IASD M2 at Paris Dauphine

Become a Kaggle Master

2: Competition, metrics

Eric Benhamou

Agenda

Part I: general concepts

1. Introduction to Kaggle (concept and API)
2. Competition, metrics
3. Validation
4. Hyper parameters tuning
5. Model ensemble with blending and stacking

Part II: Competitions

5. Predict Financial markets
6. Analyze News
7. Design your portfolio

Competition Tasks and Metrics

- In a competition, you start by examining the target metric. Understanding how your model’s errors are evaluated is key for scoring highly in every competition. When your predictions are submitted to the Kaggle platform, they are compared to a ground truth based on the target metric.

Example of metrics

- For instance, in the ***Titanic* competition** (https://www.kaggle.com/c/titanic/), all your submissions are evaluated based on *accuracy*, the percentage of surviving passengers you correctly predict. The organizers decided upon this metric because the aim of the competition is to find a model that estimates the probability of survival of a passenger under similar circumstances.

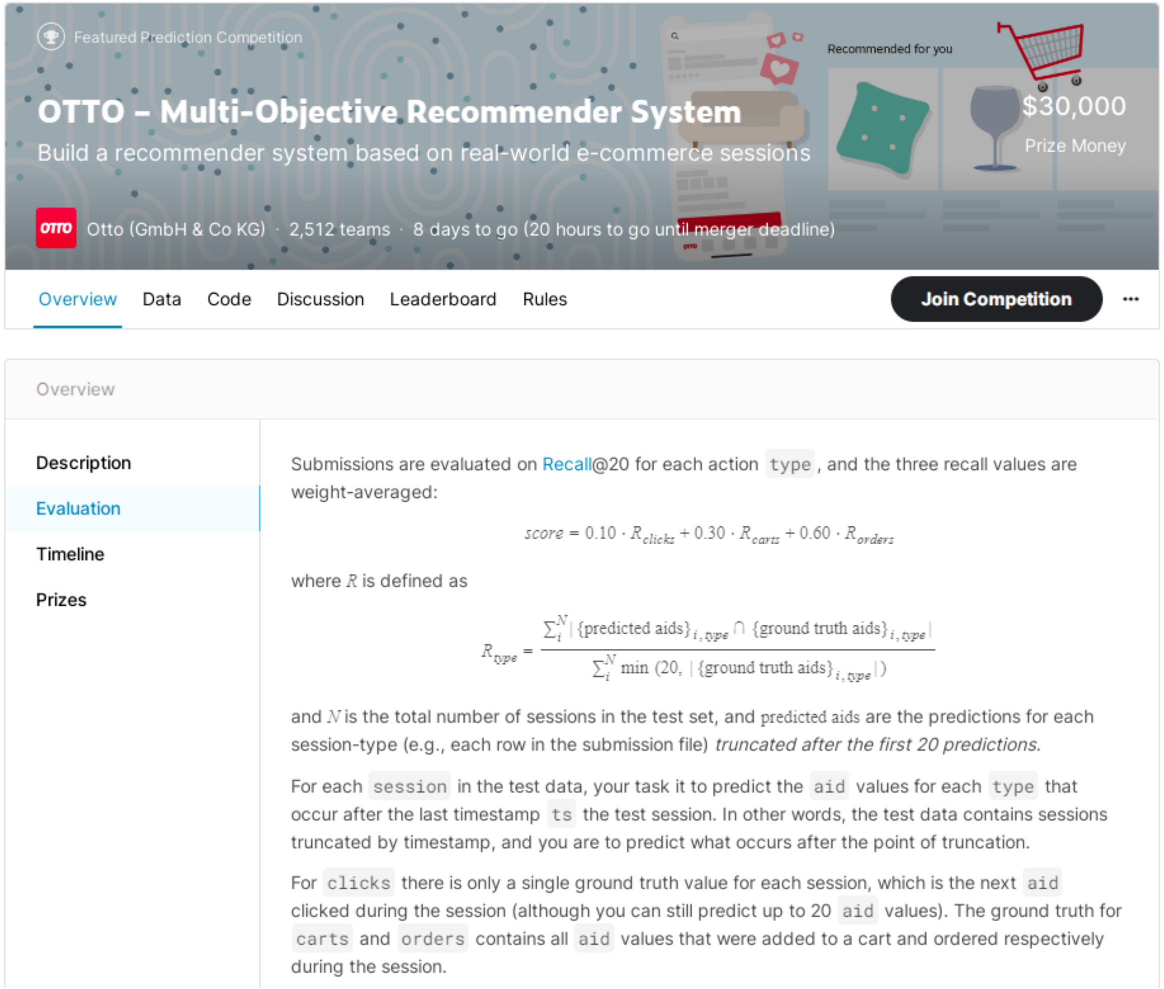
- In another knowledge competition, ***House Prices - Advanced Regression** Techniques* (https://www.kaggle.com/c/house-pricesadvanced-regression-techniques), your work will be evaluated based on an *average difference* between your prediction and the ground truth. This involves computing the logarithm, squaring, and taking the square root, because the model is expected to be able to quantify as correctly as possible the order of the price of a house on sale.

Agenda

- Evaluation metrics and objective functions
- Basic types of tasks: regression, classification, and ordinal
- The Meta Kaggle dataset
- Handling never-before-seen metrics
- Metrics for regression (standard and ordinal)
- Metrics for binary classification (label prediction and probability)
- Metrics for multi-class classification
- Metrics for object detection problems
- Metrics for multi-label classification and recommendation problems
- Optimizing evaluation metrics

Where to find the metric?

- In a Kaggle competition, you can find the evaluation metric in the left menu on the **Overview** page of the competition.
- By selecting the **Evaluation** tab, you will get details about the evaluation metric. Sometimes you will find the metric formula, the code to reproduce it, and some discussion of the metric.
- On the same page, you will also get an explanation about the submission file format, providing you with the header of the file and a few example rows.



Featured Prediction Competition
OTTO – Multi-Objective Recommender System
Build a recommender system based on real-world e-commerce sessions
Recommended for you
$30,000
Prize Money
OTTO
Otto (GmbH & Co KG) · 2,512 teams · 8 days to go (20 hours to go until merger deadline)
Overview
Data
Code
Discussion
Leaderboard
Rules
Join Competition
···
Overview
Description
Evaluation
Timeline
Prizes

Submissions are evaluated on Recall@20 for each action `type`, and the three recall values are weight-averaged:

$$score = 0.10 \cdot R_{clicks} + 0.30 \cdot R_{carts} + 0.60 \cdot R_{orders}$$

where R is defined as

$$R_{type} = \frac{\sum_i^N |\{\text{predicted aids}\}_{i,type} \cap \{\text{ground truth aids}\}_{i,type}|}{\sum_i^N \min(20, |\{\text{ground truth aids}\}_{i,type}|)}$$

and N is the total number of sessions in the test set, and predicted aids are the predictions for each session-type (e.g., each row in the submission file) *truncated after the first 20 predictions.*

For each `session` in the test data, your task it to predict the `aid` values for each `type` that occur after the last timestamp `ts` the test session. In other words, the test data contains sessions truncated by timestamp, and you are to predict what occurs after the point of truncation.

For `clicks` there is only a single ground truth value for each session, which is the next `aid` clicked during the session (although you can still predict up to 20 `aid` values). The ground truth for `carts` and `orders` contains all `aid` values that were added to a cart and ordered respectively during the session.

Terminology

- The analysis of the Kaggle evaluation metric should be **your first act** in a competition as it conditions your ranking. But let us first discuss some terminology
- A **loss function** is a function that is defined on a single data point, and, considering the prediction of the model and the ground truth for the data point, computes a penalty.
- A **cost function** takes into account the whole dataset used for training (or a batch from it), computing **a sum or average over the loss penalties of its data points**. It can comprise further constraints, such as the L1 or L2 penalties, for instance. The cost function directly affects how the training happens.

Objective function

- An **objective function** is the most general (and safe-to-use) term related to the scope of optimization during machine learning training: it comprises cost functions, but it is not limited to them. An objective function, in fact, can also take into account goals that are not related to the target: for instance, requiring sparse coefficients of the estimated model or a minimization of the coefficients' values, such as in **L1 and L2 regularizations**.
- Moreover, whereas loss and cost functions imply an optimization based on minimization, an objective function is neutral and can imply either a maximization or a minimization activity performed by the learning algorithm.

Terminology last part

- A **scoring function** suggests better prediction results if scores from the function are higher, implying a maximization process.
- An **error function** instead suggests better predictions if smaller error quantities are reported by the function, implying a minimization process.

Basic types of tasks

- The two most common tasks are **regression** tasks and **classification** tasks.

- Recently, there have also been **reinforcement learning** (**RL**) tasks, but RL doesn’t use metrics for evaluation; instead, it relies on a ranking derived from direct match-ups against other competitors

Regression

- **Regression** requires you to build a model that can predict a real number; often a positive number, but there have been examples of negative number prediction too.
- A classic example of a regression problem is *House Prices - Advanced Regression Techniques*, because you have to guess the value of a house.
- The evaluation of a regression task involves computing a distance between your predictions and the values of the ground truth. This difference can be evaluated in different ways, for instance by squaring it in order to punish larger errors, or by applying a log to it in order to penalize predictions of the wrong scale.

Classification

- When facing a **classification** task on Kaggle, there are more nuances to take into account.
- The classification, in fact, could be **binary**, **multi-class**, or **multi-label**.

- In **binary** problems, you have to guess if an example should be classified or not into a specific class (usually called the *positive* class and compared to the *negative* one).

Binary and imbalance

- Though counting the exact number of correct matches in a binary classification may seem a valid approach, this won’t actually work well when **there is an imbalance**, that is, a different number of examples, between the positive and the negative class.
- Classification based on an imbalanced distribution of classes **requires evaluation metrics that take the imbalance into account**, if you want to correctly track improvements on your model.

