INFN Machine Learning course

Prof. Amir Farbin, Prof. Daniele Bonacorsi

20-22 May 2019

Camogli, Italy

e2e ML project: classification

https://colab.research.google.com/github/afarbin/INFN-ML-Course/blob/master/notebooks/Regression-workflow-sklearn.ipynb

[credits: 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" of ML → **multi-class classification**

Get the data

Get it from sklearn:

```
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'])
```

```
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

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

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

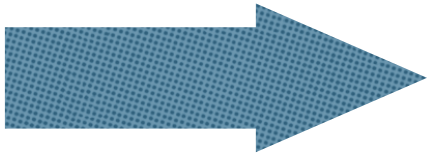
- 60k training, 10k test

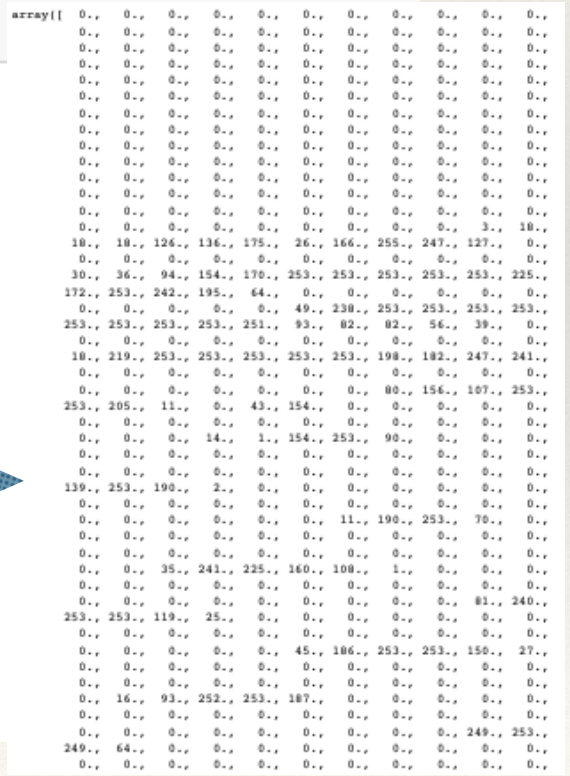
.. and it is already **shuffled**, so all CV folds will be similar

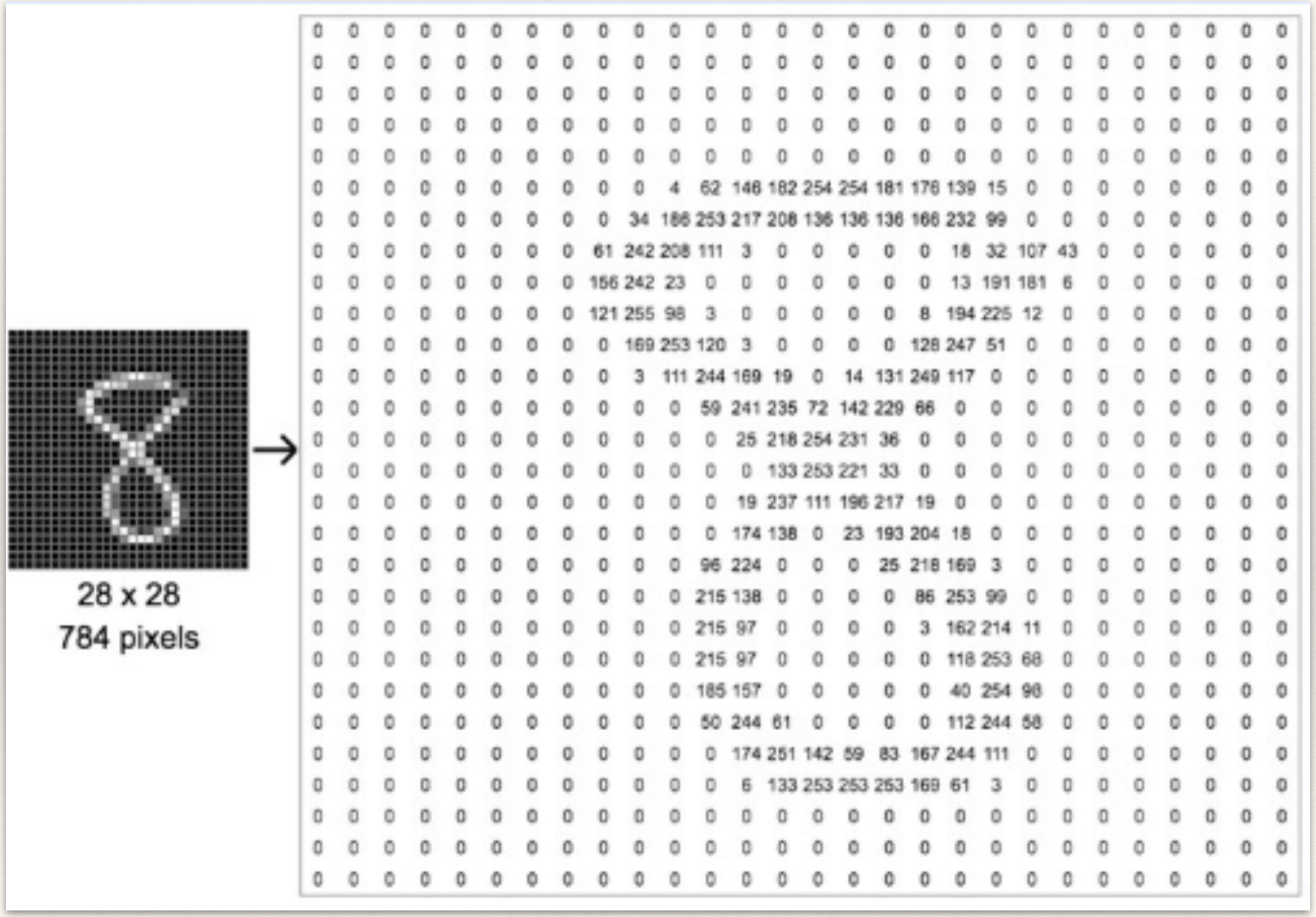
- you don't want one fold to be missing some digits

