IASD M2 at Paris Dauphine

Become a Kaggle Master

3: Validation

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 Kaggle competition, in the heat of modeling and submitting results, it may seem enough to take at face value the results you get back from the leaderboard. In the end, you may think that what counts in a competition is your ranking. This is a common error that is made repeatedly in competitions. In actual fact, you won't know what the actual leaderboard (the private one) looks like until after the competition has closed, and trusting the public part of it is not advisable because it is **quite often misleading.**

- In this chapter, we will introduce you to the importance of validation in data competitions

Contents

- You will learn about:
  1. What overfitting is and how a public leaderboard can be misleading
  2. The dreadful shake-ups
  3. The different kinds of validation strategies
  4. Adversarial validation
  5. How to spot and leverage leakages
  6. What your strategies should be when choosing your final submission

Why avoiding overfitting?

- Monitoring your performances when modeling and distinguishing when overfitting happens is a key competency not only in data science competitions but in all data science projects. Validating your models properly is one of the most important skills that you can learn from a Kaggle competition and that you can resell in the professional world

Snooping on the leaderboard

- As we previously described, in each competition, Kaggle divides the test set into a public part, which is visualized on the ongoing leaderboard, and a private part, which will be used to calculate the final scores. These test parts are usually randomly determined (although in time series competitions, they are determined based on time) and the entire test set is released without any distinction made between **public** and **private**.

How is a submission done?

- Therefore, a submission derived from a model will cover the entire test set, but only the public part will immediately be scored, leaving the scoring of the private part until after the competition has closed.

Consequences:

- Given this, three considerations arise:
  - In order for a competition to work properly, training data and test data should be from the same distribution. Moreover, the private and public parts of the test data should resemble each other in terms of distribution.
  - Even if the training and test data are apparently from the same distribution, the lack of sufficient examples in either set could make it difficult to obtain aligned results between the training data and the public and private test data.
  - The public test data should be regarded as a holdout test in a data science project: to be used only for final validation. Hence, it should not be queried much in order to avoid what is called **adaptive overfitting**, which implies a model that works well on a specific test set but underperforms on others.

Leaderboard fallacy

- Keeping in mind these three considerations is paramount to understanding the dynamics of a competition.
- In most competitions, there are always quite a few questions in the discussion forums about how the training, public, and private test data relate to each other
- It is quite common to see submissions of hundreds of solutions that have only been evaluated based on their efficacy on the public leaderboard

Shake-ups 1/

- It is also common to hear discussions about shake-ups that revolutionize the rankings. They are, in fact, a rearranging of the final rankings that can disappoint many who previously held better positions on the public leaderboard. Anecdotally, shake-ups are commonly attributed to differences between the training and test set or between the private and public parts of the test data. They are measured ex ante based on how competitors have seen their expected local scores correlate with the leaderboard feedback and ex post by a series of analyses based on two figures:

Shake-ups 2/

- A general shake-up figure based on mean(abs(private_rankpublic_rank)/number_of_teams)
- A top leaderboard shake-up figure, taking into account only the top 10% of public ranks

- https://www.kaggle.com/jtrotman/metakaggle-competition-shake-up

Shakeup

Title	Entrants	Team Count	Shakeup	Worst Drop	Median Delta	Best Rise
Human Protein Atlas - Single Cell Classification	11838	757	0.374439	-717	6	478
Generative Dog Images	7378	921	0.370199	-903	191	562
Forecast Eurovision Voting	612	22	0.363636	-19	0	20
Mayo Clinic - STRIP AI	10623	888	0.345959	-867	27	667
RSNA-MICCAI Brain Tumor Radiogenomic Classification	15612	1555	0.341757	-1527	85	1376
RSNA Intracranial Hemorrhage Detection	12942	432	0.333548	-431	11	374
Mercari Price Suggestion Challenge	26788	2380	0.322280	-2334	550	893
Jigsaw Rate Severity of Toxic Comments	12826	2301	0.312512	-2056	49	2017
The Hewlett Foundation: Automated Essay Scoring	6058	153	0.311760	-135	5	121
World Cup 2010 - Confidence Challenge	287	63	0.310406	-55	-1	59
Intel & MobileODT Cervical Cancer Screening	9404	260	0.307012	-240	25	188
The Big Data Combine Engineered by BattleFin	1807	424	0.301086	-375	12	345
BCI Challenge @ NER 2015	1734	260	0.297722	-185	20	186
HuBMAP - Hacking the Kidney	16615	1216	0.277322	-1182	218	661
Data Science Bowl 2017	22264	394	0.261795	-361	6	349

Findings

- There is adaptive overfitting; in other words, public standings usually do not fully hold in the unveiled private leaderboard.
- Most of the shake-ups are due overcrowded rankings where competitors are too near to each other, and any slight change in the performance in the private test sets causes major changes in the rankings as well as to random fluctuations.
- Shake-ups happen when the training set is very small or the training data is not **independent and identically distributed** (**i.i.d.**).

What to do?

- Since this is **quite a common** and persistent problem, we suggest a strategy more sophisticated than simply following the public leaderboard:
  - Always build **reliable cross-validation systems** for local scoring.
  - Always try to **control non-i.i.d distributions** using the best validation scheme dictated by the situation. Unless clearly stated in the description of the competition, it is not an easy task to spot non-i.i.d. distributions, but you can get hints from discussion or by experimenting using **stratified validation schemes** (when stratifying according to a certain feature, the results improve decisively, for instance).
  - **Correlate local scoring with the public leaderboard** in order to figure out whether or not they go in the same direction.
  - Test **using adversarial validation**, revealing whether or not the test distribution is similar to the training data.
  - Make your solutions more robust **using ensembling**, especially if you are working with small datasets.

What to learn?

- If you think about a competition carefully, you can imagine it as a huge system of experiments. **Whoever can create the most systematic and efficient way to run these experiments wins**.
- In fact, **in spite of all your theoretical knowledge**, you will be in competition with the hundreds or thousands of data professionals **who have more or less the same competencies as you**.
- In addition, they will be using exactly **the same data as you** and roughly **the same tools** for learning from the data (TensorFlow, PyTorch, Scikit-learn, and so on). Some **will surely have better access to computational resources**, although the availability of Kaggle Notebooks and generally decreasing cloud computing prices mean the gap is no longer so wide.

Why some wins?

- Consequently, if you look at differences in **knowledge, data, models, and available computers**, you won't find many discriminating factors between you and the other competitors that could explain huge performance differences in a competition.
- Yet, some participants consistently outperform others, implying there is some underlying success factor.

What makes the difference?

- In interviews and meet-ups, some Kagglers describe this success factor as “grit,” some others as “trying everything,” some others again as a “willingness to put everything you have into a competition.” These may sound a bit obscure and magic.
- Instead, we call it **systematic experimentation**.
- In our opinion, the key to successful participation resides in **the number of experiments you conduct and the way you run all of them**.

What does it mean?

- The **more experiments you undertake**, **the more chances you will have to crack the problem better than other participants**.
- This number certainly depends on a few factors, such as the time you have available, your computing resources (the faster the better, but as we previously mentioned, this is not such a strong differentiator per se), your team size, and their involvement in the task.
- This aligns with the commonly reported **grit and engagement** as keys for success.

But do experiments carefully!

- However, these are not the only factors affecting the result. You have to take into account that the way you run your experiments also has an impact. Fail fast and learn from it is an important factor in a competition. Of course, you need to reflect carefully both when you fail and when you succeed in order to learn something from your experiences, or **your competition will just turn into a random sequence of attempts** in the hope of picking the right solution.

Get a proper validation strategy!

- Therefore, ceteris paribus, having a proper validation strategy is the great discriminator between successful Kaggle competitors and those who just overfit the leaderboard and end up in lower-than expected rankings after a competition.

