Artificial Intelligence and Deep Machine Learning for Fin-Techs

Cristian TOMA & Marius POPA

Cyber | ICT Security Master Program – ISM
Applied Computer Science and Cybersecurity R&D Team – ACS
Department of Economic Informatics & Cybernetics – DICE | DEIC
www.ism.ase.ro | www.acs.ase.ro | www.dice.ase.ro

Make Artificial Intelligence and Machine Learning Work for

Fin-Techs

Visit us at www.fintech-ho2020.eu

or contact as at info@fintech-ho2020.eu

or follow us on LinkedIn

The development of the materials has also been supported by the project **A FINancial supervision and TECHnology compliance training programme.**

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825215 (Topic: ICT-35-2018 Type of action: CSA)

Agenda for the Presentation

1. Fundamentals of Machine Learning
2. AI – Neural Networks & Deep Learning
3. Exchange Ideas

www.dice.ase.ro

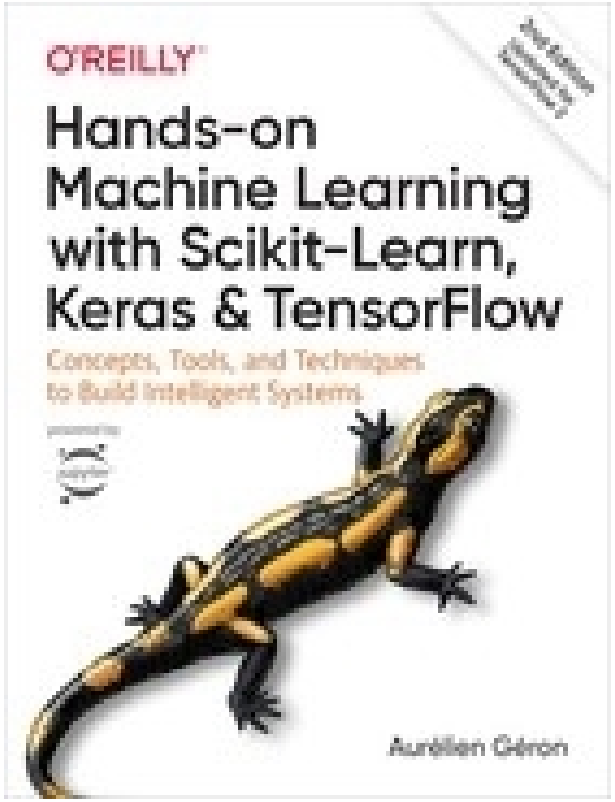
www.ism.ase.ro

Machine Learning Technology Overview: Classification, Training models, SVM, Decision Trees …

Machine Learning Fundamentals

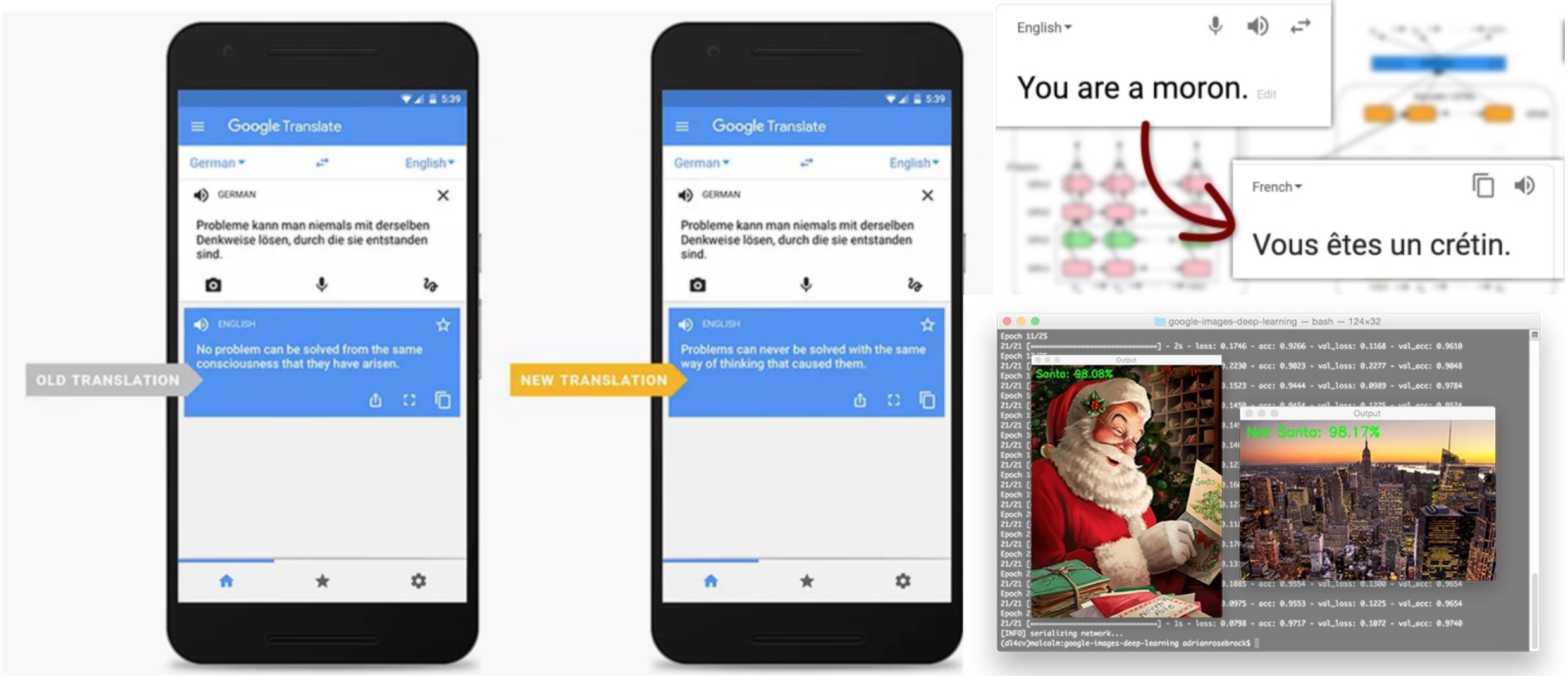
Artificial Intelligence & Machine Learning Overview

When most people hear "Machine Learning", they picture:



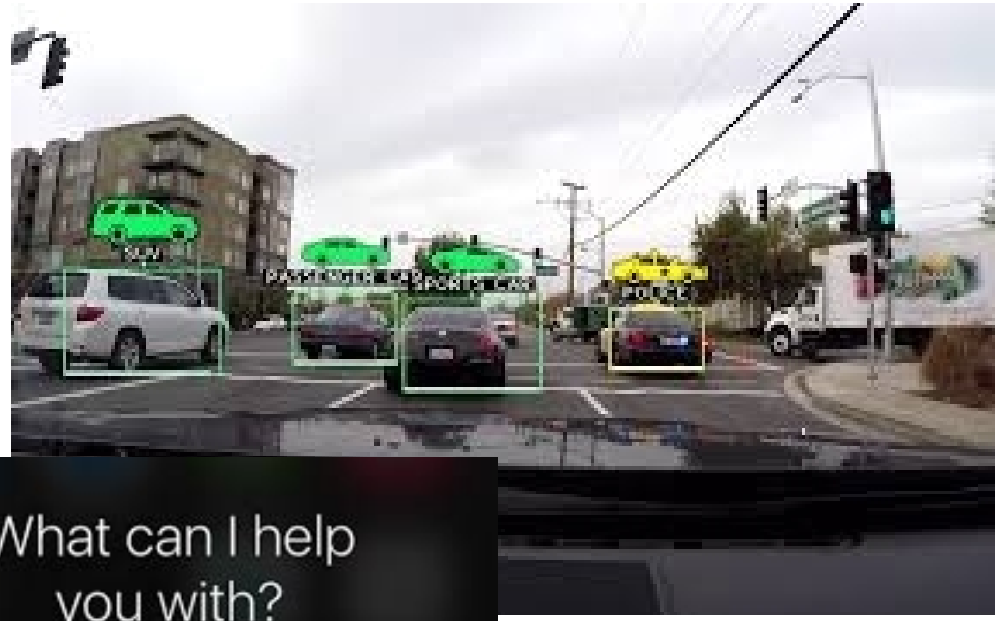
Artificial Intelligence & Machine Learning Overview

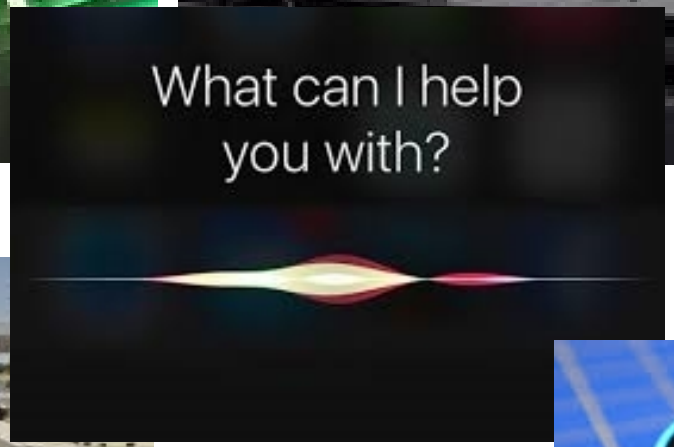
When most people hear "Machine Learning" / "Învățare automată", they picture:



Artificial Intelligence & Machine Learning Overview

When most people hear "Machine Learning," they picture?:





Artificial Intelligence & Machine Learning Overview

"But Machine Learning is not just a futuristic fantasy; it's already here. In fact, it has been around for decades in some specialized applications, such as Optical Character Recognition (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: the *spam filter*. It's not exactly a self-aware Skynet, but it does technically qualify as Machine Learning (it has actually learned so well that you seldom need to flag an email as spam anymore). It was followed by hundreds of ML applications that now quietly power hundreds of products and features that you use regularly, from better recommendations to voice search."

Machine Learning is the science (and art) of programming computers so they can *learn from data*.

Here is a slightly more general definition:

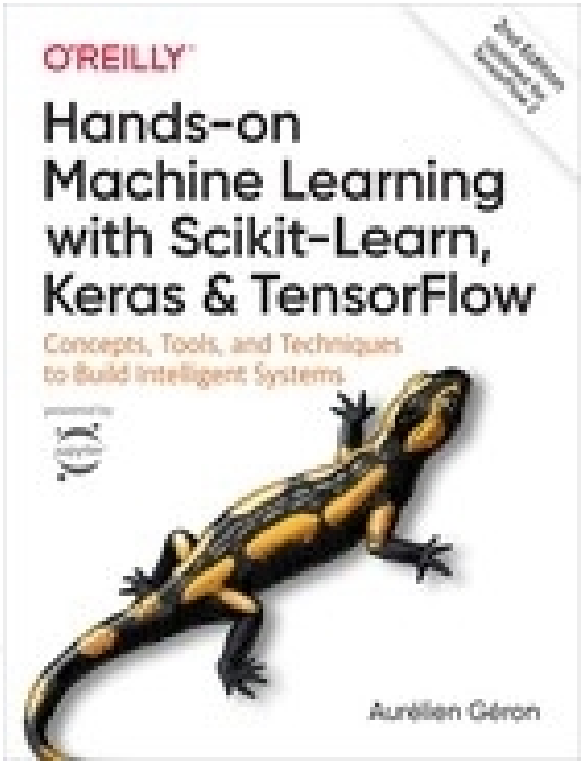
[Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.

Arthur Samuel, 1959

And a more engineering-oriented one:

A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

Tom Mitchell, 1997



Artificial Intelligence & Machine Learning Overview

"Where does Machine Learning start and where does it end?

What exactly does it mean for a machine to *learn* something?

If I download a copy of Wikipedia, has my computer really learned something?

Is it suddenly smarter?"

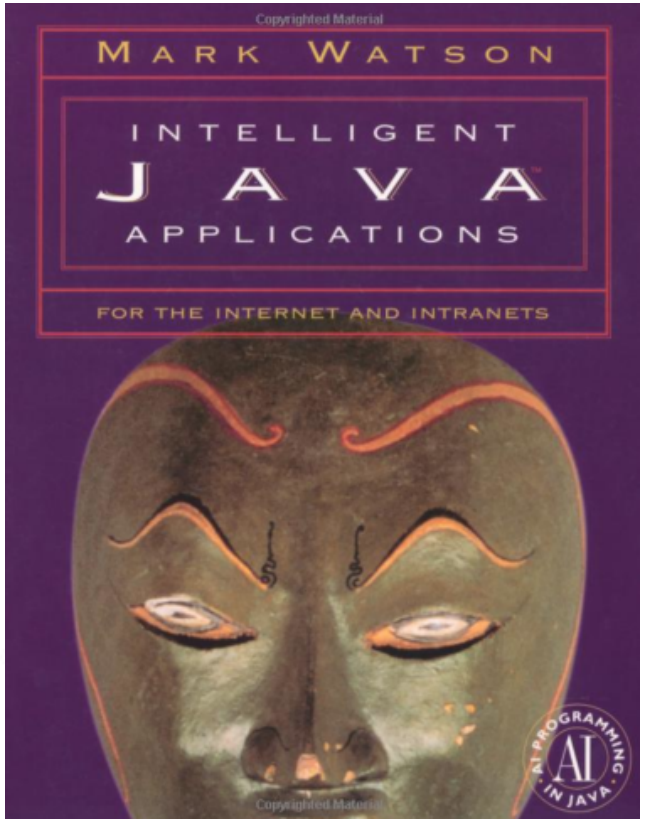
If you just download a copy of Wikipedia, your computer has a lot more data, but it is not suddenly better at any task. Thus, downloading a copy of Wikipedia is not Machine Learning.

Machine Learning is not just a futuristic fantasy; it's already here. In fact, it has been around for decades in some specialized applications, such as Optical Character Recognition (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: the *spam filter*.

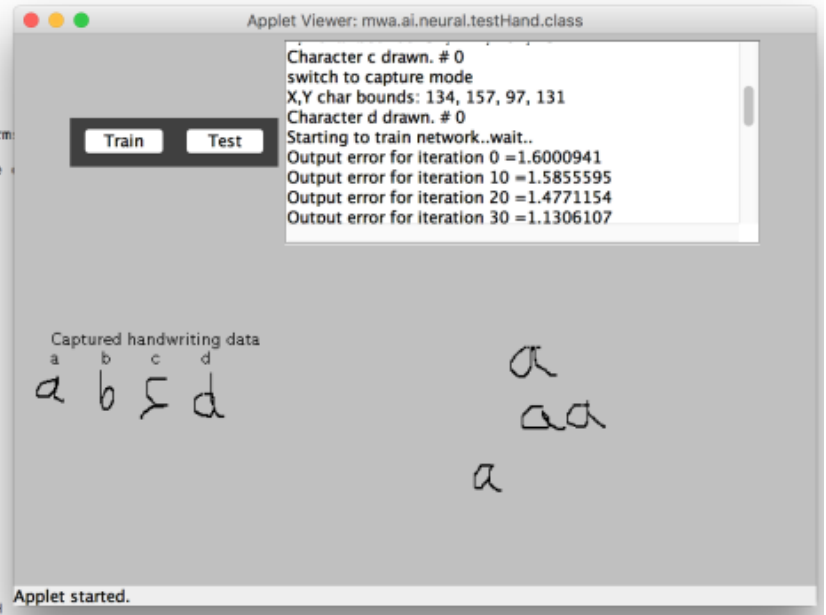
Your spam filter is a Machine Learning program that, given examples of spam emails (e.g., flagged by users) and examples of regular (nonspam, also called "ham") emails, can learn to flag spam. The examples that the system uses to learn are called the *training set*. Each training example is called a *training instance* (or *sample*). In this case, the task *T* is to flag spam for new emails, the experience *E* is the *training data*, and the performance measure *P* needs to be defined; for example, you can use the ratio of correctly classified emails. This particular performance measure is called *accuracy*, and it is often used in classification tasks.

DEMO 0 in Java

```
$ export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home
$ export PATH=.:$JAVA_HOME/bin:$PATH
$ appletviewer testHand_2.html
```



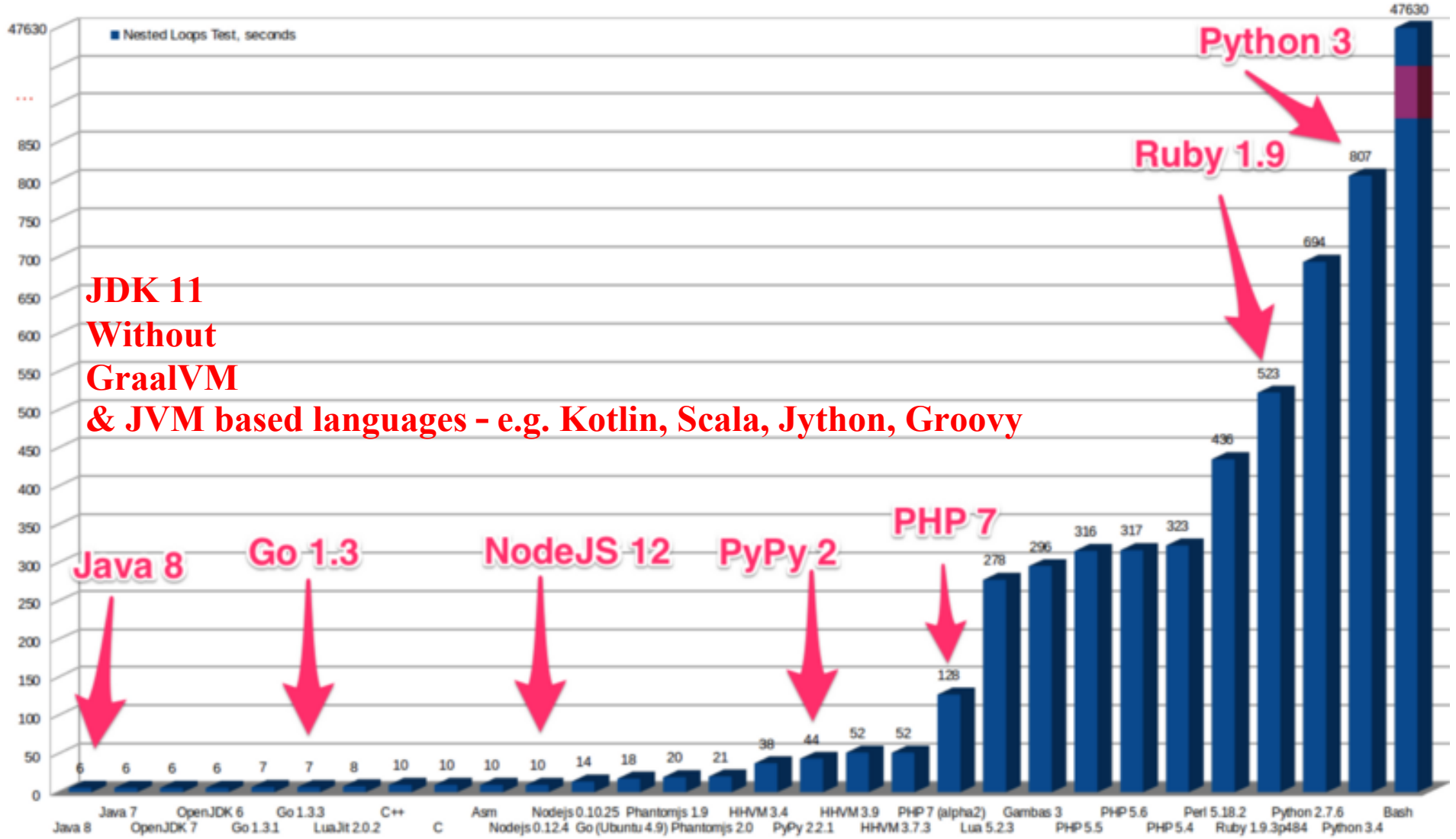

```
  {
    "cell_type": "markdown",
    "metadata": {
      "colab_type": "text",
      "id": "eTZpu6-mNG9j"
    },
    "source": [
      "## Hiding code\n",
      "[Forms example](https://colab.research.google.com/notebooks/form
      "\n",
      "The cell below will automatically load with code hidden. Double
    ]
  },
  {
    "cell_type": "code",
    "metadata": {
      "colab_type": "code",
      "id": "aXJaZqFpNK-q",
      "colab": {}
    },
    "source": [
      "#@title Give me a name {display-mode: \"form\"}\n",
      "\n",
      "# This code will be hidden when the notebook is loaded.\n"
    ],
    "execution_count": 0,
    "outputs": []
  }
]
Cristians-MacBook-Pro-2:JavaAI ctoma$ pwd
/Users/ctoma/Data/School/F0100_PhD/F0100_Doctorat/JavaAI
Cristians-MacBook-Pro-2:JavaAI ctoma$ appletviewer testHand_2.html
In BackProp constructor
Cristians-MacBook-Pro-2:JavaAI ctoma$ javac -version
javac 1.8.0_74
Cristians-MacBook-Pro-2:JavaAI ctoma$ export PATH=.:$JAVA_HOME/bin:$PATH
Cristians-MacBook-Pro-2:JavaAI ctoma$ echo $PATH
.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:/usr/local/bin:/usr/bin:/b
Cristians-MacBook-Pro-2:JavaAI ctoma$ export PATH=.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
Cristians-MacBook-Pro-2:JavaAI ctoma$ echo $PATH
.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
Cristians-MacBook-Pro-2:JavaAI ctoma$ appletviewer testHand_2.html
In BackProp constructor
1001
Run button pressed

1001
Reset button pressed
```



ISBN-10: 1558604200, ISBN-13: 978-1558604209 | https://www.amazon.com/Intelligent-Java-Applications-Internet-Intranets/dp/1558604200

Why Python for ML learning (students) and Java/Kotlin/Scala or C/C++ for Back-end Production?

Speed Benchmarking



Are AI/ML/NN algorithms DETERMINISTIC or STOCHASTIC?

Google
&
https://www.quora.com/Are-neural-networks-stochastic-or-deterministic

"Thus once the weights and the structure of a Neural Network are fixed (let's say once it has been trained), it becomes a deterministic function." …

"it's just that the training procedure is stochastic."

Is AI deterministic?

Deterministic AI environments are those on which the outcome can be determined base on a specific state. In other words, **deterministic** environments ignore uncertainty. Most real world **AI** environments are not **deterministic**. Instead, they can be classified as stochastic. Jan 12, 2017

medium.com › 6-types-of-artificial-intelligence-environments-825e3c47...

6 Types of Artificial Intelligence Environments - Jesus Rodriguez ...

Search for: Is AI deterministic?

Is machine learning non deterministic?

Machine learning is stochastic, not **deterministic**. Jul 29, 2019

towardsdatascience.com › the-limitations-of-machine-learning-a00e0c30...

The Limitations of Machine Learning - Towards Data Science

Search for: Is machine learning non deterministic?

Can artificial intelligence have free will?

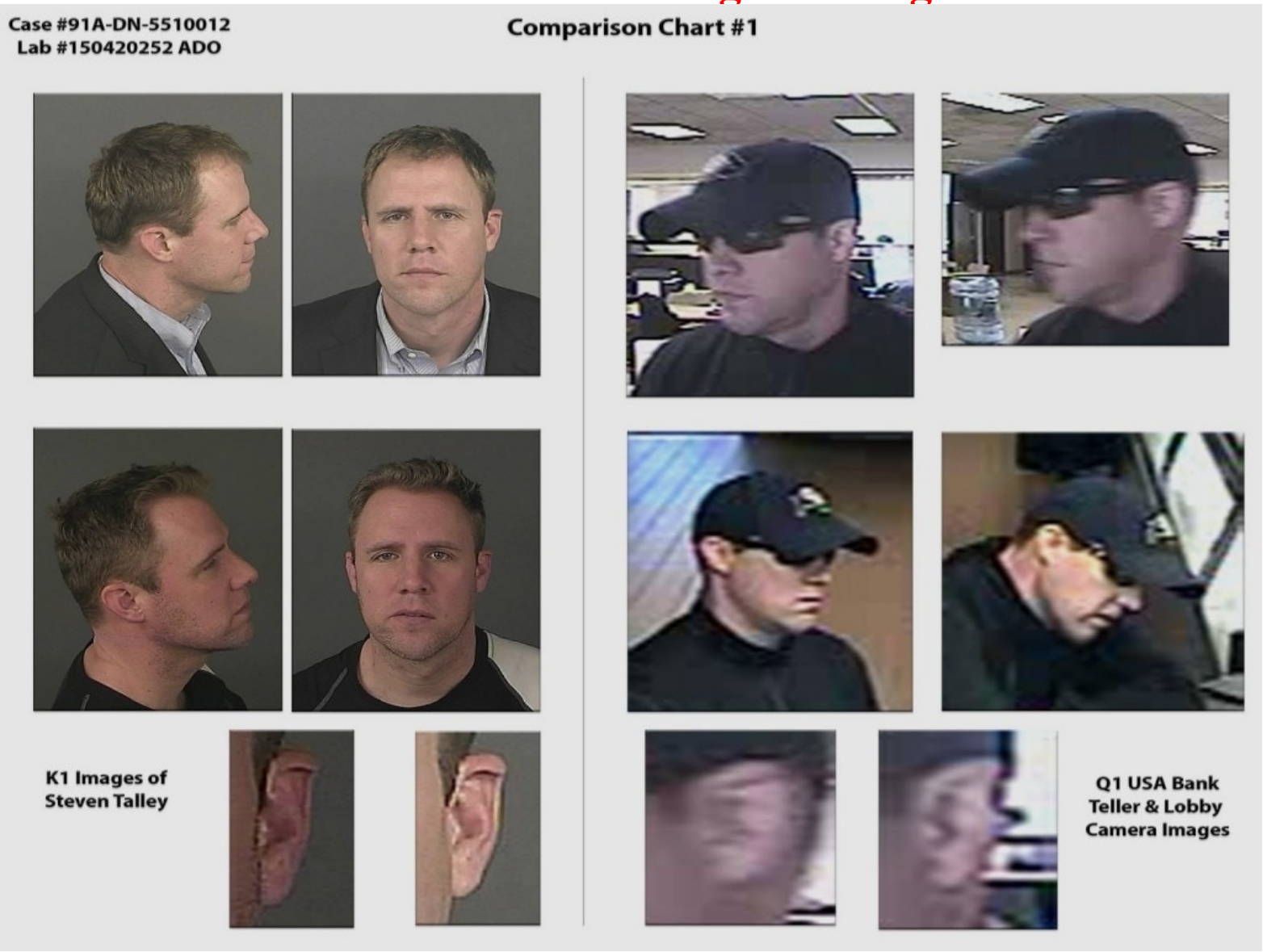
In order for robots to rule human beings, they **will need** to possess the autonomy to take decisions by themselves. They should be able to make their own choice consciously and take initiative. And for all these, they should **have** "**free will**". Dec 28, 2017

beytulhikme.org › Makaleler › 1657575238_05_Cevik_(75-87)

Will It Be Possible for Artificial Intelligence Robots to Acquire Free ...

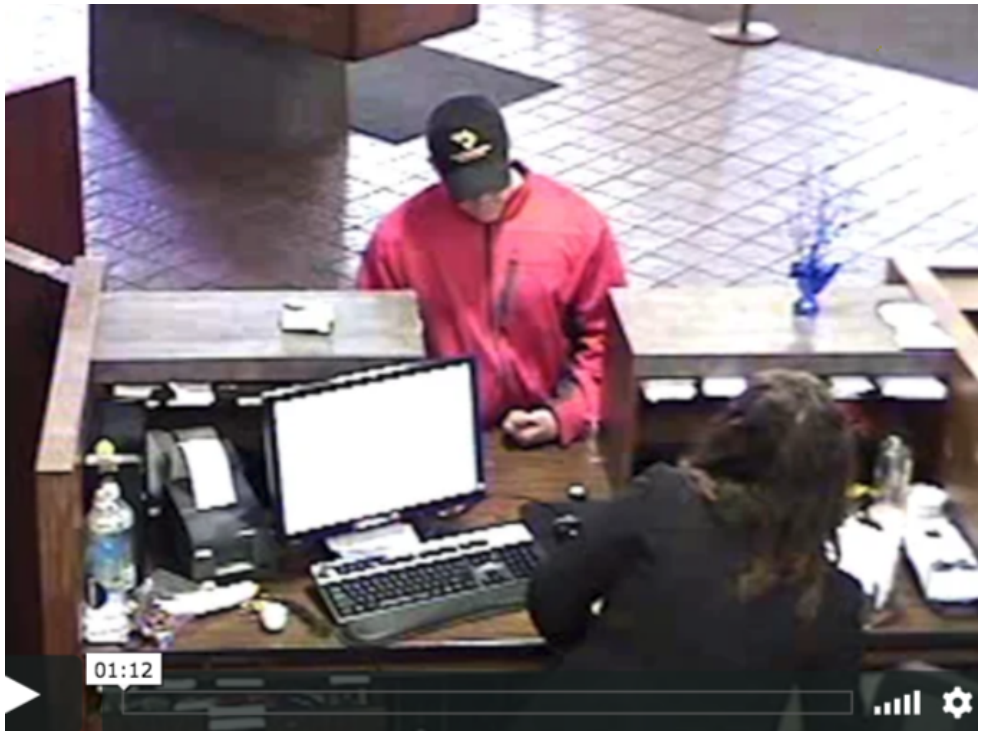
Artificial Intelligence / Neural Networks & Machine Learning

Are AI/ML/NN/Face Recognition algorithms DETERMINISTIC or STOCHASTIC?



https://theintercept.com/2016/10/13/how-a-facial-recognition-mismatch-can-ruin-your-life/

TEVE TALLEY IS hardly the first person to be arrested for the errors of a forensic evaluation.



What is a Fin-Tech? FORBES: Examples of Fin-Techs

What are examples of Fintech companies?

Examples of Fintech-related companies or products include:

- Payment infrastructure, processing and issuance, such as services provided by Square, Ant Financial, Revolut, and Stripe.
- Stock trading apps from Robinhood, TD Ameritrade, and Schwab.
- Alternative lending marketplaces, such as Prosper, LendingClub, and OnDeck.

More items... • Oct 12, 2019

www.forbes.com › sites › allbusiness › 2019/10/12 › fintech-startup-co...

10 Key Issues For Fintech Startup Companies - Forbes

What is a Fin-Tech? Examples of Fin-Techs

According to investopedia.com

View 2+ more

Ant Financial | Adyen | Qudian | Xero | SoFi | Lufax | Avant

The World's Top 10 FinTech Companies

- Ant Financial.
- Adyen.
- Qudian.
- Xero.
- SoFi.
- Lufax.
- Avant.
- ZhongAn.

More items... • Oct 13, 2019

www.investopedia.com › Investing › Stocks › Tech Stocks

The World's Top 10 FinTech Companies - Investopedia

This is the latest accepted revision, reviewed on 21 January 2020. **Revolut** Ltd is a UK financial technology company and neobank. Its services include a prepaid debit card (MasterCard or Visa), fee free currency exchange, commission free stock trading, cryptocurrency exchange and peer-to-peer payments.

Services: Peer-to-peer payments, Currency Ex...

Products: Current Accounts, Debit Cards, Insur...

Headquarters: London, England, UK

en.wikipedia.org › wiki › Revolut

Revolut - Wikipedia

What is a Fin-Tech? Examples of Fin-Techs

Fintech companies encompass a broad landscape of businesses, generally around financial-oriented services and products. ***Examples of Fintech-related companies*** or products include:

- **Payment infrastructure**, processing and issuance, such as services provided by Square, Ant Financial, Revolut, and Stripe
- Stock trading apps from Robinhood, TD Ameritrade, and Schwab
- Alternative lending marketplaces, such as Prosper, LendingClub, and OnDeck
- **Cryptocurrencies and digital cash**, a prime example of which is Bitcoin
- **Blockchain technology**, such as Ethereum
- **Insurtech**, which seeks to modernize and simplify the insurance industry, with companies such as Lemonade, Oscar, and Fabric
- **Money transfer and remittances**, including services from TransferWise, **PayPal**, and Venmo
- **Mortgage lending**, such as through LendingHome and Better Mortgage
- **Robo investment advisors**, such as Betterment and Wealthfront
- **Neobanks**, including Chime, N26, and Monzo
- Credit reporting, such as Credit Karma
- Online business loan providers such as Lendio and Kabbage
- Small business credit cards, payments, and financing, such as through Brex and Fundbox
- **Financial cybersecurity companies** seeking to protect institutions from money laundering, chargeback risk. and cybercrimes, such as Forter, EverCompliant, and CrowdStrike.
- **Infrastructure and software to power financial applications**, such as from **Plaid**

Top 10 AI / ML Applications for Finance & Fin-Techs

1. Portfolio Management – Robo-Advisors

2. Algorithmic Trading

3. High-Frequency Trading (HFT)

4. Fraud Detection

5. Loan/ Insurance Underwriting

6. Risk Management

7. Chatbots

8. Document Analysis
9. Trade Settlements
10. Money-Laundering Prevention

… Future Applications of Artificial Intelligence in Finance

Top 10 AI / ML Applications for Fin-Techs

#1. Digital Financial Coach/Advisor

Transactional bots are one of the most popular use cases in AI, probably because the range of applications is so broad — across all industries, at several levels.

#2. Transaction search & visualization

Chat-bots can also be used in banking to focus on search tasks.
Managers give access to the bot to the users' transactional data (banking transactions), and it uses NLP to detect the meaning of the request sent by the user (a search query). Requests could be related to balance inquiries, spending habits, general account information and more. The bot then processes the requests and displays the results.

#3. Client Risk Profile

A critical part of banks and insurance companies' job is the profiling of clients based on their risk score.
AI is an excellent tool for this as it can automate the categorization of clients depending on their risk profile, from low to high.

Top 10 AI / ML Applications for Fin-Techs

#4. Underwriting, Pricing & Credit Risk Assessment

Insurance companies offer underwriting services, mainly for loans and investments.
An AI-powered model can provide an instantaneous assessment of a client's credit risk, which then allows advisors to craft the most adapted offer.

#5. Automated Claims Processes

The insurance industry as we know it functions on a standard process: clients subscribe insurance, for which they pay. If the customer has a problem (sickness for health insurance, a car accident for automobile insurance, water damage for a housing insurance), she needs to activate her coverage by filing a claim. This process is often lengthy and complicated.
Transactional bots can transform the user experience into a more pleasant process.

#6. Contract Analyzer

Contract analysis is a repetitive internal task in the finance industry. Managers and advisors can delegate this routine task to a machine learning model.
Optical Character Recognition (OCR) can be used to digitize hard copy documents. An NLP model with layered business logic can then interpret, record, and correct contracts at high speed.

Top 10 AI / ML Applications for Fin-Techs

#7. Churn Prediction

Churn (or attrition) rate is a key KPI across all industries and businesses. Companies need to retain clients, and to do so, predicting coming churn can be extremely helpful to take preventive actions.
AI can support managers in this mission by providing a prioritized list of clients who show signs of considering to cancel their policy. The manager can then address this list accordingly: give a higher degree of service or improved offering.

#8. Algorithmic Trading — the most advanced ML you will never see.

Most applications of algorithmic trading happen behind the closed doors of investment banks or hedge funds. **Trading, very often, comes to analyzing data and making decisions, fast. Machine learning algorithm excels in analyzing data, whatever its size and density.**

#9. Valuation Models

Valuation models are usually applications for investment and banking in general.
The model can quickly calculate the valuation of an asset using data points around the asset and historical examples. These data points are what a human would use to value the asset (ex: the creator of a painting), but the model learns which weights to assign to each data point by using historical data.