```
%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
```






28 x 28
784 pixels

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 **Stochastic Gradient Descent** (**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.. which is the performance of this model?

```
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)
```

```
sgd_clf.predict([some_digit])
```

```
array([ True])
```

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).

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)
```

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_train_5, y_train_pred)
```

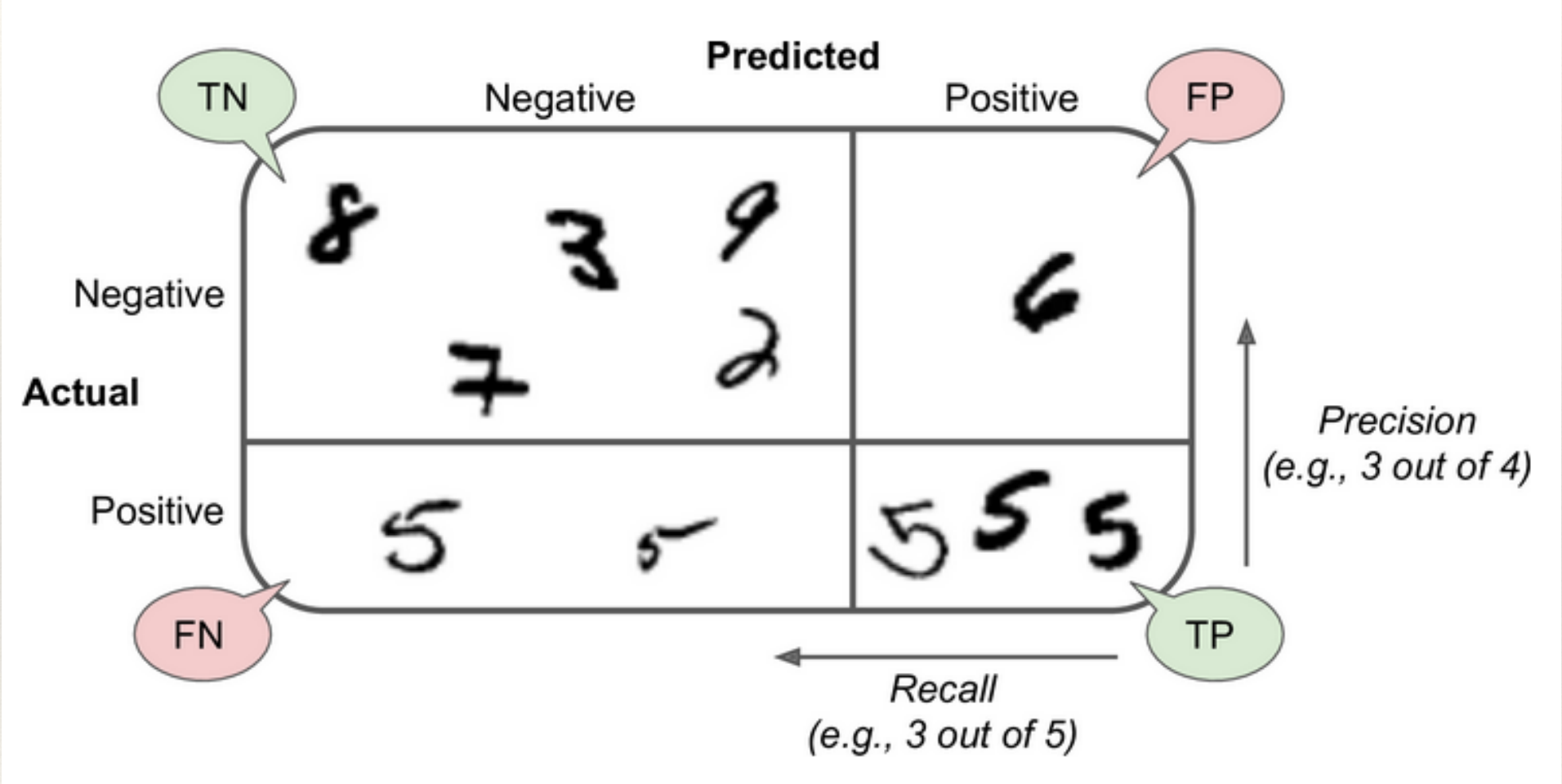
```
array([[53057,  1522],
       [ 1325,  4096]])
```

To clarify, perfection will look like this:

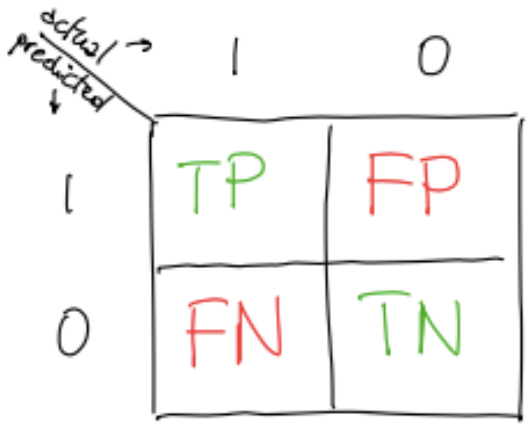
```
y_train_perfect_predictions = y_train_5  # pretend we reached perfection
confusion_matrix(y_train_5, y_train_perfect_predictions)
```

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

Confusion matrix



Precision / Recall



PRECISION

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

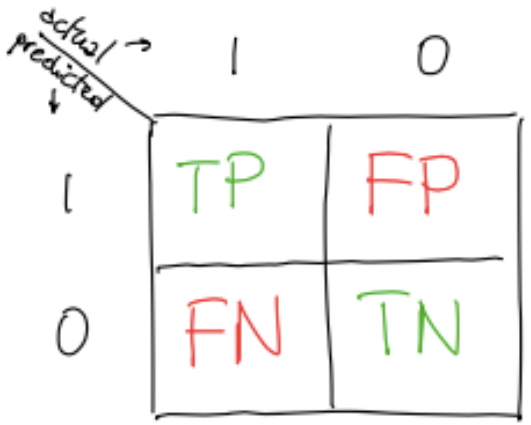
→ how "precise" have you been ?

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

$TP+FP$

better if this is high !

Precision / Recall



