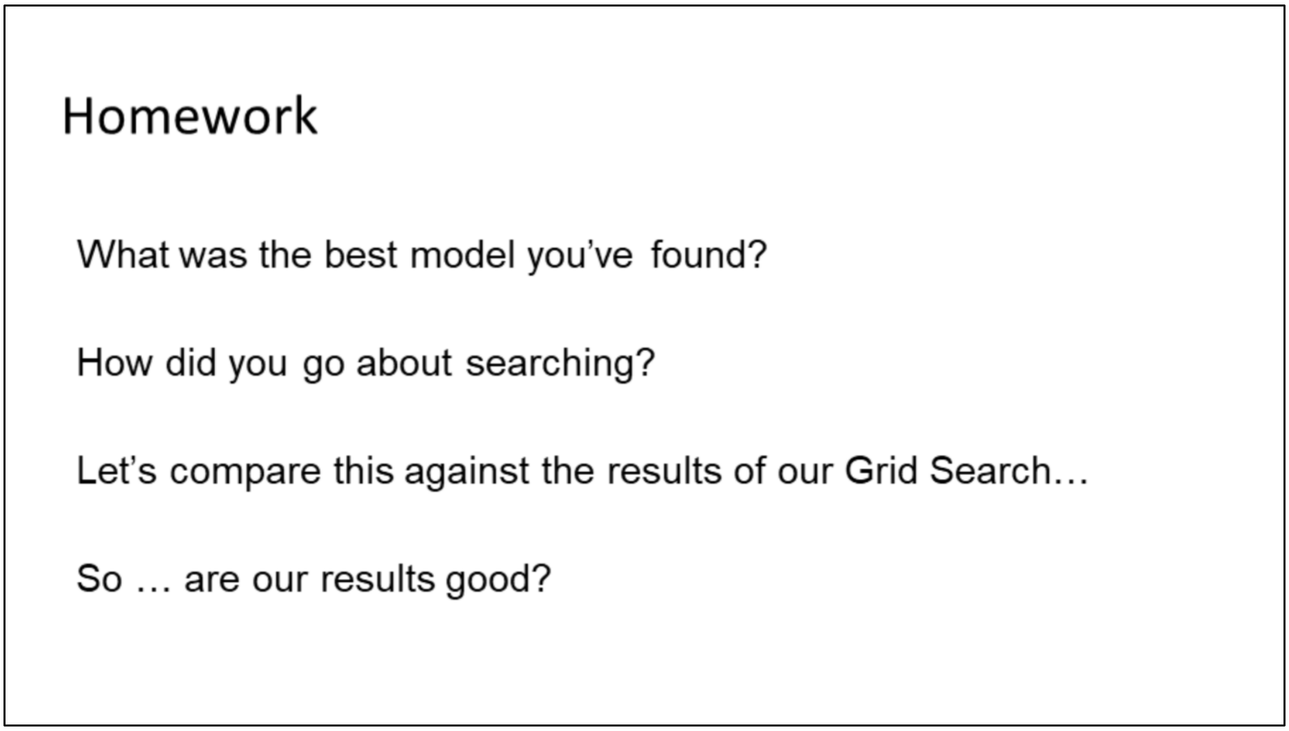
Complex Machine Learning Projects

Part 3: Define „Good“



Homework

What was the best model you've found?

How did you go about searching?

Let's compare this against the results of our Grid Search…

So … are our results good?

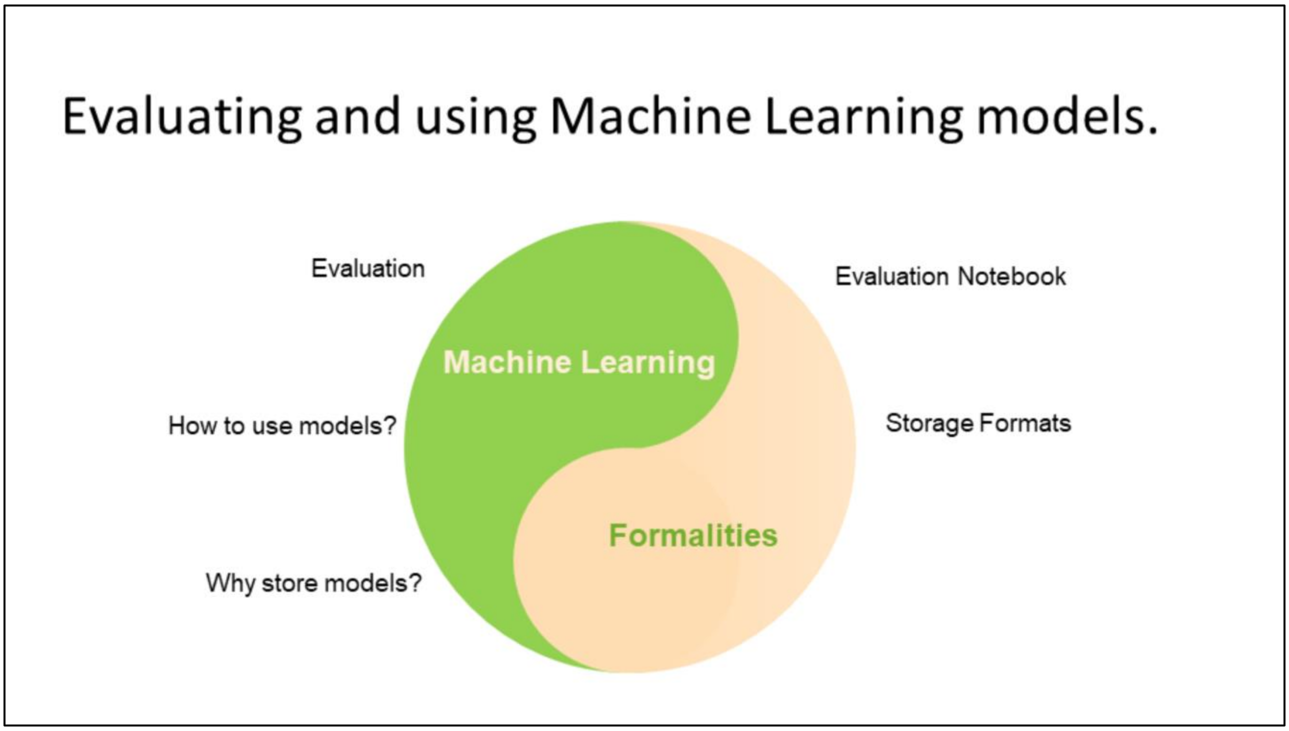
In this third part we want to talk about evaluating and finally using your models. We prepared this by giving you three questions to reflect on between sessions.

Here are a few possible answers to these questions:

How do you evaluate trained models? First and foremost, the models you have trained should be evaluated based on whether they are suitable for the purpose you want to use them for. This means, the criteria and benchmarks your model needs to fulfill depend on your purpose. This may involve evaluating the quality of the predictions in terms of classical metrics, such as accuracy, F1 score, mean error or other metrics. It may also involve evaluating the practicability of using the model in terms of memory requirements, prediction times, etc. When evaluating the result quality, you should always make sure to use data that has not been used to train or adjust the model.

