Machine Learning workshop

Prof. Daniele Bonacorsi

ISGC 2021

March 22, 2021

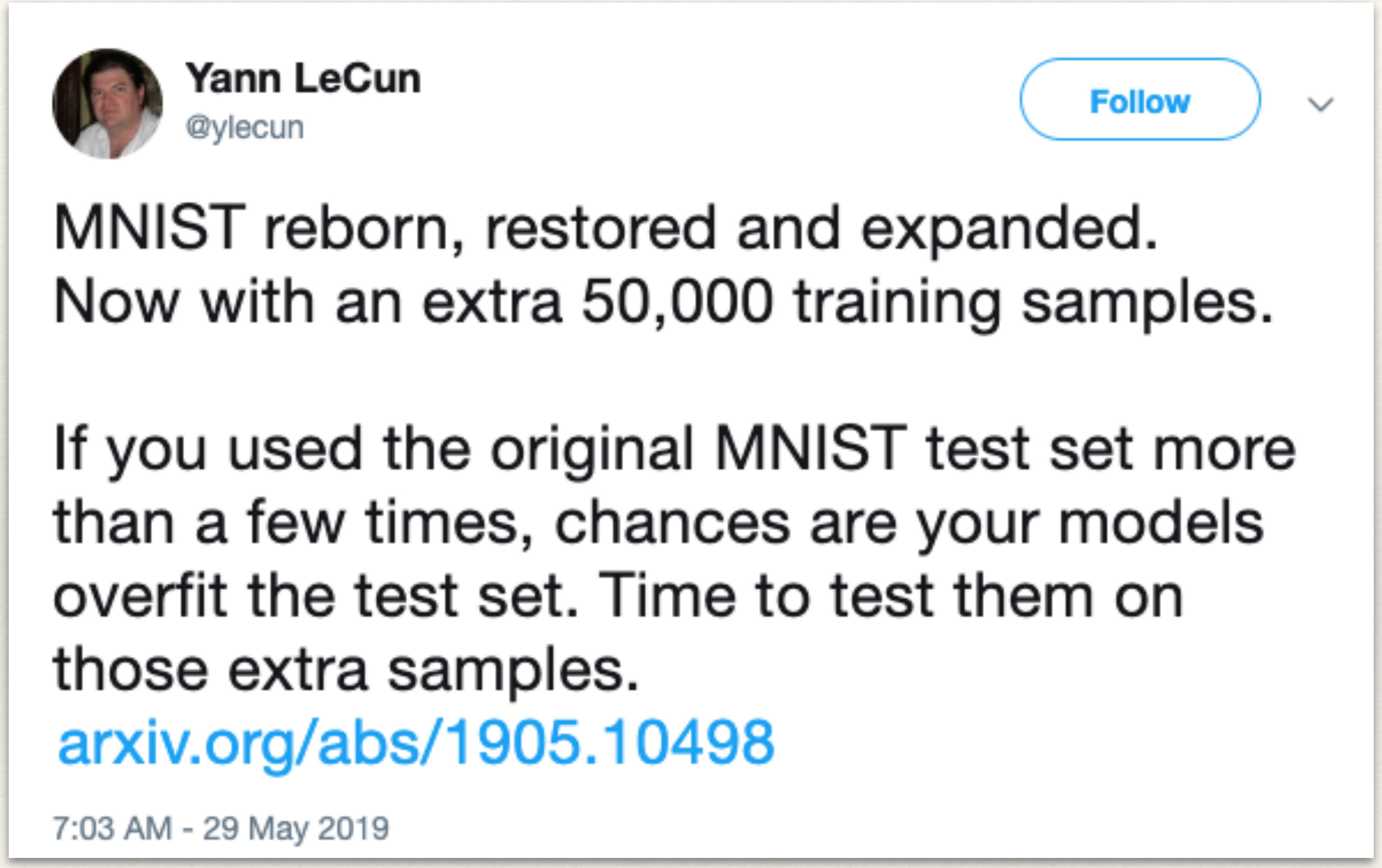
Lab on **Classification**

[credits to: A. Geron, "Hands-On Machine Learning With Scikit-Learn and Tensorflow"]

MNIST

The **MNIST** dataset is a set of 70k images of handwritten digits

- Each image is **labeled** with the digit it represents (i.e. like "this is a 3")
- 784 **features**: 28x28 pixels each, each features represent one pixel's intensity, from 0 (white) to 255 (black).
- one of the most famous "hello world" in ML → **multi-class classification**



Yann LeCun
@ylecun

MNIST reborn, restored and expanded.
Now with an extra 50,000 training samples.

If you used the original MNIST test set more than a few times, chances are your models overfit the test set. Time to test them on those extra samples.
arxiv.org/abs/1905.10498

7:03 AM - 29 May 2019

Set up, import the data, inspect (briefly) the data, perform the train-test split.

Practice **C1**

Practice **C2**

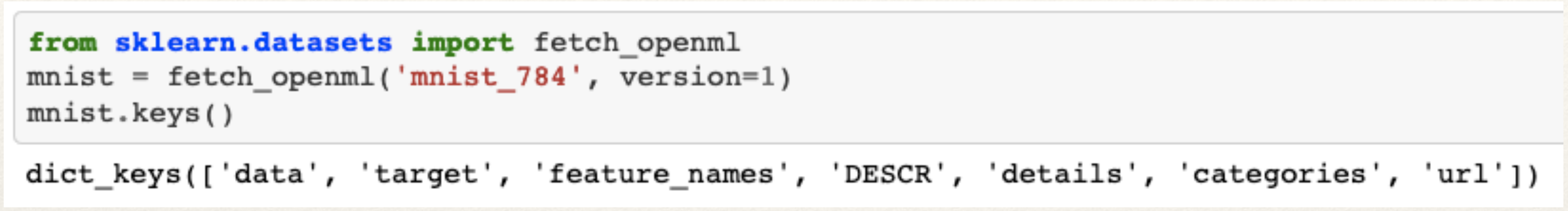
Practice **C3**

5 minutes

Time to code!

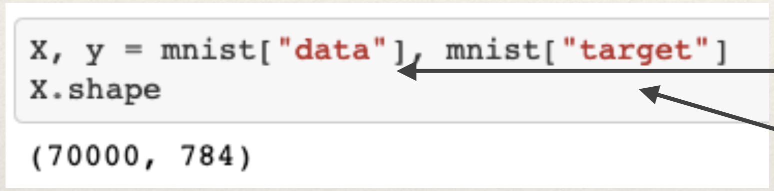
Get the data

Get it from sklearn:



```python
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details', 'categories', 'url'])
```

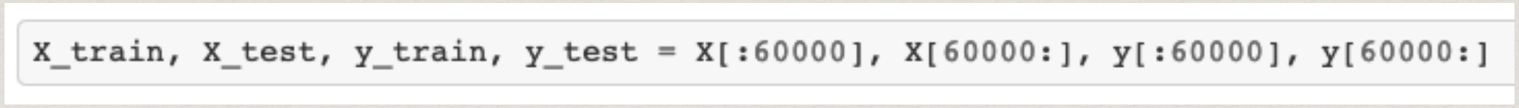


```python
X, y = mnist["data"], mnist["target"]
X.shape
```

```
(70000, 784)
```