Multi class

- When you have more than two classes, you have a **multi-class** prediction problem. This also requires the use of suitable functions for evaluation, since it is necessary to keep track of the overall performance of the model, but also to ensure that the performance across the classes is comparable (for instance, your model could underperform with respect to certain classes).
- Here, each case can be in one class exclusively, and not in any others. A good example is *Leaf Classification* https://www.kaggle.com/c/leafclassification), where each image of a leaf specimen has to be associated with the correct plant species.

Multi label

- Finally, when your class predictions are not exclusive and you can predict multiple class ownership for each example, you have a **multi-label** problem that requires further evaluations in order to control whether your model is predicting the correct classes, as well as the correct number and mix of classes.
- For instance, in *Greek Media Monitoring Multilabel Classification (WISE 2014)* (https://www.kaggle.com/c/wise-2014), you had to associate each article with all the topics it deals with.

Ordinal

- In a problem involving a prediction on an ordinal scale, you have to guess **integer numeric labels**, which are naturally ordered. As an example, the magnitude of an earthquake is on an ordinal scale. In addition, data from marketing research questionnaires is often recorded on ordinal scales (for instance, consumers’ preferences or opinion agreement).
- Since an ordinal scale is made of ordered values, ordinal tasks can be considered somewhat **halfway between regression and classification**, and you can solve them in **both ways**.

Multi class approach

- The most common way is to treat your ordinal task as a **multi-class** problem. In this case, you will get a prediction of an integer value (the class label) but the prediction will not take into account that the classes have a certain order.
- Often, probabilities will be distributed across the entire range of possible values, depicting a multi-modal and often asymmetric distribution (whereas you should expect a Gaussian distribution around the maximum probability class).

Regression approach

- The other way to solve the ordinal prediction problem is to treat it as a **regression** problem and then post-process your result.

- In this way, the order among classes will be taken into consideration, though the prediction output won’t be immediately useful for scoring on the evaluation metric. In fact, in **a regression you get a float number** as an output, **not an integer** representing an ordinal class; moreover, the result will include the full range of values between the integers of your ordinal distribution and possibly also values outside of it. **Cropping the output values and casting them into integers by unit rounding may do the trick**, but **this might lead to inaccuracies requiring some more sophisticated post-processing that we will discuss**

The Meta Kaggle dataset

- The Meta Kaggle dataset (https://www.kaggle.com/kaggle/meta-kaggle) is a collection of rich data about Kaggle’s community and activity, published by Kaggle itself as a public dataset. It contains CSV tables filled with **public activity from Competitions, Datasets, Notebooks, and Discussions.**

- The CSV tables are updated daily, so you’ll have to refresh your analysis often, but that’s worth it given the insights you can extract.

Why looking at the Meta Kaggle dataset?

- Here, we are going to use it in order to figure out what evaluation metrics have been used most frequently for competitions in the last seven years. By looking at the most common ones in this chapter, you’ll be able to start any competition from solid ground and then refine your knowledge of the metric, picking up competition-specific nuances using the discussion you find in the forums.

Looking at the most common metrics

```
import numpy as np
import pandas as pd

comps = pd.read_csv("/kaggle/input/meta-kaggle/Competitions.csv")
evaluation = ['EvaluationAlgorithmAbbreviation', 'EvaluationAlgorithmName’,
        'EvaluationAlgorithmDescription', ]
compt = ['Title', 'EnabledDate', 'HostSegmentTitle']
df = comps[compt + evaluation].copy()
df['year'] = pd.to_datetime(df.EnabledDate).dt.year.values
df['comps'] = 1
time_select = df.year >= 2015
competition_type_select = df.HostSegmentTitle.isin(['Featured', 'Research'])
```

Looking at the most common metrics

```
pivot_table = pd.pivot_table(df[time_select & competition_type_select],
            values='comps’,
            index=['EvaluationAlgorithmAbbreviation'], columns=['year'], fill_value=0.0,
             aggfunc=np.sum, margins=True).sort_values(by=('All'), ascending=False)

# to print the 20 first rows
pivot_table = pivot_table.iloc[1:, :].head(20)
```

Results

Number	year	2015	2016	2017	2018	2019	2020	2021	2022	All
1	**RMSE**	8	23	42	115	193	300	203	73	957
2	**AUC**	13	31	42	66	137	129	161	43	622
3	**FScoreMicro**	5	2	13	40	69	83	100	39	351
4	**MAE**	3	8	21	39	39	46	62	22	240
5	**MSE**	0	2	10	10	14	57	38	18	149
6	**F_{Beta} (deprecated)**	0	2	2	8	20	33	28	8	101
7	**LogLoss**	3	4	11	8	16	24	18	7	91
8	**MAP@{K}**	1	5	6	17	15	15	11	10	80
9	**FScoreMacro**	0	0	0	3	1	26	35	12	77
10	**RMSLE**	3	1	9	8	18	19	14	3	75
11	**Levenshtein Mean**	5	1	0	2	18	19	20	6	71
12	**NDCG@{K}**	1	2	7	9	12	14	7	2	54
13	**MulticlassLoss**	5	9	7	11	5	7	5	2	51
14	**MAPE**	0	0	0	2	5	14	17	3	41
15	**R2Score**	0	0	1	3	3	12	13	5	37
16	**Weighted Categorization Accuracy**	0	0	1	1	6	6	11	2	27
17	**Dice**	0	1	1	1	6	6	5	1	21
18	**Intersection Over Union Object SegmentationBeta**	0	0	0	2	0	2	11	4	19
19	**Mean Best Error AtK**	0	0	2	5	5	2	3	1	18
20	**MCAUC**	1	0	1	1	3	6	6	0	18

For a specific metric

```
metric = 'AUC'
metric_select = df['EvaluationAlgorithmAbbreviation']==metric
print(df[time_select&competition_type_select&metric_select]
[['Title', 'year']])
```

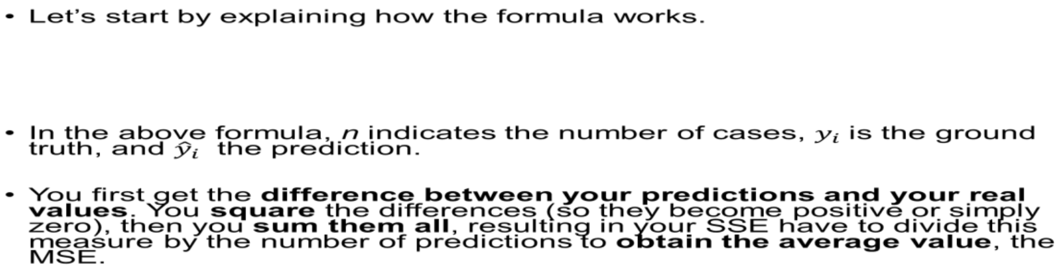
Mean squared error (MSE) and R squared

- The root mean squared error is the root of the **mean squared error** (**MSE**), which is nothing else but the mean of the good old **sum of squared errors** (**SSE**) that you learned about when you studied how a regression works.
- Here is the formula for the MSE:

$$MSE = \frac{1}{n}SSE = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y_i} - y_i)^2$$

Explanation 1/

- Let’s start by explaining how the formula works.



- Let’s start by explaining how the formula works.
- In the above formula, *n* indicates the number of cases, y_i is the ground truth, and $\hat{y}_i$ the prediction.
- You first get the **difference between your predictions and your real values**. You **square** the differences (so they become positive or simply zero), then you **sum them all**, resulting in your SSE have to divide this measure by the number of predictions to **obtain the average value**, the MSE.

- In the above formula, *n* indicates the number of cases, y_i is the ground truth, and $\hat{y}_i$ the prediction.

- You first get the **difference between your predictions and your real values**. You **square** the differences (so they become positive or simply zero), then you **sum them all**, resulting in your SSE have to divide this measure by the number of predictions to **obtain the average value**, the MSE.

Explanation 2/

- Usually, all regression models **minimize the SSE**, so you won’t have great problems **trying to minimize MSE** or its direct derivatives such as **R squared** (also called the coefficient of determination), which is given by:

$$R^2 = \frac{SSE}{SST} = \sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{(y_i - \bar{y})^2}$$

Explanation 3/

- Here, SSE (the sum of squared errors) is compared to the **sum of squares total** (**SST**), which is just the variance of the response. In statistics, in fact, SST is defined as the squared difference between your target values and their mean:

$$SST = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

- To put it another way, **R squared compares the squared errors of the model against the squared errors from the simplest model possible**, the average of the response. Since both SSE and SST have the same scale, R squared can help you to determine whether transforming your target is helping to obtain better predictions.

What about transformation?

- **Linear transformations** such as
  - minmax https://scikitlearn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
  - or standardization https://scikitlearn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- **do not change the performance** of any regressor, since they are linear transformations of the target.

What about transformation?

- **Non-linear** transformations, such as
  - the square root,
  - the cubic root,
  - the logarithm,
  - the exponentiation, and their combinations,
- should instead definitely **modify the performance of your regression model** on the evaluation metric (hopefully for the better, if you decide on the right transformation).

Root mean squared error (RMSE)

- RMSE is just the square root of MSE, but this implies some subtle change. Here is its formula:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

- In the above formula, *n* indicates the number of cases, y_i is the ground truth, and $\hat{y}_i$ the prediction.

- In MSE, **large prediction errors are greatly penalized** because of the squaring activity.
- In RMSE, this dominance is lessened because of the root effect (however, **you should always pay attention to outliers**; they can affect your model performance a lot, no matter whether you are evaluating based on MSE or RMSE).

What it means in practice?

- Consequently, depending on the problem, you can get a better fit with an algorithm using MSE as an objective function by first applying the square root to your target (if possible, because it requires positive values), then squaring the results. Functions such as the **TransformedTargetRegressor** in Scikit-learn help you to appropriately transform your regression target in order to get better-fitting results with respect to your evaluation metric.

Mean absolute error (MAE) 1/

- The **MAE** (**mean absolute error**) evaluation metric is the absolute value of the difference between the predictions and the targets. Here is the formulation of MAE:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |\widehat{y_i} - \ y_i|$$

- In the formula, *n* stands for the number of cases, y_i is the ground truth, and $\hat{y}_i$ the prediction.
- MAE is **not particularly sensitive to outliers** (unlike MSE, where errors are squared), hence you may find it is an evaluation metric in many competitions whose **datasets present outliers**.
- Moreover, you can easily work with it since many algorithms can directly use it as an objective function; otherwise, you can optimize for it indirectly by just training on the square root of your target and then squaring the predictions.