Validation

- Therefore, ceteris paribus, having a proper validation strategy is the great discriminator between successful Kaggle competitors and those who just overfit the leaderboard and end up in lower-than expected rankings after a competition.

- **Validation is the method** you use to **correctly evaluate the errors that your model produces** and to measure how its performance improves or decreases based on your experiments.

Validation is often overlooked!

- Generally, the impact of choosing proper validation is too often overlooked in favor of more quantitative factors, such as having the latest, most powerful GPU or a larger team producing submissions.

- Nevertheless, if you count only on the firepower of experiments and their results on the leaderboard, it will be like “throwing mud at the wall and hoping something will stick”

Rules of thumb

- Though the temptation to submit your top public leaderboard models may be high, always consider your own validation scores. For your final submissions, depending on the situation and whether or not you trust the leaderboard, choose your best model based on the leaderboard and your best based on your local validation results.

- If you don’t trust the leaderboard (especially when the training sample is small or the examples are non-i.i.d.), submit models that have two of the best validation scores, picking two very different models or ensembles. In this way, you will reduce the risk of choosing solutions that won’t perform on the private test set.

Why is so important to validate a model?

- Having pointed out the importance of having a method of experimenting, what is left is all a matter of the practicalities of validation. In fact, when you model a solution, you take a series of interrelated decisions:
  - 1. How to process your data
  - 2. What model to apply
  - 3. How to change the model's architecture (especially true for deep learning models)
  - 4. How to set the model's hyperparameters
  - 5. How to post-process the predictions
- Even if the public leaderboard is perfectly correlated with the private one, the limited number of daily submissions (a limitation present in all competitions) prevents you from even scratching the surface of possible tests.
- **Having a proper validation system tells you beforehand**

Bias variance trade-off

- You also hear about the capacity or expressiveness of a model as a matter of bias and variance. In this case, the bias and variance of a model refer to the predictions, but the underlying principle is strictly related to the expressiveness of a model. Models can be reduced to mathematical functions that map an input (the observed data) to a result (the predictions).
- Some mathematical functions are more complex than others, in the number of internal parameters they have and in the ways they use them:

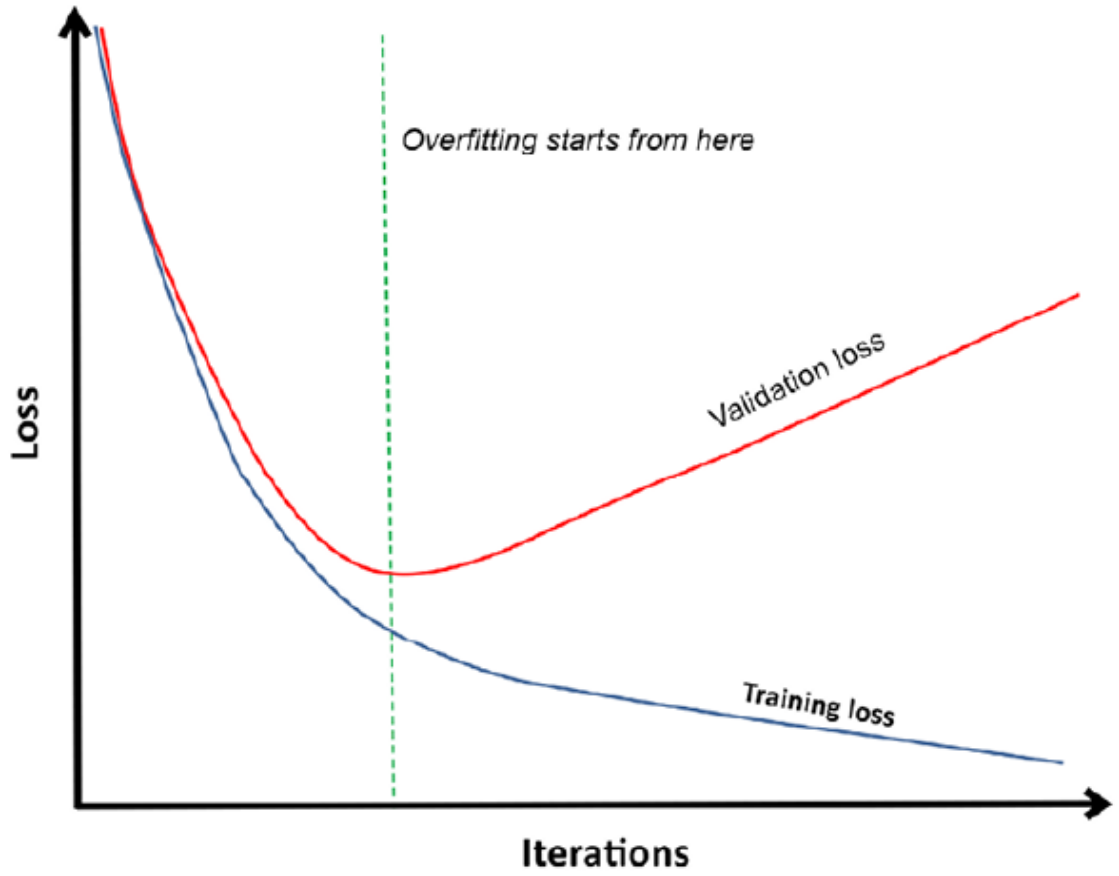
What it means?

- If the mathematical function of a model is not complex or expressive enough to capture the complexity of the problem you are trying to solve, we talk of bias, because your predictions will be limited ("biased") by the limits of the model itself.
- If the mathematical function at the core of a model is too complex for the problem at hand, we have a variance problem, because the model will record more details and noise in the training data than needed and its predictions will be deeply influenced by them and become erratic.

Variance is at stake now!

- Nowadays, given the advances in machine learning and the available computation resources, the problem is always due to variance, since deep neural networks and gradient boosting, the most commonly used solutions, often have a mathematical expressiveness that exceeds what most of the problems you will face need in order to be solved.

Difference between train and test



Try different splitting strategies

- As previously discussed, the validation loss is based on a data sample that is not part of the training set. It is an empirical measure that tells you how good your model is at predicting, and a more correct one than the score you get from your training, which will tell you mostly how much your model has memorized the training data patterns. Correctly choosing the data sample you use for validation constitutes your validation strategy.

Validation is about splitting

- To summarize the strategies for validating your model and measuring its performance correctly, you have a couple of choices:
- The first choice is to work with a holdout system, incurring the risk of not properly choosing a representative sample of the data or overfitting to your validation holdout.
- The second option is to use a probabilistic approach and rely on a series of samples to draw your conclusions on your models.

- Among the probabilistic approaches, you have cross validation, leave-one-out (LOO), and bootstrap. Among the cross-validation strategies, there are different nuances depending on the sampling strategies you take based on the characteristic of your data (simple random sampling, stratified sampling, sampling by groups, time sampling).

What all these strategies have in common?

- What all these strategies have in common is that they are sampling strategies. It means that they help you to infer a general measure (the performance of your model) based on a small part of your data, randomly selected. Sampling is at the root of statistics and it is not an exact procedure because, based on your sampling method, your available data, and the randomness of picking up certain cases as part of your sample, you will experience a certain degree of error.

Basic train test sampling

- The first strategy that we will analyze is the train-test split. In this strategy, you sample a portion of your training set (also known as the holdout) and you use it as a test set for all the models that you train using the remaining part of the data.
- The great advantage of this strategy is that it **is very simple**: you pick up a part of your data and you check your work on that part. You usually split the data **80/20 in favor of the training partition**.
- Scikit-learn, it is implemented in the train_test_split function

Limitation

- When you **have large amounts of data**, you can expect that the test data you extract is similar to (representative of) the original distribution on the entire dataset. However, since the extraction process is based on randomness, you always have **the chance of extracting a non-representative sample**.
- In particular, the **chance increases if the training sample you start from is small**. Comparing the extracted holdout partition using **adversarial validation** (more about this in a few sections) can help you to make sure you are evaluating your efforts in a correct way.