A **data** key containing an array with one row per instance and one column per feature

A **target** key containing an array with the labels



```python
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

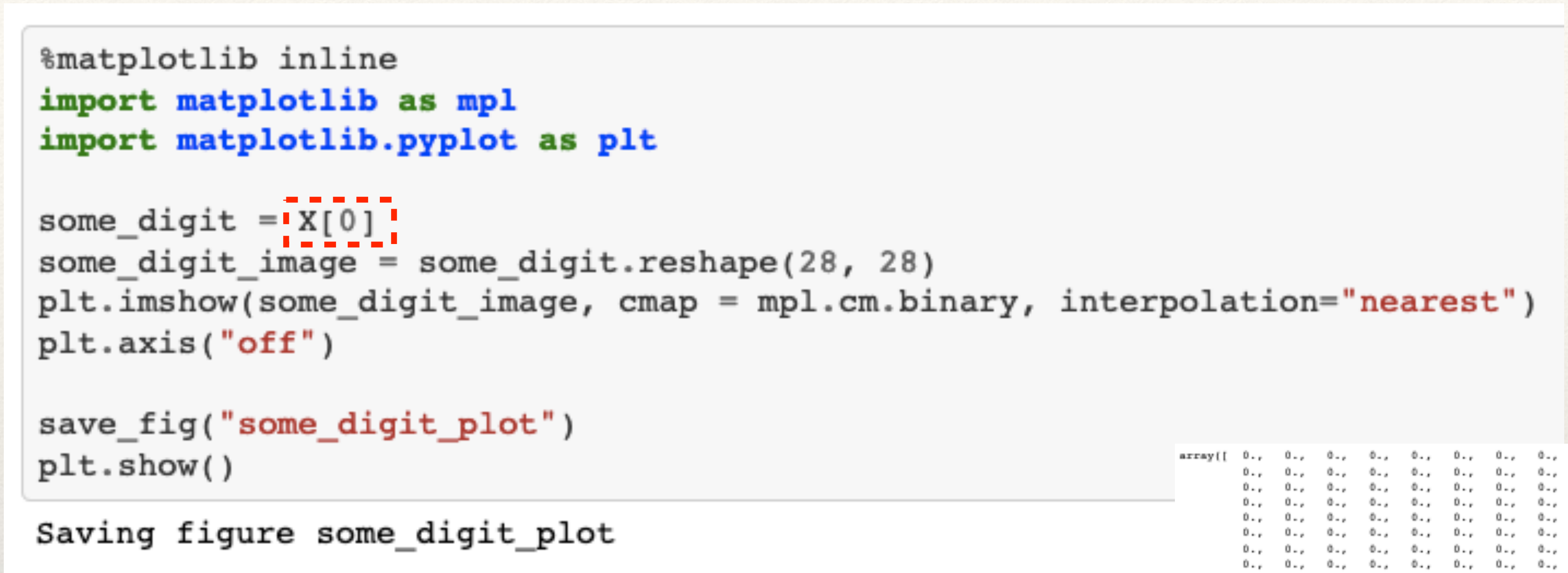
The dataset is **split into training + test**..

- 60k training, 10k test

.. and it is already **shuffled**, so all CV folds will be similar

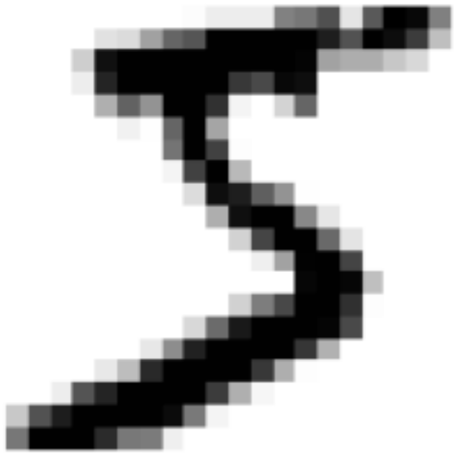
- you don't want one fold to be missing some digits

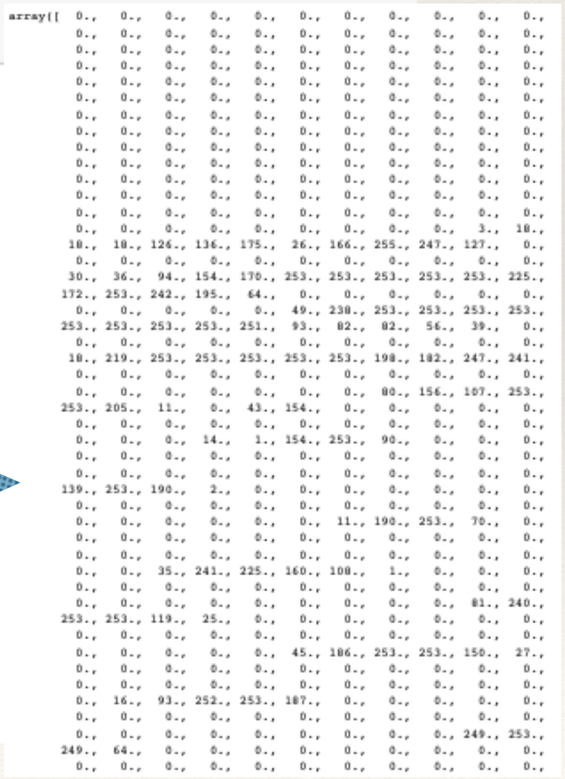
Inspect the data



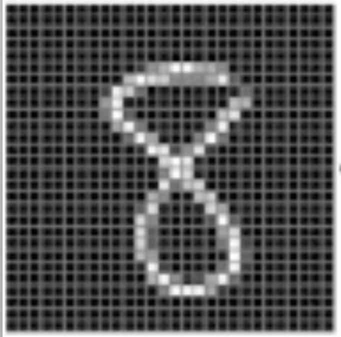
```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = mpl.cm.binary, interpolation="nearest")
plt.axis("off")

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

```
Saving figure some_digit_plot
```



