
Machine
Learning
for
Astrophysics

Machine Learning for Astrophysics

Who is this guy?

There is no single *Machine Learning* definition that fits all the possible applications and peculiarities of the field.

There is no single **Machine Learning** definition that fits all the possible applications and peculiarities of the field.

"The field of study that gives computers the ability to learn without being explicitly programmed"
(Arthur Samuel, 1958)

"ML is the process of training a piece of software, called a model, to make useful predictions from data. An ML model represents the mathematical relationship between the elements of data that an ML system uses to make predictions" (Google)

"ML is the study of computer algorithms that improve automatically through experience"
(Tom Mitchell, 1997)

There is no single **Machine Learning** definition that fits all the possible applications and peculiarities of the field.

"The field of study that gives computers the ability to learn without being explicitly programmed"
(Arthur Samuel, 1958)

"ML is the process of training a piece of software, called a model, to make useful predictions from data. An ML model represents the mathematical relationship between the elements of data that an ML system uses to make predictions" (Google)

"ML is the study of computer algorithms that improve automatically through experience"
(Tom Mitchell, 1997)

Machine Learning is an umbrella term for a set of statistical and informatic techniques to extract knowledge from **DATA**. This knowledge can be self-consistent as it is, or re-used to improve the overall performance of a ML system.

There is no single **Machine Learning** definition that fits all the possible applications and peculiarities of the field.

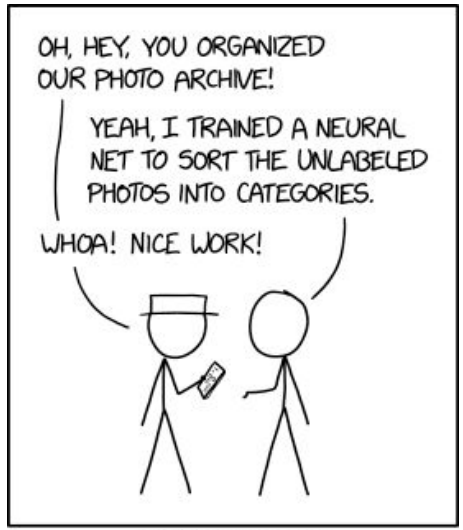
"The field of study that gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1958)

"ML is the process of training a piece of software, called a model, to make useful predictions from data. An ML model represents the mathematical relationship between the elements of data that an ML system uses to make predictions" (Google)

https://xkcd.com/1838/

"ML is the study of computer algorithms that improve automatically through experience" (Tom Mitchell, 1997)

Machine Learning is an umbrella term for a set of statistical and informatic techniques to extract knowledge from *DATA*. This knowledge can be self-consistent as it is, or re-used to improve the overall performance of a ML system.



ENGINEERING TIP:
WHEN YOU DO A TASK BY HAND, YOU CAN TECHNICALLY SAY YOU TRAINED A NEURAL NET TO DO IT.

https://xkcd.com/2173/

There is no single **Machine Learning** definition that fits all the possible applications and peculiarities of the field.

"The field of study that gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1958)

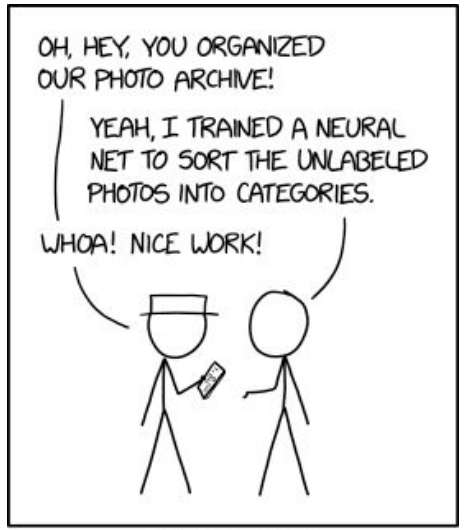
"ML is the process of training a piece of software, called a model, to make useful predictions from data. An ML model represents the mathematical relationship between the elements of data that an ML system uses to make predictions" (Google)

https://xkcd.com/1838/

"ML is the study of computer algorithms that improve automatically through experience" (Tom Mitchell, 1997)

Machine Learning is an umbrella term for a set of statistical and informatic techniques to extract knowledge from *DATA*. This knowledge can be self-consistent as it is, or re-used to improve the overall performance of a ML system.

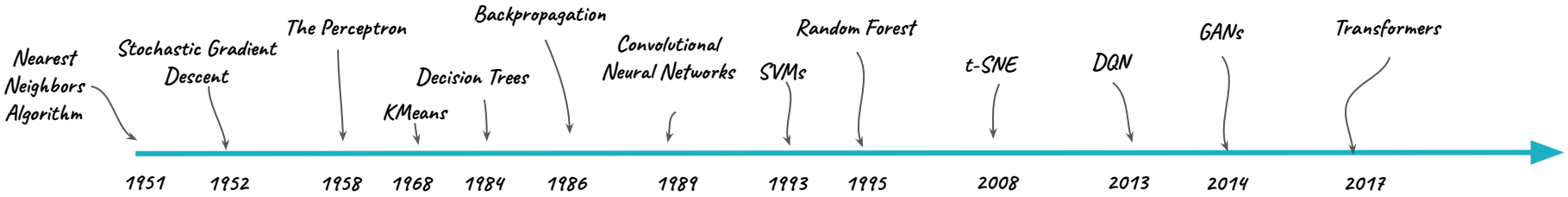
"It's easy: when I write applications for EU fundings its Artificial Intelligence, when I talk to students its Machine Learning, when I talk at conferences its Deep Learning or Neural Networks"



https://xkcd.com/2173/

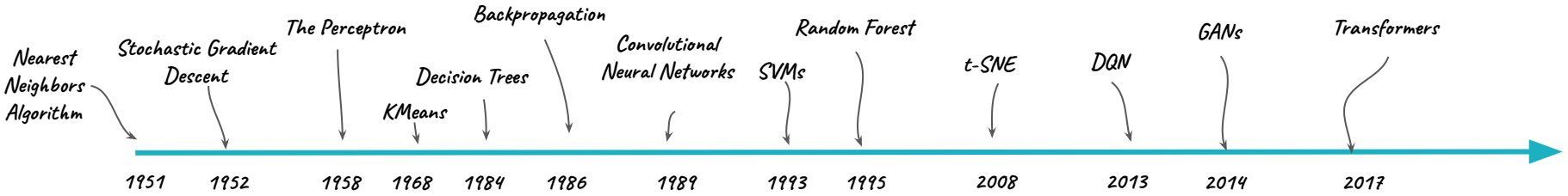
Samuel's definition dates back to 1958. Most of the things I'll show to you date before the 2000s.

So, why the field exploded in 2010s?



Samuel's definition dates back to 1958. Most of the things I'll show to you date before the 2000s.

So, why the field exploded in 2010s?



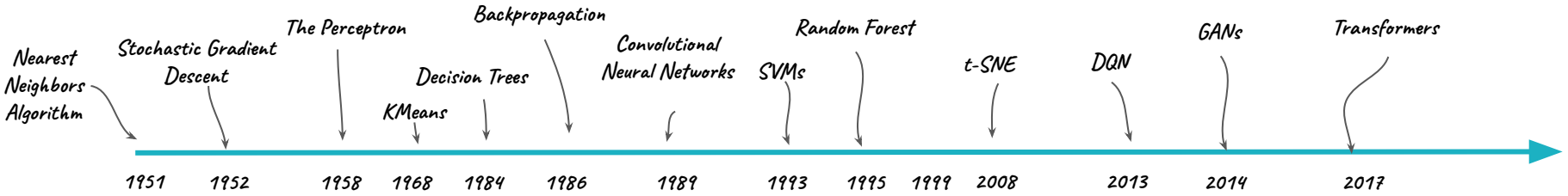
DATA

Specifically, ***BIG data***.

Most of the things you will see require large dataset to work efficiently, e.g. with acceptable quality metrics; dataset that were not that widely available before the early '00.

Samuel's definition dates back to 1958. Most of the things I'll show to you date before the 2000s.

So, why the field exploded in 2010s?



DATA

Specifically, ***BIG data***.

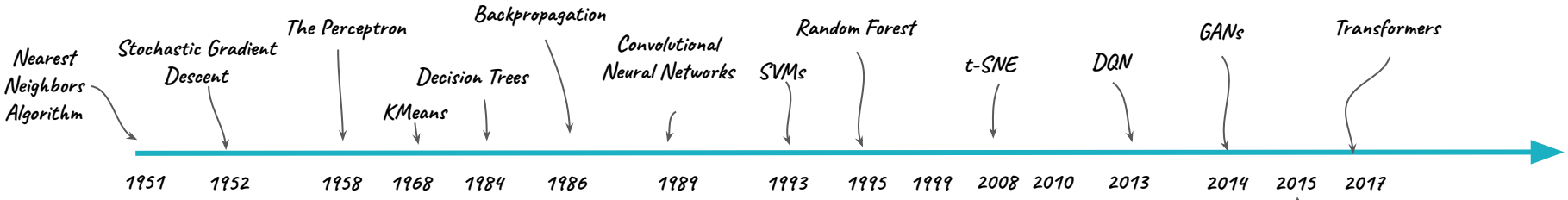
Most of the things you will see require large dataset to work efficiently, e.g. with acceptable quality metrics; dataset that were not that widely available before the early '00.

HARDWARE

To be more precise: ***GPUs***.

Samuel's definition dates back to 1958. Most of the things I'll show to you date before the 2000s.

So, why the field exploded in 2010s?



DATA

Specifically, ***BIG data***.

Most of the things you will see require large dataset to work efficiently, e.g. with acceptable quality metrics; dataset that were not that widely available before the early '00.

HARDWARE

To be more precise: ***GPUs***.

SOFTWARE

The last decade saw the deployment of lots of newly refined (and improved) techniques, all easily available on freeware software and suites.

TensorFlow

scikit learn

PyTorch

Outline

Introduction to ML & Supervised ML

- Introduction
- Regression, Regularization
- Classification, Logistic Regression
- Bias/Variance trade-off

Supervised ML strikes back

- Support Vector Machines
- Gaussian Processes
- Nearest Neighbors
- Ensemble Methods: random forests
- Gradient Boosting

Unsupervised ML

- Clustering: KMeans, DBScan, GMM, Agglomerative Clustering
- Dimensionality Reduction:
  - linear: PCA, NMF, ICA
  - manifold learning: LLE, IsoMap, t-SNE
- Anomaly Detection
- Self-Organizing Maps

Deep Learning

- A few notes on HPC and GPUs
- Basics of NN: forth- and back-propagation
- Shallow/Deep Neural Networks
- Bayesian NN, Probabilistic BNN
- Transfer Learning

Deep Learning, The Revenge

- (hints of) Reinforcement Learning
- Convolutional Neural Networks
- (Recurrent Neural Networks)
- GAN
- Autoencoders and VAE
- Transformers

Outline

Introduction to ML & Supervised ML

- Introduction
- Regression, Regularization
- Classification, Logistic Regression
- Bias/Variance trade-off

Supervised ML strikes back

- Support Vector Machines
- Gaussian Processes
- Nearest Neighbors
- Ensemble Methods: random forests
- Gradient Boosting

Unsupervised ML

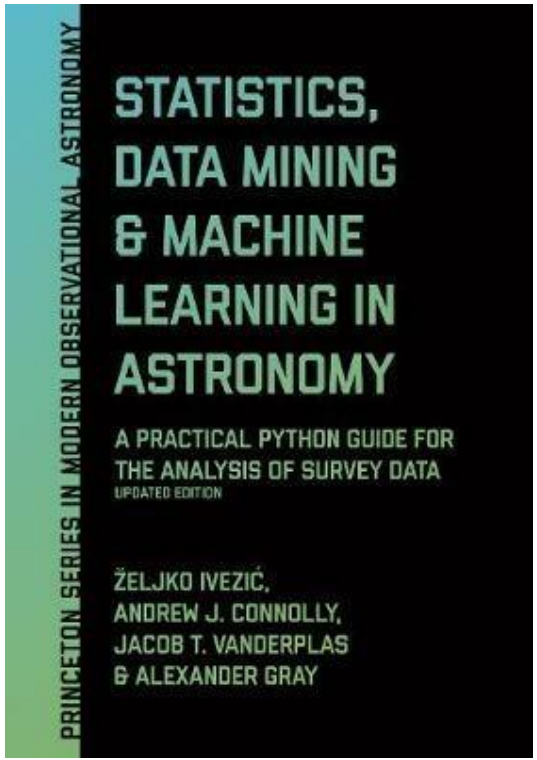
- Clustering: KMeans, DBScan, GMM, Agglomerative Clustering
- Dimensionality Reduction:
  - linear: PCA, NMF, ICA
  - manifold learning: LLE, IsoMap, t-SNE
- Anomaly Detection
- Self-Organizing Maps

Deep Learning

- A few notes on HPC and GPUs
- Basics of NN: forth- and back-propagation
- Shallow/Deep Neural Networks
- Bayesian NN, Probabilistic BNN
- Transfer Learning

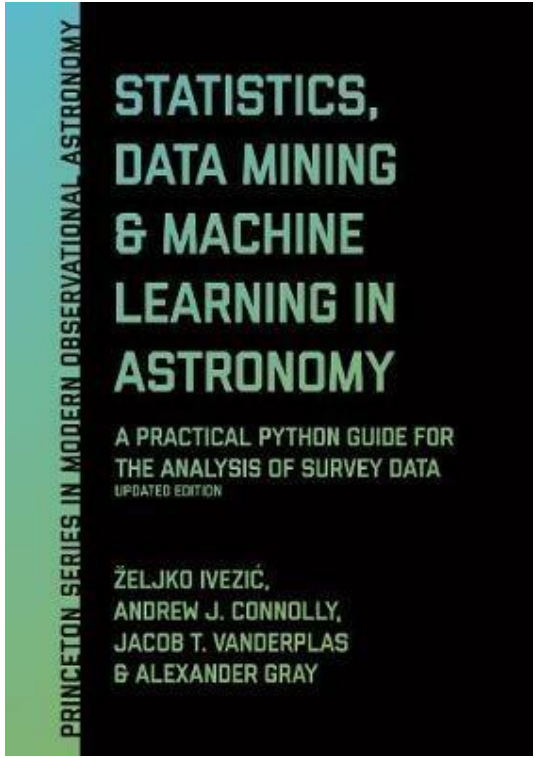
Deep Learning, The Revenge

- (hints of) Reinforcement Learning
- Convolutional Neural Networks
- (Recurrent Neural Networks)
- GAN
- Autoencoders and VAE
- Transformers



http://www.astroml.org/astroML-notebooks/

References



http://www.astroml.org/astroML-notebooks/

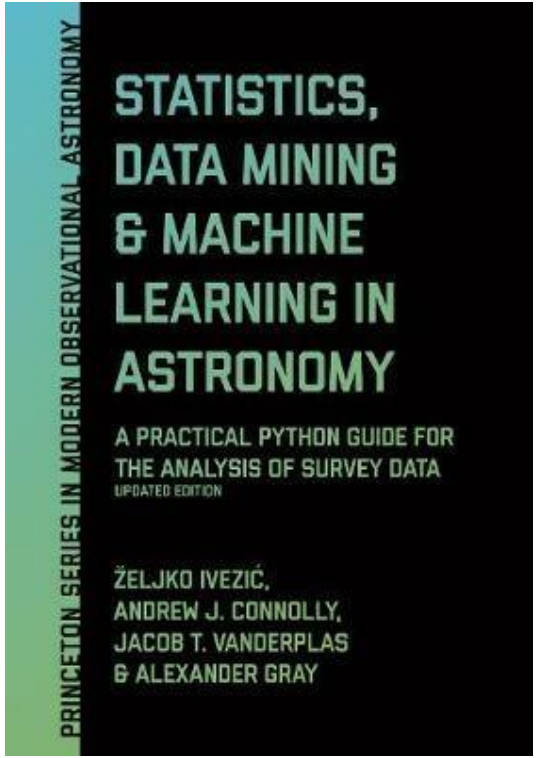
Machine Learning Crash Course

A hands-on course to explore the critical basics of machine learning.

https://developers.google.com/machine-learning/crash-course
https://developers.google.com/machine-learning/foundational-courses

https://www.coursera.org/specializations/machine-learning-introduction

References



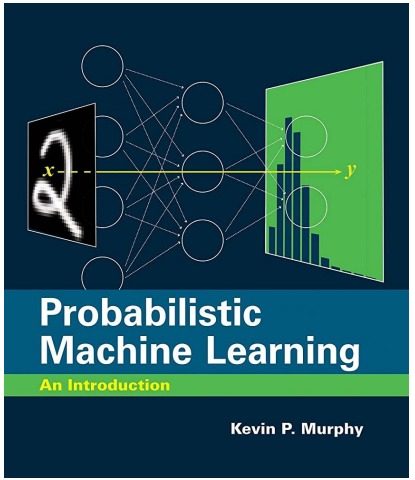
http://www.astroml.org/astroML-notebooks/

Machine Learning Crash Course

A hands-on course to explore the critical basics of machine learning.

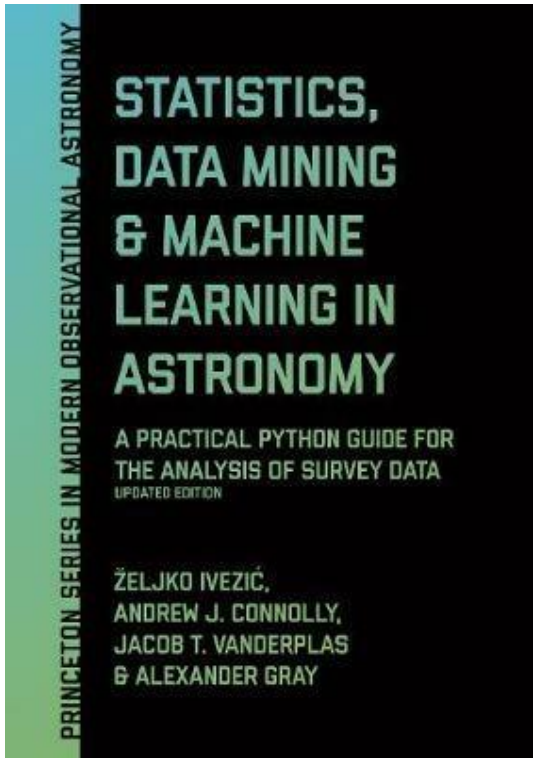
https://developers.google.com/machine-learning/crash-course
https://developers.google.com/machine-learning/foundational-courses

https://www.coursera.org/specializations/machine-learning-introduction



https://probml.github.io/pml-book/book1.html

References



http://www.astroml.org/astroML-notebooks/

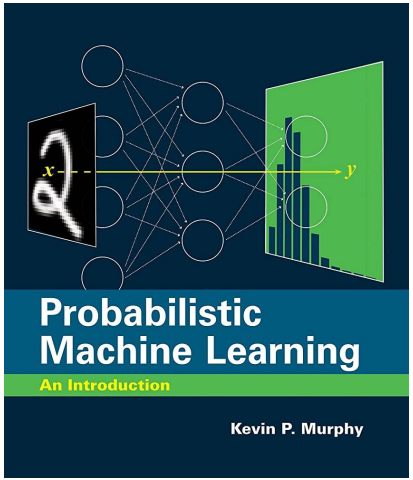
+ many more scattered through the slides

Machine Learning Crash Course

A hands-on course to explore the critical basics of machine learning.

https://developers.google.com/machine-learning/crash-course
https://developers.google.com/machine-learning/foundational-courses

https://www.coursera.org/specializations/machine-learning-introduction



"an introduction"
= 855 pages

https://probml.github.io/pml-book/book1.html

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

Goal:
learn a function that maps
$x \rightarrow y$

"This thing is a dog"

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

Goal:

learn a function that maps

$x \rightarrow y$

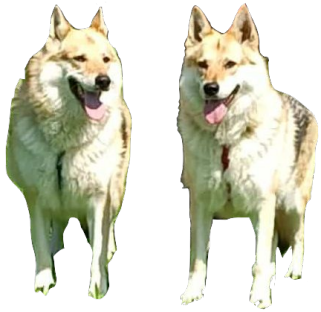
"This thing is a dog"

Unsupervised Learning

Data: x
there are no labels, only data x

Goal:

learn underlying structure of x



"These two things look alike"

The Holy Trinity of Machine Learning

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

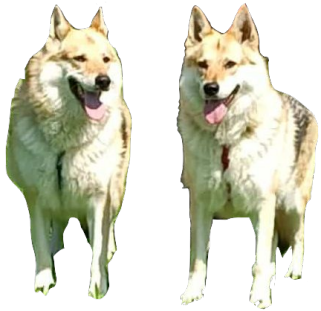
Goal:
learn a function that maps
$x \rightarrow y$

"This thing is a dog"

Unsupervised Learning

Data: x
there are no labels, only data x

Goal:
learn underlying structure of x



"These two things look alike"

Reinforcement Learning

Data: *state-action* pairs

Goal:
learn a policy π
maximizing future rewards

"Cuddling this thing will make you happy"

The Holy Trinity of Machine Learning

"Weakly Supervised Learning"

"Self-Supervised Learning"

"Semi-Supervised Learning"

Wide use of NNs to learn the policy π

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

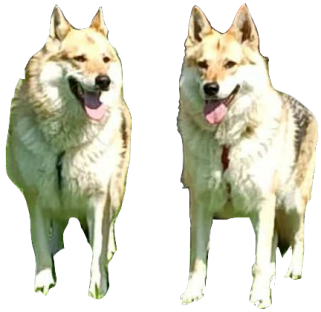
Goal:
learn a function that maps
$x \rightarrow y$

"This thing is a dog"

Unsupervised Learning

Data: x
there are no labels, only data x

Goal:
learn underlying structure of x



"These two things look alike"

Reinforcement Learning

Data: *state-action* pairs

Goal:
learn a policy π
maximizing future rewards

"Cuddling this thing will make you happy"

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

Goal:
learn a function that maps
$x \rightarrow y$

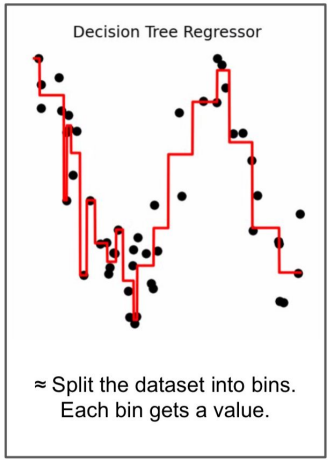
"This thing is a dog"

Build **models** that learn
how to ***combine*** **inputs**
to ***produce*** **predictions**
(even) on **never seen before data**

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

Goal:
learn a function that maps
$x \rightarrow y$



A **feature** is an input variable, describing our data → $x_1, x_2, \ldots, x_n$

A **label** is the variable we're predicting → y

An **example** is one instance of data → x
a **labeled example** has (features, label) (x, y)
an **unlabeled example** contains only features, and we want to find the label

A **model** defines the relationship between features and label, mapping **examples** to predicted labels y'.
It is defined by its own internal parameters, which are learned from the **labeled examples**.

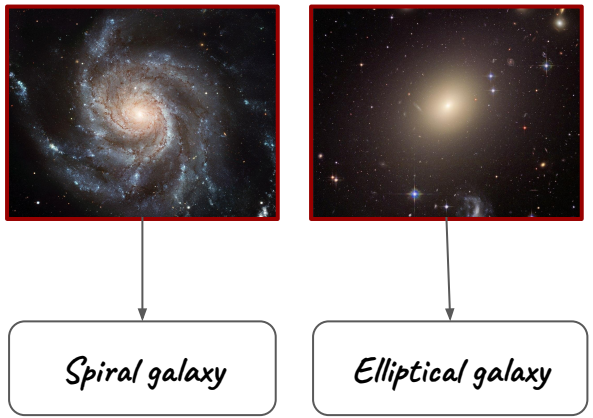
Training a model means showing the model **labeled examples** ("the ***training set***") and enable the model to gradually learn the relationships between features $x_1, .., x_n$ and label y.
Inference means applying the trained **model** to **unlabeled examples**.

A **regression model** predicts continuous values.

Supervised Learning

Data: (x, y)
x is the data, with associated labels y

Goal:
learn a function that maps
$x \rightarrow y$



A feature is an input variable, describing our data → $x_1, x_2, \ldots, x_n$

A label is the variable we're predicting → y

An **example** is one instance of data → x
a **labeled example** has (features, label) (x, y)
an **unlabeled example** contains only features, and we want to find the label

A **model** defines the relationship between features and label, mapping **examples** to predicted labels y'.
It is defined by its own internal parameters, which are learned from the **labeled examples**.

Training a model means showing the model **labeled examples** ("the ***training set***") and enable the model to gradually learn the relationships between features $x_1, .., x_n$ and label y.
Inference means applying the trained **model** to **unlabeled examples**.

A **regression model** predicts continuous values.
A **classification model** predicts discrete values (classes/categories).

We are going to start with something extremely simple and familiar.
The blueprint example of literally almost every ML courses is the Housing Prices in California (which actually tells you a lot about the US).
This is ML for Astrophysics, so we're going to use another kind of linear regression:

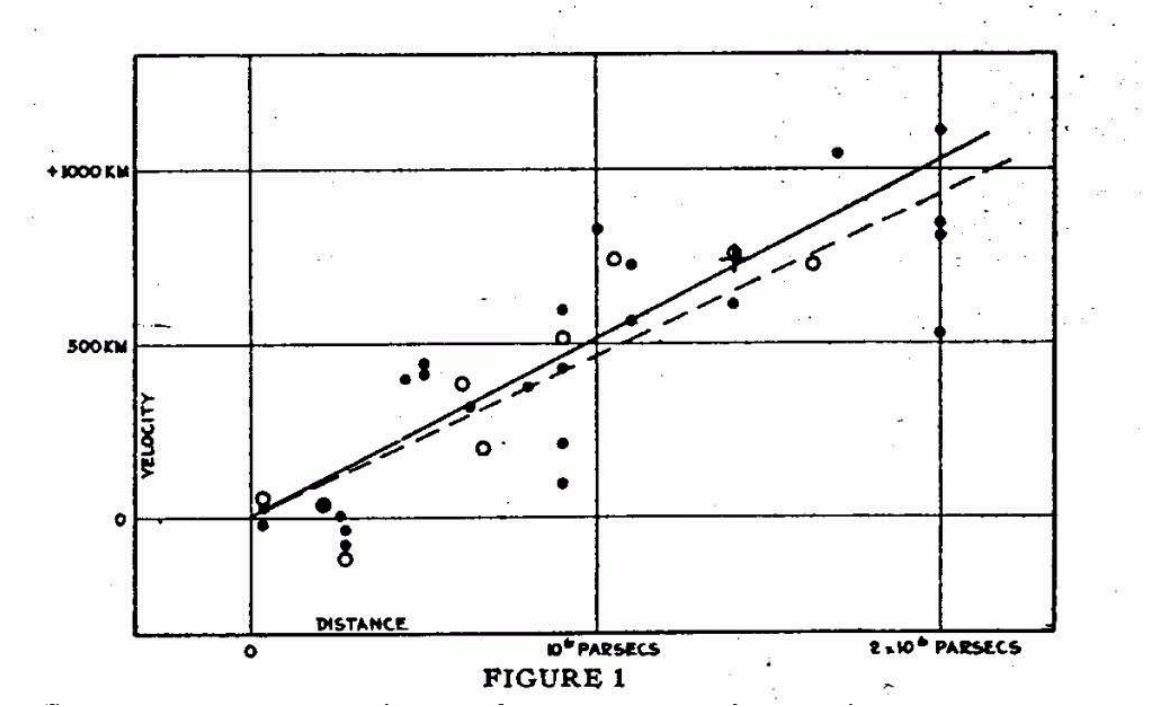


FIGURE 1

We are going to start with something extremely simple and familiar.
The blueprint example of literally almost every ML courses is the Housing Prices in California (which actually tells you a lot about the US).
This is ML for Astrophysics, so we're going to use another kind of linear regression:

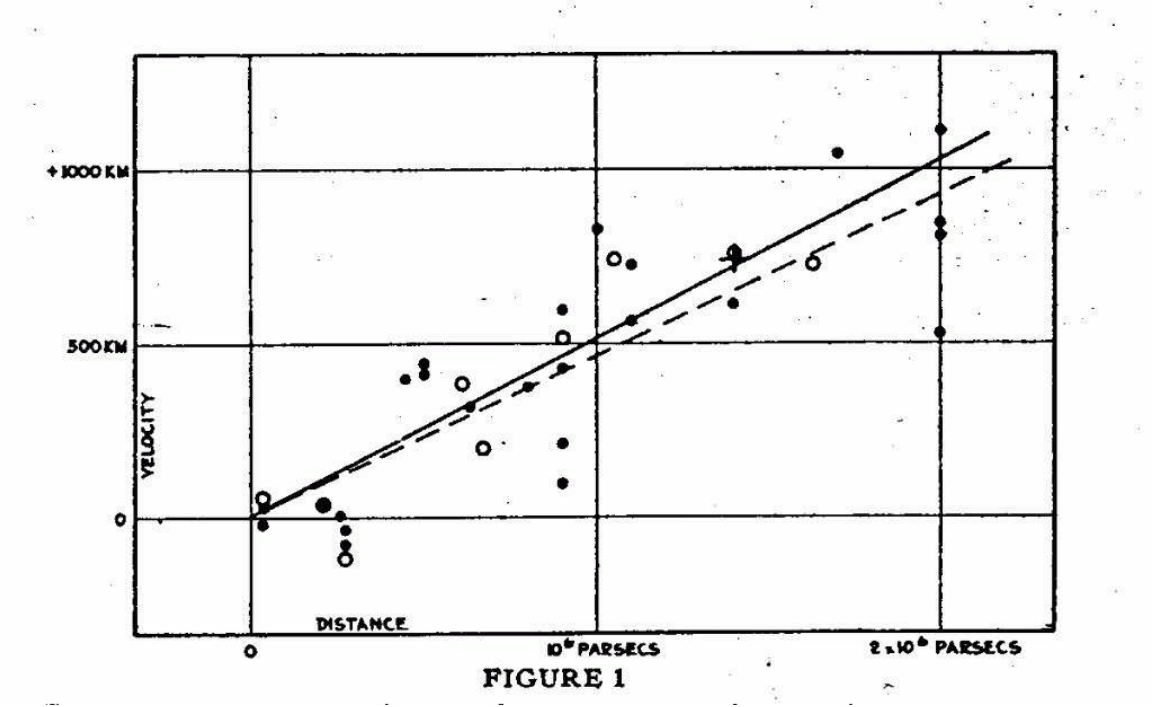
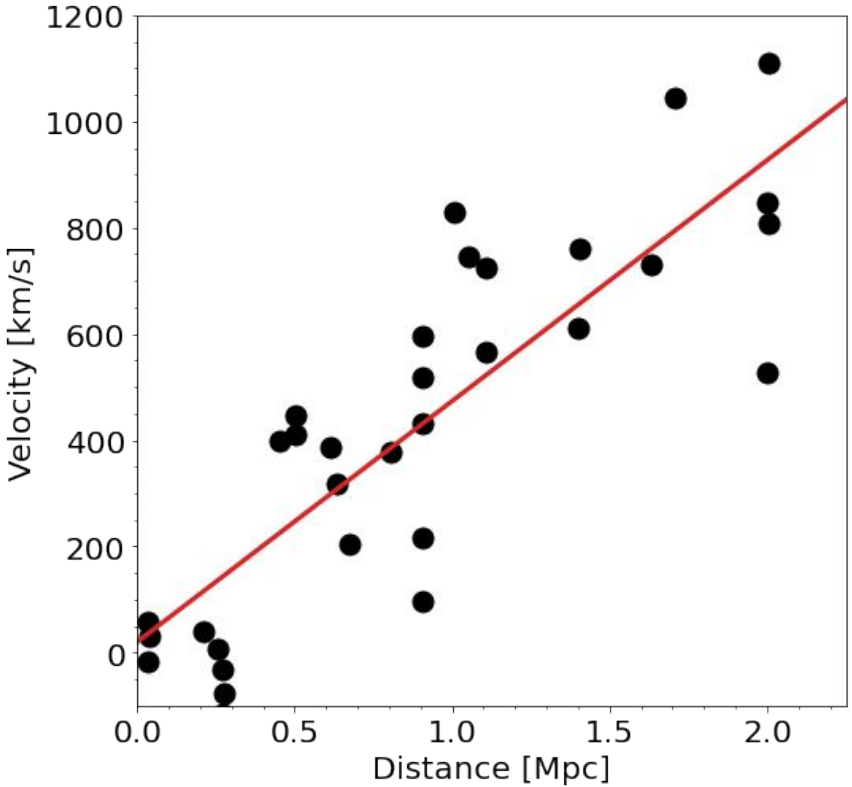


FIGURE 1

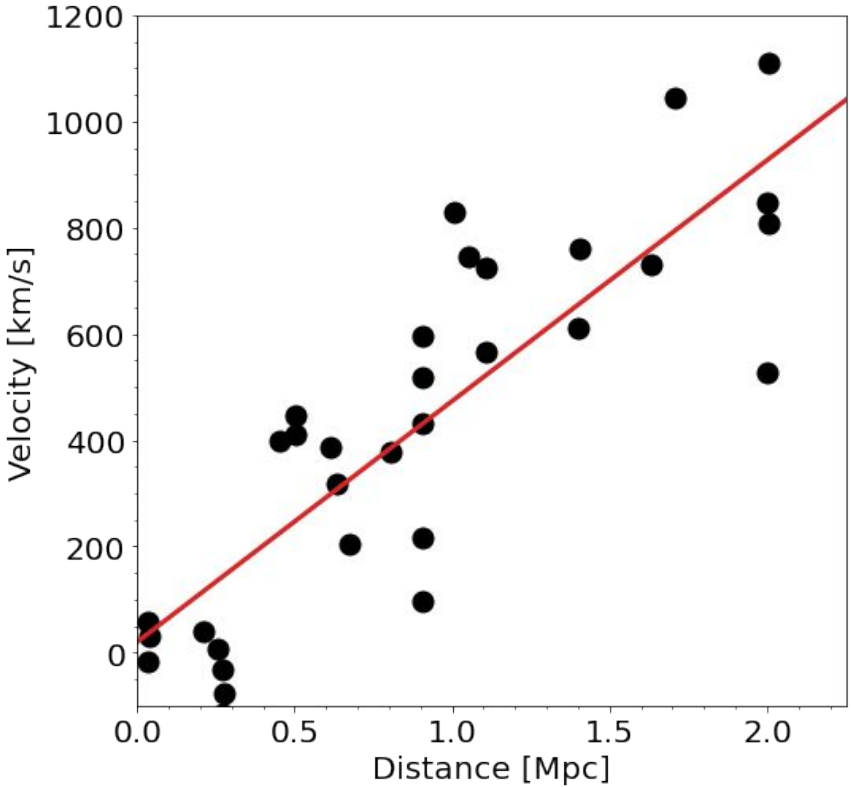
Hubble E., 1929, PNAS, 15, 168.

