



THE MACHINE LEARNING CANVAS

*A handbook for innovators and
visionary managers striving to design
tomorrow's Machine Learning systems*

LOUIS DORARD, PH.D.

“A very neat way think about your approach to embedding Machine Learning into your business.”

—*Samir Sharma, CEO at datazuum ([source](#))*

“To dissect the use cases, we recommend the approach developed by Louis Dorard [...] the Machine Learning Canvas.”

—*Carlos Escapa, Global AI/ML Practice Lead at AWS ([source](#))*

“Would you like to be able to show your Machine Learning model in only one page? Some weeks ago I wanted to do exactly this and after a couple of hours I end up with a 10-slide presentation. I wasn't happy, I wanted something simple but not simpler. Something like the business model canvas and after some research bam! There it was, the Machine Learning Canvas, by Louis Dorard. Great job mate! I love it.”

—*Christian Willig, Strategic Analytics Specialist at Energy Queensland ([source](#))*

“By far the best framework I've seen to help clients discover uses for their data, and to keep everyone focused on the same goal.”

—*Diego Ventura, Customer Success at MonkeyLearn, Inc*

“Now to complete the loop, I introduce the Machine Learning Canvas. [...] Louis has made his canvas freely available, and I will do likewise with the additions that we made to his canvas based upon our unique data science requirements.”

—*Bill Schmarzo, CTO, IoT & Analytics at Hitachi Vantara ([source](#))*

“Great framework for discussing Machine Learning problems between all stakeholders and keeping everyone on the same page.”

—Nicolas Schwartz, Tech Lead at BlaBlaCar

“The Machine Learning Canvas is providing our clients real business value by supplying the first critical entry point for their implementation of predictive applications. It has supported numerous use cases across industries.”

—Ingolf Mollat, Principal Consultant at Blue Yonder

“It’s a great communication tool between domain experts, data scientists and developers. And experience teaches us how important communication is, especially in complex projects.”

—Amir Tabakovic, VP Business Development at BigML ([source](#))

“The Machine Learning Canvas proved to be useful for developing and structuring even the most complex AI projects. I’d like to wind up by thanking Louis Dorard for his excellent thinking behind ML project principles.”

—Irina Peregud, InData Labs ([source](#))

“The Machine Learning Canvas plays a crucial role in the solution design process. It’s a remarkably simple technique for clearly defining the objectives, scope and outcomes of a project.”

—Scyfer B.V. ([source](#))

The Machine Learning Canvas

Written by Louis Dorard, PhD

Copyright © 2019 — All rights reserved

DRAFT version 0.1 released on 12 February 2019

Download and get updates at machinelearningcanvas.com











The Machine Learning Canvas

Designed for:

Designed by:

Date:

Iteration:

<div>Decisions</div> <div>How are predictions used to make decisions that provide the proposed value to the end-user?</div> <div></div>	<div>ML task</div> <div>Input, output to predict, type of problem.</div> <div></div>	<div>Value Propositions</div> <div>What are we trying to do for the end-user(s) of the predictive system? What objectives are we serving?</div> <div></div>	<div>Data Sources</div> <div>Which raw data sources can we use (internal and external)?</div> <div></div>	<div>Collecting Data</div> <div>How do we get new data to learn from (inputs and outputs)?</div> <div></div>
<div>Making Predictions</div> <div>When do we make predictions on new inputs? How long do we have to featurize a new input and make a prediction?</div> <div></div>	<div>Offline Evaluation</div> <div>Methods and metrics to evaluate the system before deployment.</div> <div></div>		<div>Features</div> <div>Input representations extracted from raw data sources.</div> <div></div>	<div>Building Models</div> <div>When do we create/update models with new training data? How long do we have to featurize training inputs and create a model?</div> <div></div>
	<div>Live Evaluation and Monitoring</div> <div>Methods and metrics to evaluate the system after deployment, and to quantify value creation.</div> <div></div>			

Contents

I. Introduction	7
Delivering value to end-users with Machine Learning	8
A canvas to structure Machine Learning systems	11
Who is the Machine Learning Canvas for?	15
II. Examples	17
Priority inbox	20
Fake reviews detection	24
Real-estate deals	28
Customer retention	32
III. The LEARN part	36
Data sources	37
Collecting data	38
Features	41
Building models	44
IV. The PREDICT part.....	48
ML task	49
Decisions	52
Making predictions	61
Offline evaluation	66
V. The EVALUATE part	76
Live evaluation and monitoring	77
About the author	78
Copyright, creative commons, and a disclaimer	79

I. Introduction

Delivering value to end-users with Machine Learning

Machine Learning (ML) systems are complex. At their core, they ingest data in a certain format, they learn from it, and they build models that are able to predict the future. A famous example in the industry is identifying fragile customers, who may stop being customers within a certain number of days (the “churn” problem). These predictions only become valuable when they are used to inform or to automate decisions (e.g. which promotional offers to give to which customers, to make them stay).

When we’re starting out on a new Machine Learning project, there’s a temptation to just grab a dataset, apply various algorithms, compute generic performance metrics, then get the value up in all sorts of ways (data pre-processing, hyper-parameter tuning, model ensembling), even when these metrics don’t have an interpretation in the application domain that makes sense. We’re so focused on these things that we end up losing track of the big picture...

If ML is to solve real problems, it means that there has to be end-users that ML systems are useful to. Another way to phrase things is that an ML system must make decisions from predictions, in a way that creates value for an end-user. Knowing them should be our first job. Our second job would be to

formulate a value proposition for that end-user, that can be translated into a Machine Learning task. We would then think of a way to monitor value creation once the system is deployed in the real world.

“Product people build things that serve some purpose and solve some problem. Unfortunately, when it comes to machine learning, most are forgetting this.”

— Chris Butler (*Empathy Mapping for AI*)

This raises another question, which should probably be addressed before starting any implementation work: how should we evaluate the system before making the decision to deploy it and to impact real people? This would be based on potential gains of using the system for the end-users, but also on costs of building and running the system. These costs would have to be estimated based on operating constraints on the serving of predictions and on the creation of models.

Today’s tools are making it increasingly easy to run learning algorithms and to create predictive models (sometimes it’s completely automated). The remaining difficulties when integrating ML in real-world applications/products are in everything that happens before and after the learning of these models, such as collecting data, preparing it into *features*, and making the machine actually *do* something based on predictions (as opposed to just showing predictions to a human).

All of the questions and activities we mentioned require a diverse skillset; they involve many different roles, from science to business. However, in many organizations there is a disconnect between the people who are able to build accurate predictive models, and those who know how to best serve the organization's objectives. It's also not uncommon to realize that time is being spent by engineers to solve the wrong problems, and to build systems that don't get used. And when they do work on the right problems, it's still a challenge to align everyone's activities. This is a general problem in any endeavor where people of different backgrounds need to team up to build something innovative that creates value.

One way to make things easier is to use a *canvas*...

A canvas to structure Machine Learning systems

A “canvas” is just a visual chart to describe a complex object, in a better way than a simple text document. Each key aspect of that object has its own “block”, and blocks are arranged on the chart in a way that makes visual sense.

Canvases are very popular in the startup community, starting with the Business Model Canvas. It provides an overview of this complex object that a Business Model is, and facilitates collaboration. Canvases have also been used for completely different purposes, with different layouts/structures (e.g. the Culture Creation Canvas and the Mobile Stickiness Canvas).