PRECISION

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

$$\text{def} = \frac{TP}{TP+FP}$$] predicted P

(should be as high as possible !)

RECALL

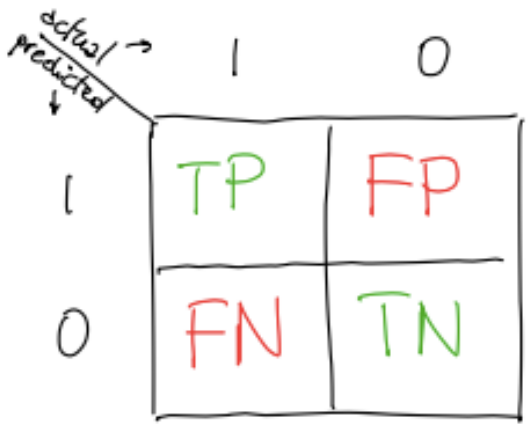
"Among all patients that actually have cancer, how many did we predict to have it ?"

→ "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



PRECISION

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

$$\text{def} = \frac{TP}{TP+FP}$$] predicted P

(should be as high as possible !)

RECALL

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

$$\text{def} = \frac{TP}{TP+FN}$$] actual P

(should be as high as possible !)

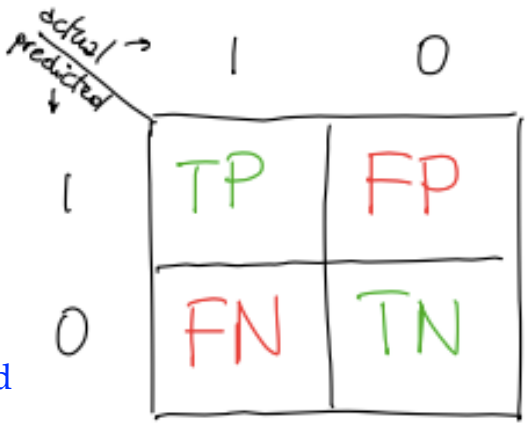
Precision / Recall

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

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

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

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



Example :

classifier that predicts $y=0$ always :

$\Rightarrow$ $TP = 0$ $\Rightarrow$ PRECISION $= 0$

RECALL $= 0$

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

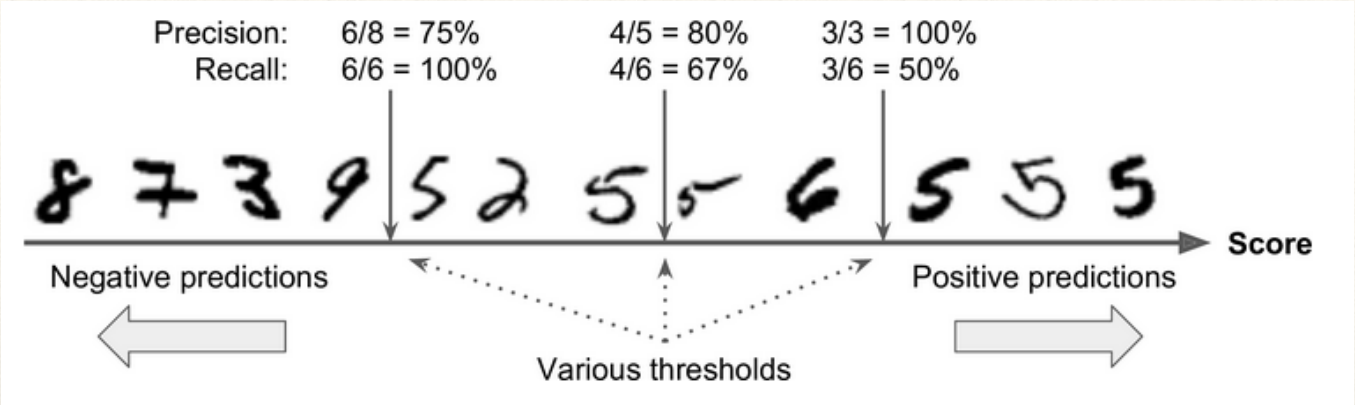
Convenient to combine them into a single metric: the **F1 score**

- **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
```