Mean absolute error (MAE) 2/

- In terms of **downside**, using MAE as an objective function results **in much slower convergence**, since you are **actually optimizing for predicting the median of the target (also called the L1 norm),** instead of the mean (also called the L2 norm), as occurs by MSE minimization.

- This results in more complex computations for the optimizer, so the training time can even grow exponentially based on your number of training cases

- see, for instance, this Stack Overflow question:

https://stackoverflow.com/questions/57243267/why-is-training-a-randomforest-regressor-with-mae-criterion-so-slow-compared-to).

Root mean squared log error (RMSLE)

- Another common transformation of MSE is **root mean squared log error** (**RMSLE**). MCRMSLE is just a variant made popular by the COVID-19 forecasting competitions, and it is the column-wise average of the RMSLE values of each single target when there are multiple ones. Here is the formula for RMSLE:

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(\hat{y}_i + 1) - \log(y_i + 1))^2}$$

- In the formula, *n* stands for the number of cases, y_i is the ground truth, and $\hat{y}_i$ the prediction.

Intuition and tips

- Since you are applying **a logarithmic transformation** to your predictions and your ground truth before all the other squaring, averaging, and rooting operations, you don’t **penalize huge differences between the predicted and the actual values**, especially when both are large numbers.

- In other words, what you care the most about when using RMSLE is ***the scale of your predictions*** *with respect to the scale of the ground truth*.

- As with **RMSE**, machine learning algorithms for regression can better optimize for RMSLE if you apply a logarithmic transformation to the target before fitting it (and then reverse the effect using the exponential function).

Metrics for classification (label prediction and probability)

- Having discussed the **metrics for regression problems**, we are going now to illustrate **the metrics for classification problems**, starting from the **binary classification problems** (when you have to predict between two classes), moving to the **multi-class** (when you have more than two classes), and then to the **multi-label** (when the classes overlap).

Accuracy

- When analyzing the performance of a binary classifier, the most common and accessible metric that is used is accuracy. A misclassification error is when your model predicts the wrong class for an example. The accuracy is just the complement of the misclassification error and it can be calculated as the ratio between the number of correct numbers divided by the number of answers:

$$Accuracy = \frac{correct\ answers}{total\ answers}$$

Intuition 1/

- As a metric, the accuracy is focused strongly on the effective performance of the model in a real setting: it tells you if the model works as expected.

- However, if your purpose is to evaluate and compare and have a clear picture of how effective your approach really is, you have to be cautious when using the accuracy because it can lead to **wrong conclusions when the classes are imbalanced** (when they have different frequencies).

- For instance, if a certain class makes up just 10% of the data, a predictor that predicts nothing but the majority class will be 90% accurate, proving itself quite useless in spite of the high accuracy.

Spotting imbalance data?

- How can you spot such a problem? You can do this easily by using a confusion matrix.

- In a confusion matrix, you create a two-way table comparing the actual classes on the rows against the predicted classes on the columns.

Confusion matrix

- You can create a straightforward one using the Scikit-learn confusion_matrix function:

```
Import sklearn
sklearn.metrics.confusion_matrix(y_true, y_pred, *, labels=None,
          sample_weight=None, normalize=None)
```

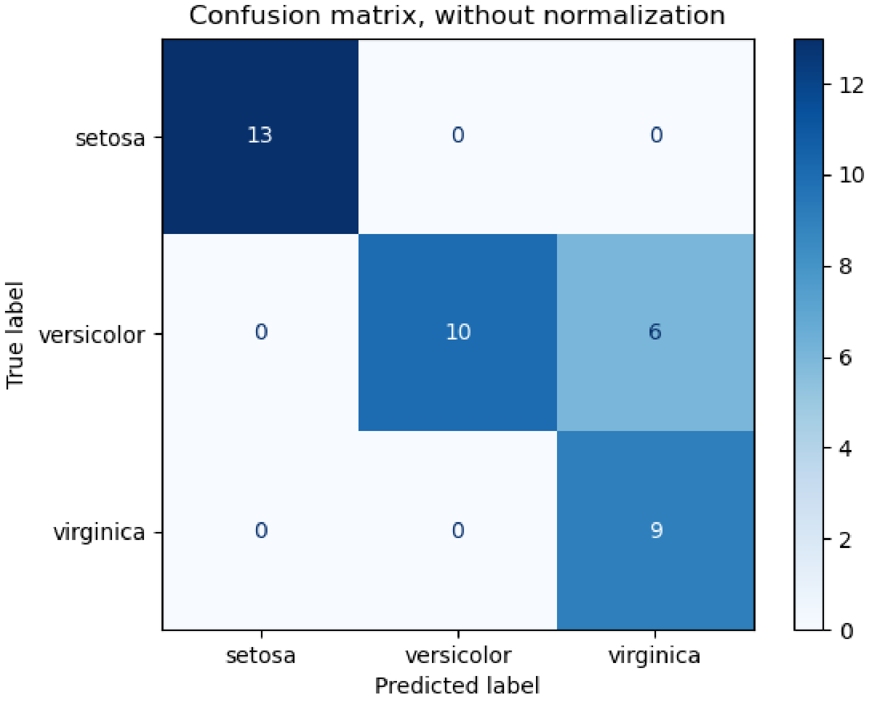
Rules

- Providing the y_true and y_pred vectors will suffice to return you a meaningful table, but you can also provide row/column labels and sample weights for the examples in consideration, and normalize (set the marginals to sum to 1) over the true examples (the rows), the predicted examples (the columns), or all the examples.

- A perfect classifier will have all the cases on the principal diagonal of the matrix. **Serious problems with the validity of the predictor are highlighted if there are few or no cases on one of the cells of the diagonal.**

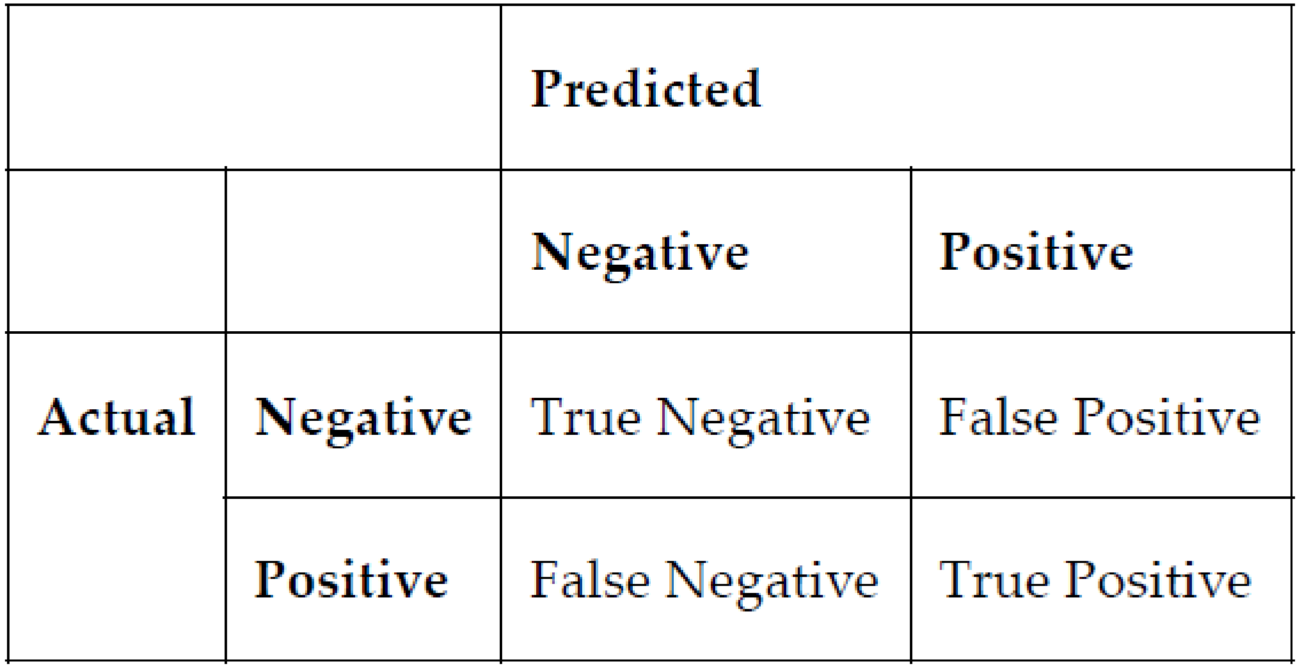
sklearn

- In order to give you a better idea of how it works, you can try the graphical example offered by Scikit-learn at https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html



Precision and recall

- To obtain the precision and recall metrics, we again start from the confusion matrix. First, we have to name each of the cells:



		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

Definition

- **TP (true positives):** These are located in the upper-left cell, containing examples that have correctly been predicted as positive ones.
- **FP (false positives):** These are located in the upper-right cell, containing examples that have been predicted as positive but are actually negative.
- **FN (false negatives):** These are located in the lower-left cell, containing examples that have been predicted as negative but are actually positive.
- **TN (true negatives):** These are located in the lower-right cell, containing examples that have been correctly predicted as negative ones.

Accuracy, precision, recall

- Using these cells, you can actually get more precise information about how your classifier works and how you can tune your model better. First, we can easily revise the accuracy formula:

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

- Then, the first informative metric is called **precision** (or **specificity**) and it is actually the accuracy of the positive cases:

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

Why choose precision?

- In the computation, only the number of true positives and the number of false positives are involved. In essence, the metric tells you how often you are correct when you predict a positive.

- Clearly, your model could **get high scores by predicting positives for only the examples it has high confidence in**. That is actually the purpose of the measure: to force models **to predict a positive class only when they are sure** and it is safe to do so.