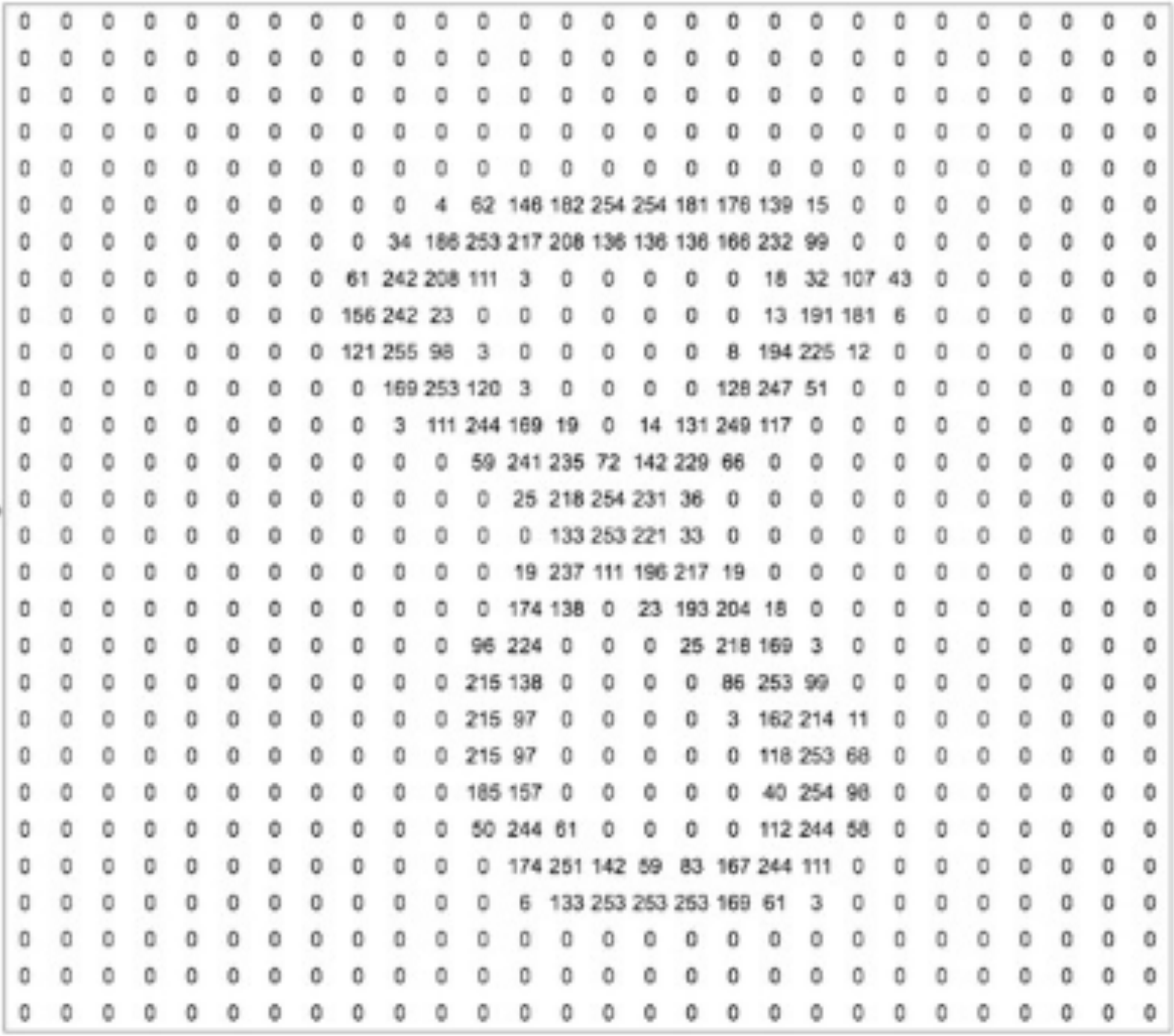


```
array([   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   3.,  18.,
         18.,  18., 126., 136., 175.,  26., 166., 255., 247., 127.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
         30.,  36.,  94., 154., 170., 253., 253., 253., 253., 253., 225.,
        172., 253., 242., 195.,  64.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,  49., 238., 253., 253., 253., 253.,
        253., 253., 253., 253., 251.,  93.,  82.,  82.,  56.,  39.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
         18., 219., 253., 253., 253., 253., 253., 198., 182., 247., 241.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,  80., 156., 107., 253.,
        253., 205.,  11.,   0.,  43., 154.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,  14.,   1., 154., 253.,  90.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
        139., 253., 190.,   2.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,  11., 190., 253.,  70.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,  35., 241., 225., 160., 108.,   1.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,  81., 240.,
        253., 253., 119.,  25.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,  45., 186., 253., 253., 150.,  27.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,  16.,  93., 252., 253., 187.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 249., 253.,
        249.,  64.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
```



28 x 28

784 pixels



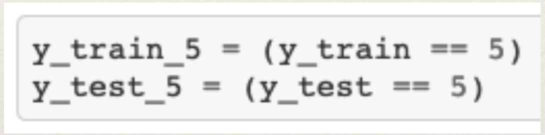
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	4	62	146	182	254	254	181	176	139	15	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	34	186	253	217	208	136	136	136	166	232	99	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	61	242	208	111	3	0	0	0	0	0	18	32	107	43	0	0	0	0	0	0
0	0	0	0	0	0	0	0	156	242	23	0	0	0	0	0	0	0	13	191	181	6	0	0	0	0	0	0
0	0	0	0	0	0	0	0	121	255	98	3	0	0	0	0	0	8	194	225	12	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	169	253	120	3	0	0	0	0	128	247	51	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	3	111	244	169	19	0	14	131	249	117	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	59	241	235	72	142	229	66	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	25	218	254	231	36	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	133	253	221	33	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	19	237	111	196	217	19	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	174	138	0	23	193	204	18	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	96	224	0	0	0	25	218	169	3	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	138	0	0	0	0	86	253	99	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	97	0	0	0	0	3	162	214	11	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	97	0	0	0	0	0	118	253	68	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	185	157	0	0	0	0	0	40	254	98	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	50	244	61	0	0	0	0	112	244	58	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	174	251	142	59	83	167	244	111	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	6	133	253	253	253	169	61	3	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Train a **binary** classifier

Simplify and build a model that works e.g. as a "**5-detector**"

- capable of distinguishing between just two classes, "5" and "not-5"

Create the label vectors (train and test sets) for this task:

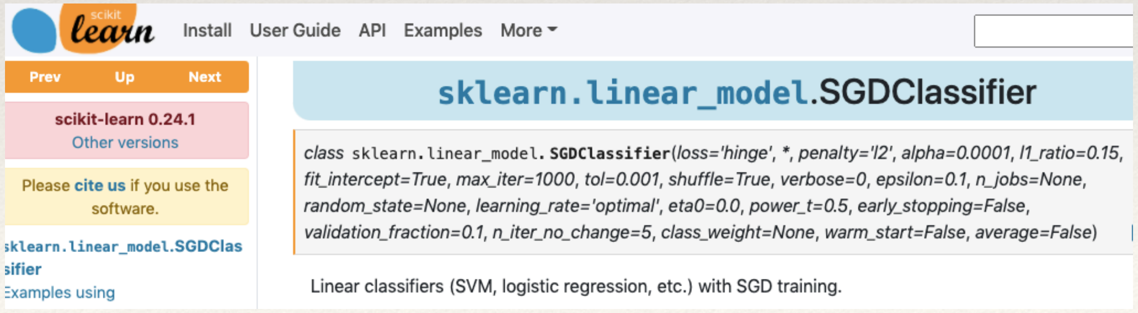


```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

Then, pick a classifier. An interesting choice is the **SGD classifier**

- capable of handling very large datasets efficiently (it deals with training instances independently, one at a time - which also makes SGD well suited for online learning)

Train and predict is easy..



sklearn.linear_model.SGDClassifier

class `sklearn.linear_model.`**`SGDClassifier`**(*loss='hinge', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, n_jobs=None, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, class_weight=None, warm_start=False, average=False*)

Linear classifiers (SVM, logistic regression, etc.) with SGD training.

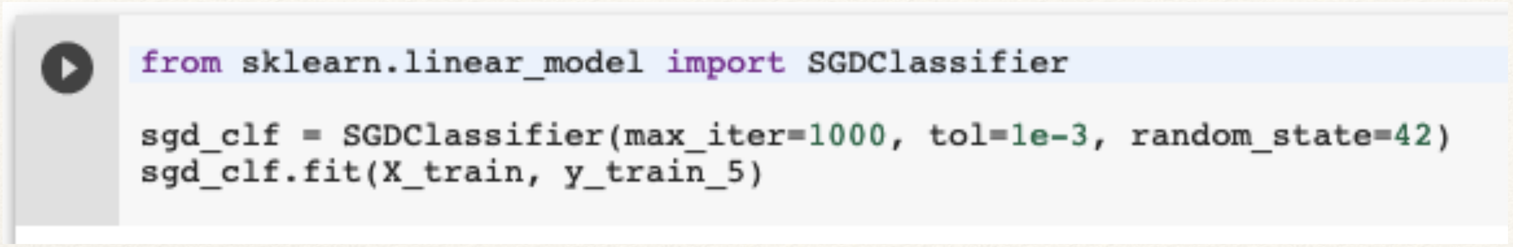
Implement a 5-detector.

Practice **C4**

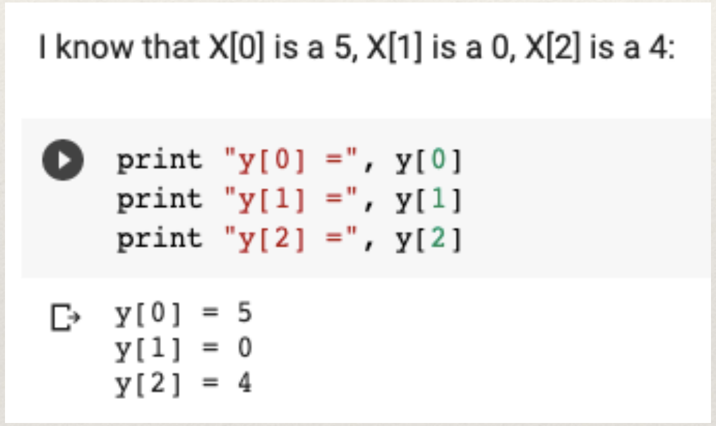
5 minutes

Time to code!

Train a **binary** classifier



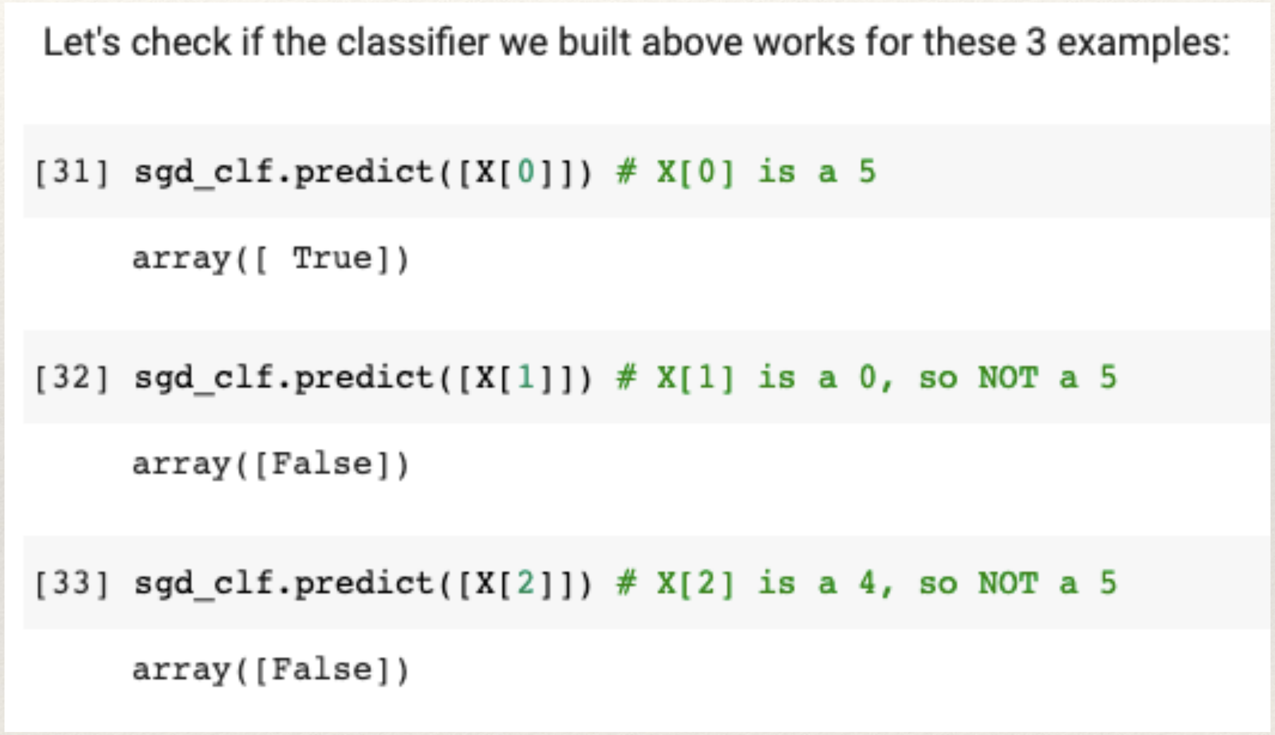
```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```



I know that X[0] is a 5, X[1] is a 0, X[2] is a 4:

```
print "y[0] =", y[0]
print "y[1] =", y[1]
print "y[2] =", y[2]
```

```
y[0] = 5
y[1] = 0
y[2] = 4
```



Let's check if the classifier we built above works for these 3 examples:

```
[31] sgd_clf.predict([X[0]]) # X[0] is a 5

     array([ True])

[32] sgd_clf.predict([X[1]]) # X[1] is a 0, so NOT a 5

     array([False])

[33] sgd_clf.predict([X[2]]) # X[2] is a 4, so NOT a 5

     array([False])
```

OK, it seems to work.. which is the performance of this model?

Compute the accuracy

- hint: use cross_val_score() function in sklearn to evaluate your SGDClassifier model using k-fold cross-validation, with k=3

Practice C5

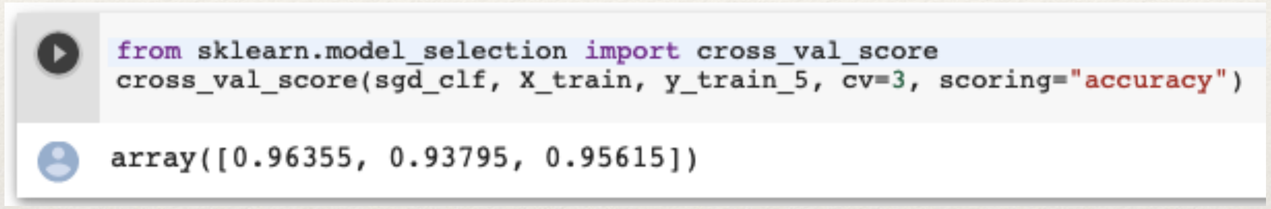
2 minutes

Time to code!