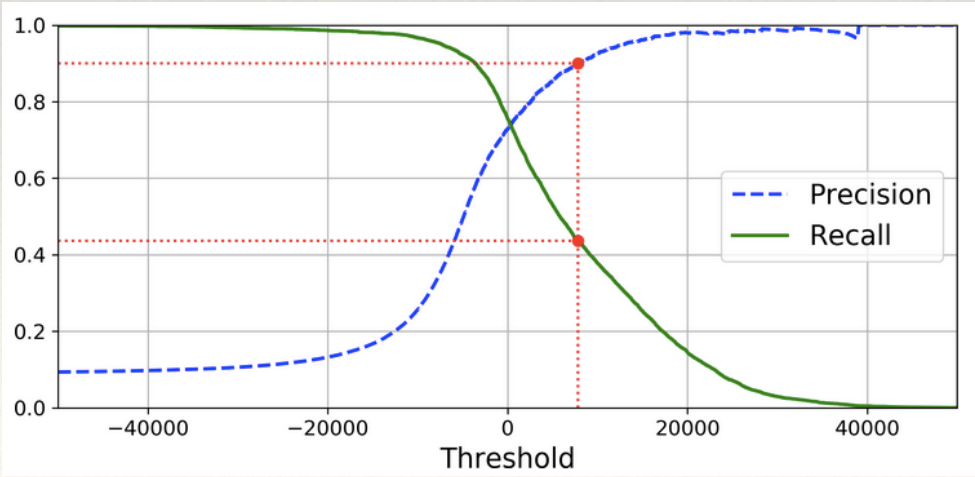
Precision/Recall trade-off

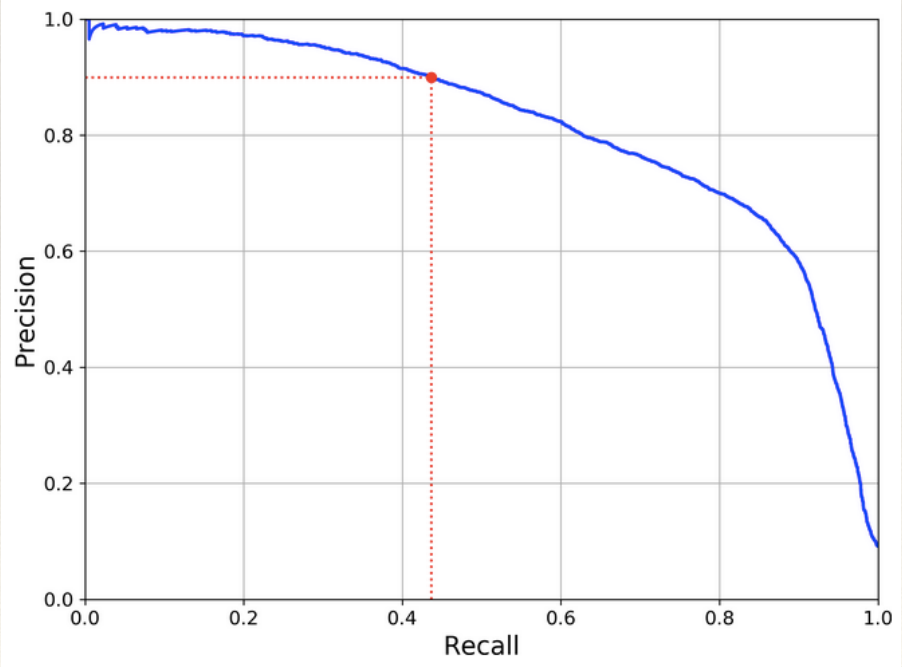


SGDClassifier, for each instance, computes a score based on a decision function, and if that score is greater/smaller than a threshold, it assigns the instance to the positive/negative class

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)