How do you store trained models? There are plenty of use cases where you need to store your trained model. Maybe you want to store the final version to be used in a software system. Or you just want to make sure you don't have to retrain it every time you use it in a notebook. A trained model is a Python class, meaning it can be stored as other Python clases with the usual persistency library. There are also often model-specific persistency functions that let you store the model in a more efficient format or let you store the structure of the model independent of the learned parameters.

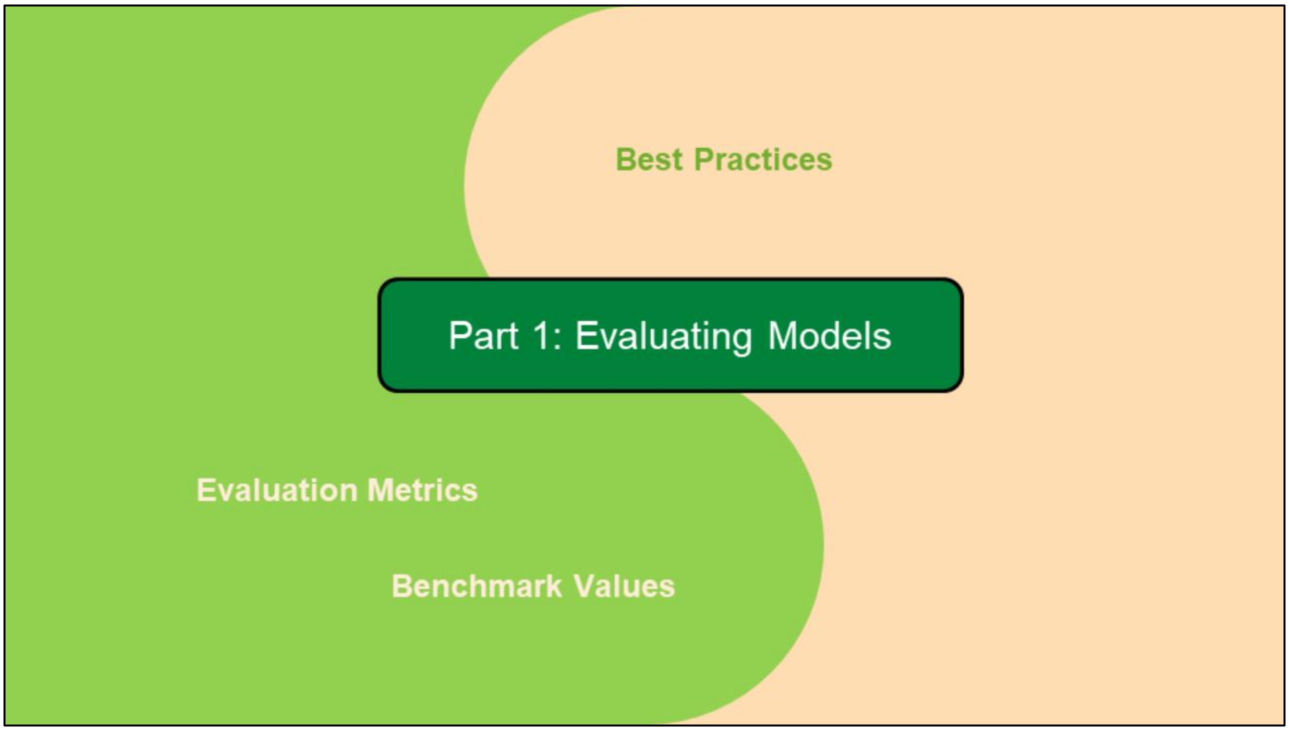
How do you use machine learning models as part of a software system? There is not a one-fits-all approach to using a Machine Learning model in a software system. As always, it depends on what you want to achieve. Maybe your machine learning model is just a cog in a bigger software system where it fulfills a specific purpose. In this case, it is likely that you will train it at design time and then load the trained model when you start your software system and use it to make predictions. Maybe your model is the central thing and you want to host it and build a user interface or API around it that enables others to use it. Or maybe you want to use it in context of Data Science and it is OK to train and use your model in the same notebook as part of researching a bigger question.



Evaluating and using Machine Learning models.

Evaluation
How to use models?
Why store models?
Machine Learning
Formalities
Evaluation Notebook
Storage Formats

Today we will talk about these three questions.
As usual, we will talk about the these topics both from a high level machine learning level and from a more formal level where we can give some practical tips.