Top 10 AI / ML Applications for Fin-Techs

#10. Augmented research tools

In investment finance, a large portion of time is spent doing research. New machine learning models increase the available data around given trade ideas.

Sentiment analysis can be used for due diligence about companies and managers. It allows an analyst to view at a glance the tone/mood of large sets of text data such as news or financial reviews. It can also provide insight into how a manager reflects their company performance.

Satellite Image Recognition can give a researcher insight into many real-time data points. Examples of such are parking lot traffic in specific locations (retailer shops, for example) or freighter traffic in the ocean. From this data, the model and the analyst can derive business insights such as the frequency of shopping at specific stores of the retailers mentioned above, the flow of shipments, routes, and so on. Advanced **NLP** techniques can help a researcher analyze a company financial reports quickly. Pulling out key topics that are of most interest to the firm.

Other data science techniques can also format and standardize financial statements.

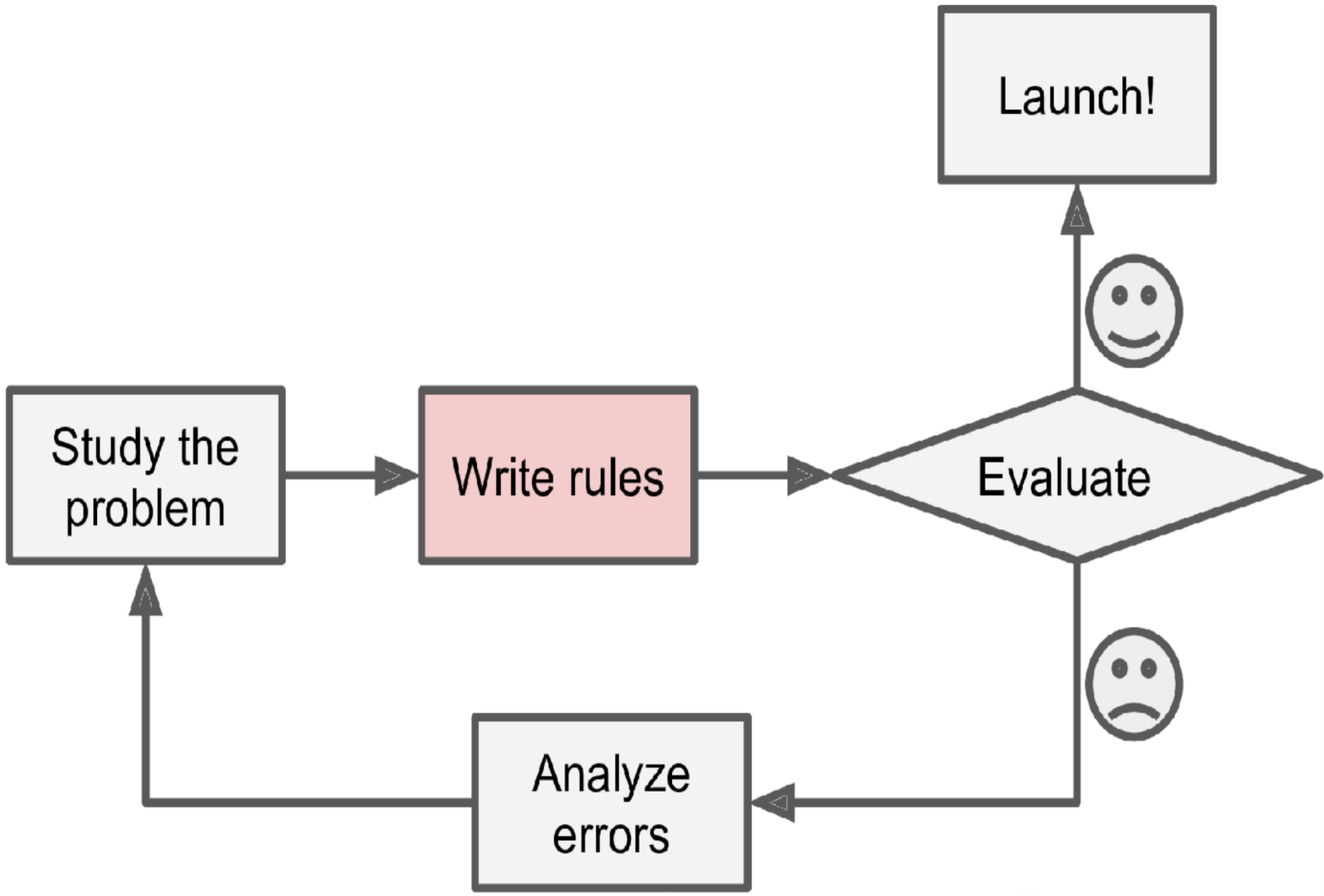
Artificial Intelligence & Machine Learning Overview

Why Use Machine Learning?

Consider how you would write a spam filter using traditional programming techniques:

First you would consider what spam typically looks like. You might notice that some words or phrases (such as “4U,” “credit card,” “free,” and “amazing”) tend to come up a lot in the subject line. Perhaps you would also notice a few other patterns in the sender’s name, the email’s body, and other parts of the email.

You would write a detection algorithm for each of the patterns that you noticed, and your program would flag emails as spam if a number of these patterns were detected. You would test your program and repeat steps 1 and 2 until it was good enough to launch. Since the problem is difficult, your program will likely become a long list of complex rules—pretty hard to maintain.



The traditional approach

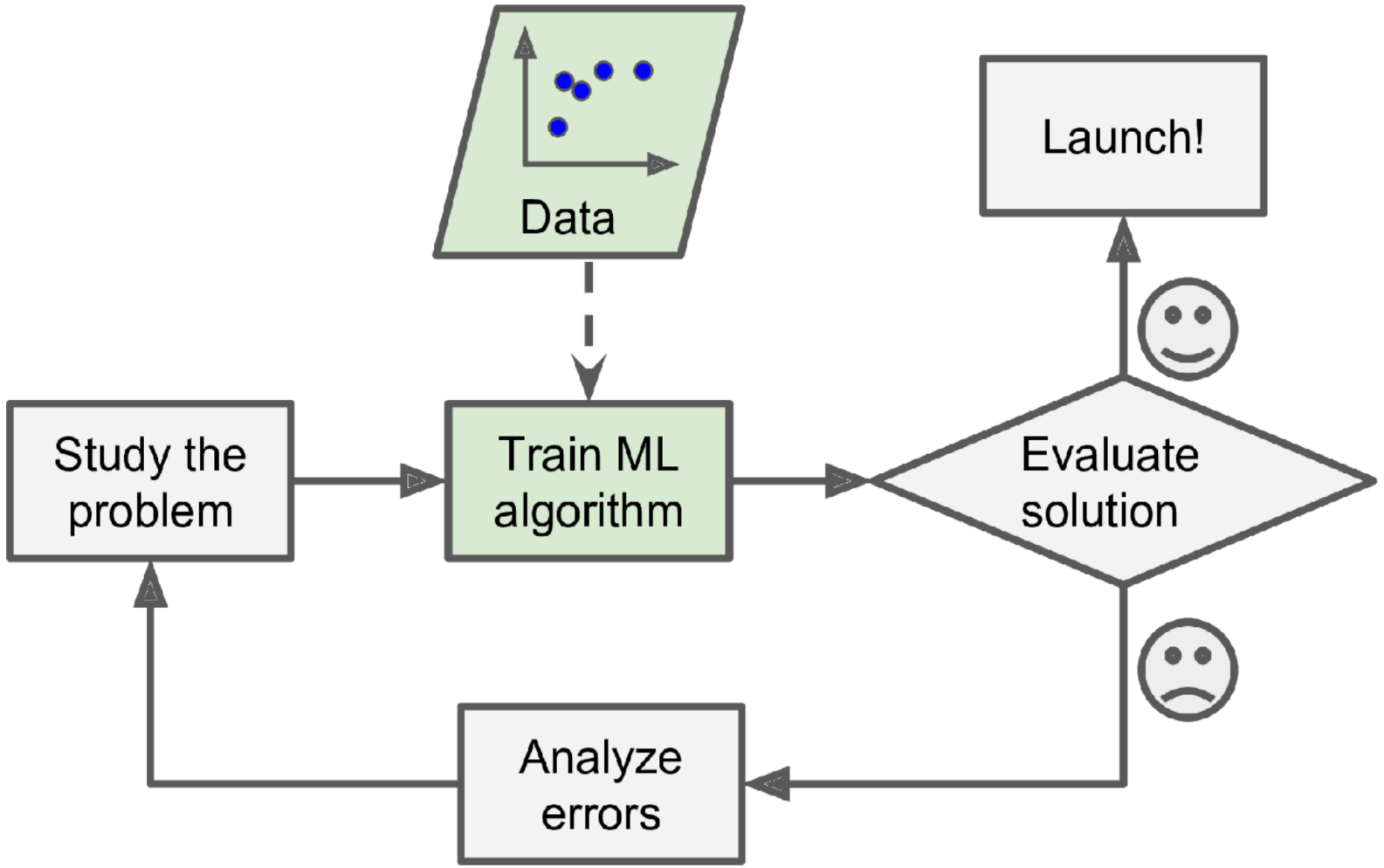
Artificial Intelligence & Machine Learning Overview

In contrast, a spam filter based on Machine Learning techniques automatically learns which words and phrases are good predictors of spam by detecting unusually frequent patterns of words in the spam examples compared to the ham examples – fig. The program is much shorter, easier to maintain, and most likely more accurate.

What if spammers notice that all their emails containing "4U" are blocked? They might start writing "For U" instead.

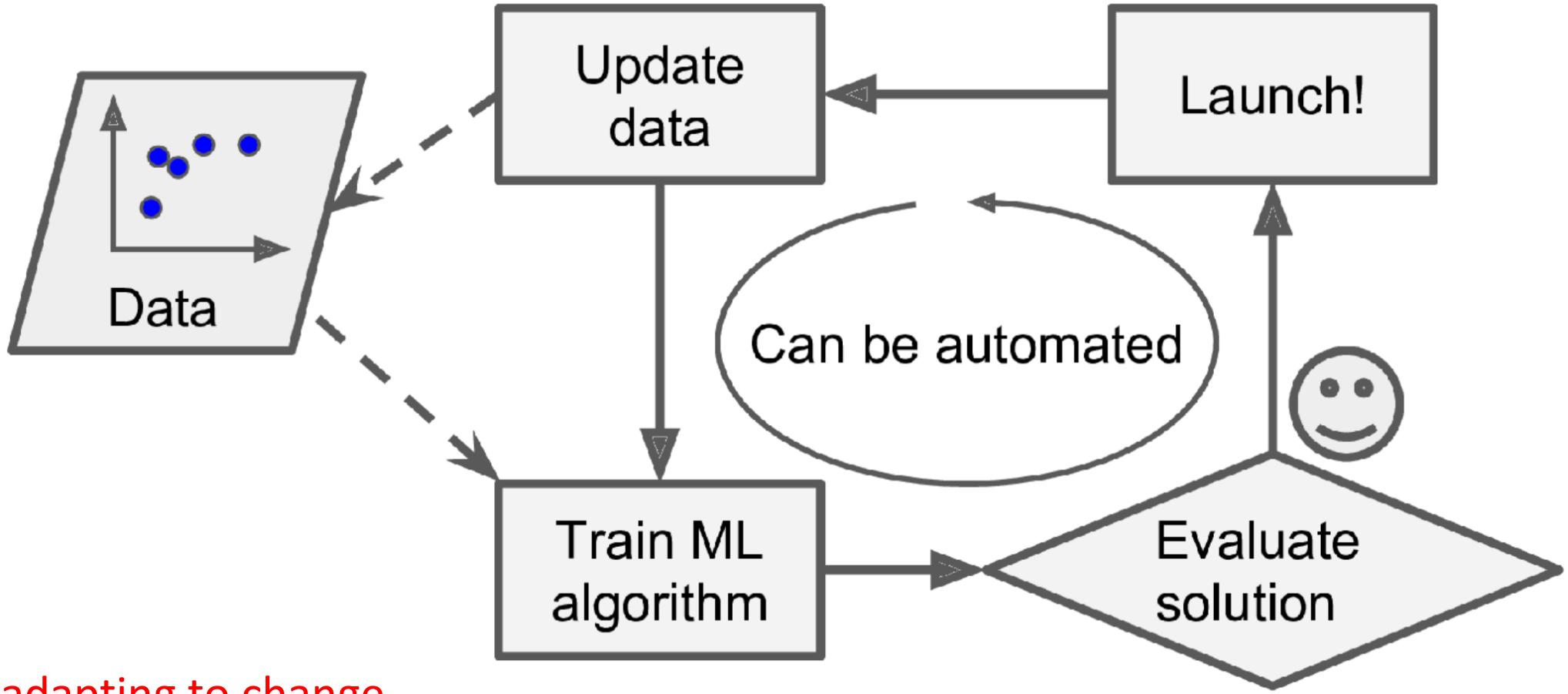
A spam filter using traditional programming techniques would need to be updated to flag "For U" emails. If spammers keep working around your spam filter, you will need to keep writing new rules forever.

The Machine Learning approach



Artificial Intelligence & Machine Learning Overview

In contrast, a spam filter based on Machine Learning techniques automatically notices that “For U” has become unusually frequent in spam flagged by users, and it starts flagging them without your intervention:



ML Automatically adapting to change

Artificial Intelligence & Machine Learning Overview

Another area where Machine Learning shines is for problems that either are too complex for traditional approaches or have no known algorithm.

For example, consider speech recognition / face recognition.

Say you want to start simple and write a program capable of distinguishing the words “one” and “two.” You might notice that the word “two” starts with a high-pitch sound (“T”), so you could hardcode an algorithm that measures high-pitch sound intensity and use that to distinguish ones and twos—but obviously this technique will not scale to thousands of words spoken by millions of very different people in noisy environments and in dozens of languages.

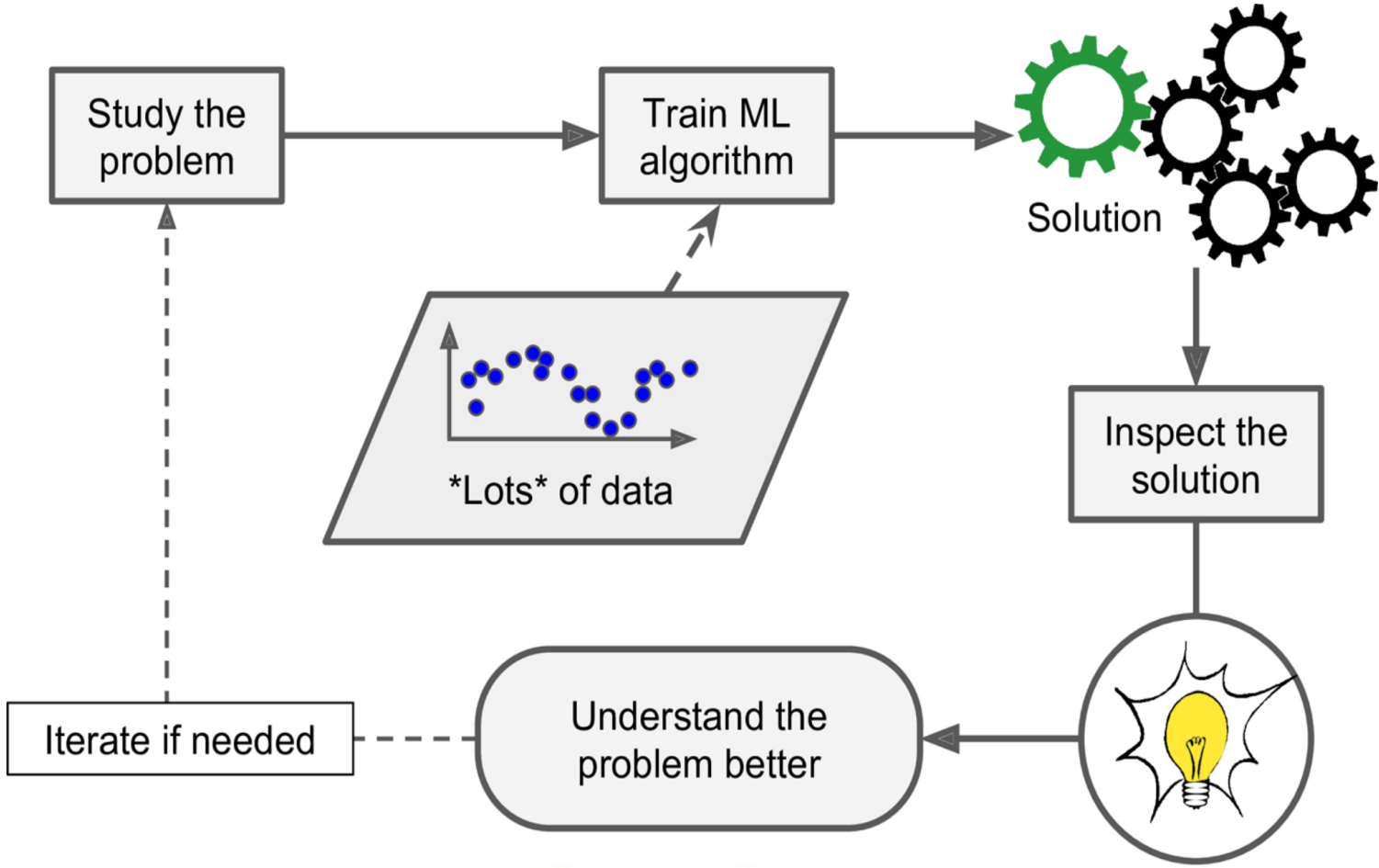
The best solution (at least today) is to write an algorithm that learns by itself, given many example recordings for each word.

Artificial Intelligence & Machine Learning Overview

Finally, Machine Learning can help humans learn – figure.

ML algorithms can be inspected to see what they have learned (although for some algorithms this can be tricky). For instance, once a spam filter has been trained on enough spam, it can easily be inspected to reveal the list of words and combinations of words that it believes are the best predictors of spam.

Sometimes, this will reveal unsuspected correlations or new trends, and thereby lead to a better understanding of the problem. Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called ***data mining***.



Artificial Intelligence & Machine Learning Overview

To summarize, Machine Learning is great for:

- ***Problems for which existing solutions require a lot of fine-tuning or long lists of rules*: one Machine Learning algorithm can often simplify code and perform better than the traditional approach.**
- ***Complex problems for which using a traditional approach yields no good solution*: the best Machine Learning techniques can perhaps find a solution.**
- ***Fluctuating environments*: a Machine Learning system can adapt to new data.**
- *Getting insights about complex problems and large amounts of data.*

Artificial Intelligence & Machine Learning Systems Types

There are so many different types of Machine Learning systems that it is useful to classify them in broad categories, based on the following criteria:

- A. Whether or not they are trained with human supervision (Supervised, Unsupervised, Semi-supervised, and Reinforcement Learning)

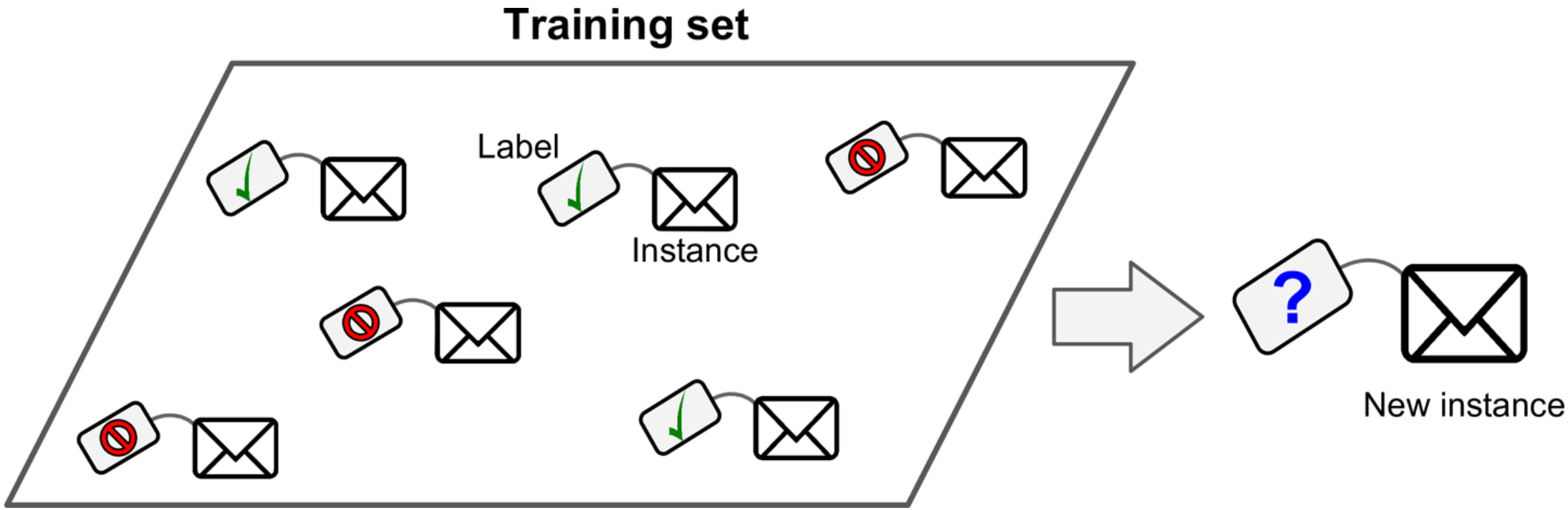
- B. Whether or not they can learn incrementally on the fly (Online versus Batch learning)

- C. Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do (Instance-based versus Model-based learning)

For example, a state-of-the-art spam filter may learn on the fly using a deep neural network model trained using examples of spam and ham; this makes it an online, model-based, supervised learning system.

A.1 Supervised Learning

In supervised learning, the training set you feed to the algorithm includes the desired solutions, called labels

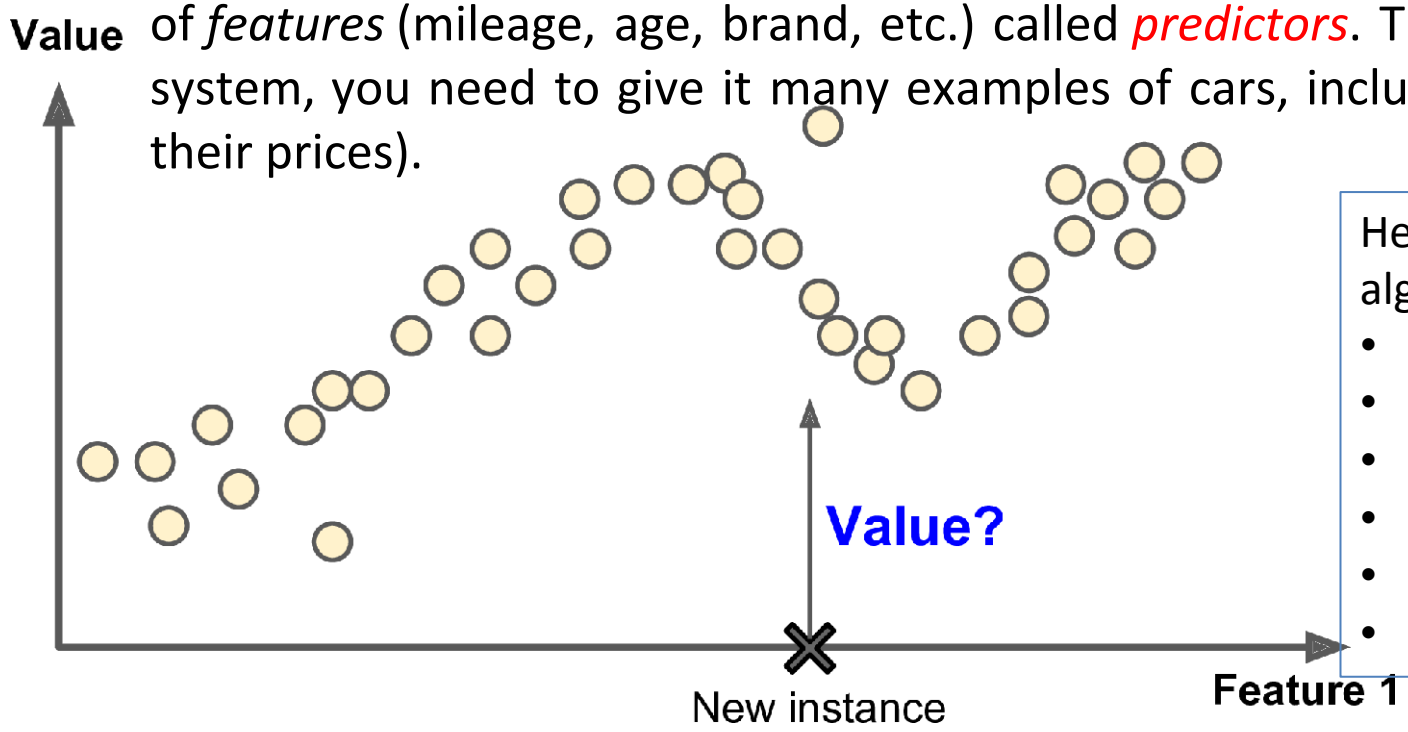


A labeled training set for spam classification (an example of supervised learning)

A.1 Supervised Learning

A typical supervised learning task is *classification*. The spam filter is a good example of this: it is trained with many example emails along with their *class* (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a *target* numeric value, such as the price of a car, given a set of *features* (mileage, age, brand, etc.) called *predictors*. This sort of task is called *regression*. To train the system, you need to give it many examples of cars, including both their predictors and their labels (i.e., their prices).



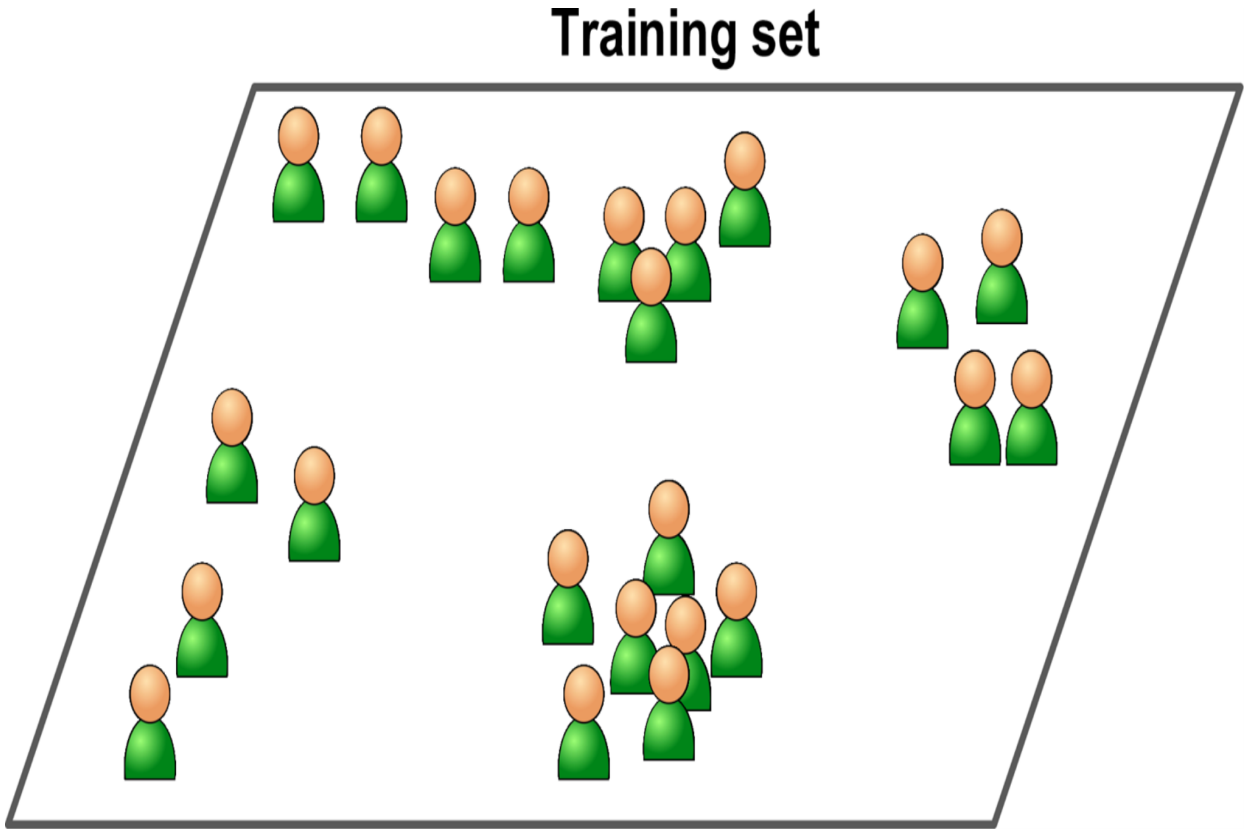
Here are some of the most important supervised learning algorithms:

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural networks

A regression problem: predict a value, given an input feature (there are usually multiple input features, and sometimes multiple output values)

A.2 Unsupervised Learning

In *unsupervised learning*, as you might guess, the training data is unlabeled. The system tries to learn without a teacher.



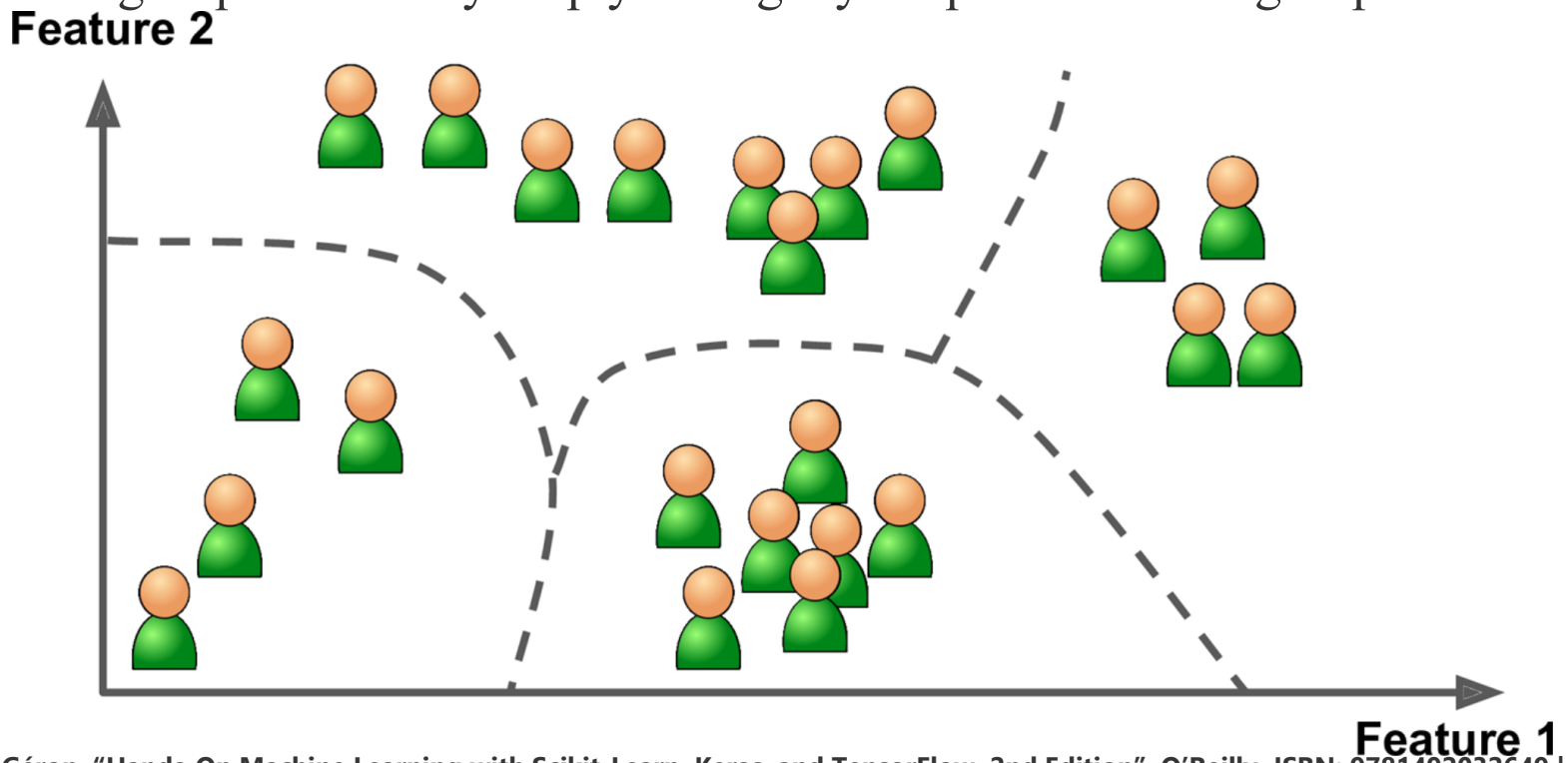
An unlabeled training set for unsupervised learning

Here are some of the most important unsupervised learning algorithms (most of these are covered in Chapters 8 / 9):

- **Clustering**
  - K-Means
  - DBSCAN
  - Hierarchical Cluster Analysis (HCA)
- **Anomaly detection and novelty detection**
  - One-class SVM
  - Isolation Forest
- **Visualization and dimensionality reduction**
  - Principal Component Analysis (PCA)
  - Kernel PCA
  - Locally Linear Embedding (LLE)
  - t-Distributed Stochastic Neighbor Embedding (t-SNE)
- **Association rule learning**
  - Apriori
  - Eclat

A.2 Unsupervised Learning

For example, say you have a lot of data about your blog's visitors. You may want to run a *clustering* algorithm to try to detect groups of similar visitors. At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40% of your visitors are males who love comic books and generally read your blog in the evening, while 20% are young sci-fi lovers who visit during the weekends. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.



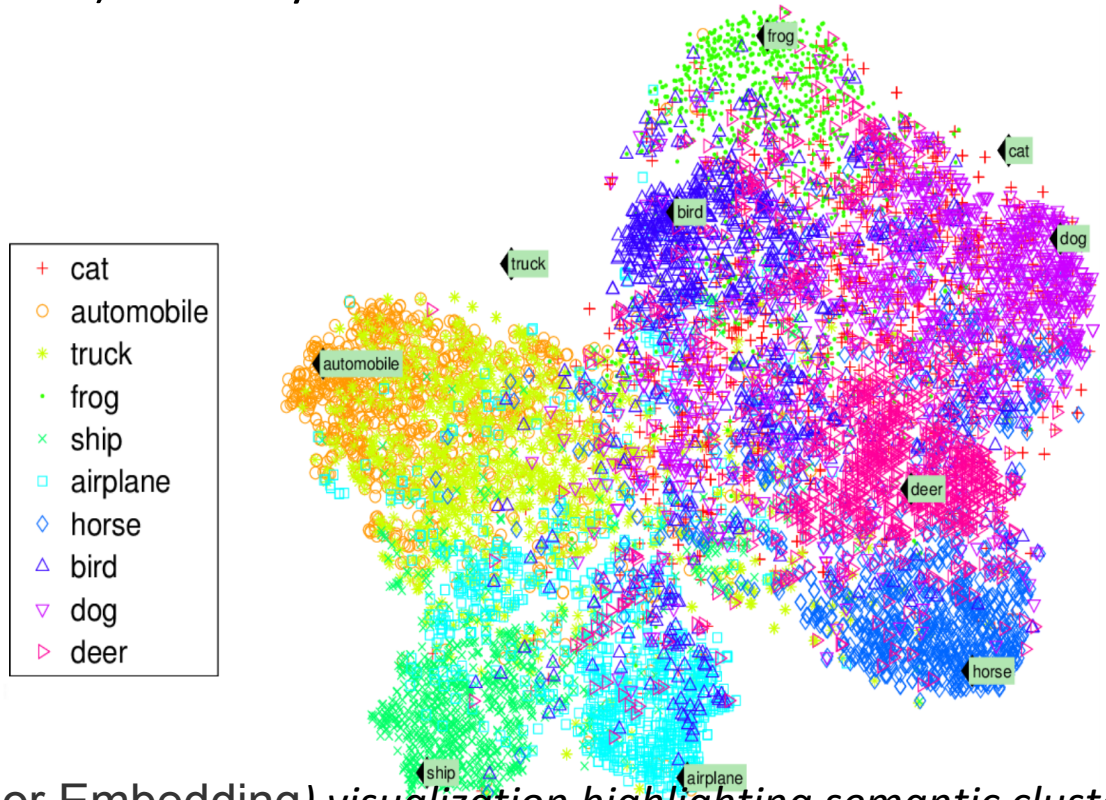
Artificial Intelligence & Machine Learning Terminology

A.2 Unsupervised Learning

Visualization algorithms are also good examples of unsupervised learning algorithms: you feed them a lot of complex and unlabeled data, and they output a 2D or 3D representation of your data that can easily be plotted. These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization) so that you can understand how the data is organized and perhaps identify unsuspected patterns.