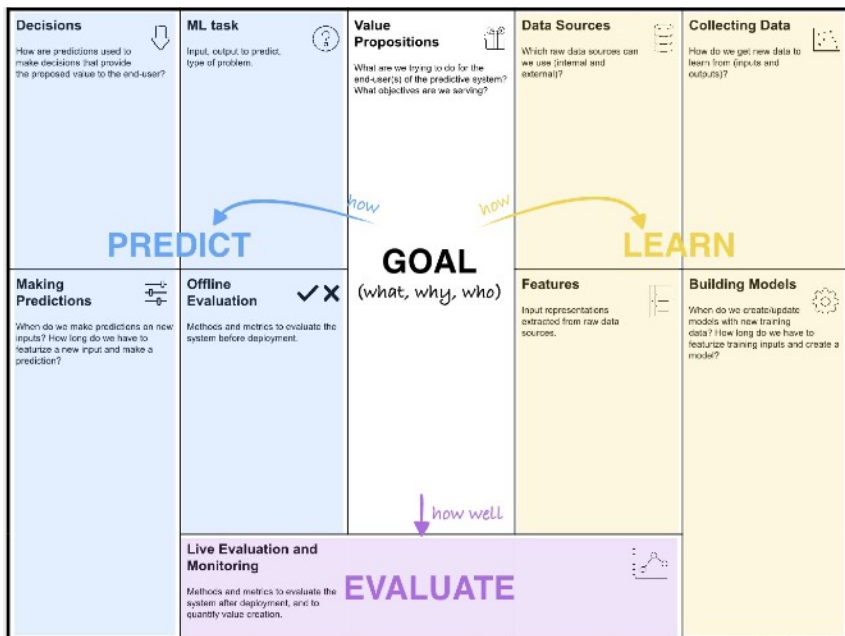
In the context of data and Artificial Intelligence, a canvas can be useful to describe the actual learning that takes place in intelligent systems:

- What data are we learning from
- How are we using predictions powered by that learning
- How are we making sure that the whole thing “works” through time?

The Machine Learning Canvas (hereafter “the Canvas”) allows to describe precisely this. It allows a team to lay down its vision for

an ML system and to communicate it. It's the first step towards making sure you connect what ML can do to your organization's objectives, and towards assessing feasibility. It should be filled in before starting any implementation work, and even before Exploratory Data Analysis.

The Canvas is what will bring us the closest to implementation, with the confidence that we are solving the right problem, and that we are choosing the right tools.



It starts with a central block dedicated to the *Value Proposition(s)* of the system where ML is going to be used. You can think of it as the *What+Why+Who*: *What* are we trying to do,

Why is it important, and *Who* is going to use the system and be impacted by it (i.e. who is the end-user). Then there's the *How*, which can be split in two parts:

- **LEARN** — The part on the right-hand side is dedicated to Learning from data. It's made of the following blocks:
 - **Data sources:** Which raw data sources can we use?
 - **Collecting data:** How do we get new data to learn from (inputs AND outputs)?
 - **Features:** Input representations to extract from raw data sources.
 - **Building models:** When do we create/update models with new training data and how long do we have for that?
- **PREDICT** — The part on the left-hand side is dedicated to Predictions, based on the models that we'll learn from data. It's made of the following blocks:
 - **ML task:** Which type (e.g. classification, regression...), what is the input, and what is the output to predict (along with possible values)?
 - **Decisions:** How are predictions used to make decisions that provide the proposed value to the end user?

- **Making predictions:** When do we make predictions on new inputs and how long do we have for that?
- **Offline evaluation:** Which methods and metrics can we use to evaluate the way predictions are going to be made and used, prior to deployment?

The last part of the Canvas (**EVALUATE**) is dedicated to measuring how well the system works and to monitoring value creation. This is where you'll specify methods and metrics to evaluate the system after deployment, and to quantify value creation.

There's a couple more things to notice about the structure of the Canvas. The top provides more of a background view, while the bottom goes into the specifics of the system. The upper left and right blocks relate to domain integration: how predictions are used and how data is collected in the domain of application. The lower left and right blocks relate to the "predictive engine" and its constraints, such as latency for making predictions and updating models.

Who is the Machine Learning Canvas for?

The creation of artificially intelligent systems that learn from data relies on different roles: data scientists, software engineers, product designers, technical and business managers. The Machine Learning Canvas allows to keep everyone on the same page and helps align their activities. The main users of the Canvas would be operational managers, technical managers, product and project managers, who provide the glue between everyone — but all are encouraged to read about it, how to use it, and to give it a go!

Head over to machinelearningcanvas.com for information on how to access the Machine Learning Canvas on Google Docs, and how to download it as Word, PDF, or other formats.

I've used the Canvas for a few years while consulting for various clients. I created the first version in 2015 and kept refining it based on practice and feedback from fellow ML practitioners. The Canvas has been used at AI Startup Weekends and featured in multiple courses, such as Persontyle's School of Data Science, Blue Yonder's Data Science Academy, and UCL Engineering's Machine Learning Academy. It is now being taught at UCL School of Management.

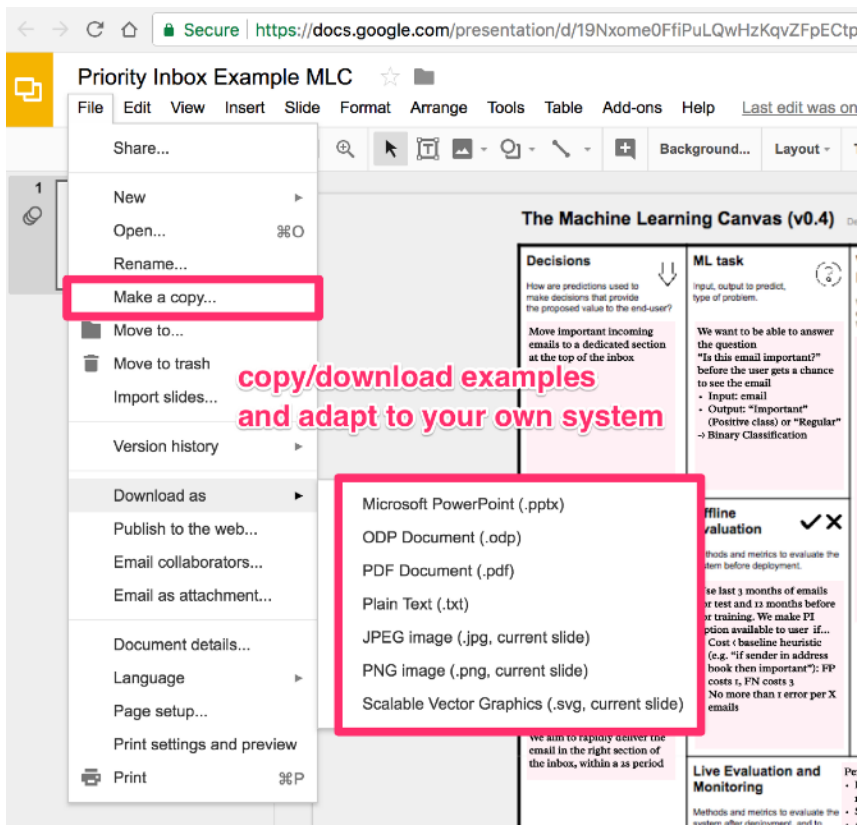
This book will teach you the basics of the Machine Learning Canvas. It assumes prior knowledge of the core concepts of

supervised learning. (You could apply the Canvas to other types of learning problems, but for the sake of simplicity we will be limiting ourselves to classification and regression problems.)