- In addition, to ensure that your test sampling is representative, especially with regard to how the training data relates to the target variable, you can use **stratification**, which ensures that the proportions of certain features are respected in the sampled data. You can use **the stratify parameter** in the train_test_split function and provide an array containing the class distribution to preserve.

Probabilistic evaluation

- We have to remark that, even if you have a representative holdout available, sometimes a simple train-test split is not enough for ensuring a correct tracking of your efforts in a competition.
- In fact, as you keep checking on this test set, you may drive your choices to some kind of adaptation overfitting (in other words, erroneously picking up the noise of the training set as signals), as happens when you frequently evaluate on the public leaderboard.
- For this reason, a **probabilistic evaluation**, though more computationally expensive, is more suited for a competition.

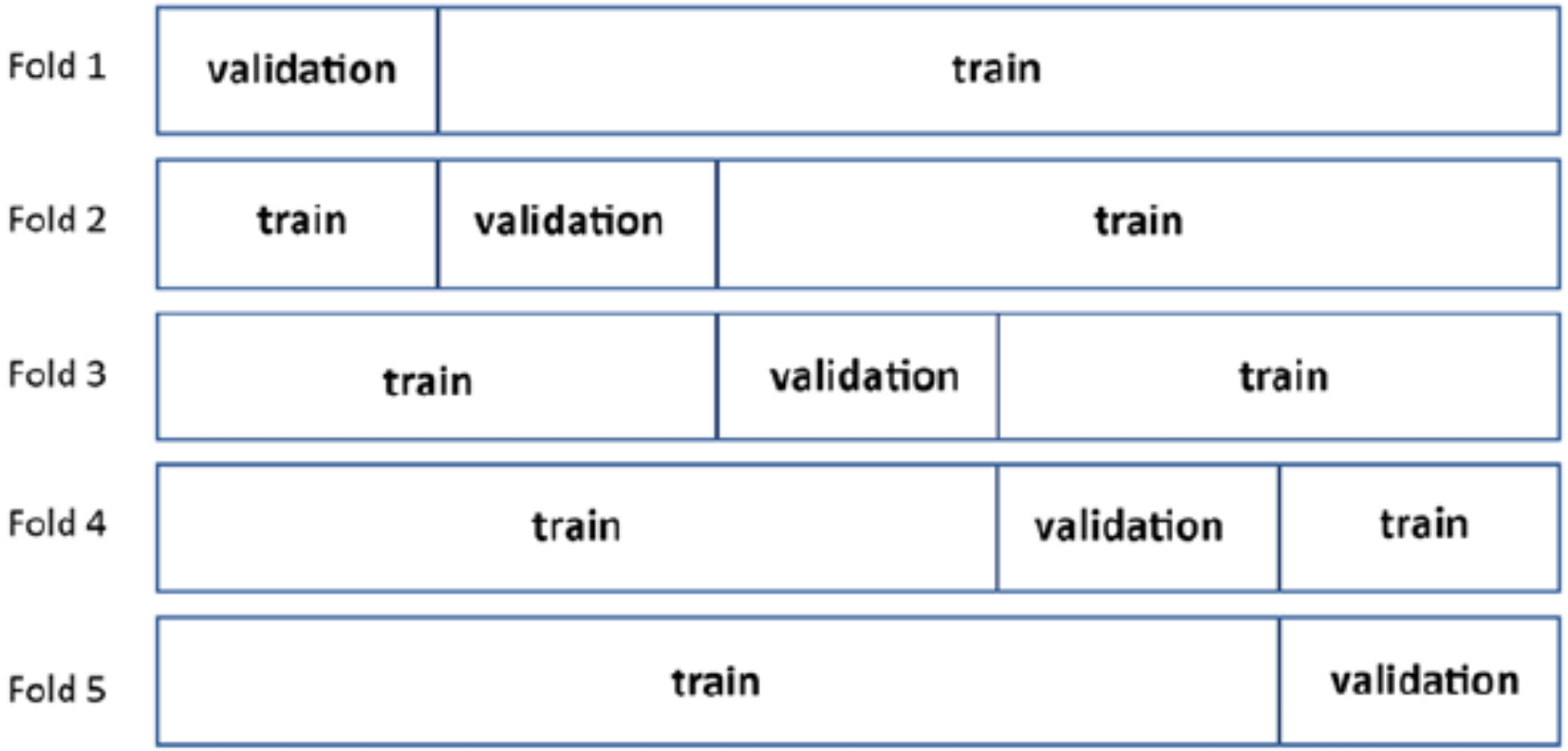
Probabilistic evaluation methods

- Probabilistic evaluation of the performance of a machine learning model is based on the statistical properties of a sample from a distribution. By sampling, you create a smaller set of your original data that is expected to have the same characteristics.
- In addition, what is left untouched from the sampling constitutes a sample in itself, and it is also expected to have the same characteristics as the original data. By training and testing your model on this sampled data and repeating this procedure a large number of times, you are basically creating a statistical estimator measuring the performance of your model.
- Probabilistic estimators naturally **require more computations** than a simple train-test split, but they **offer more confidence** that you are correctly estimating the right measure: the general performance of your model.

k-fold cross-validation

- The most used probabilistic validation method is k-fold cross validation, which is recognized as having the ability to correctly estimate the performance of your model on unseen test data drawn from the same distribution.
- There are quite a few different variations of k-fold cross-validation, but the simplest one, which is implemented in the KFold function in Scikit-learn, is based on the splitting of your available training data into k partitions. After that, for k iterations, one of the k partitions is taken as a test set while the others are used for the training of the model.
- The k validation scores are then averaged and that averaged score value is the k-fold validation score, which will tell you the estimated average model performance on any unseen data. The standard deviation of the scores will inform you about the uncertainty of the estimate.

5-fold validation scheme



How a 5-fold validation scheme is structured

What k? 1/

- One important aspect of the *k*-fold cross-validation score you have to keep in mind is that it estimates the average score of a model trained on the same quantity of data as *k - 1* folds. If, afterward, you train your model on all your data, the previous validation estimate no longer holds. As *k* approaches the number *n* of examples, you have an increasingly correct estimate of the model derived on the full training set, yet, due to the growing correlation between the estimates you obtain from each fold, you will lose all the probabilistic estimates of the validation. In this case, you'll end up having a number showing you the performance of your model on your training data (which is still a useful estimate for comparison reasons, but it won't help you in correctly estimating the generalization power of your model).

- When you reach *k = n*, you have the LOO validation method, which is useful when you have a few cases available. The method is mostly an unbiased fitting measure since it uses almost all the available data for training and just one example for testing. Yet it is not a good estimate of the expected performance on unseen data. Its repeated tests over the whole dataset are highly correlated with each other and the resulting LOO metric represents more the performance of the model on the dataset itself than the performance the model would have on unknown data.

What k? 2/

- The correct k number of partitions to choose is decided based on a few aspects relative to the data you have available:
- The smaller the k (the minimum is 2), the smaller each fold will be, and consequently, the more bias in learning there will be for a model trained on k - 1 folds: your model validated on a smaller k will be less well-performing with respect to a model trained on a larger k.
- The higher the k, the more the data, yet the more correlated your validation estimates: you will lose the interesting properties of k-fold cross-validation in estimating the performance on unseen data.

- Commonly, k is set to 5, 7, or 10, more seldom to 20 folds. We usually regard k = 5 or k = 10 as a good choice for a competition, with the latter using more data for each training (90% of the available data), and hence being more suitable for figuring out the performance of your model when you retrain on the full dataset.

