Learning from Data

Outline

- Main challenges of machine learning.
- Testing and validating models
- Model Evaluation
  - Regression metrics
  - Classification metrics
- Classification Example

Main challenges of machine learning

Main challenges of machine learning

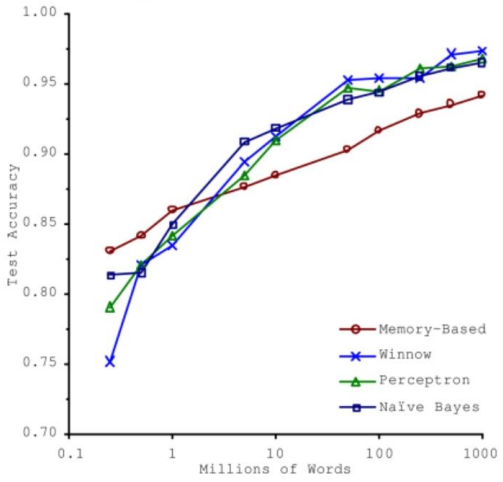
- Since your main task is to select a learning algorithm and train it on some data, the two things that can go wrong are "**bad algorithm**" and "**bad data**". Examples of bad data are:
  - Insufficient quantity of training data
  - Nonrepresentative training data
  - Poor-quality data
  - Irrelevant features

Insufficient quantity of training data

- It takes a lot of data for most ML algorithms to work properly.
- Even for very simple problems you typically need thousands of examples.
- For complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

Insufficient quantity of training data

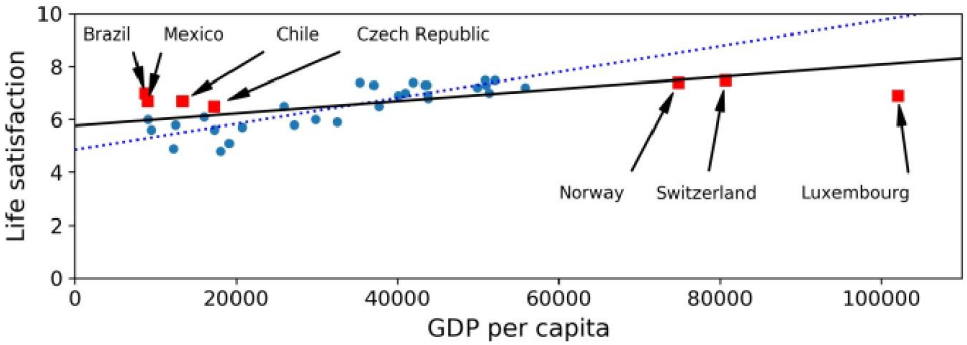
- Different Machine Learning algorithms performed almost identically well on a complex problem once they were given **enough data**.
- Small- and medium- sized datasets are still very common, and it is **not always easy or cheap** to get extra training data, so don't abandon algorithms just yet.



The importance of data versus algorithms

Nonrepresentative 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.



Nonrepresentative 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.
- If the sample is too small, you will have **sampling noise**.
- Even very large samples can be nonrepresentative if the sampling method is flawed. This is called **sampling bias**.

Poor-quality data

- If your training data is full of **errors, outliers, and noise**, it will make it harder for the system to detect the underlying patterns.
- It is often well worth the effort to spend time **cleaning up your training data**. For example:
  - 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 **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.