We are going to start with something extremely simple and familiar.
The blueprint example of literally almost every ML courses is the Housing Prices in California (which actually tells you a lot about the US).
This is ML for Astrophysics, so we're going to use another kind of linear regression:



The galaxies' distances in [Mpc] are the input **feature** x
The galaxies' velocities in [km/s] are the **label** y

We are going to start with something extremely simple and familiar.
The blueprint example of literally almost every ML courses is the Housing Prices in California (which actually tells you a lot about the US).
This is ML for Astrophysics, so we're going to use another kind of linear regression:



The galaxies' distances in [Mpc] are the input **feature** x
The galaxies' velocities in [km/s] are the **label** y

The **model** defining the relationship between the two is:

$$f_{w,b}(x) = wx + b$$

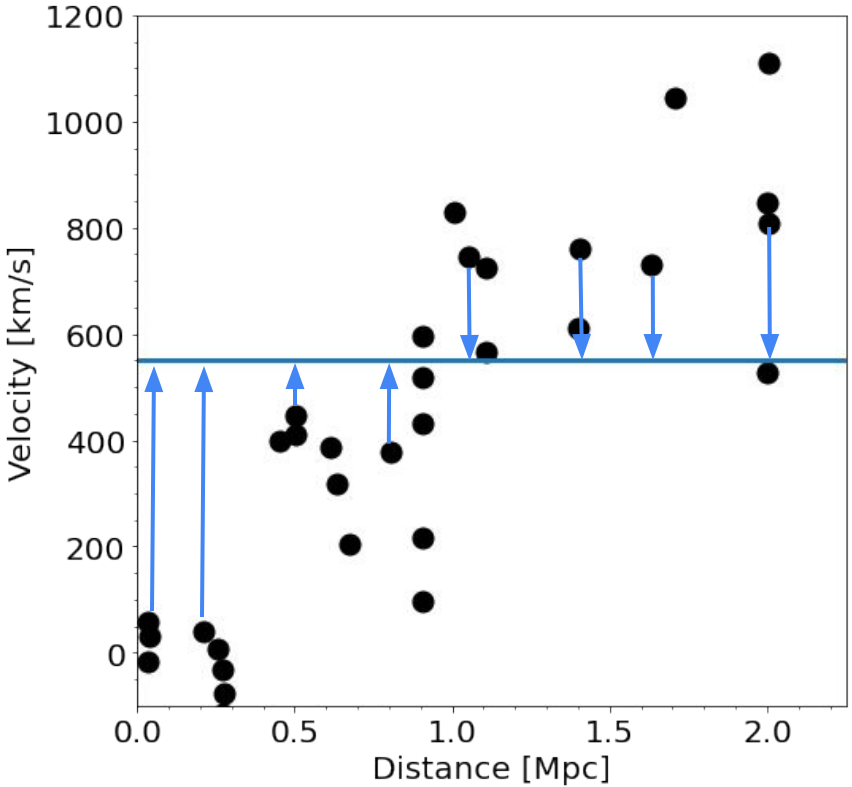
where:

b is the bias (sometimes also reported as w_0)
w is the **weight** of the **feature** x (and for multiple $x_i \rightarrow w_i$)
$\hat{y}^{(i)} = f_{w,b}(x^{(i)})$ is the predicted **label** for the unlabeled example $x^{(i)}$

Training a model simply means learning the best values for all the weights and the bias directly from the labeled examples.

As we all know (at least, I hope) this is achieved by minimizing a function accounting for the differences between the model predictions and the examples, e.g. the χ^2.

We are going to start with something extremely simple and familiar.
The blueprint example of literally almost every ML courses is the Housing Prices in California (which actually tells you a lot about the US).
This is ML for Astrophysics, so we're going to use another kind of linear regression:

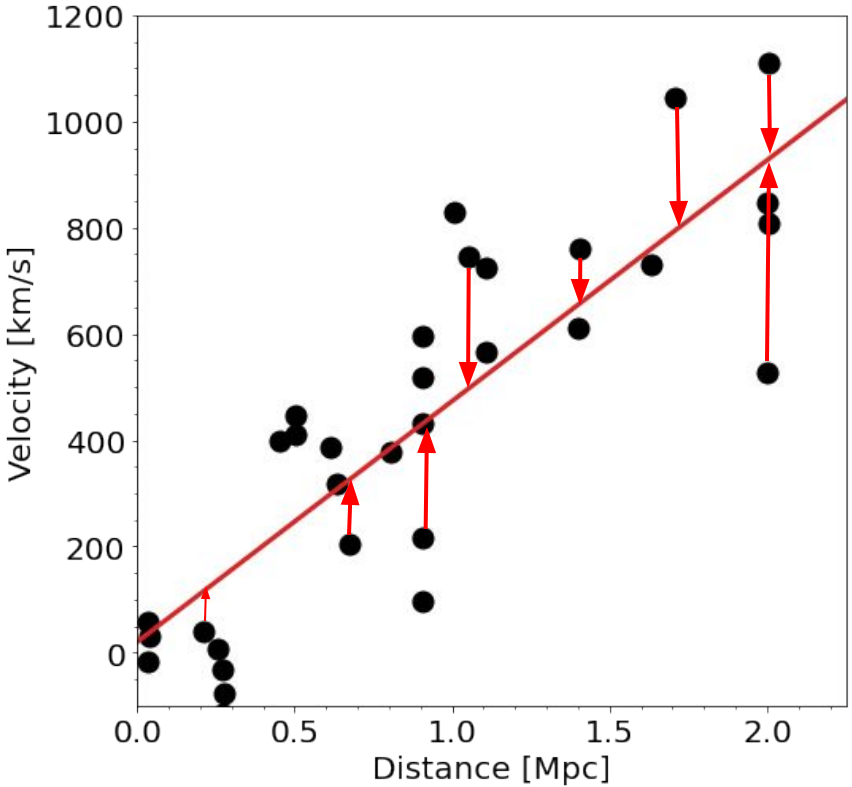


The function to minimize is called the ***cost function***.
One of the most popular choices in ML is the *mean square error* (MSE), which is the sum of the all the **squared loss** (or $\mathbf{L_2}$ **loss**).

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\mathcal{L}_i = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2$$

The goal is to find the values w_i , b that minimize $J(w_i, b)$

We are going to start with something extremely simple and familiar.
The blueprint example of literally almost every ML courses is the Housing Prices in California (which actually tells you a lot about the US).
This is ML for Astrophysics, so we're going to use another kind of linear regression:

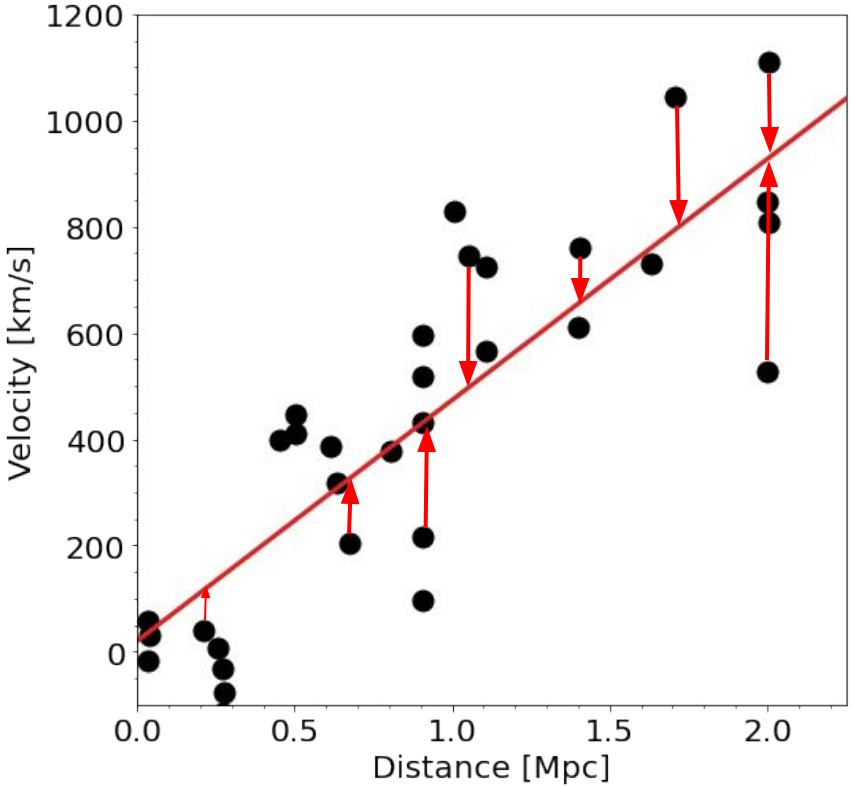


The function to minimize is called the ***cost function***.
One of the most popular choices in ML is the *mean square error* (MSE), which is the sum of the all the **squared loss** (or $\mathbf{L_2}$ **loss**).

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\mathcal{L}_i = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2$$

The goal is to find the values w_i , b that minimize $J(w_i, b)$

We are going to start with something extremely simple and familiar.
The blueprint example of literally almost every ML courses is the Housing Prices in California (which actually tells you a lot about the US).
This is ML for Astrophysics, so we're going to use another kind of linear regression:



The function to minimize is called the ***cost function***.
One of the most popular choices in ML is the *mean square error* (MSE), which is the sum of the all the **squared loss** (or $\mathbf{L_2}$ **loss**).

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\mathcal{L}_i = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2$$

The goal is to find the values w_i , b that minimize $J(w_i, b)$

MSE is commonly-used in ML, but it is neither the only practical loss function, nor the best one for all circumstances.
In fact, every problem has its own best suited cost/loss to use.

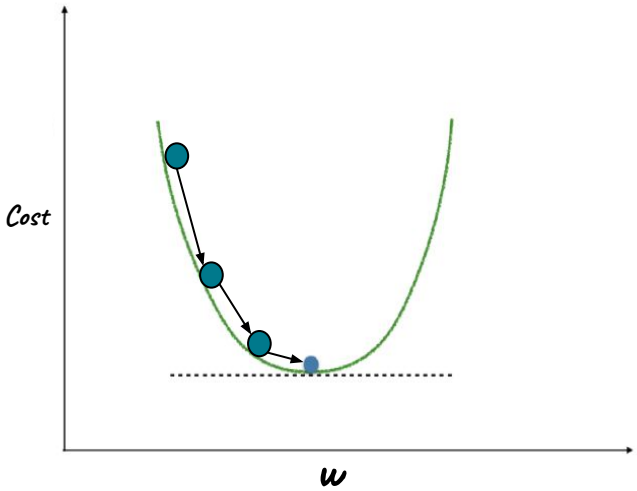
tf.keras.losses.BinaryCrossentropy
tf.keras.losses.BinaryFocalCrossentropy
tf.keras.losses.CategoricalCrossentropy
tf.keras.losses.CategoricalHinge
tf.keras.losses.CosineSimilarity
tf.keras.losses.Hinge
tf.keras.losses.Huber
tf.keras.losses.KLDivergence
tf.keras.losses.LogCosh
tf.keras.losses.MeanAbsoluteError
tf.keras.losses.MeanAbsolutePercentageError
tf.keras.losses.MeanSquaredError
tf.keras.losses.MeanSquaredLogarithmicError
tf.keras.losses.Poisson
tf.keras.losses.SparseCategoricalCrossentropy
tf.keras.losses.SquaredHinge

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)

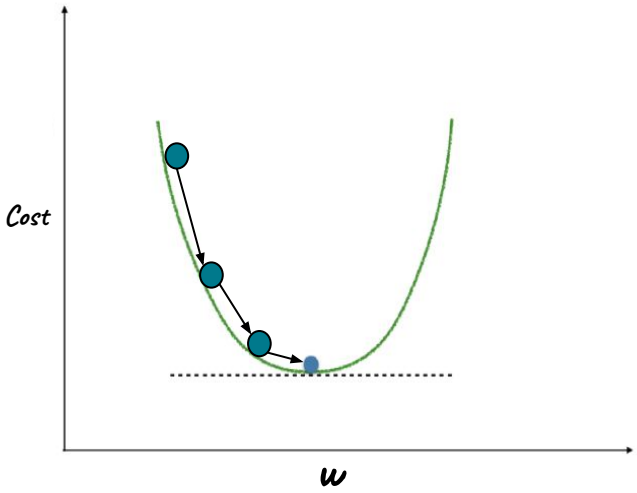


Notice that not every ML algorithm work this way (e.g. Nearest Neighbors or Random Forests do not)
However, Neural Networks work this way, so gradient descent is like Thanos, inevitable

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)



The weights are updated through an iterative process in this way:

$$w := w - \alpha \, \mathrm{d}J(w)/\mathrm{d}w$$ (the same for b)

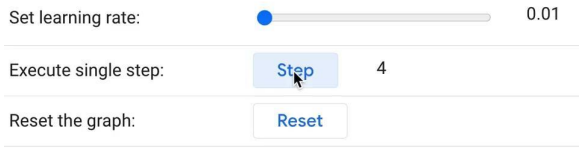
with α being the **learning rate**, a fundamental hyperparameter especially in Deep Learning.

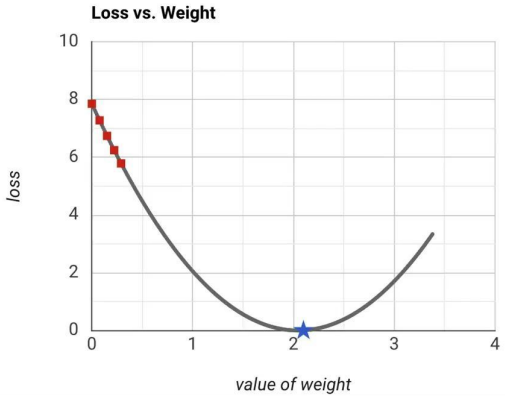
Notice that not every ML algorithm work this way (e.g. Nearest Neighbors or Random Forests do not)
However, Neural Networks work this way, so gradient descent is like Thanos, inevitable

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)





The weights are updated through an iterative process in this way:

$$w := w - \alpha \, \mathrm{d}J(w)/\mathrm{d}w$$ (the same for b)

with α being the **learning rate**, a fundamental hyperparameter especially in Deep Learning. Choosing the best learning rate is fundamental for efficiency and reaching the minimum

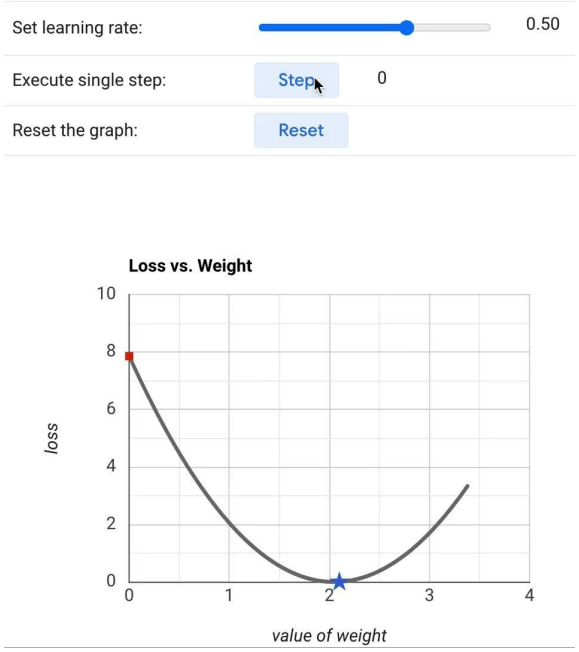
if α is too small → too slow, lots of steps to arrive to minimum

Notice that not every ML algorithm work this way (e.g. Nearest Neighbors or Random Forests do not)
However, Neural Networks work this way, so gradient descent is like Thanos, inevitable

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)



The weights are updated through an iterative process in this way:

$$w := w - \alpha \, \mathrm{d}J(w)/\mathrm{d}w$$ (the same for b)

with α being the **learning rate**, a fundamental hyperparameter especially in Deep Learning. Choosing the best learning rate is fundamental for efficiency and reaching the minimum

if α is too small → too slow, lots of steps to arrive to minimum

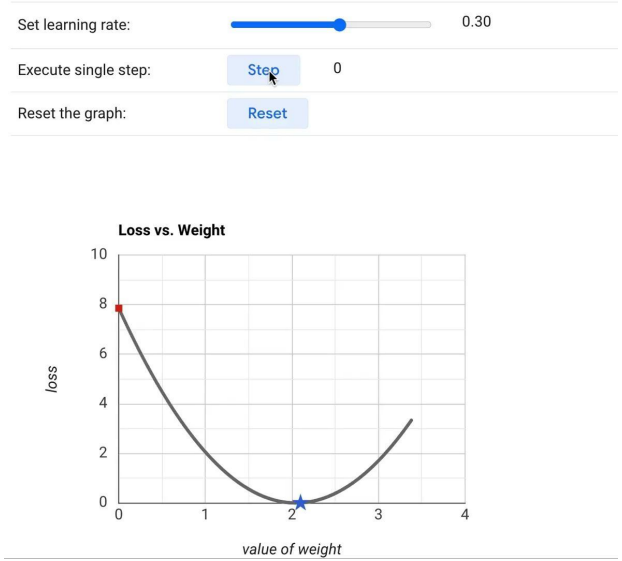
if α is too large → risk of missing the minimum and jumping over (overshoot)

Notice that not every ML algorithm work this way (e.g. Nearest Neighbors or Random Forests do not)
However, Neural Networks work this way, so gradient descent is like Thanos, inevitable

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)



The weights are updated through an iterative process in this way:

$$w := w - \alpha \, \mathrm{d}J(w)/\mathrm{d}w$$ (the same for b)

with α being the **learning rate**, a fundamental hyperparameter especially in Deep Learning. Choosing the best learning rate is fundamental for efficiency and reaching the minimum

if α is too small → too slow, lots of steps to arrive to minimum

if α is too large → risk of missing the minimum and jumping over (overshoot)

Typical values for α range between 0.01 and 1.

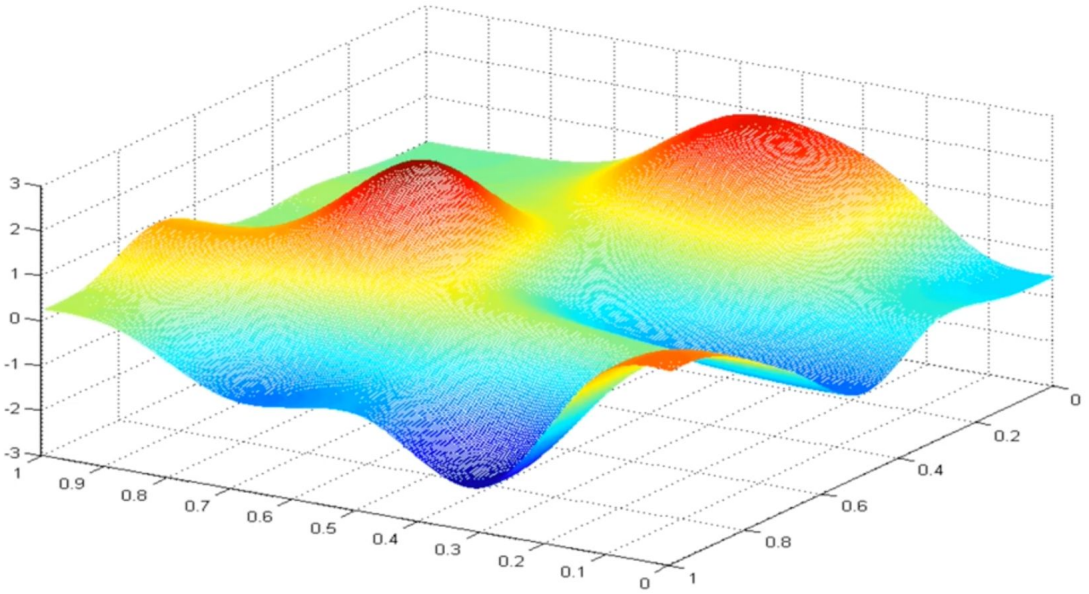
Notice that not every ML algorithm work this way (e.g. Nearest Neighbors or Random Forests do not)
However, Neural Networks work this way, so gradient descent is like Thanos, inevitable

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)

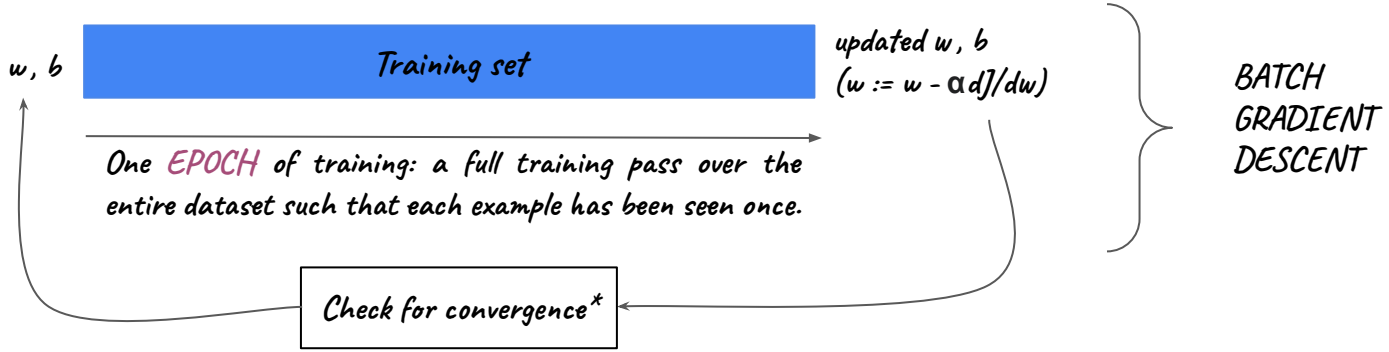
$J(w,b)$

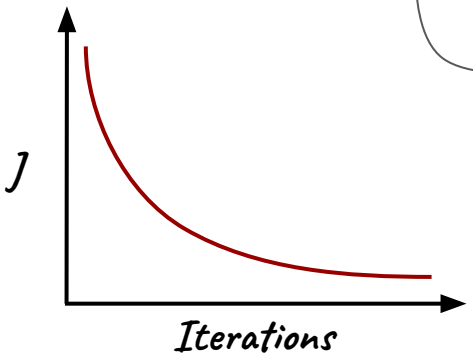


Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)





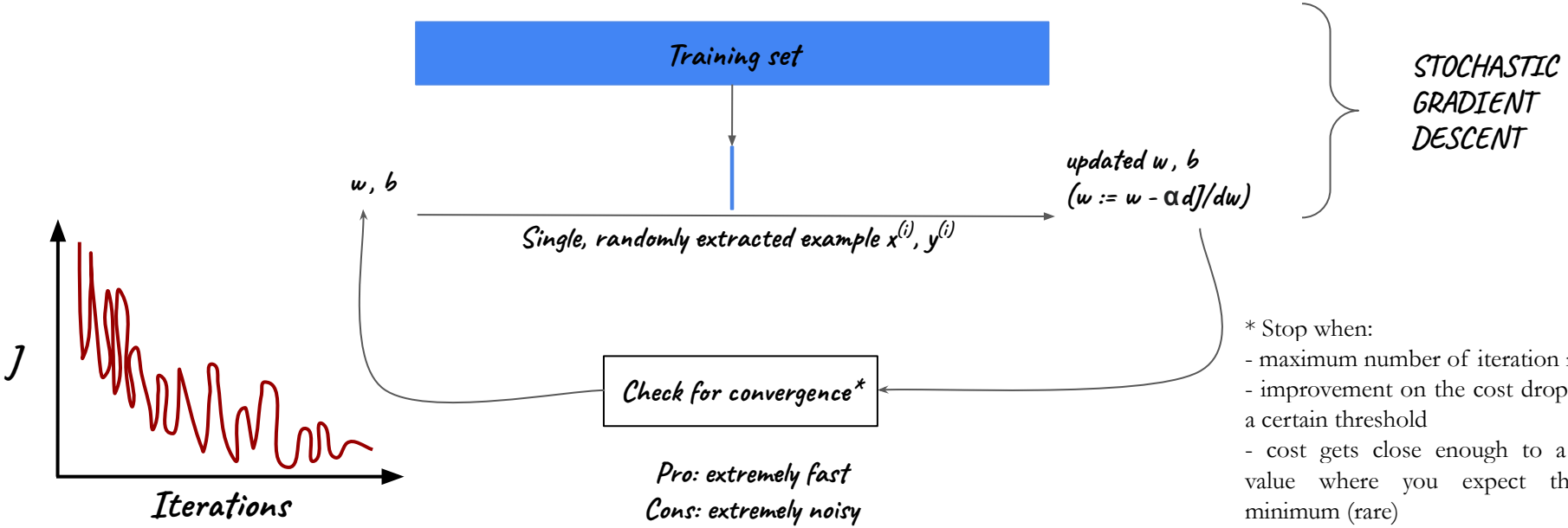
Pro: accurate measure of cost J
Cons: extremely slow for large datasets

* Stop when:
- maximum number of iteration reached
- improvement on the cost drops below a certain threshold
- cost gets close enough to a certain value where you expect the cost minimum (rare)

Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

The outline is quite simple:

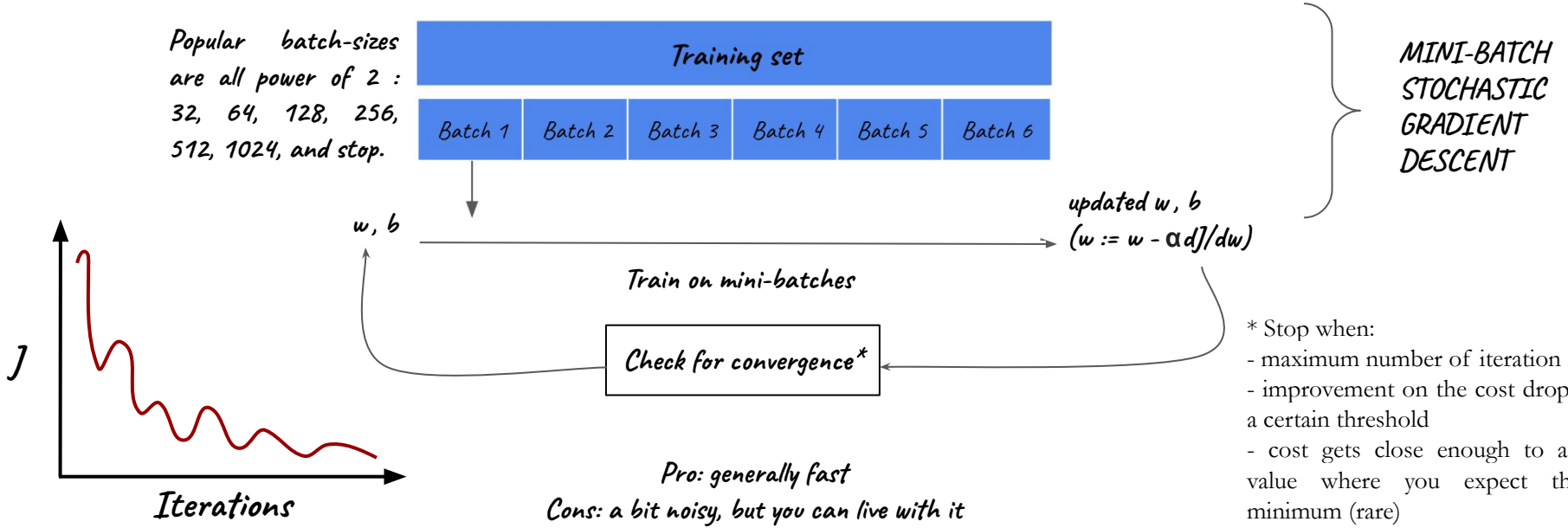
1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)



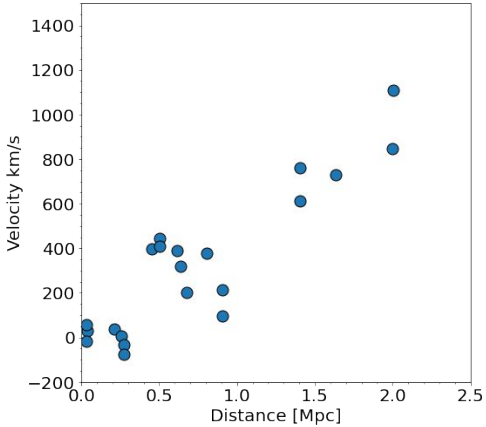
Here we do not care about sampling the cost function (it is not a posterior); we just want to get straight to the minimum, as fast as possible. This is achieved with **gradient descent**.

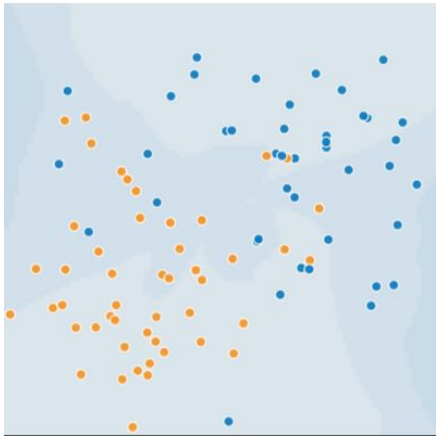
The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)

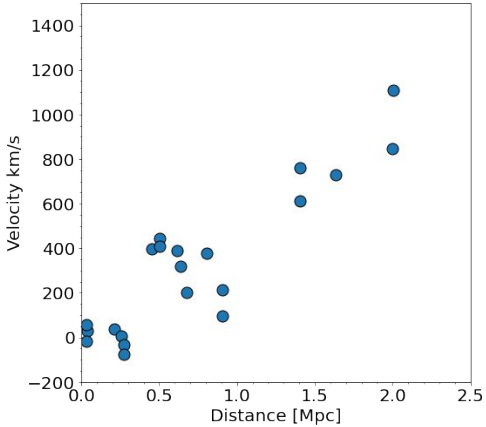


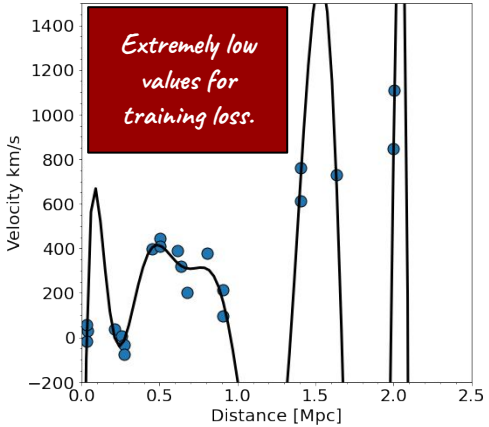
Consider these two examples, a regression and a classification.