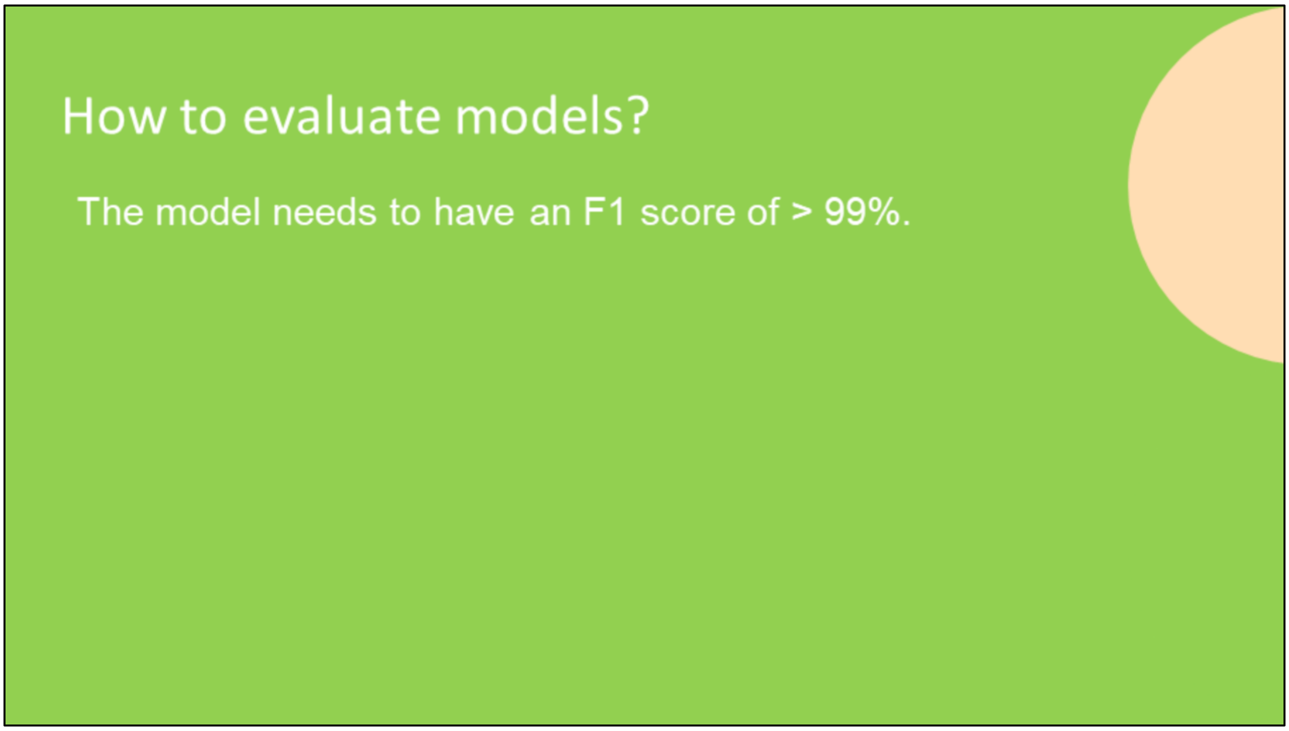


Best Practices

Part 1: Evaluating Models

Evaluation Metrics

Benchmark Values

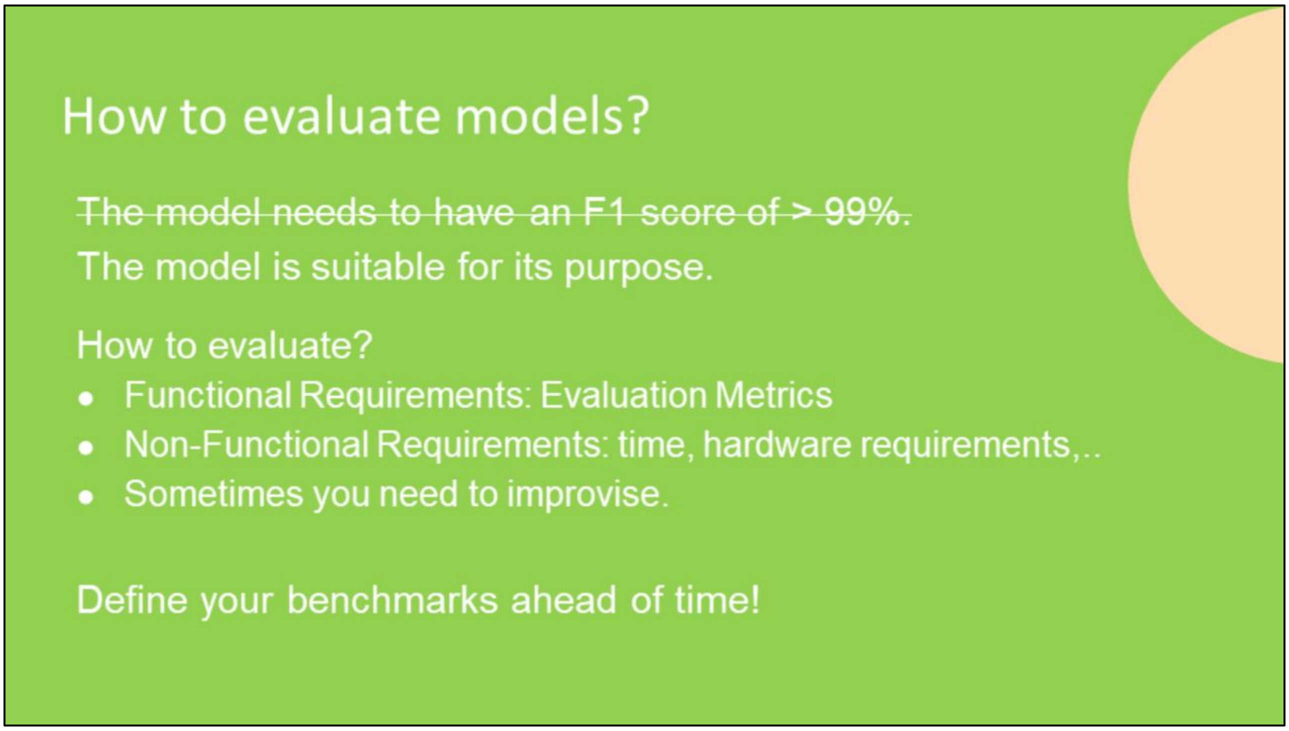


How to evaluate models?

The model needs to have an F1 score of > 99%.

So how do you evaluate models?

It’s easy. You measure the F1 score and if it is bigger than 99% it is good and if not, it is bad… NOT!


How to evaluate models?

~~The model needs to have an F1 score of > 99%.~~
The model is suitable for its purpose.

How to evaluate?
- Functional Requirements: Evaluation Metrics
- Non-Functional Requirements: time, hardware requirements,..
- Sometimes you need to improvise.

Define your benchmarks ahead of time!

Unfortunately, there is no general answer to the question of “when is a machine learning model good?”. Or at least the general answer is not as actionable as some would like: “It is good if you are able to use it for the purpose you want to use it for.”

For example: If you want to use your model inside of an autonomous vehicle to detect situations in which you need to trigger the emergency break then it needs to be extremely accurate. An accuracy of 99% may mean that one in a hundred emergency breaks is not executed correctly, which can endanger human lifes. On the other hand, if you are trying to build a prefilter for a human activity – let’s say you identify likely illegal images for a police officer that the police officer must then verify manually, guessing wrongly does not have as big of an impact. Both examples will likely also have different estimations on how critical false positives and false negatives are and which ones are more important.

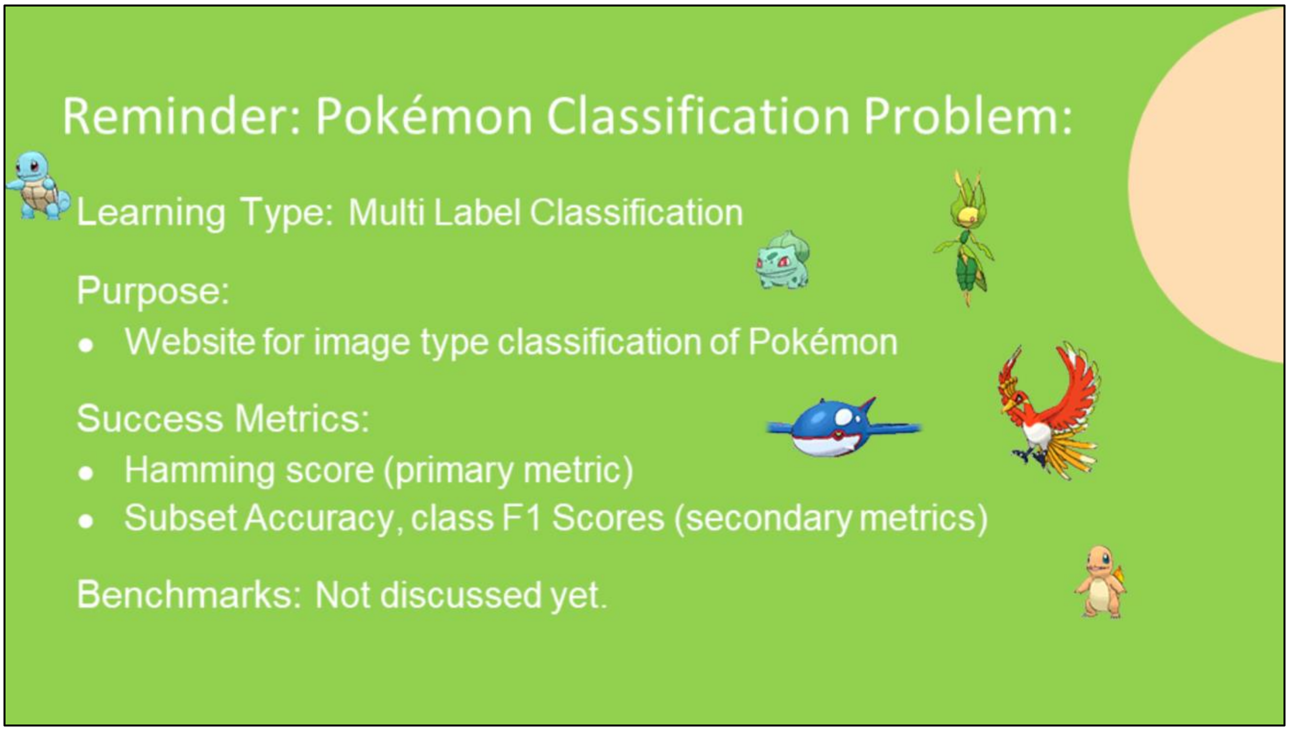
So how do you pick good evaluation metrics for your model? If you are lucky, you can use an established evaluation metric. We already discussed those a bit in our earlier sessions because we had to select them early so we can use them during hyper parameter selection. Generally, for a specific machine learning problem, you will find a set of existing metrics that you can consider to see which ones fit your purposes best. If, for example, you have a binary classification problem, you may be able to select between accuracy, Precision, Recall, F1 score and others, depending on whether and how much false positives and false negatives count for you.

