

# General vs Narrow AI

Some computer algorithms are capable of mimicking human intelligence, to reason and solve problems on their own, and to apply previously acquired knowledge on completely new types of problems. These algorithms fall into the domain of:

##### Answer the question

**50 XP**

##### Possible Answers

* Artificial Narrow Intelligence (Narrow AI)

press

* Artificial General Intelligence (General AI) **(A)**

press

* Correct! However, General AI is still science fiction, while Narrow AI dominates the current landscape of Data Science.

# Why Python?

Python is the best choice for developing machine learning solutions. Why is that the case?

##### Answer the question

**50 XP**

##### Possible Answers

* Python has a simple and beautiful syntax.

press

* Python is very versatile.

press

* Python is very flexible.

press

* Python offers rich AI-related libraries.

press

* The Python community is big and growing fast.

press

* All of the above. **(A)**

Correct! And precisely for all of these reasons Python's position as the top language for Data Science and Machine Learning will hardly be disputed in the near future.

# The elephant in the room

To further illustrate the concept of Narrow AI, let's see what happens when an algorithm trained for one problem, is given a completely unrelated input and asked for a prediction.

Specifically, what happens when you feed a picture of an elephant into a model that is trained to recognize handwritten digits?

To make things simple, a digit recognition model has been pre-trained. Your task is to feed the elephant image into it and see the result.

##### Instructions

**100 XP**

* Feed the elephant\_image into the digit\_predictor.

In [3]: elephant\_image

Out[3]:

array([[0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0,

0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,

0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1]])

# Load the test example

elephant\_image = load\_elephant()

# Apply the model on the test example

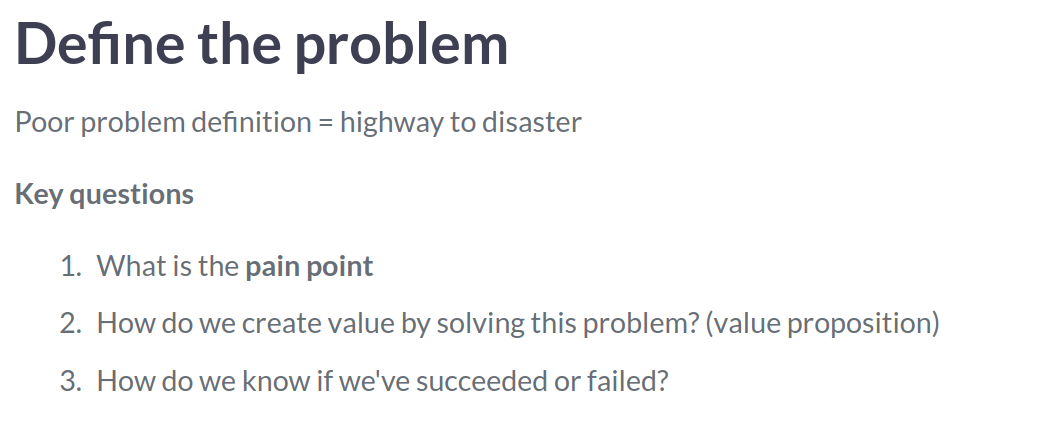
test\_digit\_predictor(elephant\_image, 'elephant')

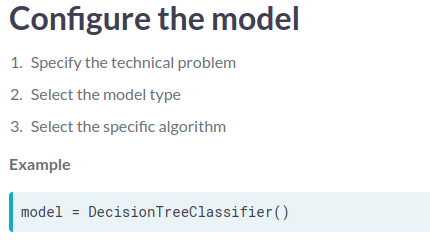
<script.py> output:

Actual digit: elephant

Recognized digit: 7

Interesting, right? Although being the pinnacle of technology, AI systems can also be easily misused and abused -- even tricked, if that is the intention!





# Parameters & hyper-parameters

Let's explore the impact of hyper-parameters on the performance of your model.

You are building a digit recognition algorithm using the LogisticRegression classifier and you are challenged to find the optimal value of its regularization parameter, C.

Lower values of C make the model more conservative and resilient to noise, while higher values give it more flexibility and capacity to capture the nuances. If you increase it too much, you risk overfitting.

Where is the "sweet spot"?