A related task is *dimensionality reduction*, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.

In Machine Learning an *attribute* is a data type (e.g., "mileage"), while a *feature* has several meanings, depending on the context, but generally means an attribute plus its value (e.g., "mileage = 15,000"). Many people use the words *attribute* and *feature* interchangeably.

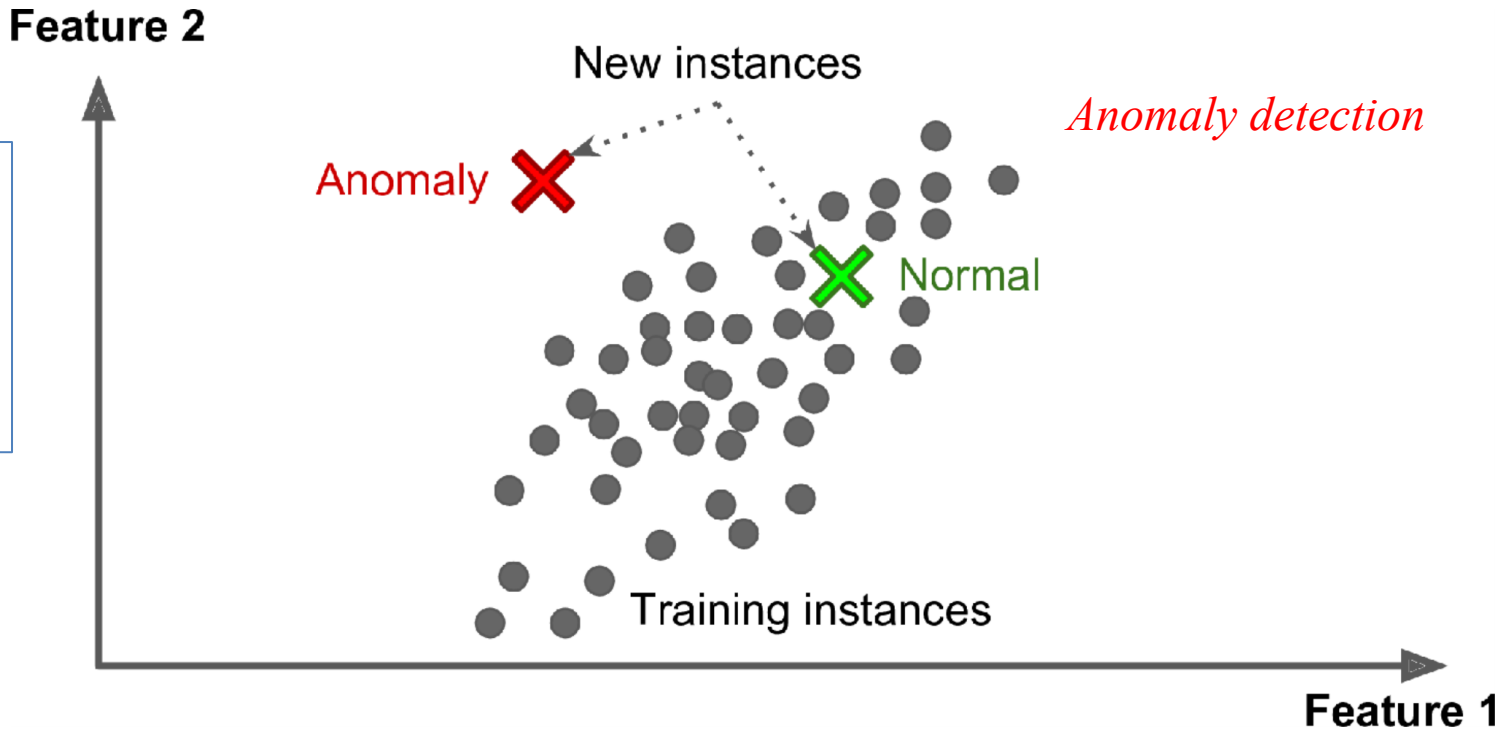


*Example of a t-SNE (*t-Distributed Stochastic Neighbor Embedding*) visualization highlighting semantic cluste*

A.2 Unsupervised Learning

Yet another important unsupervised task is anomaly detection: for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is shown mostly normal instances during training, so it learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly. A very similar task is novelty detection: it aims to detect new instances that look different from all instances in the training set. This requires having a very "clean" training set, devoid of any instance that you would like the algorithm to detect.

Finally, another common unsupervised task is **association rule learning**, in which the goal is to dig into large amounts of data and discover interesting relations between attributes.

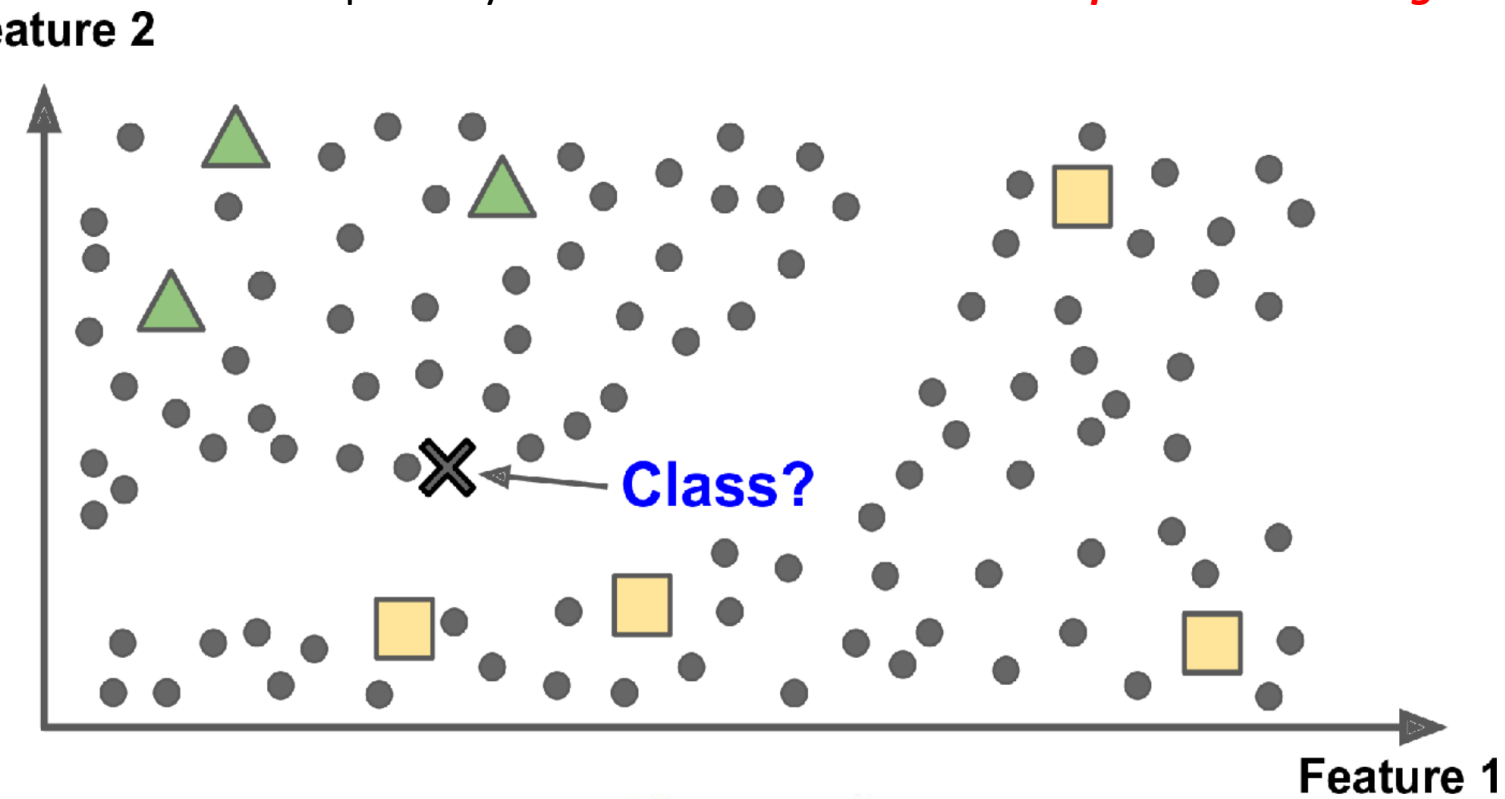


A.3 Semi-supervised Learning

Since labeling data is usually time-consuming and costly, you will often have plenty of unlabeled instances, and few labeled instances. Some algorithms can deal with data that's partially labeled. This is called ***semi-supervised learning***.

Some photo-hosting services, such as Google Photos, are good examples of this. Once you upload all your family photos to the service, it automatically recognizes that the same person A shows up in photos 1, 5, and 11, while another person B shows up in photos 2, 5, and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are. Just add one label per person and it is able to name everyone in every photo, which is useful for searching photos.

Most semi-supervised learning algorithms are combinations of unsupervised and supervised algorithms. For example, ***deep belief networks*** **(DBNs)** are based on unsupervised components called *restricted Boltzmann machines* (RBMs) stacked on top of one another. RBMs are trained sequentially in an unsupervised manner, and then the whole system is fine-tuned using supervised learning techniques.



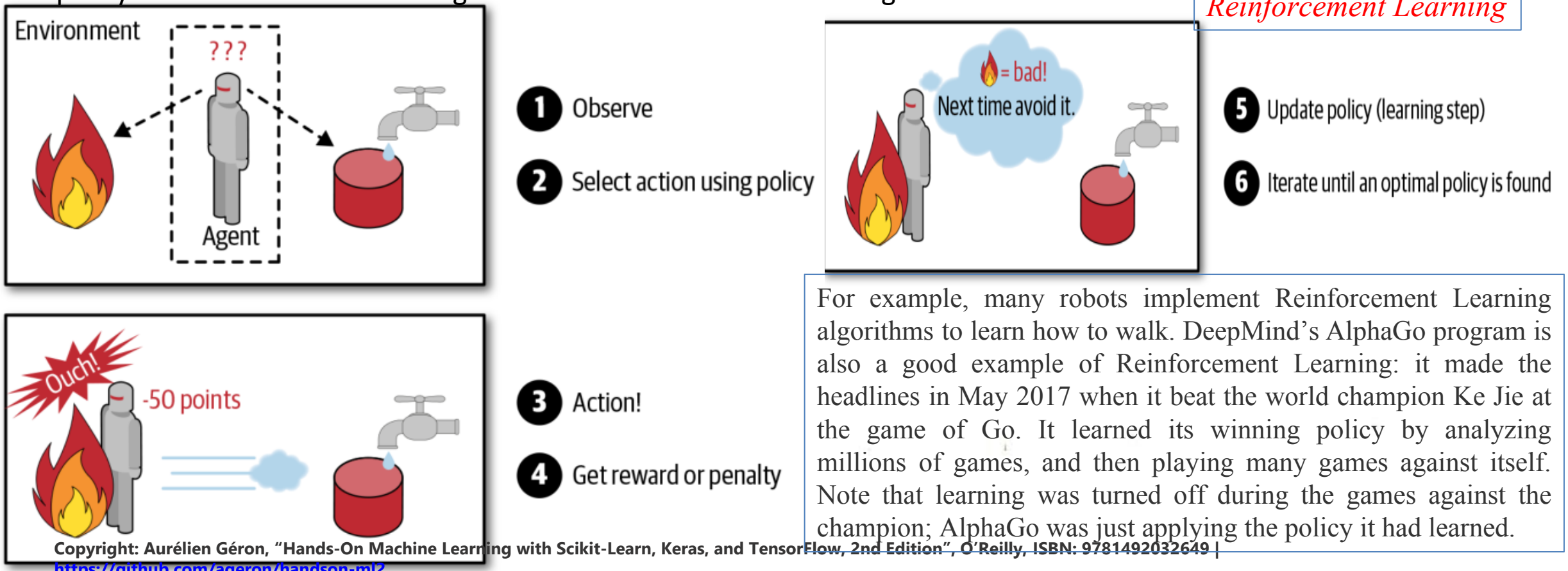
Semi-supervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

Artificial Intelligence & Machine Learning Terminology

A.4 Reinforcement Learning

Reinforcement Learning is a very different beast. The learning system, called an agent in this context, can observe the environment, select and perform actions, and get ***rewards*** in return (or ***penalties*** in the form of negative rewards, as shown in the figure). It must then learn by itself what is the best strategy, called a ***policy***, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

Reinforcement Learning



For example, many robots implement Reinforcement Learning algorithms to learn how to walk. DeepMind's AlphaGo program is also a good example of Reinforcement Learning: it made the headlines in May 2017 when it beat the world champion Ke Jie at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion; AlphaGo was just applying the policy it had learned.

B.1 Batch Learning

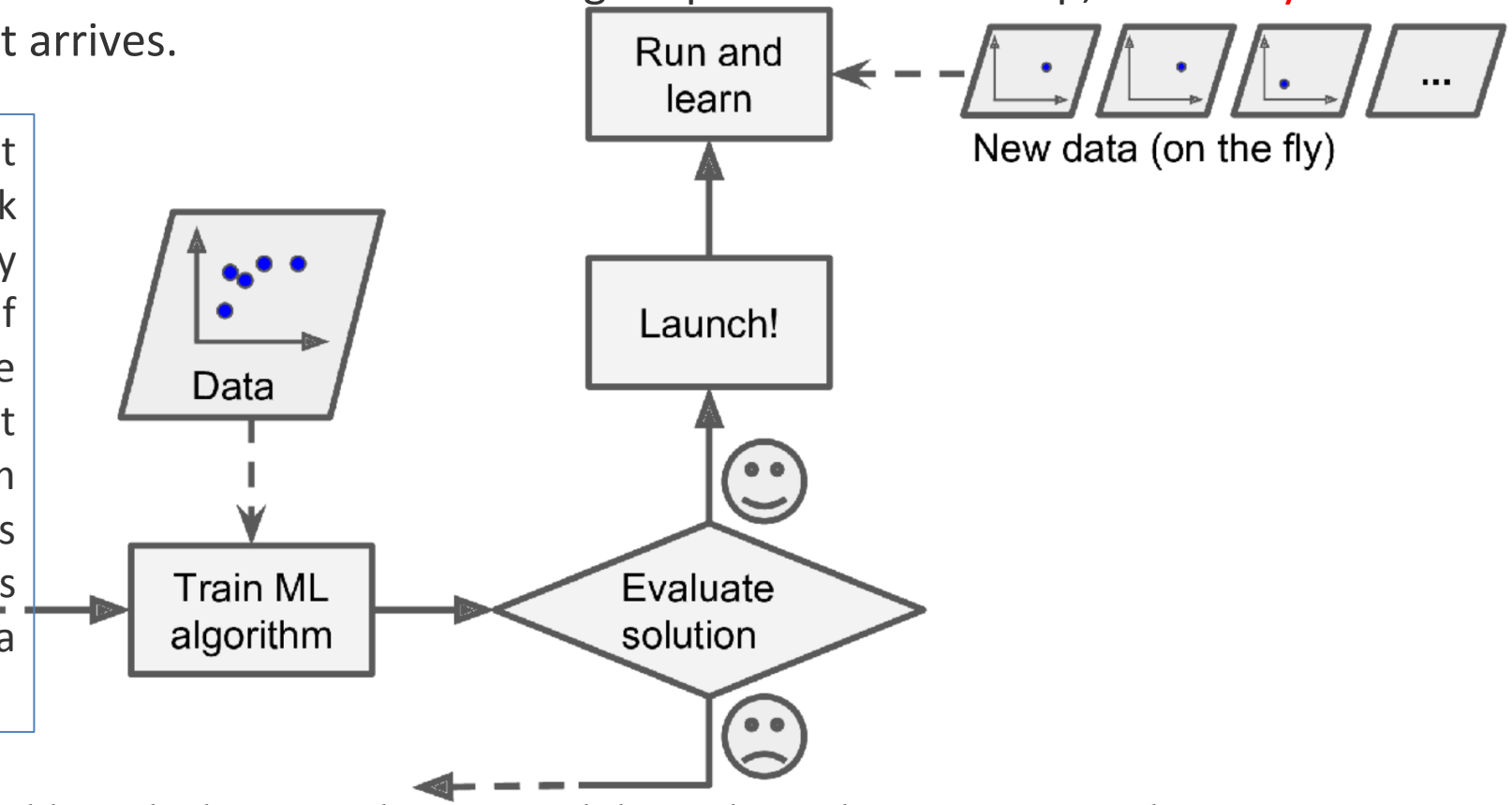
In ***batch learning***, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called ***offline learning***.
If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one.
Fortunately, the whole process of training, evaluating, and launching ***a Machine Learning system can be automated fairly easily*** (as shown in previous slide – figure: **ML Automatically adapting to change**), so ***even a batch learning system can adapt to change***. Simply update the data and train a new version of the system from scratch as often as needed.
This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (e.g., to predict stock prices), then you need a more reactive solution.

Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O, etc.). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge, it may even be impossible to use a batch learning algorithm.
Finally, if your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper.

B.2 Online Learning

In *online learning*, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives.

Online learning is great for systems that receive data as a continuous flow (e.g., stock prices) and need to adapt to change rapidly or autonomously. It is also a good option if you have limited computing resources: once an online learning system has learned about new data instances, it does not need them anymore, so you can discard them (unless you want to be able to roll back to a previous state and "replay" the data). This can save a huge amount of space.



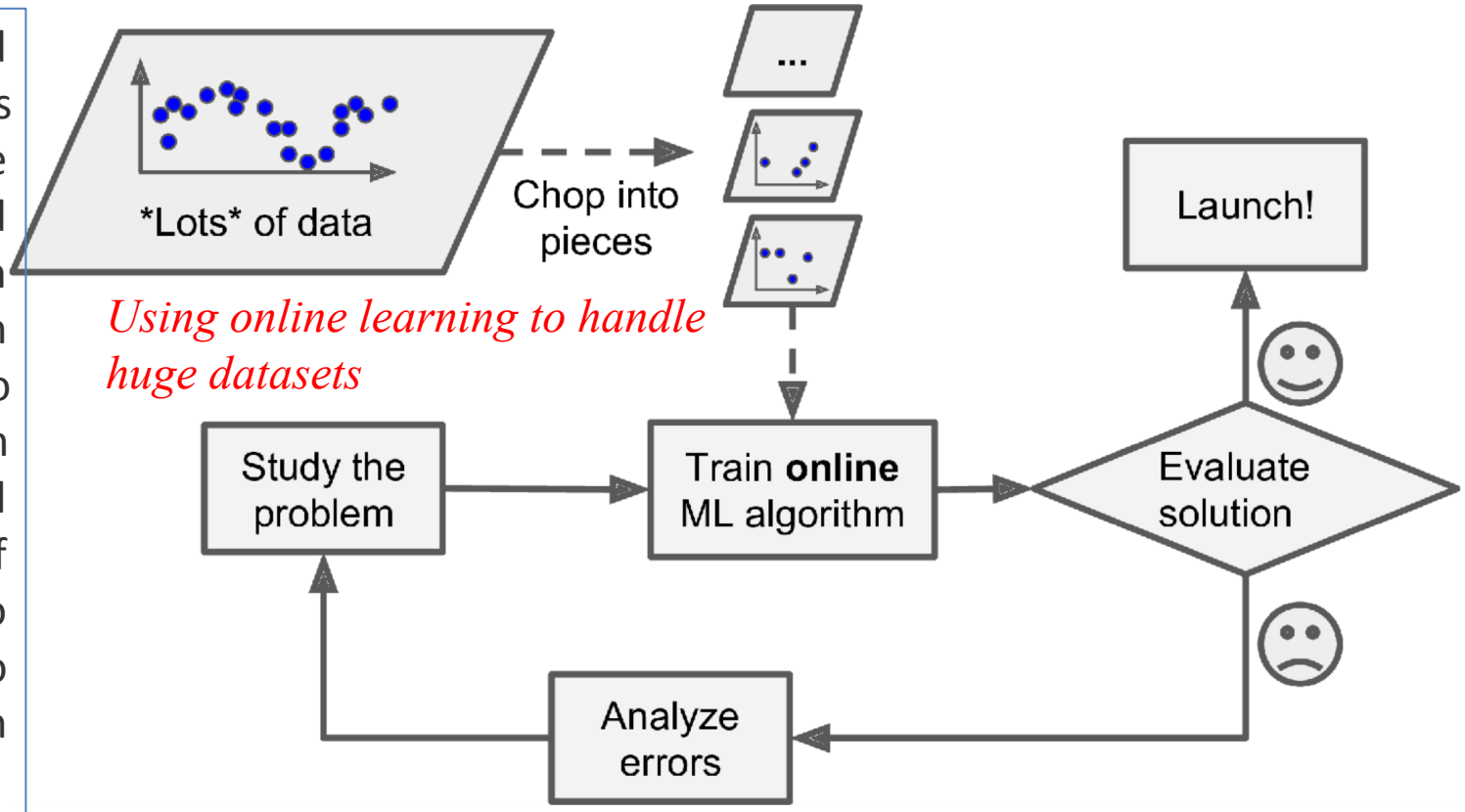
In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in.

Artificial Intelligence & Machine Learning Terminology

B.2 Online Learning

Online learning algorithms can also be used to train systems on huge datasets that cannot fit in one machine's main memory (this is called ***out-of-core* learning**). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data. Warning: *Out-of-core learning is usually done offline (i.e., not on the live system), so online learning can be a confusing name. Think of it as incremental learning.*

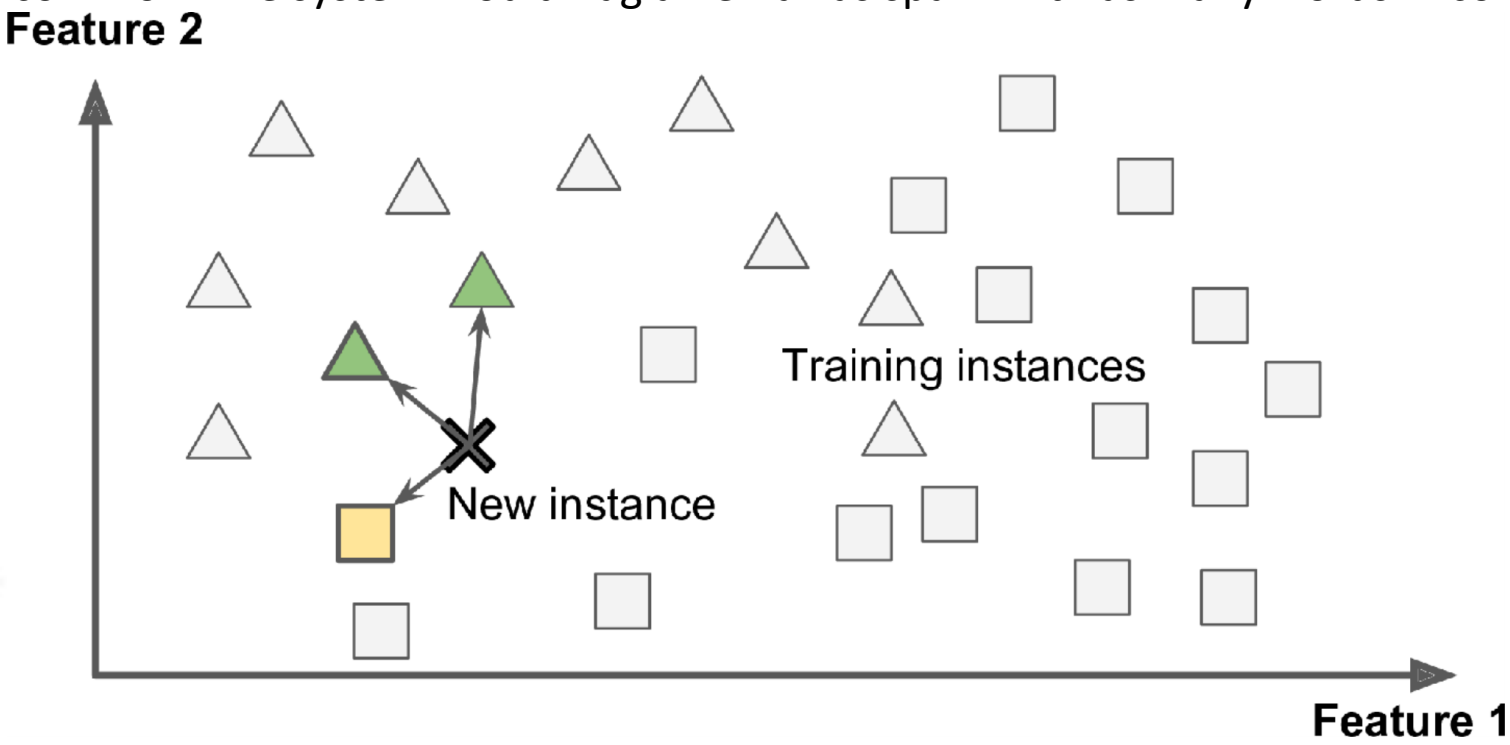
A big challenge with online learning is that if bad data is fed to the system, the system's performance will gradually decline. If it's a live system, your clients will notice. For example, bad data could come from a malfunctioning sensor on a robot, or from someone spamming a search engine to try to rank high in search results. To reduce this risk, you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data (e.g., using an anomaly detection algorithm).



Using online learning to handle huge datasets

C.1 Instance-Based Learning

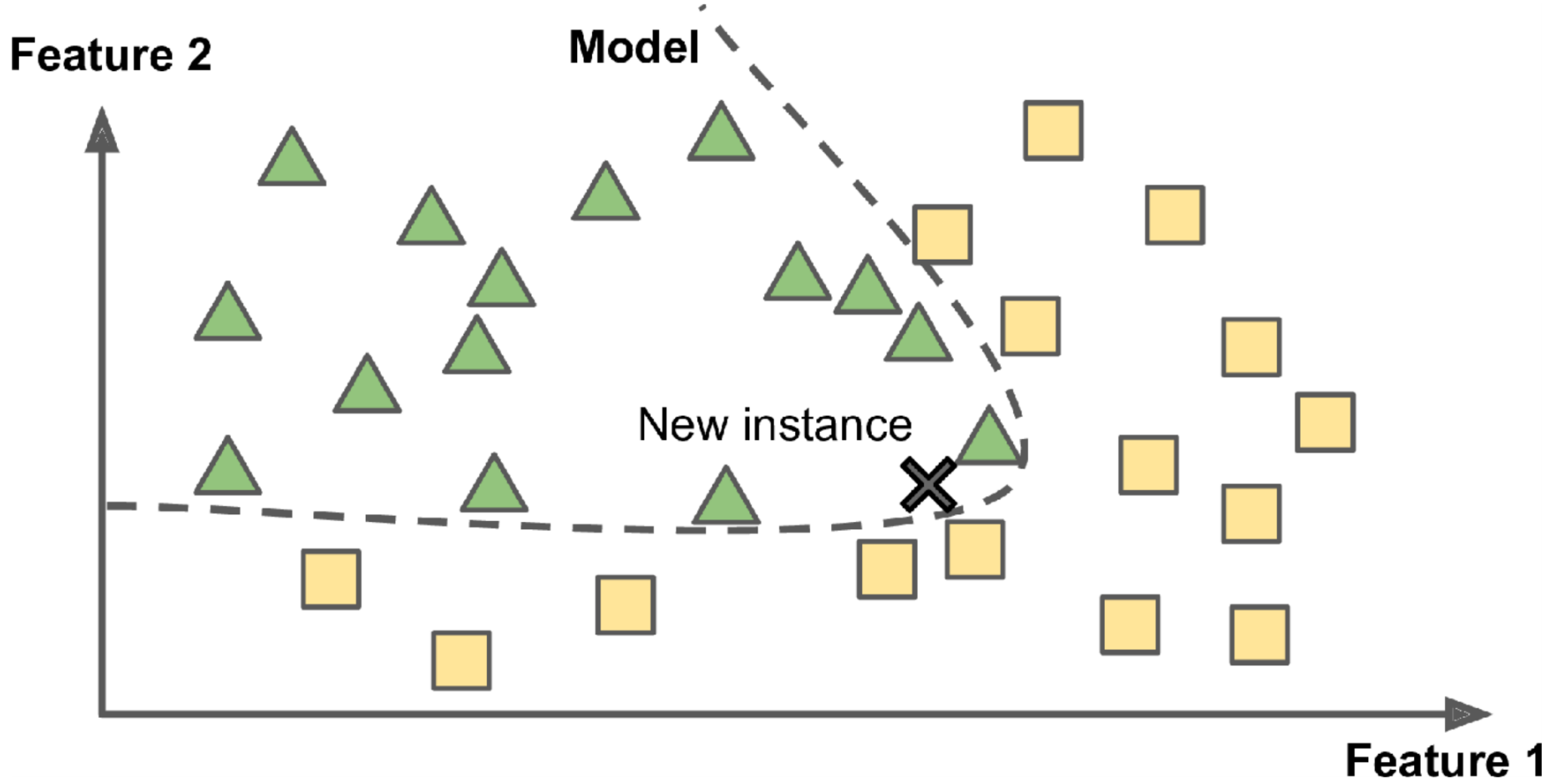
Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails that have already been flagged by users—not the worst solution, but certainly not the best. Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed to also flag emails that are very similar to known spam emails. This requires a ***measure of similarity*** between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email.



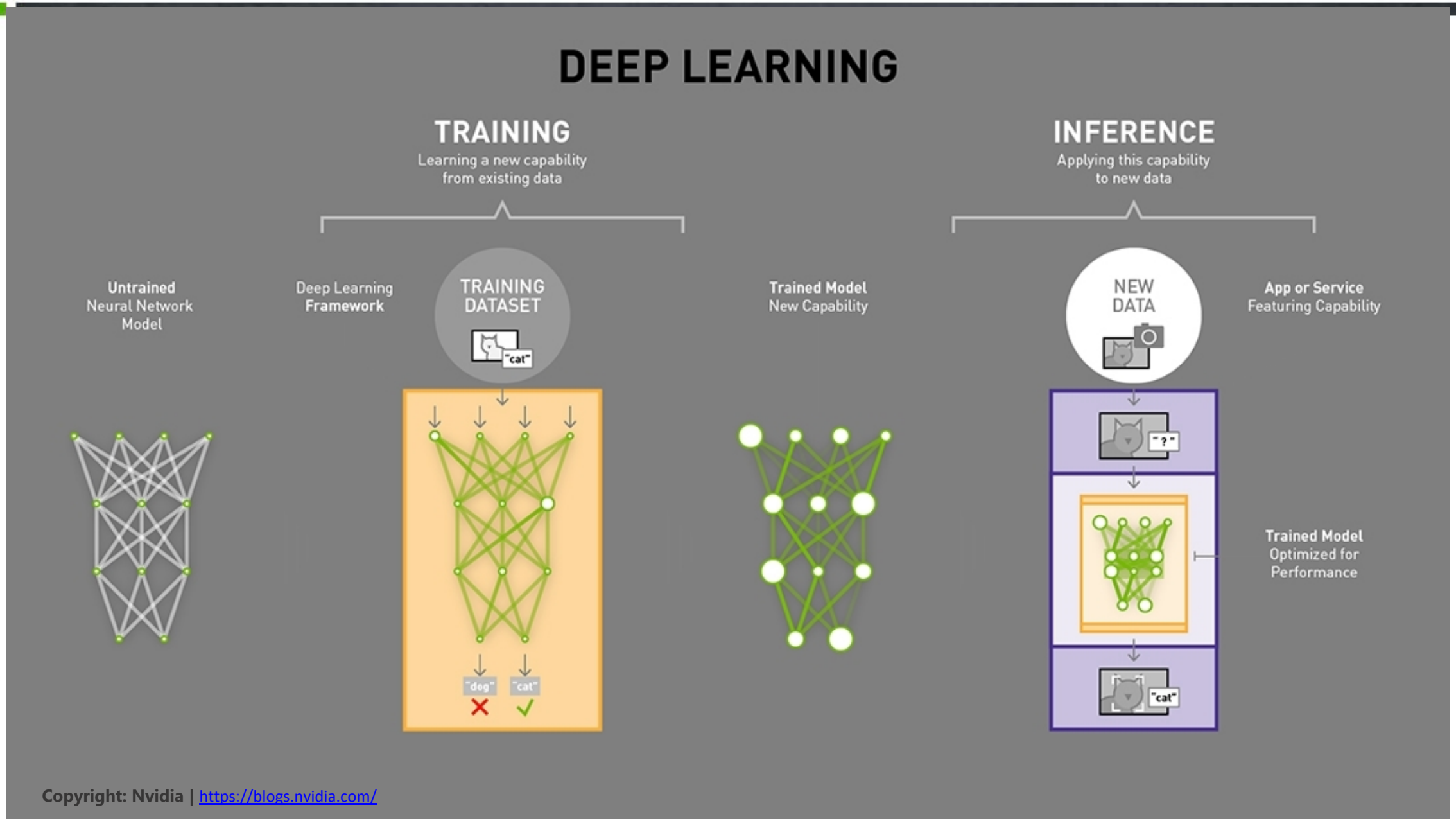
This is called ***instance-based learning***: the system learns the examples "by heart", then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in figure the new instance would be classified as a triangle because the majority of the most similar instances belong to that class.

C.2 Model-Based Learning

Another way to generalize from a set of examples is to build a model of these examples and then use that model to make ***predictions***. This is called ***model-based learning***.