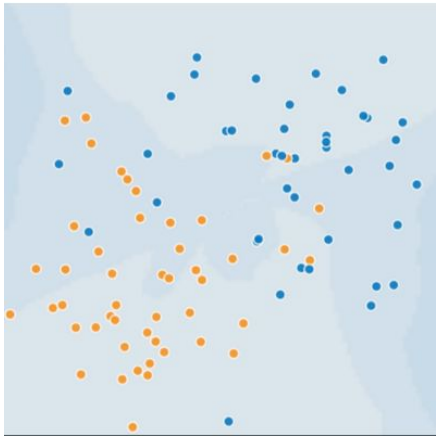




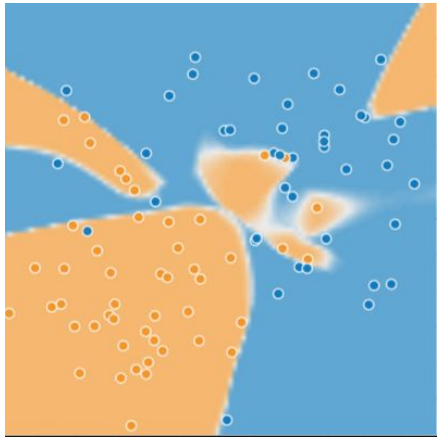
Consider these two examples, a regression and a classification.



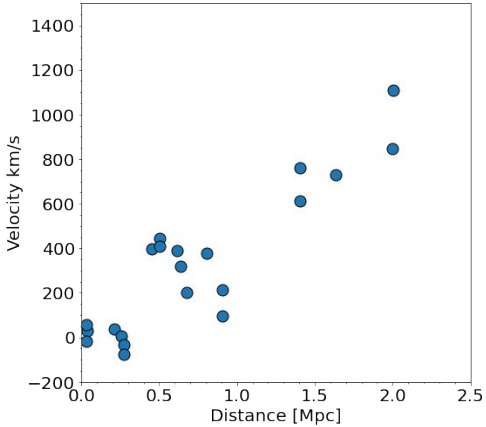


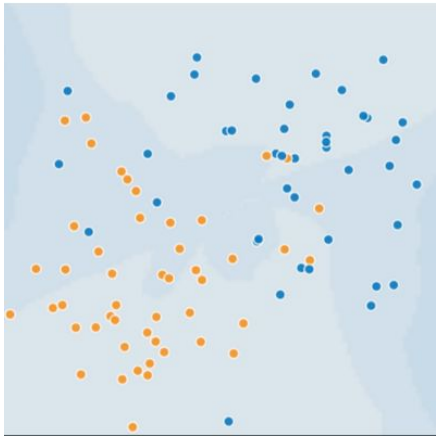


Let's fit those with (very) high-order polynomials

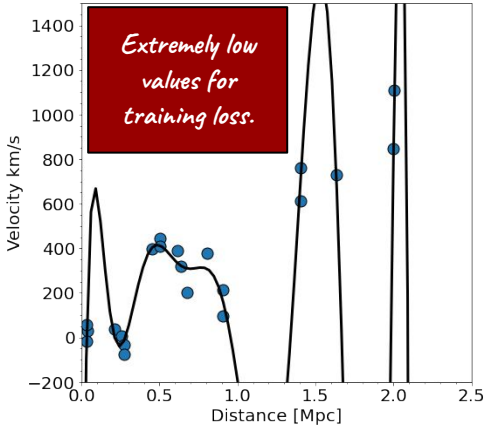


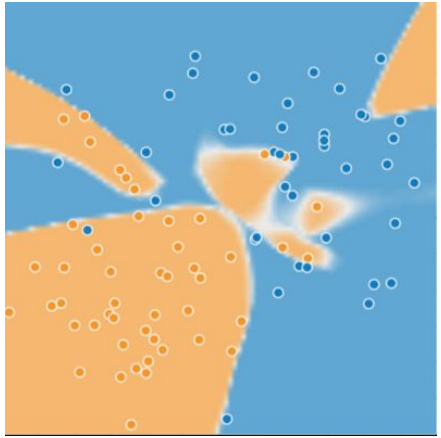
Consider these two examples, a regression and a classification.



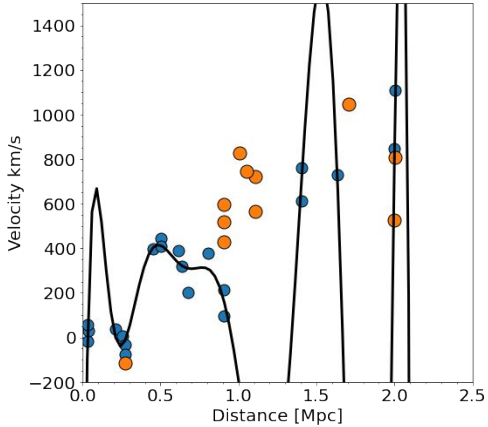


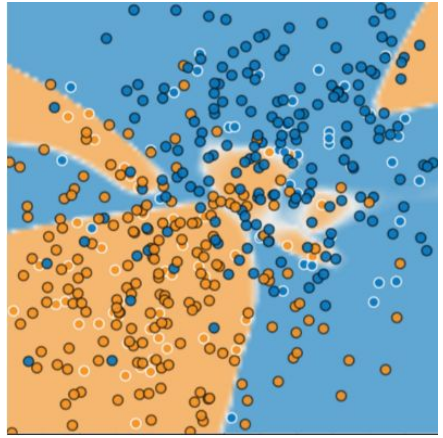
Let's fit those with (very) high-order polynomials





but then new data arrives, and those models are useless





Train/Test split

Training loss is important, must be minimized, but it's not the correct metric to assess the quality of a ML model.
Solution: split your data set into two subsets:

- **training set** → a subset to train a model.
- **test set** → a subset to test the trained model.

The model will train on, well, the **training set**, minimizing the **training loss**. Then the model will be evaluated on the **test set**, a sample that the model has never seen (and **must** never see, pay attention to e.g. data leakage).
In this way, you have a reliable metric telling how well the model adapts to new data.

80/20 split up to 10^5 examples in the test set, then increase the training set.
If you have, e.g. 1 million examples, the best split is not 80/20, is 99/1.

notice that (almost) every supervised ML application should be treated this way

Train/Dev/Test split

Training loss is important, must be minimized, but it's not the correct metric to assess the quality of a ML model.
Solution: split your data set into two subsets:

- **training set** → a subset to train a model.
- **test set** → a subset to test the trained model.

The model will train on, well, the **training set**, minimizing the **training loss**. Then the model will be evaluated on the **test set**, a sample that the model has never seen (and **must** never see, pay attention to e.g. data leakage). In this way, you have a reliable metric telling how well the model adapts to new data.

80/20 split up to 10^5 examples in the test set, then increase the training set.
If you have, e.g. 1 million examples, the best split is not 80/20, is 99/1.

Sometimes you could not avoid multiple runs on the same model, i.e. to optimize the hyperparameters. You could in principle do those multiple runs on the training set and measure the test losses, but that is at risk of generalizing too much on that test set (*"wear-out"*).

notice that (almost) every supervised ML application should be treated this way

Train/Dev/Test split

Training loss is important, must be minimized, but it's not the correct metric to assess the quality of a ML model.
Solution: split your data set into two subsets:

- **training set** → a subset to train a model.
- **test set** → a subset to test the trained model.

The model will train on, well, the **training set**, minimizing the **training loss**. Then the model will be evaluated on the **test set**, a sample that the model has never seen (and **must** never see, pay attention to e.g. data leakage). In this way, you have a reliable metric telling how well the model adapts to new data.

80/20 split up to 10^5 examples in the test set, then increase the training set.
If you have, e.g. 1 million examples, the best split is not 80/20, is 99/1.

Sometimes you could not avoid multiple runs on the same model, i.e. to optimize the hyperparameters. You could in principle do those multiple runs on the training set and measure the test losses, but that is at risk of generalizing too much on that test set (*"wear-out"*).
Solution: another split, the hold-out/cross-validation/validation/**development set**.
(most used terms are Val/Dev,
I prefer Dev but that's just me)

Same principles as above apply, e.g. 60/20/20 split up to 10^5, then increase training set.
For e.g. 1 million examples the best split is 98/1/1

notice that (almost) every supervised ML application should be treated this way

Train/Dev/Test split

Training loss is important, must be minimized, but it's not the correct metric to assess the quality of a ML model.
Solution: split your data set into two subsets:

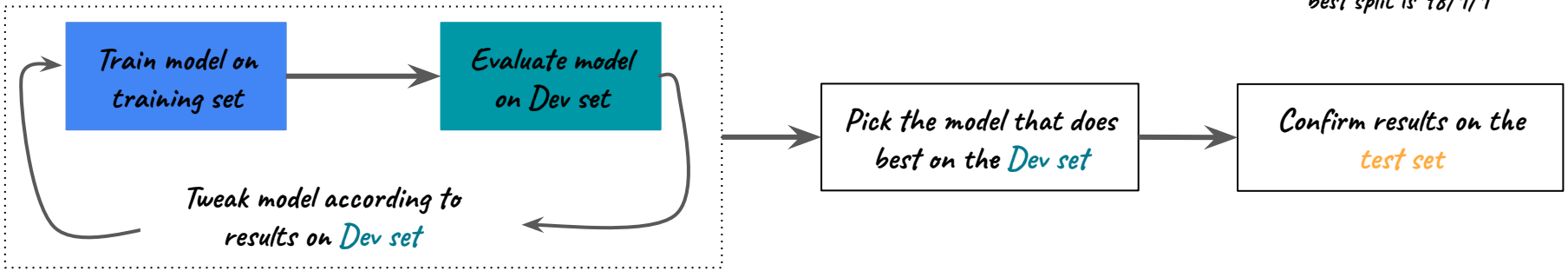
- **training set** → a subset to train a model.
- **test set** → a subset to test the trained model.

The model will train on, well, the **training set**, minimizing the **training loss**. Then the model will be evaluated on the **test set**, a sample that the model has never seen (and **must** never see, pay attention to e.g. data leakage).
In this way, you have a reliable metric telling how well the model adapts to new data.

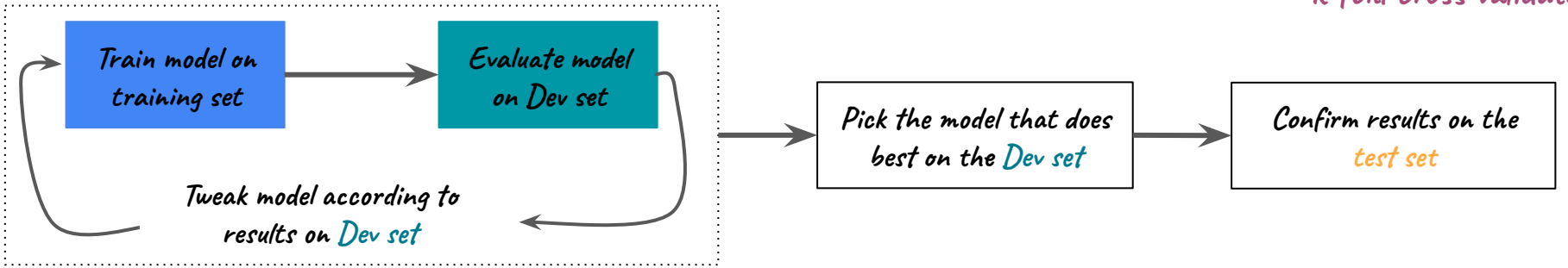
80/20 split up to 10^5 examples in the test set, then increase the training set.
If you have, e.g. 1 million examples, the best split is not 80/20, is 99/1.

Sometimes you could not avoid multiple runs on the same model, i.e. to optimize the hyperparameters. You could in principle do those multiple runs on the training set and measure the test losses, but that is at risk of generalizing too much on that test set (*"wear-out"*).
Solution: another split, the hold-out/cross-validation/validation/**development set**.
(most used terms are Val/Dev,
I prefer Dev but that's just me)
Now, the workflow becomes:

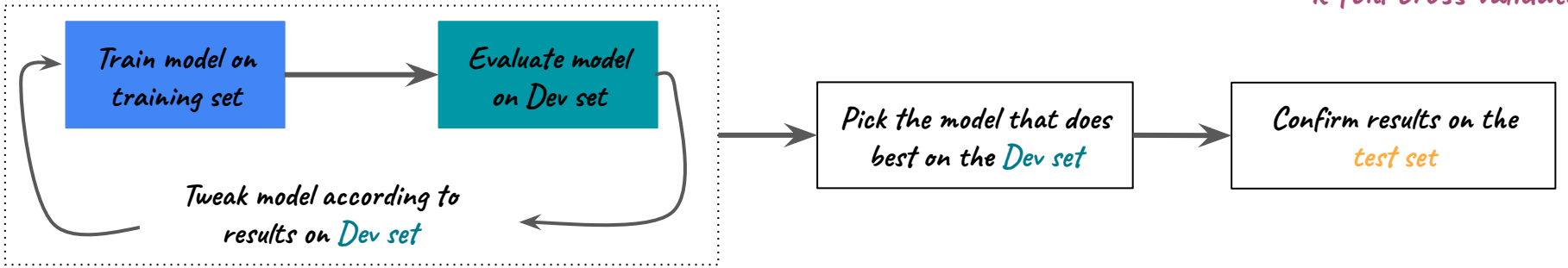
Same principles as above apply, e.g. 60/20/20 split up to 10^5, then increase training set.
For e.g. 1 million examples the best split is 98/1/1



notice that (almost) every supervised ML application should be treated this way



This workflow works smoothly as silk. However, one could argue that even in this case, you are *kinda* overfitting on that particular **Dev set**.

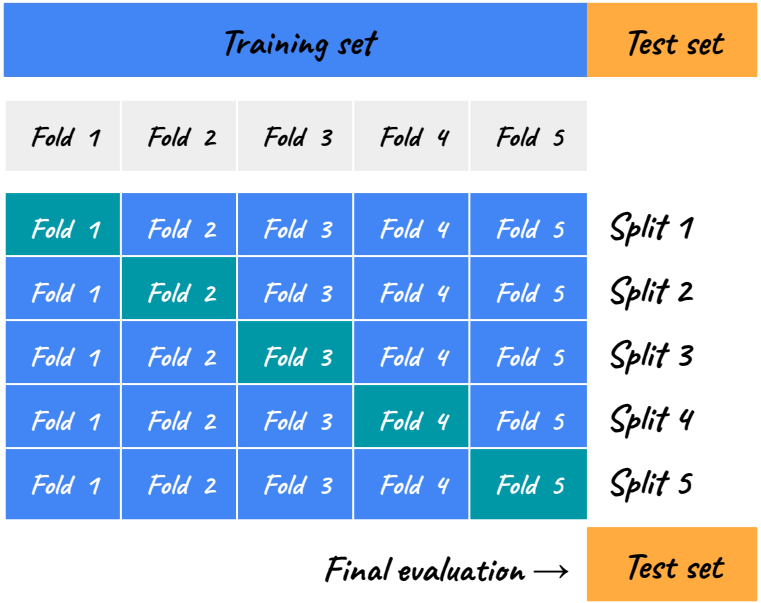


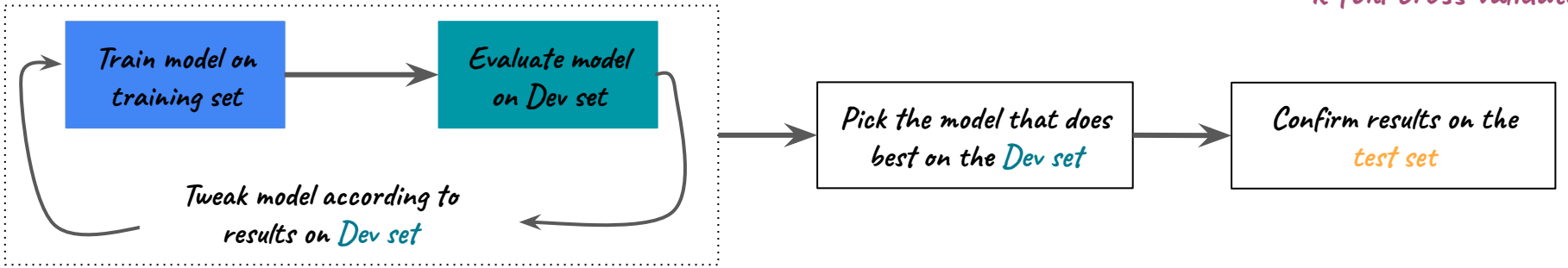
This workflow works smoothly as silk. However, one could argue that even in this case, you are *kinda* overfitting on that particular **Dev set**.

It would be better if we:

1) select a **test set**, put it aside, then split the remaining data in, e.g., *k*-subsets
2) take each unique subset as the **Dev set**, and the remaining *k–1* sets as the **training set**
3) report the average of the values computed in the loop as the *k*-fold performance measure

That's the (nested) **k-fold cross validation** technique in a nutshell.





This workflow works smoothly as silk. However, one could argue that even in this case, you are *kinda* overfitting on that particular **Dev set**.

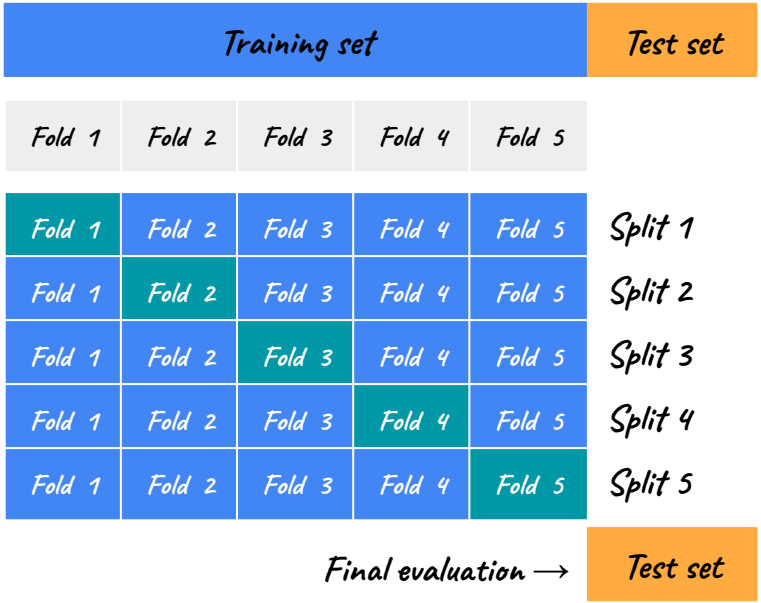
It would be better if we:

1) select a **test set**, put it aside, then split the remaining data in, e.g., *k*-subsets
2) take each unique subset as the **Dev set**, and the remaining *k–1* sets as the **training set**
3) report the average of the values computed in the loop as the *k*-fold performance measure

That's the (nested) **k-fold cross validation** technique in a nutshell.

Typical value for *k* are 5-10, depending on how much representative you need the *k*-folds to be.

It is not uncommon to find applications, especially when the model cranks do not need particularly fine-tuning, where the *k*-folds splitting is done directly at the **training** and **test** set level. It's perfectly fine, and gives a more reliable model metric than just splitting training/test altogether.



Feature engineering - representation

Machine Learning is data driven → the way you treat your data makes the difference between a good and a bad model.

(yes, even if the data itself is not that great)

Feature engineering means **transforming raw data into (useful) feature vectors**.

(expect to spend significant time doing feature engineering)

Most of ML models must represent the features as real-numbered vectors, since the feature values must be multiplied by the model weights.

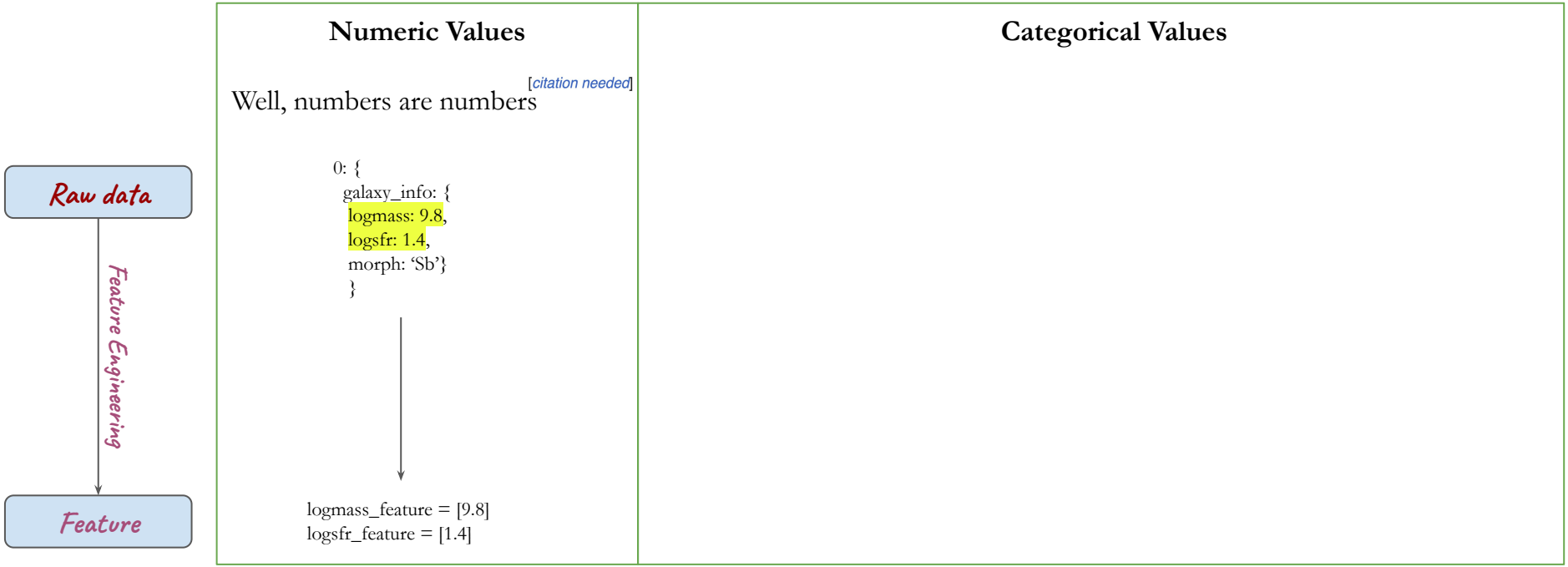
Machine Learning is data driven → the way you treat your data makes the difference between a good and a bad model.

(yes, even if the data itself is not that great)

Feature engineering means **transforming raw data into (useful) feature vectors**.

(expect to spend significant time doing feature engineering)

Most of ML models must represent the features as real-numbered vectors, since the feature values must be multiplied by the model weights.



Machine Learning is data driven → the way you treat your data makes the difference between a good and a bad model.

(yes, even if the data itself is not that great)

Feature engineering means **transforming raw data into (useful) feature vectors**.

(expect to spend significant time doing feature engineering)

Most of ML models must represent the features as real-numbered vectors, since the feature values must be multiplied by the model weights.

Numeric Values

Well, numbers are numbers [citation needed]

```
0: {
  galaxy_info: {
   logmass: 9.8,
   logsfr: 1.4,
   morph: 'Sb'}
  }
```

logmass_feature = [9.8]
logsfr_feature = [1.4]

Categorical Values

One-hot encoding (or **multi-hot encoding**)

Create a binary vector for each possible categorical feature (e.g. ['E', 'S0', 'Sa', 'Sb', 'Sc']) as follows:

- for values that apply to the example, set corresponding vector elements to 1
- set all other elements to 0.

The length of this vector is equal to the number of elements in the vocabulary.

This representation is called:

- **one-hot encoding** if a single value is 1
- **multi-hot encoding** if multiple values are 1.

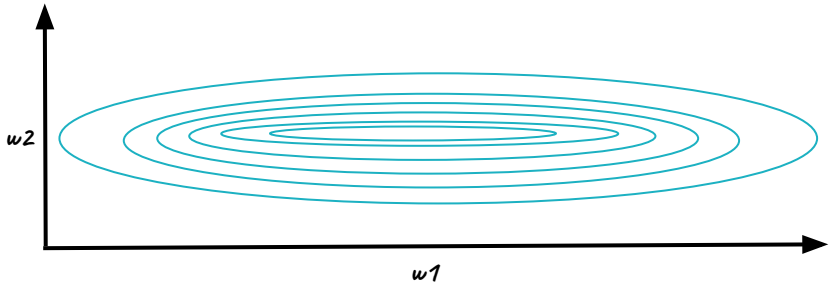
```
0: {
  galaxy_info: {
   logmass: 9.8,
   logsfr: 1.4,
   morph: 'Sb'}
  }
```

morph_feature = [0, 0, 0, 1, 0]

Especially in Astrophysics, your features might have completely different ranges.

e.g. galaxy masses from 1E8 to 1E13 Mo, star formation rates from 0.1 to 1000 Mo/yr, metallicities in 12 + log(O/H) from 8 to 9.2.

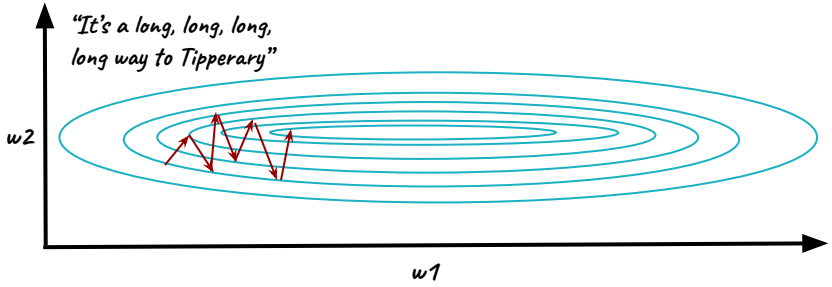
This means that your model has to run gradient descent on this:



Especially in Astrophysics, your features might have completely different ranges.

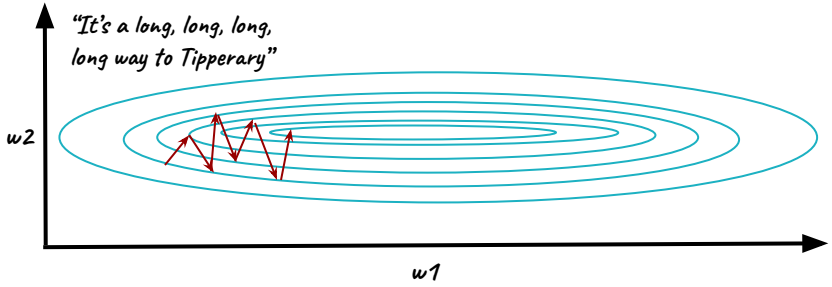
e.g. galaxy masses from 1E8 to 1E13 Mo, star formation rates from 0.1 to 1000 Mo/yr, metallicities in 12 + log(O/H) from 8 to 9.2.

This means that your model has to run gradient descent on this:

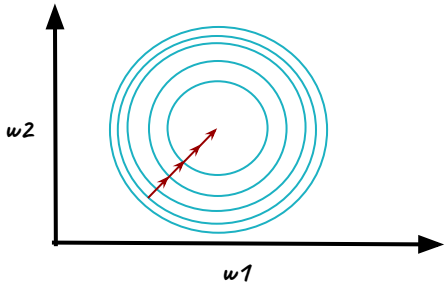


Especially in Astrophysics, your features might have completely different ranges.
e.g. galaxy masses from 1E8 to 1E13 Mo, star formation rates from 0.1 to 1000 Mo/yr, metallicities in 12 + log(O/H) from 8 to 9.2.

This means that your model has to run gradient descent on this:



Feature scaling means converting the feature values from their natural range into a standard range (e.g. 0 to 1, or -1 to +1).
Does not need to be a shared range between all features: nothing terrible happens if feature 1 ranges from -1 to +1 and feature 2 ranges from -3 to +3. However, the model will react poorly if feature 2 is instead scaled from 0 to 1000.



The two most commonly used scaler are:

Mean normalization

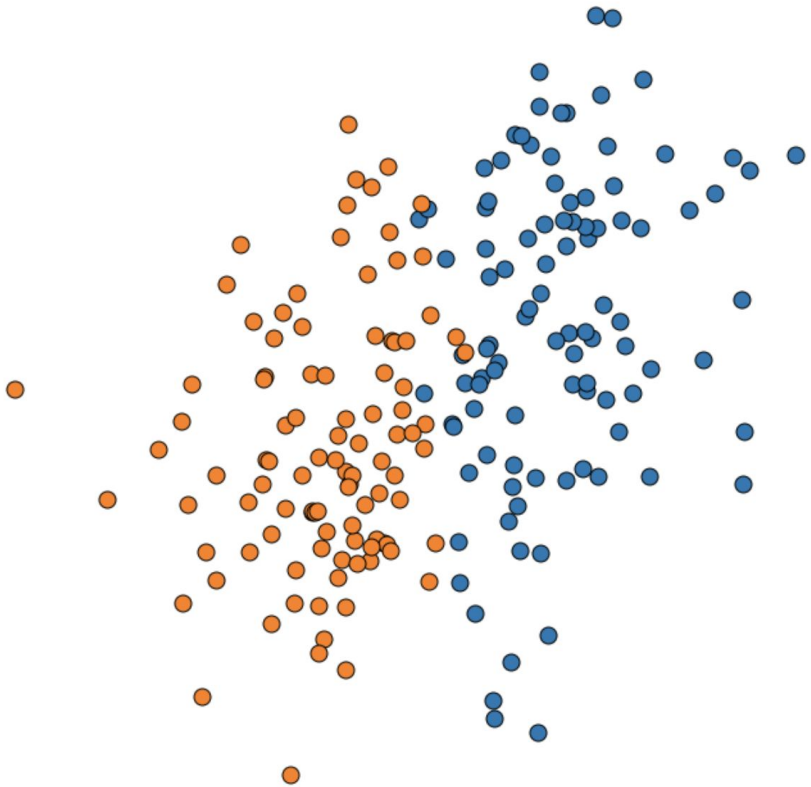
$$\frac{x-\mu}{\max x - \min x}$$

Z-score normalization

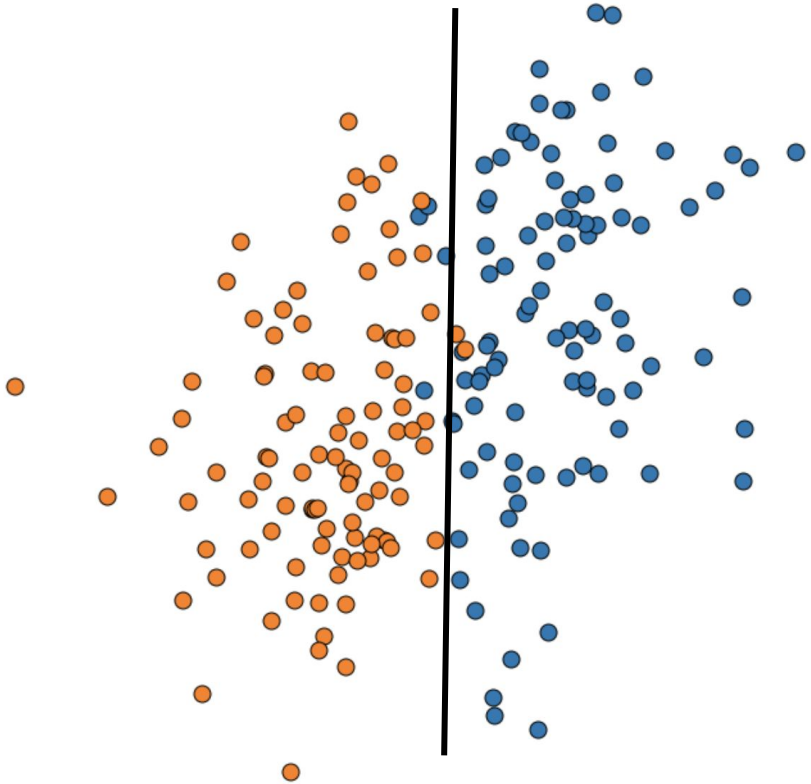
$$\frac{x-\mu}{\sigma}$$

this will help gradient descent converge more quickly, and the model to learn appropriate weights for each feature: without feature scaling, it could pay too much attention to the features having a wider range.

