1s/step - loss: 0
# Epoch 10/10
# 96/96 [=====] - 130s 1s/step - loss: 0

```

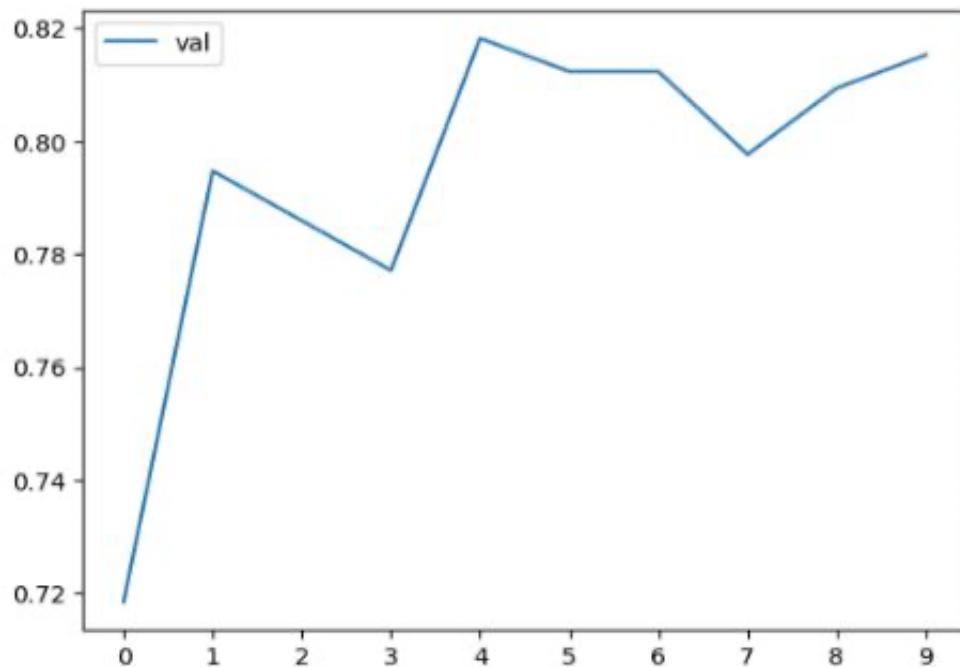
The training loss is decreasing and the accuracy for validation in the first epoch is 72%. Comparing the training accuracy with the validation accuracy we see both values increasing over time until epoch 3. There is a discrepancy between both accuracy values.

There is a difference of about 10%. Another thing is that training accuracy keeps improving but on the validation data not so much. That could mean that the model starts to overfit already. When we look at validation accuracy we see that it oscillates around 80%. In the same time the accuracy on training data is very high (it's almost 1). That are really good signs for an overfitting model. The results are saved in the history object. We're interested in training accuracy and validation accuracy.

```
history.history['accuracy']
#history.history['val_accuracy']

# Output:
# [0.6799217462539673,
#  0.829530656337738,
#  0.879726231098175,
#  0.9263363480567932,
#  0.942307710647583,
#  0.9605606198310852,
#  0.974250316619873,
#  0.991525411605835,
#  0.9788135886192322,
#  0.9853324890136719]
```

```
#plt.plot(history.history['accuracy'], label='train')
plt.plot(history.history['val_accuracy'], label='val')
plt.xticks(np.arange(10))
plt.legend()
```



The first peak is after one epoch. Maybe the best model is after 4 epochs because the training accuracy is still not very high (<95%). But the model after one epoch is already quite ok. This value of almost 80% is a good one because it's without any tuning. There are many parameters to tune. We'll tune the most important one in the next section.

ML Zoomcamp 2023 – Deep Learning – Part 8

👤 Peter

⌚ 23. November 2023

📁 Deep Learning, ML-Zoomcamp

🏷️ Deep Learning, Learning Rate, ML Zoomcamp



1. [Adjusting the learning rate](#)

1. [What's the learning rate ?](#)
2. [Trying different values](#)

Adjusting the learning rate

What's the learning rate ?

Let's use an analogy. Imagine that learning rate is how fast you can read and you read one book per quarter. That means you can read 4 books per year. Somebody else can read 1 book per day, so he can read many books per year. But maybe he is just skimming through them and looking at the table of content or just flipping the book through and looking at the concepts there. But when he tries to apply this readings, he cannot remember a lot.

Somebody else reads just one book per year and she reads very slow and takes some notes very carefully. This way she makes sure that she remember everything then she learns really well.

The many-books reader is similar to a high learning rate. This could be too fast. The expected performance on validation data could be poor, because the model tends to overfit.

The 4-books reader is similar to medium learning rate. This could be ok. The expected performance on validation data could be good.

And the last reader is similar to low learning rate. This is very effective but very very slow. The expected performance on validation data could be poor, because the model tends to underfit.

Trying different values

Therefor we use the code from the previous section and put it to a function. The middle part of this function could be separated to a function called “create_architecture”, but we’ll keep it simple here.