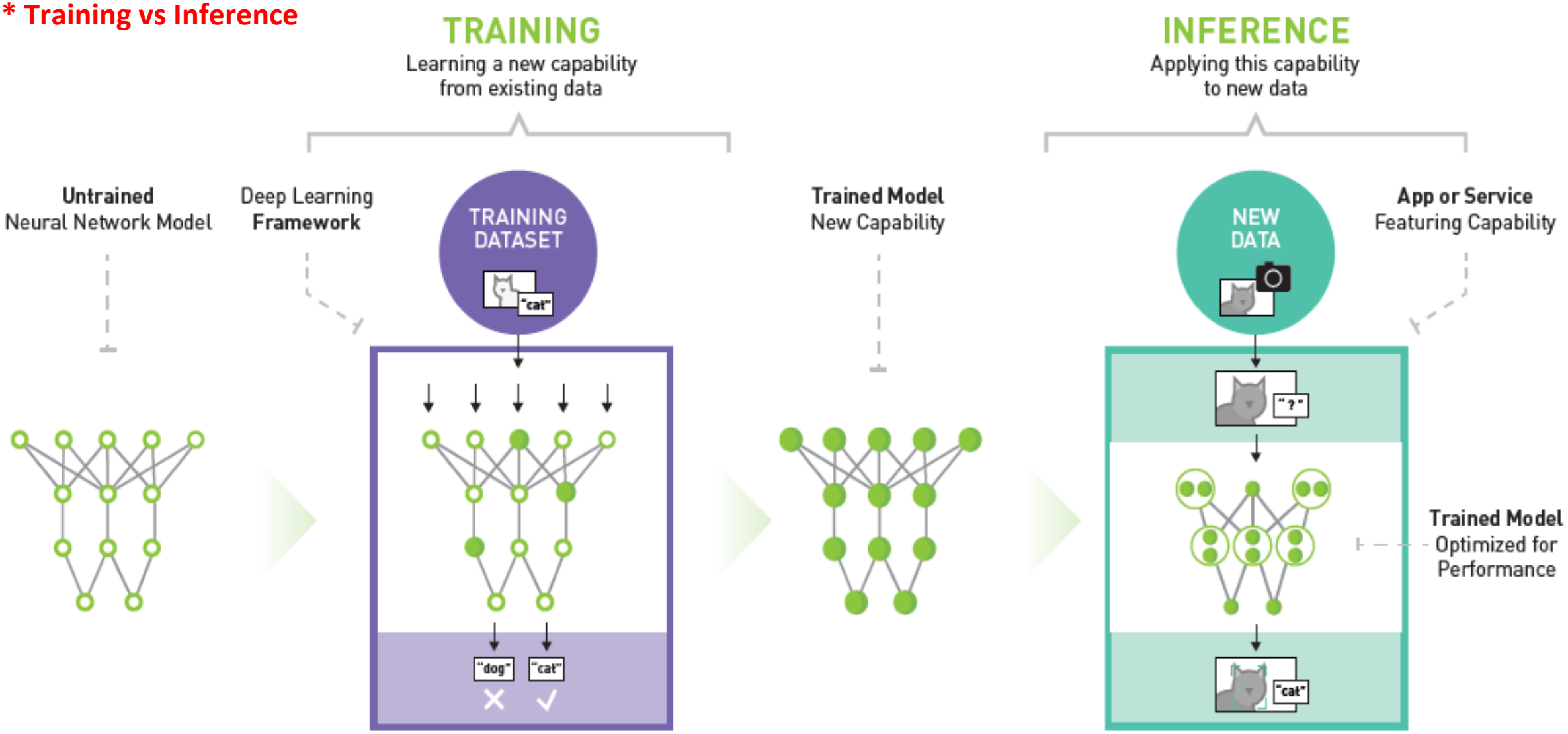


Artificial Intelligence & Machine Learning Terminology



Artificial Intelligence & Machine Learning Terminology

*** Training vs Inference**



DEMO 1 in Python/Jupiter for Model-Based Learning

For example, suppose you want to know if money makes people happy, so you download the Better Life Index data from the OECD's website (Organisation for Economic Co-operation and Development: https://homl.info/4) and stats about gross domestic product (GDP) per capita from the IMF's website (International Monetary Fund: https://homl.info/5). Then you join the tables and sort by GDP per capita. Table shows an excerpt of what you get.

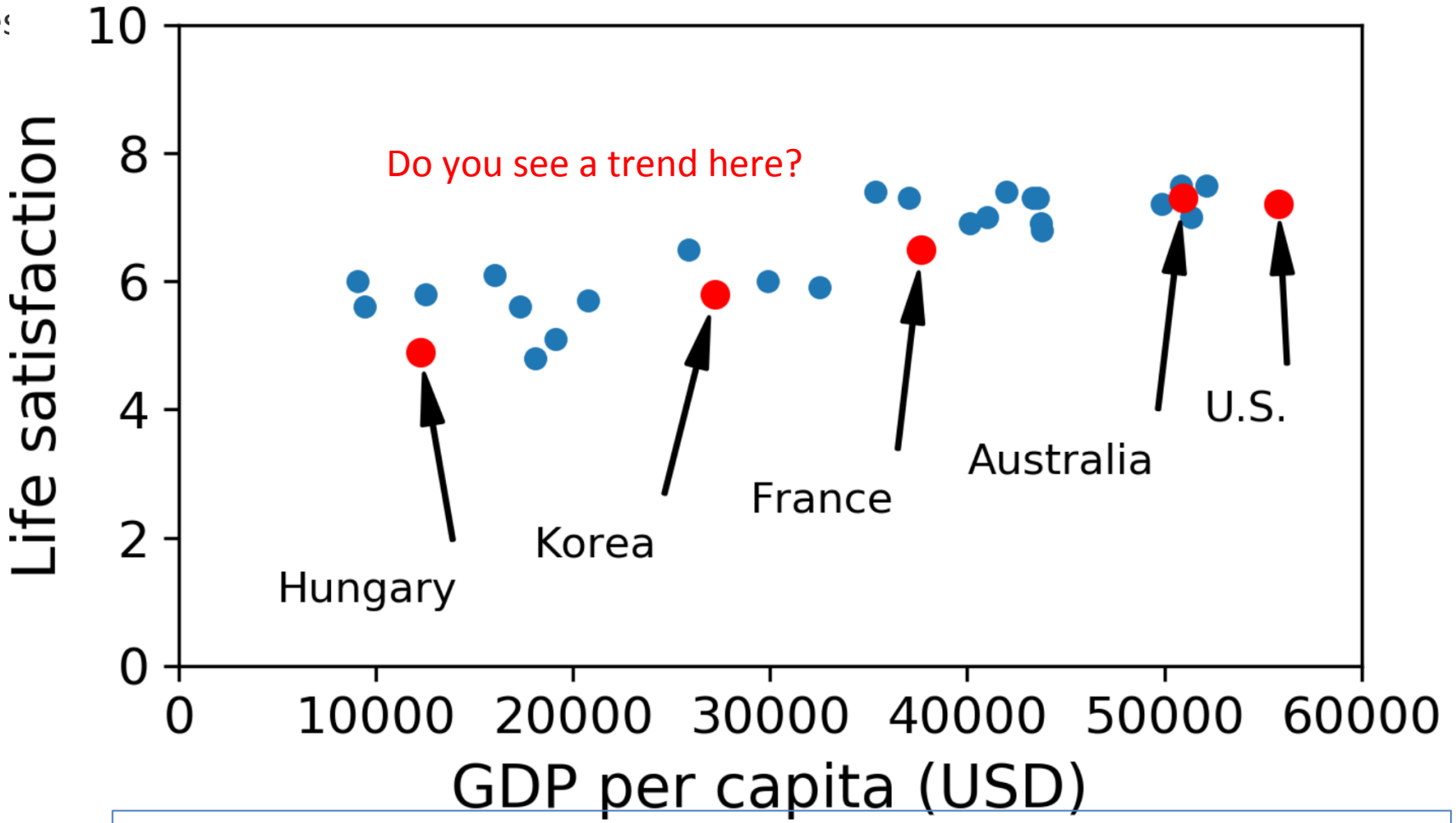
Table (Year 2015):
Does money make people happier?

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2

DEMO 1 in Python/Jupiter for Model-Based Learning

Let's plot the data for these countries

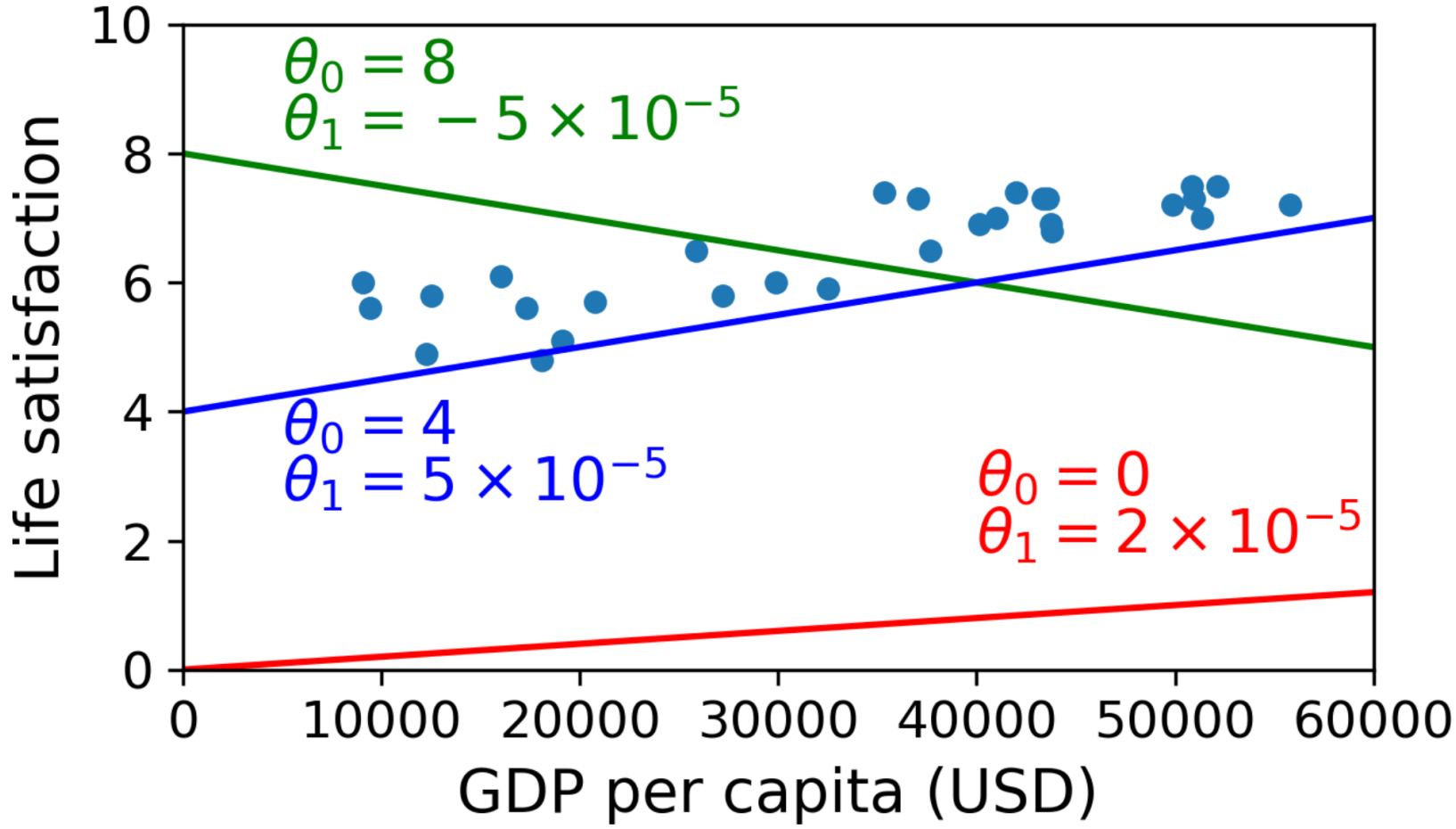
There does seem to be a trend here! Although the data is noisy (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita. This step is called model selection: you selected a linear model of life satisfaction with just one attribute, GDP per capita (Equation 1-1: A simple linear model).



Equation 1-1: $life\_satisfaction = \theta_0 + \theta 1 \times GDP\_per\_capita$

DEMO 1 in Python/Jupiter for Model-Based Learning

This model has two model parameters, θ_0 and θ_1. By tweaking these parameters, you can make your model represent any linear function, as shown in figure: ***a few possible linear models***.



DEMO 1 in Python/Jupiter for Model-Based Learning

Before you can use your model, you need to define the parameter values θ_0 and θ_1. How can you know which values will make your model perform best? To answer this question, you need to specify a performance measure. You can either define a ***utility function*** (or ***fitness function***) that measures how *good* your model is, or you can define a ***cost function*** that measures **how *bad* it is**. For **Linear Regression problems**, people typically use ***a cost function that measures the distance between the linear model's predictions and the training examples; the objective is to minimize this distance.***

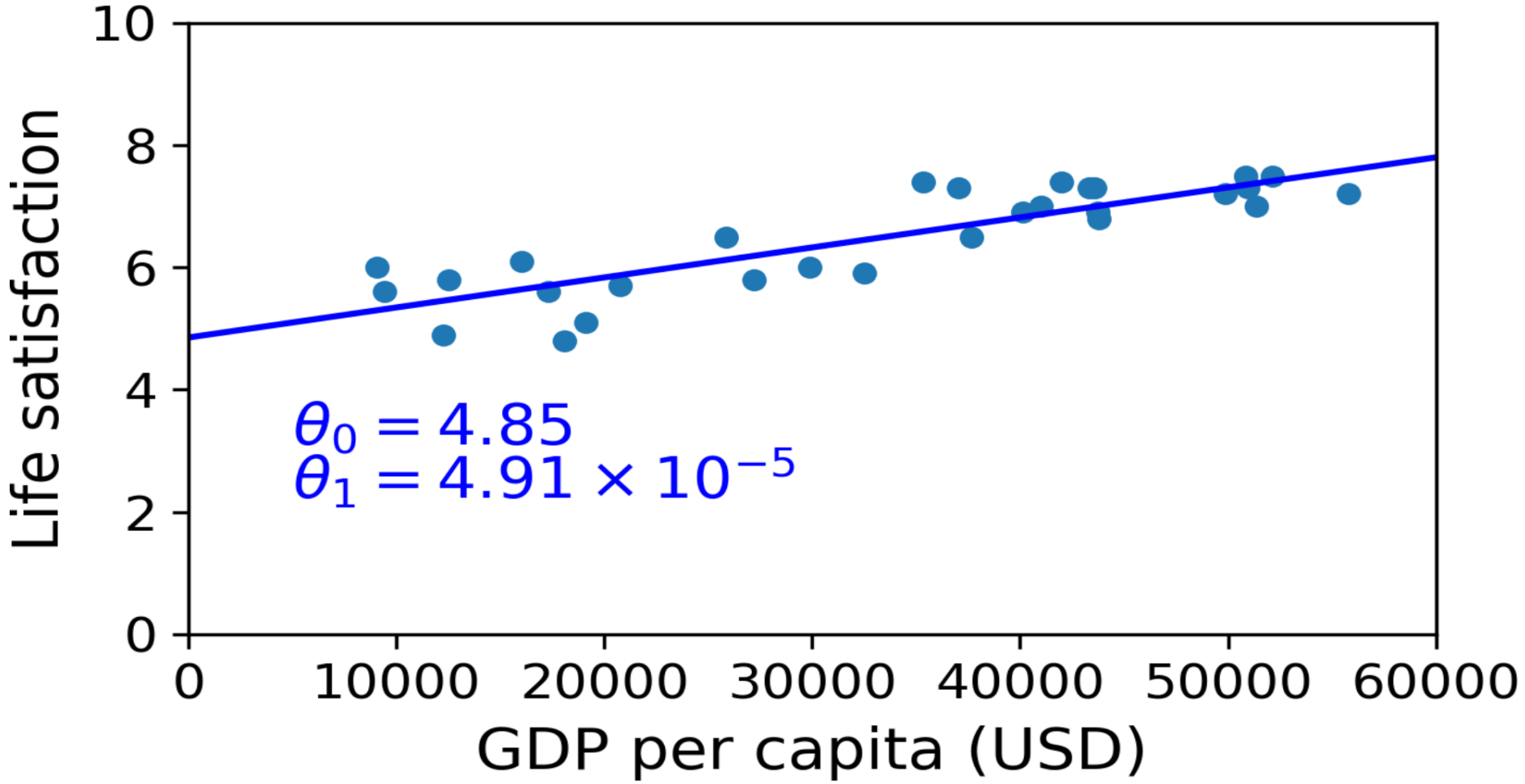
This is where t***he Linear Regression algorithm*** comes in: you feed it your ***training examples***, and it finds the parameters that make the linear model fit best to your data. This is called ***training* the model**.

In our case, the algorithm finds that the optimal parameter values are: $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$

WARNING: Confusingly, the same word "**model**" can refer to a ***type of model*** (e.g., Linear Regression), to a ***fully specified model architecture*** (e.g., ***Linear Regression with one input and one output***), or to the ***final trained model*** ready to be used for predictions (e.g., **Linear Regression with one input and one output, using $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$**). ***Model selection consists in choosing the type of model and fully specifying its architecture***. **Training a model** means **running an algorithm to find the model parameters** that will make it b***est fit the training data (and hopefully make good predictions on new data).***

DEMO 1 in Python/Jupiter for Model-Based Learning

Now the model fits the training data as closely as possible (for a linear model), as you can see ***[The linear model that fits the training data best]***



Artificial Intelligence & Machine Learning Overview

DEMO 1 in Python/Jupiter for Model-Based Learning

https://colab.research.google.com/github/ageron/handson-ml2/blob/master/01_the_machine_learning_landscape.ipynb#scrollTo=xYoqxdHtY9sA

You are finally ready to run the model to make predictions. For example, say you want to know how happy Cypriots are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up Cyprus's GDP per capita, find $22,587, and then apply your model and find that life satisfaction is likely to be somewhere around:

$4.85 + 22{,}587 \times 4.91 \times 10^{-5} = 5.96$

Training and running a linear model using Python 3.5+ and Scikit-Learn:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model
# Load the data
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")


# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Select a linear model
model = sklearn.linear_model.LinearRegression()

# Train the model
model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[22587]]  # Cyprus's GDP per capita
print(model.predict(X_new)) # outputs [[ 5.96242338]]
```

DEMO 1 in Python/Jupiter for Model-Based Learning

NOTE

If ***you had used an instance-based learning algorithm instead***, you would have found that Slovenia has the closest GDP per capita to that of Cyprus ($20,732), and since the OECD data tells us that Slovenians' life satisfaction is 5.7, you would have predicted a life satisfaction of 5.7 for Cyprus.

If you zoom out a bit and look at the two next-closest countries, you will find Portugal and Spain with life satisfactions of 5.1 and 6.5, respectively. Averaging these three values, you get 5.77, which is pretty close to your model-based prediction.
This simple algorithm is called ***k-Nearest Neighbors*** regression (in this example, ***k*** = 3).

Replacing the **Linear Regression model** with **k-Nearest Neighbors regression** in the previous code is as simple as **replacing these two lines:**

```
import sklearn.linear_model
model = sklearn.linear_model.LinearRegression()
```

With these two lines:

```
import sklearn.neighbors
model = sklearn.neighbors.KNeighborsRegressor(
    n_neighbors=3)
```

DEMO 1 in Python/Jupiter for Model-Based Learning

If all went well, your model will make good predictions. If not, you may need to use more attributes (employment rate, health, air pollution, etc.), get more or better-quality training data, or perhaps select a more powerful model (e.g., a Polynomial Regression model)

In summary:

- ***Studied the data***.
- ***Selected a model***.
- ***Trained it on the training data*** (i.e., the learning algorithm searched for the model parameter values that minimize a cost function).
- Finally, you ***applied the model to make predictions on new cases*** (this is called ***inference***), hoping that this model will generalize well.

This is what a typical Machine Learning project looks like. In the next sections (Chapter 2 from the book) you will experience this firsthand by going through ***a project end to end***. We have covered a lot of ground so far: you now know what Machine Learning is really about, why it is useful, what some of the most common categories of ML systems are, and what a typical project workflow looks like.

DEMO 1 in Python/Jupiter for Model-Based Learning

Main Challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the ***two things that can go wrong*** are D. "bad data" and/or E. "bad algorithm.":

- **D.1 Insufficient Quantity of Training Data:** For a toddler to learn what an apple is, all it takes is for you to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.
- **D.2 Non-representative Training Data:** In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.
- **D.3 Poor-Quality Data:** Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well.
- **D.4 Irrelevant Features:** As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones.

DEMO 1 in Python/Jupiter for Model-Based Learning

Main Challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the ***two things that can go wrong*** are D. "bad data" and/or E. "bad algorithm.":

- **E.1 Over-fitting the Training Data:** Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that *all* taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In Machine Learning this is called *over-fitting*: it means that the model performs well on the training data, but it does not generalize well.

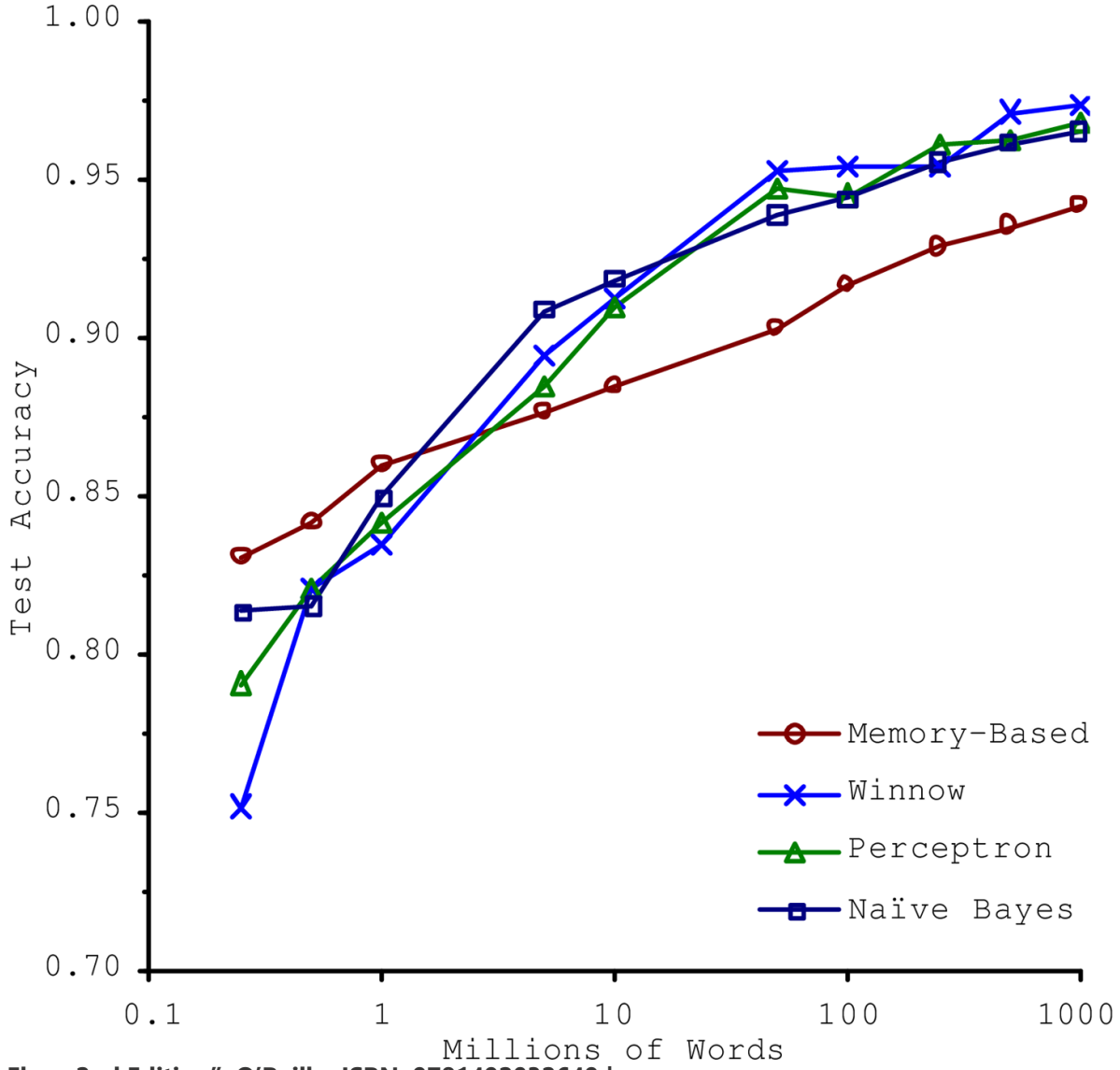
- **E.2 Under-fitting the Training Data:** As you might guess, *under-fitting* is the opposite of over-fitting: it occurs when your model is too simple to learn the underlying structure of the data.

Artificial Intelligence & Machine Learning Overview

D.1 Insufficient Quantity of Training Data: DEMO 1 in Python/Jupiter for Model-Based Learning

For a toddler to learn what an apple is, all it takes is for you to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius. ***Machine Learning is not quite there yet***; it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

In a famous paper published in 2001, Microsoft researchers Michele Banko and Eric Brill showed that very different Machine Learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation, once they were given enough data.

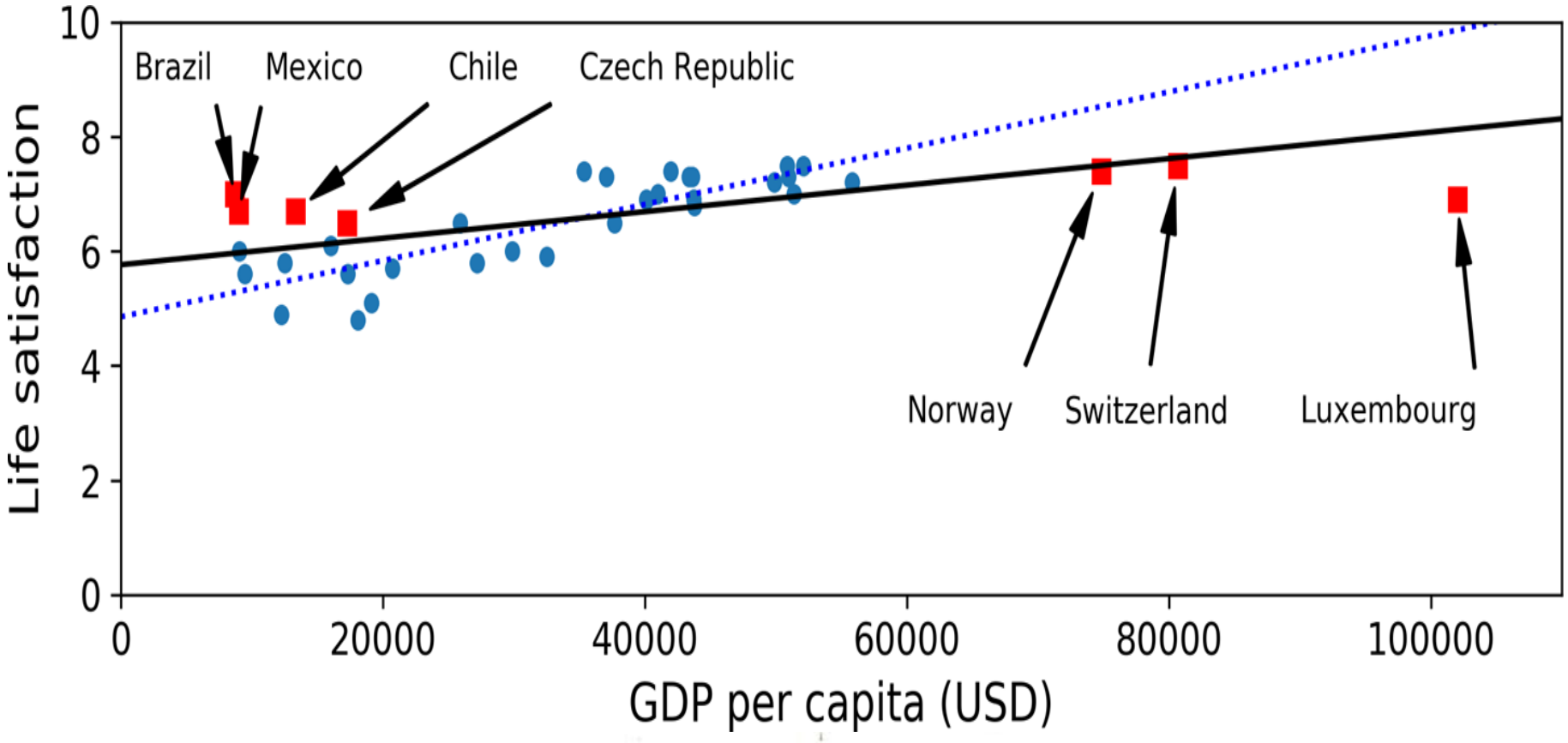


DEMO 1 in Python/Jupiter for Model-Based Learning

D.2 Non-representative Training Data:

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.

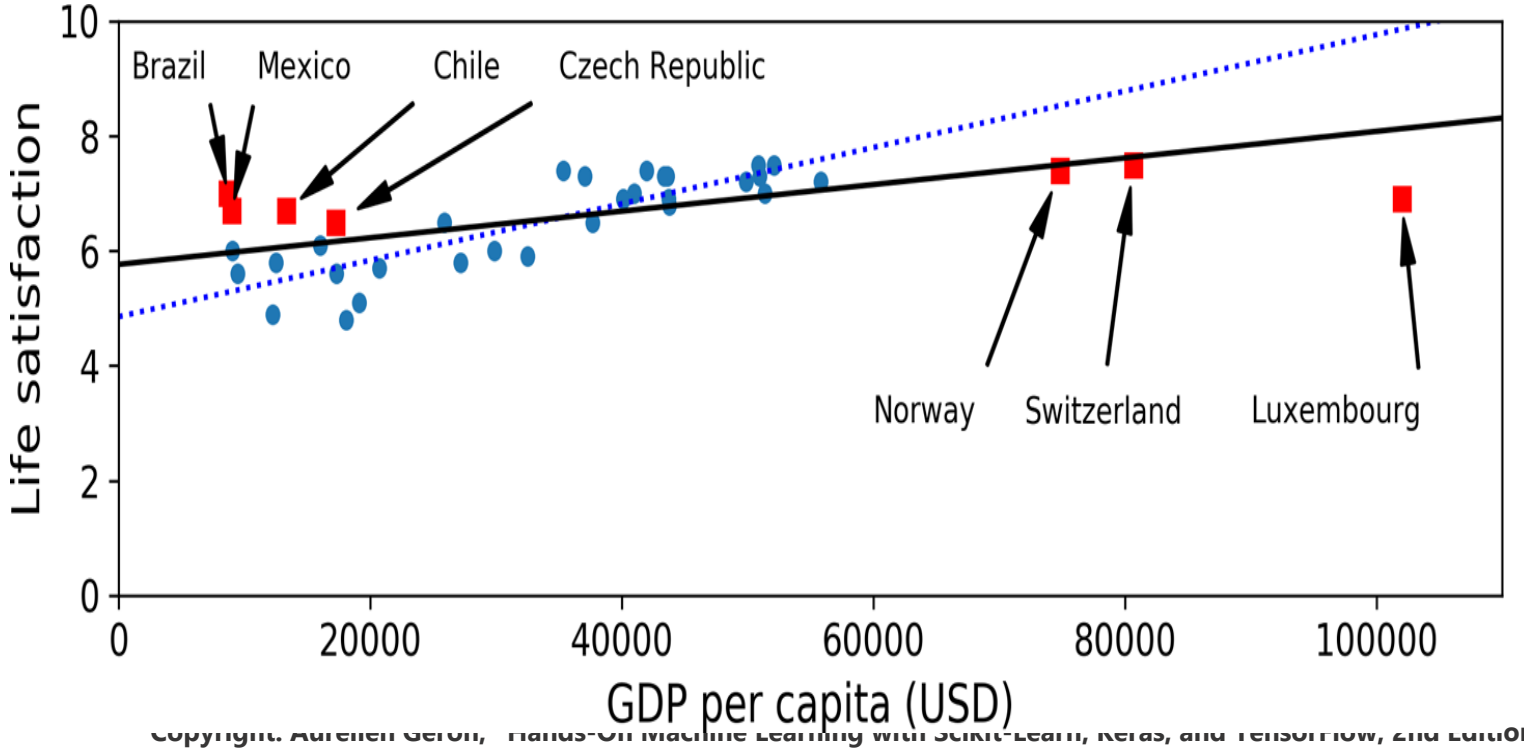
For example, the set of countries we used earlier for training the linear model was not perfectly representative; a few countries were missing. Figure shows what the data looks like when you add the missing countries.



DEMO 1 in Python/Jupiter for Model-Based Learning

D.2 Non-representative Training Data:

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem unhappier), and conversely some poor countries seem happier than many rich countries.
By using a ***non-representative training set***, we ***trained a model that is unlikely to make accurate predictions***, especially for very poor and very rich countries.



It is crucial to ***use a training set*** that ***is representative*** of the cases you want to generalize to. This is often harder than it sounds: ***if the sample is too small***, you will have **sampling noise** (i.e., ***non-representative data*** as a result of chance), but even very ***large samples can be non-representative if the sampling method is flawed***. This is called **sampling bias**.

DEMO 1 in Python/Jupiter for Model-Based Learning

D.3. Poor-Quality Data

Obviously, ***if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements),*** it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple of examples of when you'd want to clean up training data:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it.

D.4. Irrelevant Features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process, called ***feature engineering***, involves the following steps:

- ***Feature selection*** (selecting the most useful features to train on among existing features)
- ***Feature extraction*** (combining existing features to produce a more useful one—as we saw earlier, dimensionality reduction algorithms can help)
- ***Creating new features by gathering new data***

DEMO 1 in Python/Jupiter for Model-Based Learning

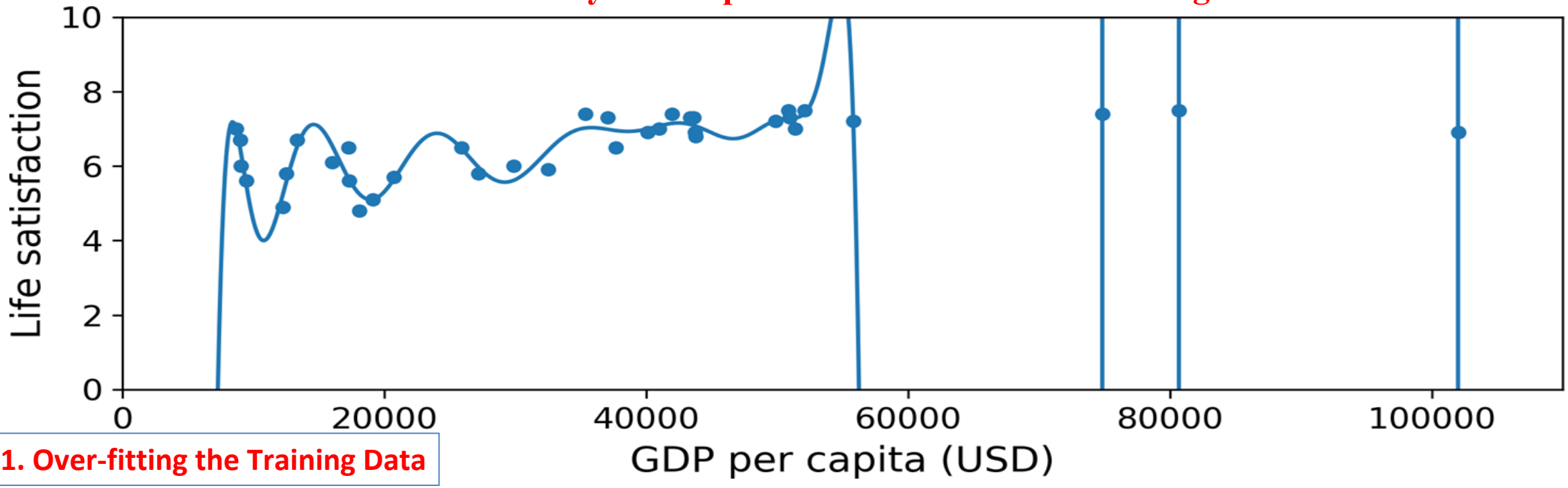
E.1. Over-fitting the Training Data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that all taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful.

In Machine Learning this is called over-fitting: it means that the model performs well on the training data, but it does not generalize well.

The figure shows an example of a high-degree polynomial life satisfaction model that strongly over-fits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

DEMO 1 in Python/Jupiter for Model-Based Learning



E.1. Over-fitting the Training Data

Complex models such as deep neural networks can detect subtle patterns in the data, but if the training set is noisy, or if it is too small (which introduces sampling noise), then the model is likely to detect patterns in the noise itself. Obviously these patterns will not generalize to new instances. For example, say you feed your life satisfaction model many more attributes, including uninformative ones such as the country's name. In that case, a complex model may detect patterns like the fact that all countries in the training data with a *w* in their name have a life satisfaction greater than 7: New Zealand (7.3), Norway (7.4), Sweden (7.2), and Switzerland (7.5). How confident are you that the *w*-satisfaction rule generalizes to Rwanda or Zimbabwe? Obviously this pattern occurred in the training data by pure chance, but the model has no way to tell whether a pattern is real or simply the result of noise in the data.

DEMO 1 in Python/Jupiter for Model-Based Learning

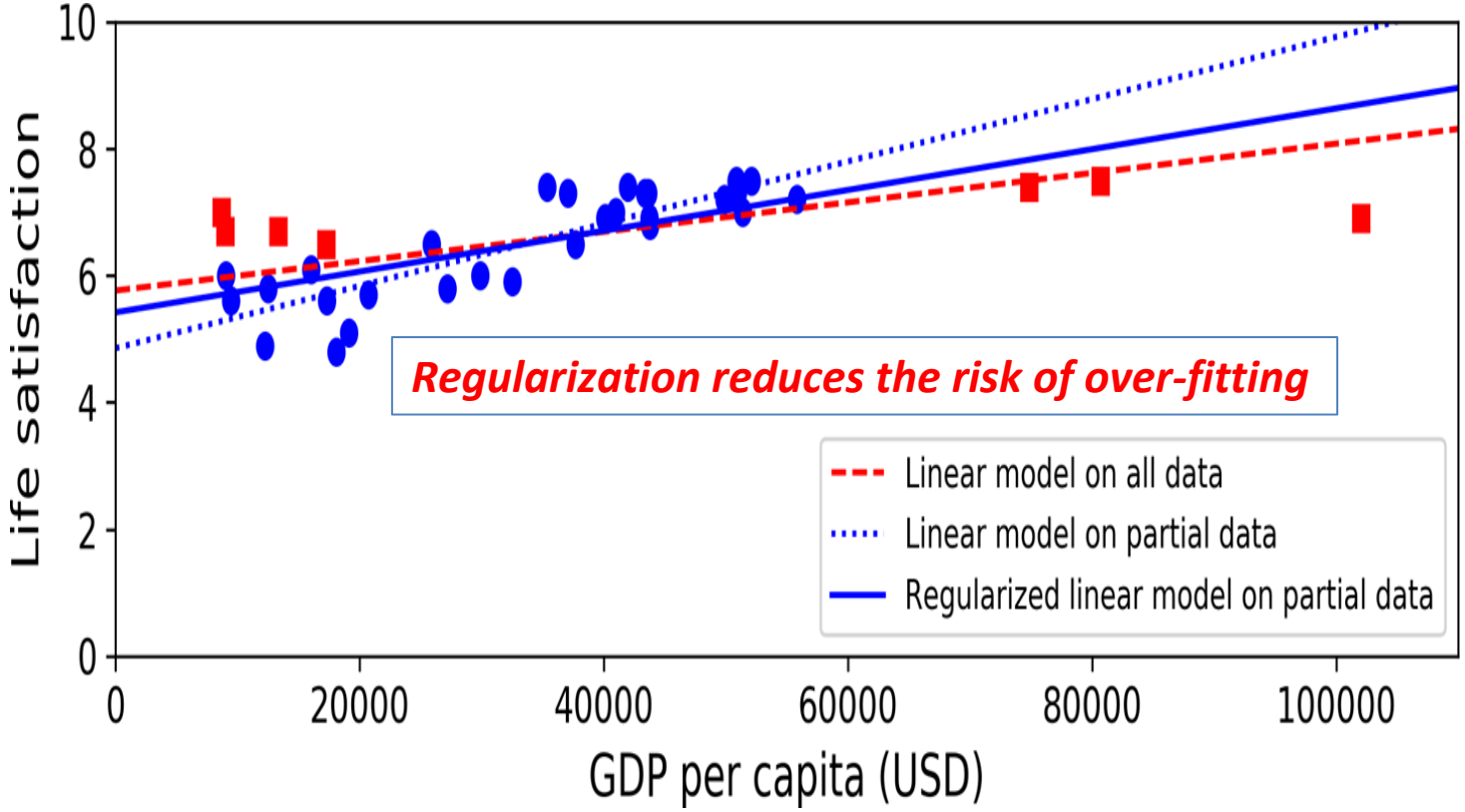
E.1. Over-fitting the Training Data

Constraining a model to make it simpler and reduce the risk of over-fitting is called ***regularization***. For example, the linear model we defined earlier has two parameters, θ_0 and θ_1. This gives the learning algorithm two ***degrees of freedom*** to adapt the model to the training data: ***it can tweak both the height (θ_0) and the slope (θ_1) of the line. If we forced $\theta_1 = 0$***, the algorithm would have only ***one degree of freedom*** and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify θ_1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that's simpler than one with two degrees of freedom, but more complex than one with just one. You want to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well.

Artificial Intelligence & Machine Learning Overview

DEMO 1 in Python/Jupiter for Model-Based Learning

The following figure shows three models. The dotted line represents the original model that was trained on the countries represented as circles (without the countries represented as squares), the dashed line is our second model trained with all countries (circles and squares), and the solid line is a model trained with the same data as the first model but with a regularization constraint. You can see that regularization forced the model to have a smaller slope: this model does not fit the training data (circles) as well as the first model, but it actually generalizes better to new examples that it did not see during training (squares).



The amount of regularization to apply during learning can be controlled by a ***hyper-parameter***. A hyper-parameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. If you set the regularization **hyper-parameter** to a very large value, you will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not **over-fit** the training data, but it will be less likely to find a good solution. Tuning **hyper-parameters** is an important part of building a Machine Learning system.

DEMO 1 in Python/Jupiter for Model-Based Learning

E.2. Under-fitting the Training Data

As you might guess, ***under-fitting*** is the opposite of ***over-fitting***: it occurs when your model is too simple to learn the underlying structure of the data.

For example, a linear model of life satisfaction is ***prone to under-fit***; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples.

Here are the main options for fixing this problem:

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (e.g., reduce the regularization hyper-parameter).

Stepping Back **DEMO 1 in Python/Jupiter for Model-Based Learning**

By now, there are a lot info about the Machine Learning. However, we went through so many concepts that you may be feeling a little lost, so let's step back and look at the big picture:

- ***Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules***.
- ***There are many different types of ML systems***: ***supervised or not, batch or online, instance-based or model-based.***
- In an ***ML project you gather data in a training set***, and you ***feed the training set to a learning algorithm***. If the algorithm is model-based, it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance-based, it just learns the examples by heart and generalizes to new instances by using a similarity measure to compare them to the learned instances.
- The system will not perform well if your training set is too small, or ***if the data is not representative***, is ***noisy***, or is ***polluted with irrelevant features*** (garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will ***under-fit***) nor too complex (in which case it will ***over-fit***).

There's just one last important topic to cover: once you have trained a model, you don't want to just "hope" it generalizes to new cases.

You want to evaluate it and fine-tune it if necessary. Let's see how to do that.

Testing and Validating

DEMO 1 in Python/Jupiter for Model-Based Learning

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well, but if your model is horribly bad, your users will complain—not the best idea.

A better option is to split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the *generalization error* (or *out-of-sample error*), and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform on instances it has never seen before.

***If the training error is low* (i.e., your model makes few mistakes on the training set) *but the generalization error is high*, it means that *your model is over-fitting the training data*.**

* It is common to use 80% of the data for training and hold out 20% for testing. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances, probably more than enough to get a good estimate of the generalization error.
* For simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

DEMO 2 in Python/Jupiter - End2End Project

End-to-End Machine Learning Project

In this chapter you will work through an example project end to end, pretending to be a recently hired ML Solutions Developer / Data Scientist at a real estate company.

Here are the main steps you will go through:

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune the model.
7. Present the solution => Launch, monitor, and maintain your system.

DEMO 2 in Python/Jupiter - End2End Project

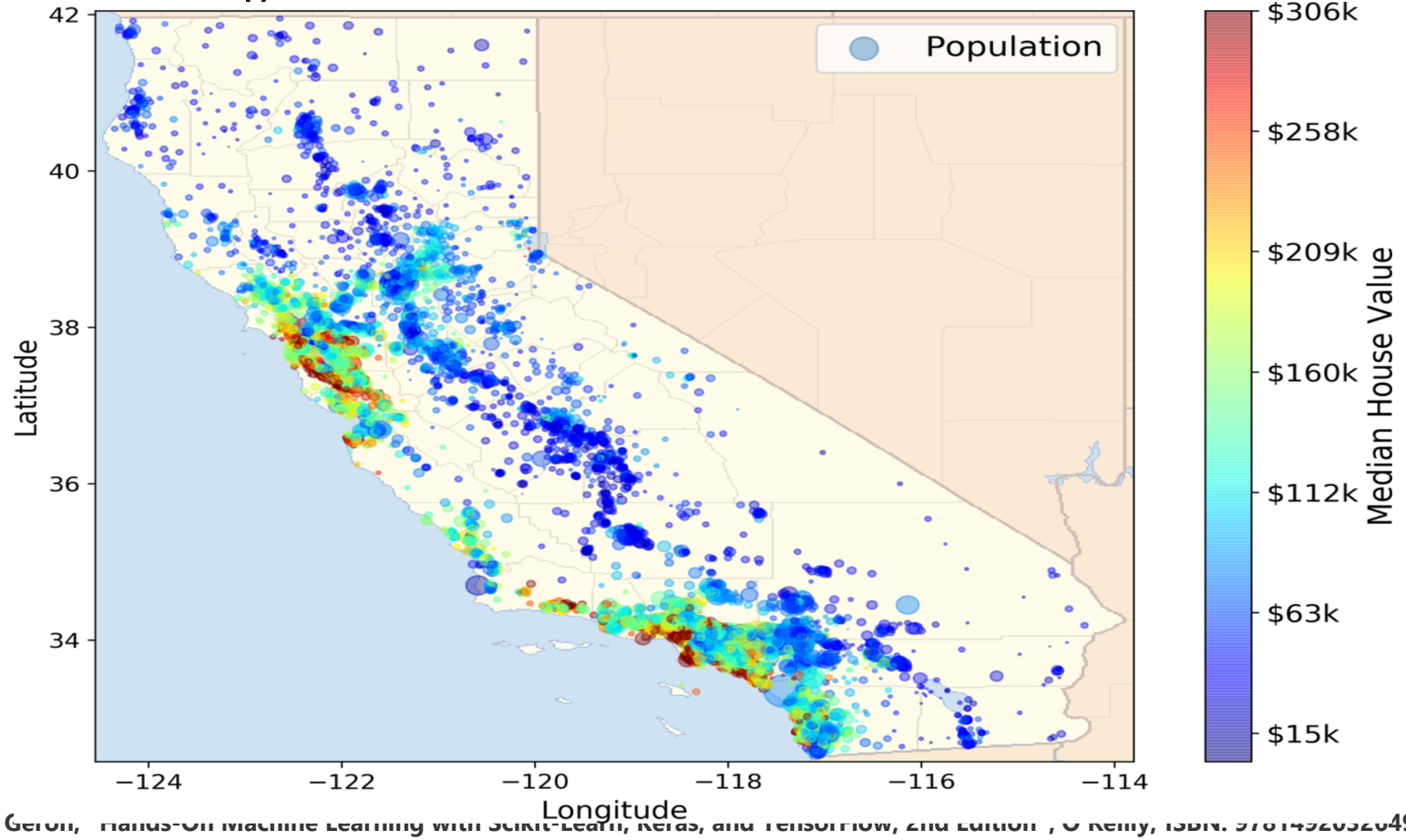
Working with Real Data

When you are learning about Machine Learning, it is best to experiment with real-world data, not artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. Here are a few places you can look to get data:

- Popular open data repositories
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets
- Meta portals (they list open data repositories)
  - Data Portals
  - OpenDataMonitor
  - Quandl
- Other pages listing many popular open data repositories
  - Wikipedia's list of Machine Learning datasets
  - Quora.com
  - The datasets subreddit

DEMO 2 in Python/Jupiter - End2End Project

The ***California Housing Prices*** dataset from the **StatLib** repository (see Figure). This dataset is based on data from the **1990 California census**. It is not exactly recent (a nice house in the Bay Area was still affordable at the time), but it has many qualities for learning ML.



DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture

Welcome to the Machine Learning Housing Corporation! Our first task is to use California census data to build a model of housing prices in the state.
This data includes metrics such as:

- the population,
- median income, and
- median housing price for each block group in California.

Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). We will call them "**districts**" for short.

The model:

1. ***should learn from this data*** and
2. ***be able to predict the median housing price in any district***, given all the other metrics.

DEMO 2 in Python/Jupiter - End2End Project

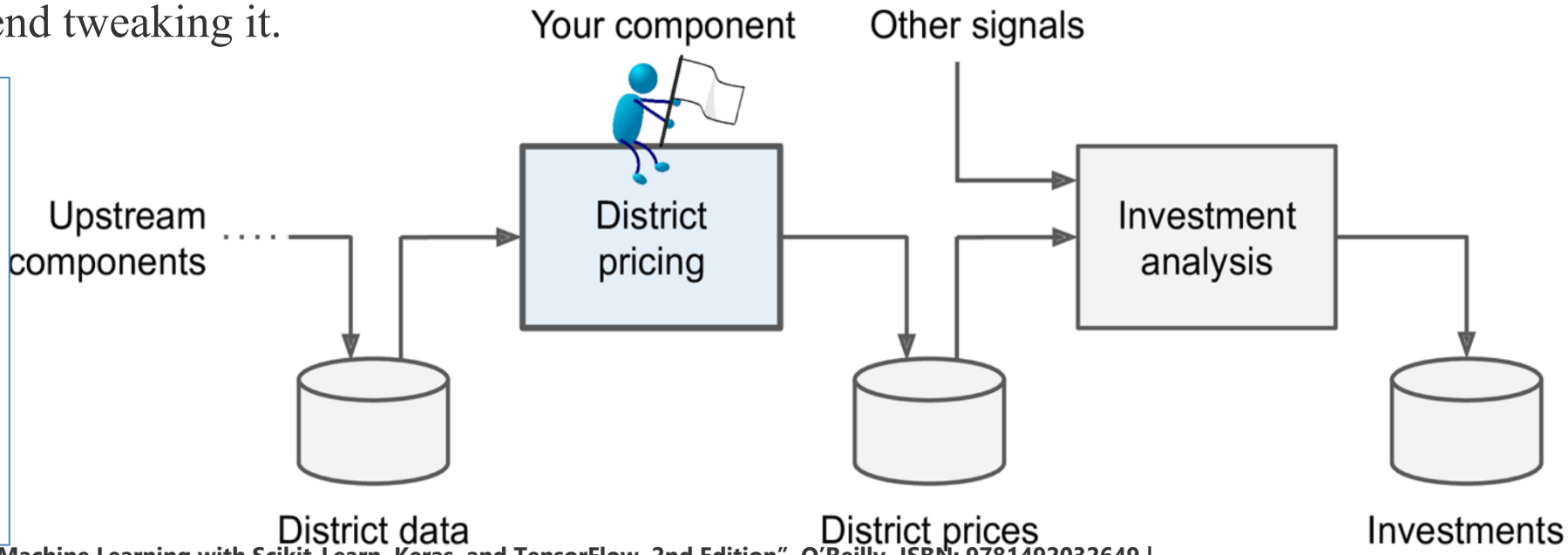
E2E.1. Look at the Big Picture - Frame the Problem

The first question to ask is ***what exactly the business objective is***. Building a model is probably not the end goal. ***How does the company expect to use and benefit from this model? Knowing the objective is important because*** it will determine:

- how you frame the problem,
- which algorithms you will select,
- which performance measure you will use to evaluate your model, and
- how much effort you will spend tweaking it.

A Machine Learning pipeline for real estate investments

The possible answers that your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system (see figure), along with many other signals. This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.



DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture - PIPELINES

A sequence of data processing components is called a data ***pipeline***. Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Components typically **run asynchronously**. Each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, some time later, the next component in the pipeline pulls this data and spits out its own output. Each component is fairly self-contained: the interface between components is simply **the data store (the file, the transaction in blockchain, the record in the relational database table, the document in NoSQL database, etc)**. This makes the system simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This **makes the architecture quite robust**.

On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale (old/deprecated) and the overall system's performance drops.

DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture

The next question to ask your management is what the current solution looks like (if any).
The current situation will often give you a reference for performance, as well as insights on how to solve the problem.
Your management's answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a district, and when they cannot get the median housing price, they estimate it using complex rules. This is costly and time-consuming, and their estimates are not great; in cases where they manage to find out the actual median housing price, they often realize that their estimates were off by more than 20%. This is why the company thinks that it would be useful to train a model to predict a district's median housing price, given other data about that district. The census data looks like a great dataset to exploit for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

With all this information, you are now ready to start designing your system.
First, you need ***to frame the problem***:
is it supervised, unsupervised, or Reinforcement Learning?
Is it a classification task, a regression task, or something else?
Should you use batch learning or online learning techniques?
Before you go on, pause and try to answer these questions for yourself.

Have you found the answers?

DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture

Answers?:

- It is clearly a typical ***supervised learning task***, since you are given labeled training examples (each instance comes with the expected output, i.e., the district's median housing price).
- It is also a typical regression task, since you are asked to predict a value. More specifically, this is a ***multiple regression problem***, since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.).
- It is also a ***uni-variate regression problem***, since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a multivariate regression problem.
- Finally, ***there is no continuous flow of data coming into the system***, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory, so ***plain batch learning*** should do just fine.

TIP: ***If the data were huge***, you could either split your batch learning work across multiple servers (using the Map-Reduce technique: ***Apache Hadoop/Spark –Big Data & Collective Intelligence***) or use an ***online learning*** technique.

DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture

Select a Performance Measure

Your next step is to select a performance measure. A typical performance measure for regression problems is the **Root Mean Square Error (RMSE).** It gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors. ***Equation*** shows the mathematical formula to compute the **RMSE**.

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

Root Mean Square Error (RMSE)

Machine Learning & End2End Project

DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture - Select a Performance Measure

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

Root Mean Square Error (RMSE)

NOTATIONS

This equation introduces several very common Machine Learning notations that we will use throughout this book:

- m is the number of instances in the dataset you are measuring the RMSE on.
  - For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then $m = 2{,}000$.
- $\mathbf{x}^{(i)}$ is a vector of all the feature values (excluding the label) of the i^{th} instance in the dataset, and $y^{(i)}$ is its label (the desired output value for that instance).
  - For example, if the first district in the dataset is located at longitude –118.29°, latitude 33.91°, and it has 1,416 inhabitants with a median income of $38,372, and the median house value is $156,400 (ignoring the other features for now), then:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1{,}416 \\ 38{,}372 \end{pmatrix}$$

and:

$$y^{(1)} = 156{,}400$$

DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture - Select a Performance Measure

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

Root Mean Square Error (RMSE)

- **X** is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the i^{th} row is equal to the transpose of $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^T$.[4]
  - For example, if the first district is as just described, then the matrix **X** looks like this:

$$\mathbf{X} = \begin{pmatrix} \left(\mathbf{x}^{(1)}\right)^T \\ \left(\mathbf{x}^{(2)}\right)^T \\ \vdots \\ \left(\mathbf{x}^{(1999)}\right)^T \\ \left(\mathbf{x}^{(2000)}\right)^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture - Select a Performance Measure

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

Root Mean Square Error (RMSE)

- *h* is your system's prediction function, also called a *hypothesis*. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ for that instance ($\hat{y}$ is pronounced "y-hat").
  - For example, if your system predicts that the median housing price in the first district is $158,400, then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158{,}400$. The prediction error for this district is $\hat{y}^{(1)} - y^{(1)} = 2{,}000$.
- RMSE(**X**,*h*) is the cost function measured on the set of examples using your hypothesis *h*.

We use lowercase italic font for scalar values (such as *m* or $y^{(i)}$) and function names (such as *h*), lowercase bold font for vectors (such as $\mathbf{x}^{(i)}$), and uppercase bold font for matrices (such as **X**).

Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the *mean absolute error* (MAE, also called the average absolute deviation; see Equation 2-2):

Equation 2-2. Mean absolute error (MAE)

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

https://github.com/ageron/handson-ml2

DEMO 2 in Python/Jupiter - End2End Project

E2E.1. Look at the Big Picture – Check the Assumption

Lastly, it is good practice to list and verify the assumptions that have been made so far (by you or others); this can help you catch serious issues early on.

For example, the district prices that your system outputs are going to be fed into a downstream Machine Learning system, and you assume that these prices are going to be used as such.
But what if the downstream system converts the prices into categories (e.g., “cheap,” “medium,” or “expensive”) and then uses those categories instead of the prices themselves? In this case, getting the price perfectly right is not important at all; your system just needs to get the category right. If that’s so, then the problem should have been framed as a classification task, not a regression task. You don’t want to find this out after working on a regression system for months.

Fortunately, after talking with the team in charge of the downstream system, you are confident that they do indeed need the actual prices, not just categories.

Great! You’re all set, the lights are green, and you can start coding now!

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data

Create the Workspace

It's time to get your hands "dirty" with code with:

- *Python 3.5+*
- *Scikit-Learn*
- *Jupyter,*
- *NumPy,*
- *pandas,*
- *Matplotlib.*

```
sudo apt install python-pip
sudo apt-get install python3-pip
sudo pip install --upgrade pip
sudo pip3 install --upgrade pip

python3 -m pip --version
python3 -m pip install --user -U virtualenv
$ export ML_PATH="$HOME/ml" # You can change the path if you prefer
$ mkdir -p $ML_PATH
$ cd $ML_PATH
$ python3 -m virtualenv my_env

$ source my_env/bin/activate # on Linux or macOS
python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn
python3 -m ipykernel install --user --name=python3

$ jupyter notebook
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Download the Data

In typical environments your data would be available in a ***Relational or NoSQL database*** (or some other common data store) and ***spread across multiple tables/documents/files***. To access it, you would first need to get your credentials and access authorizations and familiarize yourself with the data schema.

In this project, however, things are much simpler: you will just download a single compressed file, housing.tgz, whic contains a comma-separated values (CSV) file called housing.csv with all the data.

```
import os
import tarfile
import urllib

import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Take a Quick Look at the Data Structure

Let's take a look at the top five rows using the DataFrame's head() method.

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

Each row represents one district. There are 10 attributes (you can see the first 6 in the screenshot):
longitude, **latitude**, **housing_median_age**, **total_rooms**, **total_bedrooms**, **population**, **households**, **median_income**, **median_house_value**, and **ocean_proximity**.

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Take a Quick Look at the Data Structure

The info() method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of nonnull values

```
In [6]: housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude           20640 non-null float64
latitude            20640 non-null float64
housing_median_age  20640 non-null float64
total_rooms         20640 non-null float64
total_bedrooms      20433 non-null float64
population          20640 non-null float64
households          20640 non-null float64
median_income       20640 non-null float64
median_house_value  20640 non-null float64
ocean_proximity     20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

There are **20,640** instances in the dataset, which means that it is fairly small by Machine Learning standards, but it's perfect to get started.

Notice that the **total_bedrooms** attribute has only **20,433** **nonnull** values, ***meaning that 207 districts are missing this feature***.

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data – Take a Quick Look at the Data Structure

All attributes are numerical, except the **ocean_proximity** field. Its type is object, so it could hold any kind of Python object. But since you loaded this data from a CSV file, you know that it must be a text attribute. When you looked at the top five rows, you probably noticed that the values in the **ocean_proximity** column were repetitive, which means that it is probably a categorical attribute. You can find out what categories exist and how many districts belong to each category by using the **value_counts**() method:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN      9136
INLAND         6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND            5
Name: ocean_proximity, dtype: int64
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data – Take a Quick Look at the Data Structure

Let's look at the other fields. The **describe**() method shows a summary of the numerical attributes.

```
In [8]: housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000(
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.38507(
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000(
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000(
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000(
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000(

The **count, mean, min, and max** rows are self-explanatory. ***Note that the null values are ignored*** (so, for example, the count of total_bedrooms is 20,433, not 20,640). The std row shows the ***standard deviation***, which measures how the values are dispersed.

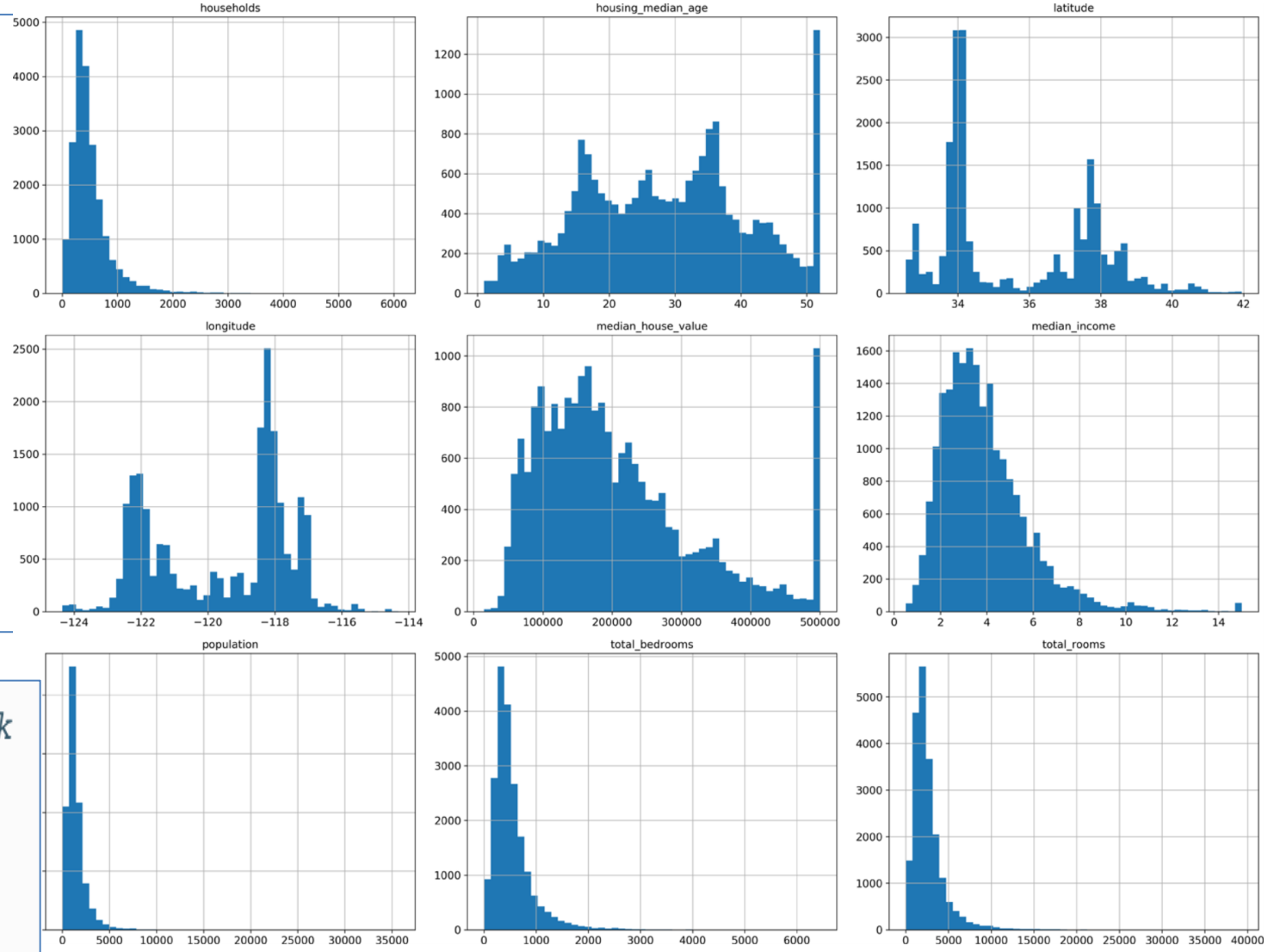
The 25%, 50%, and 75% rows show the corresponding *percentiles*: a percentile indicates the value below which a given percentage of observations in a group of observations fall. For example, 25% of the districts have a housing_median_age lower than 18, while 50% are lower than 29 and 75% are lower than 37. These are often called the 25th percentile (or first ***quartile***), the median, and the 75th percentile (or third quartile).

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Take a Quick Look at the Data Structure

Another quick way to get a feel of the type of data you are dealing with is to ***plot a histogram*** for each numerical attribute. A histogram shows the ***number of instances (on the vertical axis)*** that have a ***given value range (on the horizontal axis).*** You can either plot this one attribute at a time, or you can call the **hist**() method on the whole dataset (as shown in the following code example), and it will plot a histogram for each numerical attribute.

```
%matplotlib inline   # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Take a Quick Look at the Data Structure

There are a few things you might notice in these histograms:

1.First, the median income attribute does not look like it is expressed in US dollars (USD). After checking with the team that collected the data, you are told that the data has been scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes. The numbers represent roughly tens of thousands of dollars (e.g., 3 actually means about $30,000). Working with preprocessed attributes is common in Machine Learning, and it is not necessarily a problem, but you should try to understand how the data was computed.

2.The housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels). Your Machine Learning algorithms may learn that prices never go beyond that limit. You need to check with your client team (the team that will use your system's output) to see if this is a problem or not. If they tell you that they need precise predictions even beyond $500,000, then you have two options:

a. Collect proper labels for the districts whose labels were capped.

b. Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond $500,000).

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data – Take a Quick Look at the Data Structure

There are a few things you might notice in these histograms:

3. These attributes have very different scales. We will discuss this later, when we explore feature scaling.

4. Finally, many histograms are *tail-heavy*: they extend much farther to the right of the median than to the left. This may make it a bit harder for some Machine Learning algorithms to detect patterns. We will try transforming these attributes later on to have more bell-shaped distributions.

Warning:

Wait! Before you look at the data any further, you need to create a test set, put it aside, and never look at it.

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Create a Test Set

Creating a test set is theoretically simple: pick some instances randomly, typically 20% of the dataset (or less if your dataset is very large), and set them aside.

Well, this works, but it is not perfect: if you run the program again, it will generate a different test set! Over time, you (or your Machine Learning algorithms) will get to see the whole dataset, which is what you want to avoid.

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

You can then use this function like this:[13]

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Create a Test Set

If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted. If this is not possible, then you can try to use the most stable features to build a unique identifier. For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID like so:

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is ***train_test_split***(), which does pretty much the same thing as the function ***split_train_test***(), with a couple of additional features. First, there is a **random_state** parameter that allows you to set the random generator seed. Second, you can pass it multiple datasets with an identical number of rows, and it will split them on the same indices (this is very useful, for example, if you have a separate DataFrame for labels):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data – Create a Test Set

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias. When a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone book. They try to ensure that these 1,000 people are representative of the whole population. For example, the US population is 51.3% females and 48.7% males, so a well-conducted survey in the US would try to maintain this ratio in the sample: 513 female and 487 male.

This is called stratified sampling: the population is divided into homogeneous subgroups called strata, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population. If the people running the survey used purely random sampling, there would be about a 12% chance of sampling a skewed test set that was either less than 49% female or more than 54% female. Either way, the survey results would be significantly biased.

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data – Create a Test Set

Suppose you chatted with experts who told you that the median income is a very important attribute to predict median housing prices. You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset. Since the median income is a continuous numerical attribute, you first need to create an income category attribute. Let's look at the median income histogram more closely (back in previous figure with histograms):

- most median income values are clustered around 1.5 to 6 (i.e., $15,000–$60,000), but some median incomes go far beyond 6. It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of a stratum's importance may be biased. This means that you should not have too many strata, and each stratum should be large enough.