##### Instructions 1/4

**100 XP**

* [1](javascript:void(0))
  + Set the value of hyper-parameter C to 0.000001.

# Define the model

model = LogisticRegression(C=0.000001)

# Fit the model

model.fit(training\_inputs, training\_labels)

# Check model performance

check\_performance(model, testing\_inputs, testing\_labels)

<script.py> output:

Accuracy: 59.30 %

##### Instructions 2/4

**0 XP**

* [2](javascript:void(0))
  + Set the value of hyper-parameter C to 0.0001.

# Define the model

model = LogisticRegression(C=0.0001)

# Fit the model

model.fit(training\_inputs, training\_labels)

# Check model performance

check\_performance(model, testing\_inputs, testing\_labels)

<script.py> output:

Accuracy: 89.19 %

Set the value of hyper-parameter C to 0.01.

# Define the model

model = LogisticRegression(C=0.01)

# Fit the model

model.fit(training\_inputs, training\_labels)

# Check model performance

check\_performance(model, testing\_inputs, testing\_labels)

<script.py> output:

Accuracy: 94.59 %

Set the value of hyper-parameter C to 100.

# Define the model

model = LogisticRegression(C=100)

# Fit the model

model.fit(training\_inputs, training\_labels)

# Check model performance

check\_performance(model, testing\_inputs, testing\_labels)

<script.py> output:

Accuracy: 92.53 %

Good job! Your model was obviously over-regularized in the first iteration and increasing C gave it some slack to capture the finer variations in the data. However, increasing parameters beyond the "sweet spot" is not only useless, but often detrimental!

# Too much of a good thing

Let's inspect the issue of **overfitting** in a more visual way.

Say you have measured the temperature in your office for several days and you want to create a model to describe these oscillations.

You have (wisely) decided to tackle this challenge using a **polynomial model**. Your task is now to find the right value of the hyper-parameter representing the **degree of the polynomial.**

**But be careful!** As your measurements are quite noisy (because you used a cheap thermometer), you're in **danger of overfitting** if you exaggerate with model complexity.

A fit\_polynomial() function has been defined for you.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))
  + Set the degree of the polynomial to 1.

# Set degree to 1

fit\_polynomial(degree=1)

* + Set the degree of the polynomial to 4.

# Set degree to 4

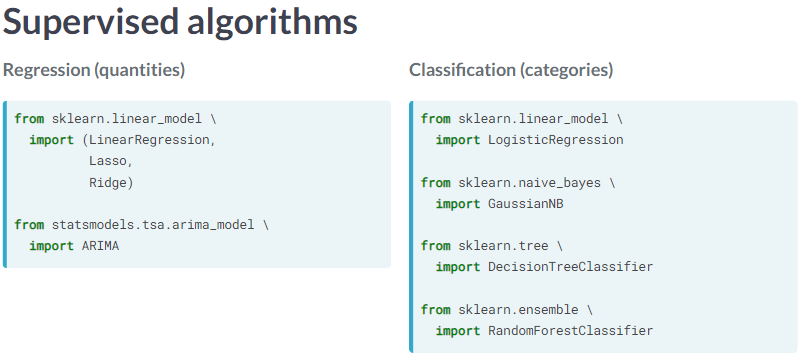
fit\_polynomial(degree=4)

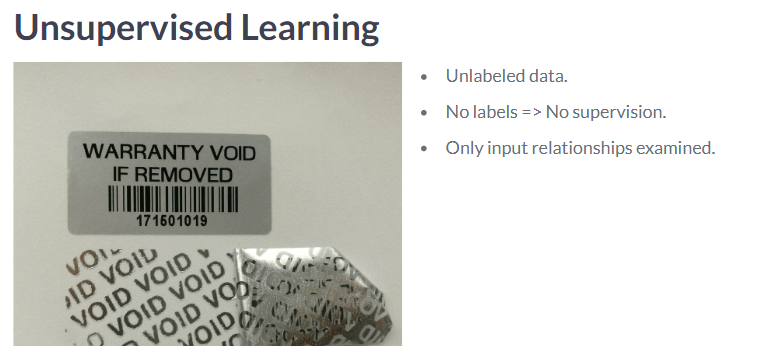
* + Set the degree of the polynomial to 30.