Measuring performance (accuracy) using CV

Use cross_val_score() function in sklearn to evaluate your SGDClassifier model using k-fold cross-validation, with k=3

- i.e. make k trainings: split the training set into k folds, train and make predictions and evaluate them on each fold using a model trained on the remaining folds



```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.96355, 0.93795, 0.95615])
```

What!? 93-96% accuracy at first attempt!? Mmh..

- think at a very dumb classifier that just classifies every single image as if it belonged to the “not-5” class: it will have 90% accuracy! (if enough data, only about 10% of the images are 5s, so if you always guess that an image is a “not-5”, you will be right roughly 90% of the time, by construction!

Accuracy is <u>not</u> the preferred performance measure for classifiers

- even worse if you are dealing with **skewed datasets** (i.e. when some classes are much more frequent than others).

Extract the **confusion matrix**.

Practice **C6** | 2 minutes | Time to code!

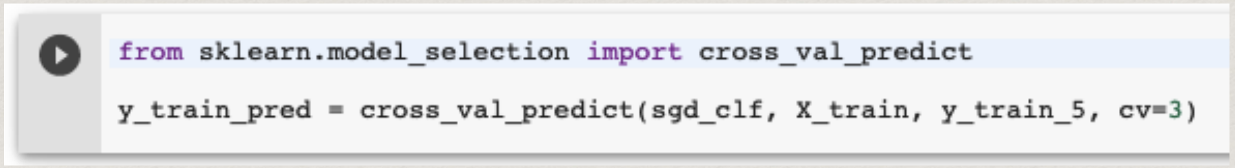
Confusion matrix

To evaluate the performance of a classifier, build the **confusion matrix**

- count misclassifications: e.g. how many times the classifier **confused** images of 5s with 3s? look in the 5th row and 3rd column of the **confusion** matrix

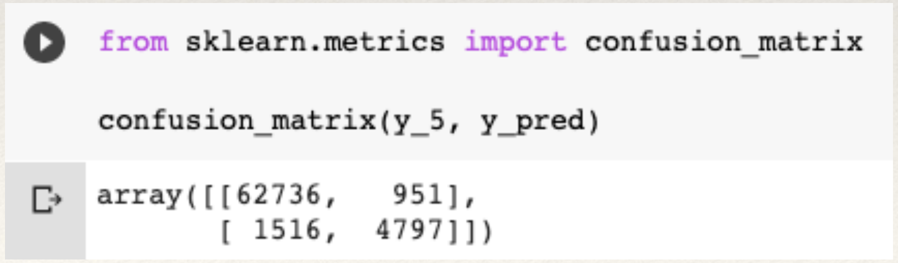
Use cross_val_predict() and confusion_matrix()

- cross_val_predict() is similar to cross_val_score(): performs K-fold CV but returns not the evaluation score, but the predictions made on each fold
- then, give the target classes (y_train_5) and the predicted classes (y_train_pred) to confusion_matrix()



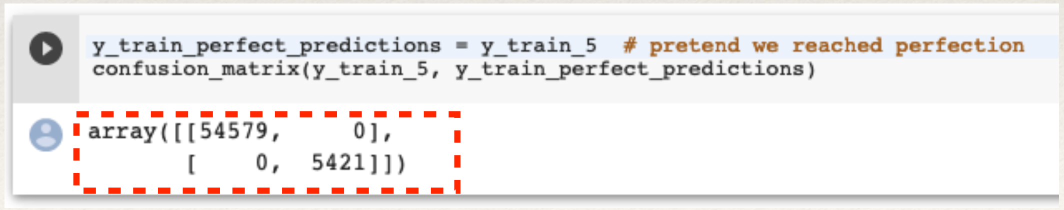
```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

To clarify, perfection will look like this:



```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_5, y_pred)
```

```
array([[62736,   951],
       [ 1516,  4797]])
```

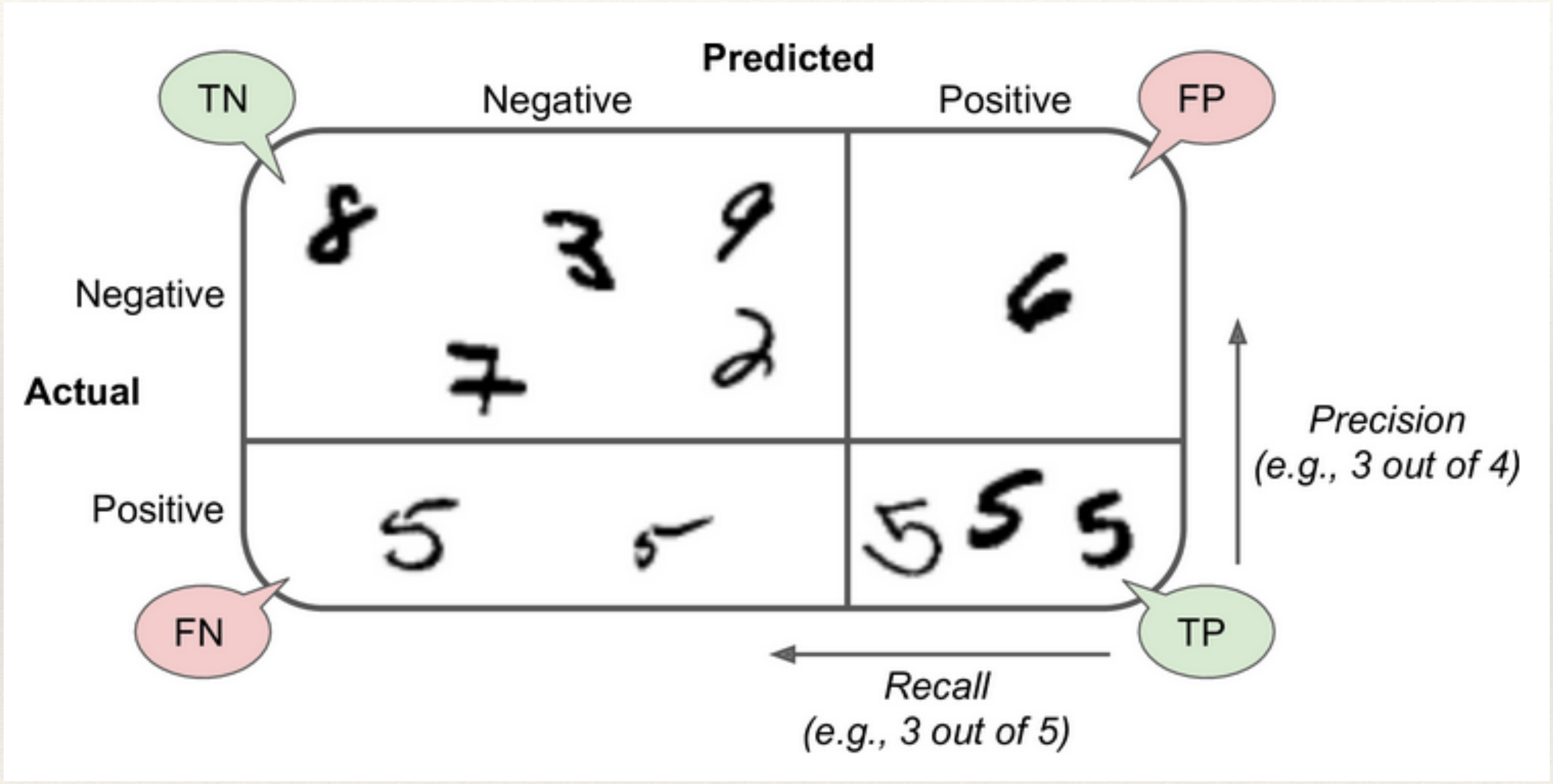