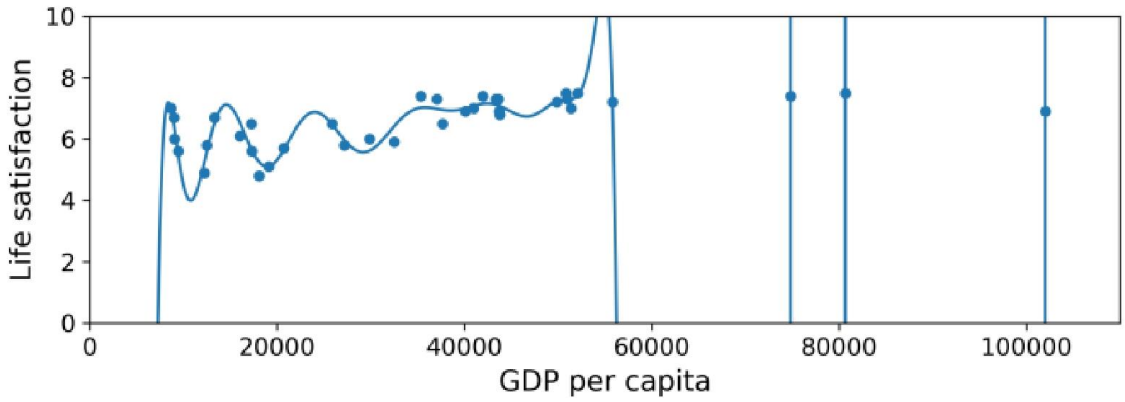
Irrelevant features

- As the saying goes: garbage in, garbage out. A critical part of the success of an ML project is coming up with a good set of features to train on. This process, called **feature engineering**, involves:
  - **Feature selection**: selecting the most useful features to train on among existing features.
  - **Feature extraction**: combining existing features to produce a more useful one (dimensionality reduction algorithms can help).
  - Creating new features by gathering new data.

Overfitting the training data

- Let's look at a couple of examples of **bad algorithms**.
- **Overfitting** means that the model performs well on the training data, but it does not generalize well (does not perform well on never-seen-before data).
- Overfitting happens when the **model is too complex** relative to the amount of the training data. The possible solutions are:

Overfitting the training data



Overfitting the training data

- The possible solutions are:
  - To **simplify the model** by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data or by constraining the model.
  - To gather more training data.
  - To reduce the noise in the training data (e.g., fix data errors and remove outliers)

Overfitting the training data

- Constraining a model to make it simpler and reduce the risk of overfitting is called **regularization**.
- Regularization is the process of penalizing hypotheses that are more complex to favor simpler, more general hypotheses.
- The amount of regularization to apply during learning can be controlled by a **hyperparameter**. A hyperparameter is a parameter of a learning algorithm (not of the model).

Underfitting the training data

- **Underfitting** is the opposite of overfitting: it occurs when your **model is too simple** to learn the underlying structure of the data.
- The main options to fix this problem are:
  - Selecting a more powerful model, with more parameters.
  - Reducing the constraints on the model (e.g., reducing the regularization hyperparameter)

Main challenges of machine learning

- To summarize:
  - The system will not perform well if your training set is too small, or if the data is not representative, noisy, or polluted with irrelevant features (garbage in, garbage out).
  - Your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit).

Testing and validating models

Training set and test set

- An ML model aims to make good predictions on new, previously unseen data. But if you are building a model from your data set, how would you get the previously unseen data? Well, one way is to divide your data set into two subsets:
  - **Training set**—a subset to train a model.
  - **Test set**—a subset to test the model

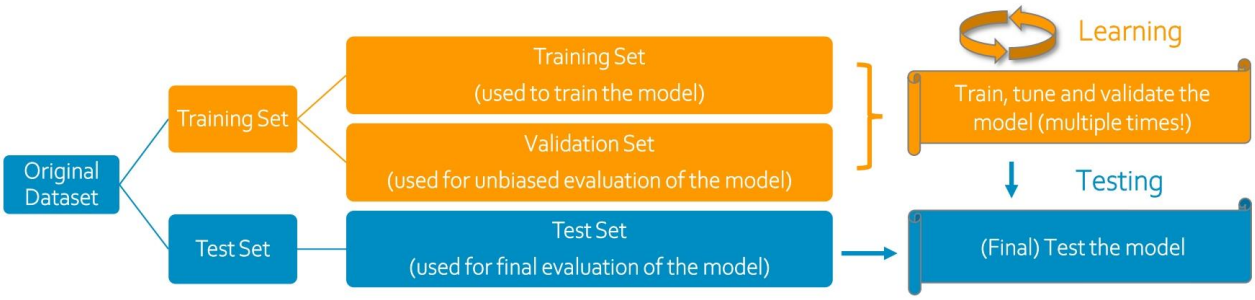
Hyperparameter tuning & model selection

- Developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the hyperparameters of the model).
- How do you choose the value of a hyperparameter?
- If the model is **evaluated multiple times using the test set**, the model may **overfit** to that specific **test set**. This can lead to overly optimistic performance estimates because the model might learn to perform well on the test set rather than truly generalizing to new, unseen data.