```
def make_model(learning_rate=0.01):
    base_model = Xception(
        weights='imagenet',
        include_top=False,
        input_shape=(150, 150, 3))
    base_model.trainable = False
    #####
    inputs = keras.Input(shape=(150, 150, 3))
    base = base_model(inputs, training=False)
    vectors = keras.layers.GlobalAveragePooling2D()(base)
    outputs = keras.layers.Dense(10)(vectors)
    model = keras.Model(inputs, outputs)
    #####
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=['accuracy'])
    return model
```

We want to iterate over different values of learning rate.

```

scores = {}

for lr in [0.0001, 0.001, 0.01, 0.1]:
    print(lr)

    model = make_model(learning_rate=lr)
    history = model.fit(train_ds, epochs=10, validation_data=val_ds)
    scores[lr] = history.history

    print()
    print()

# Output:
# 0.0001
# Epoch 1/10
# 96/96 [=====] - 135s 1s/step - loss: 1.9
# Epoch 2/10
# 96/96 [=====] - 127s 1s/step - loss: 1.4
# Epoch 3/10
# 96/96 [=====] - 135s 1s/step - loss: 1.1
# Epoch 4/10
# 96/96 [=====] - 138s 1s/step - loss: 1.0
# Epoch 5/10
# 96/96 [=====] - 137s 1s/step - loss: 0.9
# Epoch 6/10
# 96/96 [=====] - 132s 1s/step - loss: 0.8
# Epoch 7/10
# 96/96 [=====] - 136s 1s/step - loss: 0.7
# Epoch 8/10
# 96/96 [=====] - 136s 1s/step - loss: 0.7
# Epoch 9/10
# 96/96 [=====] - 130s 1s/step - loss: 0.7
# Epoch 10/10
# 96/96 [=====] - 129s 1s/step - loss: 0.6

# 0.001
# Epoch 1/10
# ...
# Epoch 10/10
# 96/96 [=====] - 125s 1s/step - loss: 1.1

del scores[0.1]
del scores[0.0001]

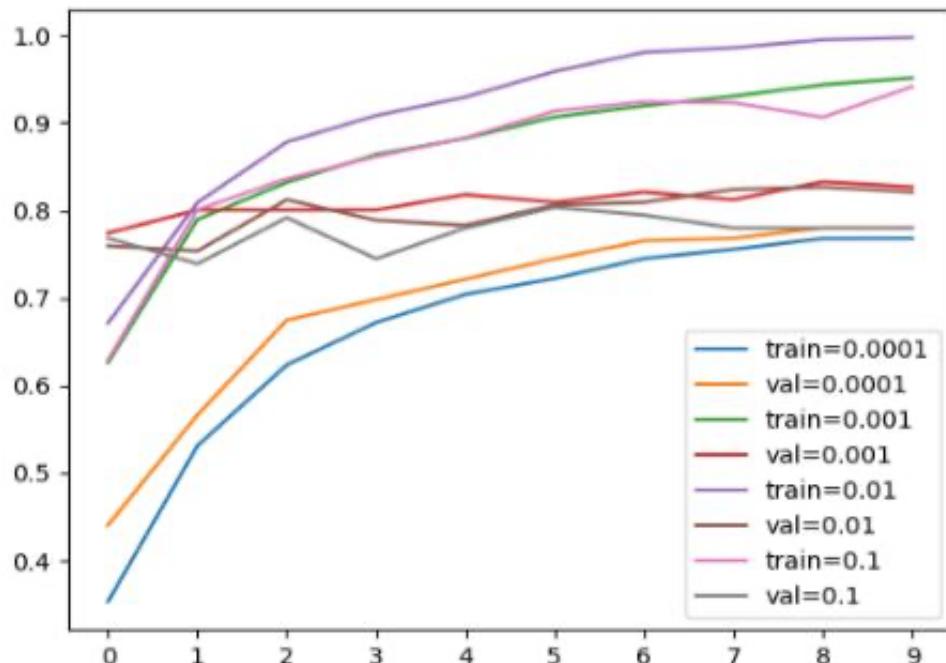
```

```

for lr, hist in scores.items():
    plt.plot(hist['accuracy'], label='train=%s' % lr)
    plt.plot(hist['val_accuracy'], label='val=%s' % lr)

plt.xticks(np.arange(10))
plt.legend()

```



learning_rate = 0.001

What we did so far we tried different models with different learning rates and the we choose this one that is best on validation data. Another interesting thing is comparing the difference between train and validation data between different learning rates – the smaller the better.

👤 Peter
⌚ 23. November 2023
💻 Deep Learning, ML-Zoomcamp

🏷️ Deep Learning, Learning Rate, ML Zoomcamp

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Deep Learning – Part 9

👤 Peter ⏰ 24. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ [Callback](#), [Checkpointing](#), [ML Zoomcamp](#), [Training](#)



1. [Checkpointing](#)

1. [Saving the best model only](#)
2. [Training a model with callbacks](#)

Checkpointing

Checkpointing is a way of saving our model after each iteration or when certain conditions are met, f.e. when the model achieves the best performance so far. This is a nice way because when the model starts to oscillate, the model after 10 iterations is not necessarily the best one.

Saving the best model only

How can we do this? After each epoch we trained we can evaluate the performance of the model on validation dataset. This we do for every epoch, then we look at the numbers and can invoke a callback. With this callback we can do anything we want. The evaluation on validation data is kind of callback. The history with all information is also kind of such a callback. This is just something we invoke after each epoch finishes

Training a model with callbacks

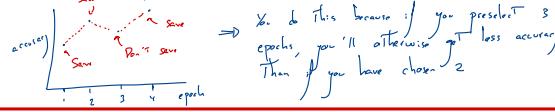
```
model.save_weights('model_v1.h5', save_format='h5')