```
y_train_perfect_predictions = y_train_5  # pretend we reached perfection
confusion_matrix(y_train_5, y_train_perfect_predictions)
```

```
array([[54579,     0],
       [    0,  5421]])
```

16 D. Bonacorsi

Confusion matrix



Precision / Recall

$$\begin{array}{c|cc}
\text{actual} \rightarrow / \text{predicted} \downarrow & 1 & 0 \\
\hline
1 & TP & FP \\
0 & FN & TN
\end{array}$$

PRECISION

"Among all patients predicted to have cancer, how many actually have it?"

→ how "precise" have you been?

$$\Rightarrow \frac{\#\ TP}{\underbrace{\#\ \text{predicted } P}_{= TP+FP}} = \frac{TP}{TP+FP}$$

better if this is high!

Precision / Recall

predicted ↓ \ actual →	1	0
1	TP	FP
0	FN	TN

PRECISION

"Among all patients predicted to have cancer, how many actually have it?"

$$\stackrel{def}{=} \frac{TP}{\underbrace{TP+FP}_{\text{predicted P}}} \quad \text{(should be as high as possible!)}$$

RECALL

"Among all patients that actually have cancer, how many did we predict to have it?"

→ "how many of the TP were "recalled" (found)?"

$$\Rightarrow \frac{\#\ TP}{\#\ \text{actual P}} = \frac{TP}{TP+FN}$$

better if this is high!

Precision / Recall

$$
\begin{array}{c|cc}
\text{predicted} \downarrow \; \text{actual} \rightarrow & 1 & 0 \\
\hline
1 & TP & FP \\
0 & FN & TN
\end{array}
$$

PRECISION

"Among all patients predicted to have cancer, how many actually have it?"

$$\overset{\text{def}}{=} \frac{TP}{\underbrace{TP+FP}_{\text{predicted P}}} \quad \text{(should be as high as possible!)}$$

RECALL

"Among all patients that actually have cancer, how many did we predict to have it?"

$$\overset{\text{def}}{=} \frac{TP}{\underbrace{TP+FN}_{\text{actual P}}} \quad \text{(should be as high as possible!)}$$

Precision / Recall

$$\text{PRECISION} \overset{\text{def}}{=} \frac{TP}{TP+FP}$$

Intuitively: the **precision** is the ability of the classifier not to label as positive a sample that is negative.

$$\text{RECALL} \overset{\text{def}}{=} \frac{TP}{TP+FN}$$

Intuitively: the **recall** is the ability of the classifier to find all the positive samples.

actual → / predicted ↓	1	0
1	TP	FP
0	FN	TN

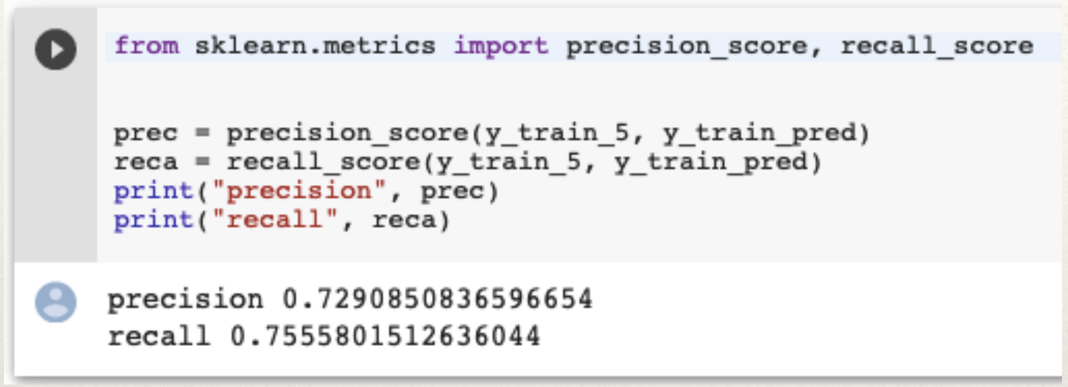
Example :

classifier that predicts $y=0$ always :

$$\Rightarrow \quad TP = 0 \quad \Rightarrow \quad \begin{array}{l} \text{PRECISION} = 0 \\ \text{RECALL} = 0 \end{array}$$

Precision, Recall, F1 score

Abandon accuracy, and compute **precision** and **recall**:



```
from sklearn.metrics import precision_score, recall_score

prec = precision_score(y_train_5, y_train_pred)
reca = recall_score(y_train_5, y_train_pred)
print("precision", prec)
print("recall", reca)
```

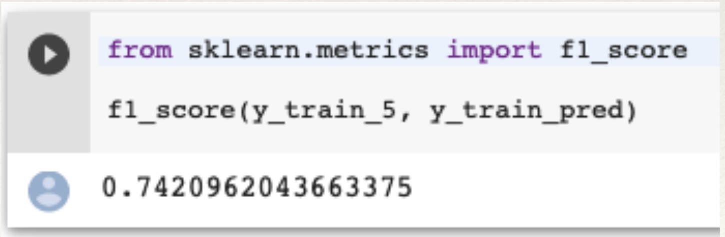
```
precision 0.7290850836596654
recall 0.7555801512636044
```

My 5-detector does not look as shiny as it did when I looked at its accuracy only…

- when it claims an image represents a 5, it is correct only 72.9% of the time
- and it detects only 75.6% of the 5s

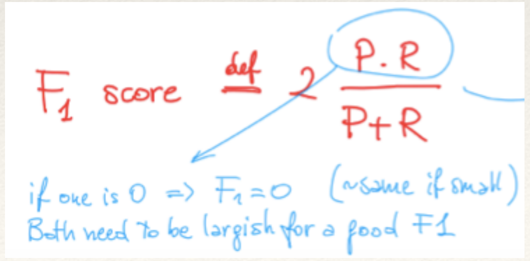
Convenient to combine them into a single metric: the **F1 score**

- **harmonic mean** of precision and recall: wrt regular mean, the harmonic mean does not treat all values equally, but gives much more weight to low values. As a result, the classifier will only get a **high F1 score if both recall and precision are high**
- additionally, good to have just one performance metric (if I need to compare 2 classifiers)



