- Our previous examples have assumed that we already had a labeled dataset to start from, and that we could immediately start training a model. In the real world, this is often not the case. You don't start from a dataset, you start from a problem
    - Imagine that you're starting your own machine learning consulting shop. You incorporate, you put up a fancy website, you notify your network. The projects start rolling in
        - A personalized photo search engine for picture-sharing social network – type in "wedding" and retrieve all the pictures you took at weddings, without any manual tagging needed
        - Flagging spam and offensive text content among the posts of a budding chat app
        - Building a music recommendation system for users of an online radio
        - Detecting credit card fraud for an e-commerce website
        - Predicting display ad click-through rate to decide which ad to serve to a given user at a given time
        - Flagging anomalous cookies on the conveyor belt of a cookie-manufacturing line.
        - Using satellite images to predict the location of as-yet unknown archeological sites.

- It would be very convenient if you could import the correct dataset from Kears and start fitting some deep learning models. Unfortunately, in the real world, you'll have to start from scratch

- In this chapter, you'll learn about a universal step-by-step blueprint that you can use to approach and solve any machine learning problem, like those in the previous list. This template will bring together and consolidate everything you've learned in chapter 4 and 5, and will given you the wider context that should anchor what you'll learn in the next chapters

- The universal workflow of machine learning is broadly structured in three parts
    - 1. Define the task – understand the problem domain and the business logic underlying what the customer asked for.
        - Collect a dataset, understand what the data represents, and choose how you will measure success on the task
    - 2. Develop a model
        - Prepare your data so that it can be processed by a machine learning model,
        - Select a model evaluation protocol and a simple baseline to beat,
        - Train a first model that has generalization power and that can overfit,
        - Regularize and tune your model until you achieve the best possible generalization performance
    - 3. Deploy the model – present your work to stakeholders, ship the model to a web server, a mobile app, a web page, or an embedded device, monitor the model's performance in the wild, and start collecting the data you'll need to build the next-generation model

1. Define the task

- You can't do good work without a deep understanding of the context of what you're doing.
  - Why is your customer trying to solve this particular problem?
  - What value will they derive from the solution – how will your model be used, and how will it fit into your customer's business processes?
  - What kind of data is available, or could be collected?
  - What kind of machine learning task can be mapped to the business problem?

1.1. Frame the problem

- Framing a machine learning problem usually involves many detailed discussions with stakeholders. Here are the questions that should be on the top of your mind
  - What will your input data be? What are trying to predict? You can only learn to predict something if you have training data available
    - For example, you can only learn to classify the sentiment of movie reviews if you have both movie reviews and sentiment annotations available.
    - As such, data availability is usually the limiting factor at this stage.
    - In many cases, you will have to resort to collecting and annotating new datasets yourself (which we'll cover in the next section)
  - What type of machine learning task are you facing?
    - Is it binary classification? Multiclass classification? Scalar regression? Vector regression? Multiclass multilabel classification? Image segmentation? Ranking? Something else, like clustering, generation, or reinforcement learning?
    - In some cases, it may be that machine learning isn't even the best way to make sense of the data, and you should use something else, such as plain old-school statistical analysis.
      - The photo search engine project is a multiclass, multilabel classification task.
      - The spam detection project is a binary classification task. If you set "offensive content" as a separate class, it's a three-way classification task
      - The music recommendation engine turns out to be better handled not via deep learning, but via **matrix factorization** (collaborative filtering)
      - The credit card fraud detection project is a binary classification task
      - The click-through-rate prediction project is a scalar regression task
      - Anomalous cookie detection is a binary classification task, but it will also require an object detection model as a first stage in order to correctly crop out the cookies in raw images. Note that the set of machine learning techniques known as "anomaly detection" would not be good fit in this setting!
      - The project for finding new archeological sites from satellite images is an image-similarity ranking task: you need to retrieve new images that look the most like known archeological sites
  - What do existing solutions look like?