In addition to selecting a metric, you should also define a benchmark value for you that you want your machine learning model to beat. If you are working on a more fine-grained problem, you may even have an idea of different ranges, that you could label to distinguish very good solutions from usable-but-mediocre solutions.

These metrics represent your functional requirements. Non-functional requirements may also be part of your evaluation. These can represent where you want your machine learning to run (i.e., how much memory you have available) or how long you can wait for a result.

Sometimes, you unfortunately also have to improvise an evaluation. An example are generative models that are intended to generate images or speech for which no easy automated function can test for correctness. Such situations may be harder to evaluate any may require a bit of creativity to find a suitable test setup.

Whatever you need to evaluate, it is a good idea to define your benchmarks ahead of time – even before you do model selection. On the one hand, you can use them already to make decisions about the utility of different models and intermediate results. On the other hand, it is easy to fall into a confirmation bias trap after seeing how your model performs and subconsciously wanting that model to be evaluated positively. Defining your benchmarks ahead of time excludes this bias.



Reminder: Pokémon Classification Problem:

Learning Type: Multi Label Classification

Purpose:

- Website for image type classification of Pokémon

Success Metrics:

- Hamming score (primary metric)
- Subset Accuracy, class F1 Scores (secondary metrics)

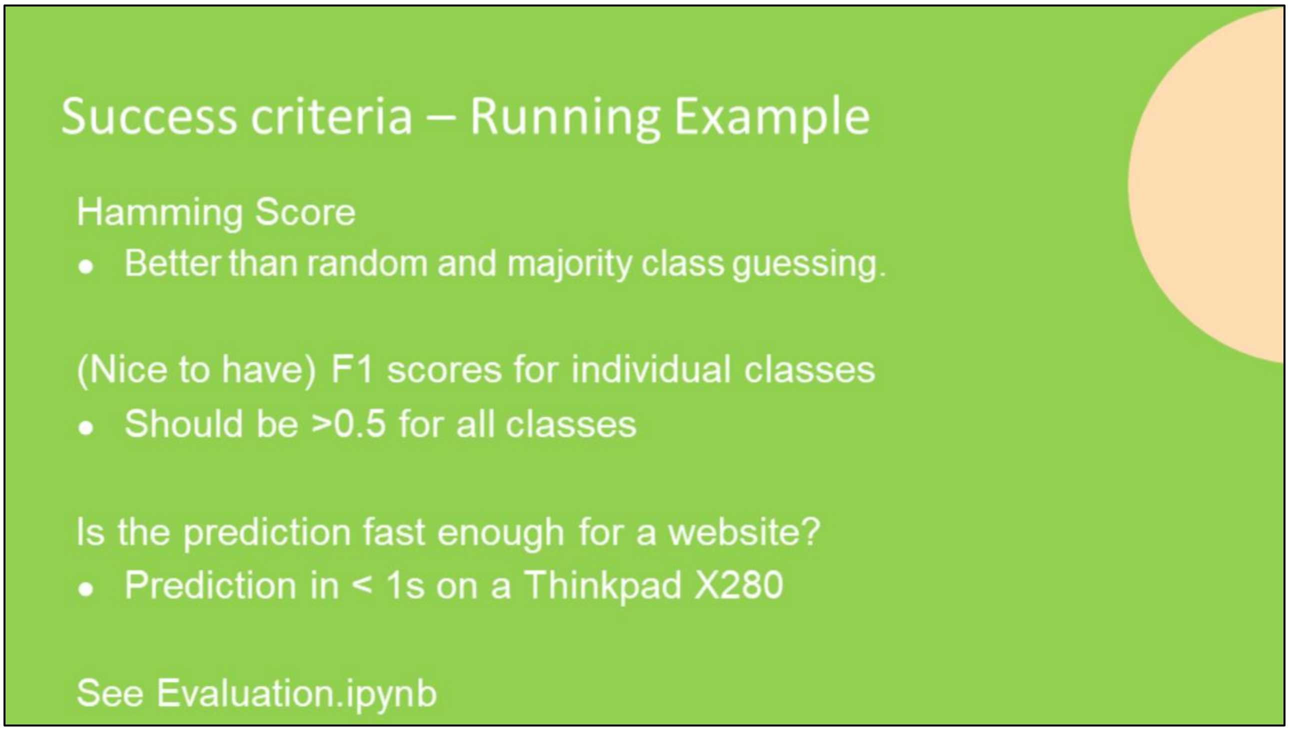
Benchmarks: Not discussed yet.

Here we see a short reminder of what we previously discussed for our running example.

Remember, We wanted to have a model that classifies Images for their Pokémon type that can be used as part of a website. The idea was to have someone upload an image and let the website tell them which Pokémon types they include.

Since Pokemon can have multiple types, we identified that this is a multi-label classification problem and decided that we want to use Hamming Score as main comparison metric. We also noted that while subset accuracy and F1 scores will not be used to make decisions, it makes sense to keep track of them as secondary metrics.

What we have not yet discussed is benchmarks in the sense of when we consider a model usable. We have only discussed using Random Guessing and Majority Class Guessing as Benchmarks to determine whether model has learned anything meaningful at all. But this is not necessarily the same. Being better than random guessing is not much of a claim.



Success criteria – Running Example

Hamming Score
- Better than random and majority class guessing.

(Nice to have) F1 scores for individual classes
- Should be >0.5 for all classes

Is the prediction fast enough for a website?
- Prediction in < 1s on a Thinkpad X280