If you're new to the area and don't have the required prior knowledge, check out my other book [Bootstrapping Machine Learning](#) for an easy-to-read introduction (someone once told me they read it in half a day at the beach 🏖️).

II. Examples

I'd like to start by introducing four example Machine Learning Canvases that all have their own specificities. Because of the vertical format of this book, I am presenting the contents of each block of the Canvas in a linear order, but you can find the original Canvases by following the links below. They'll point you to Google Slides documents that you can view / copy / download.



- Priority inbox:
- End-user: user of an email client

- Reply to important messages faster and spend less time on emails
- [Fake reviews detection:](#)
 - End-user: hotel booking platform
 - Improve hotel satisfaction and user experience of the platform with more accurate ratings
- [Real-estate deals:](#)
 - End-user: property investor
 - Make better investments by prioritizing the best deals among all properties currently on the market
- [Customer retention:](#)
 - End-user: Customer Relationship Management team
 - Reduce churn rate among high-value customers and improve success rate of retention efforts.

We'll comment on these example Canvases in the rest of the book, as they'll be used to look in more detail at how you're expected to use the Canvas.

Priority inbox

Value proposition

Make it easier for users of an email client to identify new important emails in their inbox, by automatically detecting them and making them more visible in the inbox (this detection must happen before user sees email)

The objective is that users spend less time in their inbox and reply to important emails more quickly

ML task

We want to be able to answer the question *“Is this email important?”* before the user gets a chance to see the email.

- Input: email
- Output: “Important” (Positive class) or “Regular”
⇒ Binary Classification

Data sources

- Previous email messages (as mbox files or in other type of database)

- Address book
- Calendar

Collecting data

Explicit labelling: users can manually label emails as important or not, by clicking on an icon next to each email's subject

Implicit labelling: heuristics based on user behavior after getting the email (e.g. replying fast, deleting without reading, etc.)

Decisions

Move incoming emails that have an importance score above a certain threshold, to a dedicated section at the top of the inbox

Making predictions

Every time we receive an email addressed to our user, which starts a new thread (otherwise the importance is just the same as that of the thread)

We aim to rapidly deliver the email in the right section of the inbox, within a 2 second period

Building models

One model per user, initially built on last 12 months of email data, that we update...

- When an error is signaled by the user via manual labelling
- Every 5' by adding new data from implicit labelling, if any

Features

- Content features: subject, body, attachments, size
- Social features: based on info about sender (e.g. in address book?), previous interactions, contextual (e.g. upcoming meeting w. sender)
- Email labels (typically assigned via manual rules defined by user)

Offline evaluation

Use last 3 months of emails for test and 12 months before for training. We make PI option available to user if...

- Cost < baseline heuristic (e.g. "if sender in address book then important"): FP costs 1, FN costs 3

- No more than 1 error per X emails

Live evaluation and monitoring

Per week...

- Ratio: #errors explicitly signaled by user / #emails received
- Same w. errors seen via implicit labelling
- Average time taken to reply to important emails
- Total time spent on inbox

Fake reviews detection

Value proposition

Reject fake incoming reviews and approve legit reviews automatically.

Flag fake reviews in database to stop displaying them / using them to compute average ratings. Have ratings which are closer to the truth. Improve customer experience and satisfaction (less surprises).

ML task

“Is this review legit or fake?”

- Input: review
- Output: “legit” or “fake” (Positive class)
⇒ Binary classification

Note: the distribution of outputs is typically 70-30 (legit vs fake)

Data sources

- User database

- Reviews database
- Social networks
- Crowdsourcing platform (e.g. Mechanical Turk)

Collecting data

- Initially: active learning using crowdsourcing platform
- Continuously: manual labelling, done internally
 - When explicitly requested (complaint, or model's probability in between thresholds)
 - Randomly selected reviews every day (as many as allowed for a budget of \$X / day)

Decisions

If probability of Positive...

> M: approve

< m: reject

Otherwise request human decision.

Thresholds m and M are chosen to maximize offline evaluation (performed right after model update)

Making predictions

We receive X reviews / minute on average. We can allow a delay of 1 day / review, but including 1/2 day for manual review if we're in between thresholds.

Building models

One model per language/ country

Somewhat adversarial setting...

⇒ Keep on learning

⇒ Every week we update our models by adding all the data from last week. We allow a day for this.

Features

- Content of review: rating, text, length, # capitals...
- Other predictions: sentiment, emotion, etc.
- User: basic info, # previous bookings, # approved reviews, # rejected reviews
- Metadata (e.g. IP)
- Product being reviewed (e.g. hotel chain)

- Similarity with previous reviews (total score)

Offline evaluation

Train model with data up until 1 week ago. Compute total cost on last week's data, for different values of m and M (starting at $m=0$ and $M=1$), taking into account:

- Gain of correct, automated decision = - Cost of manual decision
- Cost of FN (when review sentiment positive / negative)
- Cost of FP (smaller)

Live evaluation and monitoring

Every week:

- Average customer satisfaction
- # customer complaints
- # hotel complaints
- # manual reviews

Real-estate deals

Value proposition

Make better real-estate investments: compare price predictions with actual asking price of properties on the market, to find the best deals.

ML task

« How much is this property worth? »

- Input: property
- Output: value
⇒ regression task

OR

« Is this a good deal? »

⇒ classification task

Data sources

- Redfin
- Open data: public transports, schools, etc.

- Google Maps

Collecting data

Every week request Redfin data on:

- New properties on the market. Should contain property characteristics + asking price
- Sale records (initially: records for the past year). Should contain properties previously seen, but this time with actual sale price.

Decisions

Every week:

- Compute predictions for all houses currently on the market
- Filter out 50% randomly (hold-out set)
- Filter out properties where asking price is higher
- Prioritize best deals first and schedule visits
- Review manually and buy at asking price or lower

Making predictions

Every week we make predictions for new properties for sale (using all property info available except asking price).

Building models

Only keep data up until a year in the past.

Update model every month (with new data available).

Features

- Basic property info
- Extracted from text description: has swimming pool, has garage, etc.
- Location-based:
 - Latitude, longitude
 - Address
 - Distance to closest transports and shops
 - Average rating of schools in 5 mile radius

Offline evaluation

Test on the last month of labelled data, manually review errors and compute...

- Average percentage error
- Cost: for bad deals (sale price < asking) that were seen as good deals (asking < prediction), we would have incurred a cost of (asking - sale price) in case we would have gone through with the investment.

Live evaluation and monitoring

- Investment return (should go up)
- Time spent visiting properties (should go down as we're smarter about which we want to visit)
- Sale price compared to prediction, on hold-out set

Customer retention

Value proposition

- Context:
 - Company sells SaaS with monthly subscription
 - End-user of predictive system is CRM team
- We want to help them...
 - Identify important clients who may churn, so appropriate action can be taken
 - Reduce churn rate among high-revenue customers
 - Improve success rate of retention efforts by understanding why customers may churn

ML task

Predict answer to “Is this customer going to churn in the coming month?”

- Input: customer

- Output: 'churn' or 'no-churn' class ('churn' is the Positive class)
⇒ Binary Classification

Data sources

- CRM tool
- Payments database
- Website analytics
- Customer support
- Emailing to customers

Collecting data

Every month, we see which of last month's customers churned or not, by looking through the payments database.