# Keras uses this template for saving files.
'xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5'.format(epoch=3, val_accuracy=0.82)

# Output: 'xception_v1_03_0.840.h5'
```

`save_best_only=True` to save only when it's an improvement regarding the last best result. `mode='max'` because we want to have a maximized accuracy, if we would use a loss value we should take `mode='min'`.

```
checkpoint = keras.callbacks.ModelCheckpoint(  
    'xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5',  
    save_best_only=True, → It means that when running epochs (iterations) you only save the data if you increase the accuracy  
    monitor='val_accuracy',  
    mode='max'  
)  
Because we want to maximize accuracy. If it was:  
→ loss, we'd want to minimize
```



Now we can use this defined callback and retrain the best model using this callback.

```
learning_rate = 0.001  
  
model = make_model(learning_rate=learning_rate)  
  
history = model.fit(  
    train_ds,  
    epochs=10,  
    validation_data=val_ds,  
    callbacks=[checkpoint] → You pass the checkpoint  
)  
  
# Output:  
# Epoch 1/10  
# 96/96 [=====] - ETA: 0s - loss: 1.1087  
# 96/96 [=====] - 130s 1s/step - loss: 1  
# Epoch 2/10  
# 96/96 [=====] - 129s 1s/step - loss: 0  
# Epoch 3/10  
# 96/96 [=====] - 118s 1s/step - loss: 0  
# Epoch 4/10  
# 96/96 [=====] - 129s 1s/step - loss: 0  
# Epoch 5/10  
# 96/96 [=====] - 134s 1s/step - loss: 0  
# Epoch 6/10  
# 96/96 [=====] - 122s 1s/step - loss: 0  
# Epoch 7/10  
# 96/96 [=====] - 119s 1s/step - loss: 0  
# Epoch 8/10  
# 96/96 [=====] - 118s 1s/step - loss: 0  
# Epoch 9/10  
# 96/96 [=====] - 112s 1s/step - loss: 0  
# Epoch 10/10  
# 96/96 [=====] - 111s 1s/step - loss: 0
```

ML Zoomcamp 2023 – Deep Learning – Part 10

👤 Peter ⏰ 25. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, Dense Layer, ML Zoomcamp



1. [Adding more layers](#)

1. [Adding one inner dense layer](#)
2. [Experimenting with different sizes of inner layer](#)

Adding more layers

It's possible to add more inner layers after the vector representation. Previously we had one inner layer before outputting the prediction. This layer does some intermediate processing of the vector representation. These inner layers make the neural network more powerful.

Adding one inner dense layer

Usually adding one or two additional layers help and this is something we want to test. This layer we want to add is between the previous input and output. Let's add one inner dense layer with size of 100. For this new layer we need an activation. In neural networks each layer should have some transformation in order to achieve better performance. We will use here "relu" as activation function.

Activation functions:

- SIGMOID (mostly used for output)
- SOFTMAX (mostly used for output)
- RELU (mostly used for intermediate layers, default value)
- ...

For more information on that topic look again the mentioned CS231n course (Neural

To see how much gpu is utilized you can open a terminal right from Jupyter notebook and type “nvidia-smi”

Experimenting with different sizes of inner layer

```
def make_model(learning_rate=0.01, size_inner=100):
    base_model = Xception(
        weights='imagenet',
        include_top=False,
        input_shape=(150, 150, 3))
    base_model.trainable = False
    #####
    inputs = keras.Input(shape=(150, 150, 3))
    base = base_model(inputs, training=False)
    vectors = keras.layers.GlobalAveragePooling2D()(base)
    inner = keras.layers.Dense(size_inner, activation='relu')(vectors)
    outputs = keras.layers.Dense(10)(inner)
    model = keras.Model(inputs, outputs)
    #####
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)
    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=['accuracy'])
    return model
```

```

learning_rate = 0.001
scores = {}

for size in [10, 100, 1000]:
    print(size)

    model = make_model(learning_rate=learning_rate, size_inner=size)
    history = model.fit(train_ds, epochs=10, validation_data=val_c)
    scores[size] = history.history

    print()
    print()

# Output:
# 10
# Epoch 1/10
# 96/96 [=====] - 129s 1s/step - loss: 1.
# Epoch 2/10
# 96/96 [=====] - 138s 1s/step - loss: 1.
# Epoch 3/10
# 96/96 [=====] - 126s 1s/step - loss: 0.
# Epoch 4/10
# 96/96 [=====] - 140s 1s/step - loss: 0.
# Epoch 5/10
# 96/96 [=====] - 141s 1s/step - loss: 0.
# Epoch 6/10
# 96/96 [=====] - 127s 1s/step - loss: 0.
# Epoch 7/10
# 96/96 [=====] - 140s 1s/step - loss: 0.
# Epoch 8/10
# 96/96 [=====] - 119s 1s/step - loss: 0.
# Epoch 9/10
# 96/96 [=====] - 116s 1s/step - loss: 0.
# Epoch 10/10
# 96/96 [=====] - 137s 1s/step - loss: 0.