Recall

- However, if it is in your interest also to predict as many positives as possible, then you’ll also need to watch over the recall (or coverage or sensitivity or even true positive rate) metric:

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

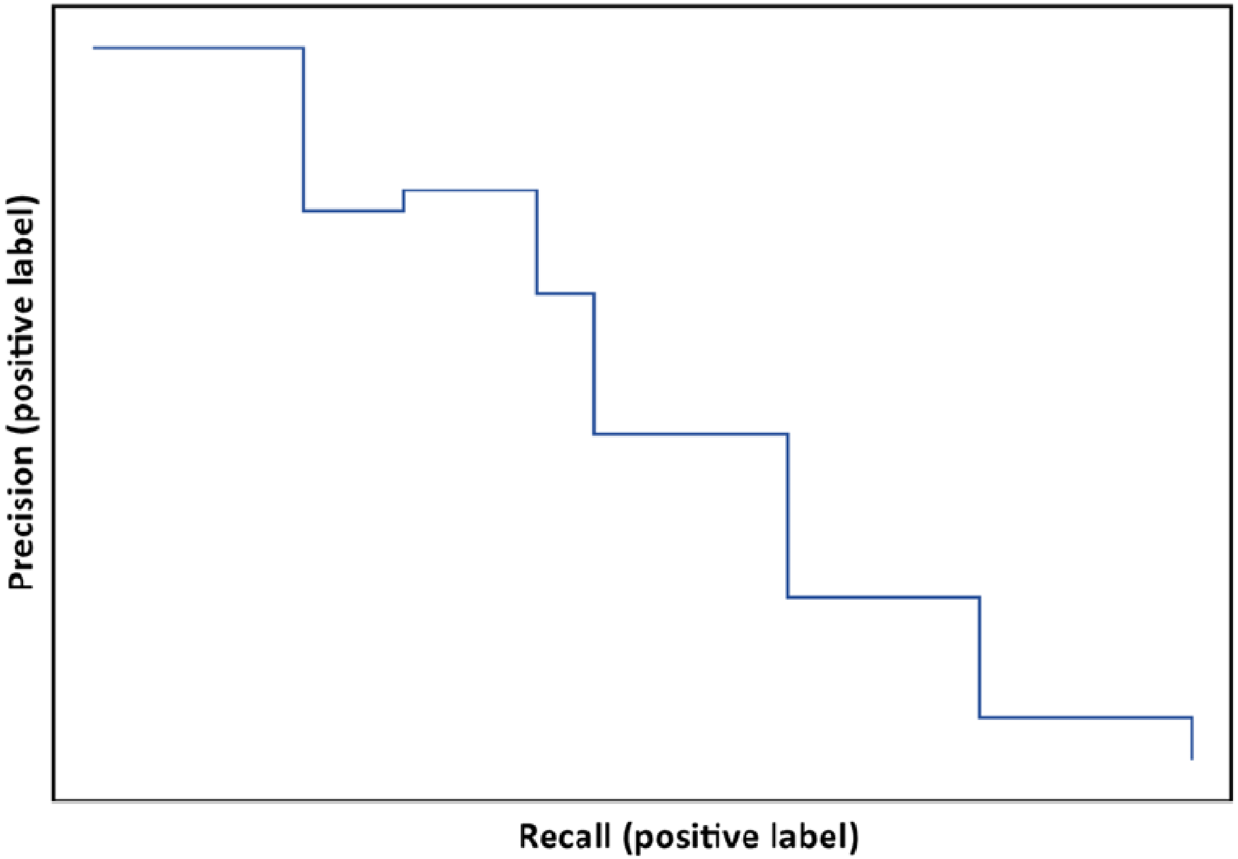
- Here, you will also need to know about false negatives.
- The interesting thing about these two metrics is that, since they are based on examples classification, and **a classification is actually based on probability** (which is usually set between the positive and negative class at the 0.5 threshold), you **can change the threshold and have one of the two metrics improved at the expense of the other.**

Precision/recall trade-off 1/

- For instance, if you increase the threshold, you will get more precision (the classifier is more confident of the prediction) but less recall. If you decrease the threshold, you get less precision but more recall.

- This is also called the **precision/recall trade-off.** The Scikit-learn website offers a simple and practical overview of this trade-off
- https://scikitlearn.org/stable/auto_examples/model_selection/plot_precision_recall.html), helping you to trace a precision/recall curve and thus understand how these two measures can be exchanged to obtain a result that better fits your needs:

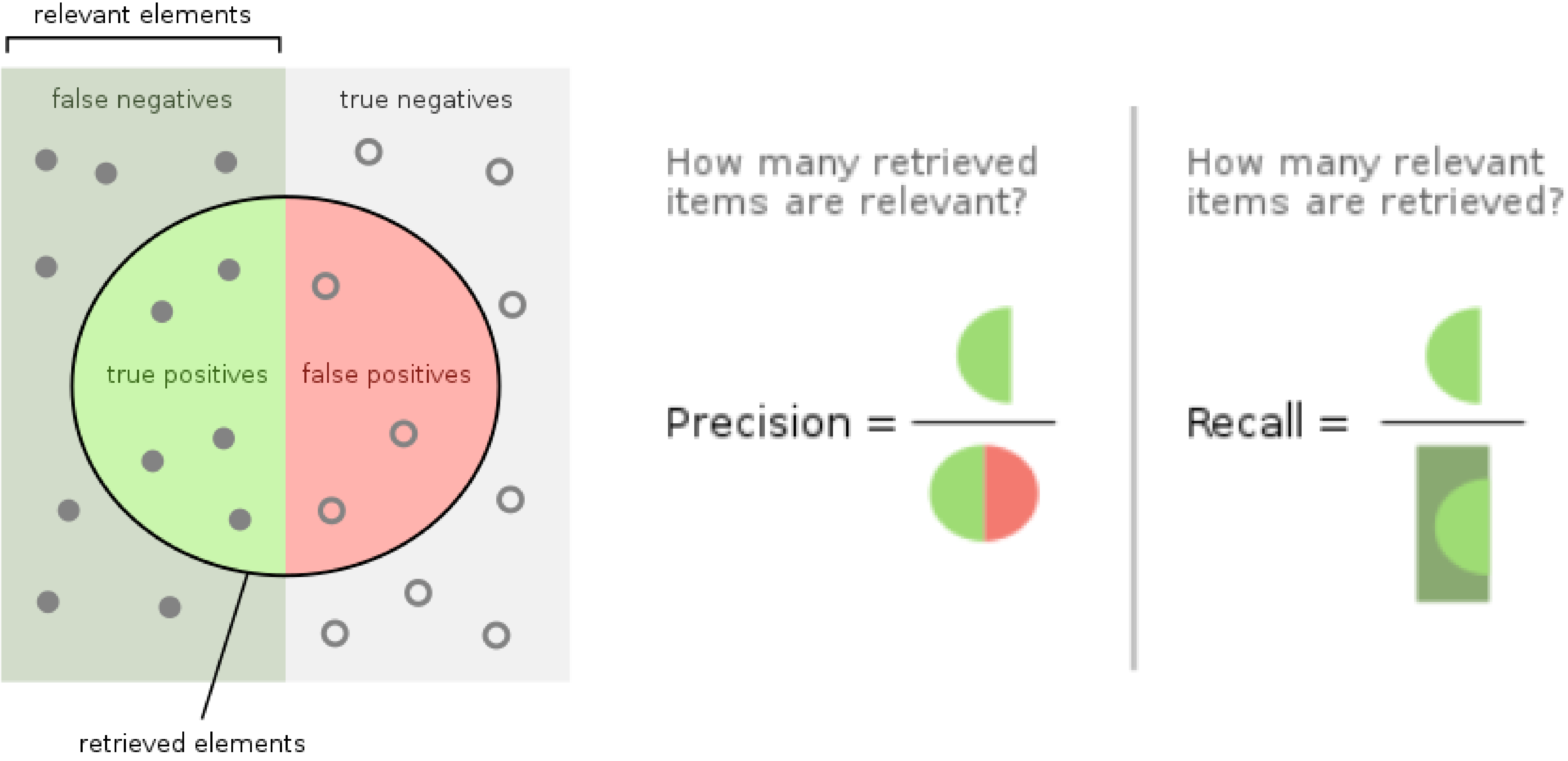
Precision/recall trade-off 2/



Average precison

- One metric associated with the precision/recall trade-off is the **average precision**.
- Average precision computes the mean precision for recall values from 0 to 1 (basically, as you vary the threshold from 1 to 0).
- **Average precision is very popular for tasks related to object detection,** which we will discuss a bit later on, but it is also very useful for classification in tabular data.
- In practice, it proves valuable when you want to monitor model performance on a very rare class (**when the data is extremely imbalanced**) in a more precise and exact way, which is **often the case with fraud detection problems**.

In picture


relevant elements
false negatives
true negatives
true positives
false positives
retrieved elements
How many retrieved items are relevant?
Precision =
How many relevant items are retrieved?
Recall =

The F1 score

- At this point, you have probably already figured out that using precision or recall as an evaluation metric is not an ideal choice because you can only optimize one at the expense of the other. For this reason, there are no Kaggle competitions that use only one
- of the two metrics. You should combine them (as in the average precision). A single metric, the F1 score, which is the harmonic mean of precision and recall, is commonly considered to be the best solution:

Formula

$$F_1 = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}}$$

$$= 2 \times \frac{Precision\ \times Recall}{Precision + Recall}$$

The F1 score 1/

- If you get a high F1 score, it is because your model has improved in precision or recall or in both. You can find a fine example of the usage of this metric in the Quora Insincere Questions Classification competition https://www.kaggle.com/c/quora-insincerequestions-classification

F-beta

- In some competitions, you also get the **F-beta** score. This is simply the weighted harmonic mean between precision and recall, and beta decides the weight of the recall in the combined score:

$$F_\beta = \frac{(1 + \ \beta^2) * (precision * recall)}{(\beta^2 * precision + recall)}$$

- Since we have already introduced the concept of threshold and classification probability, we can now discuss the log loss and ROC-AUC, both quite common classification metrics.

Log loss and ROC-AUC

- Let's start with the log loss, which is also known as cross-entropy in deep learning models. The log loss is the difference between the predicted probability and the ground truth probability

$$LogLoss = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\widehat{y_i}) + (1 - y_i)\log(1 - \widehat{y_i})]$$

- In the formula, n stands for the number of cases, y_i is the ground truth, and $\hat{y}_i$ the prediction.