        - Perhaps your customer already has a handcrafted algorithm that handles spam filtering or credit card fraud detection, with lots of *if* statement
        - Perhaps a human is currently in charge of manually handling the process under consideration – monitoring the conveyor belt at the cookie plant and manually removing the bad cookies, or crafting playlists of song recommendations to be sent out to users who liked a specific artist.
        - You should make sure you understand what systems are already in place and how they work
    - Are there particular constraints you will need to deal with?
        - For example, you could find out that the app for which you're building a spam detection system is strictly end-to-end encrypted, so that the spam detection model will have to live on the end user's phone and must be trained on an external dataset.
        - Perhaps the cookie-filtering model has such latency constraints that it will need to run on an embedded device at the factory rather than on a remote server.
        - You should understand the full context in which your work will fit

- Once you've done your research, you should know what your inputs will be, what your targets will be, and what broad type of machine learning task the problem maps to. Be aware of the hypotheses you're making at this stage:
    - You hypothesize that your targets can be predicted given your inputs
    - You hypothesize that the data's available (or that you will soon collect) is sufficiently informative to learn the relationship between inputs and targets

- Until you have a working model, these are merely hypotheses, waiting to be validated or invalidated.
    - Not all problems can be solved with machine learning; just because you've assembled examples of inputs X and targets Y doesn't mean X contains enough information to predict Y.

1.2. Collect a dataset

- Once you understand the nature of the task and you know what your inputs and targets are going to be, it's time for data collection – the most arduous, time-consuming, and costly part of most machine learning projects
    - E.g. The photo search engine project requires you to first select the set of labels you want to classify – you settle on 10,000 common image categories. Then you need to manually tag hundreds of thousands of your past user-uploaded images with labels from this set.

- As mentioned before, a model's ability to generalize comes almost entirely from the properties of the data it is trained on – the number of data points, the reliability of your labels, the quality of your features.
    - If you get an extra 50 hours to spend on a project, chances are that the most effective way to allocate them is to collect more data rather than search for incremental modelling improvements

- If you are doing supervised learning, then once you've collected inputs (such as images), you're going to need *annotations* for them (such as tags for those images) – the targets you will train your model to predict.

- **Beware of non-representative data**
  - ➢ Machine learning models can only make sense of inputs that are similar to what they've seen before. As such, it's critical that the data used for training should be *representative* of the production data. This concern should be the foundation of all your data collection work
    - ✧ Suppose you're developing an app where users take pictures of plate of food and find out the name of the dish
    - ✧ Suppose your test accuracy is 90% but received angry review from customer blaming the model predicted wrong thing 80% of the time
    - ✧ A quick look at user-uploaded data reveals that mobile picture uploads of random dishes from random restaurants taken with random smartphones look nothing like the professional-quality, well-lit, appetizing pictures you train the model on
    - ✧ Your training data wasn't representative of the production data.
  - ➢ If possible, collect data directly from the environment where your model will be used.
    - ✧ If it's not possible to train on production data, then make sure you fully understand how your training and production data (data used by users) differ, and that you are actively correcting for these differences.
  - ➢ A related phenomenon you should be aware of is ***concept drift***: when the properties of the production data change over time, causing model accuracy to gradually decay.
    - ✧ A music recommendation engine trained in the year 2013 may not be very effective today.
    - ✧ Likewise, the IMDB dataset you worked with was collected in 2011, and a model trained on it would likely not perform well on reviews from 2020, as vocabulary, expressions, and movie genres evolve over time.
    - ✧ Dealing with fast concept drift requires constant data collection, annotation, and model retraining.
  - ➢ Keep in mind that machine learning can only be used to memorize patterns that are present in your training data.
    - ✧ You can only recognize what you've seen before.
    - ✧ Using machine learning trained on past data to predict the future is making the assumption that the future will behave like the past. This often isn't the case
  - ➢ A particularly insidious and common case of non-representative data is *sampling bias*: measuring the average height of people from samples from elementary school (clearly not representative of all people)