# 100
# Epoch 1/10
# ...
# Epoch 10/10
# 96/96 [=====] - 120s 1s/step - loss: 0.

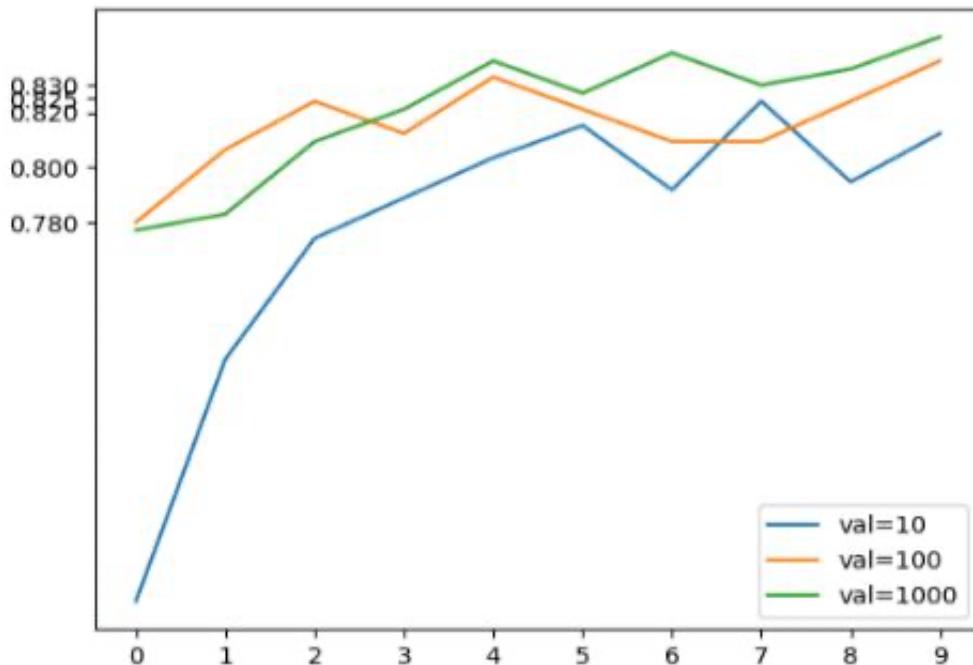
```

```

for size, hist in scores.items():
    plt.plot(hist['val_accuracy'], label='val=%s' % size)

plt.xticks(np.arange(10))
plt.yticks([0.78, 0.80, 0.82, 0.825, 0.83])
plt.legend()

```



When we cannot see an improvement compared to the less complex model then it's fine to go with this easier one. But in the next section we want to look at the effects of regularization and dropout so let's use the more complex neural network for now.

👤 [Peter](#) ⏰ [25. November 2023](#) 📄 [Deep Learning, ML-Zoomcamp](#)
 🔔 [Deep Learning, Dense Layer, ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Deep Learning – Part 11

👤 Peter 🕒 26. November 2023 📁 Deep Learning, ML-Zoomcamp
⭐ Deep Learning, Dropout, ML Zoomcamp, Regularization



Regularization and Dropout

When training a model for 10 epochs, then the model will see the same image 10 times. If there is a special sign like a logo, the model recognize it and tend to declare everything as t-shirt that has that logo. That means the model could make many mistakes in classification in validation data. That leads to a worse generalization capability. Instead the model should look at more general things like shapes instead of specific details.

What if we could randomly hide a part of the image? That is the main idea behind dropout. But in dropout we're not really hiding a part of the image but a part of the input. That means it applies the idea to inner layers.

Let's assume one fully connected dense layer with four inputs and three outputs. Dropout means that we freeze a part of this layer. That means that the frozen part does not get updated when running the current iteration. In the next iteration another part is frozen. By doing this we force the neural network to focus on the bigger picture (shape instead of details). But the output of the neural network still gets all flares. That means the output layer still looks at all parts – also the frozen ones.

Regularization means that we introduce something that doesn't let the neural network overfit to some patterns that might not exist. `droprate=0.5` means that in each iteration we freeze 50% of this layer. Dropout keeps the dimensionality of the layer.

We'll look at these points:

- Regularizing by freezing a part of the network
- Adding dropout to our model

- Experimenting with different values

```
def make_model(learning_rate=0.01, size_inner=100, droprate=0.5):
    base_model = Xception(
        weights='imagenet',
        include_top=False,
        input_shape=(150, 150, 3)
    )
    base_model.trainable = False
    #####
    inputs = keras.Input(shape=(150, 150, 3))
    base = base_model(inputs, training=False)
    vectors = keras.layers.GlobalAveragePooling2D()(base)

    inner = keras.layers.Dense(size_inner, activation='relu')(vectors)
    drop = keras.layers.Dropout(droprate)(inner)

    outputs = keras.layers.Dense(10)(drop)

    model = keras.Model(inputs, outputs)
    #####
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=['accuracy']
    )
    return model
```

The downside of dropout is that you'll need more iterations to learn something. Therefore we change the value from 10 to 30.