```
from sklearn.metrics import f1_score

f1_score(y_train_5, y_train_pred)
```

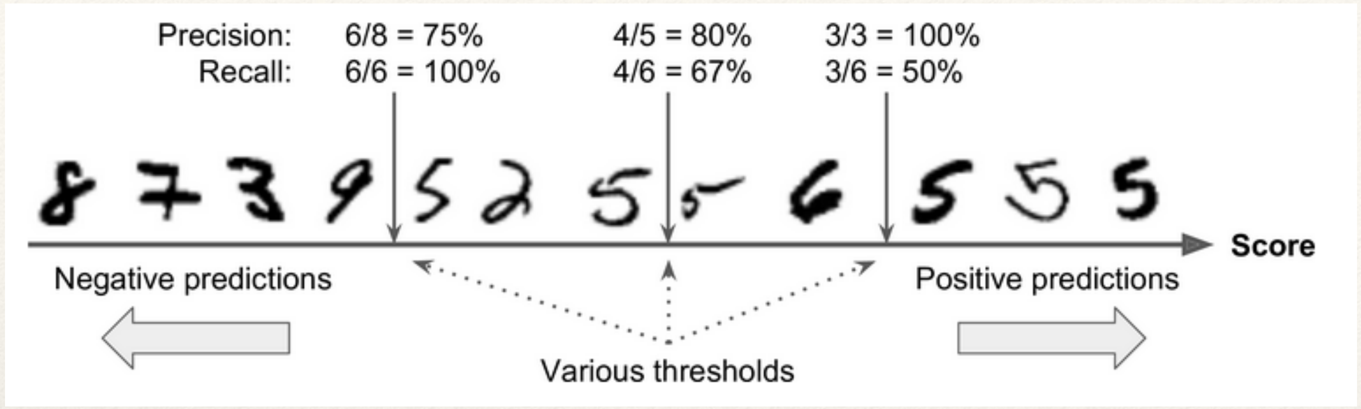
```
0.7420962043663375
```



$$F_1 \text{ score} \overset{def}{=} 2\,\frac{P \cdot R}{P+R}$$

if one is 0 => $F_1 = 0$ (~same if small)
Both need to be largish for a good F1

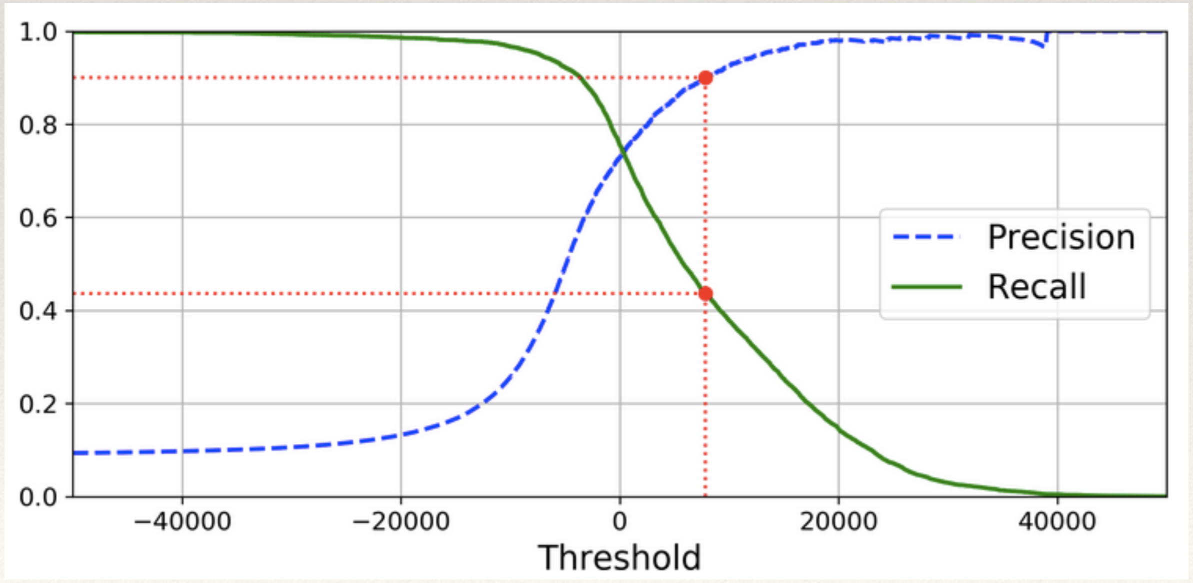
Precision/Recall trade-off



SGDClassifier, for each instance, computes a score based on a decision function, and if that score is greater/smaller than a threshold, it assigns the instance to the positive/negative class

Looking at various thresholds, it is evident that when precision increases then recall reduces, and vice versa. This is called the **precision/recall tradeoff**

"How do I choose the threshold?".

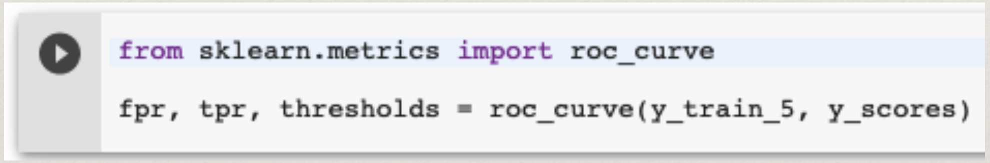


Receiver Operating Characteristic (ROC)

The **Receiver Operating Characteristic** (**ROC**) curve is another very common tool used with binary classifier

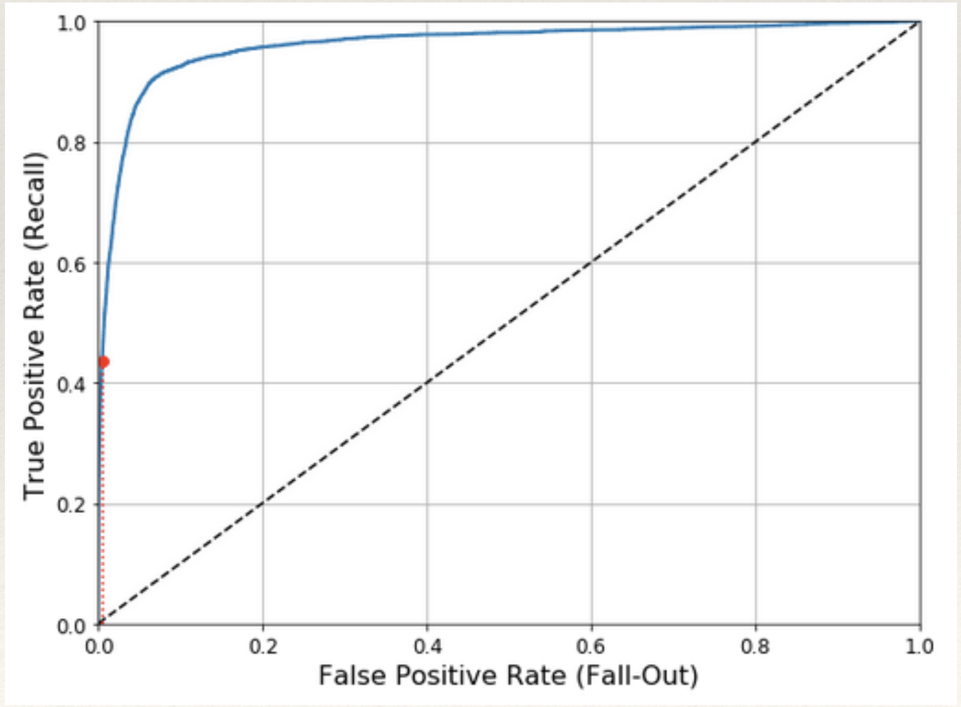
It is very similar to the precision/recall curve, but:

- it plots the **TPR** (= recall) against the **FPR** (FPR = ratio of negative instances that are incorrectly classified as positive), which is FPR=1-**TNR** (TNR = ratio of negative instances that are correctly classified as negative - also called **specificity**). In other words, the ROC curve plots sensitivity (recall) versus 1 – specificity.