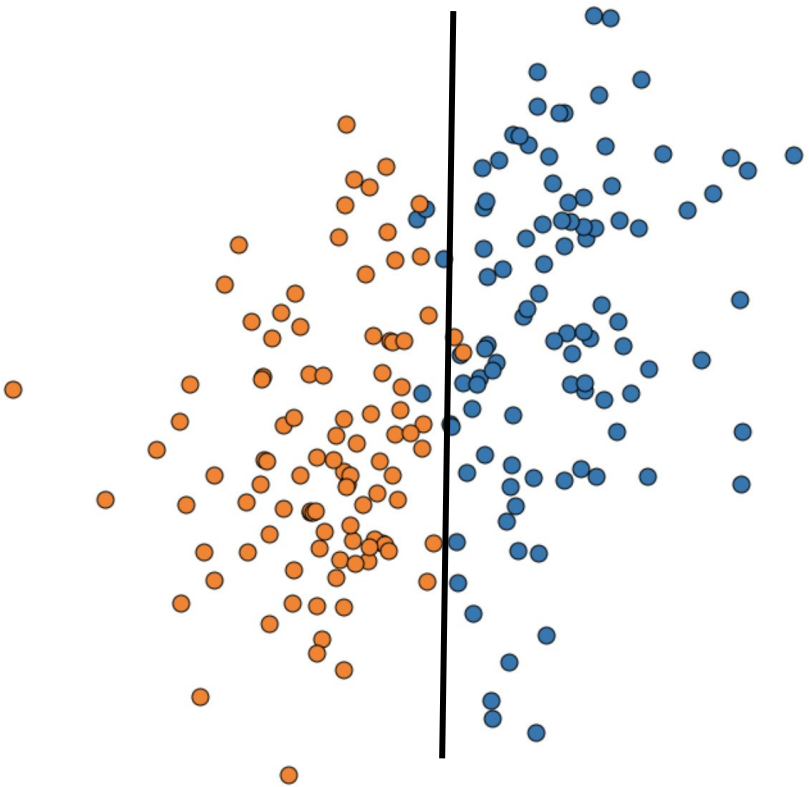
Is this a **linear** problem? ✅



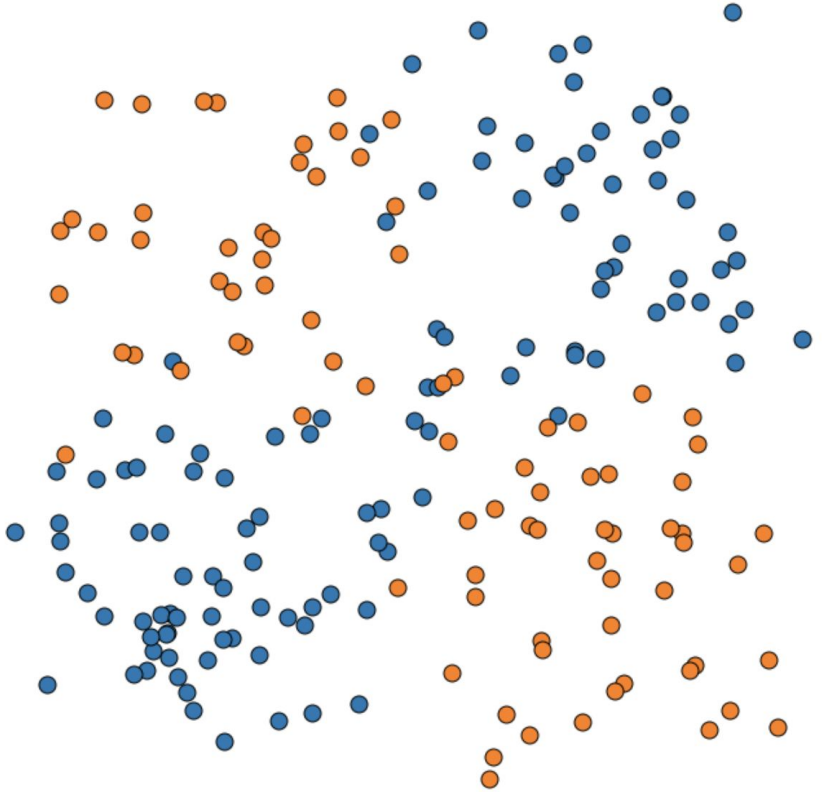
Is this a **linear** problem? ✅



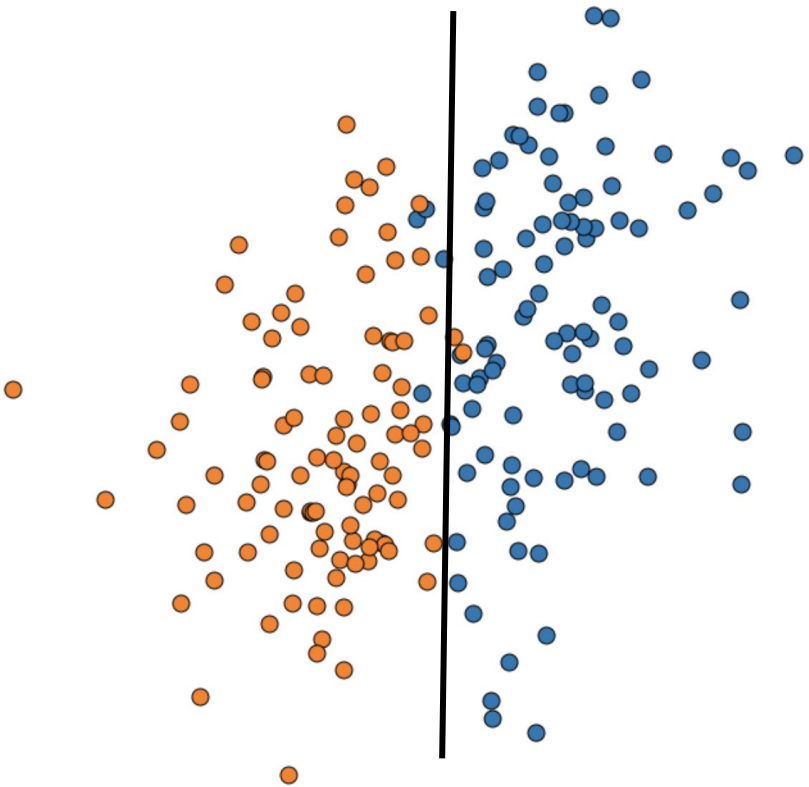
Is this a **linear** problem? ✅



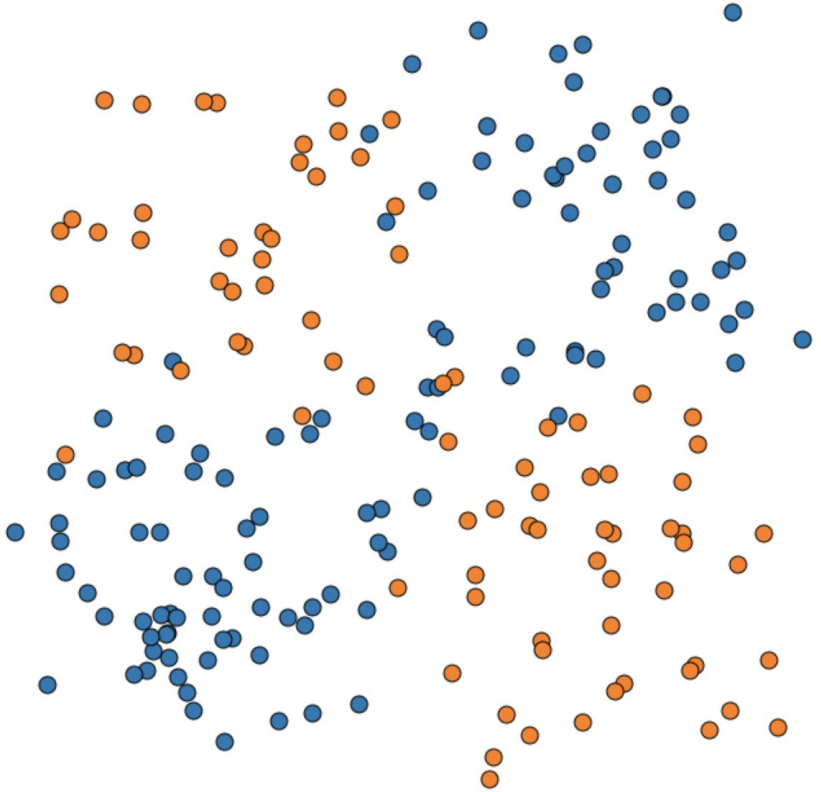
Is **this** a linear problem?



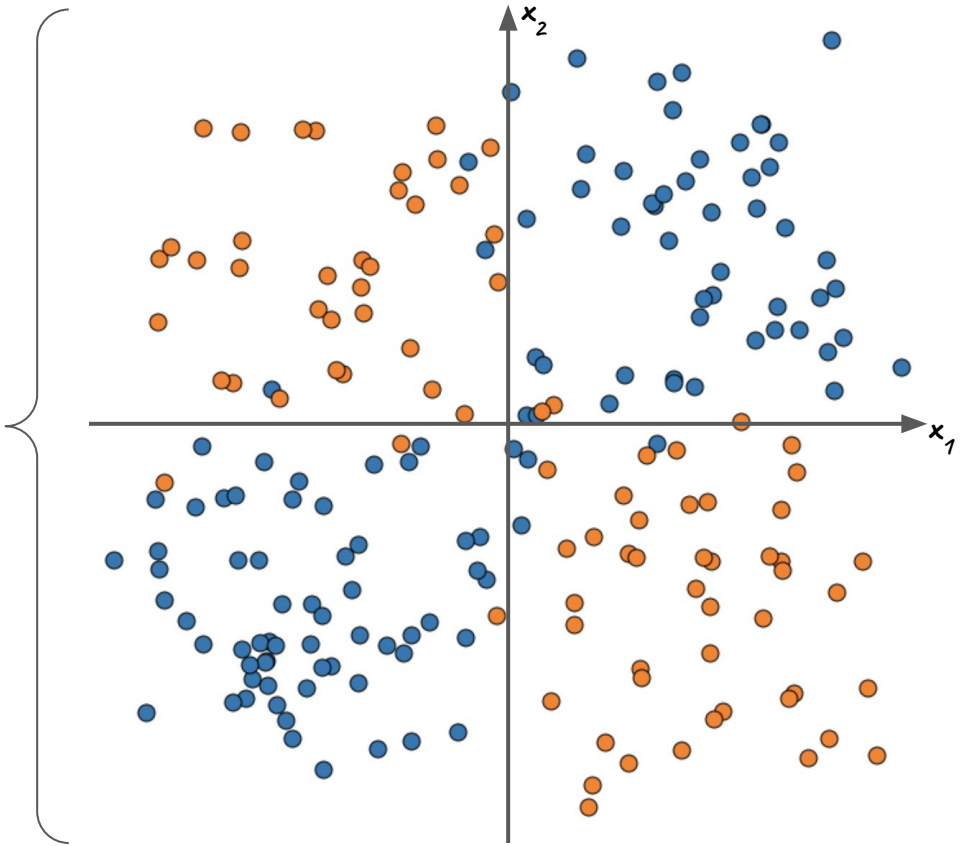
Is this a **linear** problem? ✅



Is **this** a linear problem? ❌



Is **this** a linear problem? ❌



$sgn(x_1 \cdot x_2) > 0$

$sgn(x_1 \cdot x_2) < 0$

Therefore a feature cross x_3:

$$x_3 = x_1 \cdot x_2$$

separates blue dots from orange dots.

The most general linear relation in terms of x_1, x_2, x_3

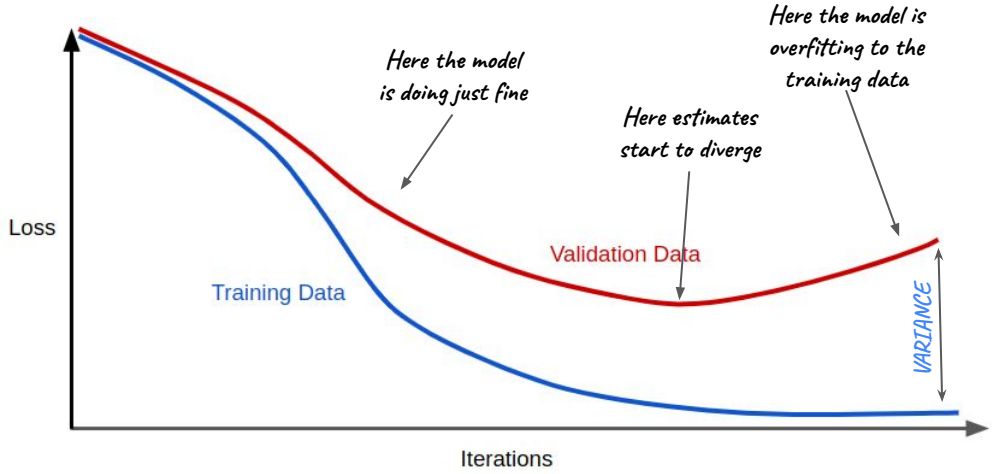
$$f(w,b) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

will find a separator, even if the problem is not linear.

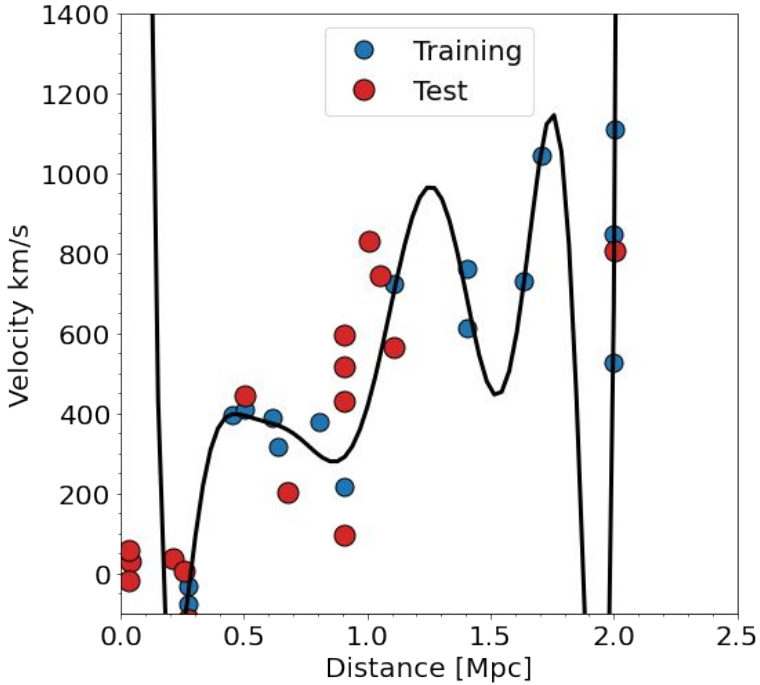
You can combine the features in every possible way, just make sure they carry useful information

* this also applies to one-hot feature vectors

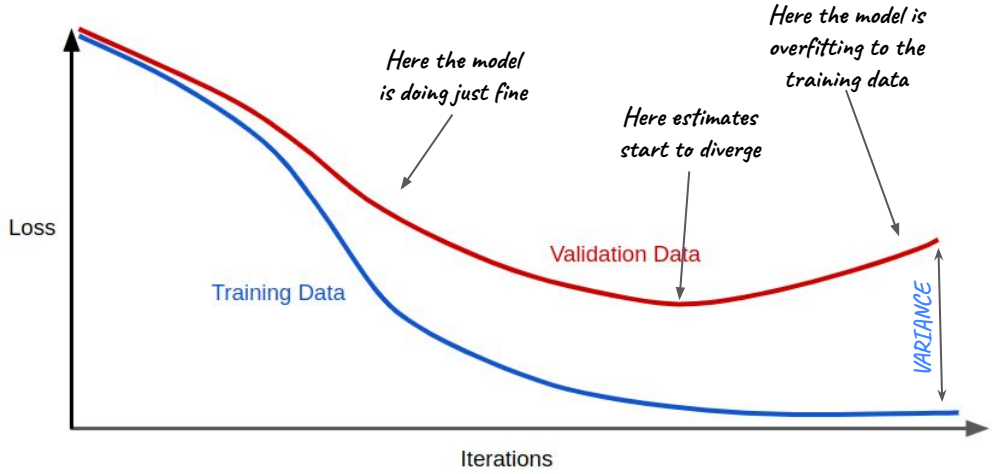
Look at this *generalization curve*:



This is a **high variance** problem (more about this later), meaning that the model fits the training set well, but does not generalize well enough to new data



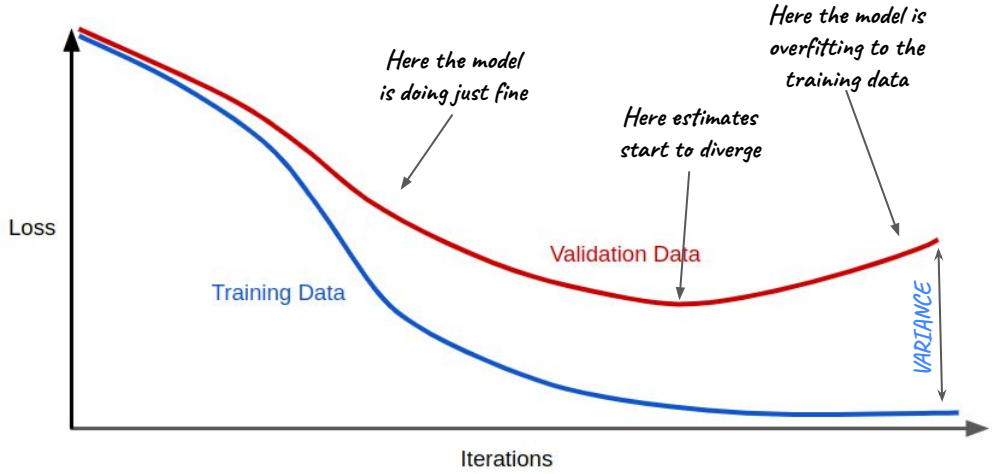
Look at this *generalization curve*:



This is a **high variance** problem (more about this later), meaning that the model fits the training set well, but does not generalize well enough to new data

Solution: keep it simple and penalize too complex models, a principle called **regularization**.
Instead of minimizing the cost function, minimize the sum of the cost and a term encompassing model complexity, the *regularization term*.
The idea is that small values for w_j, b mean simpler model, therefore less prone to overfit

Look at this *generalization curve*:



This is a **high variance** problem (more about this later), meaning that the model fits the training set well, but does not generalize well enough to new data

Solution: keep it simple and penalize too complex models, a principle called **regularization**.
Instead of minimizing the cost function, minimize the sum of the cost and a term encompassing model complexity, the *regularization term*.
The idea is that small values for w_j, b mean simpler model, therefore less prone to overfit

The new cost becomes:

$$J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

REGULARIZATION TERM

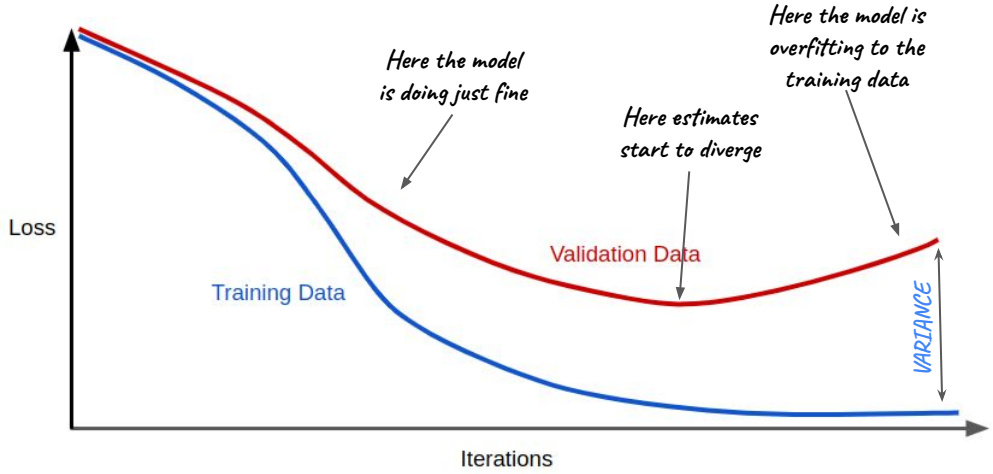
REGULARIZATION PARAMETER (>0)

λ specifies the relative importance / trade-off between
- the need to fit the data
- the need to keep w_j low, thus the model simple
→ find the right balance between the two

(also called L_2 regularization or "ridge regression")

L_2 regularization has the double effect of:
- forcing w_j to be small (close to zero)
- to distribute normally (with mean zero)

Look at this *generalization curve*:



This is a **high variance** problem (more about this later), meaning that the model fits the training set well, but does not generalize well enough to new data

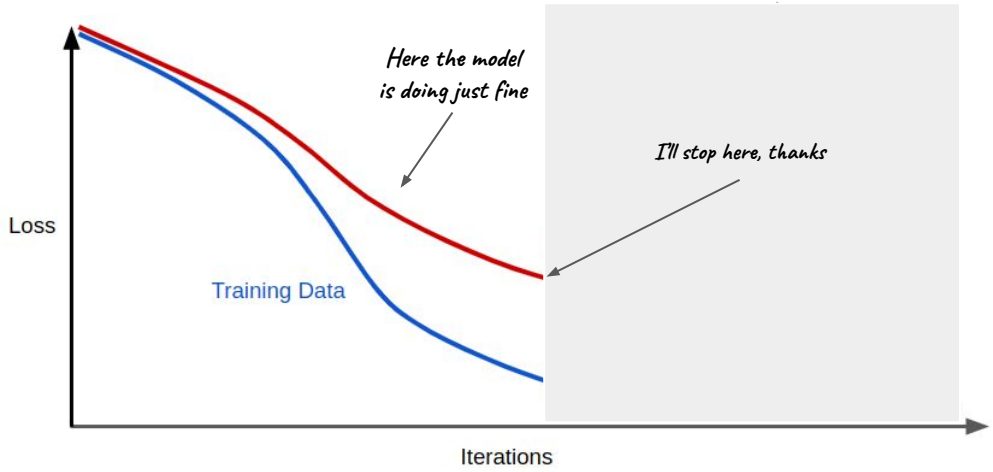
Another possibility, especially for large models: **L_1 regularization**.
L_2 forces the weights to be close to 0, but not be *exactly* 0. Some dimensions might be useless → it would be good to encourage those weights to drop to 0.
The regularization term for L_1 regularization is:

$$\frac{\lambda}{2m} \sum_{j=1}^{n} |w_j|$$

(also called "lasso* regression")

*Least Absolute Shrinkage and Selection Operator

Look at this *generalization curve*:



This is a **high variance** problem (more about this later), meaning that the model fits the training set well, but does not generalize well enough to new data

Another possible solution: **early stopping**. Stop the training before the model has the chance to overfit.

You might think it is a stupid idea, but it actually works out pretty good, and lots of NN are trained that way.

Dropout regularization: next week.

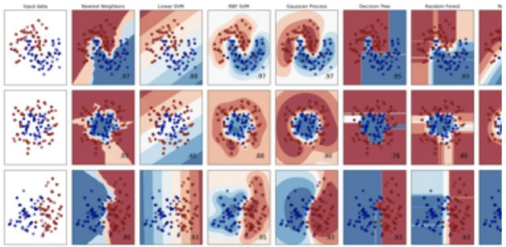
https://scikit-learn.org/stable/

Scikit-learn (sklearn) is one of the – perhaps the – most important Python library for data science and ML.
It is based on NumPy and features a wide sete of classification, regression, clustering, dimensionality reduction, hyperparameter optimization and pre-processing algorithms.
Moreover, it is full of hands-on examples literally for every possible function or module implemented.

Classification

Identifying which category an object belongs to.

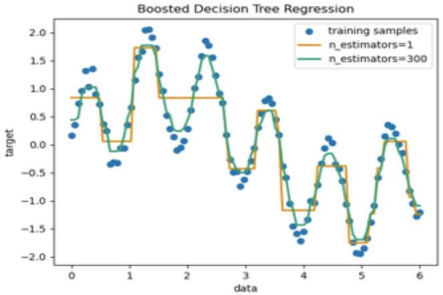
Applications: Spam detection, image recognition.
Algorithms: SVM, nearest neighbors, random forest, and more...



Regression

Predicting a continuous-valued attribute associated with an object.

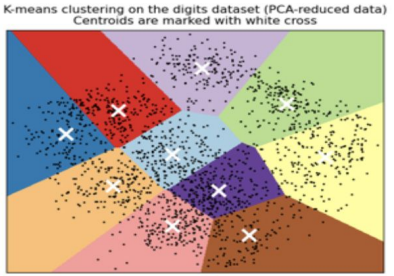
Applications: Drug response, Stock prices.
Algorithms: SVR, nearest neighbors, random forest, and more...



Clustering

Automatic grouping of similar objects into sets.

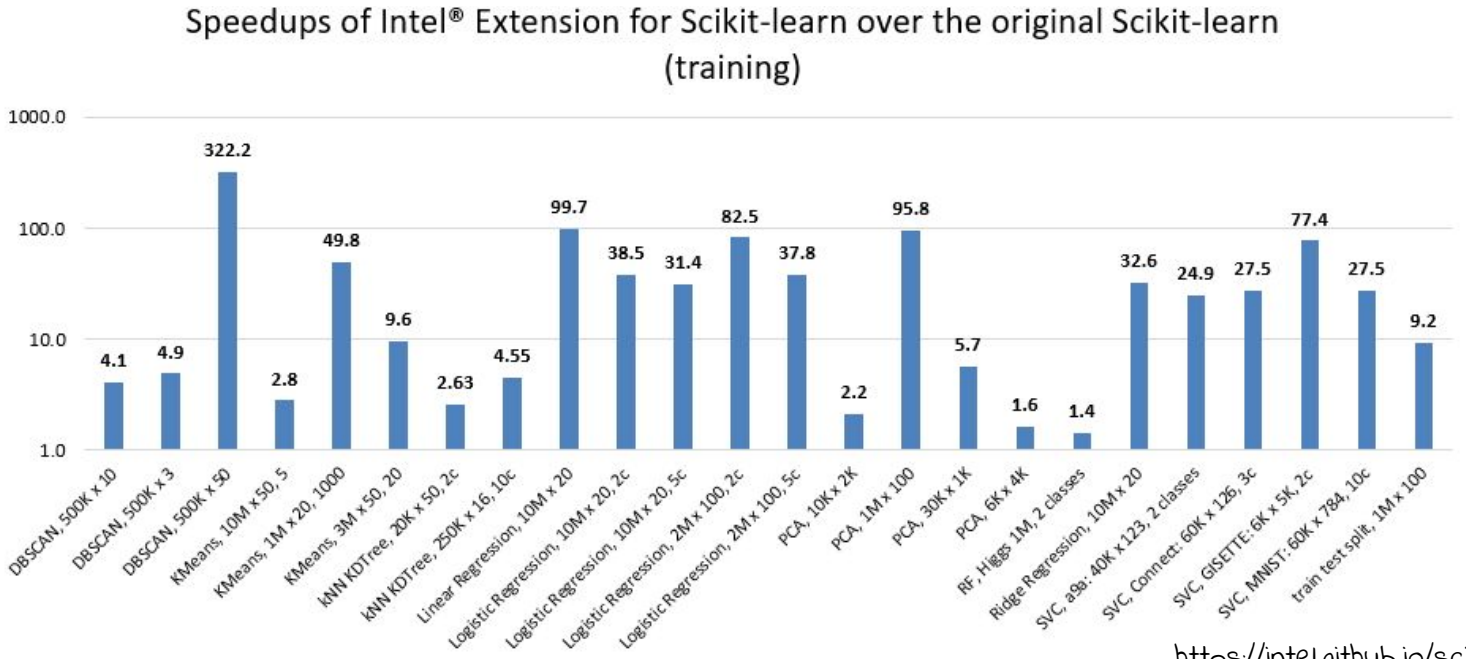
Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, and more...



https://scikit-learn.org/stable/

scikit-learn-intelex

Scikit-learn (sklearn) is one of the – perhaps the – most important Python library for data science and ML.
It is based on NumPy and features a wide sete of classification, regression, clustering, dimensionality reduction, hyperparameter optimization and pre-processing algorithms.
Moreover, it is full of hands-on examples literally for every possible function or module implemented.
If your machine runs on Intel, scikit-learn-intelex (short for the Intel® Extension for Scikit-learn) will boost the computational efficiency of your scikit-learn codes, by **a lot**.

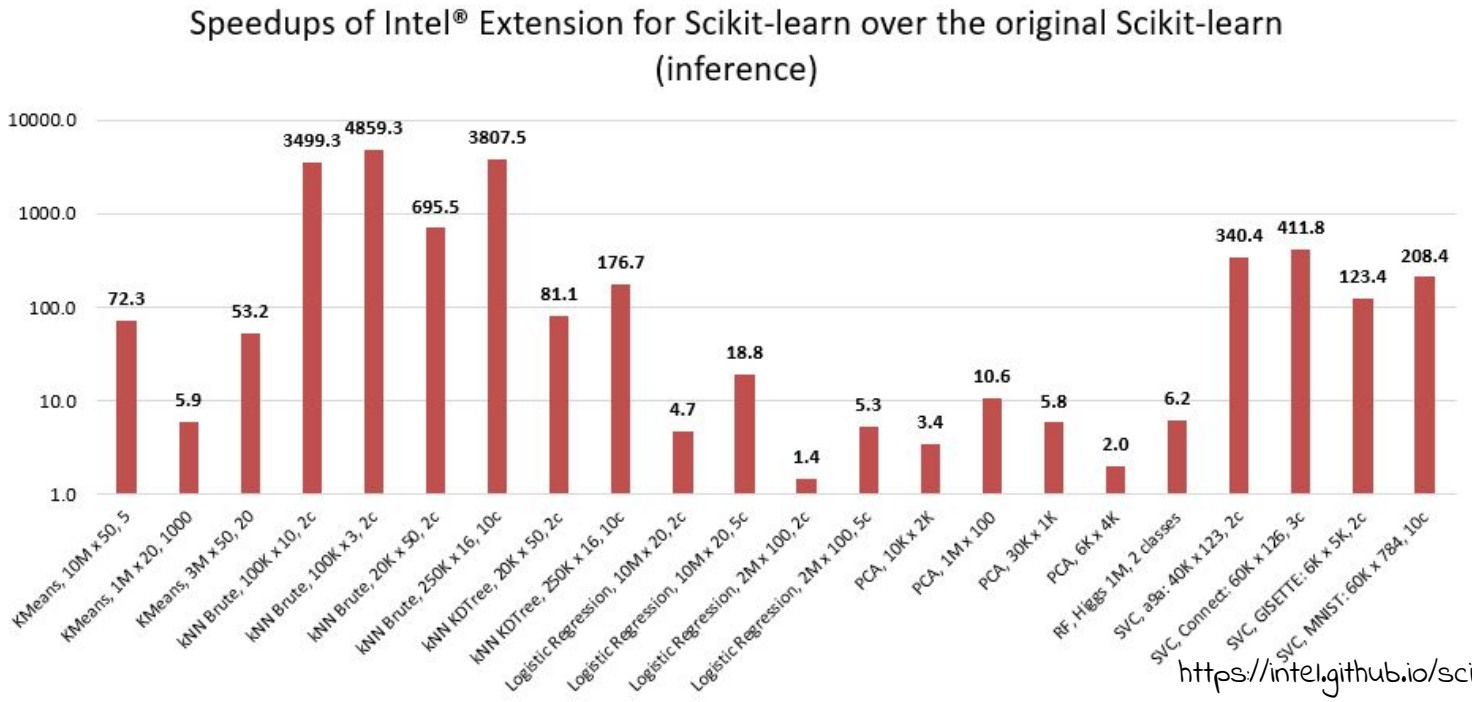


https://intel.github.io/scikit-learn-intelex/

https://scikit-learn.org/stable/

scikit-learn-intelex

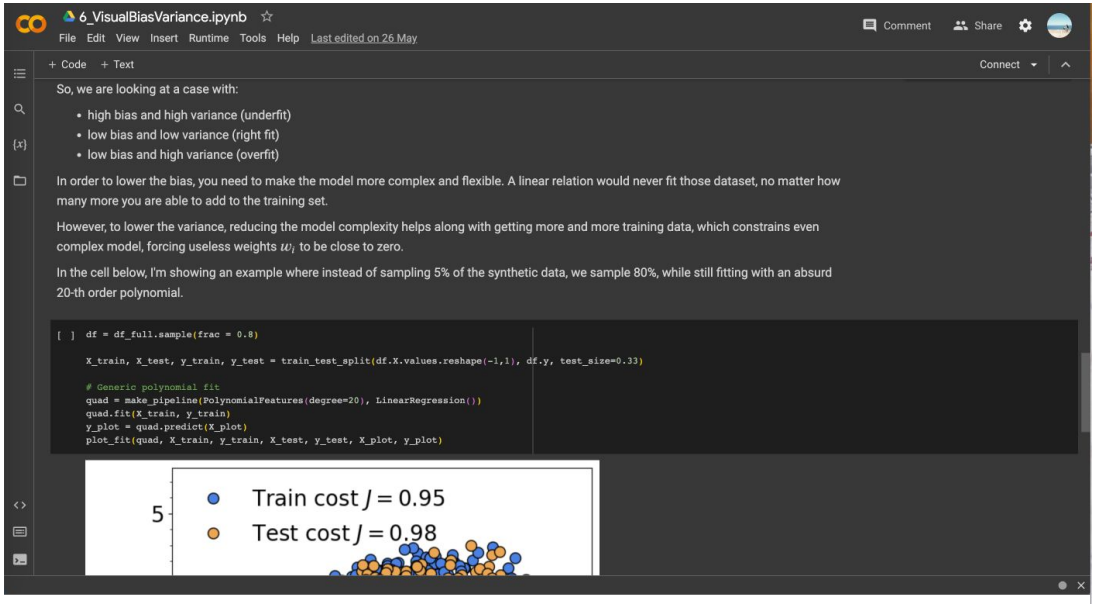
Scikit-learn (sklearn) is one of the – perhaps the – most important Python library for data science and ML.
It is based on NumPy and features a wide sete of classification, regression, clustering, dimensionality reduction, hyperparameter optimization and pre-processing algorithms.
Moreover, it is full of hands-on examples literally for every possible function or module implemented.
If your machine runs on Intel, scikit-learn-intelex (short for the Intel® Extension for Scikit-learn) will boost the computational efficiency of your scikit-learn codes, by **a lot**.



https://intel.github.io/scikit-learn-intelex/

The notebooks in this Drive folder: https://drive.google.com/drive/u/0/folders/1z1TtZMjnONYYA0bWJ_rfNUR2Ie61-AWP
and they run on Google ***Colab***, or you could either download them and run locally on you machine.
Colab, which stands for 'Colaboratory', allows you to write and execute Python codes in your browser, with:

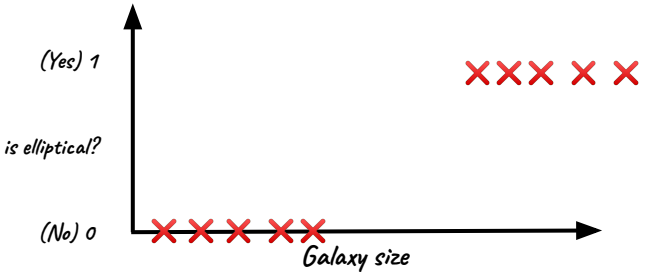
- Zero configuration required (99% True)
- Access to GPUs free of charge (well, it depends on the computational load… let's say 70% True)
- Easy sharing (100% True)



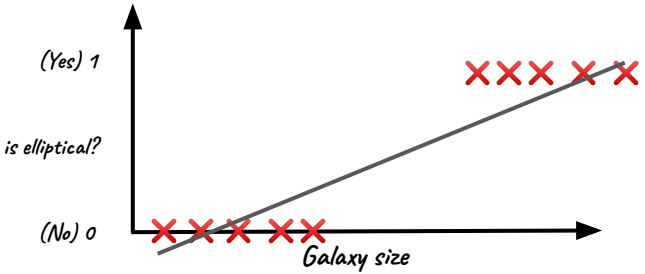
Notice that these are not exercises. They're just a way to show the things I showed in the slides in practice.