```

learning_rate = 0.001
size = 100

scores = {}

for droprate in [0.0, 0.2, 0.5, 0.8]:
    print(droprate)

    model = make_model(
        learning_rate=learning_rate,
        size_inner=size,
        droprate=droprate
    )

    history = model.fit(train_ds, epochs=30, validation_data=val_c
    scores[droprate] = history.history

    print()
    print()

# Output:
# 0.0
# Epoch 1/30
# 96/96 [=====] - 128s 1s/step - loss: 0.
# Epoch 2/30
# 96/96 [=====] - 102s 1s/step - loss: 0.
# Epoch 3/30
# 96/96 [=====] - 104s 1s/step - loss: 0.
...
# Epoch 29/30
# 96/96 [=====] - 103s 1s/step - loss: 0.
# Epoch 30/30
# 96/96 [=====] - 103s 1s/step - loss: 0.

# 0.5
# Epoch 1/30
# 96/96 [=====] - 110s 1s/step - loss: 1.
# Epoch 2/30
# 96/96 [=====] - 101s 1s/step - loss: 0.
# Epoch 3/30
# 96/96 [=====] - 102s 1s/step - loss: 0.
...
# Epoch 29/30
# 96/96 [=====] - 100s 1s/step - loss: 0.
# Epoch 30/30
# 96/96 [=====] - 100s 1s/step - loss: 0.

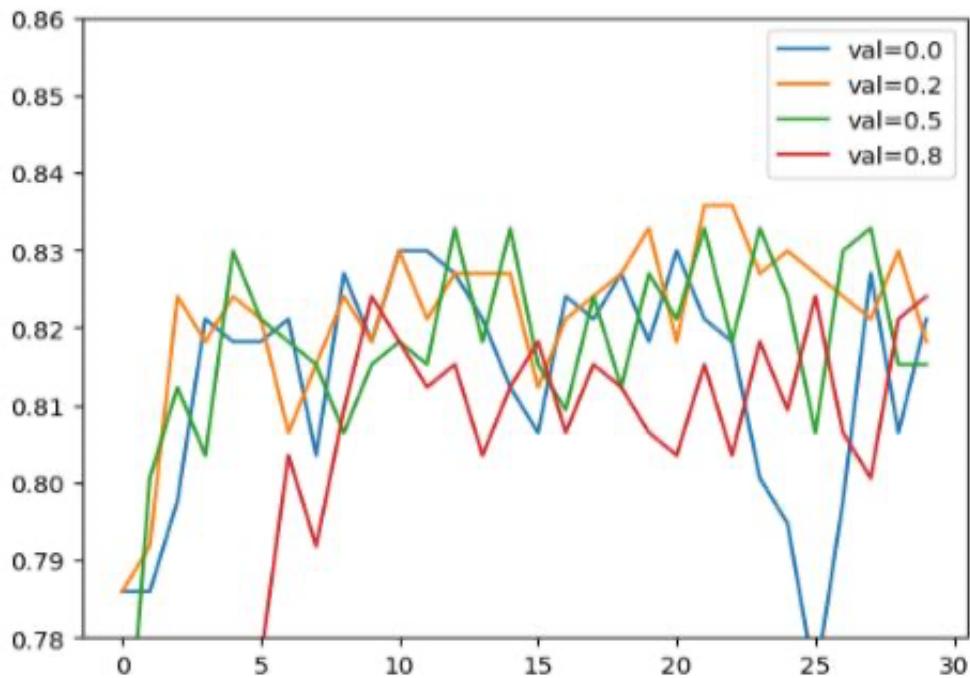
```

```

for droprate, hist in scores.items():
    plt.plot(hist['val_accuracy'], label='val=%s' % droprate)