What k? 3/

- When deciding upon what k to choose for a specific dataset in a competition, we find it useful to reflect your goals:
  - If your purpose is performance estimation, you need models with low bias estimates (which means no systematic distortion of estimates). You can achieve this by using a higher number of folds, usually between 10 and 20.
  - If your aim is parameter tuning, you need a mix of bias and variance, so it is advisable to use a medium number of folds, usually between 5 and 7.
  - Finally, if your purpose is just to apply variable selection and simplify your dataset, you need models with low variance estimates (or you will have disagreement). Hence, a lower number of folds will suffice, usually between 3 and 5.

What k? 4/

- When the size of the available data is quite large, you can safely stay on the lower side of the suggested bands.
- Secondly, if you are just aiming for performance estimation, consider that the more folds you use, the fewer cases you will have in your validation set, so the more the estimates of each fold will be correlated. Beyond a certain point, increasing *k* renders your crossvalidation estimates less predictive of unseen test sets and more representative of an estimate of how well-performing your model is on your training set. This also means that, with more folds, you can get the perfect out-of-fold prediction for stacking purposes, as we will explain in detail in our lecture about, *Ensembling with Blending and Stacking Solutions*.

k-fold variations

- Since it is based on random sampling, k-fold can provide unsuitable splits when:
- You have to **preserve the proportion of small classes**, both at a target level and at the level of features. This is typical when your **target is highly imbalanced**. Typical examples are spam datasets (because spam is a small fraction of the normal email volume) or any credit risk dataset where you have to predict the not-so-frequent event of a defaulted loan.
- You have to **preserve the distribution of a numeric variable**, both at a target level and at the level of features. This is **typical of regression problems where the distribution is quite skewed or you have heavy, long tails**. A common example is house price prediction, where you have a consistent small portion of houses on sale that will cost much more than the average house.
- Your **cases are non-i.i.d,** in particular when dealing with time series forecasting.

Solutions

- In the first two scenarios, the solution is the stratified k-fold, where the sampling is done in a controlled way that preserves the distribution you want to preserve.
- If you need to preserve the distribution of a single class, you can use **StratifiedKFold** from Scikitlearn, using a stratification variable, usually your target variable but also any other feature whose distribution you need to preserve. The function will produce a set of indexes that will help you to partition your data accordingly. You can also obtain the same result with a numeric variable, after having discretized it, using pandas.cut or Scikit-learn’s KBinsDiscretizer .

Scikit-multilearn

- You can find a solution in the **Scikit-multilearn** package (http://scikit.ml/), in particular, the Iterative Stratification command that helps you to control the order (the number of combined proportions of multiple variables) that you want to Preserve
- http://scikit.ml/api/skmultilearn.model_selection.iterative_stratification.html
- It implements the algorithm explained by the following papers:
  - Sechidis, K., Tsoumakas, G., and Vlahavas, I. (2011). *On the stratification of multi-label data*. *Machine Learning and Knowledge Discovery in Databases, 145-158*. http://lpis.csd.auth.gr/publications/sechidisecmlpkdd-2011.pdf
  - Szymański, P. and Kajdanowicz, T.; *Proceedings of the First International Workshop on Learning with Imbalanced Domains*: *Theory and Applications*, PMLR 74:22-35, 2017. http://proceedings.mlr.press/v74/szyma%C5%84ski17a.html

Other imbalance strategy

- You can use imlearn https://imbalanced-learn.org/stable/
- It contains many algorithms in the following categories, including SMOTE
  - Under-sampling the majority class(es).
  - Over-sampling the minority class.
  - Combining over- and under-sampling.
  - Create ensemble balanced sets.

What about regressions?

- You can actually make good use of stratification even when your problem is not a classification, but a regression. Using stratification in regression problems helps your regressor to fit during cross validation on a similar distribution of the target (or of the predictors) to the one found in the entire sample. In these cases, in order to have StratifiedKFold working correctly, you have to use a discrete proxy for your target instead of your continuous target.

- The first, simplest way of achieving this is to use the pandas cut function and divide your target into a large enough number of bins, such as 10 or 20:

Regression stratification

```
import pandas as pd
y_proxy = pd.cut(y_train, bins=10, labels=False)
```

In order to determine the number of bins to be used, one can use use Sturges' rule based on the number of examples available, and provide that number to the pandas cut function (https://www.kaggle.com/abhishek/step-1-create-folds)

```
import numpy as np
bins = int(np.floor(1 + np.log2(len(X_train))))
```

Alternative

- An alternative approach is to focus on the distributions of the features in the training set and aim to reproduce them. This requires the use of cluster analysis (an unsupervised approach) on the features of the training set, thus excluding the target variable and any identifiers, and then using the predicted clusters as strata. You can see an example in this Notebook
- (https://www.kaggle.com/lucamassaron/are-you-doingcross-validation-the-best-way), where first a PCA (principal component analysis) is performed to remove correlations, and then a k-means cluster analysis is performed. You can decide on the number of clusters to use by running empirical tests.

What about non i.i.d.?

- Proceeding with our discussion of the cases where k-fold can provide unsuitable splits, things get tricky in the third scenario, when you have non-i.i.d. data, such as in the case of some grouping happening among examples.
- The problem with non-i.i.d. examples is that the features and target are correlated between the examples (hence it is easier to predict all the examples if you know just one example among them). In fact, if you happen to have the same group divided between training and testing, your model may learn to distinguish the groups and not the target itself, producing a good validation score but very bad results on the leaderboard.
- The solution here is to use **GroupKFold** : by providing a grouping variable, you will have the assurance that each group will be placed either in the training folds or in the validation ones, but never split between the two.

Discovering non i.i.d.

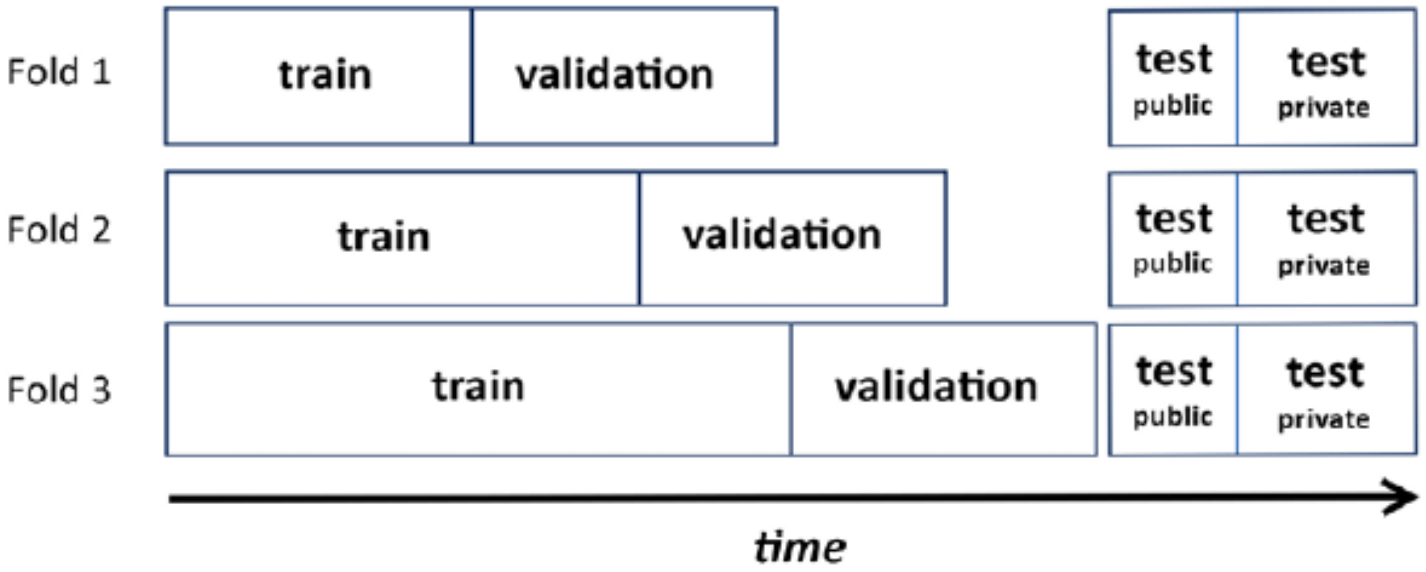
- Discovering groupings in the data that render your data non-i.i.d. is actually not an easy task to accomplish. Unless stated by the competition problem,
- you will have to rely on your ability to investigate the data (using unsupervised learning techniques, such as cluster analysis) and the domain of the problem. For instance, if your data is about mobile telephone usage, you may realize that some examples are from the same user by noticing sequences of similar values in the features.