```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

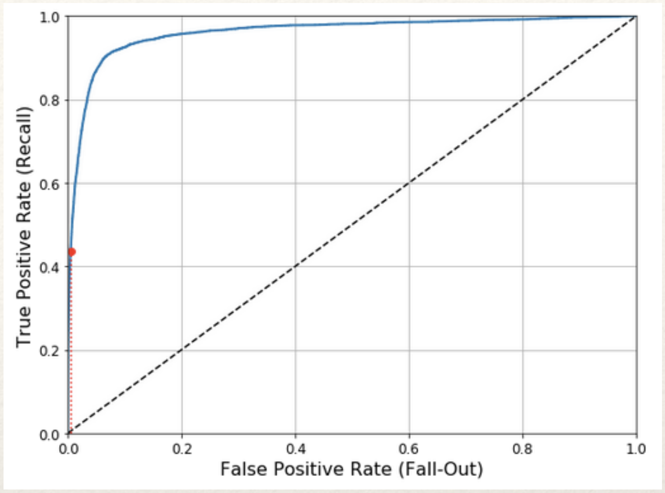
.. and then plot:



Area Under the Curve (AUC)

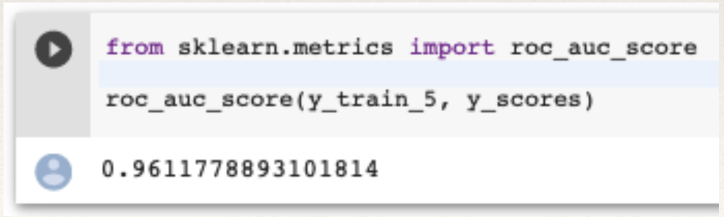
Observations on the ROC:

- the higher (lower) the TPR, the more (fewer) false positives FPR the classifier produces
- the dotted line represents the ROC curve of a purely random classifier
- a good classifier stays as far away from that line as possible, toward the top-left corner



To compare classifiers you need a number: this could be then the **Area Under the** ROC **Curve** (AUC)

- a perfect classifier will have AUC = 1
- a purely random classifier will have AUC = 0.5.



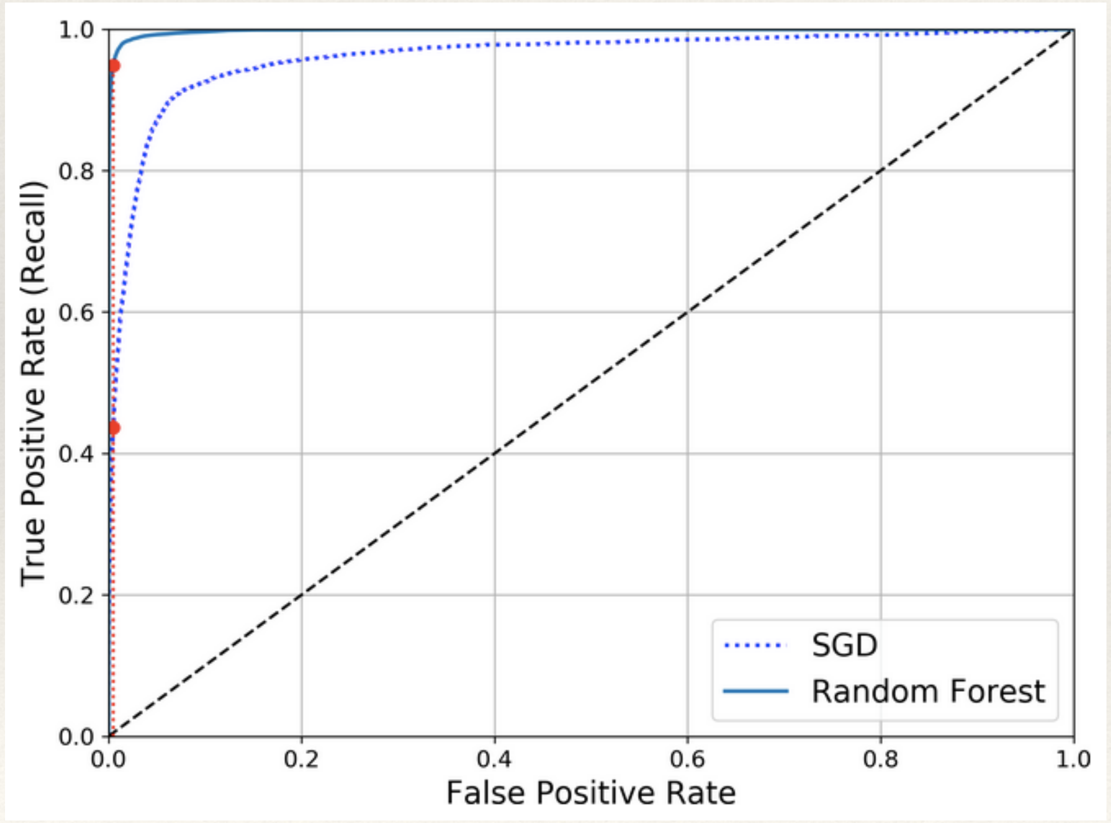
```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

```
0.9611778893101814
```

Model comparison using AUC

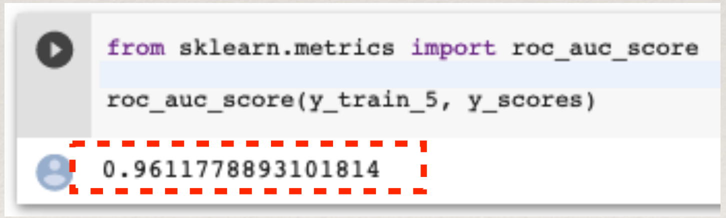
Use ROC+AUC as performance metrics. Get them for all models, and you can compare them.

- e.g. (not in the notebook) if one trains a **RandomForestClassifier** and compare its ROC curve and ROC AUC score to the **SGDClassifier**



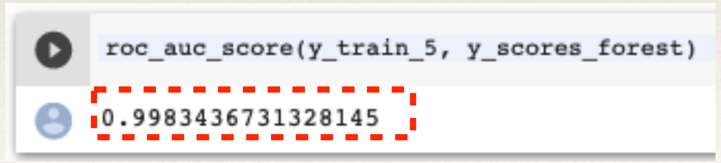
RandomForestClassifier's ROC curve looks much better than the SGDClassifier's. AUC scores also show this (below)

SGDClassifier



```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
0.9611778893101814
```

RandomForestClassifier



```
roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

MNIST recap so far

Now you recapped a bit how to:

- train a **binary classifier**
- choose the appropriate metric for your task
- evaluate your classifiers using CV
- select the precision/recall tradeoff that fits your needs, and compare various models using ROC curves and ROC AUC scores

Now let’s try to detect more than just the 5s…

That’s it,
for our Lab on **Classification**