Many problems require a probability estimate as output → enters **logistic regression**, the centerpiece of classification problems.

"One must always put oneself in the position to choose between two alternatives" (Talleyrand)

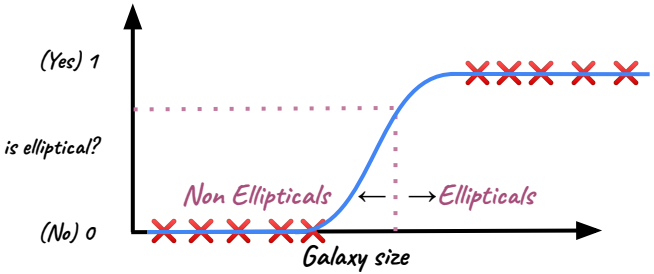


Many problems require a probability estimate as output → enters **logistic regression**, the centerpiece of classification problems.



You could in principle do linear regression on this dataset, but that would lead to terrible results, vastly influenced by outliers.

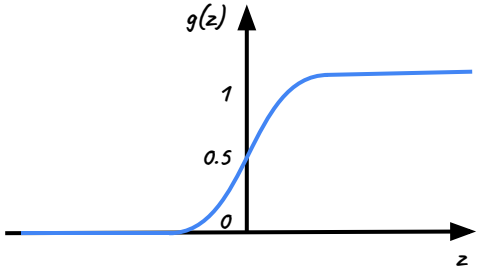
Many problems require a probability estimate as output → enters **logistic regression**, the centerpiece of classification problems.



You could in principle do linear regression on this dataset, but that would lead to terrible results, vastly influenced by outliers, no matter where you place the separator threshold.

Instead, use a **logistic** (or sigmoid) function → naturally outputs values between 0 and 1

$$g(z) = \frac{1}{1+e^{-z}}$$



And with this function, define the logistic regression model:

$$f_{\vec{w},b} = g(\vec{w} \cdot \vec{x} + b) = \frac{1}{1+e^{-(\vec{w} \cdot \vec{x} + b)}}$$

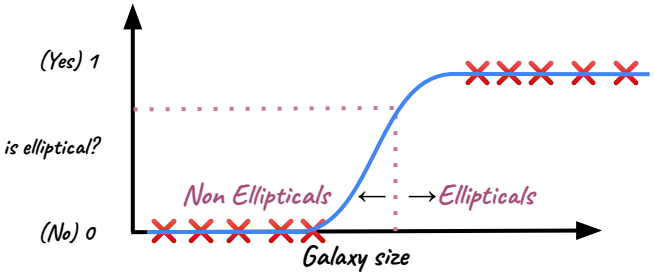
where $f_{w,b}$ gives the probability that x belongs to a certain class (e.g. if the galaxy is elliptical once given a galaxy size in kpc).
With logistic regression you train on the training set, finding the optimal values for the weights w, b.
The logistic loss function (*logloss*) evaluated on a single training example is:

$$\mathcal{L}(f_{\vec{w},b}(x^{(i)}, y^{(i)})) = \begin{cases} -\log(f_{\vec{w},b}(\vec{x}^{(i)})) & \text{if } y^{(i)} = 1 \\ -\log(1 - f_{\vec{w},b}(\vec{x}^{(i)})) & \text{if } y^{(i)} = 0 \end{cases}$$

and the cost function J is the sum of all the losses

$$\rightarrow \quad J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f_{\vec{w},b}(\vec{x}^{(i)}, y^{(i)}))$$

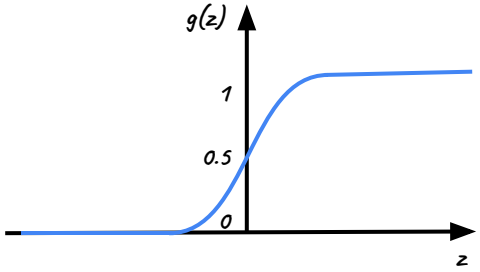
Many problems require a probability estimate as output → enters **logistic regression**, the centerpiece of classification problems.



You could in principle do linear regression on this dataset, but that would lead to terrible results, vastly influenced by outliers, no matter where you place the separator threshold.

Instead, use a **logistic** (or sigmoid) function → naturally outputs values between 0 and 1

$$g(z) = \frac{1}{1+e^{-z}}$$



And with this function, define the logistic regression model:

$$f_{\vec{w},b} = g(\vec{w} \cdot \vec{x} + b) = \frac{1}{1+e^{-(\vec{w} \cdot \vec{x} + b)}}$$

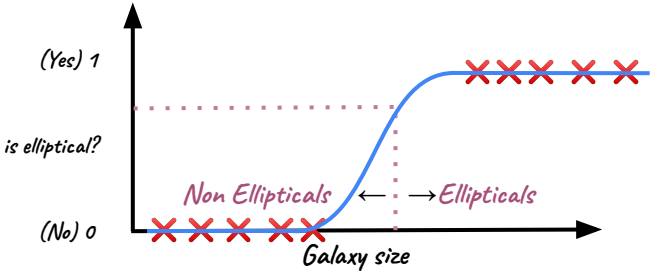
where $f_{\vec{w},b}$ gives the probability that x belongs to a certain class (e.g. if the galaxy is elliptical once given a galaxy size in kpc).
With logistic regression you train on the training set, finding the optimal values for the weights w, b.
The logistic loss function (*logloss*) evaluated on a single training example is:

$$\mathcal{L}(f_{\vec{w},b}(x^{(i)}, y^{(i)})) = -y^{(i)} \log(f_{\vec{w},b}(x^{(i)})) - (1 - y^{(i)}) \log(1 - f_{\vec{w},b}(x^{(i)}))$$

(this is technically a maximum likelihood estimator)

Classification: thresholding and decision boundaries

Logistic regression returns a probability. You can use the returned probability as is, or convert the returned probability to a binary value. To do so, a classification (or decision) threshold must be defined.

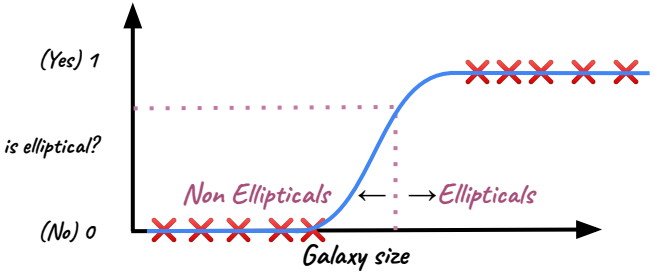


e.g. in this mock example, assuming that the blue line has been evaluated after logistic regression, I placed a 0.6 value for the threshold, to which corresponds a galaxy size value: everything on the left is labeled as non-elliptical (0), everything on the right as elliptical (1).

Therefore, I created a decision boundary between the two classes. The identification of the decision boundary between classe is a ML problem.

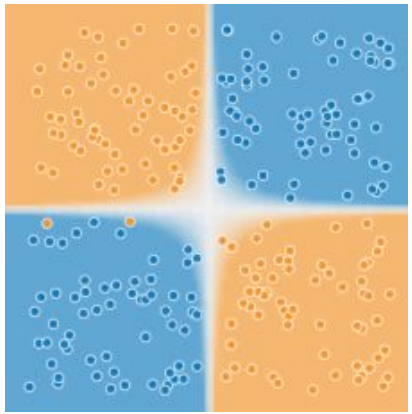
Classification: thresholding and decision boundaries

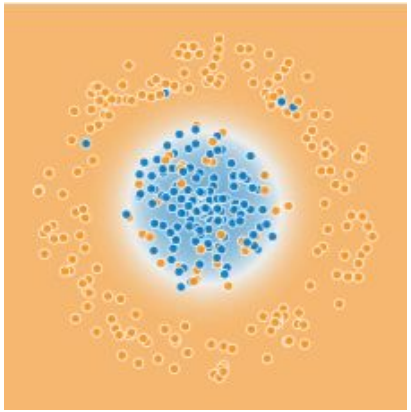
Logistic regression returns a probability. You can use the returned probability as is, or convert the returned probability to a binary value. To do so, a classification (or decision) threshold must be defined.

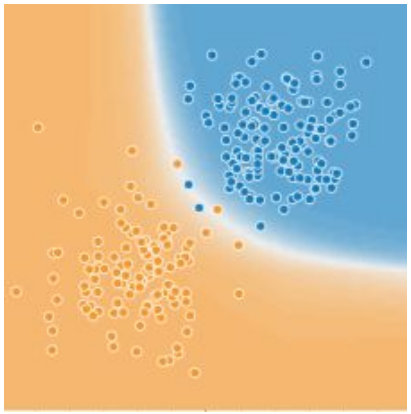


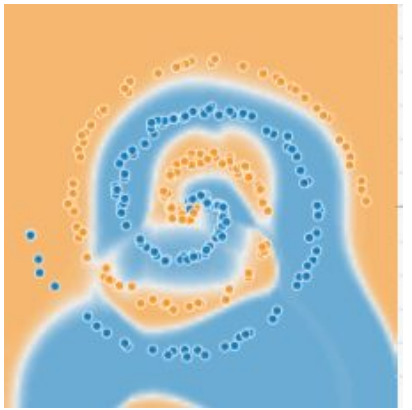
e.g. in this mock example, assuming that the blue line has been evaluated after logistic regression, I placed a 0.6 value for the threshold, to which corresponds a galaxy size value: everything on the left is labeled as non-elliptical (0), everything on the right as elliptical (1).
Therefore, I created a decision boundary between the two classes. The identification of the decision boundary between classe is a ML problem.

Decision boundary: the separator between the different classes

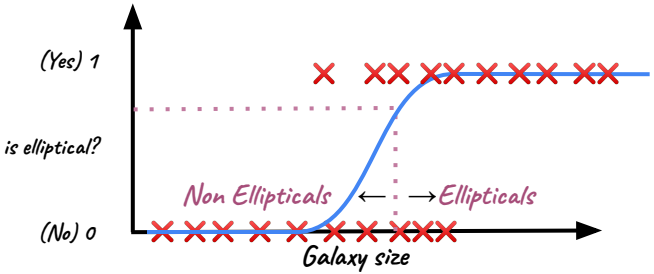






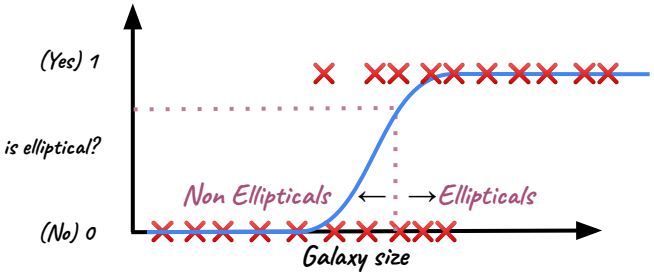


How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

How to estimate the quality of our classification model?



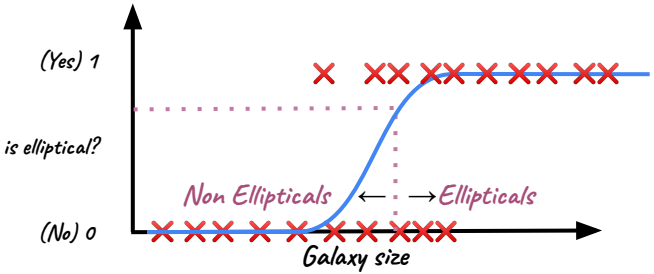
The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered:

"Confusion matrix"

False Negative (FN) - Reality: elliptical - Predicted: non-elliptical - TPs: 2	**True Positive (TP)** - Reality: elliptical - Predicted: elliptical - TPs: 8
True Negative (TN) - Reality: non-elliptical - Predicted: non-elliptical - TPs: 7	**False Positive (FP)** - Reality: non-elliptical - Predicted: elliptical - FPs: 3

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered:

"Confusion matrix"

False Negative (FN) - Reality: elliptical - Predicted: non-elliptical - TPs: 2	**True Positive (TP)** - Reality: elliptical - Predicted: elliptical - TPs: 8
True Negative (TN) - Reality: non-elliptical - Predicted: non-elliptical - TPs: 7	**False Positive (FP)** - Reality: non-elliptical - Predicted: elliptical - FPs: 3

Then define the *PRECISION* as

PRECISION = TP / (TP + FP)

("when my model said positive, was it right?")

(related to the ***purity*** of the sample)

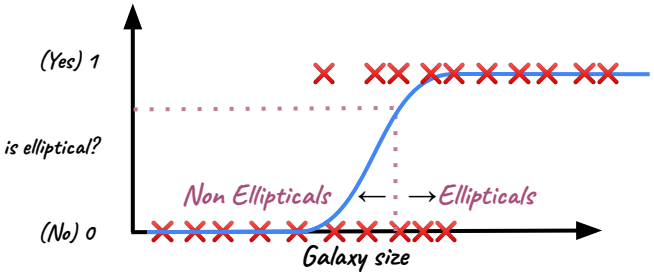
and the *RECALL* as

RECALL = TP / (TP + FN)

("out of all the possible positives, how many did the model identify? (or, how many is the model missing)")

(related to the ***completeness*** of the sample)

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered:

Then define the PRECISION as

PRECISION = TP / (TP + FP)

("when my model said positive, was it right?")

(related to the ***purity*** of the sample)

and the RECALL as

RECALL = TP / (TP + FN)

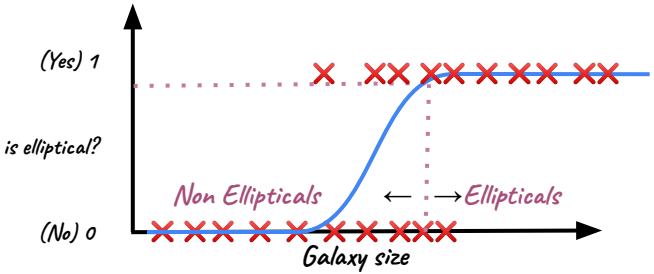
("out of all the possible positives, how many did the model identify? (or, how many is the model missing)")

(related to the ***completeness*** of the sample)

In the example above, we have:

TP = 8, FP = 3, FN = 2 → PRECISION = 0.73
RECALL = 0.80

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered:

Then define the PRECISION as

$$PRECISION = TP / (TP + FP)$$

("when my model said positive, was it right?")

(related to the ***purity*** of the sample)

and the RECALL as

$$RECALL = TP / (TP + FN)$$

("out of all the possible positives, how many did the model identify? (or, how many is the model missing)")

(related to the ***completeness*** of the sample)

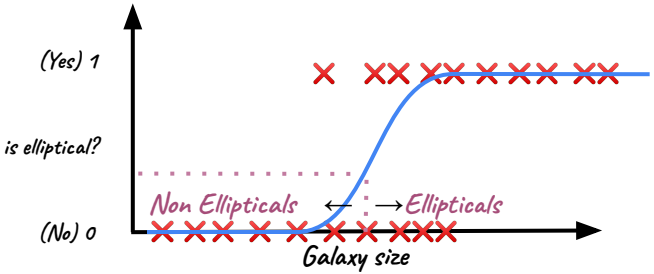
In the example above, we have:

$TP = 8,\ FP = 3,\ FN = 2 \rightarrow$ PRECISION = 0.73, RECALL = 0.80

What happens if we change the threshold?
Higher threshold (e.g. 0.9):

$TP = 7,\ FP = 2,\ FN = 3 \rightarrow$ PRECISION = 0.77, RECALL = 0.70

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered:

Then define the PRECISION as

$$PRECISION = TP / (TP + FP)$$

("when my model said positive, was it right?")

(related to the ***purity*** of the sample)

and the RECALL as

$$RECALL = TP / (TP + FN)$$

("out of all the possible positives, how many did the model identify? (or, how many is the model missing)")

(related to the ***completeness*** of the sample)

In the example above, we have:

$TP = 8,\ FP = 3,\ FN = 2 \rightarrow$ PRECISION = 0.73, RECALL = 0.80

What happens if we change the threshold?

Higher threshold (e.g. 0.9):

$TP = 7,\ FP = 2,\ FN = 3 \rightarrow$ PRECISION = 0.77, RECALL = 0.70

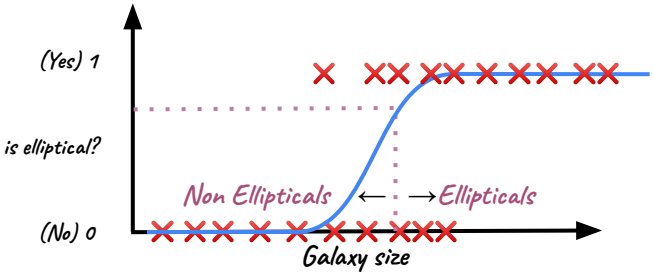
PRECISION ↑, RECALL ↓

Lower threshold (e.g. 0.4):

$TP = 9,\ FP = 4,\ FN = 1 \rightarrow$ PRECISION = 0.69, RECALL = 0.90

PRECISION ↓, RECALL ↑

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered:

Then define the PRECISION as

$$PRECISION = TP / (TP + FP)$$

("when my model said positive, was it right?")

(related to the ***purity*** of the sample)

and the RECALL as

$$RECALL = TP / (TP + FN)$$

("out of all the possible positives, how many did the model identify? (or, how many is the model missing)")

(related to the ***completeness*** of the sample)

In the example above, we have:

$TP = 8,\ FP = 3,\ FN = 2 \rightarrow$ PRECISION = 0.73, RECALL = 0.80

PRECISION/RECALL TRADE-OFF!

What happens if we change the threshold?

Higher threshold (e.g. 0.9):

$TP = 7,\ FP = 2,\ FN = 3 \rightarrow$ PRECISION = 0.77, RECALL = 0.70

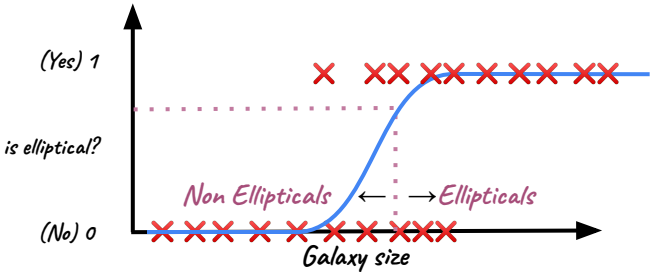
PRECISION ↑ RECALL ↓

Lower threshold (e.g. 0.4):

$TP = 9,\ FP = 4,\ FN = 1 \rightarrow$ PRECISION = 0.69, RECALL = 0.90

PRECISION ↓ RECALL ↑

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered:

Then define the *PRECISION* as

$$PRECISION = TP / (TP + FP)$$

("when my model said positive, was it right?")

(related to the ***purity*** of the sample)

and the *RECALL* as

$$RECALL = TP / (TP + FN)$$

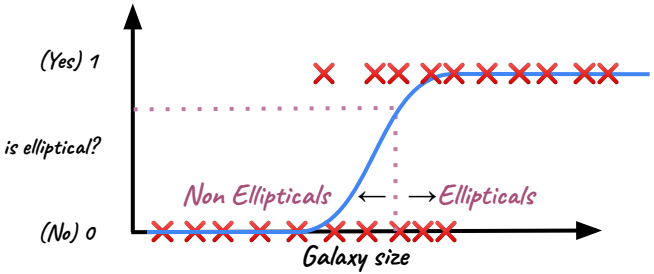
("out of all the possible positives, how many did the model identify? (or, how many is the model missing)")

(related to the ***completeness*** of the sample)

PRECISION/RECALL TRADE-OFF!

For a given model, there is a trade-off between *PRECISION* (purity) and *RECALL* (completeness).
Changing the threshold won't increase both.
To increase both, a better model must be trained

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

In order to properly assess the quality of a classification model, all the possible outcomes must be considered

Then define the *PRECISION* as

$$PRECISION = TP / (TP + FP)$$

("when my model said positive, was it right?")

(related to the ***purity*** of the sample)

and the *RECALL* as

$$RECALL = TP / (TP + FN)$$

("out of all the possible positives, how many did the model identify? (or, how many is the model missing)")

(related to the ***completeness*** of the sample)

There is a metric putting together *PRECISION* and *RECALL*, the **F1 score**

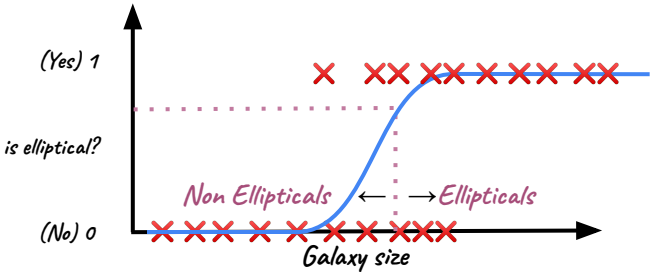
$$F1 = 2 (P \cdot R) / (P + R)$$ (the harmonic mean of precision and recall)

Threshold 0.6 ⟶ F1 = 0.763

Threshold 0.9 ⟶ F1 = 0.733

Threshold 0.4 ⟶ F1 = 0.781

How to estimate the quality of our classification model?



The first metric that should come to your mind is the accuracy, the ratio between what the model got right and all the occurrences.
It turns out that it is not a good metric, especially for class imbalanced problems, where there is a significant disparity between the number of positive and negative labels.

Precision/Recall and F1 are the most commonly used (and for the vast majority of Astrophysical application they're absolutely fine), but keep in mind that there is a whole giant zoology of metrics for classification, usually employed in Biology and Medicine.

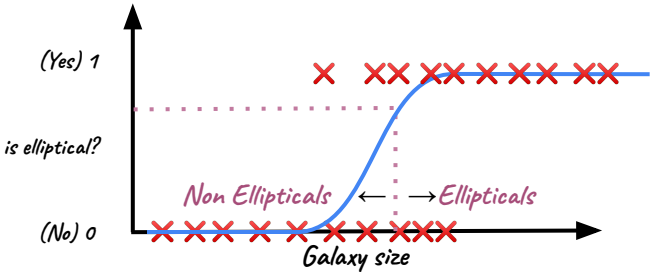
If you do not believe me, just check the leaflet (bugiardino) of e.g. Covid self-test or prescription drugs.

Sources: [4][5][6][7][8][9][10][11][12] view · talk · edit

		Predicted condition			
	Total population = P + N	Positive (PP)	Negative (PN)	Informedness, bookmaker informedness (BM) = TPR + TNR − 1	Prevalence threshold (PT) = $\frac{\sqrt{TPR \times FPR} - FPR}{TPR - FPR}$
Actual condition	Positive (P)	True positive (TP), hit	False negative (FN), type II error, miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power = $\frac{TP}{P}$ = 1 − FNR	False negative rate (FNR), miss rate = $\frac{FN}{P}$ = 1 − TPR
	Negative (N)	False positive (FP), type I error, false alarm, overestimation	True negative (TN), correct rejection	False positive rate (FPR), probability of false alarm, fall-out = $\frac{FP}{N}$ = 1 − TNR	True negative rate (TNR), specificity (SPC), selectivity = $\frac{TN}{N}$ = 1 − FPR
	Prevalence = $\frac{P}{P+N}$	Positive predictive value (PPV), precision = $\frac{TP}{PP}$ = 1 − FDR	False omission rate (FOR) = $\frac{FN}{PN}$ = 1 − NPV	Positive likelihood ratio (LR+) = $\frac{TPR}{FPR}$	Negative likelihood ratio (LR−) = $\frac{FNR}{TNR}$
	Accuracy (ACC) = $\frac{TP+TN}{P+N}$	False discovery rate (FDR) = $\frac{FP}{PP}$ = 1 − PPV	Negative predictive value (NPV) = $\frac{TN}{PN}$ = 1 − FOR	Markedness (MK), deltaP (Δp) = PPV + NPV − 1	Diagnostic odds ratio (DOR) = $\frac{LR+}{LR-}$
	Balanced accuracy (BA) = $\frac{TPR+TNR}{2}$	F_1 score = $\frac{2PPV \times TPR}{PPV+TPR} = \frac{2TP}{2TP+FP+FN}$	Fowlkes–Mallows index (FM) = $\sqrt{PPV \times TPR}$	Matthews correlation coefficient (MCC) = $\sqrt{TPR \times TNR \times PPV \times NPV} - \sqrt{FNR \times FPR \times FOR \times FDR}$	Threat score (TS), critical success index (CSI), Jaccard index = $\frac{TP}{TP+FN+FP}$

Classification: Receiver Operating Characteristic curve (ROC) and AUC

How to estimate the quality of our classification model?



Another useful diagnostic is the **ROC** curve (**receiver operating characteristic** curve), a graph showing the performance of a classification model at all classification thresholds.

The **ROC** plots two parameters:

- the True Positive Rate: *TPR = TP / (TP + FN) = RECALL*
- the False Positive Rate: *FPR = FP / (FP + TN) = FALSE ALARM RATE*

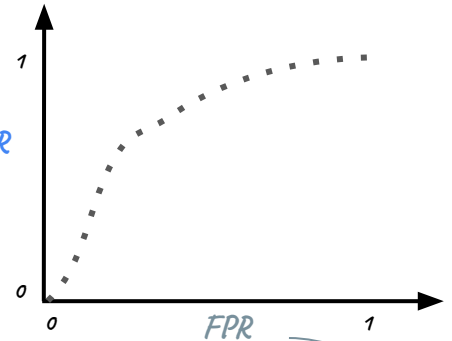
at different classification thresholds.
Lowering the threshold classifies more items as positive, thus increasing both False Positives (→ *FALSE ALARM*) and True Positives (→ *RECALL*).

Confusion matrix

True Positive (TP)	**False Negative (FN)**
False Positive (FP)	**True Negative (TN)**

TPR

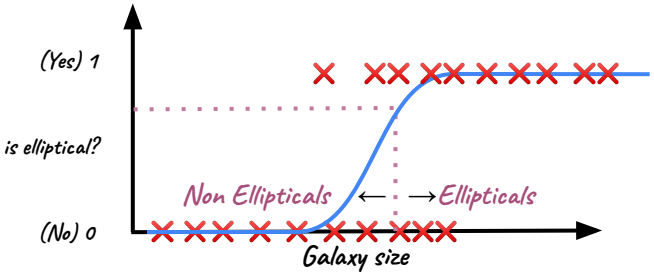
Out of all the positives, how many did the model identify?



Out of all the negatives, how many were identified as positives?

Classification: Receiver Operating Characteristic curve (ROC) and AUC

How to estimate the quality of our classification model?



Another useful diagnostic is the **ROC** curve (**receiver operating characteristic** curve), a graph showing the performance of a classification model at all classification thresholds.

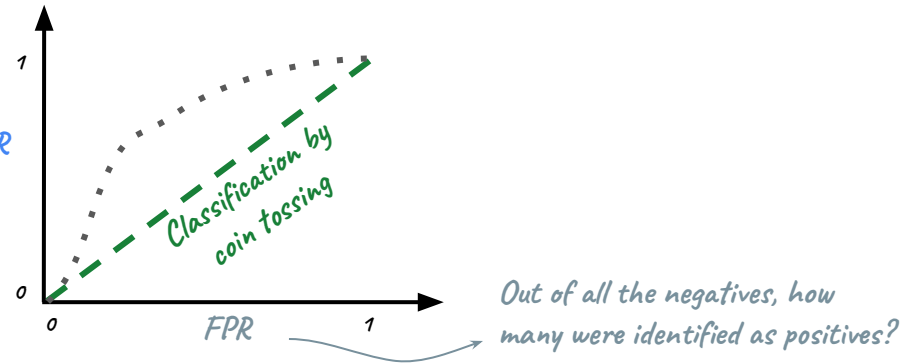
The **ROC** plots two parameters:

- the True Positive Rate: *TPR = TP / (TP + FN) = RECALL*
- the False Positive Rate: *FPR = FP / (FP + TN) = FALSE ALARM RATE*

at different classification thresholds.
Lowering the threshold classifies more items as positive, thus increasing both False Positives (→ *FALSE ALARM*) and True Positives (→ *RECALL*).

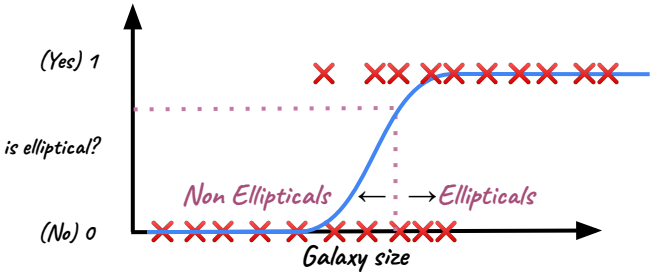
Confusion matrix

True Positive (TP)	**False Negative (FN)**
False Positive (FP)	**True Negative (TN)**



Classification: Receiver Operating Characteristic curve (ROC) and AUC

How to estimate the quality of our classification model?