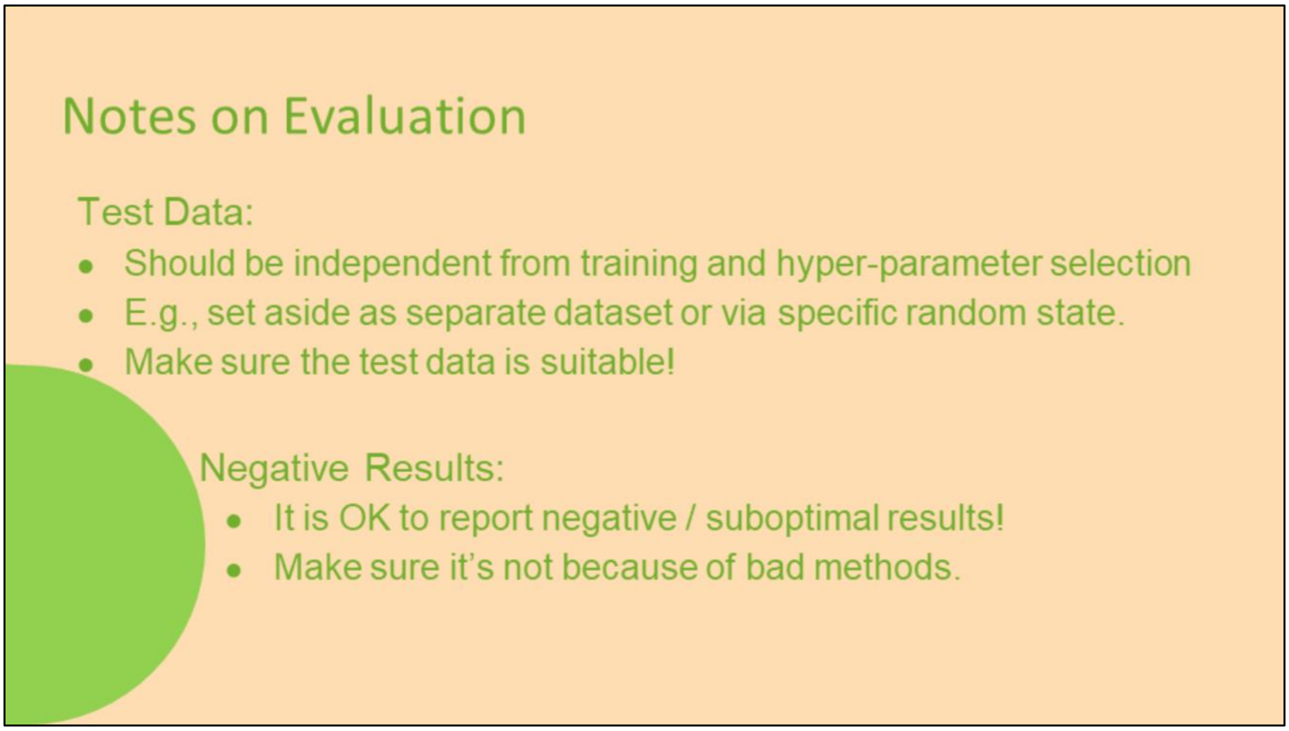
See Evaluation.ipynb

Let’s fill the gap and define definite benchmarks for our Pokémon classification.

The slide lists the three criteria that we would like to check. These criteria are derived from our intended application, including the fact that this is a teaching demonstrator. This is the reason why we only require the hamming score to be bigger than random and majority class guessing and consider the class-wise F1 scores to be nice to have. This essentially means “we would like our model to have learned something meaningful but we wouldn’t mind if it doesn’t achieve high scores because we can still use it to showcase how to deploy a model on a website.”. For a more serious application we would want a hamming score of >0.7, possibly even >0.9 and would probably also require an F1 score that is better than 0.8.

As a non-functional requirement we chose to be able to predict the result within 1 second on a Thinkpad X280. This is the machine we executed our evaluation on and it is significantly weaker than a normal Web Server. If the model is able to predict an image in less than a second on this machine we have no doubt it would be usable on a web server in under 1 Second. 1 Second has been chosen as Benchmark values because reaction times on HTML pages are usually consider irresponsive when over 2 Seconds and because we wanted to model to not take up more than 50% of that value. Again, we’ve made our job easy on this account because we build a teaching demonstrator and may have defined more fine-grained metrics that also include memory-usage in another case.

You can see the evaluation of these criteria in the notebook Evaluation.ipynb. According to this experiment, our model was evaluated to be usable but it did not even succeed in fulfillint the nice-to-have condition. It is definitely not a model that should be used outside of a no-stakes situation such as a teaching demonstrator.



Notes on Evaluation

Test Data:

- Should be independent from training and hyper-parameter selection
- E.g., set aside as separate dataset or via specific random state.
- Make sure the test data is suitable!

Negative Results:

- It is OK to report negative / suboptimal results!
- Make sure it's not because of bad methods.

A few notes on best practices for evaluation.

For evaluation, you should always use **test data** that your model has not seen before. If you use data that your model has been trained on then your evaluation will not show how your model performs on unknown data. However, generalizing to unknown data is often the hard part and should be the thing you test for. This also applies to data used during the hyper-parameter selection. While this may be a manual process, it still is based on data. So, if you've used a validation data set during hyper parameter selection, it should be different from your test data set.

If you use any form of data augmentation, like image generators, you should also make sure that your test data has not been used as source for the data augmentation. While an image rotated by five degrees is a different image, it still contains the original patterns to such an extend that it makes classifying the original image easier.

When you create a test data set you should also make sure it is suitable before proceeding. If you, for example, your test dataset does not contain all classes you want to classify you cannot use it to evaluate the missing classes. Finding this out late can be very bad as it may require you to re-run your experiments with a different training/test split.

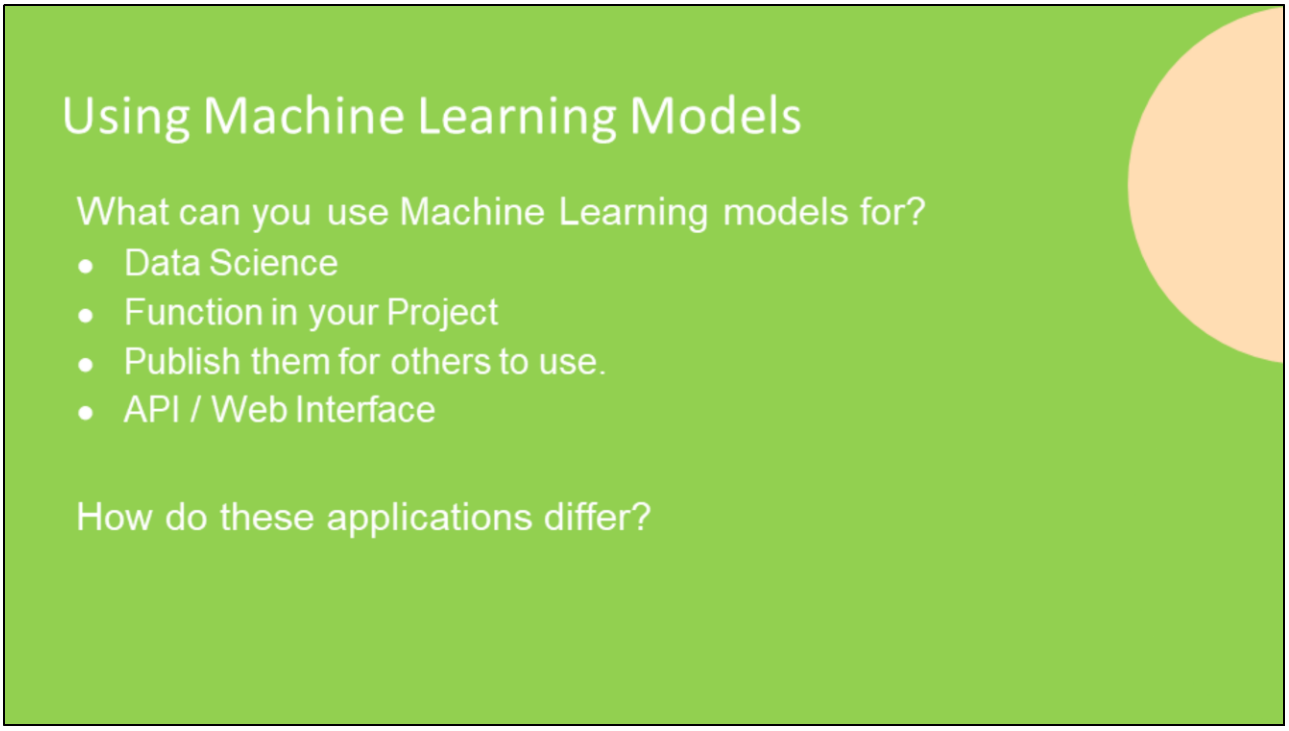
Negative results are results as well and should be reported. Sometimes you simply cannot achieve the results that you want to achieve for various reasons. A frequent reason is not having enough data. If you find out that you cannot find a good model for a given dataset, then that is something that is worth reporting – especially if you are dealing with a scientific context. By reporting negative results, you may spare others the time of trying the same. Of course, in this situation you should make extra sure that your results are not caused by bad assumptions or bad methodology.