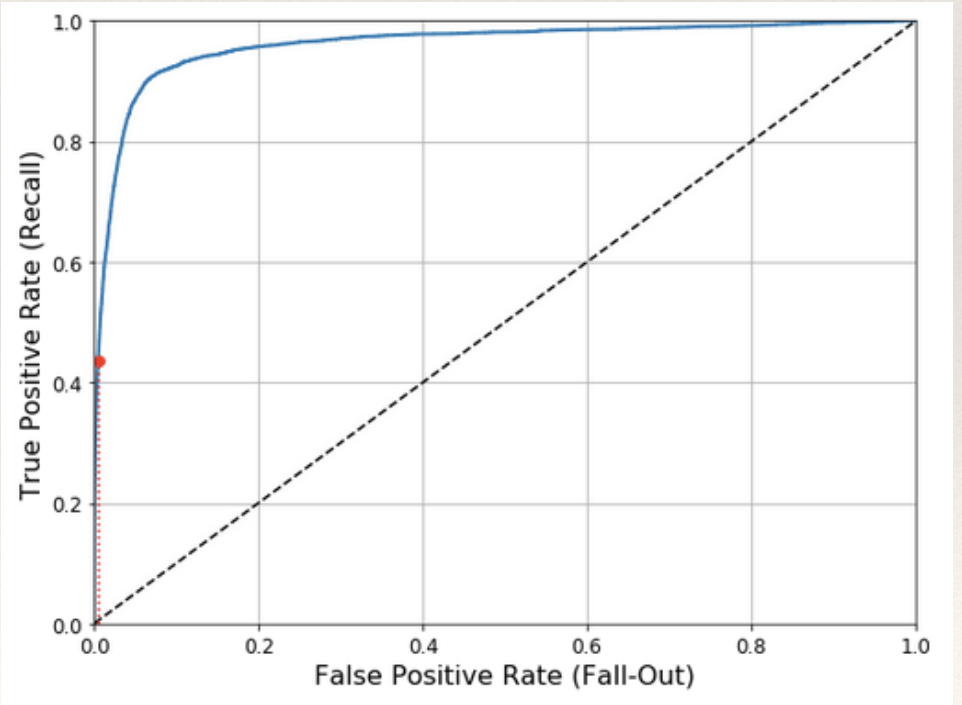
The **Receiver Operating Characteristic** (**ROC**) curve is another very common tool used with binary classifier

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.