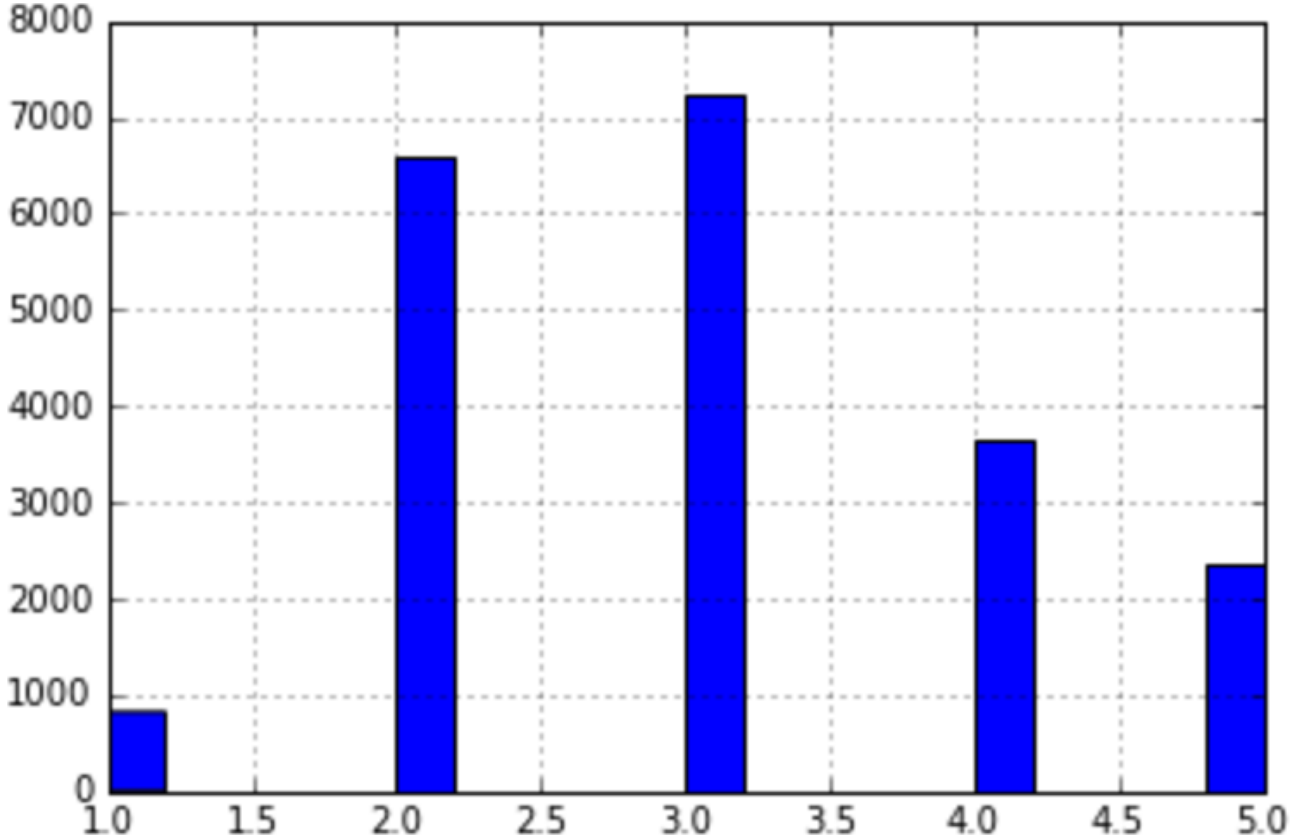
The following code uses the **pd.cut**() function to create an income category attribute with five categories (labeled from 1 to 5): category 1 ranges from 0 to 1.5 (i.e., less than $15,000), category 2 from 1.5 to 3, and so on:

```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.2. Get the Data - Create a Test Set

These income categories are represented: **housing["income_cat"].hist()**



Histogram of income categories

Machine Learning & End2End Project

E2E.2. Get the Data - Create a Test Set

We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a Machine Learning project.

Now you are ready to do stratified sampling based on the income category. For this you can use Scikit-Learn's `StratifiedShuffleSplit` class:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Let's see if this worked as expected. You can start by looking at the income category proportions in the test set:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.3. Discover and visualize the data to gain insights.

So far you have only taken a quick glance at the data to get a general understanding of the kind of data you are manipulating.

Now the goal is to go into a little more depth.

First, make sure you have put the test set aside and ***you/we are only exploring the training set***. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small, so you can just work directly on the full set.

Let's create a copy so that you can play with it without harming the training set:

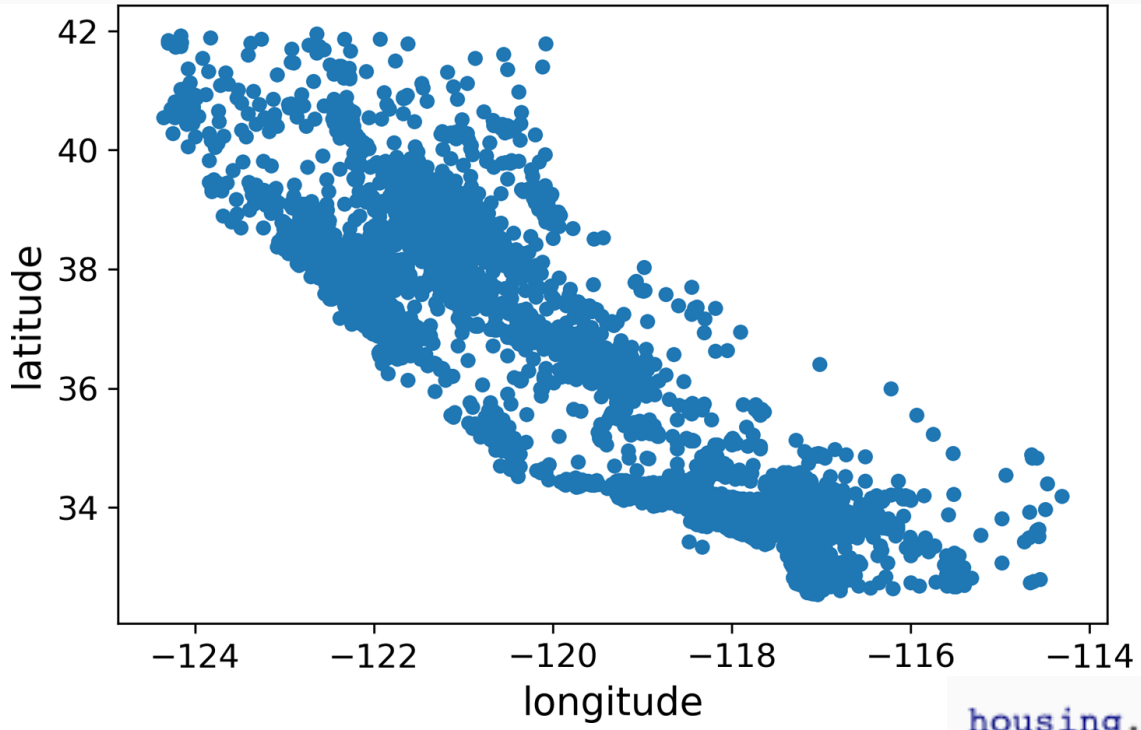
```
housing = strat_train_set.copy()
```

DEMO 2 in Python/Jupiter - End2End Project

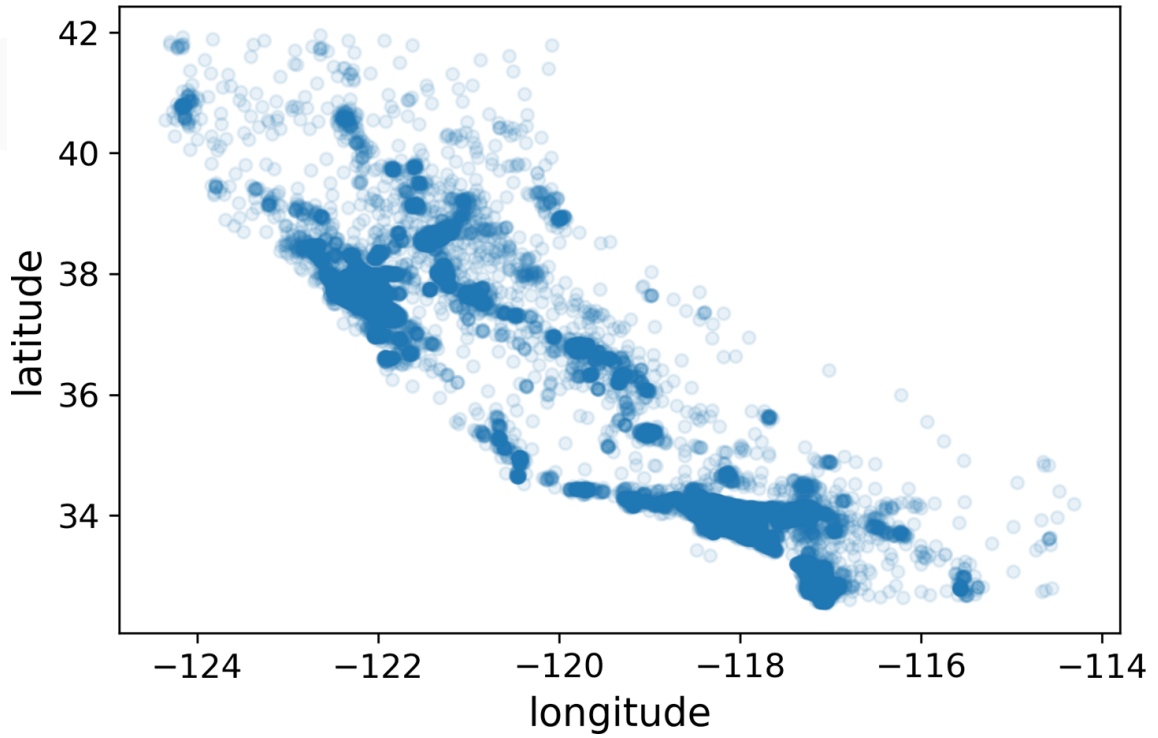
E2E.3. Discover and visualize the data to gain insights - Visualizing Geographical Data

Since there is geographical information (latitude and longitude), it is a good idea to create a scatterplot of all districts to visualize the data:

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```



A geographical scatterplot of the data



A better visualization that highlights high-density areas

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

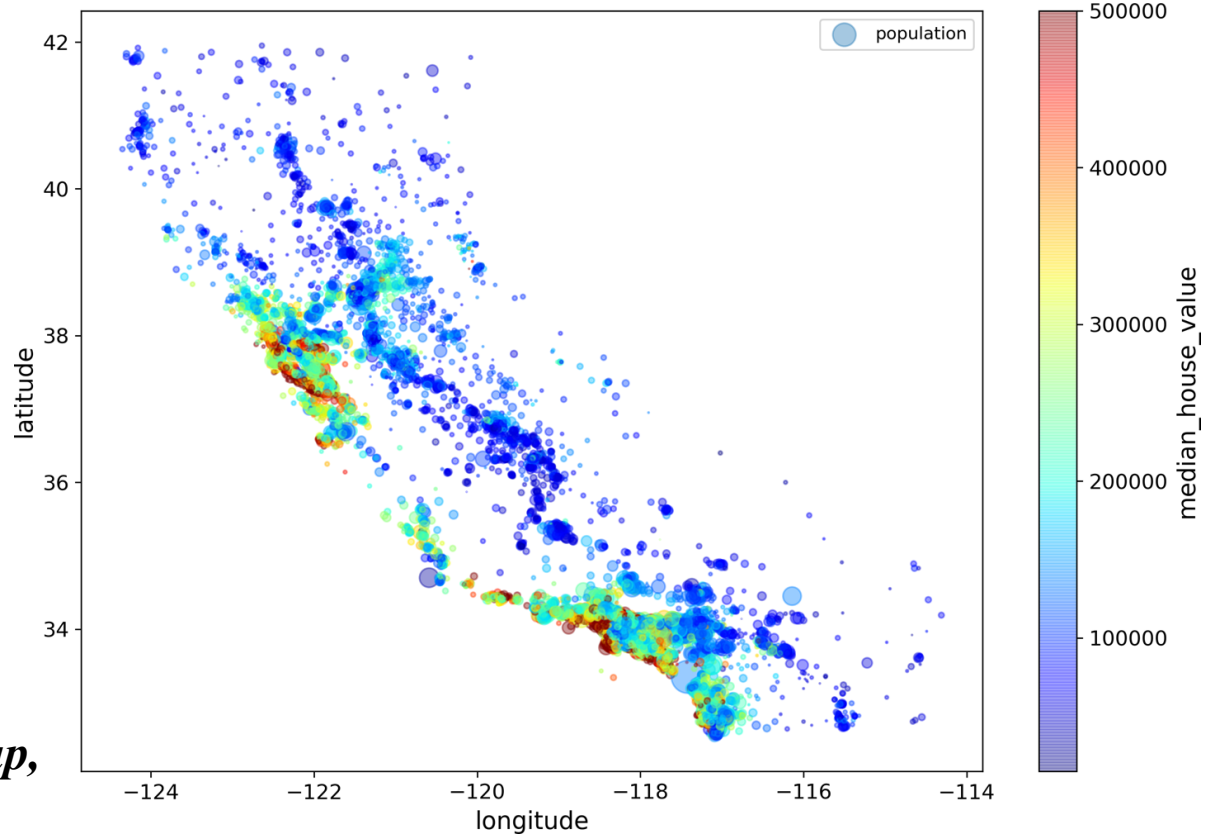
DEMO 2 in Python/Jupiter - End2End Project

E2E.3. Discover and visualize the data to gain insights - Visualizing Geographical Data

Let's look at the housing prices (figure). The radius of each circle represents the district's population (**option s**), and the color represents the price (**option c**). We will use a predefined color map (**option cmap**) called jet, which ranges from ***blue (low values)*** to ***red (high prices)*:**

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers. The ocean proximity attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.



California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

DEMO 2 in Python/Jupiter - End2End Project

E2E.3. Discover and visualize the data to gain insights - Looking for Correlations

Since the dataset is not too large, you can easily compute the *standard correlation coefficient* (also called ***Pearson's r***) between every pair of attributes using the **corr()** method:

```
corr_matrix = housing.corr()
```

Now let's look at how much each attribute correlates with the median house value:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

Given a set of observations $(x_1, y_1), (x_2,y_2),...(x_n,y_n)$, the formula for computing

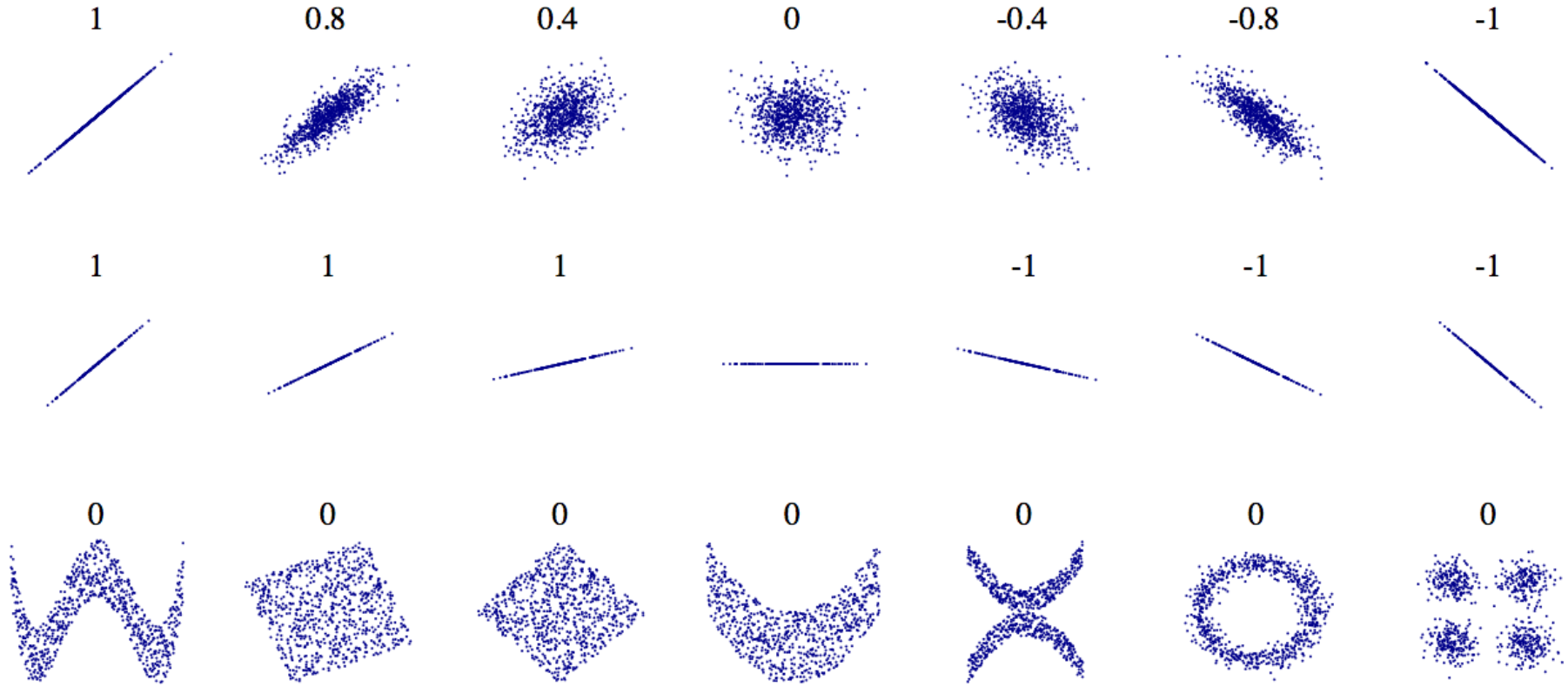
$$r = \frac{1}{n-1} \sum \left(\frac{x - \overline{x}}{s_x} \right) \left(\frac{y - \overline{y}}{s_y} \right)$$

DEMO 2 in Python/Jupiter - End2End Project

E2E.3. Discover and visualize the data to gain insights –

Looking for Correlations

The correlation coefficient ranges from –1 to 1. When it is close to 1, it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up. When the coefficient is close to –1, it means that there is a strong negative correlation; you can see a small negative correlation between the latitude and the median house value (i.e., prices have a slight tendency to go down when you go north). Finally, coefficients close to 0 mean that there is no linear correlation.



Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

DEMO 2 in Python/Jupiter - End2End Project

E2E.3. Discover and visualize the data to gain insights - Looking for Correlations

The correlation coefficient only measures linear correlations ("if *x* goes up, then *y* generally goes up/down").

It may completely miss out on nonlinear relationships (e.g., "if *x* is close to 0, then *y* generally goes up").

Note how all the plots of the bottom row have a correlation coefficient equal to 0, despite the fact that their axes are clearly not independent: these are examples of nonlinear relationships.

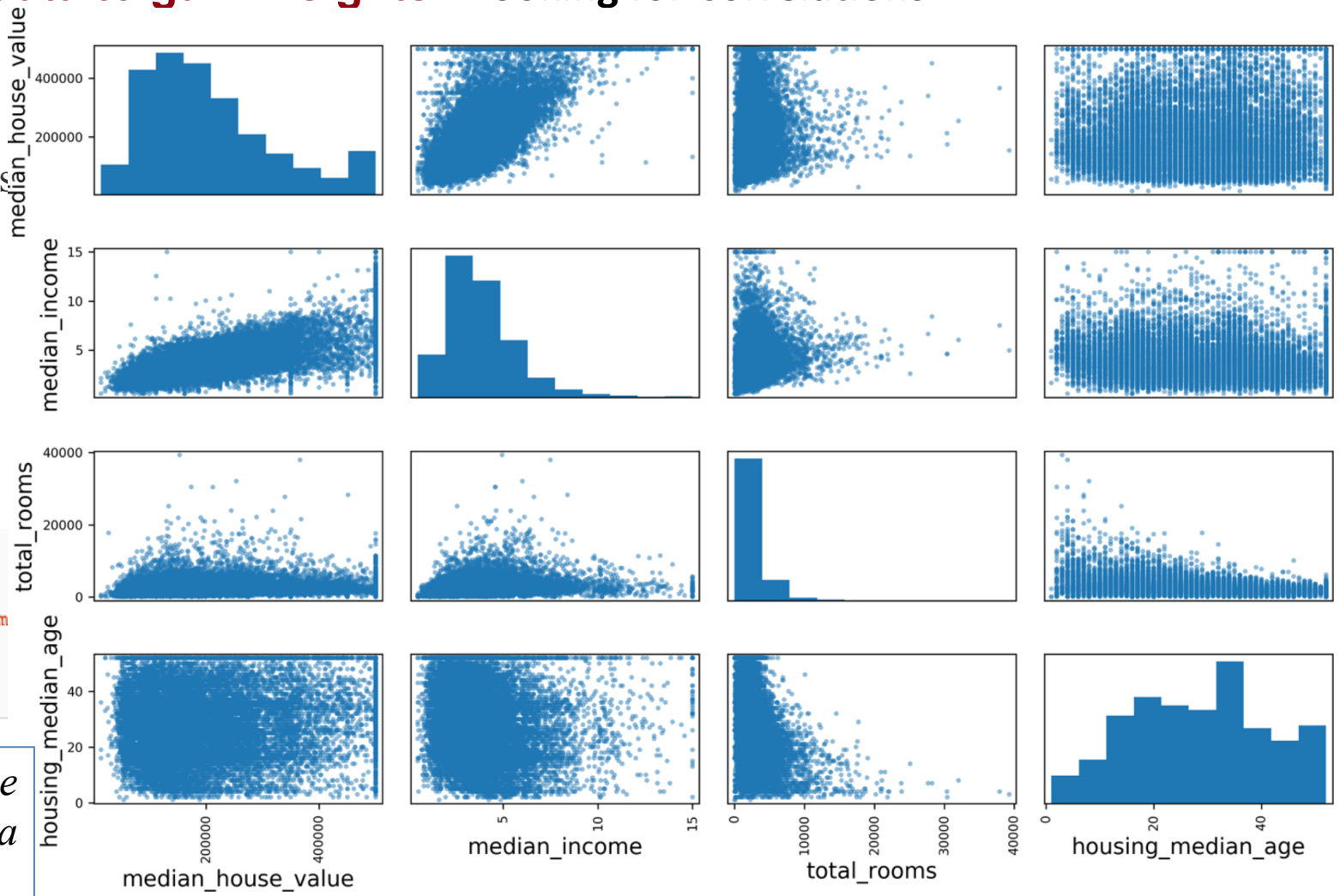
Also, the second row shows examples where the correlation coefficient is equal to 1 or –1; notice that this has nothing to do with the slope. For example, your height in inches has a correlation coefficient of 1 with your height in feet or in nanometers.

DEMO 2 in Python/Jupiter - End2End Project

E2E.3. Discover and visualize the data to gain insights – Looking for Correlations

Another way to check for correlation between attributes is to use the pandas **scatter_matrix**() function, which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you would get $11^2 = 121$ plots, which would be large; so let's just focus on a few promising attributes that seem most correlated with the median housing value:

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_room
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```



This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute

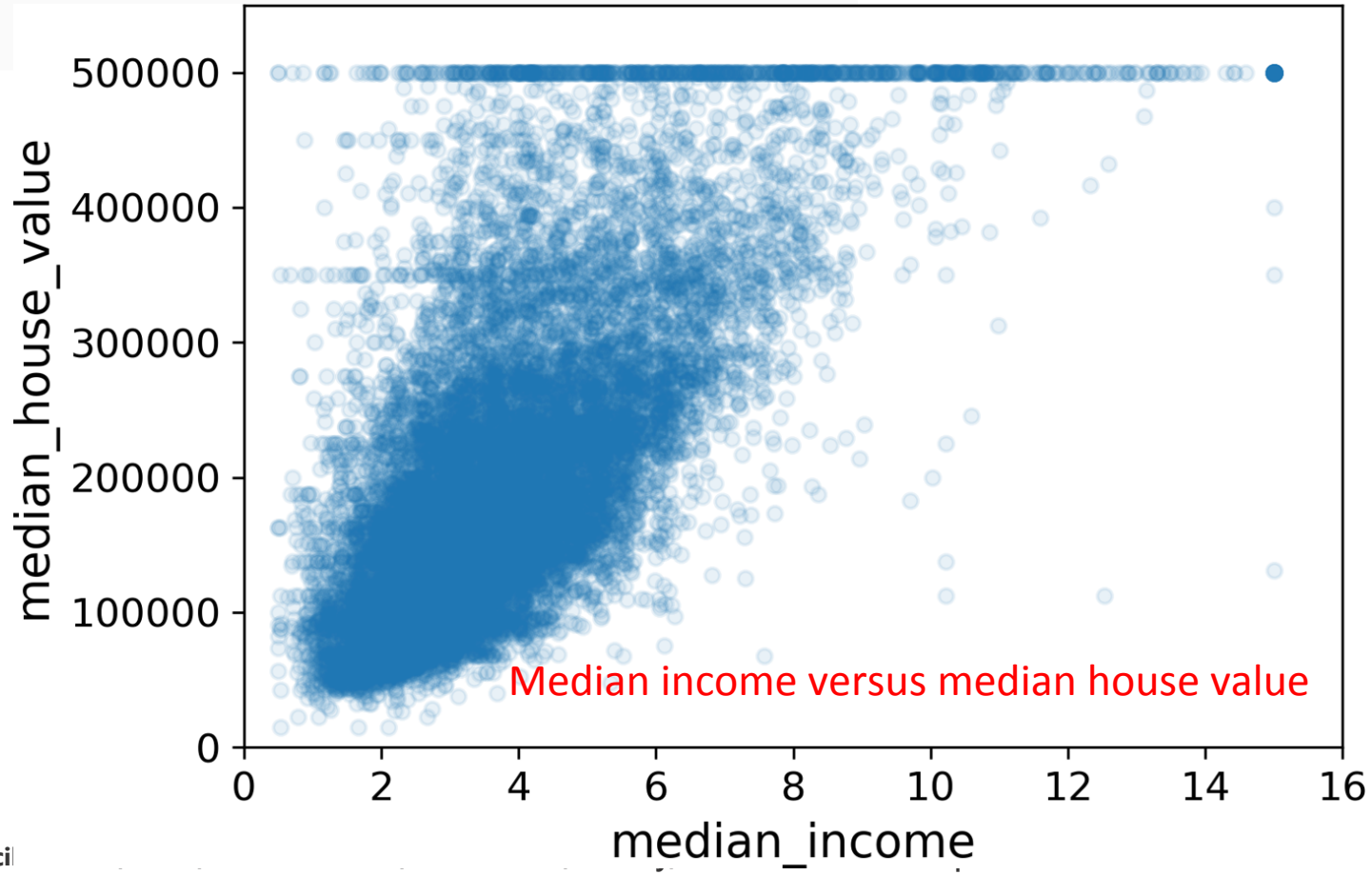
DEMO 2 in Python/Jupiter - End2End Project

E2E.3. Discover and visualize the data to gain insights - Looking for Correlations

The most promising attribute to predict the median house value is the median income, so let's zoom in on their correlation scatterplot

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
```

This plot reveals a few things.
First, the correlation is indeed very strong; you can clearly see the upward trend, and the points are not too dispersed.
Second, the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000. But this plot reveals other less obvious straight lines: a horizontal line around $450,000, another around $350,000, perhaps one around $280,000, and a few more below that.
The one may want to try removing the corresponding districts to prevent the algorithms from learning to reproduce these data quirks.



Machine Learning & End2End Project

E2E.3. Discover and visualize the data to gain insights - Looking for Correlations & Experimenting with Attribute Combinations

One last thing you may want to do before preparing the data for Machine Learning algorithms is to try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don't know how many households there are. What you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at.

Apparently houses with a lower bedroom/room ratio tend to be more expensive.

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

And now let's look at the correlation matrix again:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.4. Prepare the Data for Machine Learning Algorithms

It's time to prepare the data for your Machine Learning algorithms. Instead of doing this manually, you/someone should write functions for this purpose, for several good reasons:

- This will allow to reproduce these transformations easily on any dataset (e.g., the next time you get a fresh dataset).
- It will gradually build a library of transformation functions that you can reuse in future projects.
- It can use these functions in your live system to transform the new data before feeding it to your algorithms.
- This will make it possible for you to easily try various transformations and see which combination of transformations works best.

But first let's revert to a clean training set (by copying **strat_train_set** once again). Let's also separate the ***predictors*** and ***the labels***, since we don't necessarily want to apply the same transformations to the predictors and the ta _set):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.4. Prepare the Data for Machine Learning Algorithms - Data Cleaning

Most ***Machine Learning algorithms cannot work with missing features***, so let's create a few functions to take care of them. We saw earlier that the **total_bedrooms** attribute has some missing values, so let's fix this. You have three options:

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the values to some value (zero, the mean, the median, etc.).

You can accomplish these easily using DataFrame's dropna(), drop(), and fillna() methods:

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

If you choose option 3, you should compute the median value on the training set and use it to fill the missing values in the training set. Don't forget to save the median value that you have computed. You will need it later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data.

Scikit-Learn provides a handy class to take care of missing values: `SimpleImputer`. Here is how to use it. First, you need to create a `SimpleImputer` instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes, you need to create a copy of the data without the text attribute `ocean_proximity`:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Now you can fit the `imputer` instance to the training data using the `fit()` method:

```
imputer.fit(housing_num)
```

The `imputer` has simply computed the median of each attribute and stored the result in its `statistics_` instance variable. Only the `total_bedrooms` attribute had missing values, but we cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the `imputer` to all the numerical attributes:

```
>>> imputer.statistics_
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
>>> housing_num.median().values
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
```

Now you can use this "trained" `imputer` to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

The result is a plain NumPy array containing the transformed features. If you want to put it back into a pandas DataFrame, it's simple:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

Machine Learning & End2End Project

DEMO 2 in Python/Jupiter - End2End Project

E2E.4. Prepare the Data for Machine Learning Algorithms

SCIKIT-LEARN DESIGN

Consistency

All objects share a consistent and simple interface:

Estimators

Any object that can estimate some parameters based on a dataset is called an *estimator* (e.g., an `imputer` is an estimator). The estimation itself is performed by the `fit()` method, and it takes only a dataset as a parameter (or two for supervised learning algorithms; the second dataset contains the labels). Any other parameter needed to guide the estimation process is considered a hyperparameter (such as an `imputer`'s `strategy`), and it must be set as an instance variable (generally via a constructor parameter).

Transformers

Some estimators (such as an `imputer`) can also transform a dataset; these are called *transformers*. Once again, the API is simple: the transformation is performed by the `transform()` method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for an `imputer`. All transformers also have a convenience method called `fit_transform()` that is equivalent to calling `fit()` and then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).

Predictors

Finally, some estimators, given a dataset, are capable of making predictions; they are called *predictors*. For example, the `LinearRegression` model in the previous chapter was a predictor: given a country's GDP per capita, it predicted life satisfaction. A predictor has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a `score()` method that measures the quality of the predictions, given a test set (and the corresponding labels, in the case of supervised learning algorithms).[18]

Machine Learning & End2End Project

Inspection

All the estimator's hyperparameters are accessible directly via public instance variables (e.g., `imputer.strategy`), and all the estimator's learned parameters are accessible via public instance variables with an underscore suffix (e.g., `imputer.statistics_`).

Nonproliferation of classes

Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.

Composition

Existing building blocks are reused as much as possible. For example, it is easy to create a `Pipeline` estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.

Sensible defaults

Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.

DEMO 2 in Python/Jupiter - End2End Project

E2E.4. Prepare the Data for Machine Learning Algorithms - Handling Text and Categorical Attributes

So far we have dealt with numerical attributes, but now let's look at text attributes. In this dataset, there is just

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
       ocean_proximity
17606        <1H OCEAN
18632        <1H OCEAN
14650       NEAR OCEAN
3230            INLAND
3555         <1H OCEAN
19480           INLAND
8879         <1H OCEAN
13685           INLAND
4937         <1H OCEAN
4861         <1H OCEAN
```

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

It's not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute.

Most Machine Learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's OrdinalEncoder class

DEMO 2 in Python/Jupiter - End2End Project

E2E.4. Prepare the Data for Machine Learning Algorithms - Handling Text and Categorical Attributes

You can get the list of categories using the categories_ instance variable. It is a list containing a 1D array of categories for each ca st one categorical attribute):

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as "bad," "average," "good," and "excellent"), but it is obviously not the case for the ocean_proximity column (for example, categories 0 and 4 are clearly more similar than categories 0 and 1).

To fix this issue, a common solution is ***to create one binary attribute per category***: **one attribute equal to 1 when the category is "<1H OCEAN" (and 0 otherwise)**, **another attribute equal to 1 when the category is "INLAND" (and 0 otherwise), and so on**. This is called ***one-hot encoding***, *because only one attribute will be equal to 1 (hot), while the others will be 0 (cold)*. The new attributes are sometimes called *dummy* attributes. Scikit-Learn provides a OneHotEncoder class to ***convert categorical values into one-hot vectors***:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> cat_encoder = OneHotEncoder()
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
  with 16512 stored elements in Compressed Sparse Row format>
```

Notice that the output is a SciPy *sparse matrix*, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one-hot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the nonzero elements. You can use it mostly like a normal 2D array,[21] but if you really want to convert it to a (dense) NumPy array, just call the `toarray()` method:

```
>>> housing_cat_1hot.toarray()
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

Once again, you can get the list of categories using the encoder's `categories_` instance variable:


```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

DEMO 2 in Python/Jupiter - End2End Project

E2E.4. Prepare the Data for Machine Learning Algorithms – Handling Text and Categorical Attributes

TIP

If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding *will result in a large number of input features. This may slow down training and degrade performance.* If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the ocean_proximity feature with the *distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita)*.

Alternatively, you could replace each category with a learnable, low-dimensional vector called an *embedding*. Each category's representation would be learned during training. This is an example of *representation learning*.

DEMO 2 in Python/Jupiter - End2End Project

E2E.4. Prepare the Data for Machine Learning Algorithms - Custom Transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes. You will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not inheritance), ***all you need to do is create a class and implement three methods: fit() (returning self), transform(), and fit_transform().***

You can get the last one for free by simply adding TransformerMixin as a base class. If you add BaseEstimator as a base class (and avoid *args and **kargs in your constructor), you will also get two extra methods (get_params() and set_params()) that will be useful for automatic hyperparameter tuning.

E2E.4. Prepare the Data for Machine Learning Algorithms - Custom Transformers

For example, here is a small transformer class that adds the combined attributes we discussed earlier:

In this example the transformer has one hyperparameter, add_bedrooms_per_room, set to True by default (it is often helpful to provide sensible defaults). This hyperparameter will allow you to easily find out whether adding this attribute helps the Machine Learning algorithms or not. More generally, you can add a hyperparameter to gate any data preparation step that you are not 100% sure about. ***The more you automate these data preparation steps, the more combinations you can automatically try out, making it much more likely that you will find a great combination (and saving you a lot of time).***

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]

        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

E2E.4. Prepare the Data for Machine Learning Algorithms – Feature Scaling

One of the most important transformations you need to apply to your data is *feature scaling*. With few exceptions, ***Machine Learning algorithms don't perform well when the input numerical attributes have very different scales***. *This is the case for the housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Note that scaling the target values is generally not required.*

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

Min-max scaling (many people call this *normalization*) is the simplest: values are shifted and rescaled so that they end up ranging from 0 to 1. ***We do this by subtracting the min value and dividing by the max minus the min***. Scikit-Learn provides a transformer called MinMaxScaler for this. It has a feature_range hyperparameter that lets you change the range *if, for some reason, you don't want 0–1.*

Standardization is different: ***first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance***. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called StandardScaler for standardization.

E2E.4. Prepare the Data for Machine Learning Algorithms - Transformation Pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations. Here is a small pipeline for the numerical attributes:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

The Pipeline constructor takes a list of name/estimator pairs defining a sequence of steps. All but the last estimator must be transformers (i.e., they must have a *fit_transform*() method). The names can be anything you like (as long as they are unique and don't contain double underscores, __); they will come in handy later for hyperparameter tuning.

When you call the pipeline's fit() method, it calls *fit_transform*() sequentially on all transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it calls the fit() method.

The pipeline exposes the same methods as the final estimator. In this example, the last estimator is a **StandardScaler**, which is a transformer, so the pipeline has a transform() method that applies all the transforms to the data in sequence (and of course also a *fit_transform*() method, which is the one we used).

E2E.4. Prepare the Data for Machine Learning Algorithms - Transformation Pipelines

So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformations to each column. In version 0.20, Scikit-Learn introduced the ColumnTransformer for this purpose, and the good news is that it works great with pandas DataFrames. Let's use it to apply all the transformations to the housing data:

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

First we import the ColumnTransformer class, next we get ***the list of numerical column names*** and ***the list of categorical column names***, and then we construct a ColumnTransformer. *The constructor requires a list of tuples, where each tuple contains a name, a transformer, and a list of names (or indices) of columns that the transformer should be applied to*. In this example, we specify that the numerical columns should be transformed using the num_pipeline that we defined earlier, and the categorical columns should be transformed using a OneHotEncoder. Finally, we apply this ColumnTransformer to the housing data: ***it applies each transformer to the appropriate columns and concatenates the outputs along the second axis (the transformers must return the same number of rows)***.

Note that the OneHotEncoder returns a ***sparse matrix***, while the num_pipeline returns a ***dense matrix***. When there is such a mix of sparse and dense matrices, the ColumnTransformer estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, sparse_threshold=0.3). In this example, it returns a dense matrix. And that's it! We ***have a preprocessing pipeline that takes the full housing data and applies the appropriate transformations to each column.***

E2E.5. Select and Train a Model

At last! We have:

- framed the problem,
- got the data and explored it,
- sampled a training set and a test set, and
- written transformation pipelines to clean up and prepare the data for *Machine Learning algorithms automatically*.

You/We are now **ready** to ***select and train a Machine Learning model***.