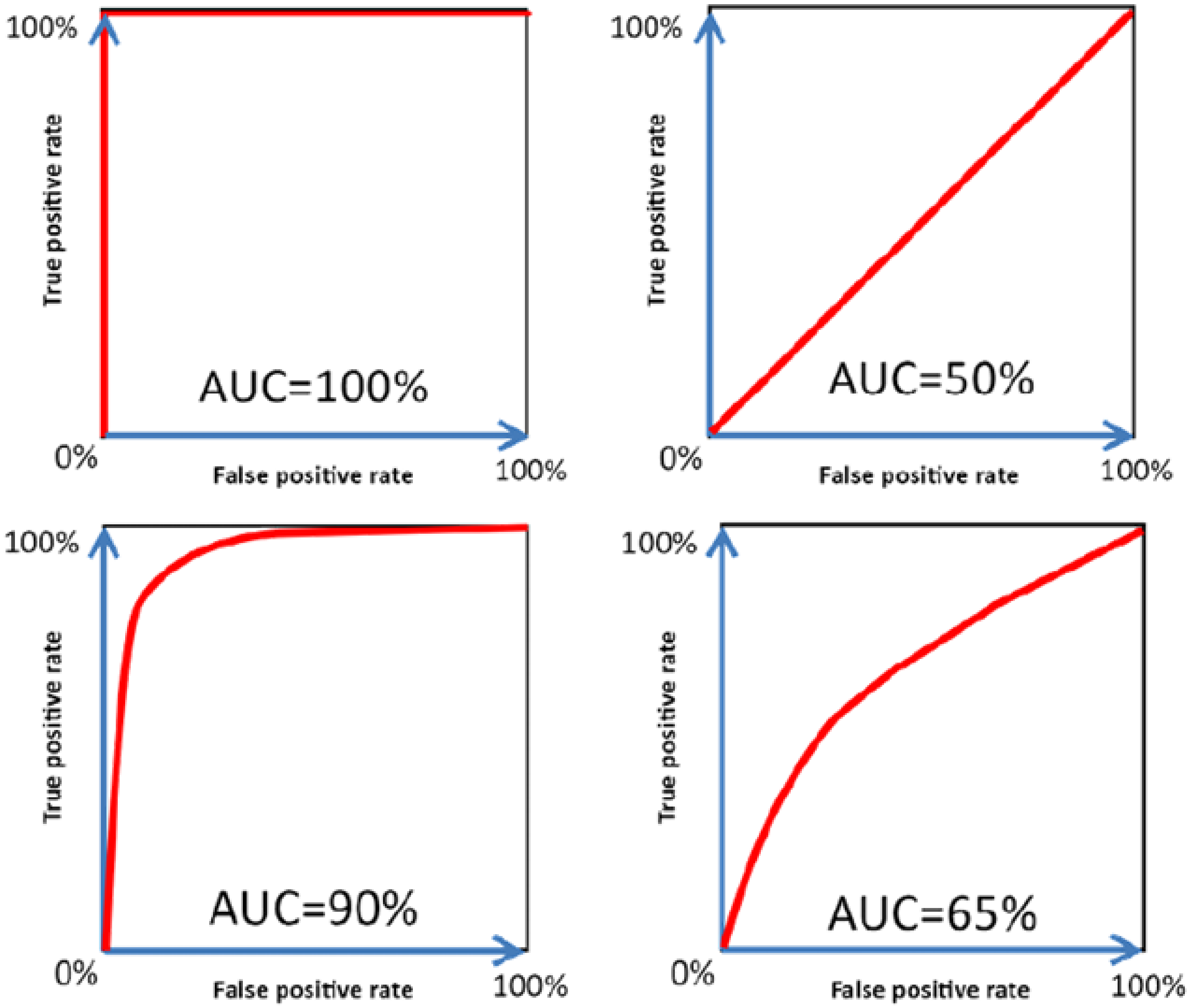
Tips

- If a competition uses the log loss, it is implied that the objective is to estimate as correctly as possible the probability of an example being of a positive class. You can actually find the log loss in quite a lot of competitions.
- We suggest you have a look, for instance, at the recent Deepfake Detection Challenge (https://www.kaggle.com/c/deepfake-detection-challenge) or at the older Quora Question Pairs (https://www.kaggle.com/c/quora-question-pairs).

ROC Curve AUC

- The ROC curve, or receiver operating characteristic curve, is a graphical chart used to evaluate the performance of a binary classifier and to compare multiple classifiers. It is the building block of the ROC-AUC metric, because the metric is simply the area delimited under the ROC curve.
- The **ROC curve** consists of the true positive rate (the recall) plotted against the false positive rate (the ratio of negative instances that are incorrectly classified as positive ones). It is equivalent to one minus the true negative rate (the ratio of negative examples that are correctly classified).

- Here are a few examples:



Different ROC curves and their AUCs

Tips

- Ideally, a ROC curve of a well-performing classifier should quickly climb up the true positive rate (recall) at low values of the false positive rate.
- A ROC-AUC between 0.9 to 1.0 is considered very good.
- **A bad classifier can be spotted by the ROC curve appearing very similar, if not identical, to the diagonal of the chart**, which represents the performance of a purely random classifier, as in the top left of the figure above; **ROC-AUC scores near 0.5 are considered to be almost random results**.

- If you are comparing different classifiers, and you are using the area under the curve (AUC), **the classifier with the higher area is the more performant one**.

Tips 1/

- If classes are **balanced**, or **not too imbalanced**, **increases in the AUC are** proportional to the effectiveness of the trained model and they can be intuitively thought of as the ability of the model to output higher probabilities for true positives.

- We also think of it **as the ability to order the examples more properly from positive to negative.**

Tips 2/

- However, when the positive class is **rare**, the AUC starts high and its increments may mean very little in terms of predicting the rare class better. As we mentioned before, in such a case, average precision is a more helpful metric.

- More details can be found in the paper: Su, W., Yuan, Y., and Zhu, M. A relationship between the average precision and the area under the ROC curve. Proceedings of the 2015 International Conference on The Theory of Information Retrieval. 2015.

Matthews correlation coefficient (MCC)

- We complete our overview of binary classification metrics with the Mathews correlation coefficient (MCC), which made its appearance in VSB Power Line Fault Detection (https://www.kaggle.com/c/vsb-power-line-fault-detection) and Bosch Production Line Performance (https://www.kaggle.com/c/bosch-productionline-performance)

- The formula for the MCC is:

$$MCC = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}}$$

Explanation

- In the above formula, TP stands for true positives, TN for true negatives, FP for false positives, and FN for false negatives. It is the same nomenclature as we met when discussing precision and recall.

- Behaving as a correlation coefficient, in other words, ranging from +1 (perfect prediction) to -1 (inverse prediction), this metric can be considered a measure of the quality of the classification even when the classes are quite imbalanced.

Neuron engineer

- In spite of its complexity, the formula can be reformulated and simplified, as demonstrated by Neuron Engineer (https://www.kaggle.com/ratthachat) in his Notebook: www.kaggle.com/ratthachat/demythifying-matthew-correlationcoefficients-mcc .
- The work done by Neuron Engineer in understanding the ratio of the evaluation metric is indeed exemplary. In fact, his reformulated MCC becomes:

$$MCC = (Pos_{precision} + Neg_{precision} - 1) * PosNegRatio$$

Formula explained 1/

- where

$$Pos_{precision} = \frac{TP}{TP + FP}$$

$$Neg_{precision} = \frac{TN}{TN + FN}$$

$$PosNegRatio = \sqrt{\frac{PosPredictionCount * NegPredictionCount}{PosLabelCount * NegLabelCount}}$$

$$PosPredictionCount = TP + FP$$

$$NegPredictionCount = TN + FN$$

Formula explained 2/

- The reformulation helps to clarify, in a more intelligible form than the original, that you can get higher performance from improving both positive and negative class precision, but that’s not enough: you also have to have positive and negative predictions in proportion to the ground truth, or your submission will be greatly penalized.

Metrics for multi-class classification

- When moving to multi-class classification, you simply use the binary classification metrics that we have just seen, applied to each class, and then you summarize them using some of the averaging strategies that are commonly used for multi-class situations.

Averaging methods

- For instance, if you want to evaluate your solution based on the F1 score, you have three possible averaging choices:

Macro averaging

- Simply calculate the *F*1 score for each class and then average all the results. In this way, each class will count as much the others, no matter how frequent its positive cases are or how important they are for your problem, resulting therefore in equal penalizations when the model doesn’t perform well with any class:

$$macro = \frac{F1_{class1}+F1_{class2}+\cdots+F1_{classN}}{N}$$

Micro averaging

- This approach will sum all the contributions from each class to compute an aggregated F1 score. It results in no particular favor to or penalization of any class, since all the computations are made regardless of each class, so it can more accurately account for class imbalances:

$$micro = F1_{class1+class2+\cdots classN}$$

Weighted macro

- As with macro averaging, you first calculate the F1 score for each class, but then you make a weighted average mean of all of them using a weight that depends on the number of true labels of each class. By using such a set of weights, you can take into account the frequency of positive cases from each class or the relevance of that class for your problem. This approach clearly favors the majority classes, which will be weighted more in the computations:

$$weighted = \ F1_{class1} * \ W_1 + F1_{class2} * W_2 + \cdots + F1_{classN} * W_n$$

$$W_1 + W_2 + \cdots + W_N = 1.0$$

Common multi-class metrics

- Common multi-class metrics that you may encounter in Kaggle competitions are:
  - **Multiclass accuracy** (weighted): Bengali.AI Handwritt*en Grapheme Classification* (https://www.kaggle.com/c/bengaliai-cv19)
  - **Multiclass log loss (MeanColumnwiseLogLoss)**: *Mechanisms of Action (MoA) Prediction* (https://www.kaggle.com/c/lish-moa/)
  - **Macro-F1** and **Micro-F1 (NQMicroF1)**: *University of Liverpool - Ion Switching* (https://www.kaggle.com/c/liverpool-ion-switching), *Human Protein Atlas Image Classification* (https://www.kaggle.com/c/human-protein-atlas-imageclassification/), *TensorFlow 2.0 Question Answering* (https://www.kaggle.com/c/tensorflow2-question-answering)
  - **Mean-F1**: *Shopee - Price Match Guarantee* (https://www.kaggle.com/c/shopeeproduct-matching/). Here, the *F*1 score is calculated for every predicted row, then averaged, whereas the Macro-F1 score is defined as the mean of classwise/ label-wise *F*1 scores.