What about time series?

- Time series analysis presents the same problem, and since data is non-i.i.d., you cannot validate by random sampling because you will mix different time frames and later time frames could bear traces of the previous ones (a characteristic called auto-correlation in statistics). In the most basic approach to validation in time series, you can use a training and validation split based on time, as illustrated below:
- Your validation capabilities will be limited, however, since your validation will be anchored to a specific time. For a more complex approach, you can use time split validation, TimeSeriesSplit , a provided by the Scikit-learn package

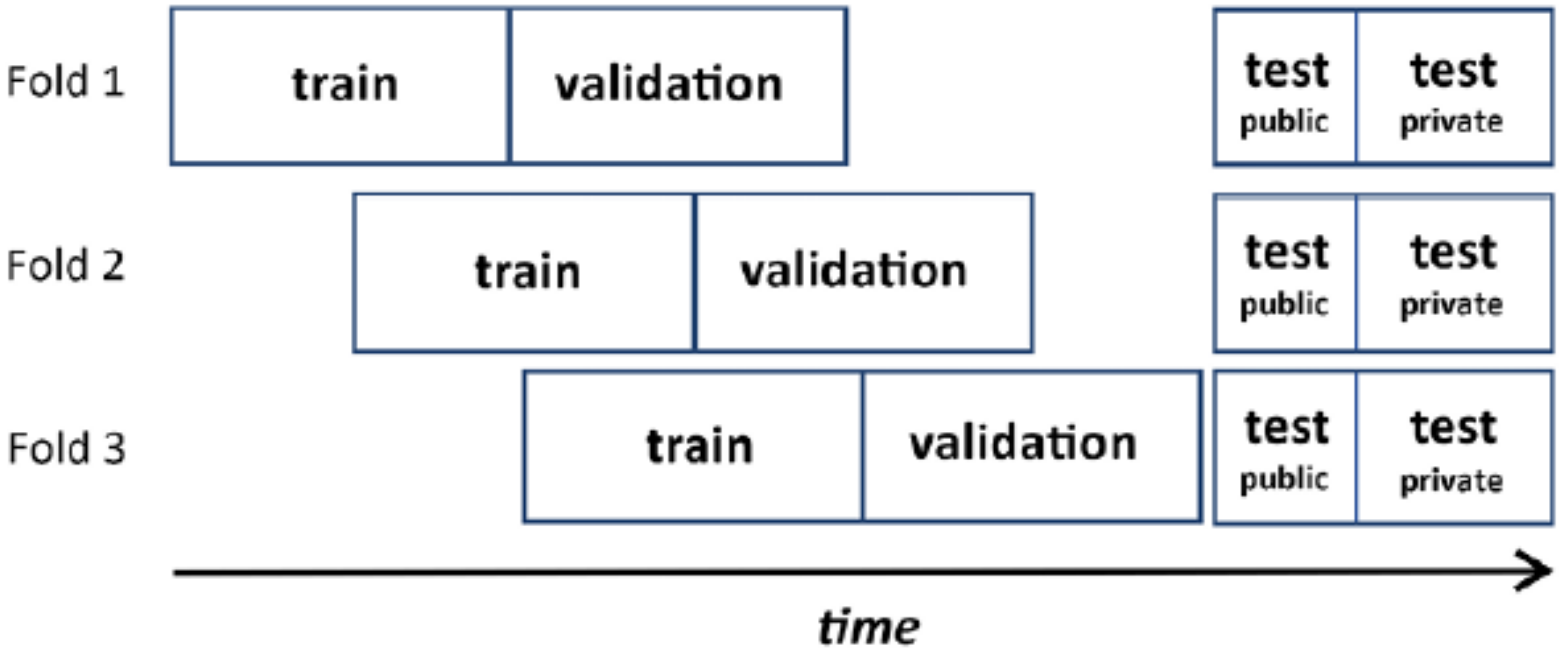
TimeSeriesSplit

- In the case of the training timeframe, the TimeSeriesSplit function can help you to set your training data so it involves all the past data before the test timeframe, or limit it to a fixed period lookback (for instance, always using the data from three months before the test timeframe for training).



Variation

- You can instead see how the strategy changes if you stipulate that the training set has a fixed lookback



What to conclude?

- In our experience, going by a fixed lookback helps to provide a fairer evaluation of time series models since you are always counting on the same training set size.
- **By instead using a growing training set size over time, you confuse the effects of your model performance across time slices with the decreasing bias in your model** (since more examples mean less bias).
- Finally, remember that TimeSeriesSplit can be set **to keep a predefined gap between your training and test time**. This is extremely useful when you are told that the test set is a certain amount of time in the future (for instance, a month after the training data) and you want to test if your model is robust enough to predict that far into the future.

Nested cross-validation

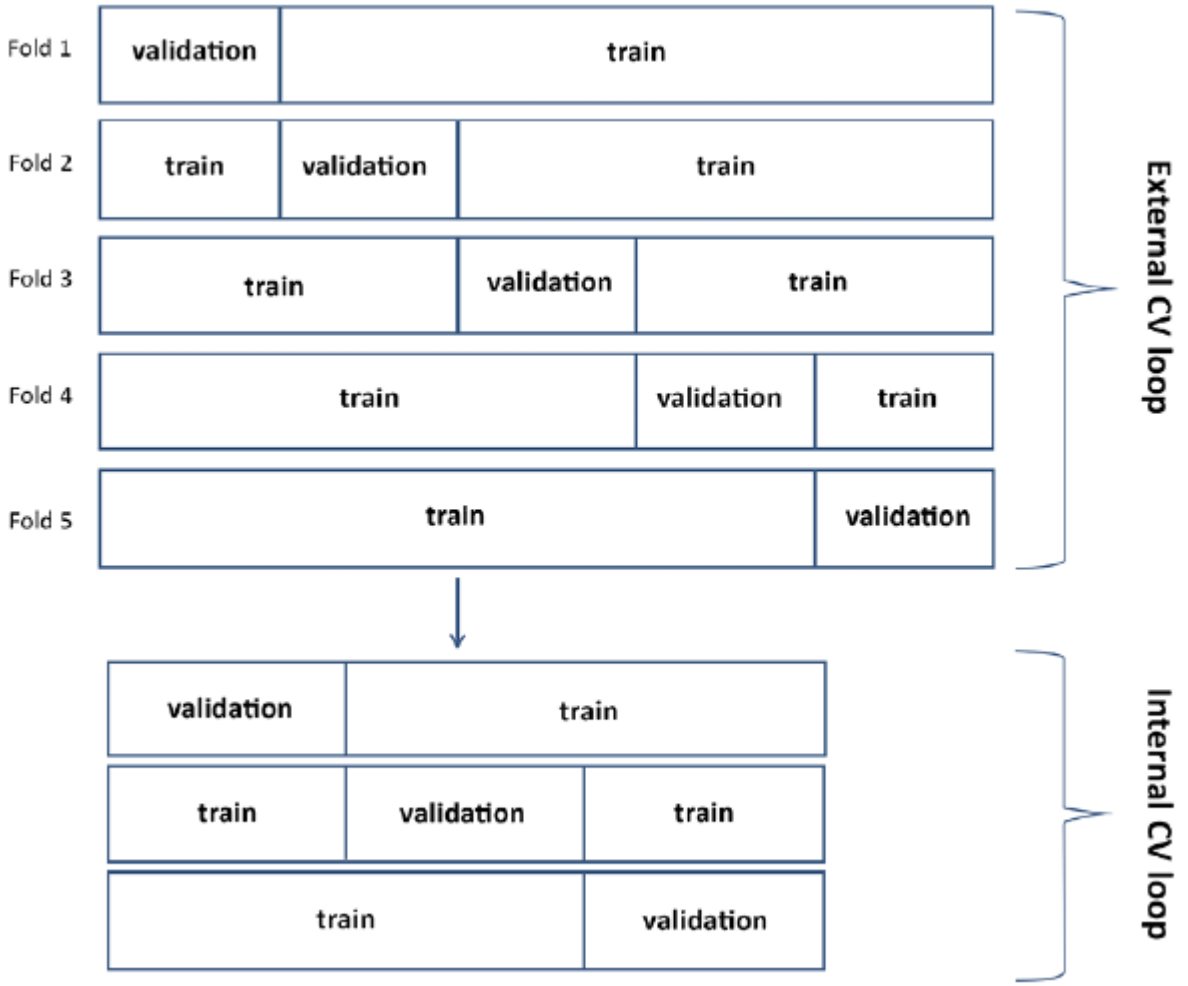
- At this point, it is important to introduce nested cross-validation. Up to now, we have only discussed testing models with respect to their final performance, but often you also need to test their intermediate performance when tuning their hyperparameters.
- In fact, you cannot test how certain model parameters work on your test set and then use the same data in order to evaluate the final performance. Since you have specifically found the best parameters that work on the test set, your evaluation measure on the same test set will be too optimistic; on a different test set, you will probably not obtain the exact same result. In this case, you have to distinguish between a validation set, which is used to evaluate the performance of various models and hyperparameters, and a test set, which will help you to estimate the final performance of the model.