Hyperparameter tuning & model selection

- How do you choose the value of a hyperparameter?
- A common solution is called **holdout validation**: Hold out part of the training set to evaluate several candidate models and select the best one. The new heldout set is called the **validation set**.

Training – validation –test sets

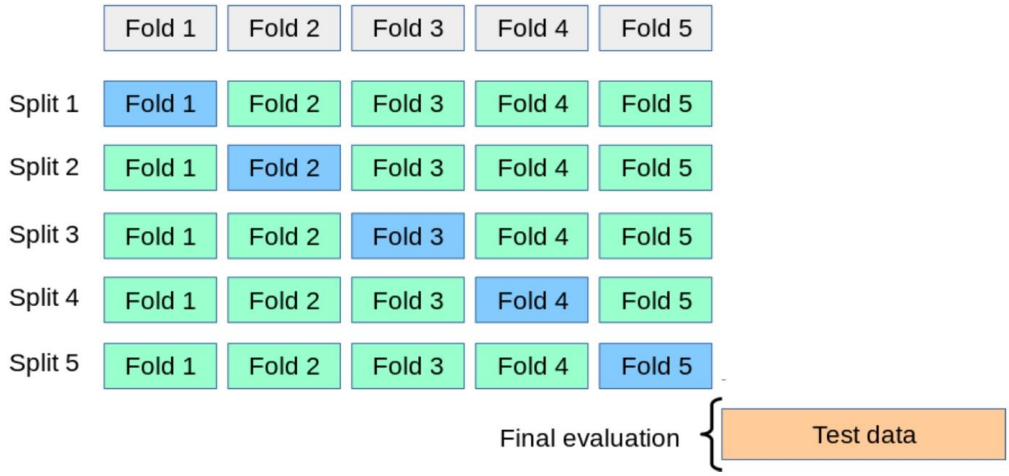


- The **test set** is **not available to the model for learning**, it is only used to ensure that the model generalizes well on **new "unseen" data**.

Hyperparameter tuning & model selection

- If the training set and the validation set are small, it is better to perform repeated **cross-validation**.
- In **K-fold cross-validation**, the training set is split into k distinct subsets called folds, then train and evaluate your model k times, picking a different fold for evaluation every time and training on the other k-1 folds. By **averaging** out all the evaluations of a model, we get a much more accurate measure of its performance.
- There is a **drawback**: the **training time is multiplied by** the **number of validation sets**.

Hyperparameter tuning & model selection

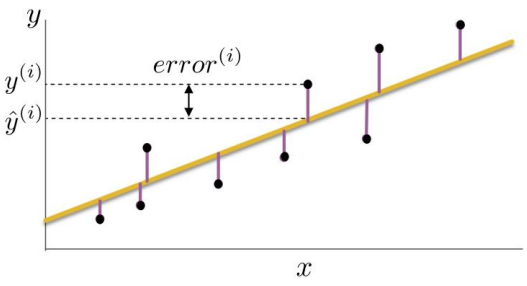


Model evaluation

Model evaluation: Regression metrics

Metrics	Equations
Mean Squared Error (MSE)	$MSE = \frac{1}{n}\sum_{i=0}^{n}(y^{(i)} - \hat{y}^{(i)})^2$
Root Mean Squared Error (RMSE)	$RMS = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(y^{(i)} - \hat{y}^{(i)})^2}$
Mean Absolute Error (MAE)	$MAE = \frac{1}{n}\sum_{i=0}^{n}\lvert y^{(i)} - \hat{y}^{(i)}\rvert$

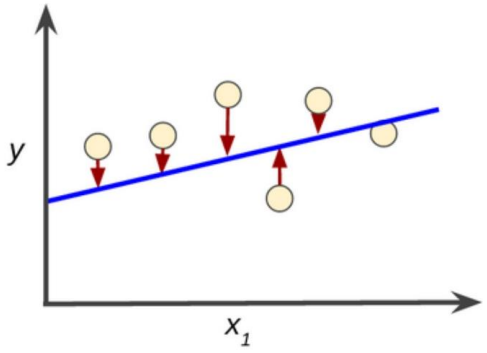
$y^{(i)}$: Data values
$\hat{y}^{(i)}$: Predicted values
$\bar{y}$: Mean value of data values,
n : Number of data records



Model evaluation: Regression metrics

- 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.