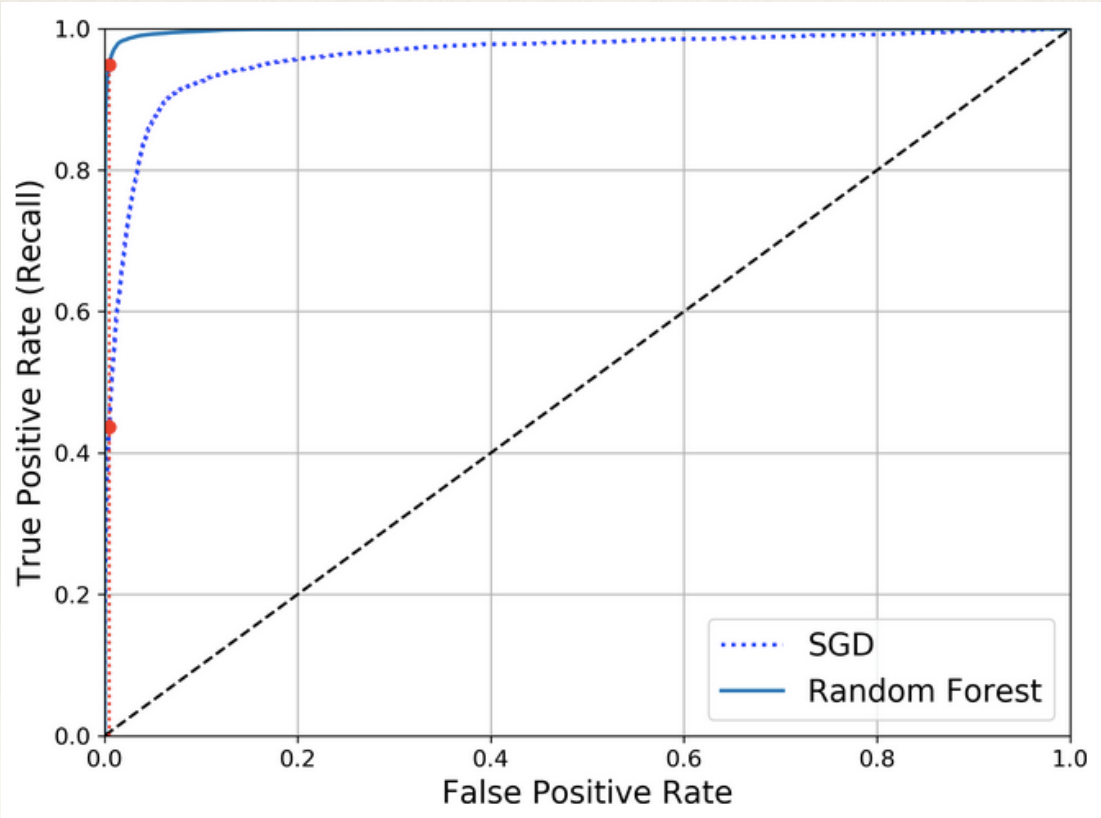
```
from sklearn.metrics import roc_auc_score

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. train 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 know 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…

Train a **multi-class** classifier

Which classifier can I use?

- some algos (such as Random Forest classifiers or naive Bayes classifiers) are capable of handling multiple classes directly
- others (such as Support Vector Machine classifiers or Linear classifiers) are strictly binary classifiers

However, there are various strategies that you can use to **perform multiclass classification using multiple binary classifiers**

- e.g. **one-versus-one** (**OvO**): train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, ..
  - for N classes, you train N×(N–1)/2 classifiers → for MNIST, 45 binary classifiers!