1.3. Understand your data

- It's pretty bad practice to treat a dataset as a black box.
  - ➢ Before you start training models, you should explore and visualize your data to gain insights about what makes it predictive, which will inform feature engineering and screen for potential issues.
    - ✧ Image data → take a look at few samples
    - ✧ Numerical data → take a look at histograms, distribution of data

        - Location data → plot on a map
        - Classification problem → check data on each class. Class balanced?
        - Check for ***target leaking***: the presence of features in your data that provide information about the targets and which may not be available in production.
            - Always ask yourself, is every feature in your data something that will be available in the same form in production?

1.4. Choose a measure of success

- To achieve success on a project, you must first define what you mean by success. Your metric for success will guide all of the technical choices your make throughout the project. It should directly align with your higher-level goals, such as the business success of your customer
    - For balanced classification with balanced classes, accuracy and area under a *receiver operating characteristic* (ROC) curve are common metrics
    - For class-imbalanced problems, ranking problems, or multilabel classification, you can use precision and recall, as well as a weighted form of accuracy of ROC AUC.
    - It isn't uncommon to have to define your own custom metrics by which to measure success

2. Develop a model

- Once you know how you will measure your progress, you can get started with model development
    - Most tutorials and research projects assume that this is the only step – skipping problem definition and dataset collection, which are assumed already done, and skipping model deployment and maintenance, which are assumed to be handled by someone else.
    - In fact, model development is only one step in the machine learning workflow, and if you ask me, it's not the most difficult one.
        - The hardest things in machine learning are framing problems and collecting, annotating, and cleaning data.

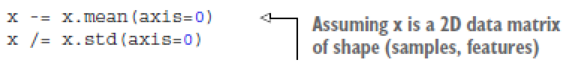
2.1. Prepare the data

- Data preprocessing aims at making the raw data at hand more amenable to neural networks.
    - This includes vectorization, normalization, or handling missing values
    - Many preprocessing techniques are domain-specific (e.g. specific to text data or image data)

- **Vectorization**
    - All inputs and targets in a neural network must typically be tensors of floating-point data (or, in specific cases, tensors of integers or strings).

- ➢ Whatever data you need to process – sound, images, text – you must first turn into tensors, a step called *data vectorization.*

- **Value normalization**
  - ➢ In general, it isn't safe to feed into a neural network data that takes relatively large values (for example, multi-digit integers, which are much larger than the initial values taken by the weights of a network) or data that is heterogeneous (for example, data where one feature is in the range 0-1 and another is in the range 100-200). Doing so can trigger large gradient updates that will prevent the network from converging.
  - ➢ To make learning easier for your network, your data should have the following characteristics
    - ✧ Take small values – typically, most values should be in the 0-1 range
    - ✧ Be homogenous – all features should take values in roughly the same range
  - ➢ Additionally, the following stricter normalization practice is common and can help, although it isn't always necessary (e.g. we didn't do this in the digit-classification example)
    - ✧ Normalize each feature independently to have a mean of 0
    - ✧ Normalize each feature independently to have a standard deviation of 1
  - ➢ In NumPy arrays



```
x -= x.mean(axis=0)
x /= x.std(axis=0)
```

Assuming x is a 2D data matrix of shape (samples, features)

- **Handling missing values**
  - ➢ What if there are feature values missing in the data? You could just discard the feature entirely, but you don't necessarily have to
    - ✧ If the feature is categorical, it's safe to create a new category that means "the value is missing" The model will automatically learn what this implies with respect to the targets
    - ✧ If the feature is numerical, avoid inputting an arbitrary value like "0", because it may create a discontinuity in the latent space formed by your features, making it harder for a model trained on it to generalize. Instead, consider replacing the missing value with the average or median value for the feature in the dataset. You could also train a model to predict the feature value given the values of other features.
  - ➢ Note that if you're expecting missing categorical features in the test data, but the network was trained on data without any missing values, the network won't have learned to ignore missing values!
    - ✧ In this situation, you should artificially generate training samples with missing entries: copy some training samples several times, and drop some of the categorical features that you expect are likely to be missing in the test data.