E2E.5. Select and Train a Model - Training and Evaluating on the Training Set

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

Done! You now have a working Linear Regression model. Let's try it out on a few instances from the training set:

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:", lin_reg.predict(some_data_prepared))
Predictions: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]
>>> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

E2E.5. Select and Train a Model – Training and Evaluating on the Training Set

It works, although the predictions are not exactly accurate (e.g., the first prediction is off by close to 40%!). Let's measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error()` function:

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.19819848922
```

This is better than nothing, but clearly not a great score: most districts' median_housing_values range between ***$120,000 and $265,000, so a typical prediction error of $68,628 is not very satisfying***. This is an example of a model underfitting the training data. ***When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough.*** As we saw in the previous sections, the main ways to fix underfitting are to ***select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model***. **This model is not regularized**, which rules out the last option.
You could try to add more features (e.g., the log of the population), but first let's try a more complex model to see how it does.

E2E.5. Select and Train a Model - Training and Evaluating on the Training Set

Let's train a DecisionTreeRegressor. This is a powerful model, capable of finding complex nonlinear relationships in the data (Decision Trees are presented in more detail in the next sections). The code should look familiar by now:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

Now that the model is trained, let's evaluate it on the training set:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data. How can you be sure? As we saw earlier, you ***don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use part of the training set for training and part of it for model validation.***

E2E.5. Select and Train a Model - Better Evaluation Using Cross-Validation

One way to evaluate the Decision Tree model would be to use the train_test_split() function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set. It's a bit of work, but nothing too difficult, and it would work fairly well. A great alternative is to use Scikit-Learn's *K-fold cross-validation* feature. The following code randomly splits the training set into 10 distinct subsets called *folds*, then it ***trains and evaluates the Decision Tree model 10 times***, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

WARNING

Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the MSE (i.e., a negative value), which is why the preceding code computes `-scores` before calculating the square root.

E2E.5. Select and Train a Model - Better Evaluation Using Cross-Validation

Let's look at the results:

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
 71115.88230639 75585.14172901 70262.86139133 70273.6325285
 75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

Now the Decision Tree doesn't look as good as it did earlier. In fact, it seems to perform worse than the Linear Regression model! Notice that cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation). The Decision Tree has a score of approximately 71,407, generally ±2,439. You would not have this information if you just used one validation set. But cross-validation comes at the cost of training the model several times, so it is not always possible.

E2E.5. Select and Train a Model – Better Evaluation Using Cross-Validation

Let's compute the same scores for the Linear Regression model just to be sure:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                              scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [66782.73843989 66960.118071   70347.95244419 74739.57052552
 68031.13388938 71193.84183426 64969.63056405 68281.61137997
 71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798348
```

That's right: the Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model.

E2E.5. Select and Train a Model – Better Evaluation Using Cross-Validation

Let's try one last model now: the RandomForestRegressor. As we may see in advanced examples (beyond the scope of the presentation), Random Forests ***work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Building a model on top of many other models*** is called *Ensemble Learning*, and it is often a great way to push ML algorithms even further. We will skip most of the code since it is essentially the same as for the other models:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
18603.515021376355
>>> display_scores(forest_rmse_scores)
Scores: [49519.80364233 47461.9115823  50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574  47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

Wow, this is much better: Random Forests look very promising. However, note that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set. Possible solutions for overfitting are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. Before you dive much deeper into Random Forests, however, you should try out many other models from various categories of Machine Learning algorithms (e.g., ***several Support Vector Machines with different kernels, and possibly a neural network***), without spending too much time tweaking the hyperparameters.

The goal is to shortlist a few (two to five) promising models.

E2E.5. Select and Train a Model - Better Evaluation Using Cross-Validation

TIP

You should save every model you experiment with so that you can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps the actual predictions as well. This will allow you to easily compare scores across model types, and compare the types of errors they make. You can easily save Scikit-Learn models by using Python's `pickle` module or by using the `joblib` library, which is more efficient at serializing large NumPy arrays (you can install this library using pip):

```
import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

E2E.6. Fine-Tune the Model – Grid Search

Let's assume that you now have a ***shortlist of promising models***. You now need to fine-tune them. One option would be to ***fiddle*** with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations.

Instead, you should ***get Scikit-Learn's GridSearchCV to search*** for you. All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code searches for the ***best combination*** of hyperparameter values for the RandomForestRegressor:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

In this example, we obtain the best solution by setting the `max_features` hyperparameter to `8` and the `n_estimators` hyperparameter to `30`. The RMSE score for this combination is 49,682, which is slightly better than the score you got earlier using the default hyperparameter values (which was 50,182). Congratulations, you have successfully fine-tuned your best model!

E2E.6. Fine-Tune the Model - Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. Now is the time to ***evaluate the final model on the test set***. There is nothing special about this process; just get the predictors and the labels from the test set, run your full_pipeline to transform the data (call ***transform***(), *not* ***fit_transform***()—you do not want to fit the test set!), and evaluate the final model on the test set:

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 47,730.2
```

E2E.6. Fine-Tune the Model – Evaluate Your System on the Test Set

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% *confidence interval* for the generalization error using scipy.stats.t.interval():

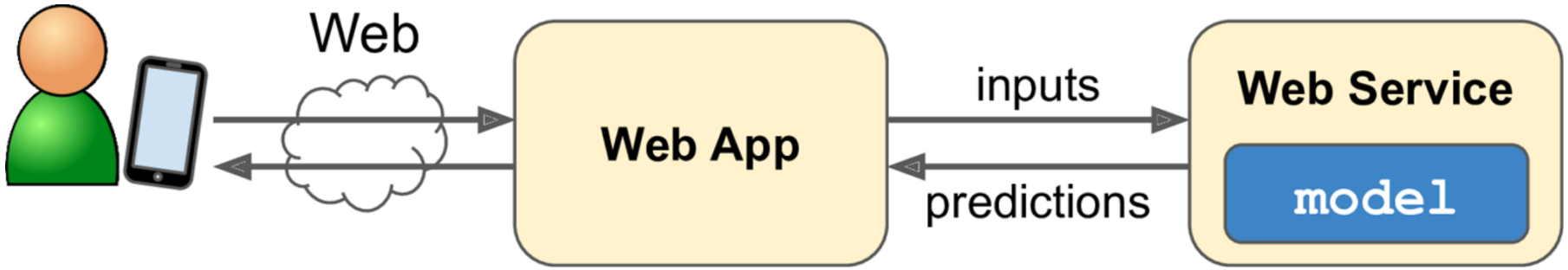
```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                          loc=squared_errors.mean(),
...                          scale=stats.sem(squared_errors)))
...
array([45685.10470776, 49691.25001878])
```

If you did a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation (because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets). It is not the case in this example, but when this happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

DEMO 2 in Python/Jupiter - End2End Project

E2E.7. Present the solution => Launch, Monitor, and Maintain Your System

Perfect, you got approval to launch! You now need to get your solution ready for production (e.g., polish the code, write documentation and tests, and so on). Then you can deploy your model to your production environment. One way to do this is to save the trained Scikit-Learn model (e.g., using joblib), including the full preprocessing and prediction pipeline, then load this trained model within your production environment and use it to make predictions by calling its predict() method. For example, perhaps the model will be used within a website: the user will type in some data about a new district and click the Estimate Price button. This will send a query containing the data to the web server, which will forward it to your web application, and finally your code will simply call the model's predict() method (you want to load the model upon server startup, rather than every time the model is used). Alternatively, you can wrap the model within a dedicated web service that your web application can query through a REST API.



E2E.7. Present the solution => Launch, Monitor, and Maintain Your System

Another popular strategy is to deploy your model on the cloud, for example on Google Cloud AI Platform (formerly known as Google Cloud ML Engine): just save your model using joblib and upload it to ***Google Cloud Storage (GCS), then head over to Google Cloud AI Platform and create a new model version, pointing it to the GCS file. That's it!*** This gives you a simple web service that takes care of load balancing and scaling for you. It take JSON requests containing the input data (e.g., of a district) and returns JSON responses containing the predictions. You can then use this web service in your website (or whatever production environment you are using). ***As we will find out, deploying TensorFlow models on AI Platform is not much different from deploying Scikit-Learn models.***

But deployment is not the end of the story. You also need to write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.

WARNING

Even a model trained to classify pictures of cats and dogs may need to be retrained regularly, not because cats and dogs will mutate overnight, but because cameras keep changing, along with image formats, sharpness, brightness, and size ratios. Moreover, people may love different breeds next year, or they may decide to dress their pets with tiny hats—who knows?

Perceptron, MLP, ANN FFN with Back-propagation, CNNs, etc.

AI NEURAL NETWORKS: Deep Learning

2. AI - NN

Artificial Neural Networks with Keras

Birds inspired us to fly, Human Eye inspired Cameras with Wider Field of View, and nature has inspired countless more inventions.

It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked ***artificial neural networks (ANNs): an ANN is a Machine Learning model inspired by the networks of biological neurons found in our brains***.
However, although planes were inspired by birds, they don't have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins.

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks such as classifying

- *billions of images (e.g., Google Images),*
- *powering speech recognition services (e.g., Apple's Siri),*
- *recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or*
- *learning to beat the world champion at the game of Go (DeepMind's AlphaGo), etc.*

2. AI – NN | Artificial Neural Networks with Keras

From Biological to Artificial Neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in **1943** by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their landmark paper: "***A Logical Calculus of Ideas Immanent in Nervous Activity***", *McCulloch and Pitts* presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using *propositional logic*. ***This was the first artificial neural network architecture***. Since then many other architectures have been invented, as we will see.

The early successes of ANNs led to the widespread belief that we would soon be conversing with ***truly intelligent machines***. When it became clear in the **1960s** that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. In the early **1980s**, ***new architectures were invented and better training techniques were developed***, ***sparking*** a revival of interest in ***connectionism*** (the study of neural networks). But progress was slow, and by the **1990s** other powerful **Machine Learning** techniques were invented, such as **Support Vector Machine - SVM**. ***These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.***

2. AI – NN | Artificial Neural Networks with Keras

From Biological to Artificial Neurons

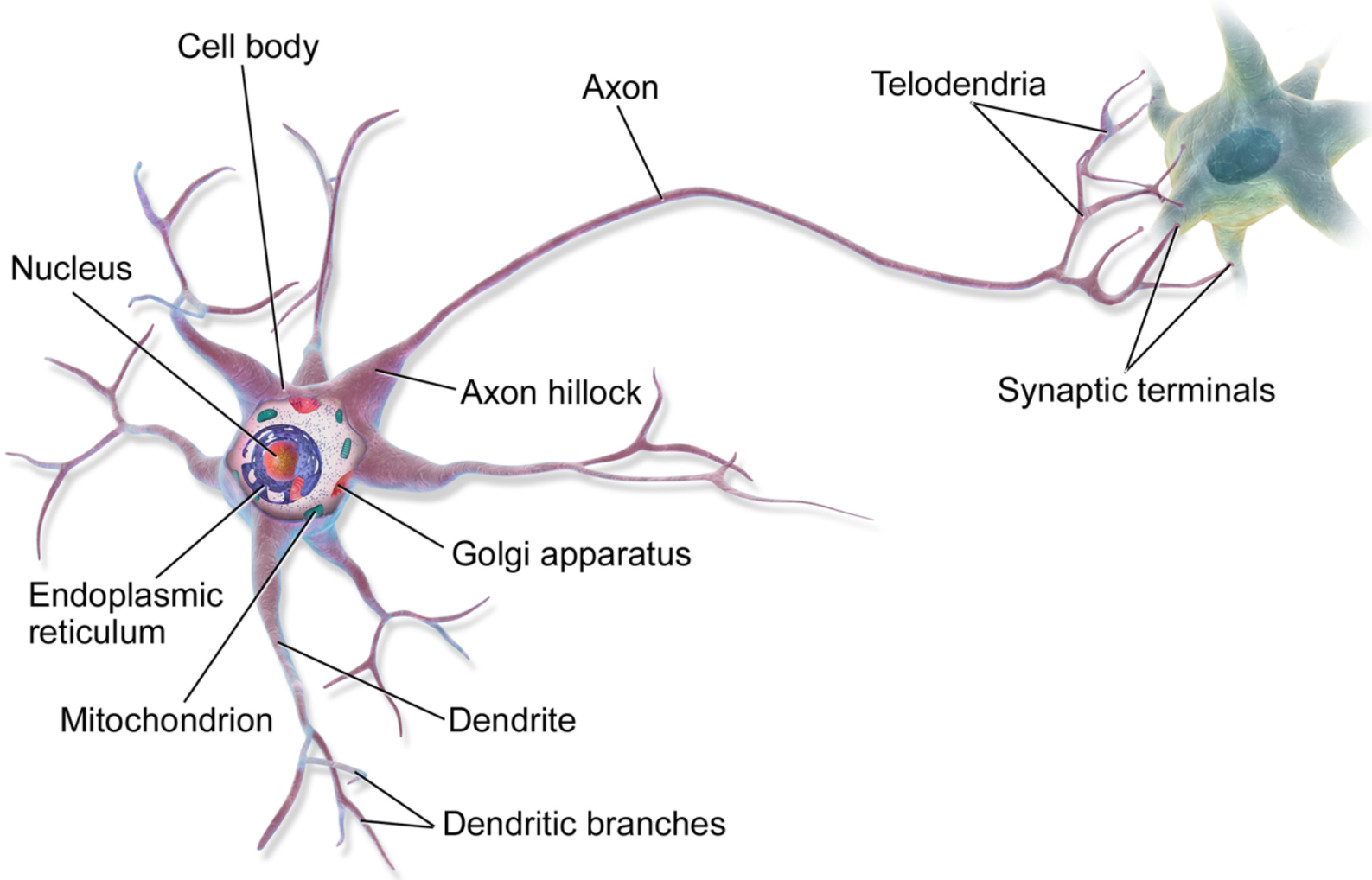
We are now (**2018**) witnessing yet another wave of interest in **ANNs**. ***Will this wave die out like the previous ones did?*** Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

- There is now a ***huge quantity of data (and BigData products)*** available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The ***tremendous increase in computing power since the 1990s*** now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's law (the number of components in integrated circuits has doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions. Moreover, ***cloud platforms*** have made this power accessible to everyone.
- ***The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s***, but these relatively small tweaks have had a huge positive impact.
- ***Some theoretical limitations of ANNs have turned out to be benign in practice***. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (and when it is the case, they are usually fairly close to the global optimum).
- ***ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news***, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

2. AI – NN | Artificial Neural Networks with Keras

Biological Neurons

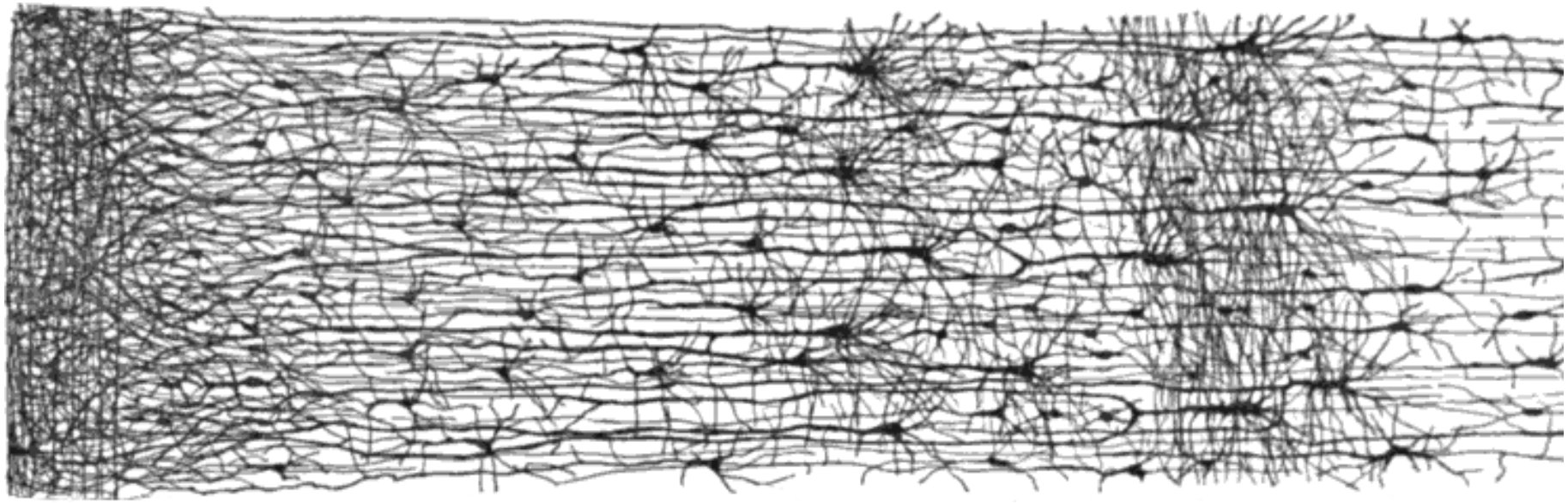
Before we discuss artificial neurons, let's take a quick look at a biological neuron. It is an unusual-looking cell mostly found in animal brains. It's composed of a cell body containing the nucleus and most of the cell's complex components, many branching extensions called dendrites, plus one very long extension called the axon. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called telodendria, and at the tip of these branches are minuscule structures called synaptic terminals (or simply synapses), which are ***connected to the dendrites or cell bodies of other neurons.*** Biological neurons produce short electrical impulses called action potentials (APs, or just signals) which travel along the axons and make the synapses release chemical signals called neurotransmitters. ***When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).***



Biological neuron

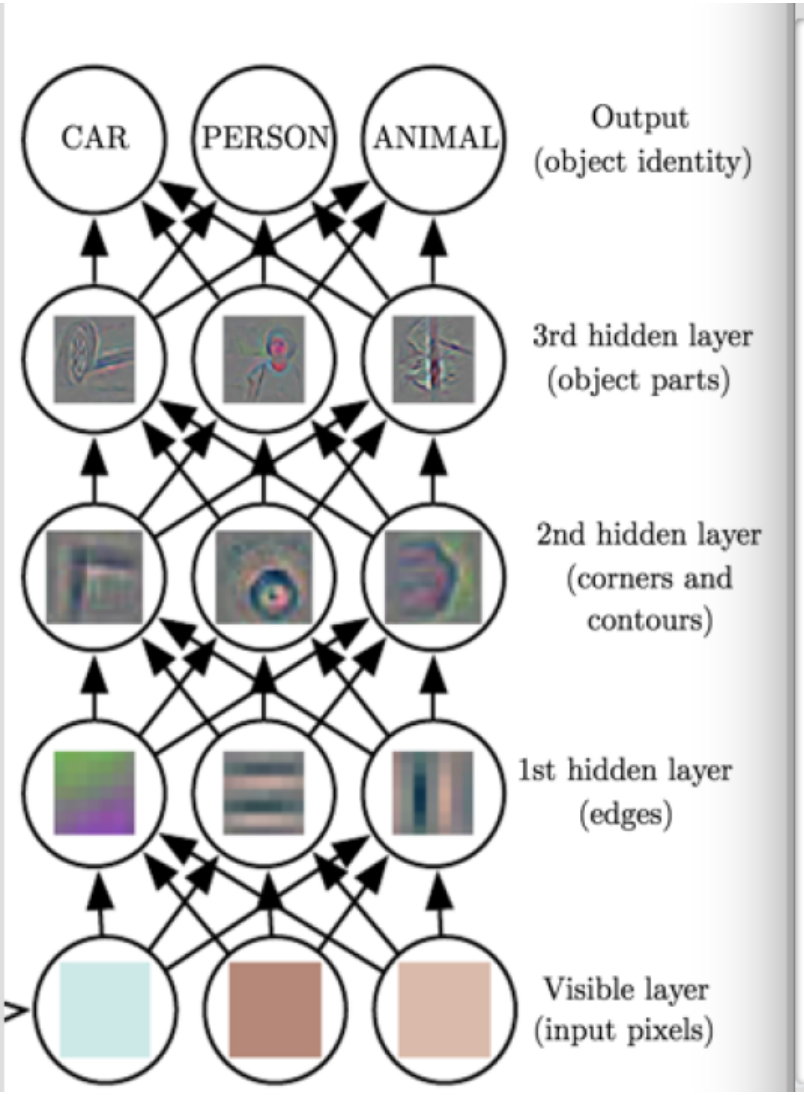
Artificial Neural Networks with Keras

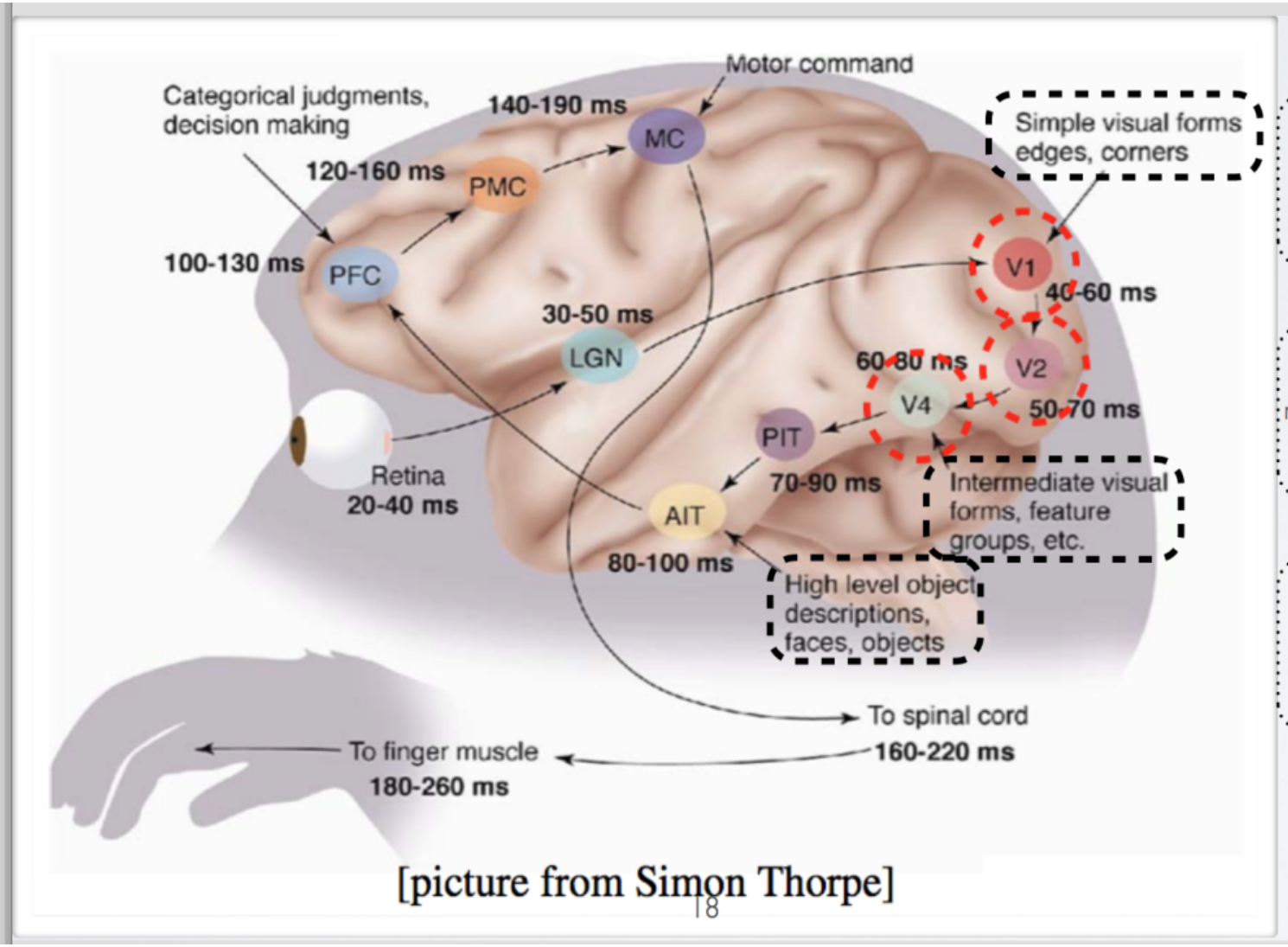
Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of ***biological neural networks (BNNs)*** is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, especially in the cerebral cortex (i.e., the outer layer of your brain):

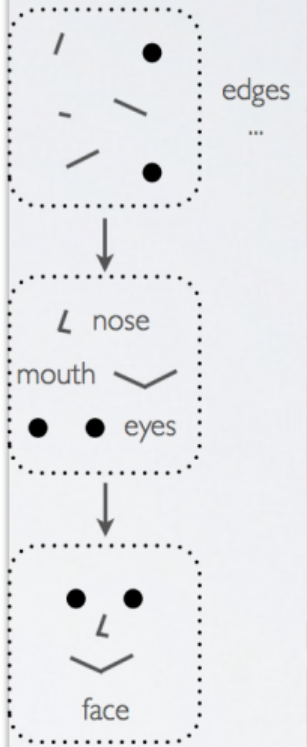


Multiple layers in a biological neural network (human cortex)

2. AI – NN | Artificial Neural Networks



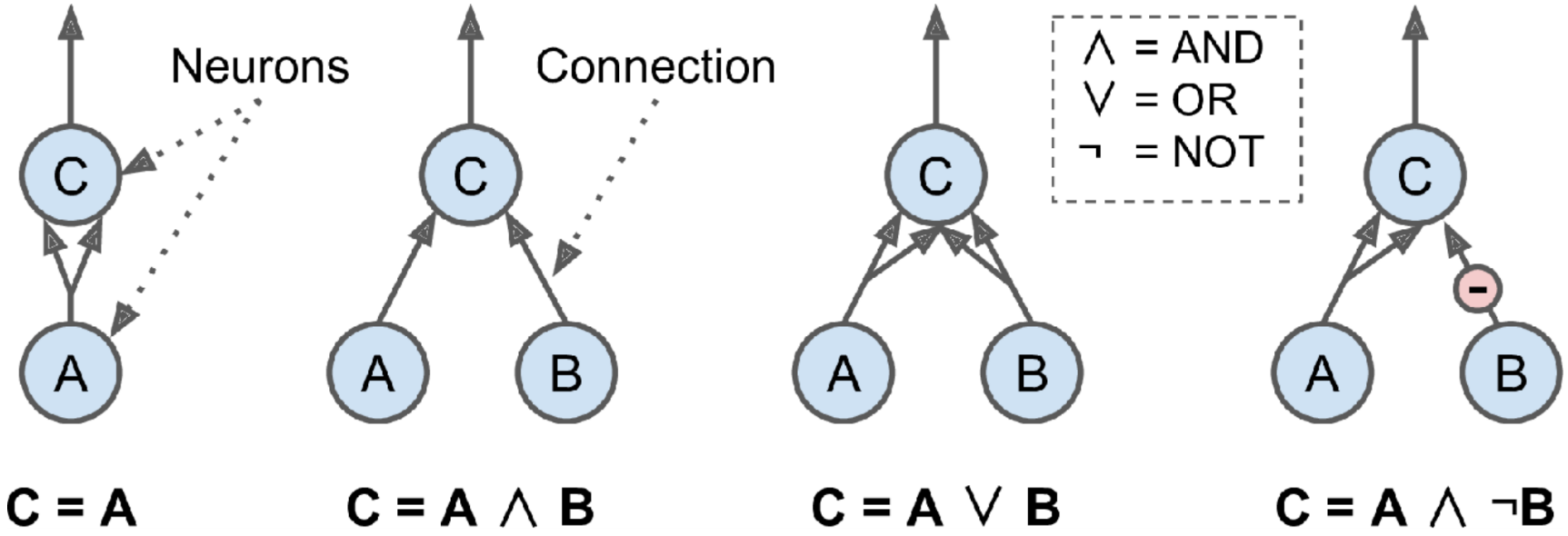




2. AI – NN | Artificial Neural Networks with Keras

Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an ***artificial neuron***: ***it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active***. In their paper, they showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. ***To see how such a network works, let's build a few ANNs that perform various logical computations, assuming that a neuron is activated when at least two of its inputs are active.***

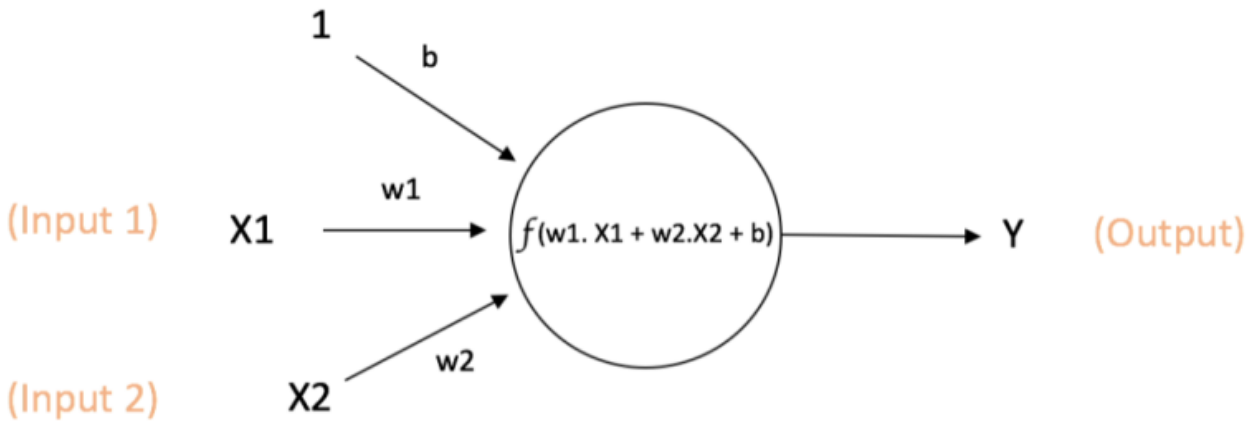


2. AI – NN | Artificial Neural Networks with Keras

The Neuron

A Single Neuron

The basic unit of computation in a neural network is the **neuron**, often called a **node** or **unit**. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated **weight** (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function f (defined below) to the weighted sum of its inputs as shown in Figure 1 below:



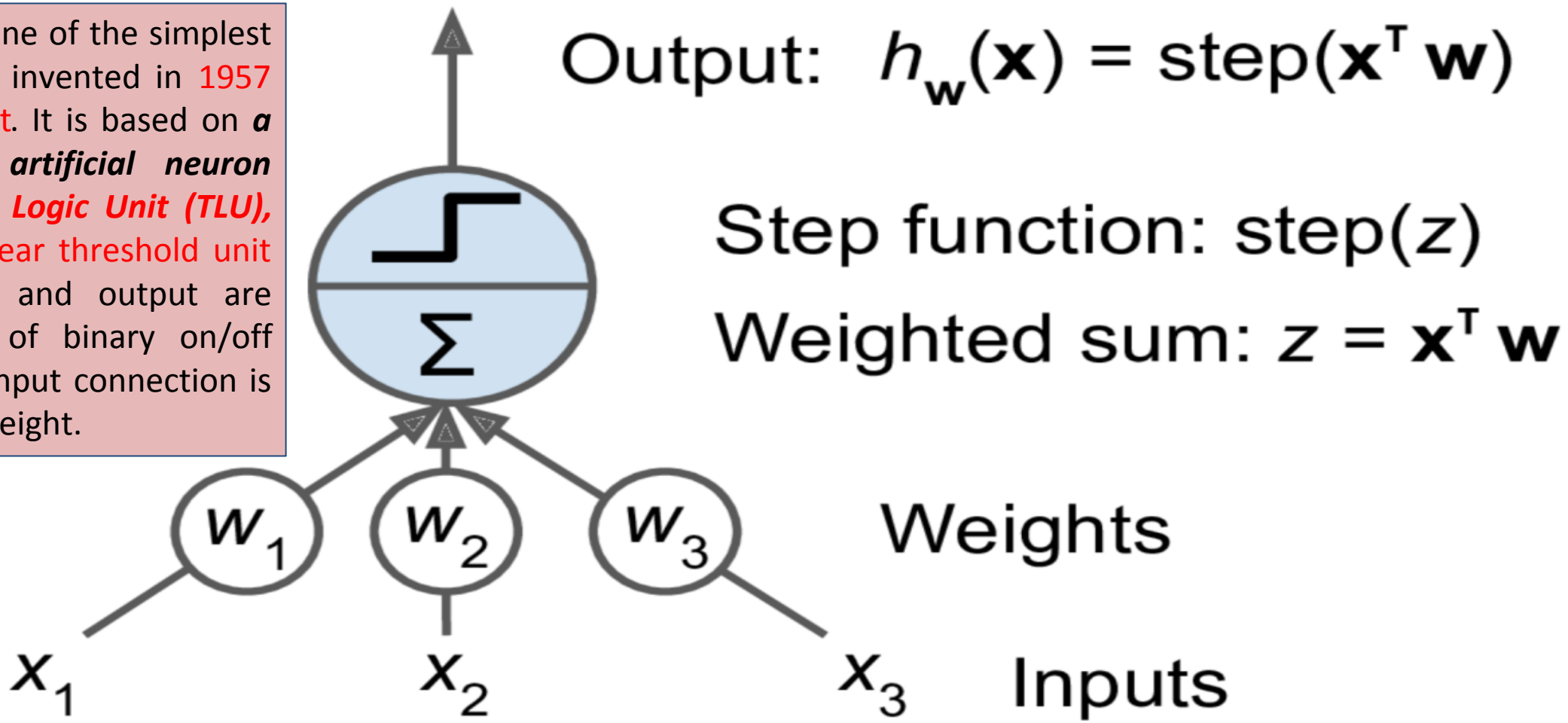
Output of neuron = Y= $f(w1.\ X1 + w2.X2 + b)$

2. AI – NN | Artificial Neural Networks with Keras

The Perceptron

The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{x}^\intercal \mathbf{w}$), then applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$, where $z = \mathbf{x}^\intercal \mathbf{w}$.

The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on ***a slightly different artificial neuron*** called ***a Threshold Logic Unit (TLU),*** or sometimes a linear threshold unit (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight.



Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function

2. AI – NN | Artificial Neural Networks with Keras

The Perceptron step functions

The most common step function used in Perceptrons is the *Heaviside step function* (see Equation 10-1). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons (assuming threshold = 0)

$$\text{heaviside}\ (z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\ (z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class. ***Otherwise it outputs the negative class (just like a Logistic Regression or linear SVM classifier).*** *You could, for example, use a single TLU to classify iris flowers based on petal length and width (also adding an extra bias feature $x_0 = 1$, just like we did in previous sections). Training a TLU in this case means finding the right values for w_0, w_1, and w_2 (the training algorithm is discussed shortly).*

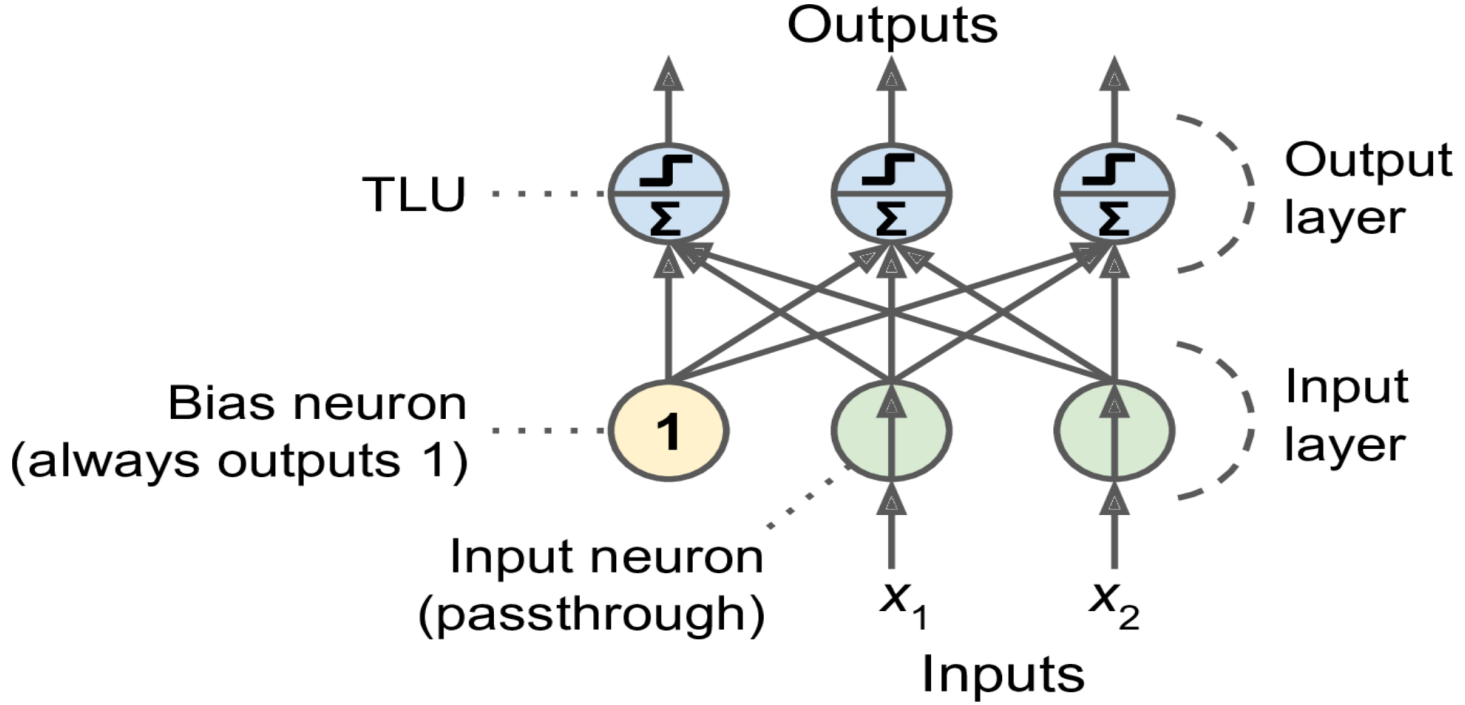
The Perceptron

A Perceptron is simply composed of a single layer of TLUs, ***with each TLU connected to all the inputs***. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is called a ***fully connected layer***, or a ***dense layer***. ***The inputs of the Perceptron are fed to special pass-through neurons called input neurons: they output whatever input they are fed***. All the input neurons form the ***input layer.***

Moreover, an extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a ***bias neuron***, which outputs 1 all the time. ***A Perceptron with two inputs and three outputs (and 1 bias)*** is represented in figure and ***this Perceptron can classify instances simultaneously into three different binary classes, which makes it a multi-output classifier***.

Thanks to the magic of linear algebra, the equation (*Computing the outputs of a fully connected layer*) makes it possible to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

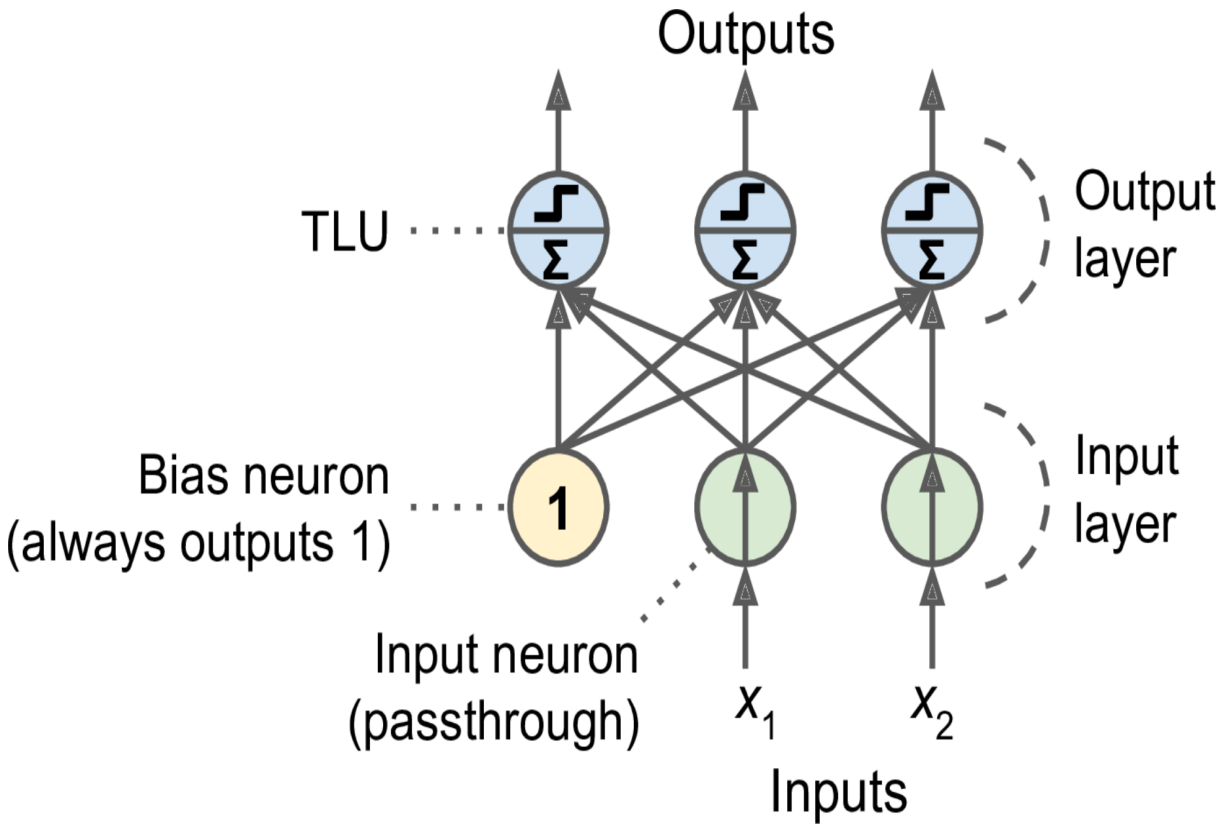


The Perceptron

In this equation:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

- As always, **X** represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix **W** contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector **b** contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function **ϕ** is called the *activation function*: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).



The Perceptron

So, *how is a Perceptron trained?* The Perceptron training algorithm proposed by Rosenblatt was largely inspired by ***Hebb's rule***. In his *1949 book* ***The Organization of Behavior*** *(Wiley), Donald Hebb* suggested that ***when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger***. Siegrid Löwel later summarized Hebb's idea in the catchy phrase, ***"Cells that fire together, wire together"***; that is, the connection weight between two neurons tends to increase when they fire simultaneously. ***This rule later became known as Hebb's rule (or Hebbian learning).*** Perceptrons ***are trained using a variant of this rule*** that takes into account the error made by the network when it makes a prediction; ***the Perceptron learning rule reinforces connections that help reduce the error.*** More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta \left(y_j - \hat{y}_j\right) x_i$$

In this equation:

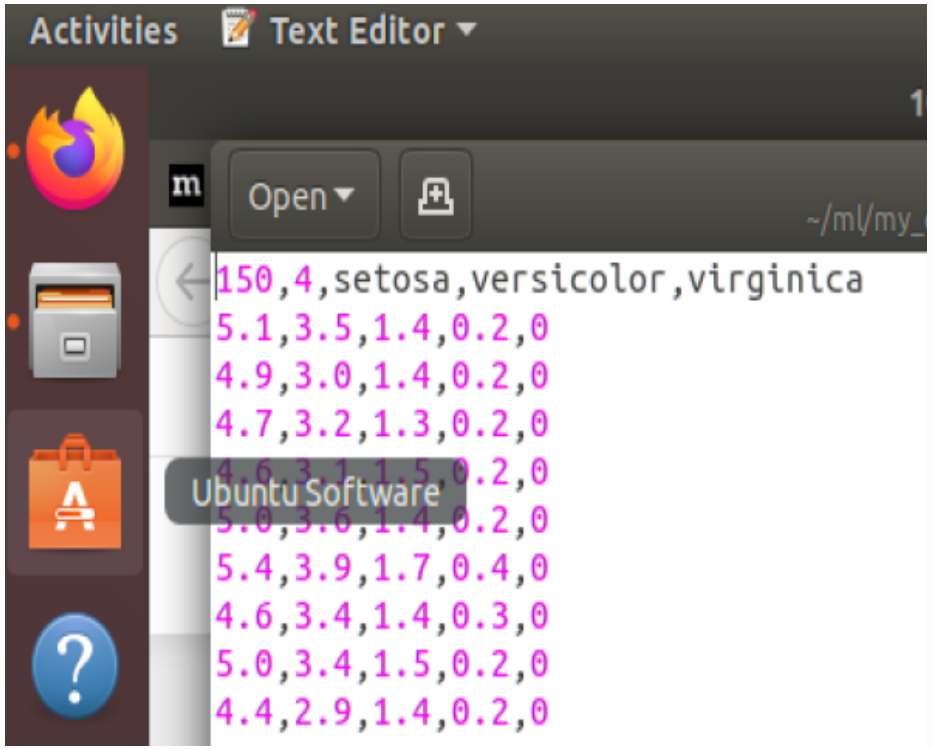
Perceptron learning rule (weight update)

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution. This is called the ***Perceptron convergence theorem***.

The Perceptron

Scikit-Learn provides a Perceptron class that implements ***a single-TLU network***. It can be used pretty much as you would expect: for example, on the iris dataset (introduced in previous sections as **MNIST** - Modified National Institute of Standards and Technology database)



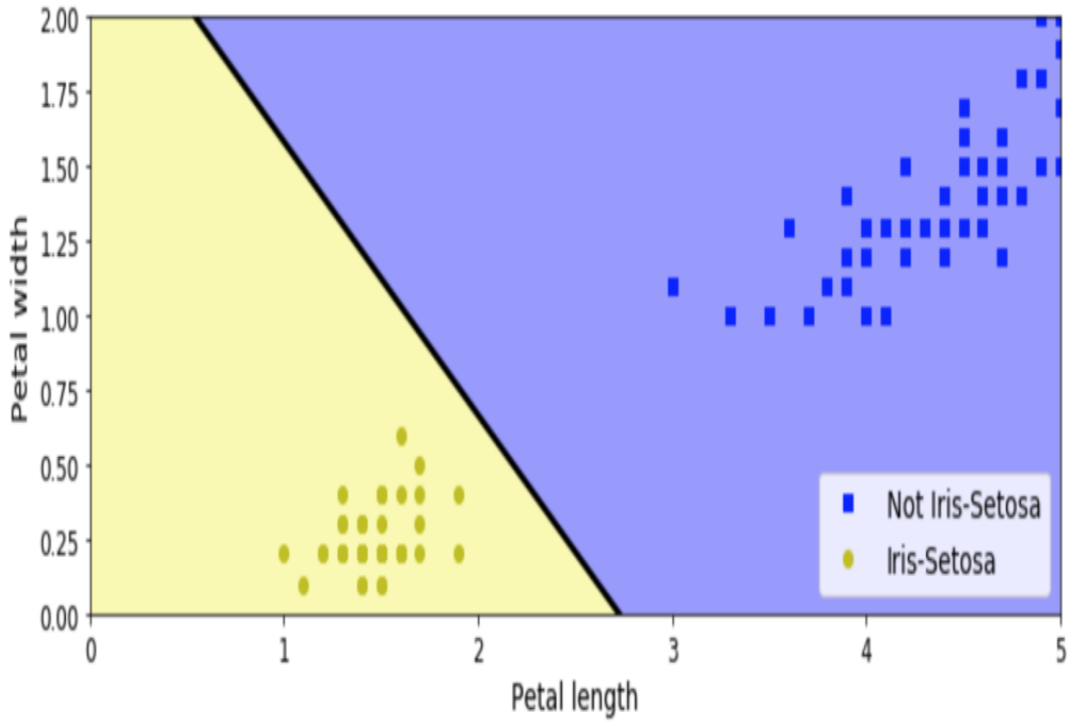

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

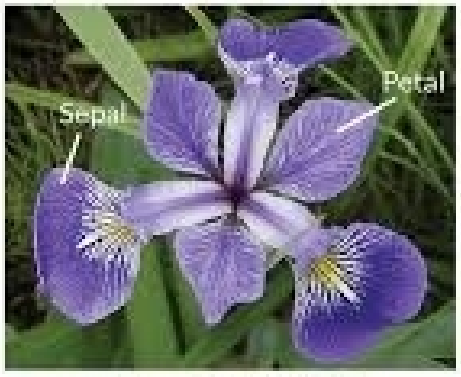
2. AI - NN | Artificial Neural Networ

The Perceptron

```
a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-", linewidth=3)
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)

save_fig("perceptron_iris_plot")
plt.show()
```

Saving figure perceptron_iris_plot





Iris Versicolor

Iris Setosa

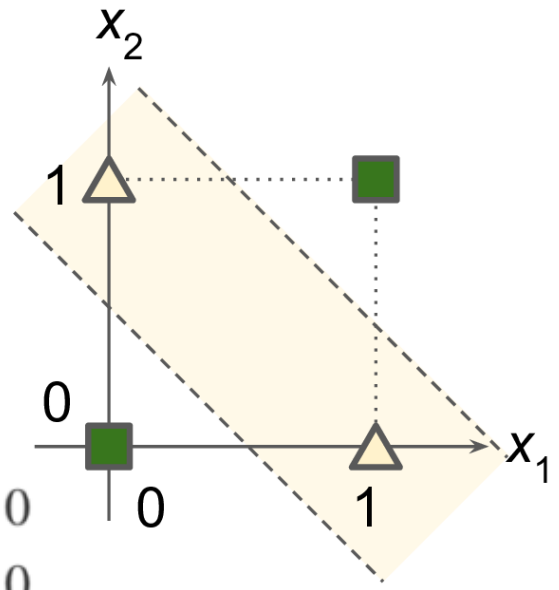
Iris Virginica

MLP - Multi-Layer Perceptron

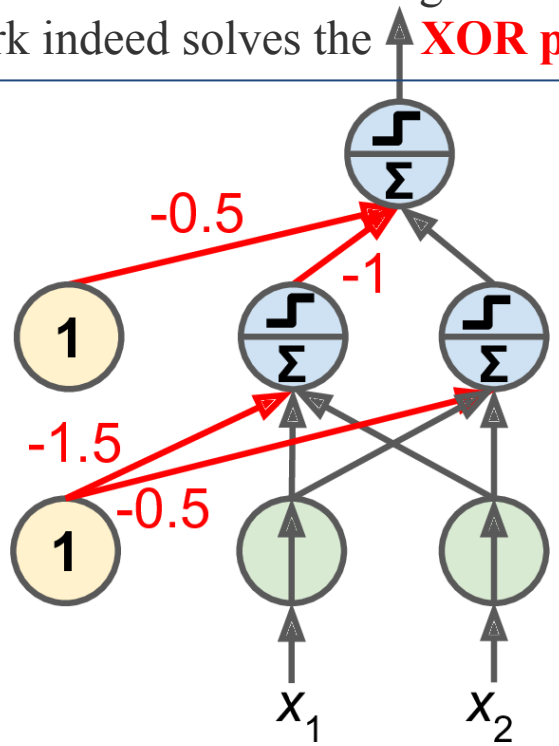
In their **1969** monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons—in particular, the fact that ***they are incapable of solving some trivial problems (e.g., the Exclusive OR (XOR) classification problem***; see the left side of figure). ***This is true of any other linear classification model (such as Logistic Regression classifiers)***.

It turns out that some of the limitations of Perceptrons can be eliminated by **stacking multiple Perceptrons**. ***The resulting ANN is called a Multilayer Perceptron (MLP)***. An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the right side of figure: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. All connections have a weight equal to 1, except the four connections where the weight is shown. Try verifying that this network indeed solves the **XOR problem**!

Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0



$$\text{heaviside}\ (z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



(-0.5 => **0**)

(-0.5*1 + 1*(-1) + 1)

(0.5 =>1 , 1.5 => 1)

(-1.5+1*1+1,-0.5+1*1+1)

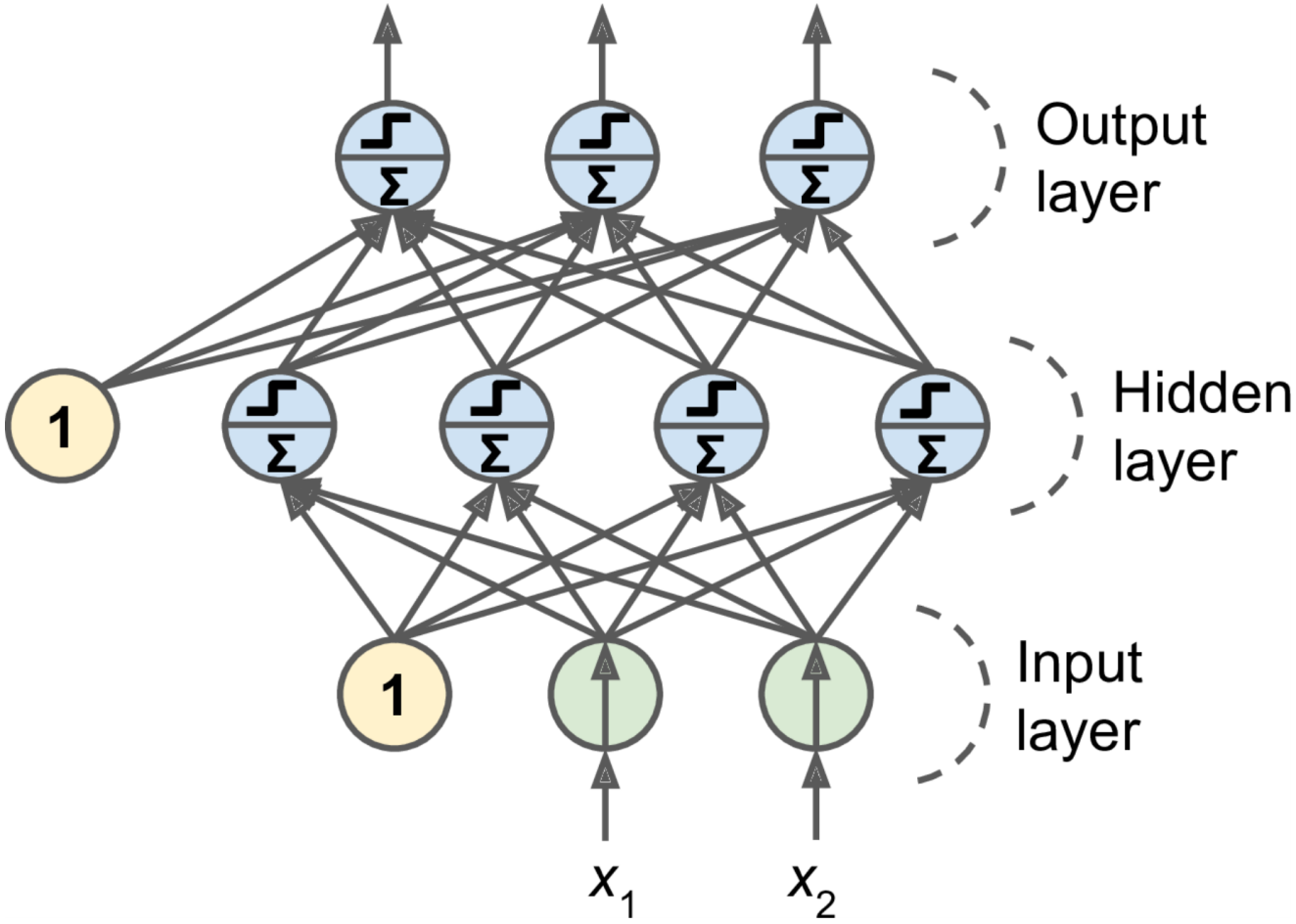
(**1** , **1**)

MLP -Multilayer Perceptron & Back-propagation

An MLP is composed of:

- one (passthrough) ***input layer***,
- one or more layers of TLUs, called ***hidden layers***, and
- one final layer of TLUs called the ***output layer***.

The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a ***feed-forward neural network (FNN).***



Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

MLP -Multilayer Perceptron & Back-propagation

When an **ANN** contains a ***deep stack of hidden layers***, it is called a **deep neural network (DNN).** The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations.

Even so, many people talk about Deep Learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. ***But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper that introduced the back-propagation training algorithm, which is still used today.*** In short, it is **Gradient Descent** (introduced in previous sections) using an efficient technique for computing the gradients automatically:

- in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter.

*In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gradients, it just performs a regular **Gradient Descent** step, and the whole process is repeated until the network converges to the solution.*

MLP -Multilayer Perceptron & Back-propagation

In order for this algorithm (Back-propagation) to work properly, its authors made a key change to the MLP's architecture: they ***replaced the step function*** with the ***logistic (sigmoid) function***, $\sigma(z) = 1 / (1 + \exp(-z))$.

In fact, the **back-propagation algorithm** works well with many other activation functions, not just the logistic function. Here are two other popular choices:

The hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$

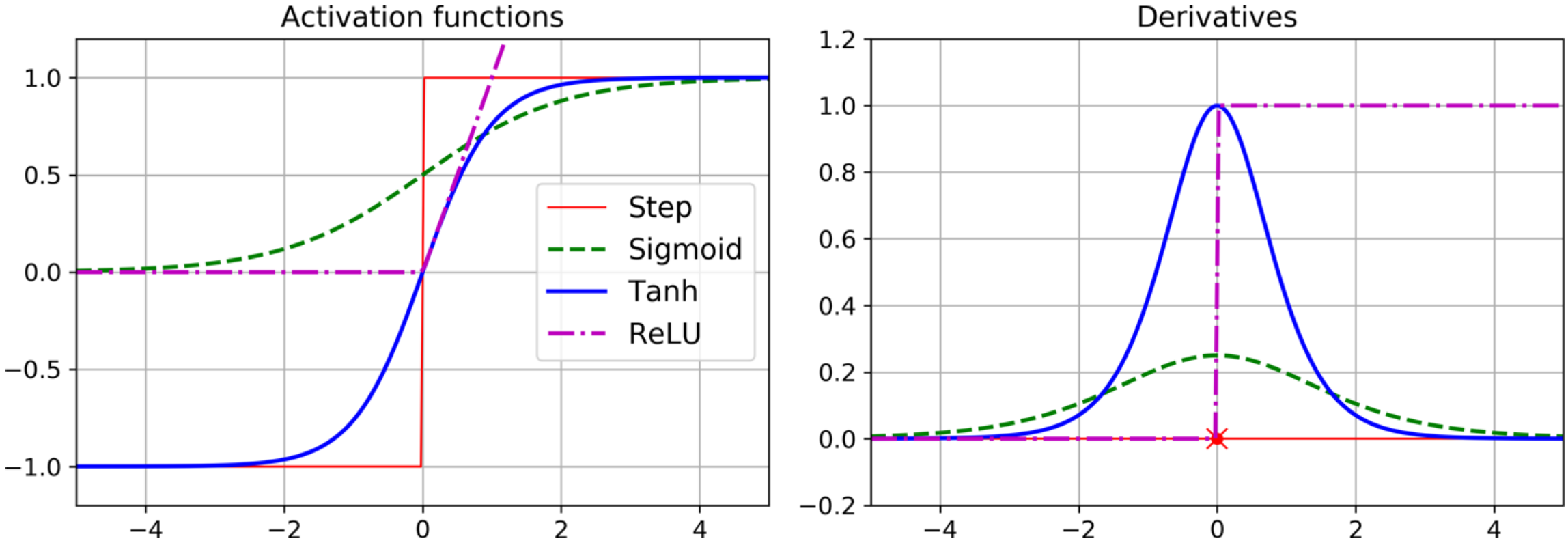
Just like the logistic function, this activation function is *S*-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The Rectified Linear Unit function: $ReLU(z) = \max(0, z)$

The ReLU function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.[12] Most importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent (we will come back to this in Chapter 11).

MLP - Multilayer Perceptron & Back-propagation

These **popular activation functions** and their derivatives are represented in figure:



Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

OK! You know where neural nets came from, what their architecture is, and how to compute their outputs.

You've also learned about the back-propagation algorithm. But what exactly can you do with them?

Regression & Classification!

Regression MLP - Multilayer Perceptron

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values. If you want to guarantee that the output will always be positive, then you can use the ReLU activation function in the output layer.

Alternatively, you can use the ***softplus** activation function*, which is a smooth variant of ReLU: $\text{softplus}(z) = \log(1 + \exp(z))$. It is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and then scale the labels to the appropriate range: 0 to 1 for the logistic function and –1 to 1 for the hyperbolic tangent.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.

2. AI – NN | Artificial Neural Networks & Deep ML

Regression MLP – **Multilayer Perceptron**

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLP - Multilayer Perceptron

MLPs can also be used for classification tasks. For a ***binary classification problem***, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

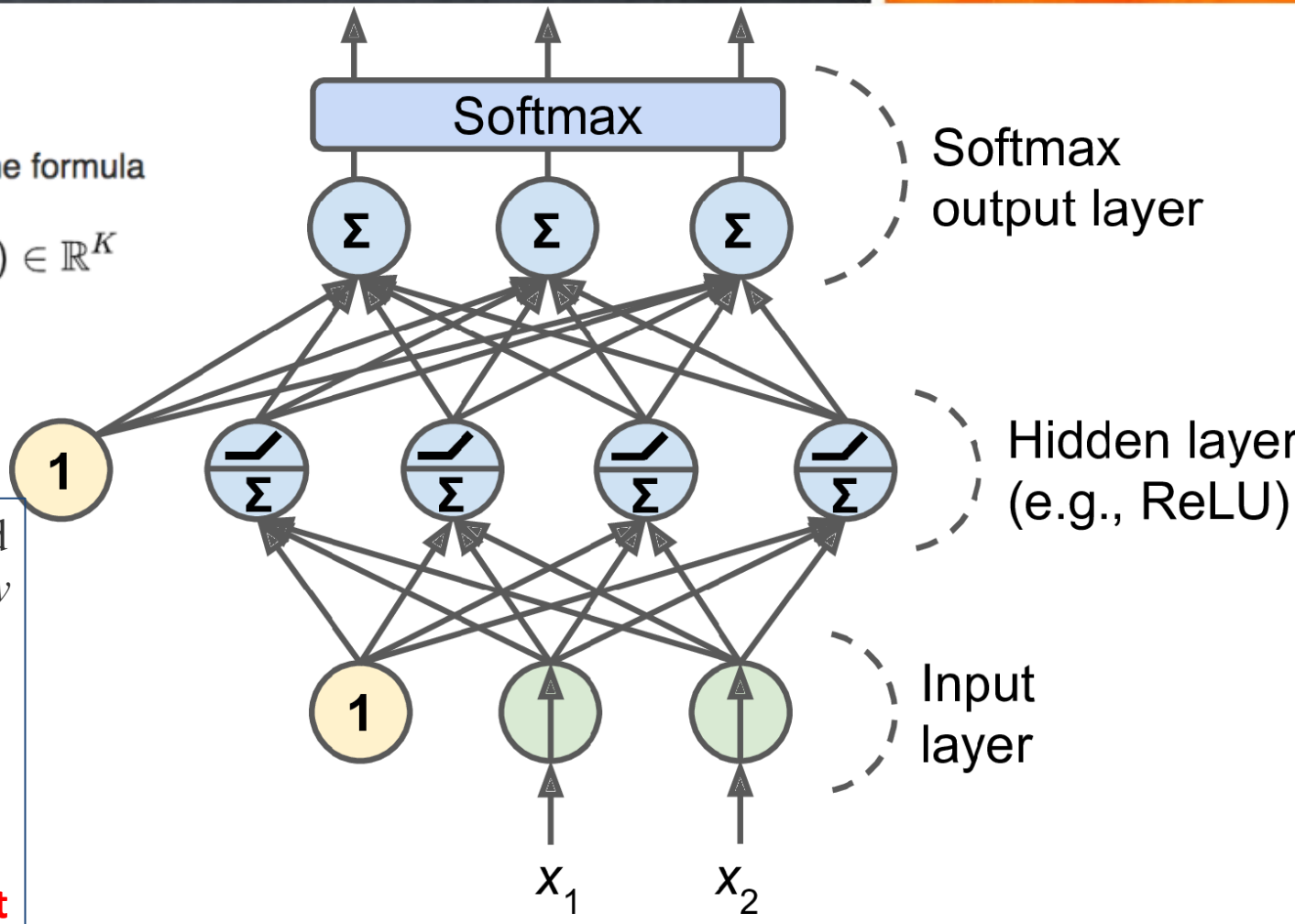
MLPs can also easily handle mu***lti-label binary classification tasks*** (see **Classification**). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or non-urgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have non-urgent ham, urgent ham, non-urgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see figure). The **softmax** function (introduced in Training Models) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive). This is called **multiclass classification**.

Classification MLP - Multilayer Perceptron

The standard (unit) softmax function $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ is defined by the formula

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$



A modern MLP (including ReLU and softmax) for classification

Play with various neural network architectures and visualize their outputs using the *TensorFlow Playground:*

https://playground.tensorflow.org/
https://github.com/tensorflow/playground

Now you have all the concepts you need to start implementing MLPs with Keras! BUT after going through full numeric example and returning to the first JAVA example from the previous slides.

Classification MLP - Multilayer Perceptron

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

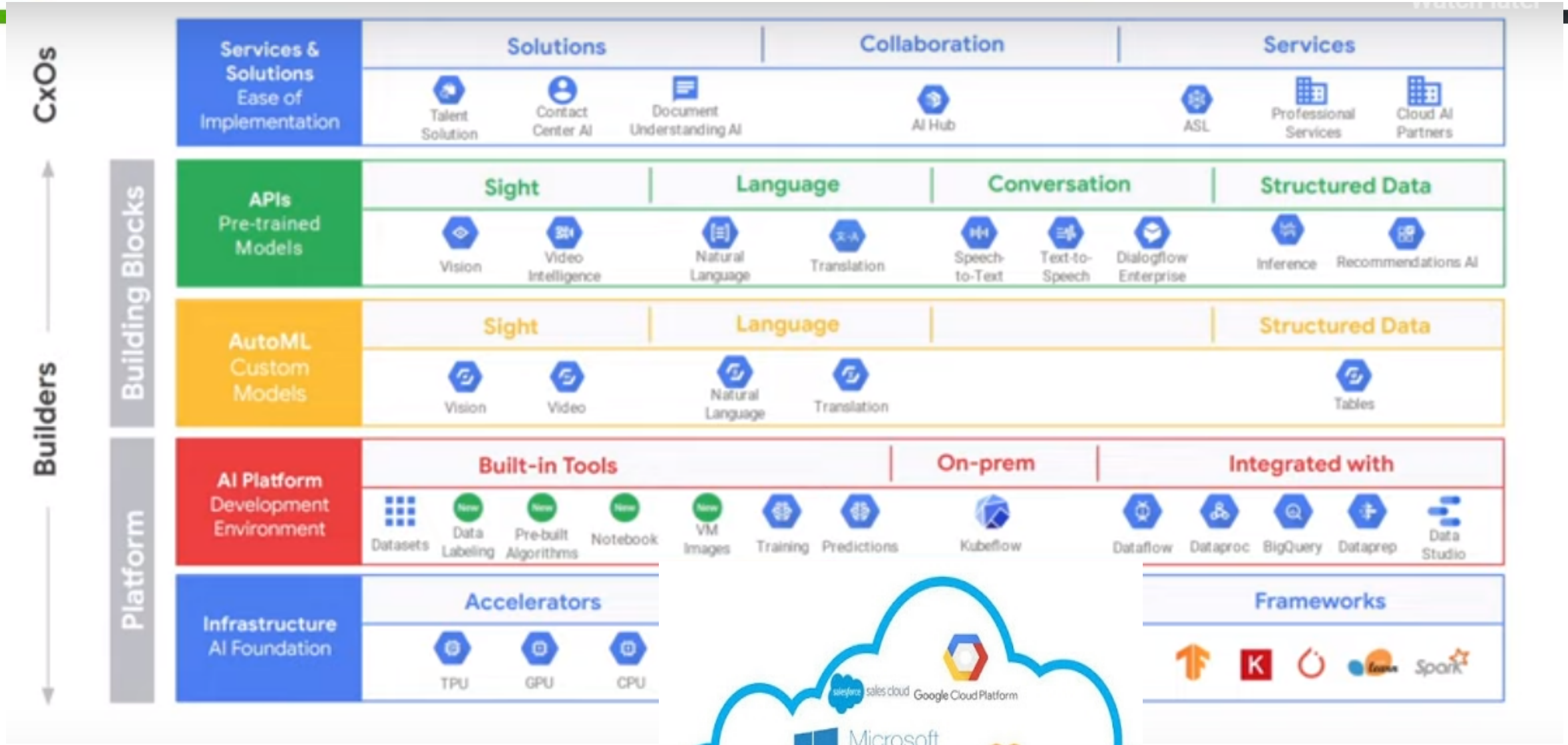
How would be the future on the A.I / M.L. & Deep Learning?

Communicate & Exchange Ideas

Tools & Install – Python, PIP, Jupyter, SciKit, TenserFlow, Keras

```
# sudo rm /var/cache/apt/archives/lock # python --version
# sudo rm /var/lib/dpkg/lock # install python if necessary
sudo apt install python-pip
sudo apt-get install python3-pip
sudo pip install --upgrade pip
sudo pip3 install --upgrade pip

python3 -m pip --version
python3 -m pip install --user -U virtualenv
$ export ML_PATH="$HOME/ml" # You can change the path if you prefer
$ mkdir -p $ML_PATH
$ cd $ML_PATH
$ python3 -m virtualenv my_env

$ source my_env/bin/activate # on Linux or macOS
python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn
python3 -m ipykernel install --user --name=python3

python3 -m pip install -U tensorflow & python3 -m pip install -U pydot & python3 -m pip install -U graphviz
sudo apt-get install graphviz
$ jupyter notebook
```

Artificial Intelligence – A.I. & ML Cloud Providers



Q & A


рахмат
danke
謝謝
ngiyabonga
spas
teşekkür ederim
Баярлалаа
mersi
kia ora
barka
welalin
tack
mahalo
tapadh leat
спасибо
faafetai lava
vinaka
спасибі
blagodaram
dank je
misaotra
paldies
grazzi
matondo
kiitos
dankie
nanni
nandri
dhanyavad
mauruuru
thank you
gracias
хвала
asante
manana
obrigada
tenki
bedankt
hvala
bayarlalaa
gràcie
köszönöm
dankon
aciū
akun
enkosi
djiere dieuf
tau
mochchakkeram
murakoze
дякую
mamnun
dziękuję
chnorakaloutioun
gratias ago
gràcies
sulpáy
chokrane
go raibh maith agat
sobodi
dėkuji
taiku
sagolun
sukriya
kop khun krap
obrigado
mèsi
didi madloba
arigatô
takk
dakujem
trugarez
ありがとう
najis tuke
tanemirt
grazie
terima kasih
rahmet
diolch
dhanyavadagalu
shukriya
merce
мерси
kam sah hamnida
rahmat
ευχαριστώ
xiexie
merci
তোমাকে ধন্যবাদ
감사합니다

www: ism.ase.ro | acs.ase.ro | dice.ase.ro

Scan the Tag
to get the web
Mobile Address