Confusion matrix

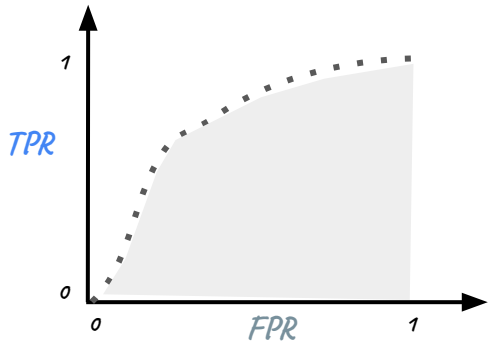
True Positive (TP)	False Negative (FN)
False Positive (FP)	True Negative (TN)

Another useful diagnostic is the **ROC** curve (**receiver operating characteristic** curve), a graph showing the performance of a classification model at all classification thresholds.

The **ROC** plots two parameters:

- the True Positive Rate: *TPR = TP / (TP + FN) = RECALL*
- the False Positive Rate: *FPR = FP / (FP + TN) = FALSE ALARM RATE*

at different classification thresholds.
Lowering the threshold classifies more items as positive, thus increasing both False Positives (→ *FALSE ALARM*) and True Positives (→ *RECALL*).

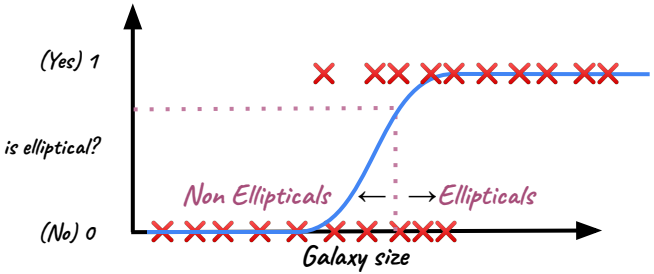


The **Area Under the ROC Curve** (**AUC**) provides an aggregate measure of performance across all possible classification thresholds.
AUC represents the probability that the model ranks a random positive example more highly than a random negative example.

AUC = 0.0 → my model is always wrong
AUC = 1.0 → my model is always right

Classification: Receiver Operating Characteristic curve (ROC) and AUC

How to estimate the quality of our classification model?



Confusion matrix

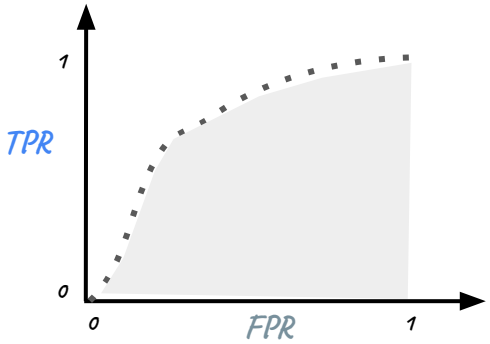
True Positive (TP)	False Negative (FN)
False Positive (FP)	True Negative (TN)

Another useful diagnostic is the **ROC** curve (**receiver operating characteristic** curve), a graph showing the performance of a classification model at all classification thresholds.

The **ROC** plots two parameters:

- the True Positive Rate: TPR = TP / (TP + FN) = RECALL
- the False Positive Rate: FPR = FP / (FP + TN) = FALSE ALARM RATE

at different classification thresholds.
Lowering the threshold classifies more items as positive, thus increasing both False Positives (→ FALSE ALARM) and True Positives (→ RECALL).

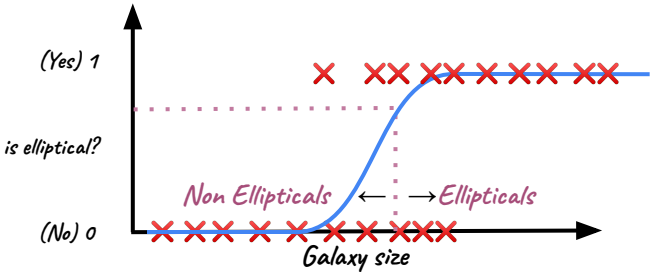


AUC has two great advantages:

- *scale-invariance*: it measures how well predictions are ranked, rather than their absolute values.
- *classification-threshold-invariance*: measures the quality of the model's predictions irrespective of what classification threshold is chosen.

Classification: Receiver Operating Characteristic curve (ROC) and AUC

How to estimate the quality of our classification model?



Confusion matrix

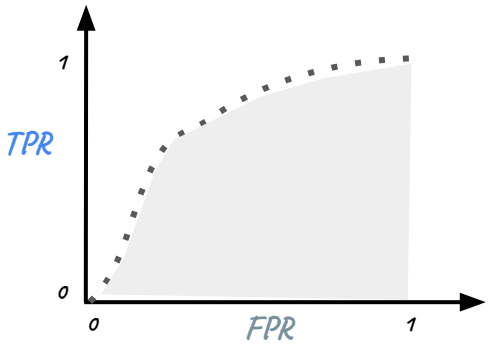
True Positive (TP)	False Negative (FN)
False Positive (FP)	True Negative (TN)

Another useful diagnostic is the **ROC** curve (**receiver operating characteristic** curve), a graph showing the performance of a classification model at all classification thresholds.

The **ROC** plots two parameters:

- the True Positive Rate: *TPR = TP / (TP + FN) = RECALL*
- the False Positive Rate: *FPR = FP / (FP + TN) = FALSE ALARM RATE*

at different classification thresholds.
Lowering the threshold classifies more items as positive, thus increasing both False Positives (→ *FALSE ALARM*) and True Positives (→ *RECALL*).



AUC also has two *dis*advantages:

- *scale-invariance*: sometimes we really do need well calibrated probability outputs, and AUC won't tell us about that.
- *classification-threshold-invariance*: sometimes it is critical to minimize one type of classification error (TPs or FNs).

What if I have multiple possible classes, and not just two? In Hubble's view, a galaxy could be an elliptical, a spiral, a lenticular or irregular.

Multiclass-Classification: One-vs-All /One and Softmax

What if I have multiple possible classes, and not just two? In Hubble's view, a galaxy could be an elliptical, a spiral, a lenticular or irregular.
A solution: **one-vs-all** (or **one-vs-rest**) algorithm, where each class probability is evaluated against all the others. That is:

1) Elliptical vs [spiral, lenticular, irregular]
2) Spiral vs [elliptical, lenticular, irregular]
3) Lenticular vs [elliptical, spiral, irregular]
4) Irregular vs [elliptical, spiral, lenticular]

In this case you are training four models.
→ the predicted class is the one with highest probability

Multiclass-Classification: One-vs-All /One and Softmax

What if I have multiple possible classes, and not just two? In Hubble's view, a galaxy could be an elliptical, a spiral, a lenticular or irregular.
A solution: **one-vs-all** (or **one-vs-rest**) algorithm, where each class probability is evaluated against all the others. That is:

1) Elliptical vs [spiral, lenticular, irregular]
2) Spiral vs [elliptical, lenticular, irregular]
3) Lenticular vs [elliptical, spiral, irregular]
4) Irregular vs [elliptical, spiral, lenticular]

In this case you are training four models.
→ the predicted class is the one with highest probability

There is also the **one-vs-one** algorithm (e.g. used in Support Vector Machines), where each possible classification is evaluated in couples, so in this example you will be evaluating #Classes * (#Classes– 1)) / 2 = (4*(4-1))/2 = 6 probabilities.
This is a fairly reasonable approach when the total number of classes is small (e.g., those four), since OvO is slightly more accurate than OvA, but becomes increasingly inefficient as the number of classes rises, or for large datasets (e.g. millions of rows) and slow models (e.g. neural networks).

Multiclass-Classification: One-vs-All /One and Softmax

What if I have multiple possible classes, and not just two? In Hubble's view, a galaxy could be an elliptical, a spiral, a lenticular or irregular.
A solution: **one-vs-all** (or **one-vs-rest**) algorithm, where each class probability is evaluated against all the others. That is:

1) Elliptical vs [spiral, lenticular, irregular]
2) Spiral vs [elliptical, lenticular, irregular]
3) Lenticular vs [elliptical, spiral, irregular]
4) Irregular vs [elliptical, spiral, lenticular]

In this case you are training four models.
→ the predicted class is the one with highest probability

There is also the **one-vs-one** algorithm (e.g. used in Support Vector Machines), where each possible classification is evaluated in couples, so in this example you will be evaluating #Classes * (#Classes– 1)) / 2 = (4*(4-1))/2 = 6 probabilities.
This is a fairly reasonable approach when the total number of classes is small (e.g., those four), since OvO is slightly more accurate than OvA, but becomes increasingly inefficient as the number of classes rises, or for large datasets (e.g. millions of rows) and slow models (e.g. neural networks).
A similar idea to OvA, widely used in Deep Learning, is the **softmax** function:

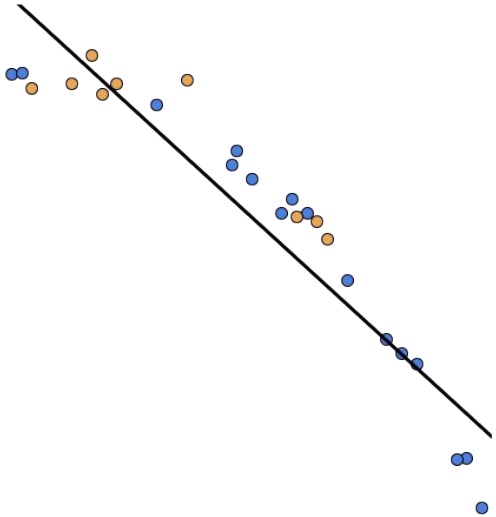
$$P(y = j|\vec{x}) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

where $z_j = \vec{w}_j \cdot \vec{x} + b_j$

And the loss becomes the **crossentropy loss**:

$$\mathcal{L}(a_1, \ldots, a_N, y) = \begin{cases} -\log a_1 & \text{if } y = 1 \\ -\log a_2 & \text{if } y = 2 \\ \ldots & \\ -\log a_N & \text{if } y = N \end{cases} = -\log a_j \quad \text{if } y = j$$

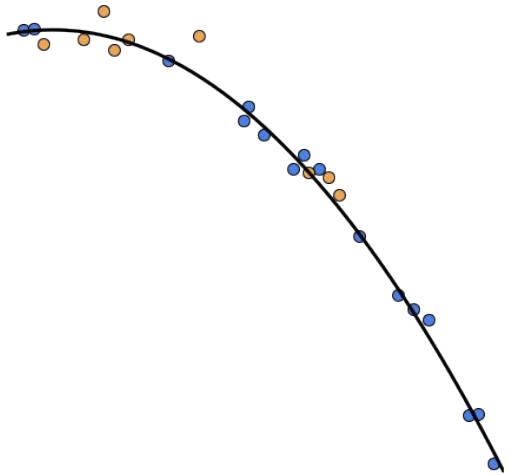
Take a look at these three fits to the same data:



$f(w,b) = w_1 x + b$

High bias model(underfit)

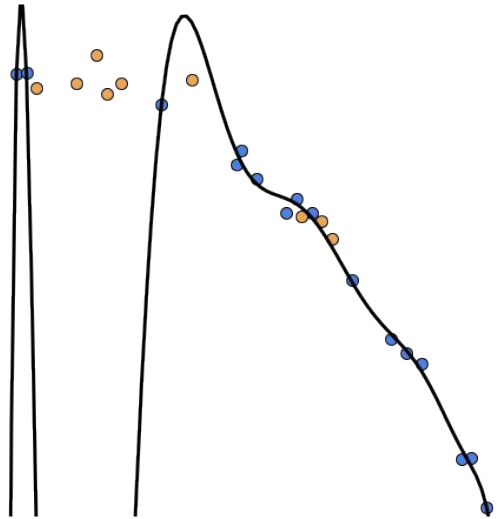
J_{train} is high
J_{dev} is high



$f(w,b) = w_1 x + w_2 x^2 + b$

Right model

J_{train} is low
J_{dev} is low

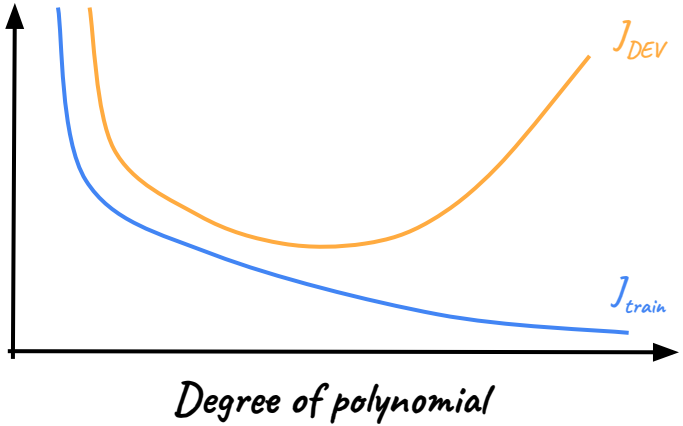


$f(w,b) = w_1 x + w_2 x^2 + \ldots + b$

High variance model (overfit)

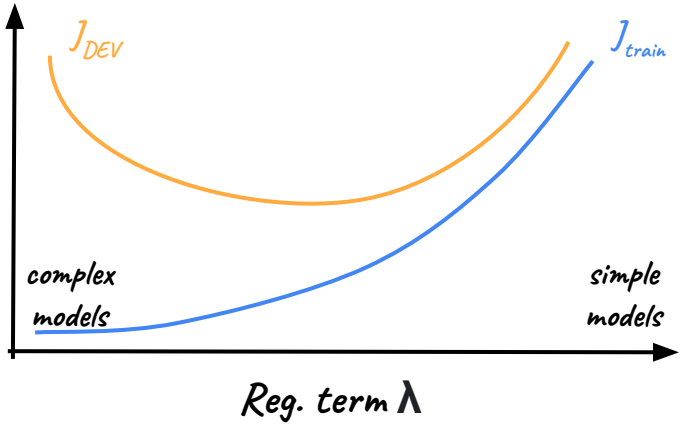
J_{train} is low
J_{dev} is high

To put it graphically, complex models will be prone to overfitting (high variance), simple models will be prone to underfit (high bias).

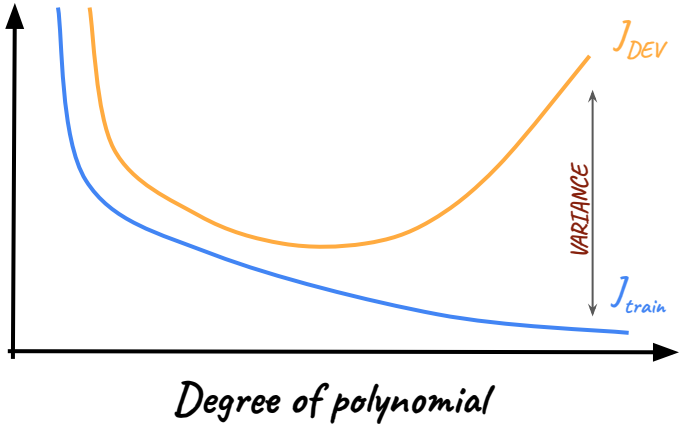


(notice how J_{DEV} usually is always higher than J_{train})

Regularization (λ specifically) influences the bias/variance of the model, specularly to what the polynomial degree does.

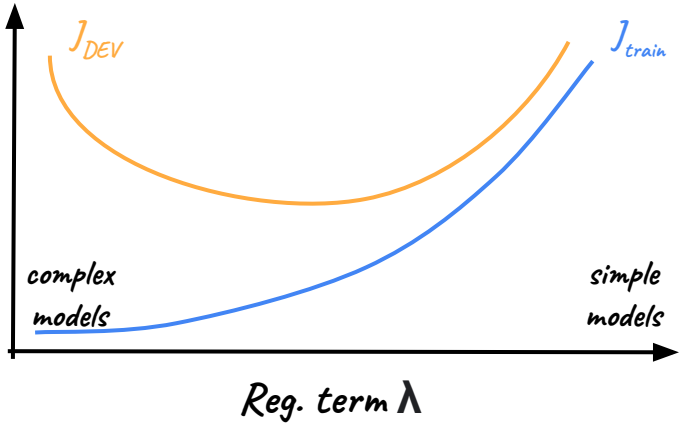


To put it graphically, complex models will be prone to overfitting (high variance), simple models will be prone to underfit (high bias).



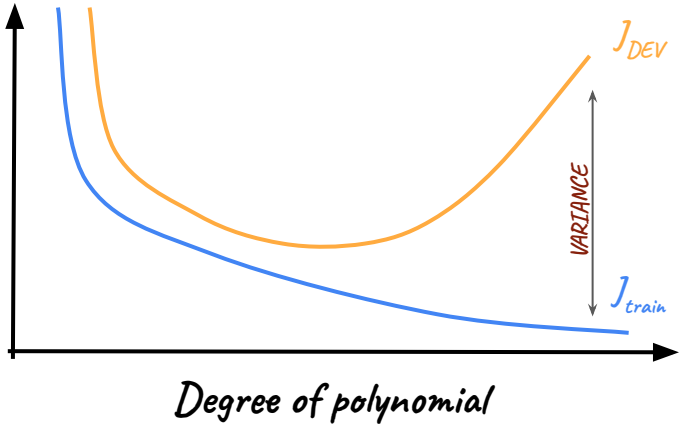
(notice how J_{DEV} usually is always higher than J_{train})

Regularization (λ specifically) influences the bias/variance of the model, specularly to what the polynomial degree does.



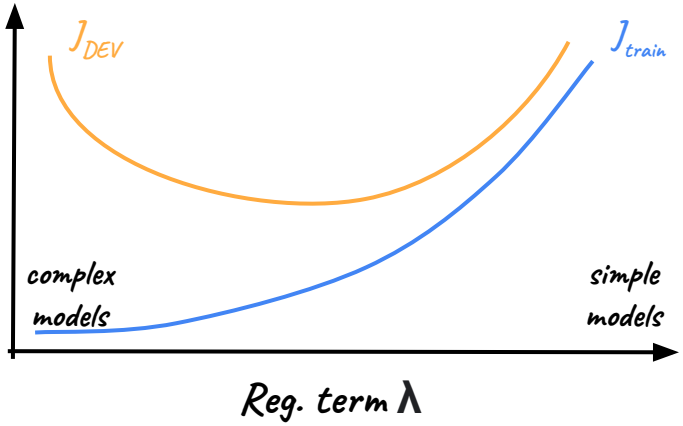
Ok, the variance can be viewed as the metrics displacement between the training and the DEV/test sets.
So, when we talk about high bias, is high with respect to… what?

To put it graphically, complex models will be prone to overfitting (high variance), simple models will be prone to underfit (high bias).



(notice how J_{DEV} usually is always higher than J_{train})

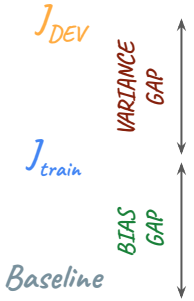
Regularization (λ specifically) influences the bias/variance of the model, specularly to what the polynomial degree does.



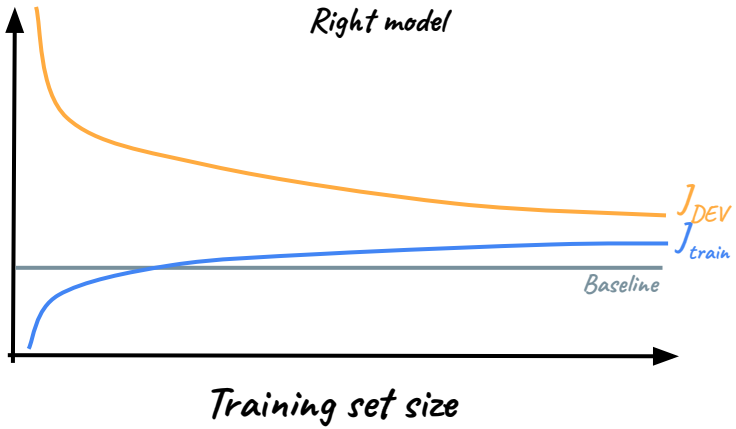
Ok, the variance can be viewed as the metrics displacement between the training and the DEV/test sets.
So, when we talk about high bias, is high with respect to… what?

Baseline level for evaluating a ML model

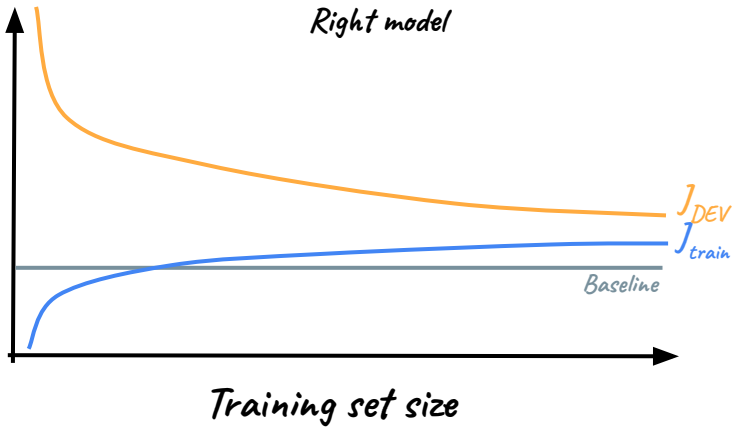
- Human level performance
- Competing algorithm performance
- Guess based on gut feel...ehm, "experience"

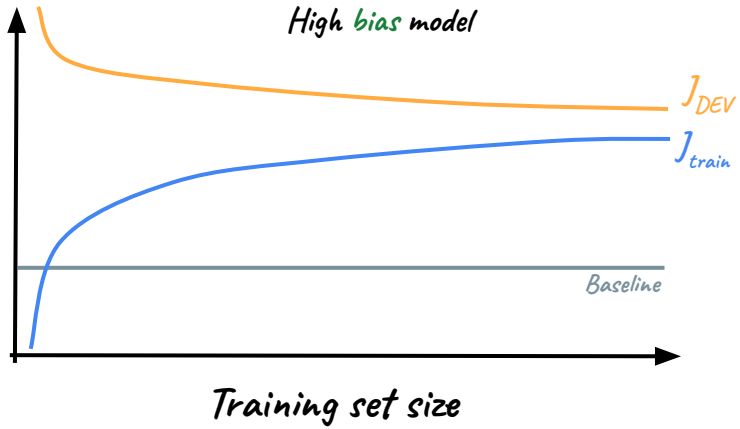


Does adding more training data help the overall model performance?

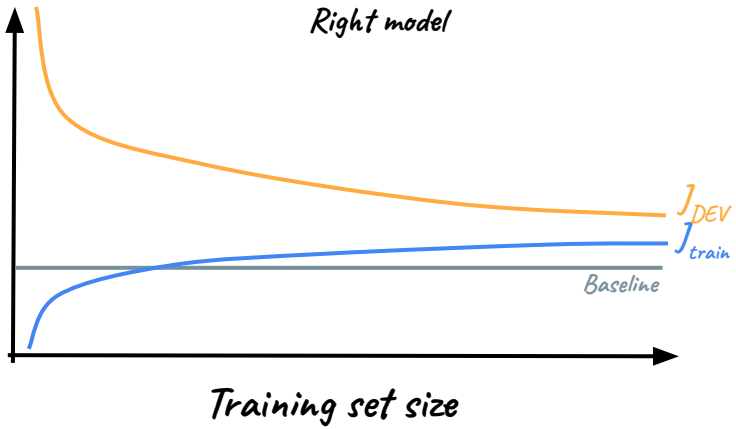


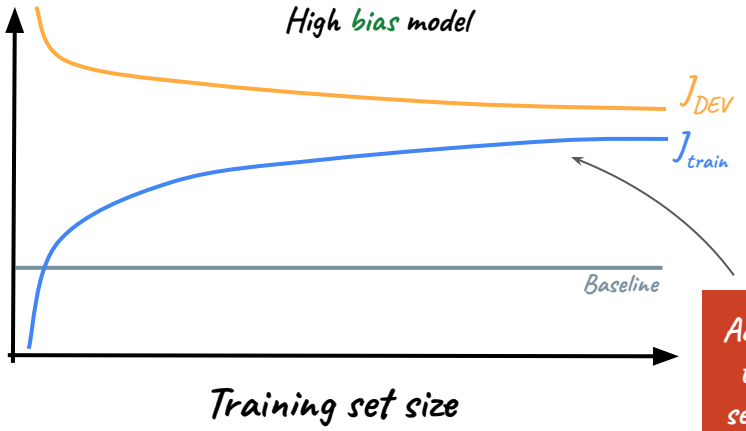
Does adding more training data help the overall model performance?





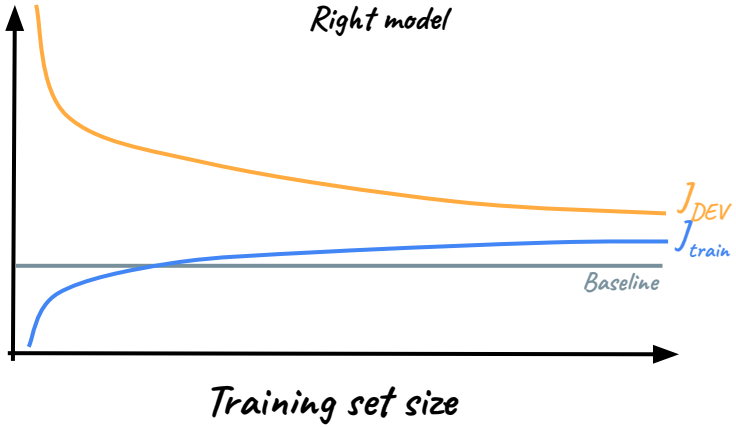
Does adding more training data help the overall model performance?

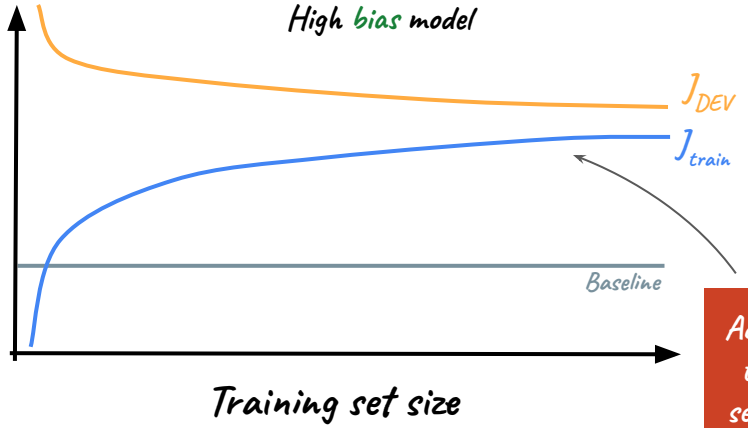




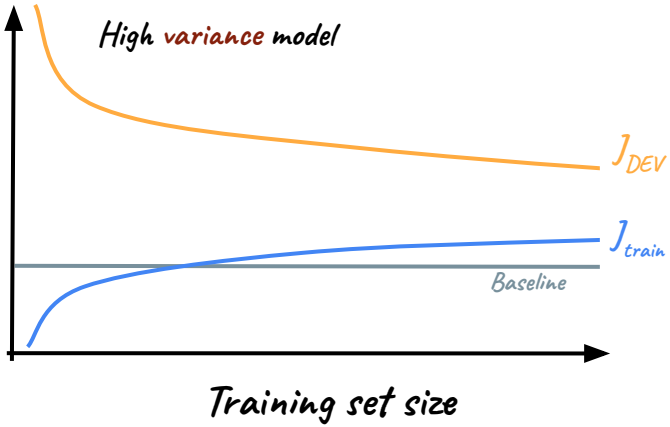
Adding more data to the training set won't help you lowering the bias!

Does adding more training data help the overall model performance?

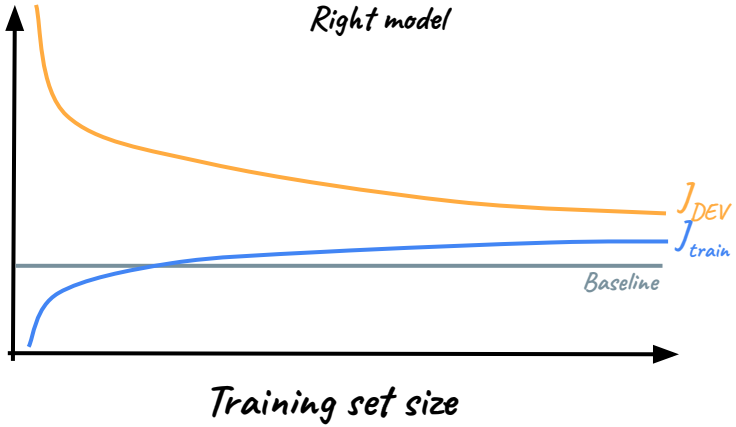


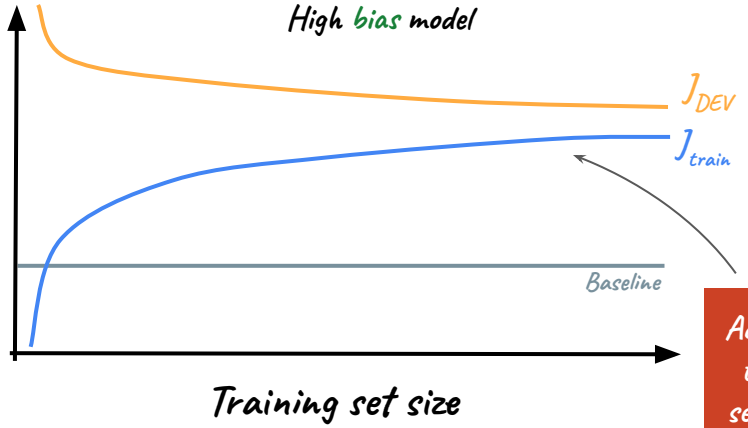


Adding more data to the training set won't help you lowering the bias!

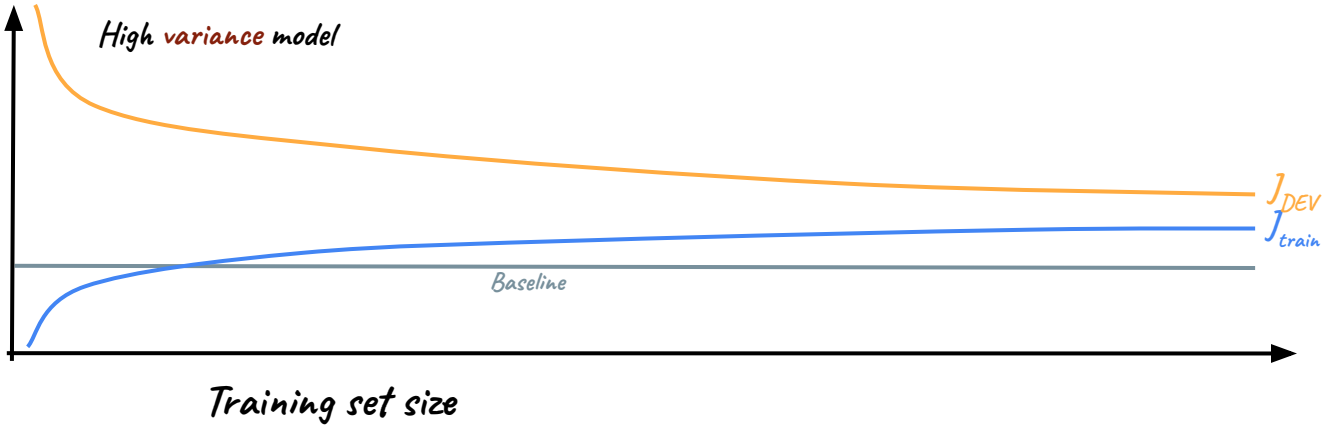


Does adding more training data help the overall model performance?