2.2. *Choose an evaluation protocol*

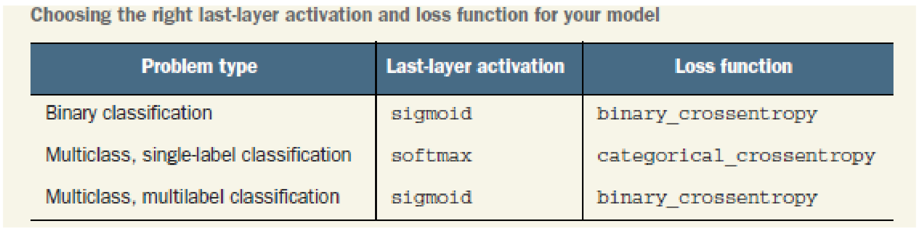
- Recall that the purpose of a model is to achieve generalization, and every modeling decision you will make throughout the model development process will be guided by ***validation metrics*** that seek to measure generalization performance.

- ➢ The goal of your validation protocol is to accurately estimate what your success metric of choice (such as accuracy) will be on actual production data. The reliability of that process is critical to building a useful model
- ➢ Recall that we reviewed three common evaluation protocol
    - ✧ Holdout validation – when have plenty of data
    - ✧ K-fold cross-validation – have too few samples for the first option to be reliable
    - ✧ Iterated K-fold validation – this is for performing highly accurate model evaluation when little data is available
- ➢ Pick one of these. Most of the time, the first will work well enough.
    - ✧ But always be mindful of the *representativity* of the validation set, and be careful not to have redundant samples between your training set and your validation set.

2.3. *Beat a baseline*

- As you start working on the model, your initial goal is to achieve *statistical power*: that is, to develop a small model that is capable of beating a simple baseline. At this stage, there are three most important things you should focus on
    - ➢ Feature engineering – filter out uninformative features (feature selection) and use your knowledge of the problem to develop new features that are likely to be useful.
    - ➢ Selecting the correct architecture priors – what type of model architecture will you use?
        - ✧ A densely connected network, a convnet, a recurrent neural network, a Transformer? Is deep learning even a good approach for the task, or should you use something else?
    - ➢ Selecting a good-enough training configuration
        - ✧ What loss function should you use? What batch size and learning rate?

- **Picking the right loss function**
    - ➢ It's often not possible to directly optimize for the metric that measures success on a problem
    - ➢ Sometimes there is no easy way to turn a metric into a loss function
        - ✧ Ideally, a loss function should be computable for as little as a single data point and must be differentiable.
        - ✧ For instance, the widely used classification metric ROC AUC can't be directly optimized. Hence, in classification tasks, it's common to optimize for a proxy metric of ROC AUC, such as crossentropy. In general, you can hope that the lower the crossentropy gets, the higher the ROC ACU will be.
    - ➢ The following table can help you choose a last-layer activation and a loss function for a few common problem types



Choosing the right last-layer activation and loss function for your model

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`

- For most problems, there are existing templates you can start from.
  - ➢ Make sure you research prior works to identify engineering techniques and model architectures that are most likely to perform well on your task.