Associated inputs are customer "snapshots" taken last month.

Decisions

On 1st day of every month:

- Randomly filter out 50% of customers (hold-out set)

- Filter out 'no-churn'
- Sort remaining by descending (churn prob.) x (monthly revenue) and show prediction path for each
- Target as many customers as suggested by simulation

Making predictions

Every month we (re-)featurize all current customers and make predictions for them.

We do this overnight (along with building the model that powers these predictions and evaluating it).

Building models

Every month we create a new model from the previous month's hold-out set (or the whole set, when initializing this system).

We do this overnight (along with offline evaluation and making predictions).

Features

Basic customer info at time t : age, city, etc.

Events between $(t - 1 \text{ month})$ and (t) :

- Usage of product: # times logged in, functionalities used, etc.
- Customer support interactions
- Other contextual, e.g. devices used

Offline evaluation

Before targeting customers:

- Evaluate new model's accuracy on pre-defined customer profiles
- Simulate decisions taken on last month's customers (using model learnt from customers 2 months ago). Compute ROI w. different # customers to target & hypotheses on retention success rate (is it >0 ?)

Live evaluation and monitoring

- Accuracy of last month's predictions on hold-out set
- Compare churn rate & lost revenue between last month's hold-out set and remaining set
- Monitor ($\# \text{non-churn among targeted}$) / $\# \text{targets}$
- Monitor ROI (based on difference in lost revenue & cost of retention campaign)

III. The LEARN part

Data sources

“Which raw data sources can we use?”

This step is preliminary to thinking about the actual data to be fed to ML algorithms, which will be extracted from sources listed here. You won’t have to think just yet about what exactly to extract, but it’s useful to indicate the type of each source, as it can hint at constraints associated to it (e.g. latency, volume of requests, throughput). For instance, you could be using different types of **internal/external databases, APIs, static files, web scraping**, etc., as data sources. Some of these sources could even be the outputs of other Machine Learning systems—it can be very useful to clearly identify those types of dependencies.

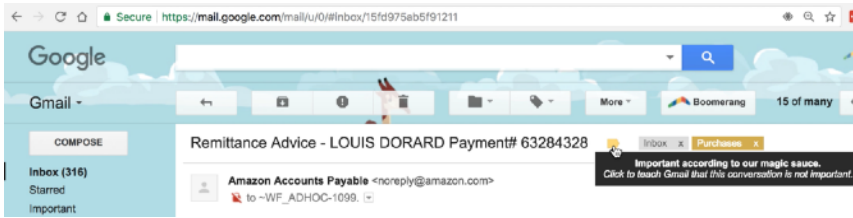
In the fake reviews example, the [MTurk](#) or [CrowdFlower](#) API can be used as a source to get output data. In the churn example, I’ve seen cases where the data sources could be a mix of tools—one for Customer Relationship Management, online payments, web analytics, emailing... You would have to be able to get data from all of these by connecting to their APIs, and then to merge that data.

Collecting data

“How do we get new data to learn from (inputs and outputs)?”

The question above mentions inputs and outputs, but often times it is **getting example outputs which can be a barrier to using ML**, so this is what you should focus on! There can be a cost associated to it, for instance in the fake reviews example where you need humans to manually look at example reviews and assign them a *fake* or *real* label (typically via a human-powered API like that of MTurk or CrowdFlower, which have straightforward cost structures that allow you to anticipate how much data you can afford).

In the priority inbox example, users could provide output data by explicitly saying that an email is important or not, via the email client interface. This is probably going to be “small” data, but it can be very important to have mechanisms to get that user feedback (i.e. design the interface accordingly) and to identify that output data as **explicit feedback**, as we’ll illustrate later in [*Building models*](#).



Designing for ML: Gmail allows you to tell it when it got your priorities wrong. A few years ago, it would also explain why it predicted an email was important. Now it's just magic sauce.

Sometimes there are ways to **be smart and get lots of output data “implicitly”**. We can figure out whether an email was important or not by using simple heuristics *after* users have had the chance to interact with the email; for instance, if they deleted without reading, then the email wasn't important. **Data augmentation or data generation** techniques can also be used in certain domains (e.g. [image data](#) or [natural language data](#)).

In other examples (such as real-estate deals), we'd **just need to wait** (for properties to sell) to get output data for free. When inputs and outputs are not collected at the same time, it can be useful to refine the description of the input that the output corresponds to: in the churn example, we just need to wait for a month to know if a customer churned or not, and the corresponding input is the customer “snapshot” taken a month before.

When setting out to build an ML system, our main concern is to get an initial dataset to learn a first model, so we can build a

Proof of Concept. I recommend to also think about **collecting data in the longer term** as soon as possible: how we'll continue to get new data, when we'll get it and how much we'll get (based on how nature works, how open your wallet is, etc.).

Note that *Collecting data* is placed at the top right of the Canvas, because *Decisions* is at the top left, and both are related to "domain integration": we measure what's going on in the domain with data, we use this data to build internal models that power predictions, and those predictions are the base of decisions/actions made in the domain. This can lead to **feedback loops**, which I'll discuss again in the section on [*Decisions*](#). One example of this is churn, where we may correctly predict that someone would churn, use that prediction to do something to prevent them from churning, and thus observe the following month that they didn't churn...

Finally, keep in mind that every data collection process is biased. Thinking about how your ML system will integrate with the domain helps **make the bias more explicit**, and it helps verify that the **inputs you collect for learning are representative** of the inputs on which you'll have to make predictions.

Features

“Input representations extracted from raw data sources.”

You want to make predictions on objects: a house, an email, a review, a customer. These are the *inputs* to the prediction problems. These inputs have to have a computer representation with numerical, categorical, or textual values. Those “feature” values must be chosen in a way that allows to characterize inputs well enough so that the outputs (house value, email importance, review fakeness, customer fragility) could be determined from the features. If there are many features, try to group them in **feature families** (see priority inbox example, with “content features” and “social features”). This can be useful for presentation purposes, but also later on for optimizing models.

address	bedrooms	bathrooms	size_sqft	lot_size	price
7103 Wolf Rivers Ave Las Vegas NV 89131	4	4	3,811	13,939	495,000
10669 Oak Crest Ave Las Vegas NV 89144	3	2	1,622	5,662	240,000
128 Celia Pl Las Vegas NV 89145	4	2	1,895	6,534	165,000
517 Carpenter Dr Las Vegas NV 89107	4	2	1,286	6,534	120,000

Sample real-estate data from realtor.com. Each row corresponds to a property, each column to a feature, except for the last column which is the output. Is it reasonable to say that ‘price’ could be determined from everything else?

You should aim to **specify** which features you want to consider well enough **for a data engineer to extract them from the data sources** you listed previously. Let's say that in the real-estate example you want "proximity to public transports" as a feature. What would you need to determine its value for a given property? Probably open data can help. Is it listed in data sources? How would you access it (e.g. Excel file or API)? You also need to know the geo-location of the property: do you already have it in the sources, or do you just have an address—in which case you would need to add to your sources a geocoding service? But to start with, what exactly is "proximity"? Is it binary? Is it a number of close-by public transport stations? Is it a distance to the closest one?

Often times you'll use features that are aggregates of raw data, and you should think about **what exactly the aggregation is over**. For instance, for spacial data you should determine a radius. For temporal data you should specify a period of time; for instance, one way to represent a customer could be with the number of times they used the service *in the last X months*. It's useful to highlight such parameters as they can impact the performance of the whole system (it's not just about the parameters of ML algorithms).