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



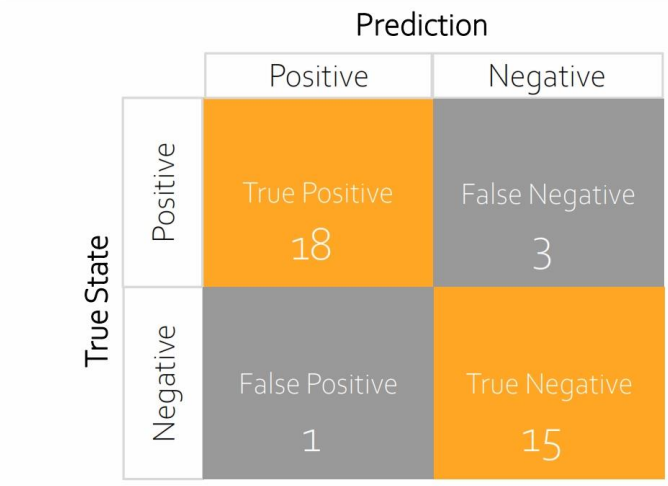
Model evaluation: Regression metrics

- If there are many outliers, in that case, you may consider using the **mean absolute error(MAE)**.

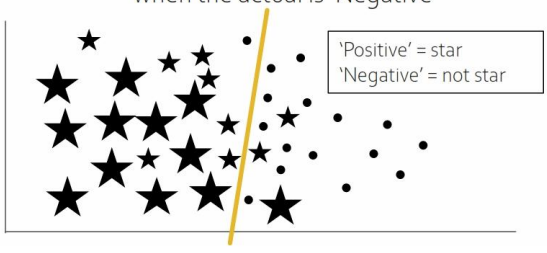
$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

- The RMSE is more sensitive to outliers than the MAE.

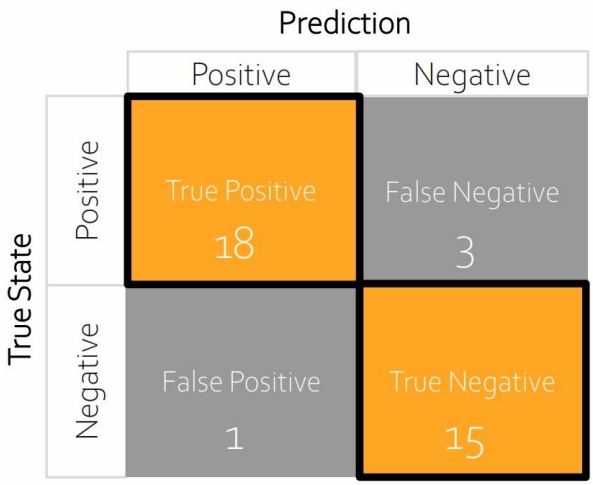
Model evaluation – Classification metrics



True Positive: Predicted 'Positive' when the actual is 'Positive'
False Positive: Predicted 'Positive' when the actual is 'Negative'
False Negative: Predicted 'Negative' when the actual is 'Positive'
True Negative: Predicted 'Negative' when the actual is 'Negative'



Classification metrics: Accuracy



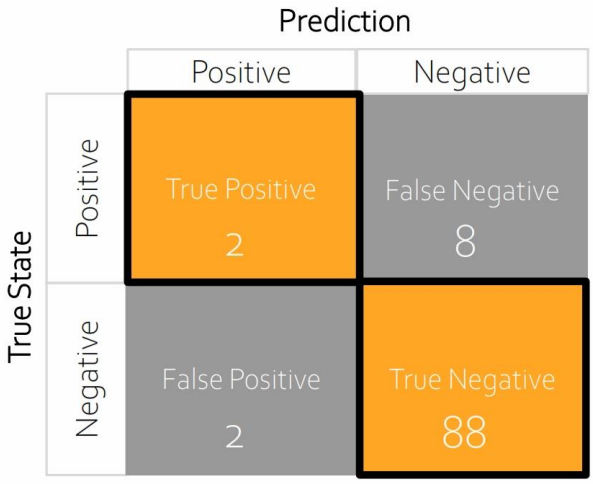
Accuracy: The percent (ratio) of cases classified correctly