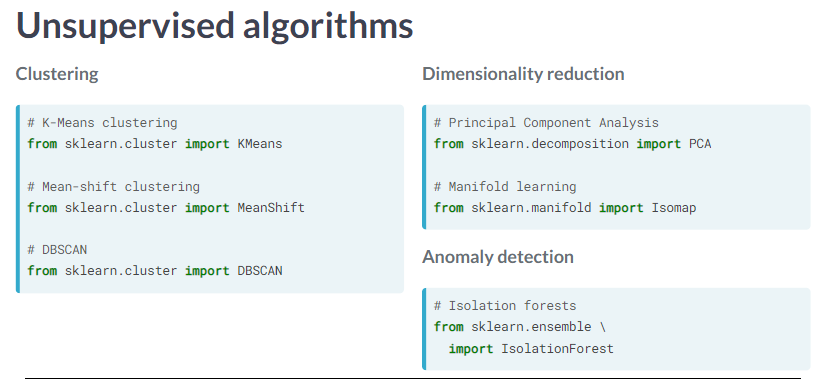
# Set degree to 30

fit\_polynomial(degree=30)

Setting the degree of the polynomial to 1 captured the trend, but was not really impressive. Setting it to 4 looked better, but increasing to 30 did not. Lesson learned: increasing the model complexity beyond the sweet spot just makes it capture more and more noise.







# Guess the flavor I

A model is trained based on a collection of pictures to recognize cats and dogs in pictures, annotated with appropriate labels. Into which domain does this model fall?

##### Answer the question

**50 XP**

##### Possible Answers

* 

This model falls into the domain of **reinforcement learning**.

press1

* 

This model falls into the domain of **unsupervised learning**.

press2

* 

This model falls into the domain of **supervised learning**. **(A)**

press

"Yes! As the model uses labeled data, it falls in the model of supervised learning."

# Guess the flavor II

You want to apply machine learning on a large collection of news articles and cluster them into groups, so that you could identify and count recurring topics, while ignoring the existing news categorization.

Which flavor of Machine Learning algorithms is the most appropriate for this task?

##### Answer the question

**50 XP**

##### Possible Answers

* 

**Reinforcement learning** is the most appropriate approach for this task.

press1

* 

**Unsupervised learning** is the most appropriate approach for this task. **(A)**

press2

* 

**Supervised learning** is the most appropriate approach for this task.

Press

"Yes! As you are trying to cluster data without using labels, unsupervised learning is appropriate for this task."

# Simple digit recognition

In this exercise,you want to build, train and test the simplest digit recognition algorithm.

However simple or complex the Machine Learning problem at hand may be, it will always contain the following steps:

* Data loading, preparation and splitting into the train and test partitions
* Model selection and training ("fitting")
* Model performance assessment

For this purpose, you will load a dataset containing 1797 images of individual digits, 8x8 pixels in size.

Just remember that you use regressors for predicting quantities and classifiers when dealing with categories at the output.

evaluate\_predictions() has been defined for you.

##### Instructions

**100 XP**

* Select the right model for the task at hand: use LinearRegression() if it’s a regression problem and DecisionTreeClassifier() if it’s a classification problem.
* Fit the model using the training inputs X\_train and training labels y\_train.
* Generate predictions on the X\_test DataFrame.
* Evaluate model performance using testing inputs y\_test and testing labels prediction\_results.

# Select the model appropriate for the task

model = DecisionTreeClassifier()

# Train the model

model.fit(X=X\_train , y=y\_train)

# Generate predictions

prediction\_results = model.predict(X=X\_test)

# Test the model

evaluate\_predictions(y\_true=y\_test,

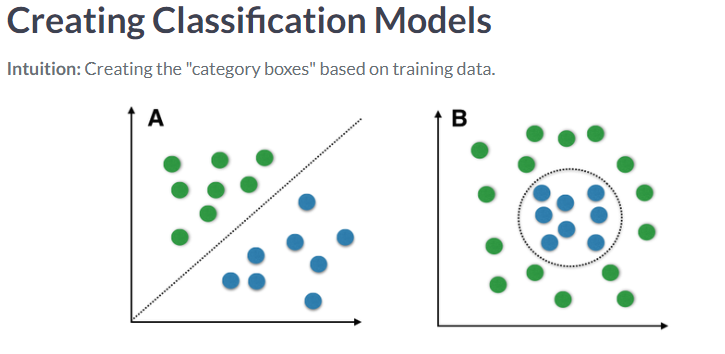
y\_pred=prediction\_results)

<script.py> output:

Accuracy: 0.00 % (Linear regression)

<script.py> output:

Accuracy: 84.01 % (DecisionTreeClassifier)



# Regression or Classification I

Your company has a problem with too many suppliers failing to meet their delivery deadlines, which creates a lot of downstream problems in your production and many angry customers.