plt.ylim(0.78, 0.86)
plt.legend()

```



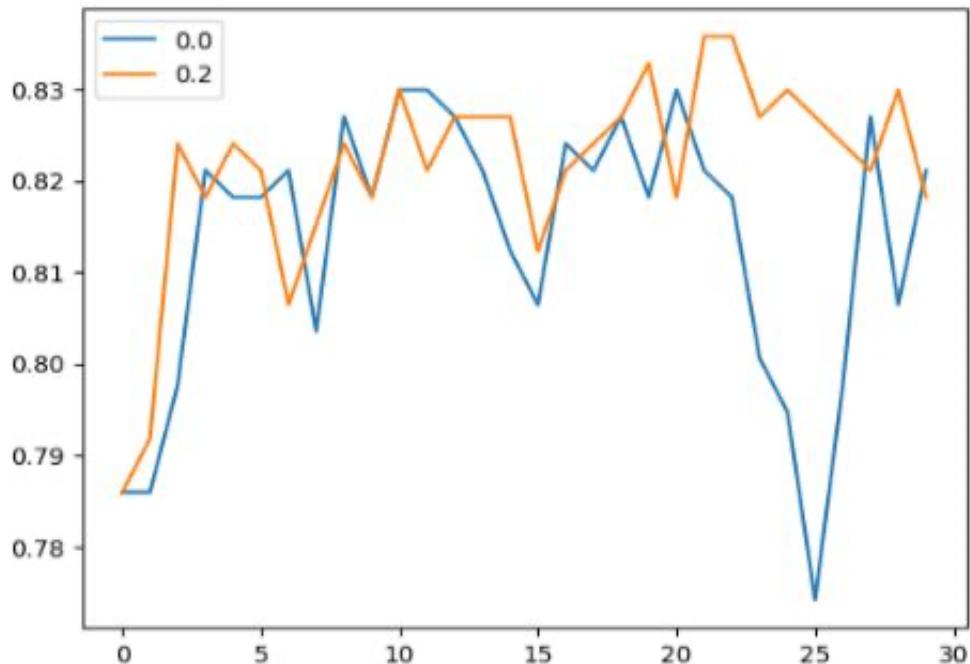
```

hist = scores[0.0]
plt.plot(hist['val_accuracy'], label=0.0)

hist = scores[0.2]
plt.plot(hist['val_accuracy'], label=0.2)

plt.legend()
# plt.plot(hist['accuracy'], label=('val=%s' % drop_rate))

```



ML Zoomcamp 2023 – Deep Learning – Part 12

👤 Peter ⏰ 27. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Data Augmentation, Deep Learning, ML Zoomcamp, Training



1. [Data Augmentation](#)

1. [Different data augmentations](#)
2. [Training a model with augmentations](#)
3. [How to select data augmentations?](#)

Data Augmentation

In the last section we talked about how to stabilize the network performance while we use dropout. This way we focus on overall shape instead of focussing on details like logos. It works because in each epoch another part of the network (not of the image) is hided/frozen. So small details like logos become more irrelevant.

Data augmentation is another approach for solving this problem which involves generating more images from existing ones. Let's imagine we take our t-shirt image and generate 10 more images, so then the neural network will not see exactly the same image every time.

Different data augmentations

There are different possible image transformations, that are also combinable:

- Flip an image vertically and horizontally
- Rotate an image
- Shift an image
- Shear an image (for example only move the upper right and lower right corners)
- Zoom in or out a bit (is like shrinking and extending)
- Change an image in other ways like brightness or contrast
- black patch (This means what I used to illustrate dropout. There is really a black patch)

that is randomly put on an image. That really hides a part of the image)

In Keras there is an image data generator. There is also a Jupyter notebook in mlbookcamp-code repository under chapter-07-neural-nets/07-augmentations.ipynb.

Training a model with augmentations

```
train_gen = ImageDataGenerator(  
    preprocessing_function=preprocess_input,  
    rotation_range=30,  
    width_shift_range=10,  
    height_shift_range=10,  
    shear_range=10,  
    zoom_range=0.1,  
    cval=0.0,  
    horizontal_flip=False,  
    vertical_flip=True,  
)
```

How to select data augmentations?

- Use your own judgement (does it make sense?) -> For example if you don't expect to see horizontally flipped images, then this wouldn't make sense here as well.
- Look at the dataset, what kind of variations are there?
 - Are the objects always centered? -> If not you can think about shifting and rotation.
- Tune it as a hyperparameter – Try different augmentations and see what works and what doesn't.
 - Train it with new augmentation for 10-20 epochs. If it's better then use it, if not then don't use it. If it's the same or similar result then train for some more epochs (like 20) and make the comparison again.

Playing around with the parameters of the previous snippet, the working parameters decrease a bit. The parameters can be applied to the training dataset, but we leave the validation dataset unchanged, because we want to have consistent results. Remember the parameter changes happen randomly and cannot be reproduced.

```
train_gen = ImageDataGenerator(  
    preprocessing_function=preprocess_input,  
    shear_range=10,  
    zoom_range=0.1,  
    vertical_flip=True,  
)  
  
train_ds = train_gen.flow_from_directory(  
    './clothing-dataset-small/train',  
    target_size=(150, 150),  
    batch_size=32  
)  
  
val_gen = ImageDataGenerator(preprocessing_function=preprocess_in  
val_ds = val_gen.flow_from_directory(  
    './clothing-dataset-small/validation',  
    target_size=(150, 150),  
    batch_size=32,  
    shuffle=False  
)  
  
# Output:  
# Found 3068 images belonging to 10 classes.  
# Found 341 images belonging to 10 classes.
```

```
train_gen = ImageDataGenerator(  
    preprocessing_function=preprocess_input,  
    #    vertical_flip=True,  
)  
  
train_ds = train_gen.flow_from_directory(  
    './clothing-dataset-small/train',  
    target_size=(150, 150),  
    batch_size=32  
)  
  
val_gen = ImageDataGenerator(preprocessing_function=preprocess_in  
val_ds = val_gen.flow_from_directory(  
    './clothing-dataset-small/validation',  
    target_size=(150, 150),  
    batch_size=32,  
    shuffle=False  
)
```