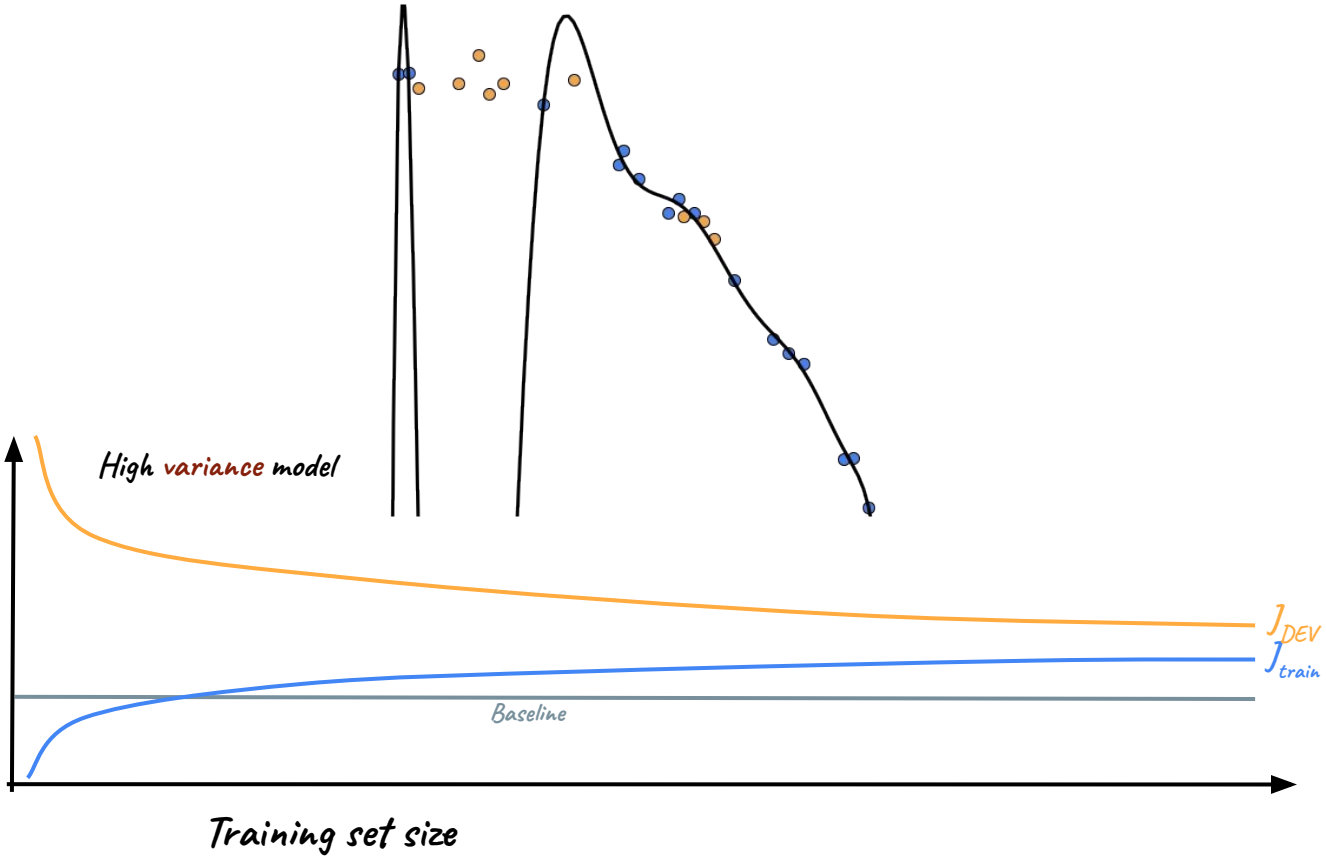


Adding more data to the training set won't help you lowering the bias!



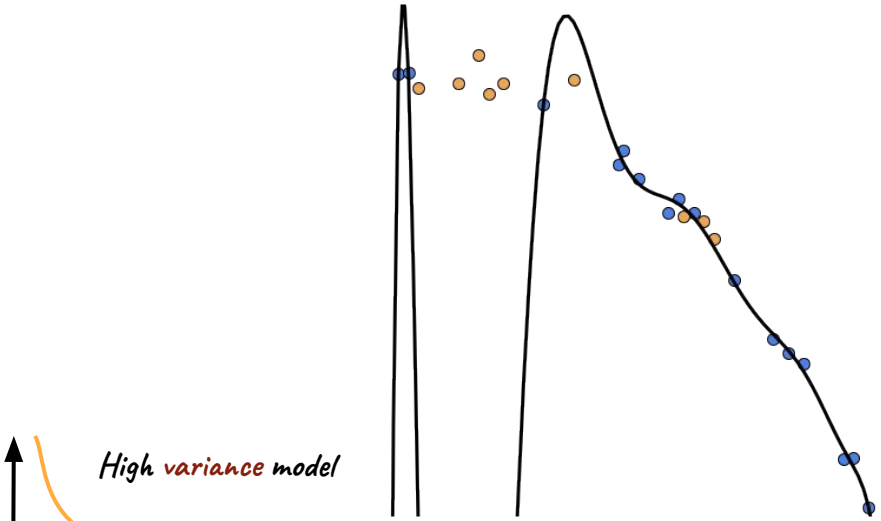
But it certainly helps lowering the variance!

Does adding more training data help the overall model performance?

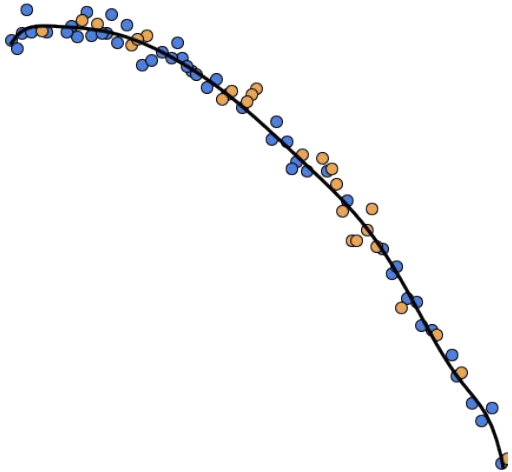


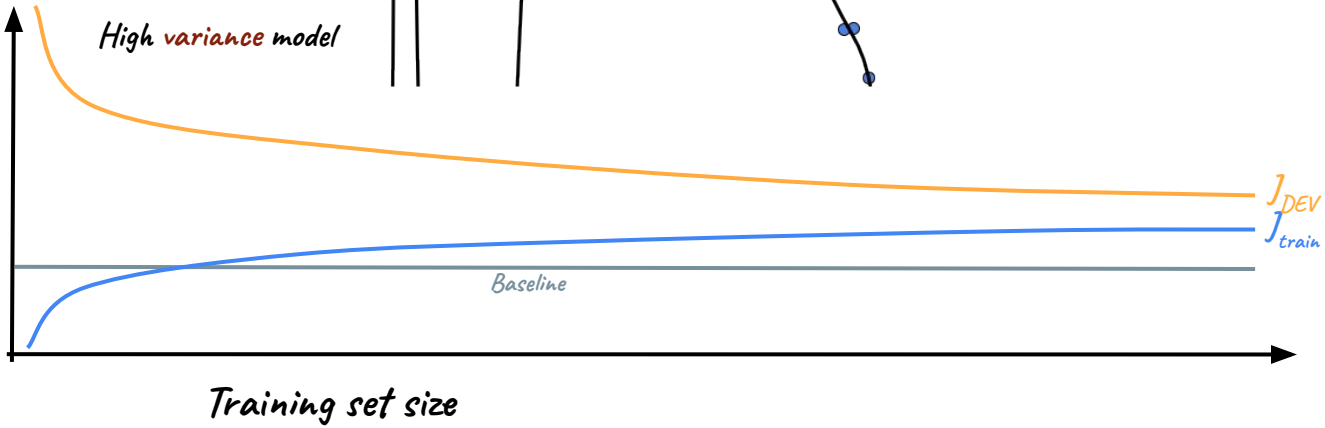
But it certainly helps lowering the variance!

Does adding more training data help the overall model performance?



Adding more training data





But it certainly helps lowering the variance!

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples
- Try smaller set of features
- Try getting additional features
- Try adding polynomial features
- Try decreasing the regularization λ
- Try increasing the regularization λ

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features
- Try getting additional features
- Try adding polynomial features
- Try decreasing the regularization λ
- Try increasing the regularization λ

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features
- Try adding polynomial features
- Try decreasing the regularization λ
- Try increasing the regularization λ

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features
- Try decreasing the regularization λ
- Try increasing the regularization λ

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features → fixes high bias
- Try decreasing the regularization λ
- Try increasing the regularization λ

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features → fixes high bias
- Try decreasing the regularization λ → fixes high bias
- Try increasing the regularization λ

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features → fixes high bias
- Try decreasing the regularization λ → fixes high bias
- Try increasing the regularization λ → fixes high variance

When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features → fixes high bias
- Try decreasing the regularization λ → fixes high bias
- Try increasing the regularization λ → fixes high variance

And the other way round:

- high variance → get more training examples OR simplify the model
- high bias → make your model more flexible/complex/powerful

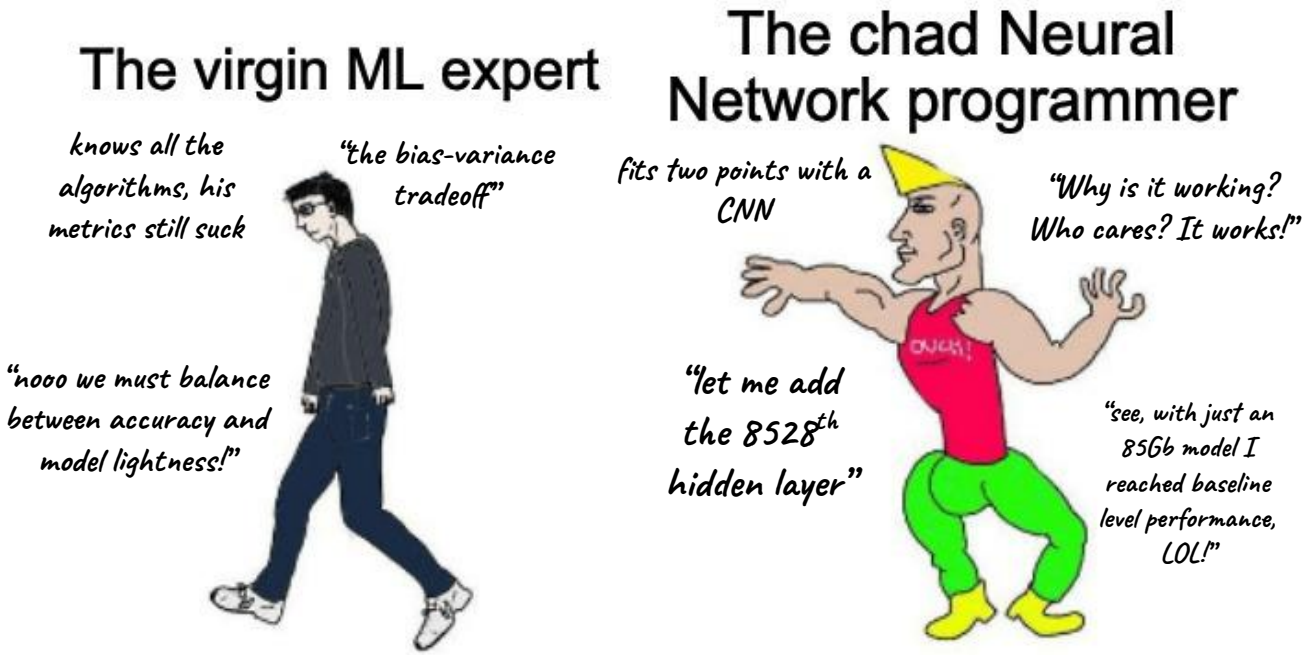
When your model presents bias/variance issue, you could try different solutions before scrapping the model architecture altogether:

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features → fixes high bias
- Try decreasing the regularization λ → fixes high bias
- Try increasing the regularization λ → fixes high variance

And the other way round:

- high variance → get more training examples OR simplify the model
- high bias → make your model more flexible/complex/powerful

Neural Networks offer a way out the bias/variance tradeoff dilemma (with some caveats) since a (large) NN are by nature a low-bias machine, as adding more and more hidden layers/units increases the model complexity… whose variance can be kept under control by increasing the number of examples in the training set (data augmentation trick), of course as long as it is properly regularized.
They're like the lazy, brute-force-lover, piledriver-coder paradise.
We'll see next week.



Neural Networks offer a way out the bias/variance tradeoff dilemma (with some caveats) since a (large) NN are by nature a low-bias machine, as adding more and more hidden layers/units increases the model complexity… whose variance can be kept under control by increasing the number of examples in the training set (data augmentation trick), of course as long as it is properly regularized.
They're like the lazy, brute-force-lover, piledriver-coder paradise.
We'll see next week.

Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

(credits: Kyle Boone introduction at SOM Machine 2021)

Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

(credits: Kyle Boone introduction at SOM Machine 2021)

Regression:

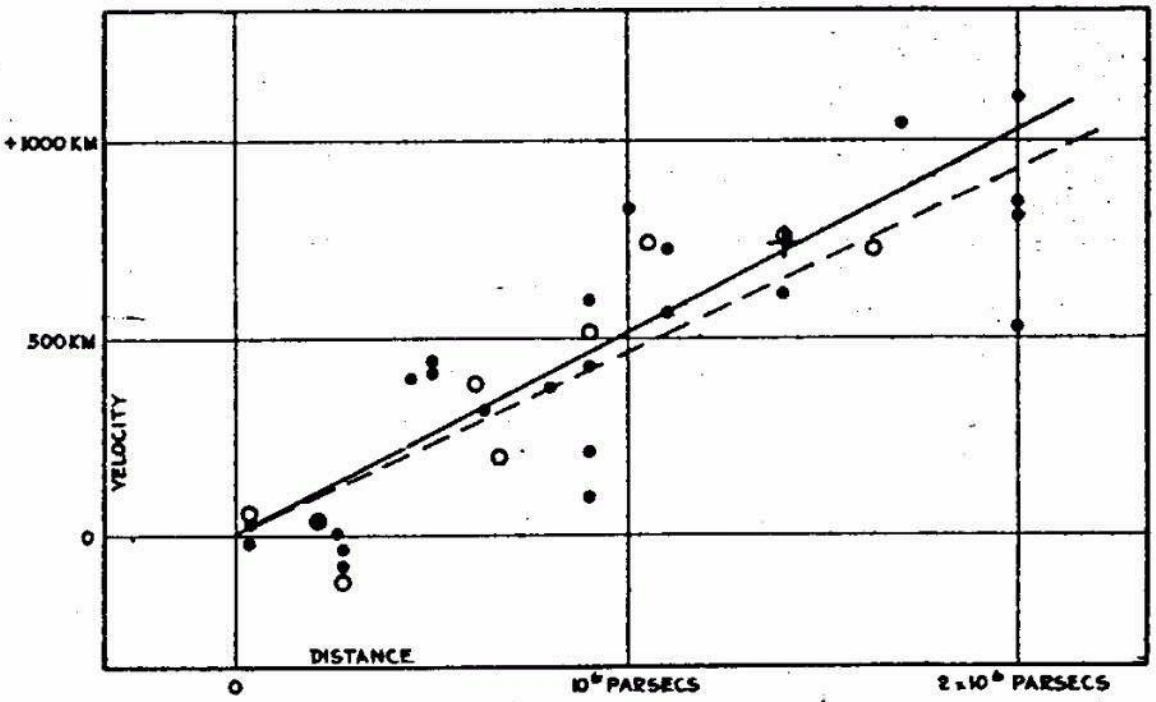
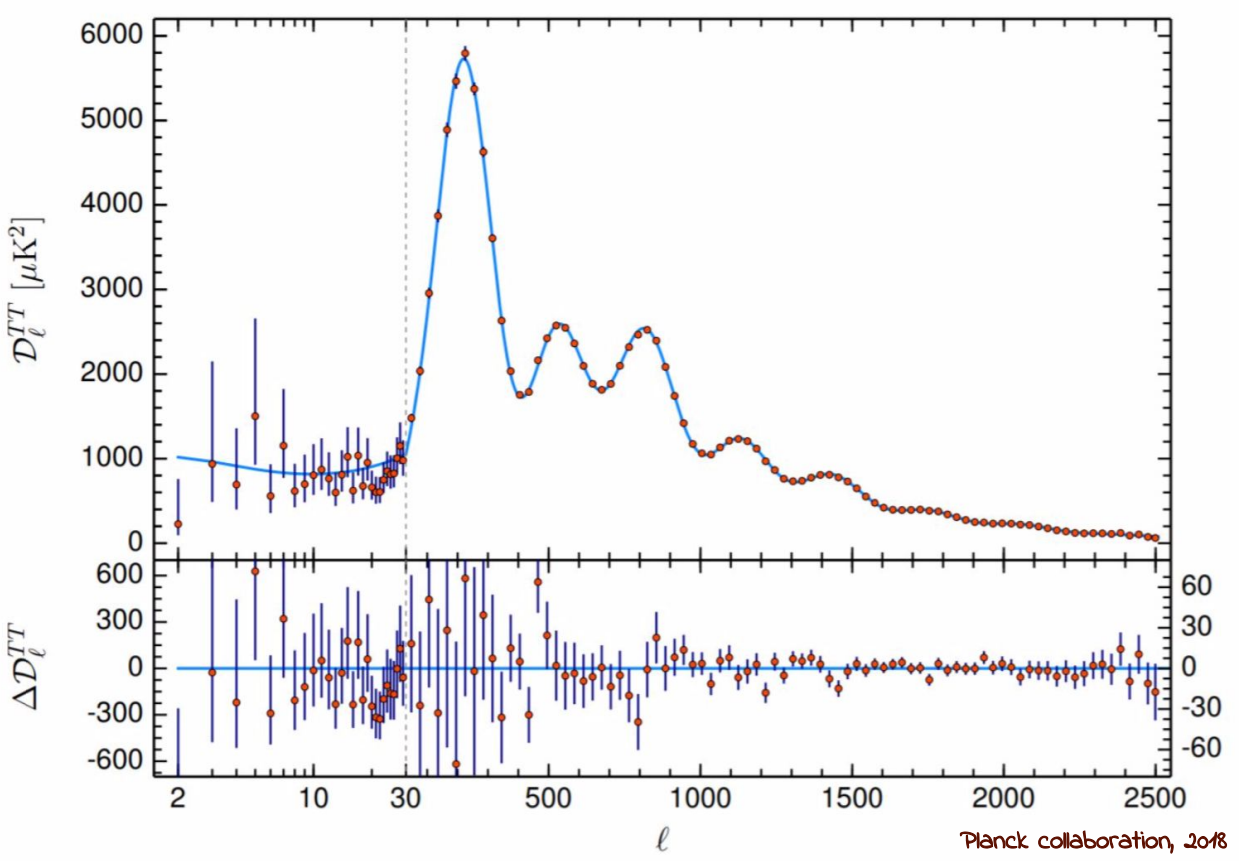


FIGURE 1

Hubble E., 1929, PNAS, 15, 168

Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

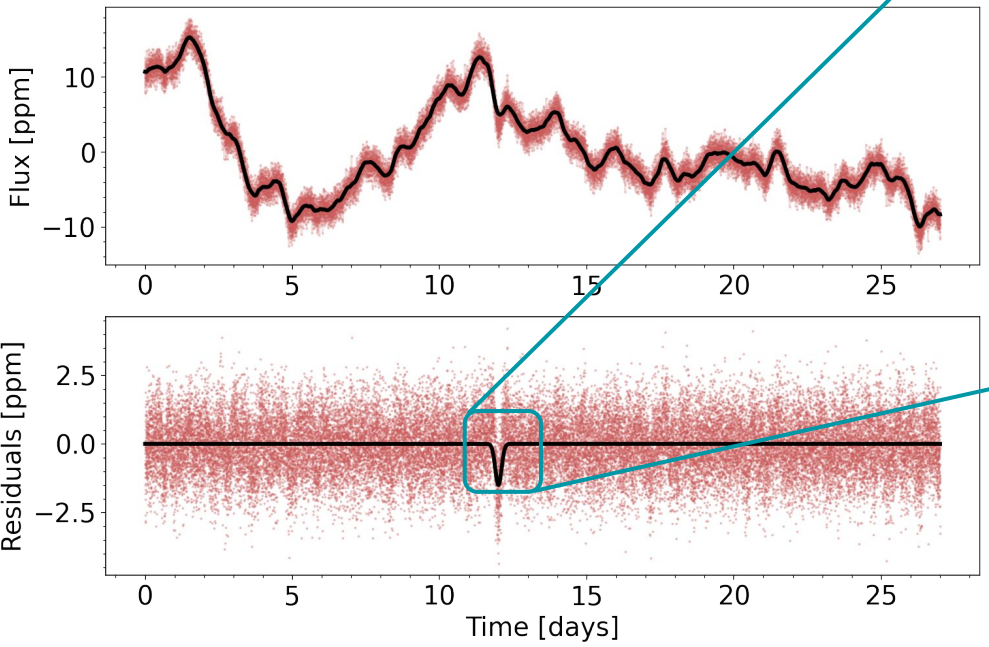
Regression:

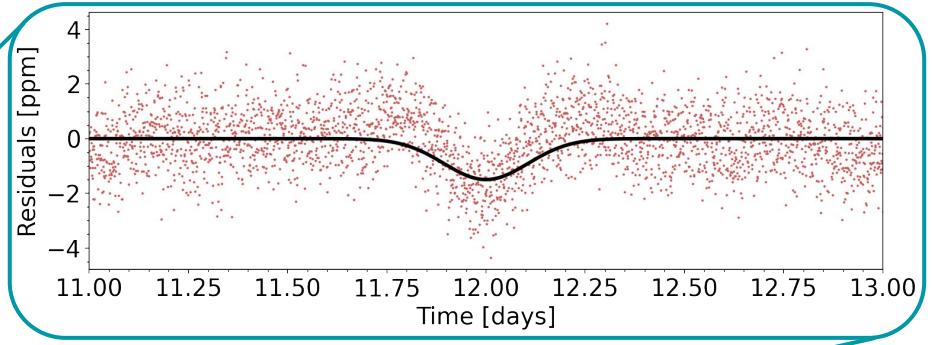


Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Regression:

We saw in the Notebooks how regularized linear/polynomial regression can be used to detrend light curves.



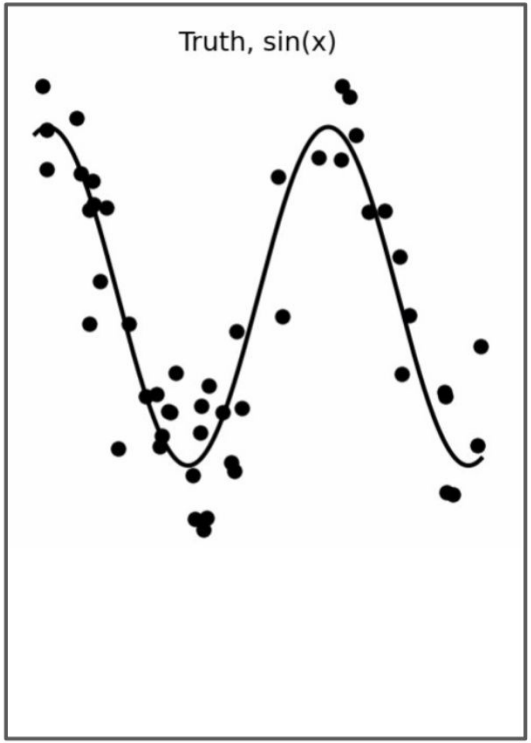


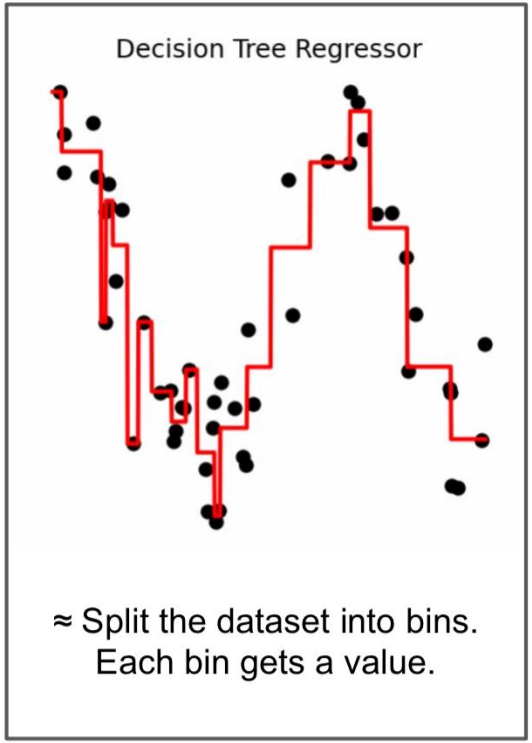
There are too many examples of transit curve fitting to place a single reference, so I am going to channel my inner Solomon and cite no-one

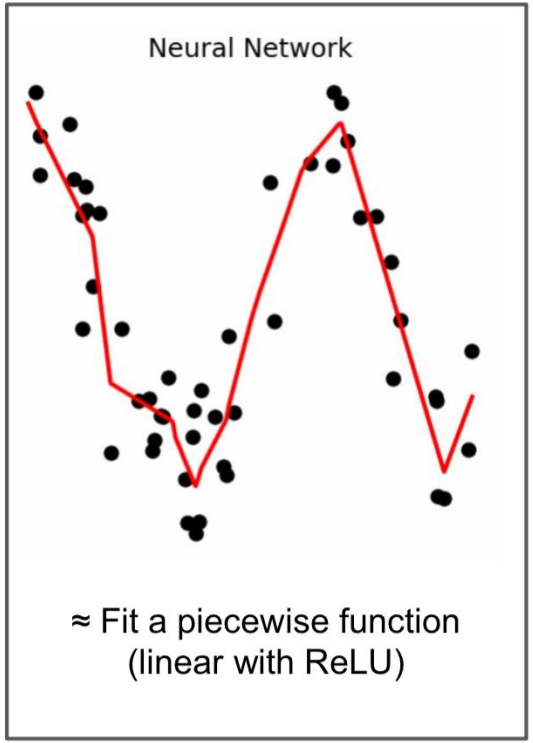
Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Regression:

With enough parameters (and training time) **any** function can be approximated, to a certain accuracy



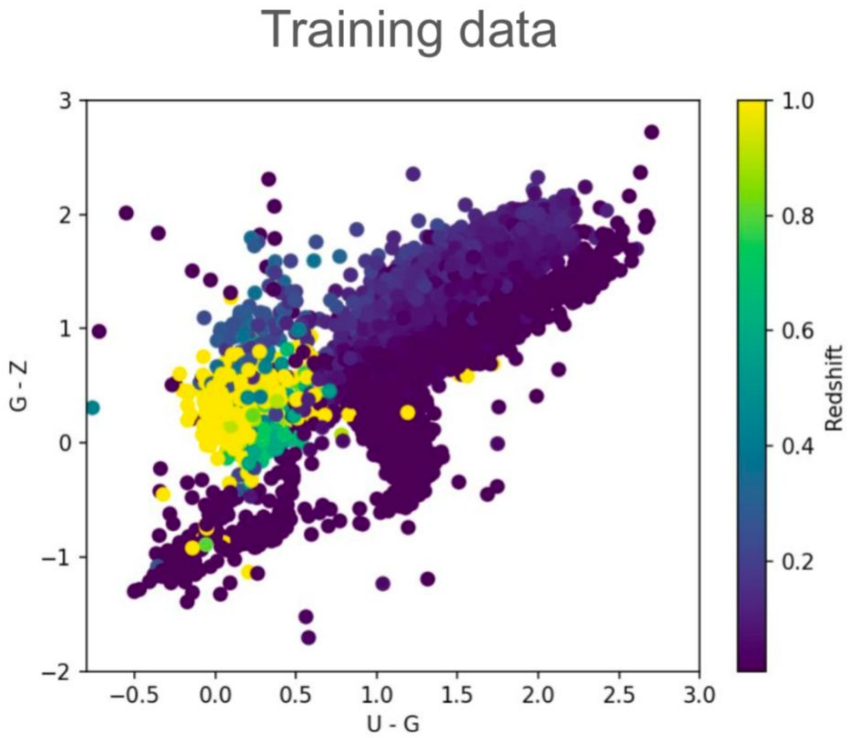


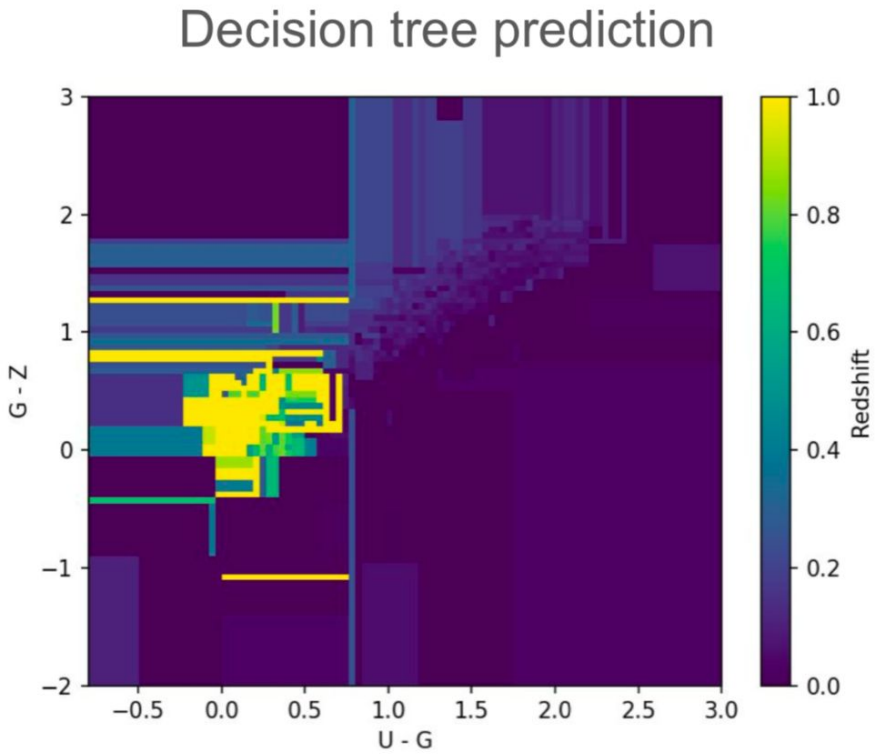


Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Regression:

Use ***random forests*** or ***NN algorithms*** to infer a galaxy redshift directly from the observed photometry



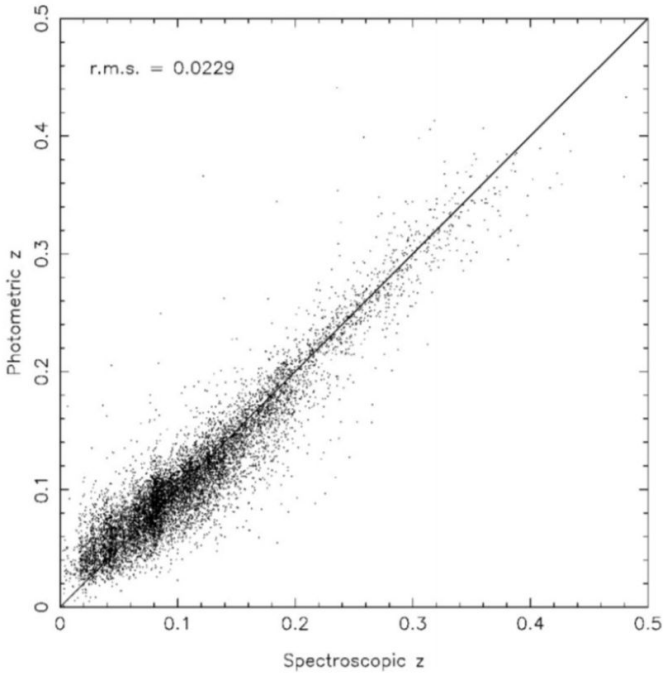


Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

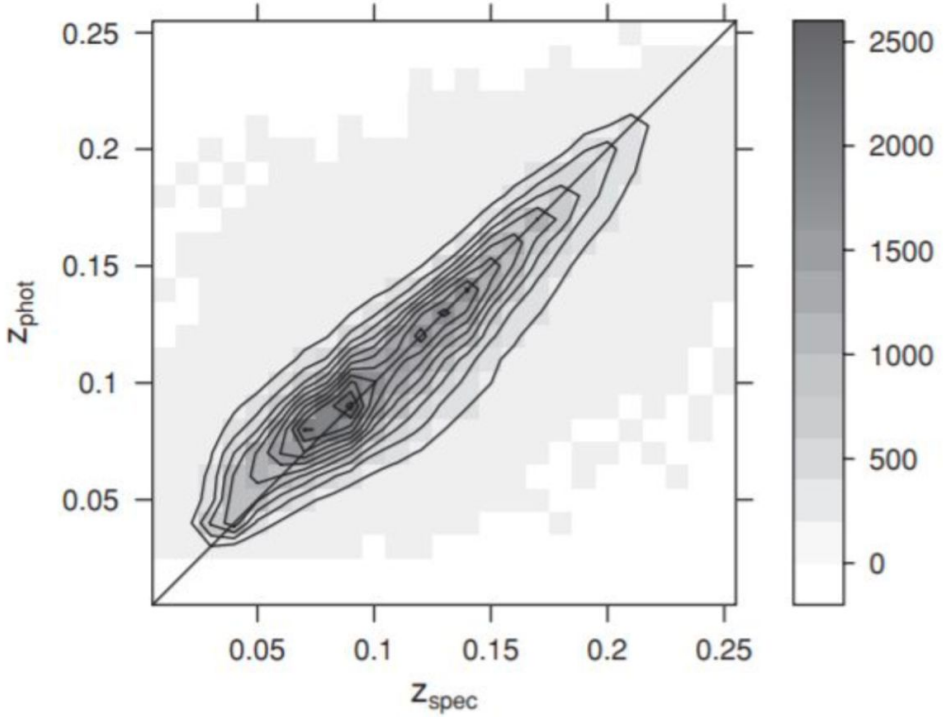
Regression:

Use **random forests** or **NN algorithms** to infer a galaxy redshift directly from the observed photometry

 Anjuz, Collister & Lahav 2004



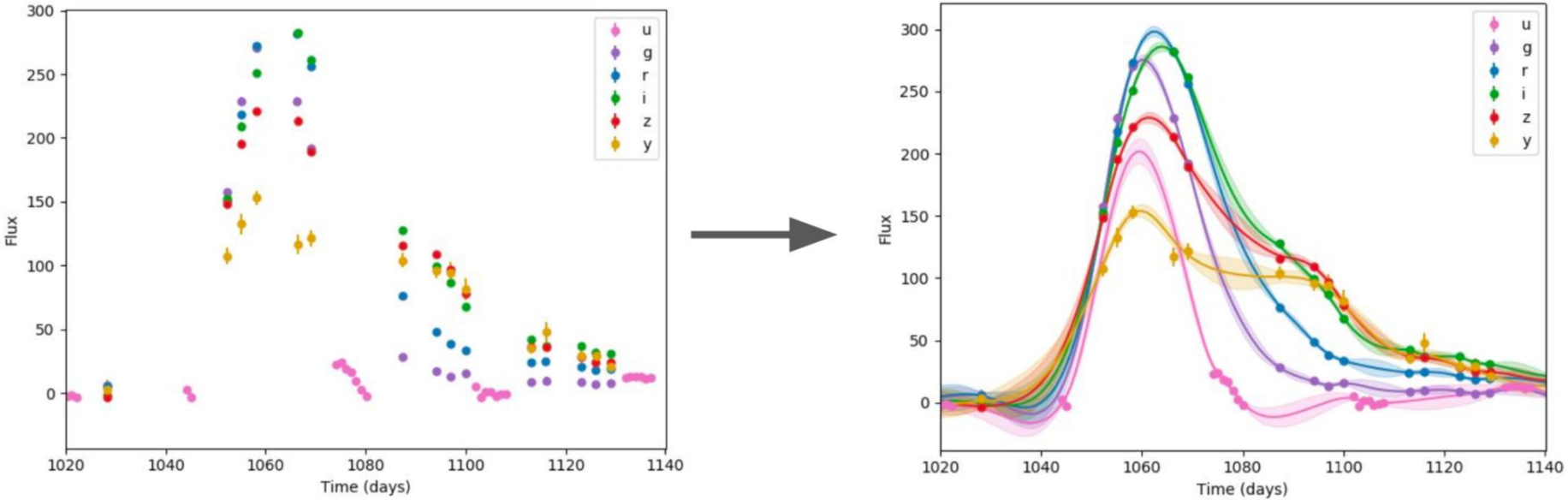
Carliles et al. 2010 with Random Forests



Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Regression:

Use **Gaussian Processes** to produce models with uncertainties from discrete data
e.g. SNae lightcurves, or (again) exoplanets, in general whatever time-series application you could think of.