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

$$Accuracy = \frac{18 + 15}{18 + 1 + 3 + 15} = 0.89$$

*$(bad)\ 0 \leq Accuracy \leq 1\ (good)$

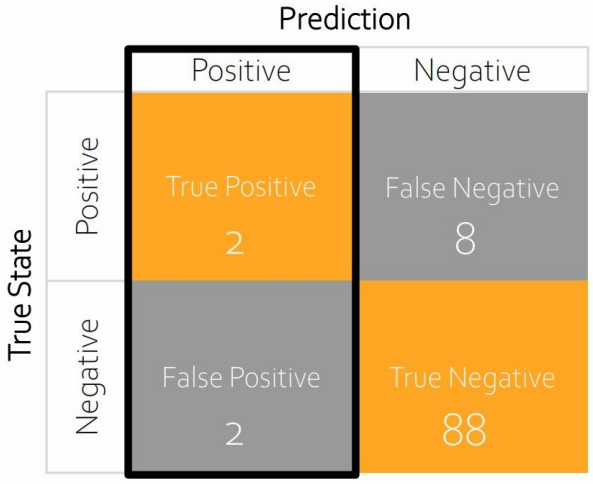
Classification metrics: Accuracy



High Accuracy Paradox: Accuracy is misleading when dealing with imbalanced datasets - few True Positives, the 'rare' class, and many True Negatives, the 'dominant' class. High Accuracy even when few True Positives.

$$Accuracy = \frac{2 + 88}{2 + 2 + 8 + 88} = 0.90$$

Classification metrics: Precision



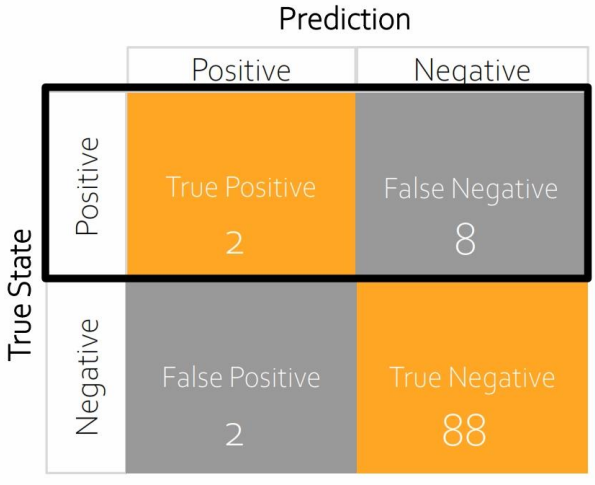
Precision: Accuracy of a predicted positive outcome

$$Precision = \frac{TP}{TP + FP}$$

$$Precision = \frac{2}{2+2} = 0.50$$

$^{*}(bad)\ 0 \leq Precision \leq 1\ (good)$

Classification metrics: Recall



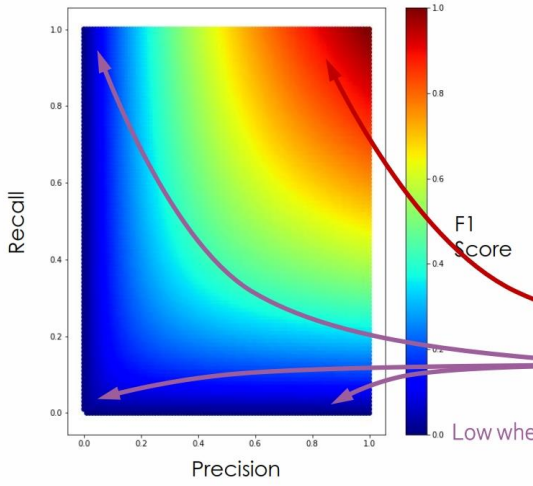
Recall: Measures model's ability to predict a positive outcome

$$Recall = \frac{TP}{TP + FN}$$

$$Recall = \frac{2}{2+8} = 0.20$$

$^{*}(bad)\ 0 \leq Recall \leq 1\ (good)$

Classification Metrics: F_1 Score



F1 Score: A combined metric, the harmonic mean of Precision and Recall.