To a certain extent, all data is "temporal". The balance that a customer has in their account changes through time, but even their age or their country could change. When you're using that sort of information, you need to specify **at which point in time to**

extract the value. This goes back to the “snapshot” idea that I mentioned earlier.

In many ML systems, most of the complexity—and the whole feasibility of the system—is linked to features, so it’s worth thinking this through!

Building models

“When do we create/update models with new training data? How long do we have to featurize training inputs and create a model?”

Earlier on I said that you should think about collecting data in the long term, because this is how you’ll be able to keep your models relevant. The question is: should you re-train models as soon as new data is available? This might take too much time and too many computational resources. Otherwise, how often should that be?

There are two reasons why you would want to update models: because having more data could lead to better models, or because the dynamics of whatever phenomenon or behavior you’re capturing with data might have changed. There are techniques to quantify that change and detect when it’s significant, but domain knowledge can also be used to **know when it’s useful to update models**.



Things change. Update your models.

In real-estate, you probably don't need to update your model every second, but updating every couple of years is likely to be too infrequent. A reasonable compromise could be every month. In the priority inbox example, updating models immediately after getting explicit user feedback would provide the best experience, and it should be enough to add implicit feedback data to the training set once per day. One could think that detecting fakeness of reviews is like detecting sentiment, where the nature of the problem doesn't change in time... but fakesters will constantly introduce new patterns of fakeness as their objective is to cheat the systems in place!

The domain will also dictate constraints such as **how much time can be allowed for model building**. Remember that before we can train a model, we need to create a training set of inputs and outputs, and we need to (re)compute feature values for the

inputs—this could actually be what takes the most time. In priority inbox, I would say that giving a few minutes to retrain the model is reasonable (same scale as the frequency of getting emails). In the customer retention example, we would typically need predictions to be available in the morning, so building the model, evaluating it, and using it on all current customers could be done overnight.

In the Canvas I filled in for customer retention, I mentioned a “hold-out set”. I’ll explain this in the part on [Feedback loops](#) of the section on [Decisions](#). In the real-estate deals one, I also wrote that we would only keep data which is up to 1 year old. In the same way we want to use the freshest data to keep our models relevant, we may also want to **remove data which is too old** and doesn’t capture the present situation.

For a given ML system, you could be creating and using **several models in parallel**. In priority inbox, for instance, you should have one model per user: what *I* think is important might not be important to *you*, so our email client shouldn’t use the same model! If an international platform like TripAdvisor would implement a fake review detection system, they would need one model per language; they might even want a different model for different countries that speak (almost) the same language—e.g. one model for the US and another one for the UK—because the dynamics might be different in those countries. At the extreme end of the spectrum are the ML systems used in platforms like Salesforce Einstein, where several models are created for each

customer: this makes more models than there are Data Scientists in the world!

The *Model building* section of the Canvas can also be used to specify other constraints such as model size—which would be particularly important in smartphone applications or embedded devices. Knowing all constraints related to model-building is essential so you can consider **the right ML algorithms to use in your system**, before you start implementing it.

IV. The PREDICT part

ML task

Input, output to predict, type of problem

Input

I like to think of ML tasks as questions in a certain format, for which the system we're building gives answers. The question has to be about a certain "object" of the real world (which we call the input). In the supervised learning paradigm—which we're focusing on in this book—we would make the system learn from example objects AND from the answers for each of them. The question to answer must be specific, for instance:

- "Is this email important?"
- "How much is this property worth?"

The inputs in those questions are an email and a property. The [*Data Sources*](#) listed in the LEARN part of the Canvas should provide information about these inputs. The attributes/characteristics of inputs that allow us to represent them in a computer program, would be listed in the [*Features*](#) box.

Output

The answer to the question (i.e. the output) is usually provided after a certain period of time—which is why we’re interested in “predicting” them, and in building predictive models. How and when the answer is observed should be covered in the [Collecting data](#) box of the Canvas. In the real-estate example, the output is provided by the market when the property has found a buyer.

I recommend listing all possible outputs (for a classification task), or their range (for a regression task). If your domain knowledge gives you any expectations on what their distribution should look like (e.g. ~10% of ‘Important’ emails, or log-values of properties being Gaussian), you should also share this in the Canvas. It will be useful to compare it to the distribution of the outputs you actually collect. Clarifying this can give you a first idea of which metrics to consider when evaluating predictions’ accuracy (e.g. mean absolute percentage error or mean squared logarithmic error for real-estate price prediction).

Baselines

Once you’ve decided on a prediction task, think about what a human would do to approach this task. This can be very useful to start thinking about data preparation and features to extract (to be listed under [Features](#)); it can also give insights for the [Building models](#) box (e.g. how often to update your model).

If humans are already performing this prediction task, their performance will provide a baseline to compare to, once you've decided on an evaluation method. If not, is there any alternative way (or an "old way") to provide predictions? A simple heuristic could also provide a baseline; you may expect its performance to be terrible, but at least that gives you something to compare your future model to (and check that its performance is strictly above). The variables used by your heuristic could also serve as features for your ML model.

Decisions

When and how are predictions used to make decisions that provide the proposed value to the end-user?

We want our ML system to predict the answers to the question we wrote in the [ML task](#) box, before the true answers can be observed. These predictions might be very difficult to get right. Getting accurate predictions of values that would only be observed after a long period of time can be impressive; but in the end, predictions are just information, and they don't do anything useful on their own... What we need is to turn predictions into decisions that deliver the value that we proposed (as the starting point of our ML system creation). Asking yourself "What if I had perfect predictions?" makes it easier to think about these decisions, before spending too much time on model building.

"Great predictive modeling is important, but as products become more sophisticated, it disappears into the plumbing."
— [Jeremy Howard](#)

You can then start thinking about how to evaluate those decisions, and not just the predictions they're based on. We will cover this in more detail in the section on [Offline evaluation](#) (and

we will also see that there can be situations where perfect predictions could have no value at all).

When

Another good way to make things concrete in terms of what to do with predictions, is to consider when exactly decisions should be made. This could be dictated by how frequently new inputs pop up in the domain of application, for instance, or by when/how often end-users will be using the ML system. The latter depends on how we can—or how we choose to—integrate the system in their workflows.

In the priority inbox case, the decisions would simply be to move incoming emails that are detected as important, to a dedicated section at the top of the inbox. We don't control users' workflows, and they could be logging into their email client at any time; therefore, it's best to make predictions when new inputs pop up, i.e. when new emails are received.

In other cases, such as customer retention or real-estate investment, it sounds reasonable to work with end-users to agree on a new workflow that uses the ML system, and to decide how often the system will make its decisions (or prescriptions) available (e.g. on a weekly or monthly basis). This will then have an impact on how much into the future we want to predict things, hence on the ML task. For instance, if we build a churn prevention system that will be used once per month, it makes

sense to predict churn within a month's time, or maybe within a couple of months. It doesn't make sense to predict within a week's time, and it's overkill to predict churn within a year's time.

Features as levers

In some ML use cases, your input could be made up of features which are given and fixed, and others that you could control. For instance, in product/pricing optimization, we would control the price of products, and in loan default prediction, we might be able to change loan characteristics such as amount and duration. We can think of the controllable features as levers that can be pulled to influence an outcome (that we're predicting). The decisions to make could be to adjust these feature values in order to maximize a certain quantity (e.g. predicted sales), or the probability of observing a desired outcome (loan reimbursement). Read more in [Designing great data products](#).