Nested cross validation

- If you are using a test-train split, this is achieved by splitting the testpart into two new parts. The usual split is 70/20/10 for training, validation, and testing, respectively (but you can decide differently). If you are using cross-validation, you need nested cross-validation; that is, you do cross-validation based on the split of another rossvalidation. Essentially, you run your usual cross-validation, but when you have to evaluate different models or different parameters, you run cross-validation based on the fold split.

What it means? 1/

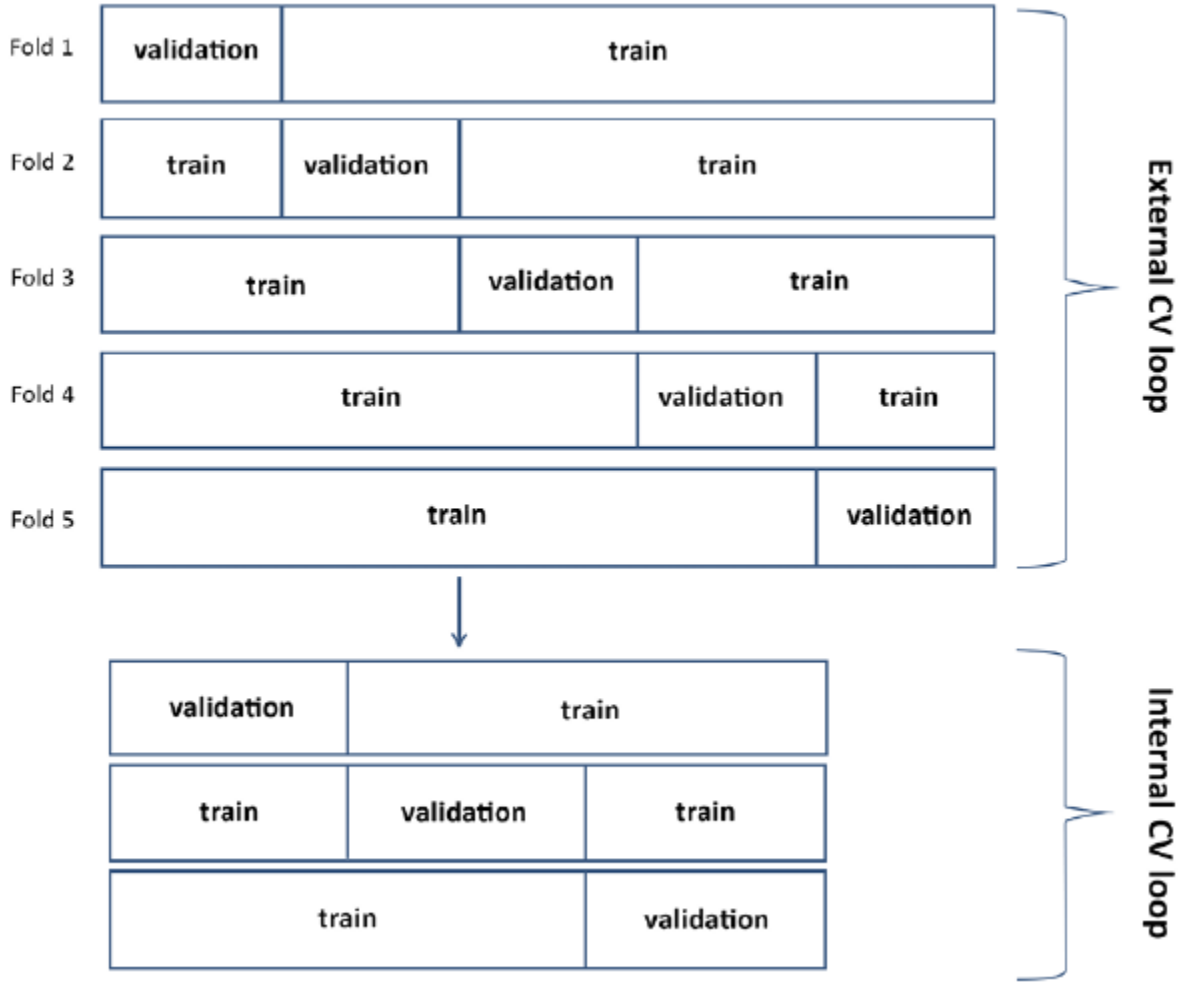
- Let us see on an example the internal and external cross-validation structure.



What it means? 2/

- In external part, you determine the portion of the data used to test your evaluation metric.

- In internal part, which is fed by the training data from the external part, you arrange training/validation splits in order to evaluate and optimize specific model choices, such as deciding which model or hyperparameter values to pick:



What to prefer?

- This approach has the advantage of making your test and parameter search fully reliable, but in doing so you incur a couple of problems: A reduced training set, since you first split by cross-validation, and then you split again
- More importantly, it requires a huge amount of model building: if you run two nested 10-fold cross-validations, you'll need to run 100 models
- Especially for this last reason, some Kagglers tend to ignore nested cross-validation and risk some adaptive fitting by using the same cross-validation for both model/parameter search and performance evaluation, or using a fixed test sample for the final evaluation.

What do we recommend?

- In our experience, this approach can work as well, though it may result in overestimating model performance and overfitting if you are generating out-of-fold predictions to be used for successive modeling (something we are going to discuss in the next section). We always suggest you try the most suitable methodology for testing your models. If your aim is to correctly estimate your model's performance and reuse its predictions in other models, remember that using nested cross-validation, whenever possible, can provide you with a less overfitting solution and could make the difference in certain competitions.

Producing out-of-fold predictions (OOF)

- An interesting application of cross-validation, besides estimating your evaluation metric performance, is producing test predictions and out-of-fold predictions. In fact, as you train on portions of your training data and predict on the remaining ones, you can:
- **Predict on the test set**: The average of all the predictions is often more effective than re-training the same model on all the data: this is an ensembling technique related to blending, which will be dealt when talking about *Ensembling with Blending and Stacking Solutions.*
- **Predict on the validation set**: In the end, you will have predictions for the entire training set and can re-order them in the same order as the original training data. These predictions are commonly referred to as out-of-fold (OOF) predictions and they can be extremely useful.