```

learning_rate = 0.001
size = 100
droprate = 0.2

model = make_model(
    learning_rate=learning_rate,
    size_inner=size,
    droprate=droprate
)

history = model.fit(train_ds, epochs=50, validation_data=val_ds)

# Output:
# Epoch 1/50
# 96/96 [=====] - 146s 1s/step - loss: 1.
# Epoch 2/50
# 96/96 [=====] - 145s 2s/step - loss: 0.
# Epoch 3/50
# 96/96 [=====] - 140s 1s/step - loss: 0.
...
# Epoch 49/50
# 96/96 [=====] - 115s 1s/step - loss: 0.
# Epoch 50/50
# 96/96 [=====] - 120s 1s/step - loss: 0.

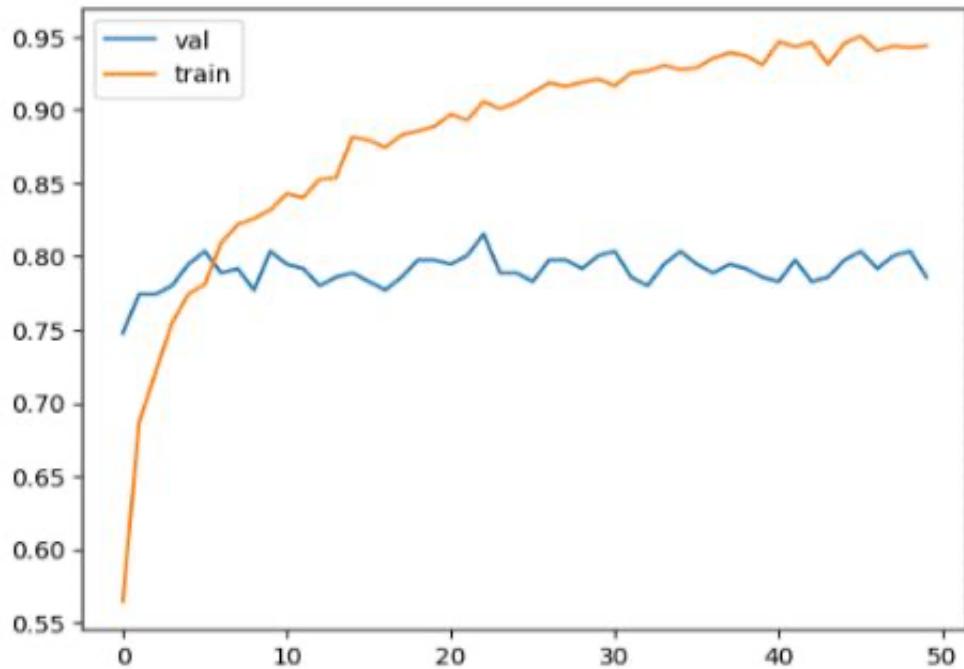
```

When doing augmentation it can happen, that the utilization of the GPU is not >90%. The reason could be that the augmentation is done by the CPU. After generating the new images, the GPU can fit the model. This is done for every epoch. To avoid that there are some complex things. While doing augmenting and fitting in a sequential order (first CPU: augmentation, second GPU: fitting) the CPU can start the second augmentation process right after the first one, then CPU and GPU are more utilized over time. Keras doesn't do this, but there are ways of doing this. (Google: tensorflow training pipeline with preprocessing -> tensorflow.org/guide/data_performance, and tensorflow.org/guide/data

```

hist = history.history
plt.plot(hist['val_accuracy'], label='val')
plt.plot(hist['accuracy'], label='train')
plt.legend()

```



Doing this testing with data augmentation we realize that this is not really helpful in this case, usually it is. Alexey said “Tuning neural networks is more art than science”. For this case here we can go with our untuned network that has an accuracy of around 84% which is sufficient for most of the use cases.

👤 [Peter](#) ⏰ [27. November 2023](#) 📄 [Deep Learning, ML-Zoomcamp](#)
🏷️ [Data Augmentation](#), [Deep Learning](#), [ML Zoomcamp](#), [Training](#)

Leave a comment

Write a comment...

[Comment](#)

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

ML Zoomcamp 2023 – Deep Learning – Part 13

👤 [Peter](#) ⏰ [28. November 2023](#) 📁 [Deep Learning, ML-Zoomcamp](#)
👉 [Deep Learning, ML Zoomcamp, Training](#)



Training a larger model

In the sections before we use images of size 150×150 because the model can be trained faster (here: 4 times faster). This is a good way when experimenting with parameters. Now we want to train on bigger images – let's use images of size 299×299 . Therefor we reuse the previous code for training and do a few changes.

We will use a new parameter `input_size` with default value of 150.

Train a 299×299 model