Your boss tells you that you have a very rich database of supplier data and asks you to build an algorithm for early prediction of suppliers which are too risky to work with.

What kind of model should you use?

##### Answer the question

**50 XP**

##### Possible Answers

* 

**Regression** should be used to predict risky suppliers.

press1

* 

**Classification** should be used to predict risky suppliers. **(A)**

Press

Exactly. You are trying to find a separation boundary between the domains of low-risk and high-risk suppliers, which is textbook example of a classification problem.

# When regression means classification

OK, so you have a classification problem at your hands. Neat.

Now you talk to your client, which tells you that the algorithm needs to be implemented on their legacy systems, with very poor computing resources.

It turns out you can only run the simplest of models there and you have only two options: LinearRegression() and LogisticRegression().

But you need a classifier and these models have "regression" in their name, can you really pull this off?

If you paid attention in the previous video lesson, you know there's no reason to worry so just go ahead and select the right algorithm for your problem.

##### Instructions

**100 XP**

* Select the right algorithm for your classification problem.

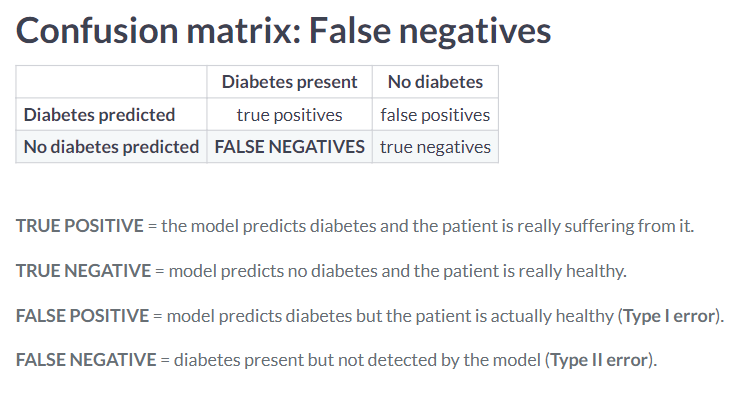
# Select the model

model = LogisticRegression()

# Train the model

model.fit(X\_train, y\_train)

Good job! Not all machine learning algorithms have intuitive names and some names are plain contradictory – Logistic Regression being the best example of that. Luckily this is not the norm, but the one you just saw is usually the most confusing one.



# SPAMtastic!

You have built an e-mail SPAM filtering algorithm.

Which metric is best suited to quantify the percentage of emails your algorithm flags as spam that are actually spam?

##### Answer the question

**50 XP**

##### Possible Answers

* 

Accuracy.

press1

* 

Precision. **(A)**

press2

* 

Recall.

Press

Correct! Still, you should examine recall also, in order to get a sense of how many real spam emails are actually flagged as such.

# Hold-out

You already know about the danger of overfitting, which occurs when your model learns the training data too well, but then performs poorly when faced with new data.

Because of that, you've been urged to always test your model using data that wasn't previously used for training.

But don't take our word for it, see for yourself!

You will use a dataset consisting of two classes. 60% of the data has been selected for training and stored in X\_train and y\_train. The remaining 40% is stored in variables X\_test and y\_test.

You will train a RandomForestClassifier() model and see the difference in performance:

* when it's applied on the very same data used to train it
* when it's applied on data just slightly different from the training set

##### Instructions 1/2

**50 XP**

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Test the model on the same data it used for training.