OOF predictions

- The first use of OOF predictions is to estimate your performance since you can compute your evaluation metric directly on the OOF predictions. The performance obtained is different from the cross validated estimates (based on sampling); it doesn’t have the same probabilistic characteristics, so it is not a valid way to measure generalization performance, but it can inform you about the performance of your model on the specific set you are training on.

- A second use is to produce a plot and visualize the predictions against the ground truth values or against other predictions obtained from different models. This will help you in understanding how each model works and if their predictions are correlated.

What about meta-features?

- The last use is to create meta-features or meta-predictors. This will also be fully explored in *Ensembling with Blending and Stacking*, but it is important to remark on now, as OOF predictions are a byproduct of cross-validation and they work because, during cross-validation, your model is always predicting on examples that it has not seen during training time.
- Since every prediction in your OOF predictions has been generated by a model trained on different data, these predictions are unbiased and you can use them without any fear of overfitting (though there are some caveats that will be discussed in the chapter about tabular competitions).

OOF predictions

- Generating OOF predictions can be done in two ways:
  - By coding a procedure that stores the validation predictions into a prediction vector, taking care to arrange them in the same index position as the examples in the training data
  - By using the Scikit-learn function cross_val_predict , which will automatically generate the OOF predictions for you

- We will be seeing this second technique in action when we look at adversarial validation later in this lecture.

Subsampling

- There are other validation strategies aside from k-fold cross validation, but they do not have the same generalization properties.
- We have already discussed LOO, which is the case when k = n (where n is the number of examples). Another choice is subsampling. Subsampling is similar to k-fold, but you do not have fixed folds; you use as many as you think are necessary (in other words, take an educated guess). You repetitively subsample your data, each time using the data that you sampled as training data and the data that has been left unsampled for your validation.
- By averaging the evaluation metrics of all the subsamples, you will get a validation estimate of the performances of your model.

ShuffleSplit

- Since you are systematically testing all your examples, as in k-fold, you actually need quite a lot of trials to have a good chance of testing all of them. For the same reason, some cases may be tested more than others if you do not apply enough subsamples. You can run this sort of validation using **ShuffleSplit** from Scikit-learn.

The bootstrap

- Finally, another option is to try the bootstrap, which has been devised in statistics to conclude the error distribution of an estimate; for the same reasons, it can be used for performance estimation.
- The bootstrap requires you to draw a sample, with replacement, that is the same size as the available data. At this point, you can use the bootstrap in two different ways:
  - As in statistics, you can bootstrap multiple times, train your model on the samples, and compute your evaluation metric on the training data itself. The average of the bootstraps will provide your final evaluation.
  - Otherwise, as in subsampling, you can use the bootstrapped sample for your training and what is left not sampled from the data as your test set.

Rule of thumb

- In our experience, the first method of calculating the evaluation metric on the bootstrapped training data, often used in statistics for linear models in order to estimate the value of the model's coefficients and their error distributions, is much less useful in machine learning.
- This is because many machine learning algorithms tend to overfit the training data, hence you can never have a valid metric evaluation on your training data, even if you bootstrap it.

Efro Efron and Tibshirani feedback

- For this reason, Efron and Tibshirani (see Efron, B. and Tibshirani, R. Improvements on cross-validation: the 632+ bootstrap method. Journal of the American Statistical Association 92.438 (1997): 548-560.) proposed the 632+ estimator as a final validation metric.

- At first, they proposed a simple version, called the 632 bootstrap:

$$Err_{.632} = 0.368 * err_{fit} + 0.632 * err_{bootstrap}$$

Interpretation

- In this formula, given your evaluation metric *err*, err_{fit} is your metric computed on the training data and $err_{bootstrap}$ is the metric computed on the bootstrapped data.
- However, in the case of an overfitted training model, , err_{fit} would tend to zero, rendering the estimator not very useful. Therefore, they developed a second version of the 632+ bootstrap:

$$Err_{.632} + (1 - w) * err_{fit} + w * err_{bootstrap}$$

parameters

- Where *w* is such that:

$$w = \frac{0.632}{1 - 0.632R}$$

$$R = \frac{err_{bootstrap} - err_{fit}}{\gamma - err_{fit}}$$

- Here you have a new parameter, γ, which is the no-information error rate, estimated by evaluating the prediction model on all possible combinations of targets and predictors. Calculating is indeed intractable, as discussed by the developers of Scikit-learn (https://github.com/scikit-learn/scikitlearn/issues/9153).
- Given the limits and intractability of using the bootstrap as in classical statistics for machine learning applications, you can instead use the second method, getting your evaluation from the examples left not sampled by the bootstrap.

In practice

- In this form, the bootstrap is an alternative to cross-validation, but as with subsampling, it requires **building many more models and testing them than for cross-validation**.
- However, it makes sense to know about such alternatives in case your cross-validation is showing too high a variance in the evaluation metric and you need more intensive checking through testing and re-testing.

implementation

```
import random
def Bootstrap(n, n_iter=3, random_state=None):
    """
    Random sampling with replacement cross-validation generator.
    For each iter a sample bootstrap of the indexes [0, n) is
    generated and the function returns the obtained sample
    and a list of all the excluded indexes.
    """
    if random_state:
        random.seed(random_state)
        for j in range(n_iter):
            bs = [random.randint(0, n-1) for i in range(n)]
            out_bs = list({i for i in range(n)} - set(bs))
            yield bs, out_bs
```

Conclusion for bootstrap

- In conclusion, the bootstrap is indeed an alternative to cross validation. It is certainly more widely used in statistics and finance.
- In machine learning, the golden rule is to use the *k*-fold cross validation approach. However, we suggest not forgetting about the bootstrap in all those situations where, due to outliers or a few examples that are too heterogeneous, you have a large standard error of the evaluation metric in cross-validation. In these cases, the bootstrap will prove much more useful in validating your models.

Tuning your model validation system

- At this point, you should have a complete overview of all possible validation strategies. When you approach a competition, you devise your validation strategy and you implement it. Then, you test if the strategy you have chosen is correct.

Golden rule

- As a golden rule, be guided in devising your validation strategy by the idea that you have to replicate the same approach used by the organizers of the competition to split the data into training, private, and public test sets. Ask yourself
  - How the organizers have arranged those splits.
  - Did they draw a random sample?
  - Did they try to preserve some specific distribution in the data?
  - Are the test sets actually drawn from the same distribution as the training data?

Why does this matter?

- These are not the questions you would ask yourself in a real-world project. Contrary to a real-world project where you have to generalize at all costs, a competition has a much narrower focus on having a model that performs on the given test set (especially the private one).
- If you focus on this idea from the beginning, you will have more of a chance of finding out the best validation strategy, which will help you rank more highly in the competition.

Trial and error process

- Since this is a trial-and-error process, as you try to find the best validation strategy for the competition, you can systematically apply the following two consistency checks in order to figure out if you are on the right path:

1. First, you have to check if your local tests are consistent, that is, that the single cross-validation fold errors are not so different from each other or, when you opt for a simple train-test split, that the same results are reproducible using different train-test splits.
2. Then, you have to check if your local validation error is consistent with the results on the public leader board.

- If you’re failing the first check, you have a few options depending on the following possible origins of the problem:
  - You don’t have much training data
  - The data is too diverse and every training partition is very different from every other (for instance, if you have too many high cardinality features, that is, features with too many levels – like zip codes – or if you have multivariate outliers)

- In both cases, the point is that you lack data with respect to the model you want to implement. Even when the problem just appears to be that the data is too diverse, plotting learning curves will make it evident to you that your model needs more data.