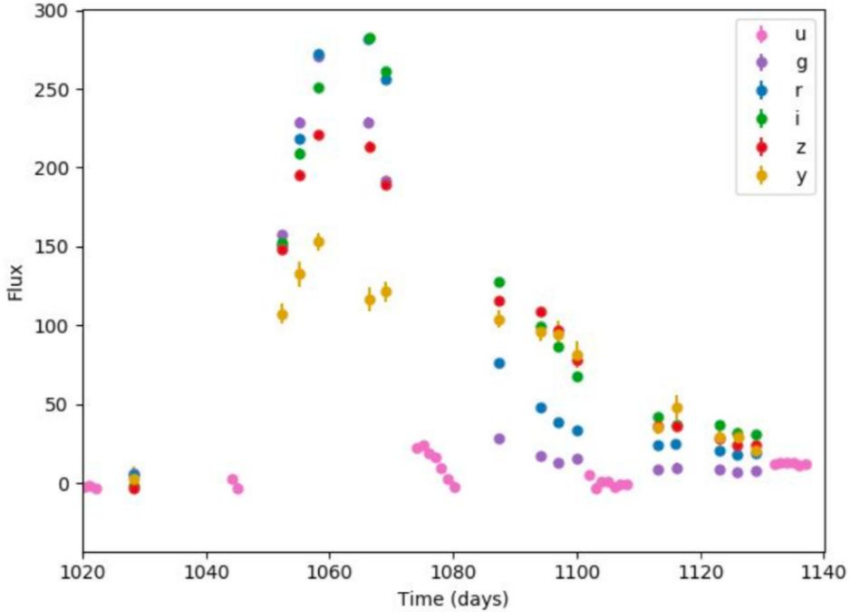


Boone et al., 2019

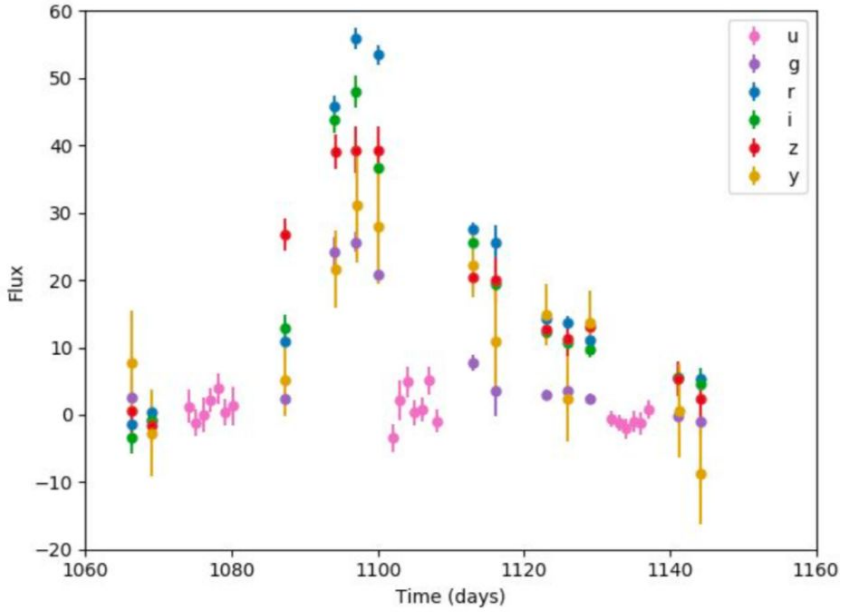
Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Classification:

Train model to tell, e.g. SNae lightcurves apart.



Type Ia Lightcurves



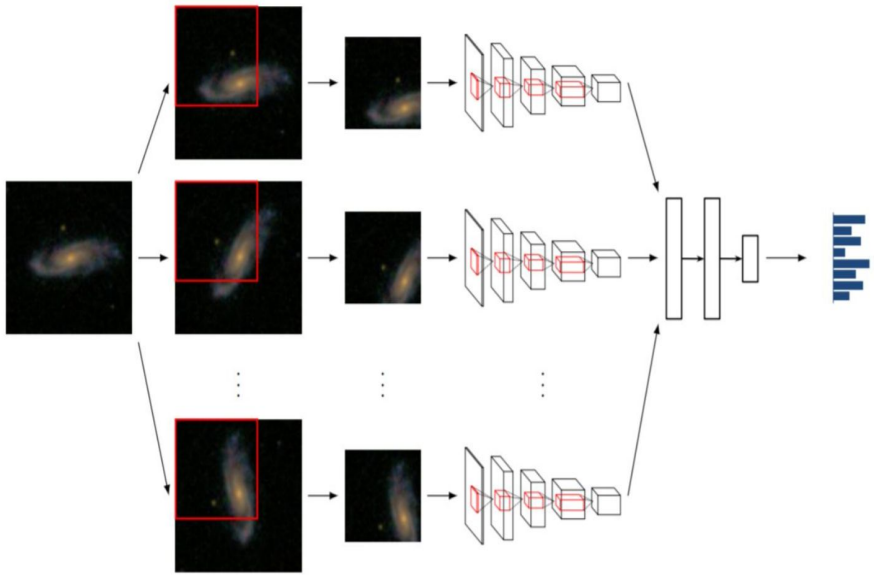
Type II Lightcurves

Boone et al, 2019

Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

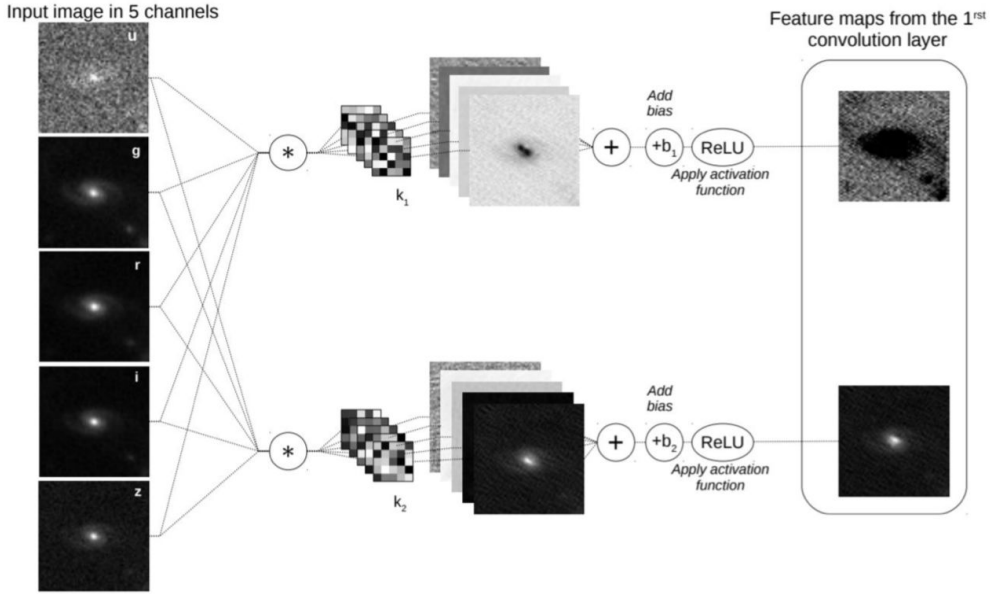
Classification:

Train model to tell, e.g. galaxy morphology directly from images:



Dieleman et al., 2014

or the photometric redshifts using also the spatial information:

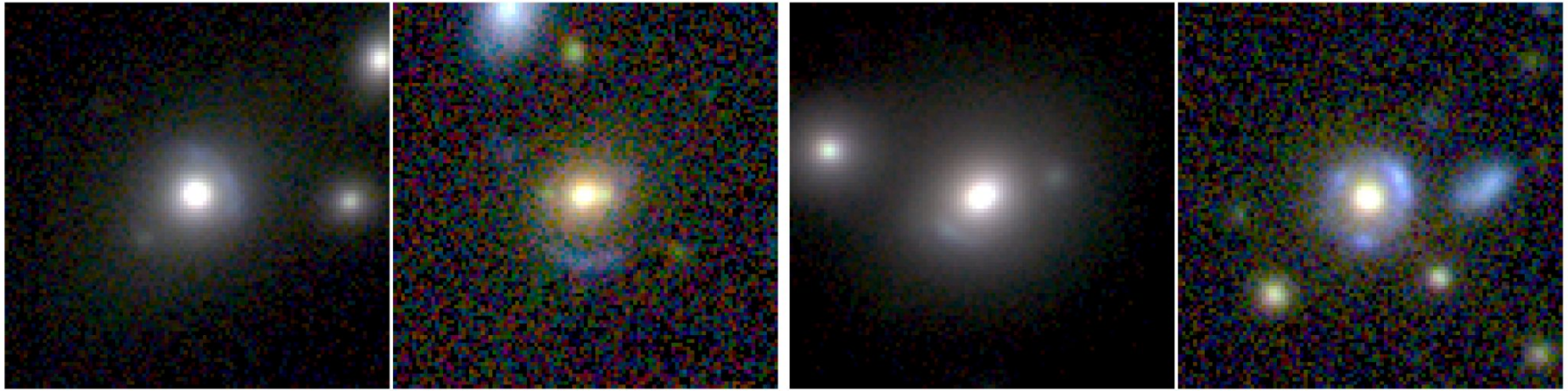


Pasquet et al., 2019

Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Classification:

Train model to e.g., automatically find strong lenses in wide fields

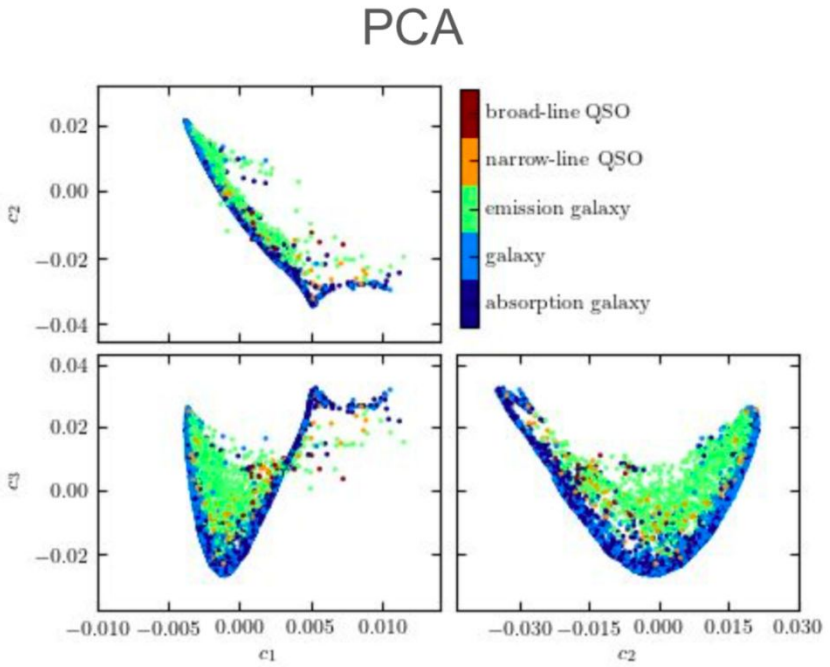


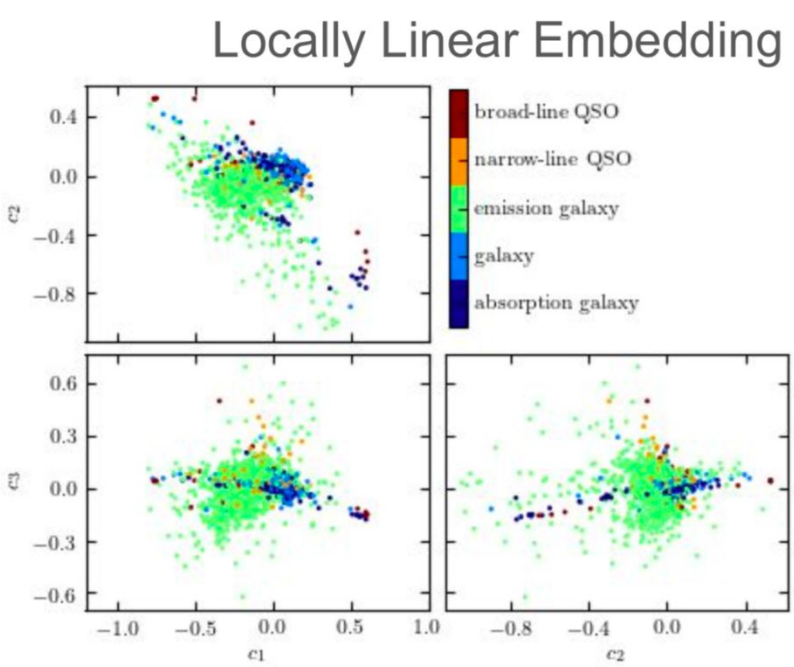
Petrillo et al., 2019

Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Unsupervised:

Use dimensionality reduction to, e.g., automatically discriminate between different kind of galaxies:



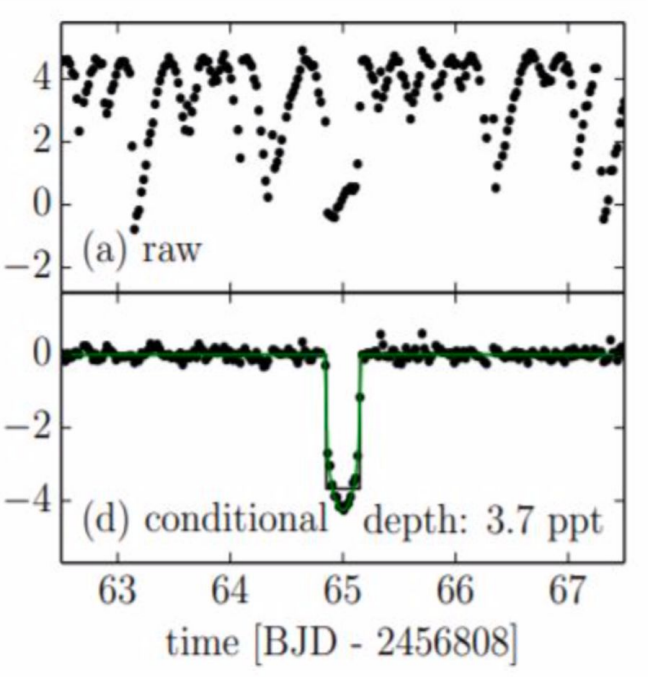


VanderPlas et al, 2014

Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Unsupervised:

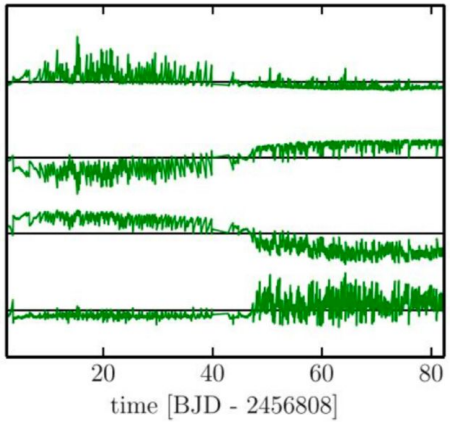
Use PCA to detrend light curves:



Find noise in common between observations.

Remove common noise

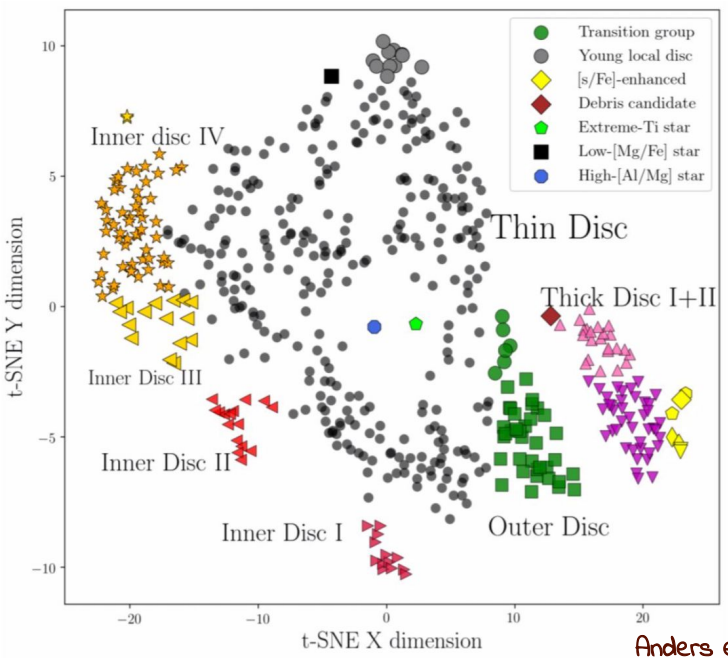
Foreman-Mackey et al, 2015



Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

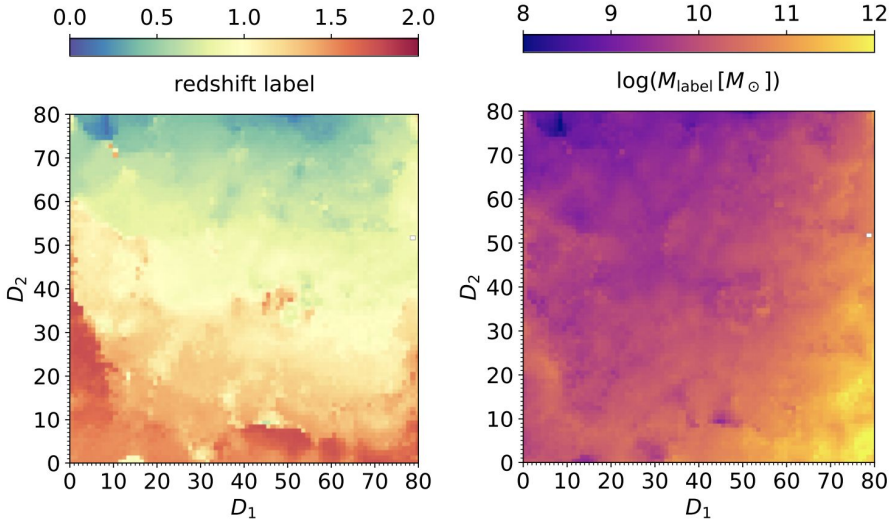
Unsupervised:

Use t-SNE to dimensionally reduce a complex 13-feature space (here, abundances) to a 2-dimensional space embedding, visualizing a hidden subgroup structure:



Anders et al, 2018

Use Self-Organizing Maps to learn complex mapping in the feature space, e.g. the photometry-redshift relation, or even photometry-physical properties. A case of unsupervised learning turning into supervised learning:

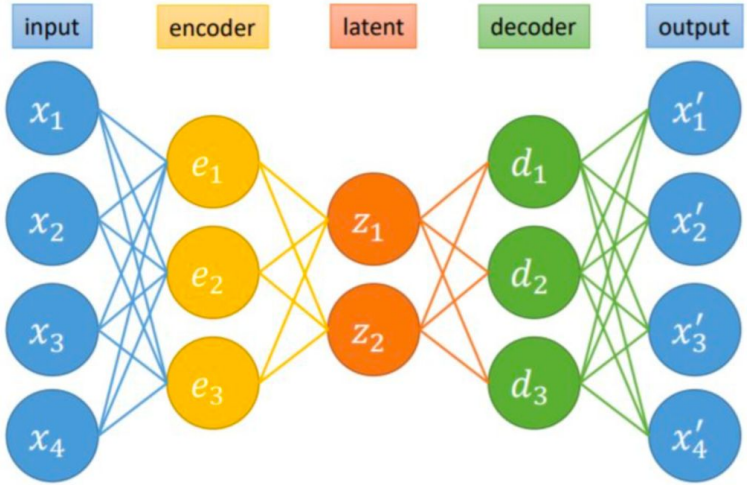


Davidzon et al, 2022

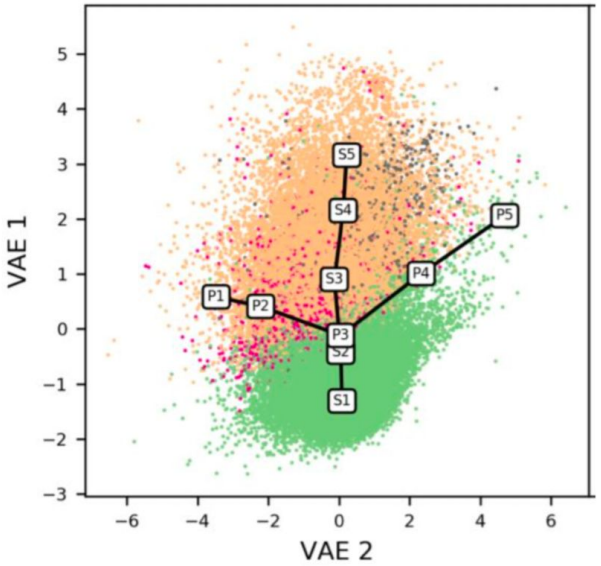
Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Unsupervised:

Autoencoders: neural networks with a bottleneck layer, encoding a low-dimensional representation. Useful for e.g. dimensionality reduction.



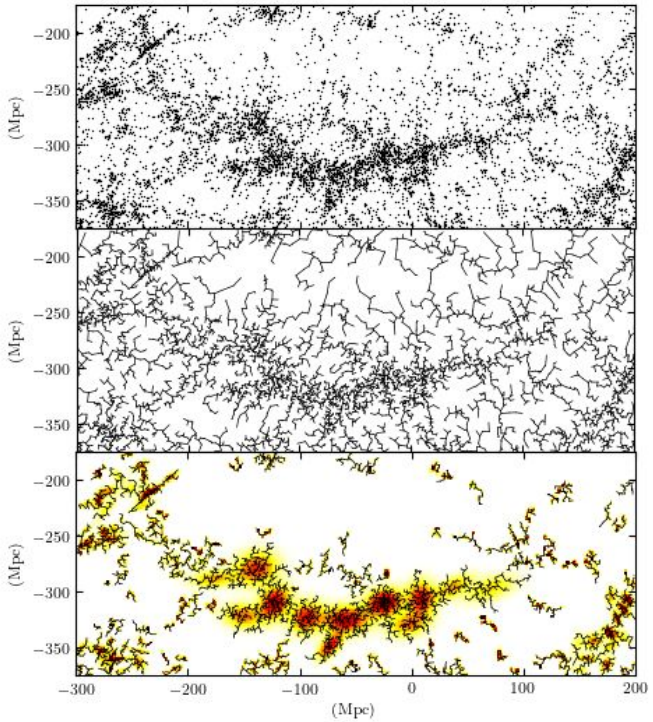
Portillo et al., 2020



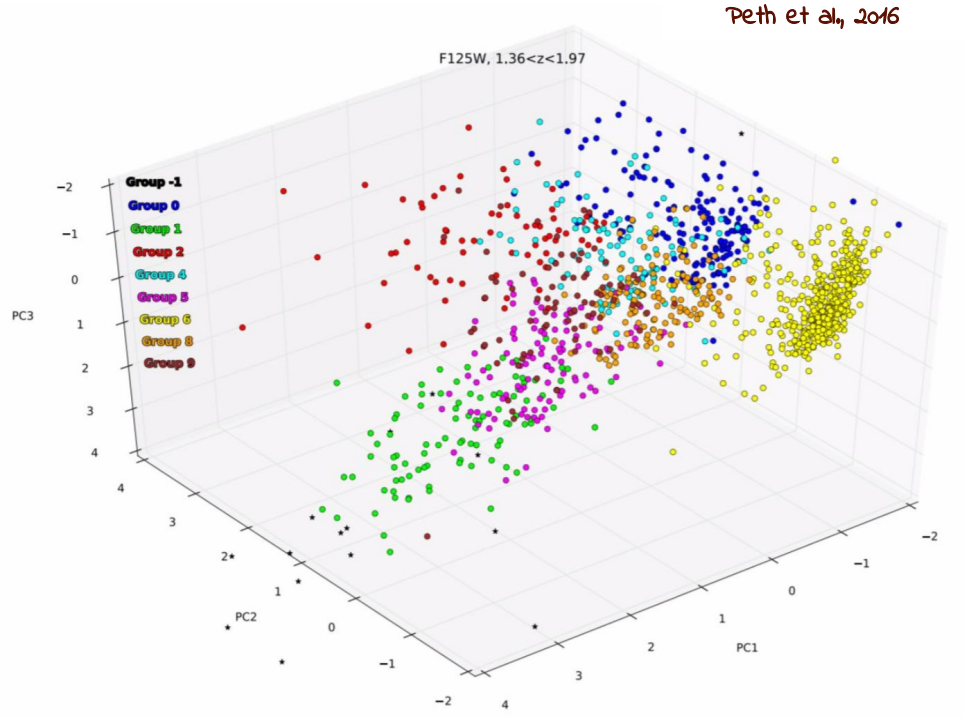
Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

Unsupervised:

Clustering: find close example in feature space (or on space-space) with associated properties



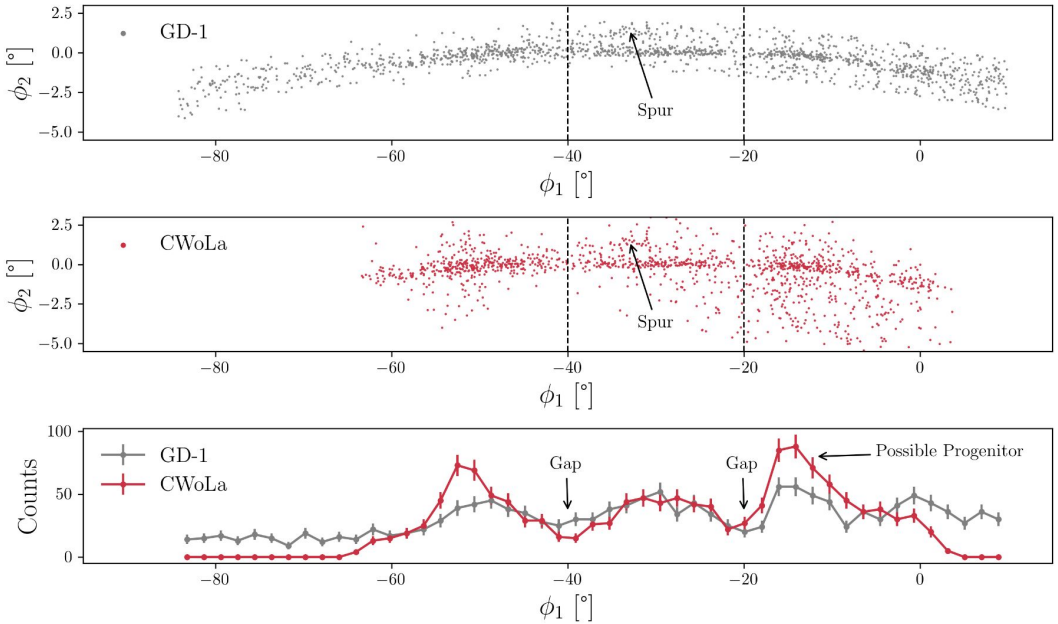
Peth et al., 2016



Now that you saw the basics (the very, very, very basics) of ML, I can show you a series of real-world applications.

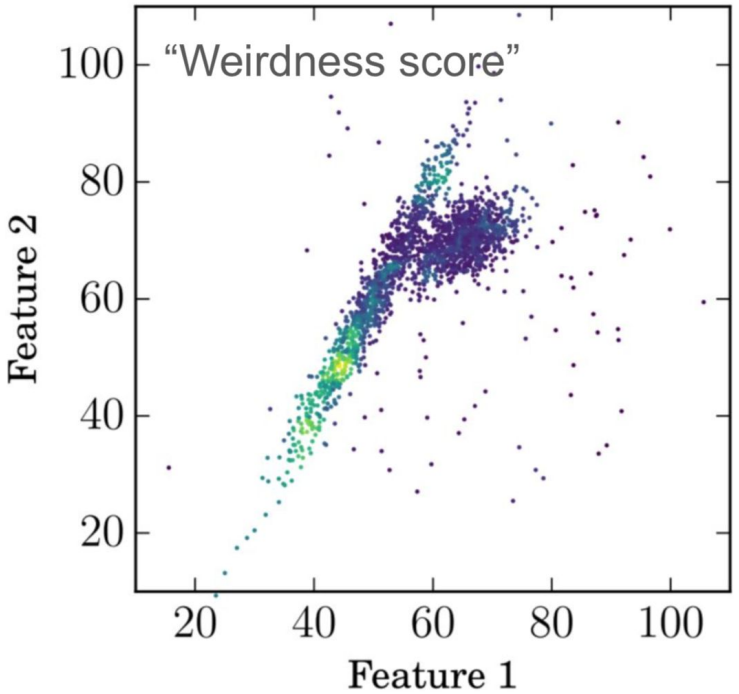
Unsupervised:

Anomaly detection: identify spectra that are very different from the rest of the sample or from known objects.



Gaia CWoLa - Pettee et al., 2023

Baron and Poznanski, 2017



There are lots and lots of algorithm out there. Thankfully, there are also lots of tutorials and suggestions that apply to your particular application.