Quadratic Weighted kappa

- Then there is also **Quadratic Weighted Kappa**, which we will explore later on as a smart evaluation metric for ordinal prediction problems. In its simplest form, the **Cohen Kappa** score, it just measures the agreement between your predictions and the ground truth. The metric was actually created for measuring **inter-annotation agreement**, but it is really versatile and has found even better uses.

Inter-Annotation agreement

- What is inter-annotation agreement? Let’s imagine that you have a labelling task: classifying some photos based on whether they contain an image of a cat, a dog, or neither. If you ask a set of people to do the task for you, you may incur some erroneous labels because someone (called the judge in this kind of task) may misinterpret a dog as a cat or vice versa. The smart way to do this job correctly is to divide the work among multiple judges labeling the same photos, and then measure their level of agreement based on the Cohen Kappa score.

Cohen kappa

- Therefore, the Cohen Kappa is devised as a score expressing the level of agreement between two annotators on a labeling (classification) problem:

$$k = (p_0 - p_e)/(1 - \ p_e)$$

- In the formula, p_0 is the relative observed agreement among raters, and p_e is the hypothetical probability of chance agreement. Using the confusion matrix nomenclature, this can be rewritten, as:

$$k = \frac{2 * (TP * TN - FN * FP)}{(TP + FP) * (FP + TN) + (TP + FN) * (FN + TN)}$$

Some comment on the formula

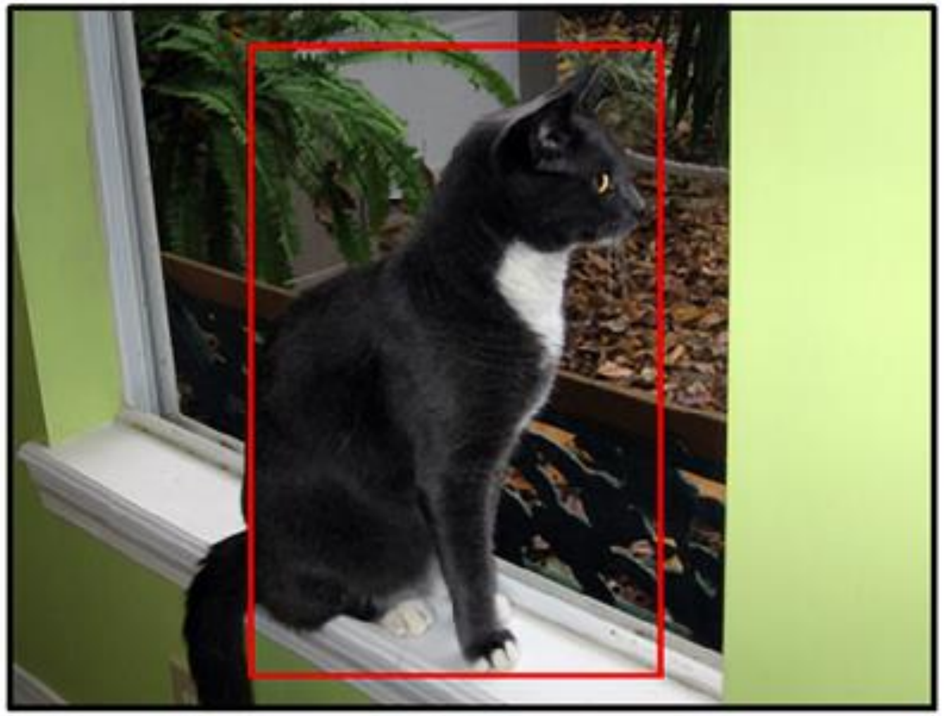
- The interesting aspect of this formula is that the score takes into account the empirical probability that the agreement has happened just by chance, so the measure has a correction for all the most probable classifications. The metric ranges from 1, meaning complete agreement, to -1, meaning the judges completely oppose each other (total disagreement).
- Values around 0 signify that agreement and disagreement among the judges is happening by mere chance. This helps you figure out if the model is really performing better than chance in most situations.

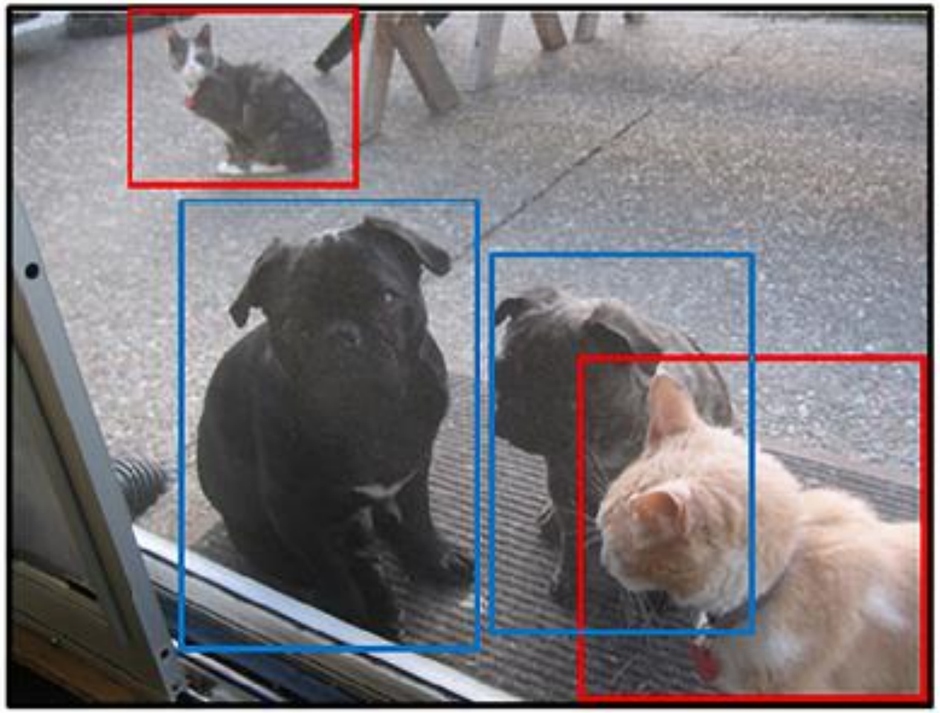
Specific case of object detection

- In case your Kaggle competition is about object detection and localization, there are slight twists compared to standard machine learning problems

Object detection

- In object detection, you don’t have to classify an image, but instead find relevant portions of a picture and label them accordingly.



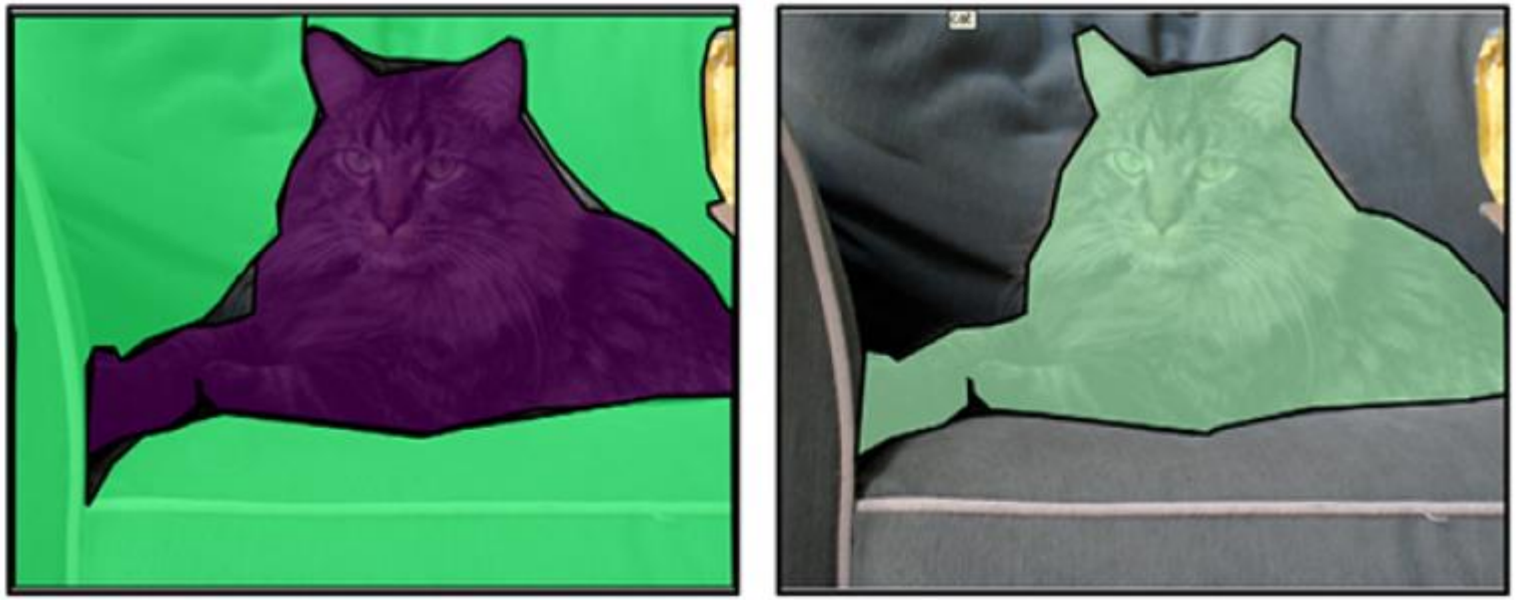


How to handle this?