Strategy

- In this case, unless you find out that moving to a simpler algorithm works on the evaluation metric (in which case trading variance for bias may worsen your model's performance, but not always), your best choice is to use an extensive validation approach. This can be implemented by:
  - Using larger k values (thus approaching LOO where k = n). Your validation results will be less about the capability of your model to perform on unseen data, but by using larger training portions, you will have the advantage of more stable evaluations.
  - Averaging the results of multiple k-fold validations (based on different data partitions picked by different random seed initializations).
  - Using repetitive bootstrapping.

Check what others find

- Keep in mind that when you find unstable local validation results, you won’t be the only one to suffer from the problem. Usually, this is a common problem due to the data’s origin and characteristics. By keeping tuned in to the discussion forums, you may get hints at possible solutions. For instance, a good solution for high cardinality features is target encoding; stratification can help with outliers; and so on.

Tips

- The situation is different when you've passed the first check but failed the second; your local cross-validation is consistent but you find that it doesn't hold on the leaderboard. In order to realize this problem exists, you have to keep diligent note of all your experiments, validation test types, random seeds used, and leaderboard results if you submitted the resulting predictions. In this way, you can draw a simple scatter plot and try fitting a linear regression or, even simpler, compute a correlation between yourlocal results and the associated public leaderboard scores.
- It costs some time and patience to annotate and analyze all of these, but it is the most important meta-analysis of your competition performances that you can keep track of.

What is your validation score is lower?

- When the mismatch is because your validation score is systematically lower or higher than the leaderboard score, you actually have a strong signal that something is missing from your validation strategy, but this problem does not prevent you from improving your model. In fact, you can keep on working on your model and expect improvements to be reflected on the leaderboard, though not in a proportional way.
- However, systematic differences are always a **red flag**, implying something is different between what you are doing and what the organizers have arranged for testing the
- model.

What can go wrong?

- An even worse scenario occurs when your local cross-validation scores do not correlate at all with the leaderboard feedback. This is really a red flag. When you realize this is the case, you should immediately run a series of tests and investigations in order to figure out why, because, regardless of whether it is a common problem or not, the situation poses a serious threat to your final rankings. There are a few possibilities in such a scenario:

Potential reasons 1/

- You figure out that the test set is drawn from a different distribution to the training set. The adversarial validation test (that we will discuss in the next section) is the method that can enlighten you in such a situation.
- The data is non-i.i.d. but this is not explicit. For instance, in *The Nature Conservancy Fisheries Monitoring* competition (https://www.kaggle.com/c/the-nature-conservancyfisheries-monitoring), images in the training set were taken from similar situations (fishing boats). You had to figure out by yourself how to arrange them in order to avoid the model learning to identify the target rather than the context of the images (see, for instance, this work by *Anokas*: https://www.kaggle.com/anokas/finding-boatids).

Potential reasons 2/

- The multivariate distribution of the features is the same, but some groups are distributed differently in the test set. If you can figure out the differences, you can set your training set and your validation accordingly and gain an edge. You need to probe the public leaderboard to work this out.
- The test data is drifted or trended, which is usually the case in time series predictions. Again, you need to probe the public leaderboard to get some insight about some possible postprocessing that could help your score, for instance, applying a multiplier to your predictions, thus mimicking a decreasing or increasing trend in the test data.

Probing 1/

- As we've discussed before, probing the leaderboard is the act of making specifically devised submissions in order to get insights about the composition of the public test set. It works particularly well if the private test set is similar to the public one. There are no general methods for probing, so you have to devise a probing methodology according to the type of competition and problem.
- For instance, in the paper Climbing the Kaggle Leaderboard by Exploiting the Log-Loss Oracle (https://export.arxiv.org/pdf/1707.01825), Jacob explains how to get fourth position in a competition without even downloading the training data.

Probing 2/

- With regard to regression problems, in the recent 30 Days of ML organized by Kaggle, Hung Khoi explained how probing the leaderboard helped him to understand the differences in the mean and standard deviation of the target column between the training dataset and the public test data (see: https://www.kaggle.com/c/30-days-ofml/discussion/269541).

RMSE demystified!

- He used the following equation:

$$RMSE^2 = MSE = variance + (mean - guessed_{value})^2$$

- Essentially, you need just two submissions to solve for the mean and variance of the test target, since there are two unknown terms variance and mean.

- You can also get some other ideas about leaderboard probing from Chris Deo Deo (https://www.kaggle.com/cdeotte) from this post,
- https://www.kaggle.com/cdeotte/lb-probing-strategies-0-890-2ndplace , relevant to the Don't Overfit II competition (https://www.kaggle.com/c/dont-overfit-ii).

Probing issue

- If you want to get a feeling about how probing information from the leaderboard is a double-edged sword, you can read about how Zahar Chikishev managed to probe information from the LANL Earthquake Prediction competition, ending up in 87th place in the private leaderboard after leading in the public one:
- https://towardsdatascience.com/how-to-lbprobe-on-kaggle-c0aa21458bfe

Using adversarial validation

- As we have discussed, cross-validation allows you to test your model's ability to generalize to unseen datasets coming from the same distribution as your training data. Hopefully, since in a Kaggle competition you are asked to create a model that can predict on the public and private datasets, you should expect that such test data is from the same distribution as the training data. In reality, this is not always the case.
- Even if you do not overfit to the test data because you have based your decision not only on the leaderboard results but also considered your cross-validation, you may still be surprised by the results. This could happen in the event that the test set is even slightly different from the training set on which you have based your model. In fact, the target probability and its distribution, as well as how the predictive variables relate to it, inform your model during training about certain expectations that cannot be satisfied if the test data is different from the training data.

Advise

- Hence, it is not enough to avoid overfitting to the leaderboard as we have discussed up to now, but, in the first place, it is also advisable to find out if your test data is comparable to the training data. Then, if they differ, you have to figure out if there is any chance that you can mitigate the different distributions between training and test data and build a model that performs on that test set.

- Adversarial validation has been developed just for this purpose. It is a technique allowing you to easily estimate the degree of difference between your training and test data. This technique was long rumored among Kaggle participants and transmitted from team to team until it emerged publicly thanks to a post by Zygmunt Zając (https://www.kaggle.com/zygmunt) on his FastML blog.

- The idea is simple: take your training data, remove the target, assemble your training data together with your test data, and create a new binary classification target where the positive label is assigned to the test data. At this point, run a machine learning classifier and evaluate for the ROC-AUC evaluation metric (we discussed this metric in the previous chapter on Detailing Competition Tasks and Metrics).
- If your ROC-AUC is around 0.5, it means that the training and test data are not easily distinguishable and are apparently from the same distribution. ROC-AUC values higher than 0.5 and nearing 1.0 signal that it is easy for the algorithm to figure out what is from the training set and what is from the test set: in such a case, don’t expect to be able to easily generalize to the test set because it clearly comes from a different distribution.

Example

- You can find an example Notebook written for the Sberbank Russian Housing Market competition (https://www.kaggle.com/c/sberbank-russianhousing-market) that demonstrates a practical example of adversarial validation and its usage in a competition here:
- https://www.kaggle.com/konradb/adversarialvalidation-and-other-scary-terms

Suggestion

- Since your data may be of different types (numeric or string labels) and you may have missing cases, you'll need some data processing before being able to successfully run the classifier. Our suggestion is to use the random forest classifier because:

advise

- The random forest is a flexible algorithm based on decision trees that can do feature selection by itself and operate on different types of features without any pre-processing, while rendering all the data numeric. It is also quite robust to overfitting and you don’t have to think too much about fixing its hyperparameters.
- You don’t need much data processing because of its tree-based nature.
- For missing data, you can simply replace the values with an improbable negative value such as -999, and you can deal with string variables by converting their strings into numbers (for instance, using the Scikit-learn label encoder, sklearn.preprocessing.LabelEncoder).
- As a solution, it performs less well than one-hot encoding, but it is very speedy and it will work properly for the problem.