  - advantage: each classifier needs to be trained only on the part of the training set for the two classes that it must distinguish. E.g. Some algos (such as SVM classifiers) scale poorly with the size of the training set, OvO is preferred (it is faster to train many classifiers on small training sets than training few classifiers on large training sets)
- e.g. **one-versus-all** (**OvA**): train 10 binary classifiers, one for each digit; when you want to classify an image, you get the decision score from each classifier for that image and select the class whose classifier outputs the highest score
  - preferred for most binary classification algos

Error analysis on a full, multi-class classification for MNIST

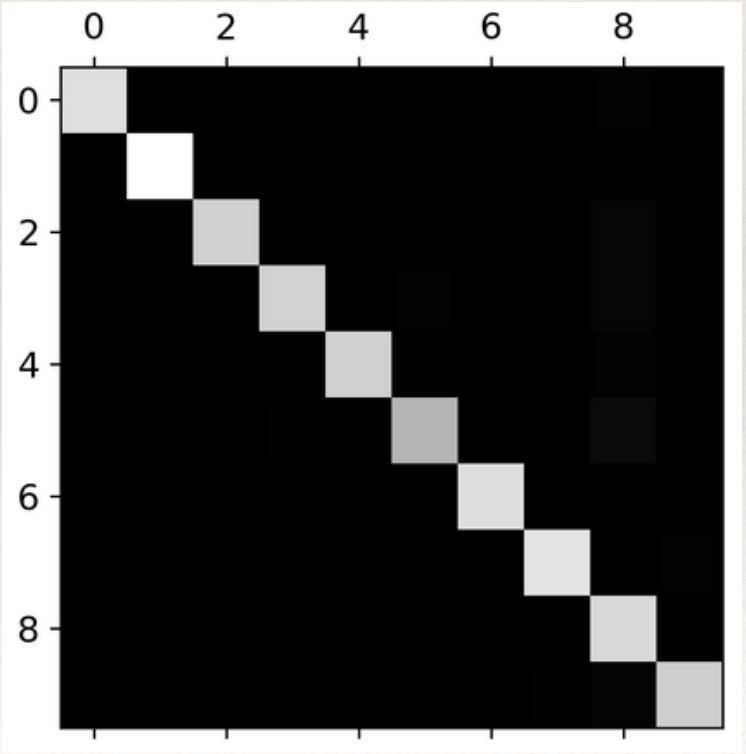
Assume you selected and trained your model. Now you want to improve it: you need first to **analyse the type of errors it makes**

- start by checking the confusion matrix
- note it is not 2x2 now, but 10x10 better to visualize it (matplotlib)

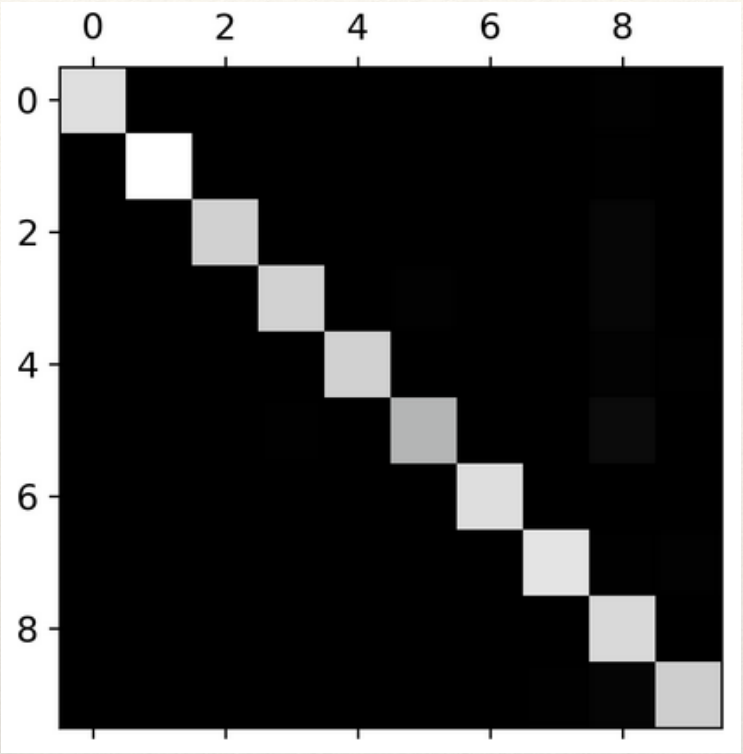
```
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