# Model setup

model = RandomForestClassifier(n\_estimators=5, max\_depth=20)

# Model fitting

model.fit(X\_train, y\_train)

# Testing on training data

test\_and\_show\_accuracy(model,

X\_test=X\_train,

y\_test=y\_train)

<script.py> output:

Accuracy: 96.67 %

##### Instructions 2/2

**50 XP**

* [2](javascript:void(0))
  + Test the model on the hold-out dataset, that is, the data the model hasn't seen during training.

# Model setup

model = RandomForestClassifier(n\_estimators=5, max\_depth=20)

# Model fitting

model.fit(X\_train, y\_train)

# Testing on testing data

test\_and\_show\_accuracy(model,

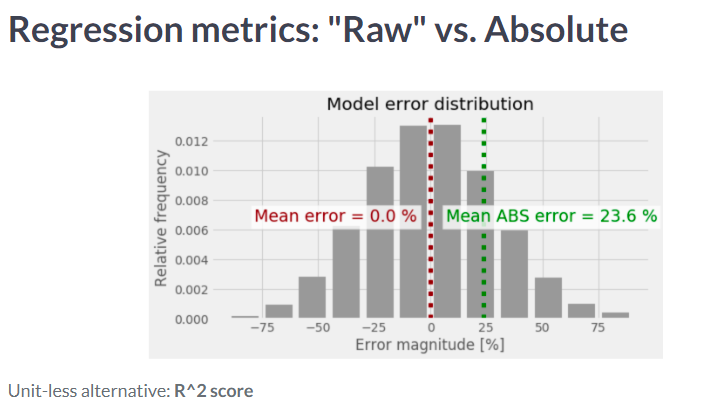
X\_test=X\_test,

y\_test=y\_test)

<script.py> output:

Accuracy: 81.33 %

Whoah! So when evaluating on training data, the accuracy reaches 97.1%, but on unseen data it drops to 79%? This is almost a 20% difference, something you should definitely realize and solve before putting your model in production!



# Tailor made

You need to estimate the goodness-of-fit of your regression model to your data in a relative, unit-less manner. Which metric should be your first choice?

##### Answer the question

**50 XP**

##### Possible Answers

* 

The **mean absolute error**.

press1

* 

The **median absolute error**.

press2

* 

The **R^2 score**. **(A)**

Press

Correct! Using R^2 score is a nice and simple measure of goodness-of-fit which all Data Scientists should understand and use.

# Going non-linear

You're about to see how it's possible to model non-linear relationships using linear regression models.

To make this "transition", you don't have to change anything in the model, you just have to generate:

* higher order features: (a)→(a,a2,a3,...)(a)→(a,a2,a3,...), and
* interaction features: (a,b)→(a∗b,a2∗b,a∗b2,...)(a,b)→(a∗b,a2∗b,a∗b2,...)

You will first try to fit a purely linear model to a quadratic process and check the R^2 score.

After that you'll use the function PolynomialFeatures() to generate, well, polynomial features and see how much better your fit is -- both visually and according to the R^2 score.

Finally, you've been provided with the custom function check\_model\_fit() that plots the model predictions against actual data and prints the R^2 score of your model.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Select the appropriate method to call the model training procedure.

# Select the appropriate method to call the model training procedure

linear\_model.fit(X=x\_train, y=y\_train)

# Check the "goodness-of-fit" of the fitted model

check\_model\_fit(model=linear\_model,

x=x\_train,

y=y\_train)

<script.py> output:

R^2 score: 81.22 %

##### Instructions 2/2

**50 XP**

* [2](javascript:void(0))
* Using the original dataset x\_train, create a new dataset x2\_train with 2nd degree polynomial features.
* Fit the linear\_model using the newly created dataset.
* Check the goodness of fit of your fitted linear model.

# Create a new dataset x2\_train with 2nd degree polynomial features.

poly = PolynomialFeatures(degree=2)

x2\_train = poly.fit\_transform(x\_train)

# Fit the linear model using the newly created dataset

linear\_model.fit(X=x2\_train, y=y\_train)

# Check the "goodness-of-fit" of the fitted model

check\_model\_fit(model=linear\_model,

x=x2\_train,

y=y\_train)

<script.py> output:

R^2 score: 93.13 %

Amazing! A linear model that perfectly follows a quadratic process, that is the magic trick with polynomial features! And look how easy it is to produce them using scikit-learn's standard functions. Still, always make sure to understand the underlying principles thoroughly!