If you have used data to train your model or to select hyper parameters, then there is a chance that the model has overfit to this data and would perform worse on other data.

when evaluating. This is data that has not been used to fine tune or learn your model.

Part 2: How to use models?

Common Use Cases


Using Machine Learning Models

What can you use Machine Learning models for?

- Data Science
- Function in your Project
- Publish them for others to use.
- API / Web Interface

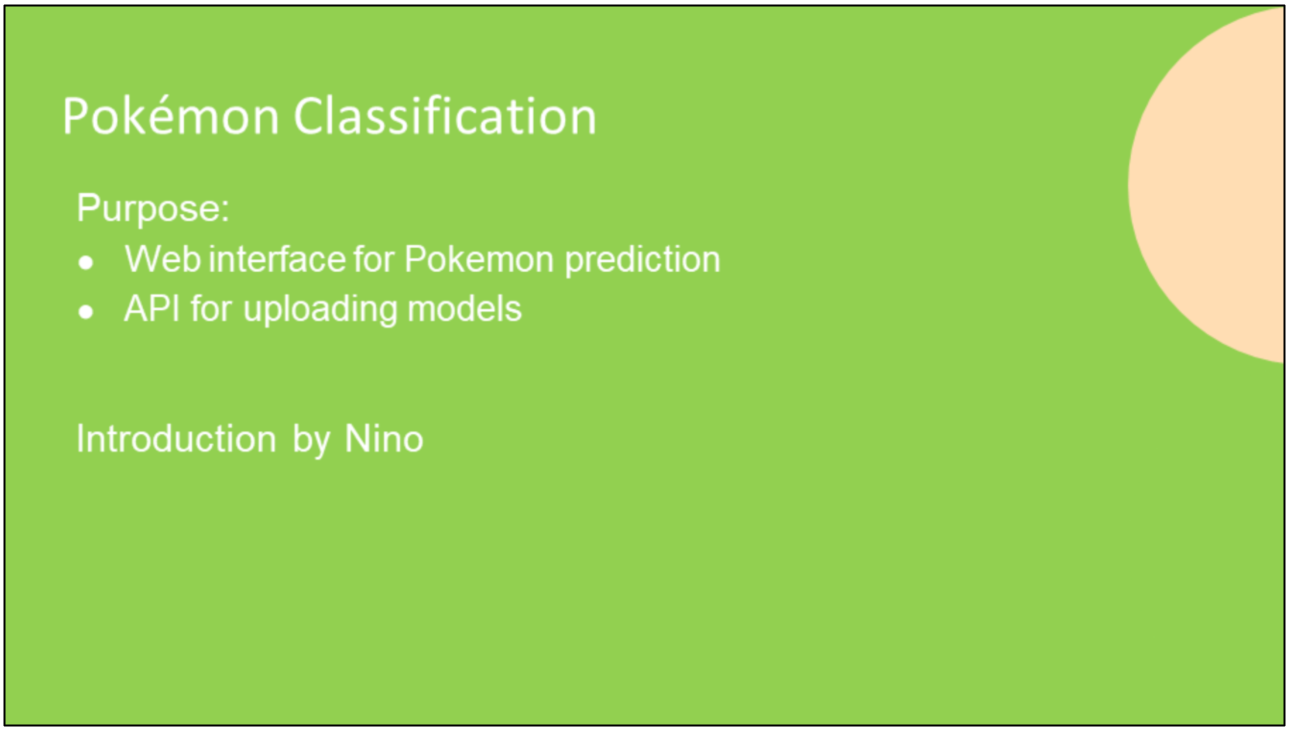
How do these applications differ?

On a high level of abstraction, machine learning is just a different way to define a function. Rather than writing the function with instructions you are learning the function from data. And just as there are different ways to use a function, there are different use cases for machine learning models.

A few common use cases are as follows:

1) In Data Science, Machine Learning is often used to uncover relations in data. This is usually done by a Person via a Python notebook. The machine learning model is learned and then analyzed or used in some way within the notebook.
2) Sometimes you also use machine learning as a function in a larger software engineering project. For example, you could learn a function that predicts how much the user prefers a certain item and use that function in scope of a web shop to order your results or fill a recommendation box. In such a project you often learn the machine learning model at design time and then use the learned model in your software system by loading it and using it to make predictions.
3) Sometimes, the machine learning model is also of interest to others and you want to publish it to them for use in their software project. In this case you don’t use the model itself but publish a file containing the model. If this is your use case you may have to think about issues like versioning and a mechanism to name and publish different snapshots of weights.
4) Finally, your machine learning model may be deployed in the web for others to use, either in the form of an API that programmers can make calls to or in form of a web interface that can be used to provide inputs. An example are recent

generative models, like Stable Diffusion and ChatGPT which are made available via websites such as huggingface.