Confidence values

Decisions are often based on the model's confidence in its predictions. Let's take the hotel reviews case. If the model is very confident that the review is real/fake, we can let the system automatically decide to accept/reject that review. Otherwise, we may leave the decision to a human. A confidence value (usually

between 0 and 1) is given by the model along with each of its predictions, and it can be used to automate decisions when above certain thresholds.

We could also base the decision of the priority inbox system on the confidence in an email being important, and a threshold. Adjusting the threshold would change how many emails would make it to the priority inbox.

For customer retention, I propose to do the following:

- Filter out customers who are not predicted to churn, and anomalous customers
- Sort customers by descending value of (churn probability times monthly revenue)
- Target the first K customers in the list.

K is a parameter whose value will need to be fixed, and so are the thresholds I mentioned earlier. Changing them, or in general any parameter of the decision-making system, will have an impact on the performance of the whole ML system. We'll need to discuss performance evaluation and to define metrics of interest, before we can start looking for optimal parameter values.

Prescriptions or fully-automated decisions?

The ultimate vision when building intelligent systems can be to automate decisions completely. In certain domains, decisions made by a machine alone are better than those made with a human in the loop. In others, involving humans is hardly feasible, so automation is just a necessity. The priority inbox system falls into that category. Actually, it does something very similar to a spam filter: it makes predictions on incoming emails and moves them into a dedicated box (“priority” or “spam”) accordingly. We take this for granted today, but if you think about the time when spam filters were introduced, people were getting way less emails than today, and all of a sudden they were told: “we’re going to automatically hide some of your incoming emails”. This was probably not that easy to accept! What gain are you providing to the end-user, which can make it easier for them to accept ML-powered decisions? Can you explain how the decision was made, to gain their trust? For priority inbox, Gmail used to make this explanation available from the user interface, but now they just invoke their “magic sauce”—see the screenshot I shared in [Collecting data](#).

If possible, I recommend to avoid full automation when you’re just starting out with ML. Instead, you could list all possible decisions to be made, have the machine sort them based on predictions and confidence values, and have a human review. The machine would be providing prescriptions, and the final decisions would be yours. For the real-estate example, you could

put the best deals at the top of the list of properties to visit (i.e. filter out properties where asking price is higher than predicted price, and sort the rest by the difference between predicted price and asking price). For customer retention, you may want to sort the list of all customers, from most likely to churn to least likely. You may also want to mix this churn probability with other quantities, such as monthly revenue for each customer.

It can be very useful to add some additional information to the list of decisions, to help the end-user review. First, we could add explanations for all the predictions that were made, which can give insights to fine-tune decisions. For example, if our model is very confident that a given customer will churn, it would be useful to know why (e.g. because they make a lot of international calls) so that the customer support team could adapt their retention efforts (e.g. they could let the customer know about the international add-on, which would save them money). In general, this is also useful to get users to trust the system, and to check whether the predictions make sense—always a good idea before enabling full automation!

Second, you could add anomaly scores for all the inputs on which predictions were made. If there is a high anomaly score, it means that we made a prediction for an input which is significantly different from all the inputs that were seen in training. It's an additional difficulty to find the right automatic response/fallback for an anomalous input... Again, it's probably best to have a human review things.

Feedback loops

In [Collecting data](#), I made an important remark about the churn prevention case: we may correctly predict that customer *c* is at high risk of churning, use that prediction to do something to prevent them from churning, and thus observe the following month that they didn't churn. This would be great because we would be reducing churn, but not so great because it would lead to inconsistent data! We probably predicted that *c* would churn because of similarities with other customers who churned, but the following month we would be adding to our data that *c* did not churn; the training data would contain similar inputs that have dissimilar outputs...

Creating a “hold-out” set of customers, as a very first step, provides a solution to this issue. In my example Canvas, the hold-out set is chosen to contain 50% of all current customers, chosen at random (the ratio could be different, but it's important that inputs are chosen at random). As the name indicates, these customers are held out of all the decisions that will follow. That means they won't be impacted by decisions, and we'll be able to observe, one month later, who did churn and who did not. There is no issue here in using these observations as additional training data for our predictive model—and as we said in [Collecting data](#), we should aim at continuously collecting data and growing the training set!

In general, when the decisions we make have an impact on what we observe in the domain of application, we say that there is a “feedback loop” (the decisions that impact observations are based on predictive models, and predictive models are built from observations). Such feedback loops are also found in recommender systems: we observe users’ interests in items that were presented to them, which are items that our model predicted could be of interest in the first place! This can result in filter bubbles. If we don’t allocate slots for randomly chosen items in our recommendations (what we would call “exploration”), we may never discover other (and potentially stronger) interests of our users. There is a trade-off between exploitation of the model (to gain value from it in the short term), and exploration.

One example where feedback loops can be dangerous is predictive policing. A predictive model is used to allocate police across a city, arrest counts are used to help update the model, and police ends up being repeatedly sent back to the same neighborhoods regardless of the true crime rate. (See Runaway Feedback Loops in Predictive Policing for a solution.)

Towards Reinforcement Learning?

One solution to deal with the feedback loop issue could be to add features to our training data, to characterize the decision which was made for each input. This way, the model would have a

chance to find that the churn/no-churn output is related to the customer, or to the action that was taken on that customer, if any. This is something that Reinforcement Learning (RL) models well, but it's a harder problem for two reasons: initially we have zero data about decisions/actions, and the model has to understand more complex relationships (impact of customer profile on output, and impact of action). Therefore, I recommend to work on a 1st version of your Canvas without RL, and to integrate RL in a 2nd version of your Canvas. When implementing a Machine Learning system, you may want to start with the one described in the 1st version of your Canvas, but you could already start collecting data on decisions/actions, which will serve when implementing RL later on.

Taking the domain into account

Finally, you may need to take domain-specific logic and constraints into account in your decision-making system. For instance, in the churn prevention case, you would not want to target the same customer two months in a row, and you would have a fixed marketing budget. In a product recommendation system that would be based on users' predicted interests, you would not want to show products that are out of stock—which is something that your predictive model has no idea about—and you may want to post-process recommendations in order to introduce some diversity, in a way that makes sense in the domain.

Making predictions

Technical constraints on predictions made to support decisions: volume, frequency, time, etc.



Predictions. Delivered exactly when you need them.

Frequency and volume

After filling out the [Decisions](#) box, we would know when predictions should be made, but not necessarily how many of them and at which frequency. For churn prevention, we know that there would be 1 prediction per customer per month, but it would be useful to also give an idea of how many customers

there are. For priority inbox, we would want to estimate the number of users and the average frequency of incoming emails. For the real-estate deals case, how many new properties can we expect on the market every week? For fake reviews detection, how often do we get new reviews?

In all these examples, the number of predictions to make is the same as the number of decisions. However, that may not always be the case. For instance, if you would build a marketing campaign personalization system, for each customer you would be making as many predictions as there are personalization options (so you can determine which is best for that customer).