- Note that it's not always possible to achieve statistical power.
  - ➢ If you can't beat a simple baseline after trying multiple reasonable architectures, it may be that the answer to the question you're asking isn't present in the input data, in which case you must go back to the drawing board

2.4. Scale up: develop a model that overfits

- Once you've obtained a model that has statistical power, the question becomes, is your model sufficiently powerful? Does it have enough layers and parameters to properly model the problem at hand?
  - ➢ To figure out how big a model you'll need, you must develop a model that overfits. This is fairly easy
    - ✧ 1. Add layers
    - ✧ 2. Make the layers bigger
    - ✧ 3. Train for more epochs
  - ➢ Always monitor the training loss and validation loss, as well as the training and validation values for any metrics you care about.
    - ✧ When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting

2.5. Regularize and tune your model

- Once you've achieved statistical power and you're able to overfit, you know you've on the right path. At this point, your goal becomes to maximize generalization performance
  - ➢ This phase will take the most time: you'll repeatedly modify your model, train it, evaluate on your validation data, modify it again, and repeat, until the model is as good as it can get. Here are some things you should try
    - ✧ Try different architectures; add or remove layers
    - ✧ Add dropout
    - ✧ If your model is small, add L1 and L2 regularization
    - ✧ Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration
    - ✧ Optionally, iterate on data curation or feature engineering: collect and annotate more data, develop better features, or remove features that don't seem to be informative

- It's possible to automate a large chunk of this work by using *automated hyperparameter tunning software*, such as KerasTuner, covered in chapter 13

        - Note that repeatedly using validation process will eventually cause your model to overfit to the validation process. This makes the evaluation process less reliable.

- Once you’ve developed a satisfactory model configuration, you can train your final production model on all the available data (training and validation) and evaluate it one last time on the test set.
    - If it turns out that performance on the test set is significantly worse than the performance measured on the validation data, this may mean either that your validation procedure wasn’t reliable after all, or that you began overfitting to the validation data while tuning the parameters of the model.
    - In this case, you may want to switch to a more reliable evaluation protocol (such as iterated K-fold validation)

3. Deploy the model

- Your model has successfully cleared its final evaluation on the test set – it’s ready to be deployed and to begin its productive life

3.1. Explain your work to stakeholders and set expectations

- Success and customer trust are about consistently meeting or exceeding people’s expectations. The actual system you deliver is only half of that picture; the other half is setting appropriate expectations before launch.
    - The expectations of non-specialists towards AI systems are often unrealistic
        - They might expect the system to understand its tasks and is capable of exercising human-like common sense in the context of the task
            - To address this, you should consider showing some examples of the *failure modes* of your model (for instance, show what incorrectly classified samples look like, especially those for which the misclassification seems surprising)
        - They might also expect human-level performance, especially for processes that were previously handled by people. Because most machines learning models are imperfectly trained to approximate human-generated labels, they do not nearly get there.
            - You should clearly convey model performance expectations.
            - Avoid using abstract statements like “the model has 98% accuracy”, and prefer taking, for instance, about false negative rates and false positive rates.
    - You should also make sure to discuss with stakeholders the choice of key launch parameters
        - For instance, the probability threshold at which a transaction should be flagged (different thresholds will produce different FN and FP rates). Such decision involves trade-offs that can only be handled with a deep understanding of the business context.

3.2. Ship an inference model

- A machine learning project doesn't end when you arrive at a Colab notebook that can save a trained model. You rarely put in production the exact same Python model object that you manipulated during training.
    - ➢ First, you may want to export your model to something other than Python
        - ✧ You production environment may not support Python at all – for instance, if it's a mobile app or an embedded system
        - ✧ If the rest of the app isn't in Python, the use of Python to serve a model may induce significant overhead
    - ➢ Secondly, since your production model will only be used to output predictions (a phase called *inference*), rather than for training, you have room to perform various optimizations that can make the model faster and reduce its memory footprint.