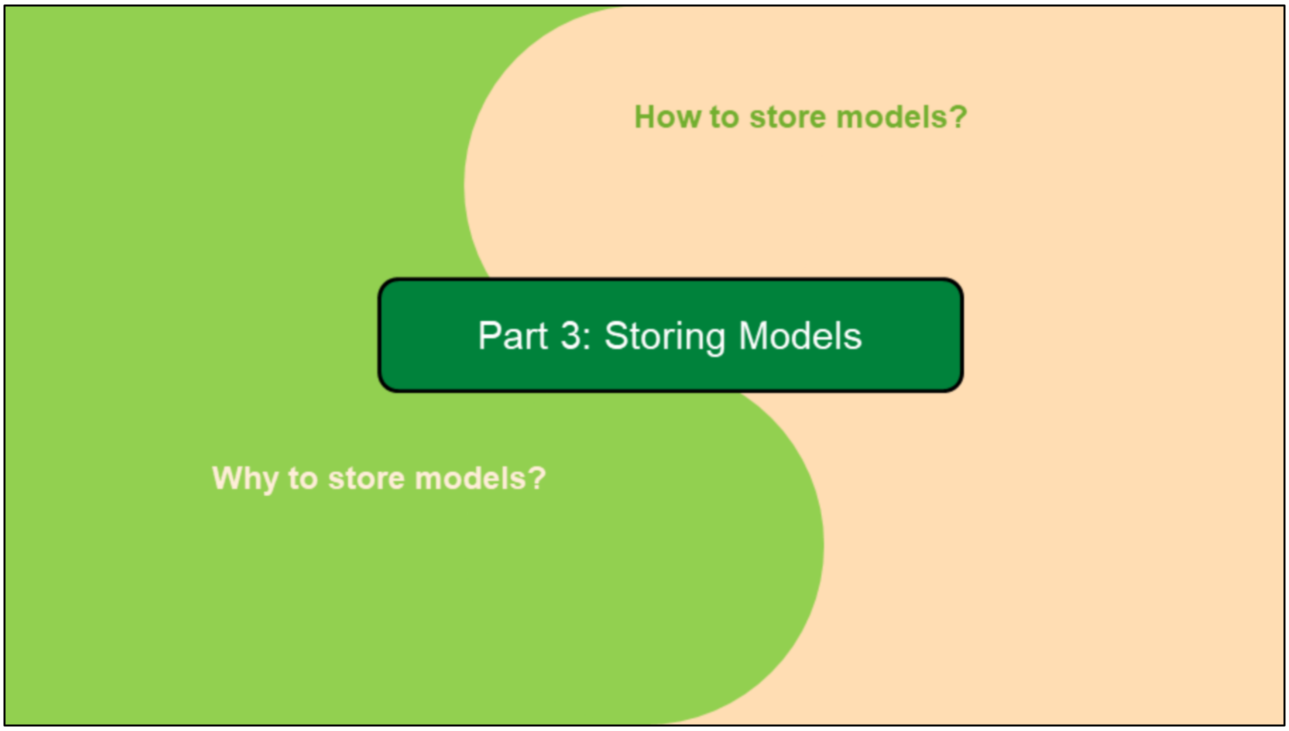
Pokémon Classification

Purpose:

- Web interface for Pokemon prediction
- API for uploading models

Introduction by Nino

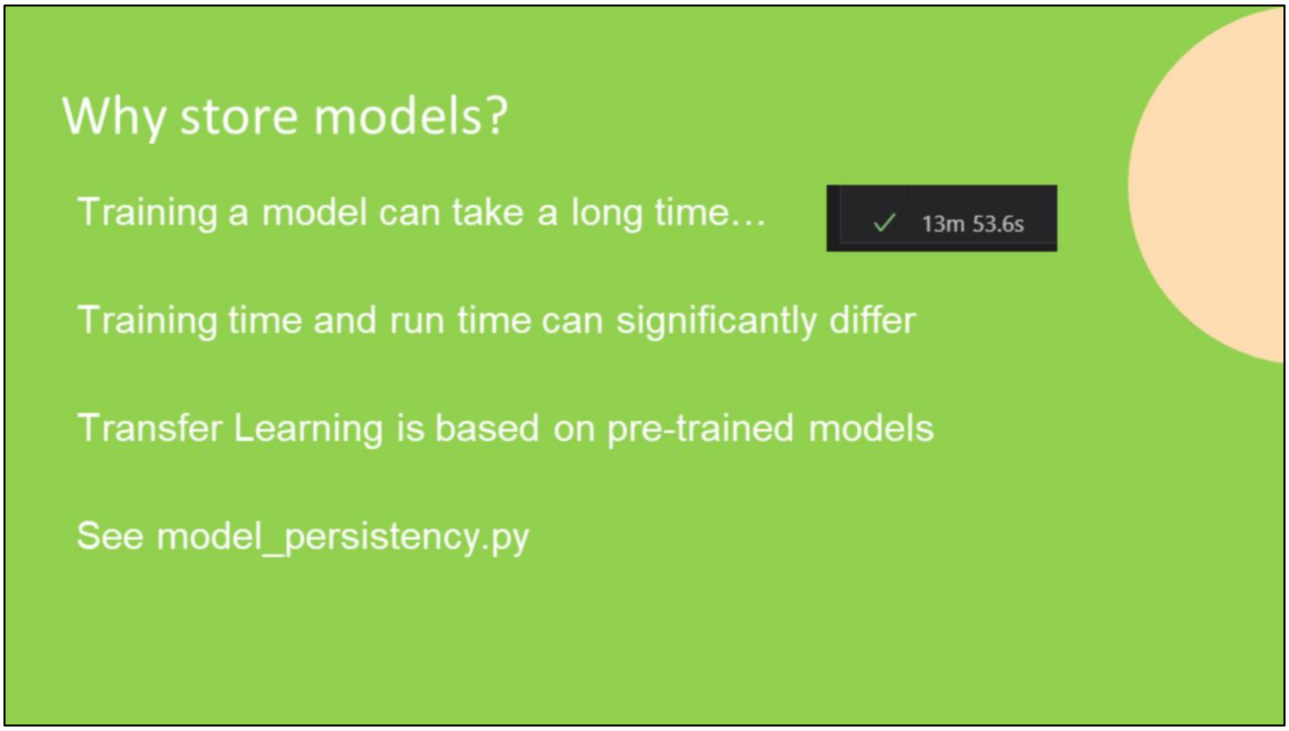
TODO (describe when demonstrator is finished)



Part 3: Storing Models

How to store models?

Why to store models?



Why store models?

Training a model can take a long time…

```
✓ 13m 53.6s
```

Training time and run time can significantly differ

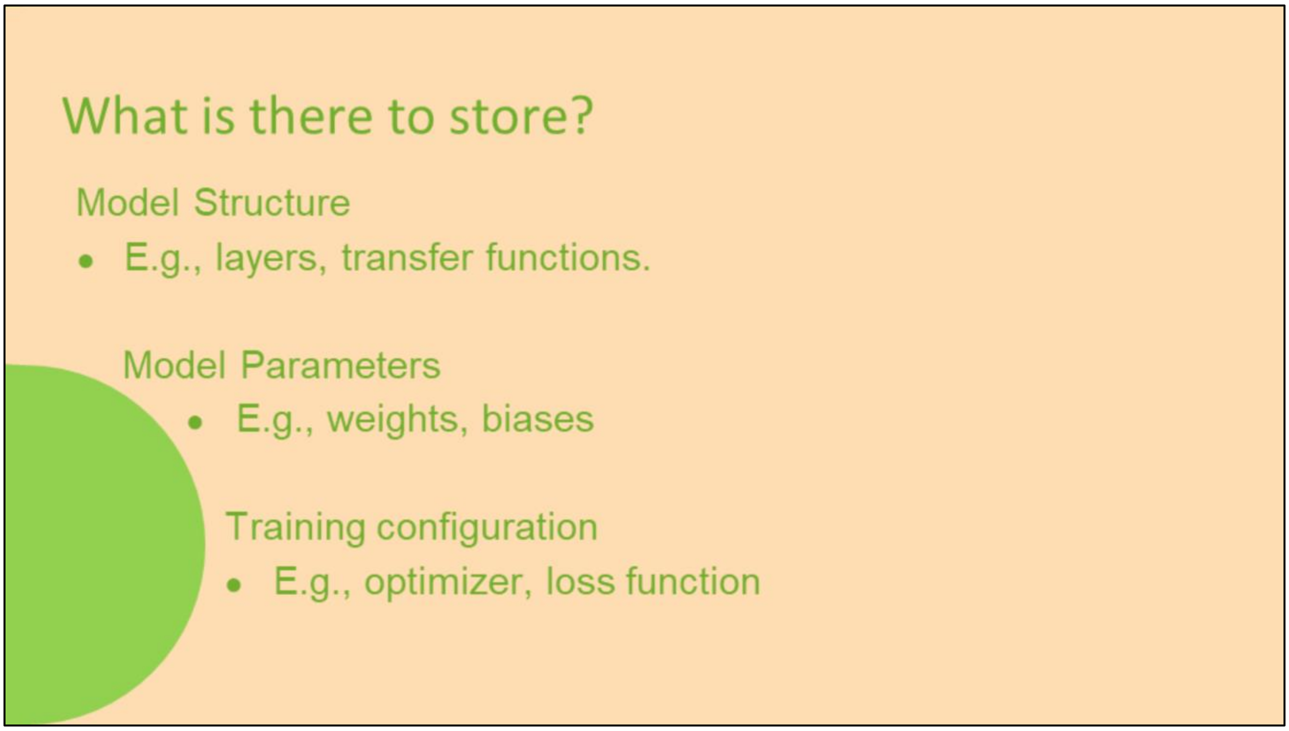
Transfer Learning is based on pre-trained models