As you're developing a better understanding of which predictions will be made and when, I recommend to revisit the [Features](#) box (in the *LEARN* part of the Canvas). It's important to make sure that, at prediction time, you'll be able to use all of the features you listed. Consider reimplementing priority inbox in Gmail, for instance. You may be tempted to access the *isStarred* property, which is available in the Gmail API and tells whether an email was starred or not. You could extract a dataset that includes this variable, build and evaluate a model, and get very high accuracy. However, *isStarred* would always be False when predicting the importance of a new email, because the end-user would not have had the chance to interact with that email (and therefore to star it)... The best way to avoid these issues is to ask yourself, for each feature, how to extract its value from the data sources *at prediction time*.

You could also revisit [Collecting data](#), and reflect on differences between the inputs that will be collected and added to the training data, and the inputs on which you will be making predictions (the best scenario being when the former are representative of the latter).

Time

The time available for each prediction would be informed by the domain of application, by what's acceptable for the end-user, and by the time that's already allocated for computing decisions (if any). It would include time for the model to make a prediction, plus time to extract feature values. Often times, it is actually this featurization which is the longest, as it involves accessing various databases and running feature extraction methods.

In the customer retention case, we would need to “refresh” the feature-representations of customers every time new predictions are made on them (i.e. every month, in my example); this would involve recomputing variables such as how many times the customer logged in, average time spent using the product, etc. In the priority inbox case, we would need to recompute “social features” such as time taken to reply to the sender’s previous emails.

Other constraints

In addition to prediction volume, it is useful to think about any additional constraints on making predictions, and to consider where predictions might be made. For instance, we might need our system to be robust to loss of internet connectivity, in which case we'll want to make predictions directly on the end-user's device.

If you're integrating an ML system in an IoT application, predictions would either be made on the edge or in the cloud. Maybe devices will not be powerful enough to run predictions in time, and our constraints will clearly point towards cloud usage; or maybe prediction volume would be too big and would point towards making predictions on-device (assuming all the input features would be available locally). In the case where you control the hardware that runs your ML system, you would also want to choose that hardware in consideration of technical constraints. Could predictions be made in batches? (If so, you may want to equip your device with a GPU; otherwise you would stick to a CPU, or you may use an FPGA if time is critical.) In the case where you don't control the hardware (e.g. predictions are made on the end-user's device), you may have to test your system on a variety of hardware.

Monitoring in production

It is best practice to deploy predictive models into production as Application Programming Interfaces (APIs). This makes it easier to integrate them into the end-users' applications. The constraints on making predictions would translate into constraints for the API that powers predictions. When monitoring this API, you will be able to check that constraints remain satisfied in production, and also that the actual volume of predictions matches your expectations (if not, this could indicate issues such as undeclared consumers, which can end up creating hidden feedback loops—see the paper on [technical debt in ML systems](#)).

Offline evaluation

“Methods and metrics to evaluate the system before deployment.”

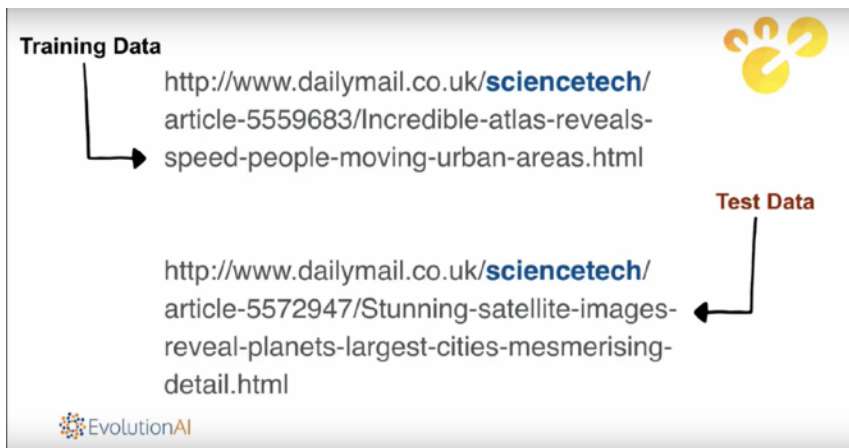
Evaluating an ML system entails evaluating its model(s), and it should be performed every time a new model is created. Evaluating the model’s accuracy isn’t uninteresting, but what we really want to do is to pre-evaluate the future impact of decisions, so we can build trust that the system is ready for deployment. I like to think of the offline evaluation as a simulation, where we’re trying to answer “how well would we do if we would deploy that system on these test cases?”.

Test cases

For the simulation to be trust-worthy, it should run on test cases that are representative of the cases that the system would encounter in the real world. Similarly, we should train the system on data which is representative of what the system will be tested on. But we shouldn’t let the system know too much about what it will be tested on...

How should we split the available data into a training set and a test set? A common mistake is to split randomly. In his talk at

PAPIs Europe 2018 on [the limits of decision making with AI](#), Martin Goodson gives the example of a news classification system which would find a rule that would be specific to DailyMail articles; if the test set also contains DailyMail articles, it will do well on these, which will be reflected positively in the evaluation. However, the system should find rules that are general to all news sites, as it may be used primarily on sites that were not in your training data. If the train-test split is random, your evaluation results could be overly optimistic. It's important to apply knowledge of the domain and of the problem, in order to segregate data into training and test sets.



Slide from Martin Goodson's talk at PAPIs on [the limits of decision making with AI](#)

If inputs are time-bound, as is the case with customer snapshots, a random split could lead to having inputs in your training set

which would be posterior to the inputs in your test set. This would make the evaluation very difficult to interpret...

It is good practice to choose the test set in a way that makes it easy to present meaningful results, and to interpret them in the domain where the system will be used. One way to do this is to use the most recent data as test, so we can answer “how well would we have done if we had deployed this system X days/weeks/months ago?”. This is what I chose to do in all the example Canvases.

Note that there is some ambiguity in the fake review detection Canvas: it’s not clear whether this system is built for one specific website where reviews are being left, or if it is meant to be generic and applicable to multiple websites of a certain type (e.g. hotel reviews on TripAdvisor, Booking, etc.). In the latter case, we would need to segregate reviews according to the websites where they were posted, in addition to splitting time-wise.

Test set size

Another thing to consider regarding the test set, and that you may want to indicate on the Canvas, is its size. In [Machine Learning Yearning](#), Andrew Ng has two things to say on this topic:

- “Your test set should be big enough to give you a confident estimate of the final performance of your system.”
- “The old heuristic of a 70–30 train-test split does not apply for problems where you have a lot of data; the dev and test sets can be much less than 30% of the data”

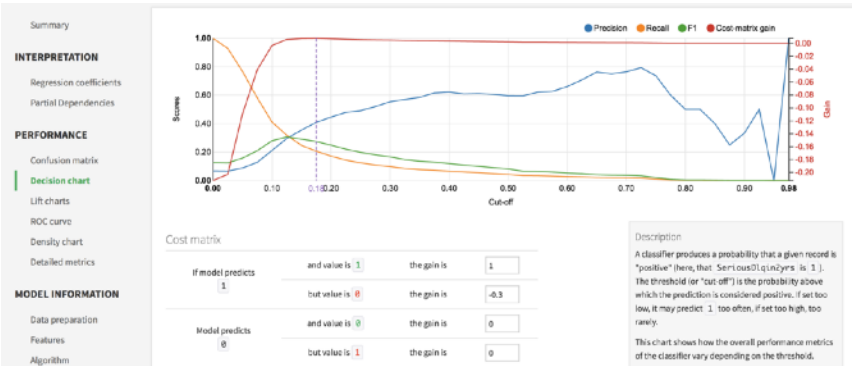
Another way to phrase this is that your test set should be big enough for your results to have meaning in the domain of application. One consequence is that if you plan model updates every X days, then you should test on at least X days’ worth of data. It could be more—for instance, if you’re modeling behavior that has a yearly pattern, you’ll want your test set to be over a period of 1 year (or more).