- **Deploying a model as a REST API**
    - ➢ This is perhaps the common way to turn a model into a product: install TensorFlow on a server or cloud instance, and query the model's predictions via a REST API.
        - ✧ You could build your own serving app using something like Flask (or any other Python web development library), or
        - ✧ Use TensorFlow's own library for shipping models as APIs, called *TensorFlow Serving* (https://www.tensorflow.org/tfx/guide/serving). With TensorFlow Serving, you can deploy a Keras model in minutes
    - ➢ You should use this deployment setup when
        - ✧ The application that will consume the model's prediction will have reliable access to the internet.
        - ✧ The application does not have strict latency requirements: the request, inference, and answer round trip will typically take around 500 ms.
        - ✧ The input data sent for inference is not highly sensitive: the data will need to be available on the server in a decrypted form, since it will need to be seen by the model (but note that you should use SSL encryption for the HTTP request and answer).
    - ➢ For instance, the image search engine project, the music recommender system, the credit card fraud detection project, and the satellite imagery project are all good fits for serving via a REST APIC
    - ➢ An important question when deploying a model as a REST API is whether you want to hose the code on your own, or whether you want to use a fully managed third-party cloud service
        - ✧ For instance, Cloud AI Platform, a Google product, lets you simply upload your TensorFlow model to Google Cloud Storage (CGS), and it gives you an API endpoint to query it. It takes care of many practical details such as bathing predictions, load balancing, and scaling

- **Deploying a model on a device**
    - ➢ Sometimes, you may need your model to live on the same device that runs the application that uses it – maybe a smartphone, an embedded ARM CPU on a robot, or a microcontroller on a tiny device.
        - ✧ You may have seen a camera capable of automatically detecting people and faces in the scenes you pointed it at: that was probably a small deep learning model running directly on the camera

- You should use this setup when
  - Your model has strict latency constraints or needs to run in a low-connectivity environment. If you're building an immersive augmented reality application, querying a remote server is not a viable option.
  - Your model can be made sufficiently small that it can run under the memory and power constraints of the target device. You can use the TensorFlow Model Optimization Toolkit to help with this (https://www.tensorflow.org/model_optimization)
  - Getting the highest possible accuracy isn't mission critical for your task. There is always a trade-off between runtime efficiency and accuracy, so memory and power constraints often require you to ship a model that isn't quite as good as the best model you could run on a large GPU
  - The input data is strictly sensitive and thus shouldn't be decryptable on a remote server
- Our spam detection model will need to run on the end user's smartphone as part of the chat app, because messages are end-to-end encrypted and thus cannot be read by a remotely hosted model
- To deploy a Keras model on a smartphone or embedded device, your go-to solution is TensorFlow Lite (https://www.tensorflow.org/lite).
  - It's a framework for efficient on-device deep learning inference that runs on Android and iOS smartphones, as well as ARM64-based computers, Raspberry Pi, or certain microcontrollers.
  - It includes a converter that can straightforwardly turn your Keras model into the TensorFlow Lite format

- **Deploying a model in the browser**
  - Deep learning is often used in browser-based or desktop-based JavaScript applications. While it is usually possible to have the application query a remote model via a REST API, there can be key advantages in having the model run directly in the browser, on the user's computer (utilizing GPU resources if they're available)
  - Use this setup when
    - You want to offload compute to the end user, which can dramatically reduce server costs
    - The input data needs to stay on the end user's computer or phone.
      - For instance, in our spam detection project, the web version and the desktop version of the chat app (implemented as a cross-platform app written in JavaScript) should use a locally run model
    - Your application has strict latency constraint.
      - While a model running on the end user's laptop or smartphone is likely to be slower than one running on a large GPU on your own server, you don't have the extra 100 ms of network round trip.
    - You need your app to keep working without connectivity, after the model has been downloaded and cached
  - You should only go with this option if your model is small enough that it won't hog the CPU, GPU, or RAM of your user's laptop or smartphone.
  - In addition, since the entire model will be downloaded to the user's device, you should make sure that nothing about the model needs to stay confidential.

    - ✧ Be mindful of the fact that, given a trained deep learning model, it is usually possible to recover some information about the training data: better not to make your trained model public if it was trained on sensitive data
  - ➢ To deploy a model in JavaScript, the TensorFlow ecosystem includes TensorFlow.js (https://www.tensorflow.org/js), a JavaScript library for deep learning that implements almost all of the Keras API (original developed under the working name WebKeras) as well as many lower-level TensorFlow APIs.
    - ✧ You can easily import a saved Keras model into TensorFlow.js to query it as part of your browser-based JavaScript app or your desktop Electron app.