- In order to describe the spatial location of an object, in object detection we use bounding boxes, which define a rectangular area in which the object lies. A bounding box is usually specified using two (x, y) coordinates: the upper-left and lower-right corners. In terms of a machine learning algorithm, finding the coordinates of bounding boxes corresponds to applying a regression problem to multiple targets. However, you probably won't frame the problem from scratch but rely on pre-built and often pre-trained models such as Mask R-CNN
  - (https://arxiv.org/abs/1703.06870), RetinaNet
  - (https://arxiv.org/abs/2106.05624v1), FPN
  - (https://arxiv.org/abs/1612.03144v2), YOLO
  - (https://arxiv.org/abs/1506.02640v1), Faster R-CNN
  - (https://arxiv.org/abs/1506.01497v1), or SDD
  - (https://arxiv.org/abs/1512.02325).

What about image segmentation?

- In **segmentation**, you instead have a classification at the pixel level



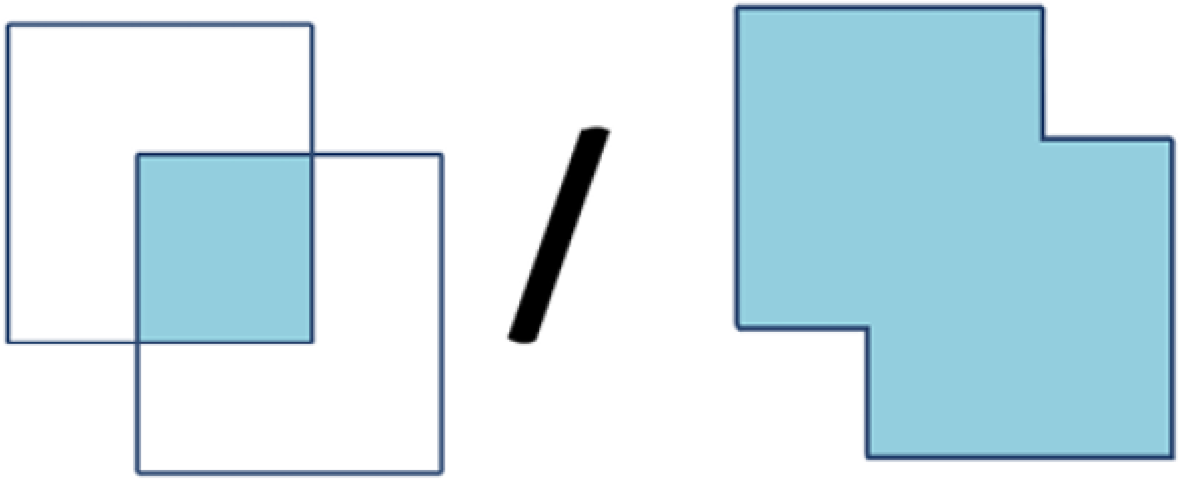
Metrics recommended

- There are two metrics that are used much more, especially in competitions: the intersection over union and the dice coefficient.

Intersection over union (IoU)

- The intersection over union (IoU) is also known as the Jaccard index. When used in segmentation problems, using IoU implies that you have two images to compare: one is your prediction and the other is the mask revealing the ground truth, which is usually a binary matrix where the value 1 stands for the ground truth and 0 otherwise.
- In the case of multiple objects, you have multiple masks, each one labeled with the class of the object. When used in object detection problems, you have the boundaries of two rectangular areas (those of the prediction and the ground truth), expressed by the coordinates of their vertices. For each classified class, you compute the area of overlap between your prediction and the ground truth mask, and then you divide this by the area of the union between your prediction and the ground truth, a sum that takes into account any overlap.
- In this way, you are proportionally penalized both if you predict a larger area than what it should be (the denominator will be larger) or a smaller one (the numerator will be smaller):

Visually

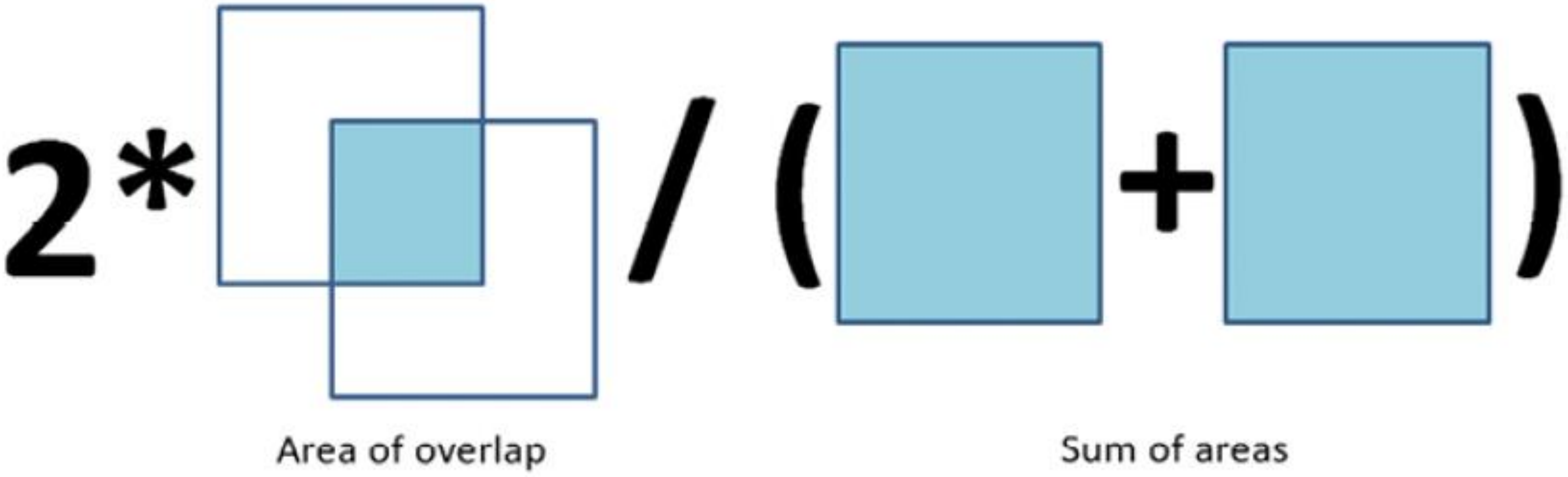


Examples

- Here are some examples of competitions where IoU has been used:
  - *TGS Salt Identification Challenge* (https://www.kaggle.com/c/tgssalt-identification-challenge/) with Intersection Over Union Object Segmentation
  - iMaterialist (Fashion) 2019 at FGVC6 (https://www.kaggle.com/c/imaterialist-fashion-2019-FGVC6) with Intersection Over Union Object Segmentation With Classification
  - Airbus Ship Detection Challenge (https://www.kaggle.com/c/airbus-ship-detection) with Intersection Over Union Object Segmentation Beta

Dice

- The other useful metric is the Dice coefficient, which is the area of overlap between the prediction and ground truth doubled and then divided by the sum of the prediction and ground truth areas:



Difference with Jaccardi

- In this case, with respect to the Jaccardi index, you do not take into account the overlap of the prediction with the ground truth in the denominator. Here, the expectation is that, as you maximize the area of overlap, you predict the correct area size. Again, you are penalized if you predict areas larger than you should be predicting. In fact, the two metrics are positively correlated and they produce almost the same results for a single classification problem.

- The differences actually arise when you are working with multiple classes. In fact, both with IoU and the Dice coefficient, when you have multiple classes you average the result of all of them. However, in doing so, the IoU metric tends to penalize the overall average more if a single class prediction is wrong, whereas the Dice coefficient is more lenient and tends to represent the average performance.

Going further

- IoU and Dice constitute the basis for all the more complex metrics in segmentation and object detection. By choosing an appropriate threshold level for IoU or Dice (usually 0.5), you can decide whether or not to confirm a detection, therefore a classification. At this point, you can use previously discussed metrics for classification, such as precision, recall, and *F*1, such as is done in popular object detection and segmentation challenges such as Pascal VOC (http://host.robots.ox.ac.uk/pascal/VOC/voc2012) or COCO (https://cocodataset.org).

Metrics for multi-label classification and recommendation problems

- Recommender systems are one of the most popular applications of data analysis and machine learning, and there are quite a few competitions on Kaggle that have used the recommendation approach. For instance, the Quick, Draw! Doodle Recognition Challenge was a prediction evaluated as a recommender system. Some other competitions on Kaggle, however, truly strived to build effective recommender systems (such as Expedia Hotel Recommendations: https://www.kaggle.com/c/expediahotelrecommendations)
- and RecSYS, the conference on recommender systems (https://recsys.acm.org/), even hosted one of its yearly contests on Kaggle (RecSYS2013: https://www.kaggle.com/c/yelp-recsys-2013).

MAP@{K})

- **Mean Average Precision at K (MAP@{K})** is typically the metric of choice for evaluating the performance of recommender systems, and it is the most common metric you will encounter on Kaggle in all the competitions that try to build or approach a problem as a recommender system.

- There are also some other metrics, such as the **precision at k, or P@K**, and the average precision at k, or AP@K, which are loss functions, in other words, computed at the level of each single prediction.

- Understanding how they work can help you better understand the MAP@K and how it can perform both in recommendations and in multi-label classification.

What about multi-label classifications

- In fact, analogous to recommender systems, **multi-label classifications imply that your model outputs a series of class predictions**.

- Such results could be evaluated using some average of some binary classification metrics (such as in *Greek Media Monitoring Multilabel Classification (WISE 2014)*, which used the mean *F*1 score: https://www.kaggle.com/c/wise-2014) as well as metrics that are more typical of recommender systems, such as MAP@K.
- In the end, you can deal with both recommendations and multi-label predictions as ***ranking tasks***, which translates into a set of ranked suggestions in a recommender system and into a set of labels (without a precise order) in multi-label classification. **MAP@{K}**

MAP@{K}

- MAP@K is a complex metric and it derives from many computations. In order to understand the MAP@K metric fully, let’s start with its simplest component, the precision at k (P@K). In this case, since the prediction for an example is a ranked sequence of predictions (from the most probable to the least), the function takes into account only the top k predictions, then it computes how many matches it got with respect to the ground truth and divides that number by k. In a few words, it is quite similar to an accuracy measure averaged over k predictions.