Fairly good, since most images are on the main diagonal, which means that they were classified correctly.

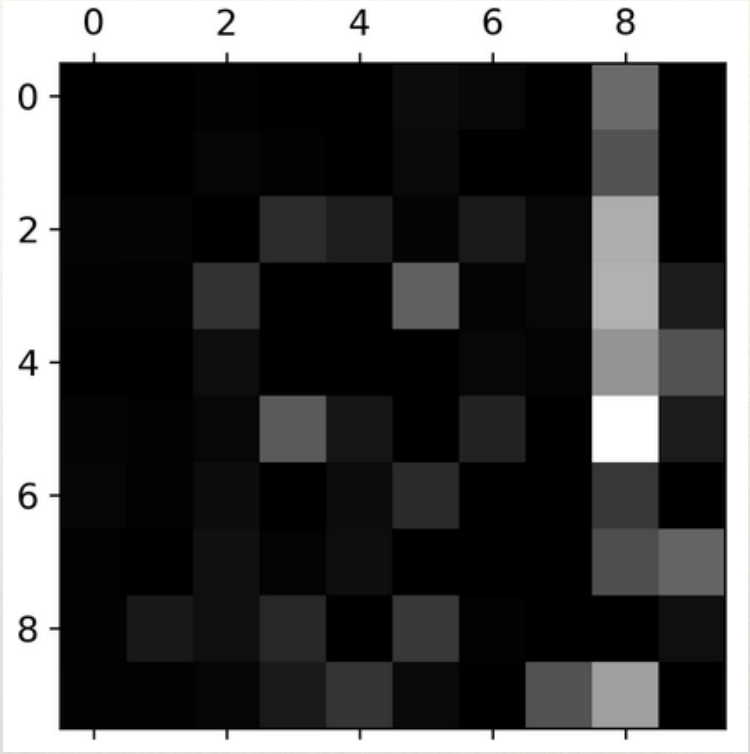
5s is a bit darker though. Less statistics for 5s? classifier performs worse on 5s wrt other numbers?



Error analysis on a full, multi-class classification for MNIST



Normalise each value in the confusion matrix to the number of images in the corresponding class



Observations and possible actions:

- many images get misclassified as 8s. 5s are often misclassified as 8s and as 3s
- **you should reduce the false 8s** (gather more training data that looks like 8s but are not? engineer new features that would help the classifier (e.g. count # closed loops in digits, preprocess the image to highlight more e.g. loops, ..)
- **you should improve on 3s vs 5s** (too similar: try a non-linear model? insist on image preprocessing to ensure that they are well centered and not too rotated? ..)

MNIST recap

Now you know how to:

- get the data and inspect it
- train a binary classifier
- measuring performance using CV → accuracy
- confusion matrix
- precision, recall and their trade-off
- F1 score
- ROC, AUC
- train a multi class classifier
- error analysis

Well done! That's quite something for a cold start!

Remember to be curious and browse/play the (verbose) code..