- **Inference model optimization**
  - ➢ Optimizing your model for inference is especially important when deploying in an environment with strict constraints on available power and memory (smartphones and embedded devices) or for applications with low latency requirements.
  - ➢ You should always seek to optimize your model before importing into TensorFlow.js or exporting it to TensorFlow Lite
  - ➢ There are two popular optimization techniques
    - ✧ Weight pruning: not every coefficient in a weight tensor contributes equally to predictions.
      - ▫ It is possible to considerably lower the number of parameters in the layers of your model by only keeping the most significant ones.
      - ▫ This reduces the memory and compute footprint of your model, at a small cost in performance metrics.
      - ▫ By deciding how much pruning you want to apply, you are in control of the tradeoff between size and accuracy
    - ✧ Weight quantization: Deep learning models are trained with single-precision floating-point (*float32*) weights.
      - ▫ However, it's possible to *quantize* weights to 8-bit signed integers (*int8*) to get an inference-only model that's a quarter the size but remains near the accuracy of the original model
  - ➢ The TensorFlow ecosystem includes a weight pruning and quantization toolkit (https://www.tensorflow.org/model_optimization) that is deeply integrated with the Keras API

3.3. Monitor your model in the wild

- You've exported an inference model, you've integrated it into your application, and you've done a dry run on production data – the model behaved exactly as you expected.
  - ➢ Even this is not the end. Once you're deployed a model, you need to keep monitoring its behavior, its performance on new data, its interaction with the rest of the application, and its eventual impact on business metrics
    - ✧ If possible, do a regular manual audit of the model's predictions on production data. It's generally possible to reuse the same infrastructure as for data annotation: send some fraction

of the production data to be manually annotated, and compare the model's predictions to the new annotations.

    - For instance, you should definitely do this for the image search engine and bad-cookie flagging system

  - When manual audits are impossible, consider alternative evaluation avenues such as user surveys (for example, in the case of the spam and offensive-content flagging system)

3.4. Maintain your model

- Lastly, no model lasts forever. You've already learned about *concept drift*: over time, the characteristics of your production data will change, gradually degrading the performance and relevance of your model.
  - The lifespan of your music recommender system will be counted in weeks.
  - For the credit card fraud detection systems, it will be days
  - A couple of years in the best case for the image search engine
- As soon as your model has launched, you should be getting ready to train the next generation that will replace it. As such
  - Watch out for changes in the production data. Are new features becoming available? Should you expand or otherwise edit the label set?
  - Keep collecting and annotating data, and keep improving your annotation pipeline over time. In particular, you should pay special attention to collecting samples that seem to be difficult for your current model to classify – such samples are the most likely to help improve performance

Summary

- When you take on a new machine learning project, first define the problem at hand
  - Understand the broader context of what you're setting out to do – what's the end goal and what are the constraints?
  - Collect and annotate a dataset; make sure you understand your data in depth
  - Choose how you'll measure success for your problem – what metrics will you monitor on your validation data?

- Once you understand the problem and you have an appropriate dataset, develop a model
  - Prepare your data
  - Pick your evaluation protocol: holdout validation? K-fold validation? Which portion of the data should you use for validation?
  - Achieve statistical power: beat a simple baseline
  - Scale up: develop a model that can overfit
  - Regularize your model and tune its hyperparameters, based on performance on the validation data.

        - ✧ A lot of machine learning research tends to focus only on this step, but keep the big picture in mind

- When your model is ready and yields good performance on the test data, it's time for deployment
    - ➢ First, make sure you set appropriate expectations with stakeholders
    - ➢ Optimize a final model for inference, and ship a model to the deployment environment of choice – web server, mobile, browser, embedded device, etc.
    - ➢ Monitor your model's performance in production, and keep collecting data so you can develop the next generation of the model