When evaluating your system, you also want to make sure that you can make predictions/decisions sufficiently fast (which would be determined by test set size and time constraints listed in [Making predictions](#)).

Metrics and performance constraints

Performance of your system can be measured in a number of ways, but Andrew Ng recommends the following: “Choose a single-number evaluation metric for your team to optimize. If there are multiple goals that you care about, consider combining them into a single formula or defining satisfying and optimizing

metrics.” Also, if we are to decide on whether to deploy to production, it’s probably best to focus on a domain-specific metric that quantifies the impact of our system.



So many performance metrics... Which one to choose?! (Screenshot of [Dataiku's](#) Data Science Studio)

As I already said, I like to think of an evaluation as a simulation. A good way to measure the performance of the system is to compute the gain of using the system, compared to not using it or to using something else, when running the simulation. It's a much better story to tell than reporting somewhat abstract accuracy metrics (e.g. the F1-score of a classifier). You might need to make certain assumptions for that; for instance, in the customer retention example, you should state your assumption for the success rate for your targeted retention efforts.

The first step towards finding your domain-specific metric is to interpret the meaning of errors. Let's consider binary classification to start with. There are two types of errors: *False*

Positives and *False Negatives*. In churn prediction, a False Positive is a customer who you think would churn, but eventually didn't. A False Negative is a customer who you didn't think would churn, but eventually did. The second step is to assign cost values for all possible errors. In our example, the cost of an FP would be the cost of targeting a customer, and the cost of an FN would be 0 (we lose this customer, and we would have lost them anyways without our system). Similarly, you can come up with gains (i.e. negative costs) for True Positives and True Negatives. The gain for a True Positive can be estimated by the revenue brought by the customer multiplied by the assumed success rate of our retention campaign, minus the cost of targeting the customer. The gain for a True Negative is 0. The cost/gain values are fixed for the problem at hand, and could be written in the Canvas. When running the simulation, we would just count the number of FP, FN, TP, TN, multiply each value by the associated cost value, and sum everything up.


		Prediction	
		Positive	Negative
Reference	Positive	True Positive	
	Negative	False Positive	True Negative

Illustration of a “confusion matrix” stolen from [Human-Centered Machine Learning](#)

Remember I said it’s easy to come up with situations where a perfect model would be useless? Imagine that our model predicts churn with perfect accuracy, hence we choose to target all customers predicted to churn. Imagine that the lifetime value of a customer is 10€, that the success rate of our retention campaign is 20%, and that the average cost of targeting a customer is 2€. In that case, the gain for a True Negative is still 0, and the gain for a True Positive is bounded by $10\text{€} * 20\% - 2\text{€} = 0$! The only options to improve things are to bring the targeting cost down, or to improve the success rate of our campaign; the latter might be achieved with personalization of the campaign, or by predicting which customers would be the most receptive and by only targeting them.

Let's move on to a regression case. The cost of an error could be a function of the amount of the error, but also of the input on which the error was made. In the real-estate deals example, the cost/gain function is based on the way predictions are used in our investment strategy, and it aims at estimating how much money we lose/make: if we decide to invest in what is predicted to be a good deal (*asking price* < *prediction*), but is essentially a bad deal (*sale price* < *asking price*), we incur a cost of (*asking* — *sale price*).

We would want to minimize the total cost in our simulation, but we would also want to make sure that no single cost is bigger than our bankroll—otherwise we would go bankrupt! In addition to a performance metric to optimize, it is common to have to satisfy performance constraints such as maximum amount of error, or maximum number/proportion of errors of a certain kind that we can tolerate. In the priority inbox example, we may want to make sure the number of important emails that weren't detected as such (False Negatives) is less than 1 every X days (based on the degradation of user experience that we can tolerate).

One last obvious constraint on the performance of the system is that it should be higher than that of the baseline. This means that before you learn any models from data, you should implement the offline evaluation and the baseline. Actually, if your baseline is a heuristic that satisfies all of your performance constraints, and if the offline evaluation/simulation shows that it provides

gains, you should deploy that baseline before doing any ML—if you haven’t done so already!

Validation set and parameter tuning

Until now, we’ve discussed measuring the performance of our system *via a test set*. That dataset should only be used to decide if all is ok to deploy to production. You should not use the test set to tune any parameters of the model-building process, nor of the decision process. The example Canvases are kept fairly simple and omit to mention that our available data should consist of not 2 but 3 datasets: a test set, a training set, and a *validation set*—which is used for parameter tuning and model selection.

When considering different approaches to model building, such as neural networks and ensembles of trees for example, we’ll run each of them on the training set, and we’ll apply them to the validation set in order to “validate” our choice of approach. The validation set can be used for all choices to be made when building the system: this can be parameters of the feature extraction, data pre-processing, or model-building procedures (also called “hyper-parameters”), or parameters of the decision system (such as the number of customers K to target, or the thresholds used for automatic decisions).

If you’re wondering why we can’t just use the test set for that, the reason lies in the fact that when we try different choices and

compare their performance on a given dataset (the validation set), we are using that dataset to *learn* which choice is the best on the problem at hand, and every learning should be tested with new data (i.e. the test set).

The validation set can consist of two subsets: an “eyeball validation set” and a “blackbox validation set”, as described by Andrew Ng in ML Yearning (in Andrew’s terminology, a validation set is called a *dev set*). The eyeball validation set is the one that will contain cases to be examined as part of the *manual error analysis*. It would typically contain a randomly selected subset of the whole validation set, but also edge cases or pre-defined cases where you would want to manually review models’ behavior (i.e. the errors, predictions, and explanations behind predictions), and make sure that it is satisfactory. You may not want to list all these cases in the Canvas, but you might want to give insights into how the validation subsets are created, which and how much data they contain.

As with the test set, you want to run your baseline against the validation set. You should also consider simple models (e.g. rule-based) when doing model selection, so you can quantify the real performance gain that an increase in model complexity offers (keep in mind that more complex models are typically more costly to maintain).

V. The EVALUATE part

Live evaluation and monitoring

Methods and metrics to evaluate the system after deployment, and to quantify value creation

Coming soon in version 0.2 — get updates at
machinelearningcanvas.com

About the author



Louis Dorard is an independent consultant and Machine Learning coach. He helps corporations and startups integrate ML into their products. He has held workshops at major companies such as Amazon, Deloitte, EDF, Intel, Konica Minolta, and he has coached smaller businesses and growing startups in their usage of ML. Louis holds a PhD in ML from University College London, and he has been working in the field for more than 10 years.

He is also the author of Bootstrapping Machine Learning, General Chair of PAPIs.io (international conferences on ML applications and APIs), and Adjunct Teaching Fellow at UCL School of Management.

Follow Louis on Twitter at @louisdorard, get in touch and find out more about him at louisdorard.com!

Copyright, creative commons, and a disclaimer

This handbook on the Machine Learning Canvas is protected by Copyright © 2019 Louis Dorard — All rights reserved.

The Canvas itself is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#). It is free to download, use, and build upon by anyone.

No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from the author, except for brief excerpts in reviews or analysis.

While every precaution has been taken in the preparation of this book, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

The author reserves the right to release subsequent editions as separate paid products or as part of training courses, training kits, or other product formats.