```
def make_model(input_size=150, learning_rate=0.01, size_inner=100, droprate=0.5):

    base_model = Xception(
        weights='imagenet',
        include_top=False,
        input_shape=(input_size, input_size, 3)
    )

    base_model.trainable = False

    #####
    inputs = keras.Input(shape=(input_size, input_size, 3))
    base = base_model(inputs, training=False)
    vectors = keras.layers.GlobalAveragePooling2D()(base)

    inner = keras.layers.Dense(size_inner, activation='relu')(vectors)
    drop = keras.layers.Dropout(droprate)(inner)

    outputs = keras.layers.Dense(10)(drop)

    model = keras.Model(inputs, outputs)

    #####
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    loss = keras.losses.CategoricalCrossentropy(from_logits=True)

    model.compile(
        optimizer=optimizer,
        loss=loss,
        metrics=['accuracy']
    )

return model
```

```
| input_size = 299
```

```
train_gen = ImageDataGenerator(  
    preprocessing_function=preprocess_input,  
    shear_range=10,  
    zoom_range=0.1,  
    horizontal_flip=True  
)  
  
train_ds = train_gen.flow_from_directory(  
    './clothing-dataset-small/train',  
    target_size=(input_size, input_size),  
    batch_size=32  
)  
  
val_gen = ImageDataGenerator(preprocessing_function=preprocess_inp  
val_ds = train_gen.flow_from_directory(  
    './clothing-dataset-small/validation',  
    target_size=(input_size, input_size),  
    batch_size=32,  
    shuffle=False  
)  
  
# Output:  
# Found 3068 images belonging to 10 classes.  
# Found 341 images belonging to 10 classes.  
  
checkpoint = keras.callbacks.ModelCheckpoint(  
    'xception_v4_1_{epoch:02d}_{val_accuracy:.3f}.h5',  
    save_best_only=True,  
    monitor='val_accuracy',  
    mode='max'  
)
```

```
learning_rate = 0.0005
size = 100
droprate = 0.2

model = make_model(
    input_size=input_size,
    learning_rate=learning_rate,
    size_inner=size,
    droprate=droprate
)

history = model.fit(train_ds, epochs=50, validation_data=val_ds,
                     callbacks=[checkpoint])

# Output:
# Epoch 1/50
# 96/96 [=====] - 574s 6s/step - loss: 1.0
# Epoch 2/50
# 96/96 [=====] - 572s 6s/step - loss: 0.0
# Epoch 3/50
# 96/96 [=====] - 546s 6s/step - loss: 0.0
...
# Epoch 49/50
# 96/96 [=====] - 573s 6s/step - loss: 0.0
# Epoch 50/50
# 96/96 [=====] - 587s 6s/step - loss: 0.0
```

👤 Peter ⏰ 28. November 2023 📚 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, ML Zoomcamp, Training

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

knowMLedge.com, 

ML Zoomcamp 2023 – Deep Learning – Part 14

👤 Peter ⏰ 29. November 2023 📁 Deep Learning, ML-Zoomcamp
🏷️ Deep Learning, ML Zoomcamp



1. [Using the model](#)

1. [Loading the model](#)
2. [Evaluating the model](#)
3. [Getting predictions](#)

Using the model

In the last section we trained the final model – model on bigger images and saved the best one which we want to use now to test it on test dataset and make predictions. In this article we will cover:

- Loading the model
- Evaluating the model
- Getting predictions

Loading the model

```
import tensorflow as tf
from tensorflow import keras

from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.applications.xception import preprocess_inp
```

```
test_gen = ImageDataGenerator(preprocessing_function=preprocess_in
test_ds = test_gen.flow_from_directory(
    './clothing-dataset-small/test',
    target_size=(299, 299),
    batch_size=32,
    shuffle=False
)
# Output: Found 372 images belonging to 10 classes.
```

```
| #model = keras.models.load_model('xception_v4_1_13_0.903.h5')
| model = keras.models.load_model('xception_v1_09_0.839.h5')
```

Evaluating the model

The evaluation is done quite easily just use the evaluate function of the model and provide the test data. The output consists of two numbers. The first value is the loss value and the second one is the accuracy on test dataset. When the accuracy is almost the same like the model performance was before means that the model does not overfit and we trained a good model.

```
model.evaluate(test_ds)
# Output:
# 12/12 [=====] - 57s 5s/step - loss: 0.7
# [0.712611973285675, 0.8064516186714172]
```

Getting predictions

Because we have a good model now we can apply it to an image.

```
import numpy as np
path = 'clothing-dataset-small/test/pants/c8d21106-bbdb-4e8d-83e4-
img = load_img(path, target_size=(299, 299))
x = np.array(img)
X = np.array([x])
X.shape
# Output: (1, 299, 299, 3)
X = preprocess_input(X)
pred = model.predict(X)
```

```
classes = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

dict(zip(classes, pred[0]))
```

Here we don't have probabilities but we have numbers up to 10, so this are the logits as mentioned before. This are the raw predictions. We can turn them into probabilities but we can treat them as relative likelihood of belonging to this class. That means we see a large likelihood belonging to the pants class. For getting the probability you can apply the softmax by your own.

👤 [Peter](#) ⏰ [29. November 2023](#) 📁 [Deep Learning, ML-Zoomcamp](#)
🏷️ [Deep Learning, ML Zoomcamp](#)

Leave a comment

This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

[knowMLedge.com](#), 