$$F1\ Score = \frac{2 * (Precision * Recall)}{Precision + Recall}$$

$$^*(bad)\ 0 \leq F1\ Score \leq 1\ (good)$$

Low when one or both of the Precision and Recall are low

High when both Precision and Recall are high

Classification

MNIST dataset

- The MNIST dataset is a set of 70,000 small images of handwritten digits.
- Each image is labeled with the digit it represents.
- Studied so much that it is often called the "Hello World" of ML.
- Scikit-Learn provides many helper functions to download popular datasets. MNIST is one of them.

```
>>> 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'])
```

- Datasets loaded by Scikit-Learn generally have a similar dictionary structure including:
  - 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

MNIST dataset

- There are 70,000 images, and each image has 784 features. Each image is 28×28 pixels.
- Grab an instance's feature vector, reshape it to a 28×28 array, and display it using matplotlib's imshow()

```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

```
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")
plt.show()
```

MNIST dataset

- The label is a string. Let's cast y to integers.
- The MNIST dataset is split into a training set (the first 60,000 images) and a test set (the last 10,000 images).
- The training set is already shuffled.

```
>>> y[0]
'5'

>>> y = y.astype(np.uint8)

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

Training a binary classifier

- Simplify the problem to identify one digit, number 5.
- Create the target vectors for this classification task.
- Pick a classifier and train it. In this example, a stochastic gradient descent (SGD) classifier is used.
- Use the trained classifier to detect images of the number 5.

```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
# True for all 5s, False for all other digits.

from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

Performance measures

- Evaluating a classifier is significantly trickier than evaluating a regressor.
- Use the cross_val_score() function to evaluate your SGDClassifier mode using K-fold cross-validation, with three 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])
```

- **Accuracy** is generally **not the preferred performance measure** for classifiers, especially when you are dealing with **skewed datasets** (i.e., when some classes are much more frequent than others).

Confusion Matrix

- To compute the confusion matrix, you first need to have a set of predictions, so they can be compared to the actual targets. You can use the cross_val_predict() function:

```
from sklearn.model_selection import cross_val_predict

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

- Now you are ready to get the confusion matrix using the confusion_matrix() function.

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53057,  1522],
       [ 1325,  4096]])
```

Confusion Matrix

- **Each row** in a confusion matrix represents an **actual class**, while **each column** represents a **predicted class**.
- A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal:

```
>>> 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]])
```

Precision and Recall

- The **accuracy of the positive predictions** is called the **precision** of the classifier.

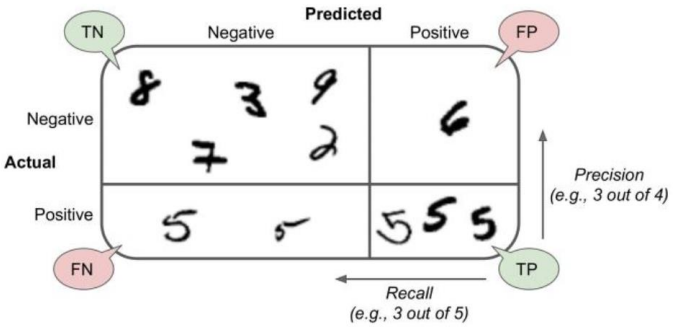
$$precision = \frac{true\ positives}{no.of\ predicted\ positive} = \frac{TP}{TP + FP}$$

- Precision is typically used along with another metric named **recall**, also called **sensitivity** or **true positive rate** (TPR): this is the ratio of positive instances that are correctly detected by the classifier.

$$recall = \frac{true\ positives}{no.of\ actual\ positive} = \frac{TP}{TP + FN}$$

Precision and Recall

Actual Class	Predicted Class: Positive	Predicted Class: Negative	
Positive	True Positive (TP)	False Negative (FN) **Type II Error**	**Sensitivity** $\frac{TP}{(TP+FN)}$
Negative	False Positive (FP) **Type I Error**	True Negative (TN)	**Specificity** $\frac{TN}{(TN+FP)}$
	Precision $\frac{TP}{(TP+FP)}$	**Negative Predictive Value** $\frac{TN}{(TN+FN)}$	**Accuracy** $\frac{TP+TN}{(TP+TN+FP+FN)}$



F_1 score

- Scikit-Learn provides functions to compute precision and recall.