See model_persistency.py

Some of the use cases we have discussed will require you to store your models. There is several situations in which this becomes beneficial:

1) Sometimes training can take a long time and you don't want to do it every time you use the model – even if it is in scope of a data science notebook.
2) Sometimes you want to just use a finished model in your software engineering project but don't want to have to train it yourself – or at least not train it yourself at run time.
3) Some forms of machine learning are based on partially trained models. This means you will have to store these models somewhere and make them available.

Storing the model usually means persisting them into some kind of file where they can be loaded later on. This gives you a clear distinction between the process of training and the process of using a model. It allows you to decouple the two and effectively use the model multiple times in different ways without having to repeat the training.



What is there to store?

Model Structure

- E.g., layers, transfer functions.

Model Parameters

- E.g., weights, biases

Training configuration

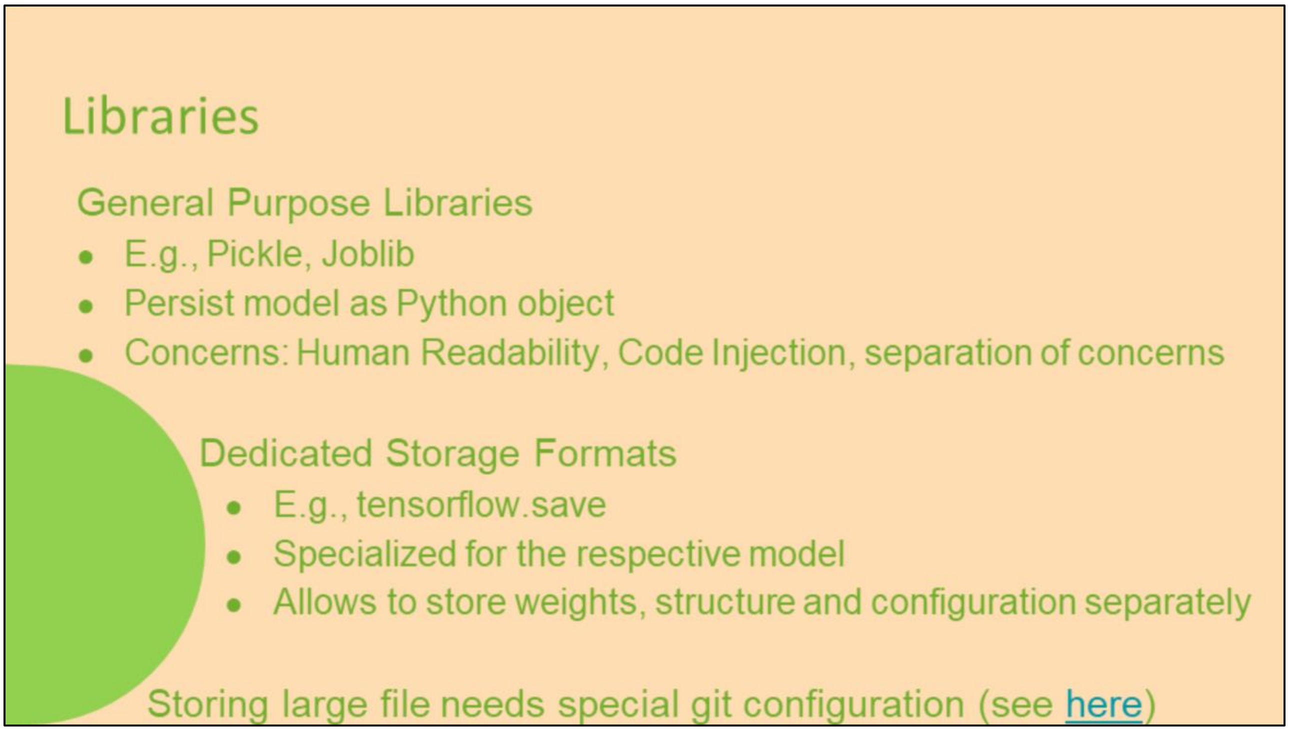
- E.g., optimizer, loss function

There are different parts you could store for a machine learning model.

The structure of a model is the first thing. If we are talking about neural networks, the structure consists of the layers, transfer functions, etc. loading the structure would enable someone to train the same model structure for a different problem. The training configuration is another thing you may store for the same reason – to enable someone to load them and set up a similar experiment.

The learned parameters of the model can also be stored. In case of neural networks those are the weights and biases that are learned as result of the training process. While these are specific to the model structure (models of different structures will have different numbers of parameters), it can sometimes make sense to store different weight sets for one particular model, e.g., if they have been learned on different data or represent snapshots of the learning progress that you may want to return to later.

Of course, sometimes you don’t need this kind of fine-grained control and mainly want to dump your model into a file to load it later. This can also be done.



Libraries

General Purpose Libraries
- E.g., Pickle, Joblib
- Persist model as Python object
- Concerns: Human Readability, Code Injection, separation of concerns

Dedicated Storage Formats
- E.g., tensorflow.save
- Specialized for the respective model
- Allows to store weights, structure and configuration separately

Storing large file needs special git configuration (see here)

In terms of libraries for storing models there are two general types of libraries.

On the one hand, there are **general purpose persistence libraries** such as Picke or Joblib. These libraries enable you to store Python objects into files and load them later. If the machine learning model is a Python object, these libraries usually work well, although they may require some additional setup (e.g., Pickle requires the right libraries to be loaded when restoring a model).

Using these general libraries is often easy and convenient. However, they also suffer some drawback, which may or may not become a concern to you:
1) Human Readability: the information may be stored in a file that may not be easily accessible for a human to read.
2) Code injection may become an issue. At least this is the case with Pickle. If you want to build a system that loads external models – e.g., ones uploaded by the user, then someone may be able to attack your system by causing malicious code to be injected while loading your model.
3) The libraries usually don't enable you to separate between the three concerns discussed on the previous side. However, it can be possible to separate the concerns on the programming level and then store the three parts separately.

Some machine learning libraries also provide **dedicated storage format**. For example, tensorflow provides dedicated storage functions, like the *save* function to store neural networks into files. These are usually more optimized to the machine learning use case and make it easier for you to separate the different concerns. It is also possible to build your own storage formats – e.g., a JSON format that stores the training setup and can be loaded. While this requires you to write your own store/load code, it can be worth it if you have an exact behavior in mind that is not provided by existing persistence libraries.

One thing you may run into when storing models is Gits 100 MB file limit. If this applies to you, you probably want to configure git to use the large files extension for such files. A link for how to do this can be found in the slide.

Questions?