- A bit more complex in terms of computation, but conceptually simple, the average precision at k (AP@K) is the average of P@K computed over all the values ranging from 1 to k. In this way, the metric evaluates how well the prediction works overall, using the top prediction, then the top two predictions, and so on until the top k predictions.

- Finally, MAP@K is the mean of the AP@K for the entire predicted sample, and it is a metric because it comprises all the predictions in its evaluation.

MAP@5

- Here is the MAP@5 for instance you can find in the *Expedia Hotel Recommendations* competition (https://www.kaggle.com/c/expedia-hotel-recommendations):

$$MAP@5 = \frac{1}{|U|} \sum_{u=1}^{|U|} \sum_{k=1}^{\min(5,n)} P(k)$$

- In the formula, is the number of user recommendations, P(k) is the precision at cutoff k, and n is the number of predicted hotel clusters (you could predict up to 5 hotels for each recommendation).

Optimizing evaluation metrics

- Summing up what we have discussed so far, an objective function is a function inside your learning algorithm that measures how well the algorithm’s internal model is fitting the provided data. The objective function also provides feedback to the algorithm in order for it to improve its fit across successive iterations. Clearly, since the entire algorithm’s efforts are recruited to perform well based on the objective function, if the Kaggle evaluation metric perfectly matches the objective function of your algorithm, you will get the best results.

In practice!

- Unfortunately, this is not frequently the case. Often, the evaluation metric provided can only be approximated by existing objective functions. Getting a good approximation, or striving to get your predictions performing better with respect to the evaluation criteria, is the secret to performing well in Kaggle competitions. When your objective function does not match your evaluation metric, you have a few alternatives:

Strategies

- Modify your learning algorithm and have it incorporate an objective function that matches your evaluation metric, though this is not possible for all algorithms (for instance, algorithms such as LightGBM and XGBoost allow you to set custom objective functions, but most Scikit-learn models don’t allow this).
- Tune your model’s hyperparameters, choosing the ones that make the result shine the most when using the evaluation metric.
- Post-process your results so they match the evaluation criteria more closely. For instance, you could code an optimizer that performs transformations on your predictions (probability calibration algorithms are an example, and we will discuss them at the end of the chapter).

In reality

- Having the competition metric incorporated into your machine learning algorithm is really the most effective method to achieve better predictions, though only a few algorithms can be hacked into using the competition metric as your objective function.
- The second approach is therefore the more common one, and many competitions end up in a struggle to get the best hyperparameters for your models to perform on the evaluation metric.

Leverage Kaggle

- If you already have your evaluation function coded, then doing the right crossvalidation or choosing the appropriate test set plays the lion share. If you don’t have the coded function at hand, you have to first code it in a suitable way, following the formulas provided by Kaggle.

Tips 1/

- Invariably, doing the following will make the difference:
  - Looking for all the relevant information about the evaluation metric and its coded function on a search engine Browsing through the most common packages (such as Scikit-learn: https://scikitlearn.org/stable/modules/model_evaluation.html#model-evaluation or TensorFlow: https://www.tensorflow.org/api_docs/python/tf/keras/losses)
  - Browsing GitHub projects (for instance, Ben Hammer’s Metrics project: https://github.com/benhamner/Metrics)
  - Asking or looking around in the forums and available Kaggle Notebooks (both for the current competition and for similar competitions)
  - In addition, as we mentioned before, querying the Meta Kaggle dataset (https://www.kaggle.com/kaggle/meta-kaggle) and looking in the Competitions table will help you find out which other Kaggle competitions used that same evaluation metric, and immediately provides you with useful code and ideas to try out

Example: Custom metrics and custom objective functions

- As a first option when your objective function does not match your evaluation metric, we learned above that you can solve this by creating your own custom objective function, but that only a few algorithms can easily be modified to incorporate a specific objective function.
- The good news is that the few algorithms that allow this are among the most effective ones in Kaggle competitions and data science projects. Of course, creating your own custom objective function may sound a little bit tricky, but it is an incredibly rewarding approach to increasing your score in a competition.
- For instance, there are options to do this when using gradient boosting algorithms such as XGBoost, CatBoost, and LightGBM, as well as with all deep learning models based on TensorFlow or PyTorch.

Change the metric

- You can find great tutorials for custom metrics and objective functions in TensorFlow and PyTorch here:
  - https://towardsdatascience.com/custom-metrics-in-keras-and-howsimple-they-are-to-use-in-tensorflow2-2-6d079c2ca279
  - https://petamind.com/advanced-keras-custom-loss-functions/
  - https://kevinmusgrave.github.io/pytorch-metriclearning/extend/losses/

- These will provide you with the basic function templates and some useful suggestions about how to code a custom objective or evaluation function.

Example

- If you want just to get straight to the custom objective function you need, you can try this Notebook by RNA
  - (https://www.kaggle.com/bigironsphere):
  - https://www.kaggle.com/bigironsphere/loss-function-librarykeras-pytorch/notebook
- It contains a large range of custom loss functions for both TensorFlow and PyTorch that have appeared in different competitions.

GBDT case

- If you need to create a custom loss in LightGBM, XGBoost, or CatBoost, as indicated in their respective documentation, you have to code a function that takes as inputs the prediction and the ground truth, and that returns as outputs the gradient and the hessian.

- You can consult this post on Stack Overflow for a better understanding of what a gradient and a hessian are:
- https://stats.stackexchange.com/questions/231220/how-tocompute-the-gradient-and-hessian-of-logarithmic-lossquestion-is-based

Code implementation perspective

- From a code implementation perspective, all you have to do is to create a function, using closures if you need to pass more parameters beyond just the vector of predicted labels and true labels.

- Here is a simple example of a focal loss (a loss that aims to heavily weight the minority class in the loss computations as described in Lin, T-Y. et al. Focal loss for dense object detection: https://arxiv.org/abs/1708.02002) function that you can use as a model for your own custom functions:

Code your own loss

```
from scipy.misc import derivative
import xgboost as xgb

def focal_loss(alpha, gamma):
    def loss_func(y_pred, y_true):
        a, g = alpha, gamma

        def get_loss(y_pred, y_true):
            p = 1 / (1 + np.exp(-y_pred))

        loss = (-(a * y_true + (1 - a) * (1 - y_true)) * ((1 - (y_true * p + (1 - y_true) * (1 - p))) ** g) * (
                    y_true * np.log(p) + (1 - y_true) * np.log(1 - p)))
        return loss
        partial_focal = lambda y_pred: get_loss(y_pred, y_true)
        grad = derivative(partial_focal, y_pred, n=1, dx=1e-6)
        hess = derivative(partial_focal, y_pred, n=2, dx=1e-6)
        return grad, hess
    return loss_func

xgb = xgb.XGBClassifier(objective=focal_loss(alpha=0.25, gamma=1))
```

Comment

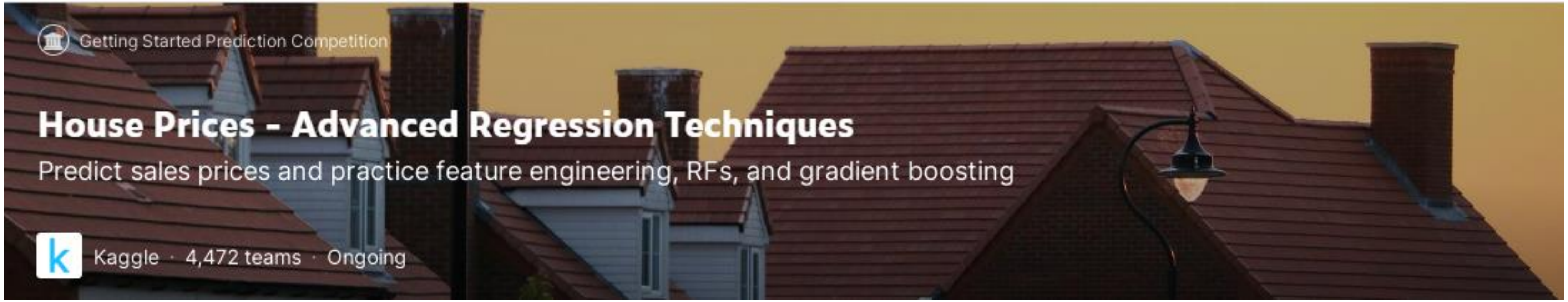
- In the above code snippet, we have defined a new cost function, **focal_loss**, which is then fed into an XGBoost instance's object parameters.
- The example is worth showing because the focal loss requires the specification of some parameters in order to work properly on your problem (alpha and gamma).
- The more simplistic solution of having their values directly coded into the function is not ideal, since you may have to change them systematically as you are tuning your model
- Instead, in the proposed function, when you input the parameters into the focal_loss function, **they reside in memory** and they are referenced by the loss_func function that is returned to XGBoost. The returned cost function, therefore, will work, referring to the alpha and gamma values that you have initially instantiated.

Summary

- In this chapter, we have discussed evaluation metrics in Kaggle competitions. First, we explained how an evaluation metric can differ from an objective function. We also remarked on the differences between regression and classification problems.
- For each type of problem, we analyzed the most common metrics that you can find in a Kaggle competition. After that, we discussed the metrics that have never previously been seen in a competition and that you won’t likely see again. Finally, we explored and studied different common metrics, giving examples of where they have been used in previous Kaggle competitions.
- We then proposed a few strategies for optimizing an evaluation metric. In particular, we recommended trying to code your own custom cost function and provided suggestions on possible useful post-processing steps.
- You should now have grasped the role of an evaluation metric in a Kaggle competition. You should also have a strategy to deal with every common or uncommon metric, by retracing past competitions and by gaining a full understanding of the way a metric works. In the next chapter, we are going to discuss how to use evaluation metrics and properly estimate the performance of your Kaggle solution by means of a validation strategy.

Example of a regression challenge

- House Prices - Advanced Regression Techniques



Getting Started Prediction Competition

House Prices - Advanced Regression Techniques

Predict sales prices and practice feature engineering, RFs, and gradient boosting

Kaggle · 4,472 teams · Ongoing

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques/overview/tutorials