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

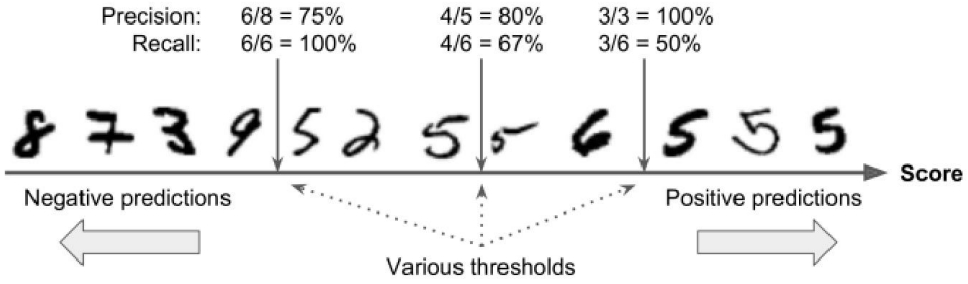
- F_1 score is the harmonic mean of precision and recall.

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} = \frac{TP}{TP + \frac{1}{2}(FN + FP)}$$

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

Precision/recall tradeoff

- Increasing precision reduces recall, and vice versa.



ROC curve

- The receiver operating characteristic **(ROC) curve** is another common tool used with **binary classifiers**.
- The ROC curve plots sensitivity (recall) versus 1 – specificity.
- To plot the ROC curve, compute the TPR and FPR for various threshold values, using the roc_curve() function.

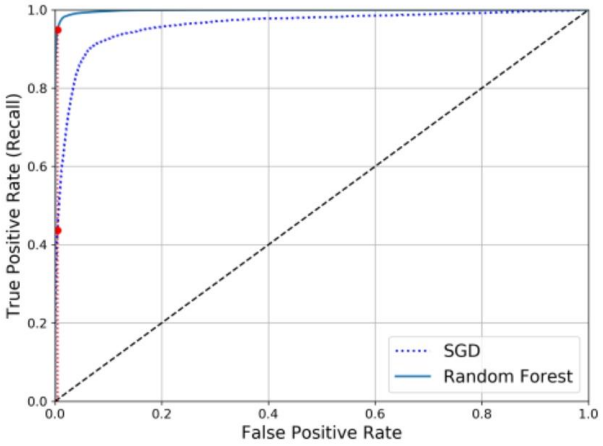
```
from sklearn.metrics import roc_curve

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


def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # dashed diagonal
    [...] # Add axis labels and grid

plot_roc_curve(fpr, tpr)
plt.show()
```

The AUC

- One way to compare classifiers is to measure the area under the curve (AUC).

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

- Use the **precision/recall curve** whenever the **positive class is rare** or when you care more about the false positives than the false negatives, and the ROC curve otherwise.



Multiclass classification

- Binary classifiers distinguish between two classes, multiclass classifiers can distinguish between more than two classes.
- Some algorithms (such as random forest 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.
- Various strategies that you can use to perform multiclass classification using multiple binary classifiers:
  - One-versus-all (OvA)
  - One-versus-one (OvO)

One-versus-all (OvA)

- To create a system that can classify the digit images into 10 classes:
  - **Train 10 binary classifiers**, one for each digit.
  - To classify an image, get the decision score from each classifier for that image.
  - Select the class whose classifier outputs the highest score.

One-versus-One (OvO)

- To create a system that can classify the digit images into 10 classes:
  - **Train a binary classifier for every pair of digits**.
- If there are N classes, you need to **train $N \times (N-1)/2$ classifiers**. For the MNIST problem, this means training 45 binary classifiers!
- **Main advantage**: **Each classifier** only needs to be **trained on** the **part of the training set** for the two classes that it must distinguish.