IASD M2 at Paris Dauphine

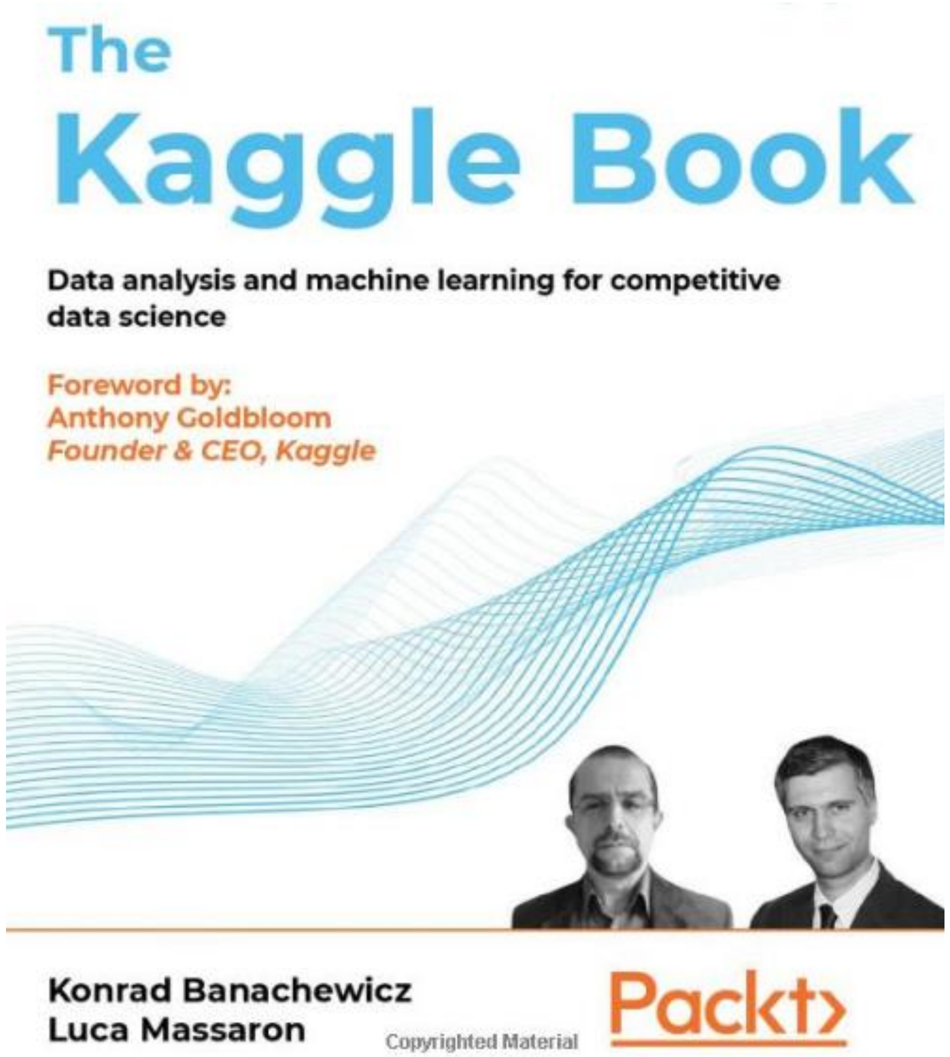
Become a Kaggle Master

5: Model ensemble with blending and stacking

Eric Benhamou

Acknowledgement

The materials of this course is entirely based on the **seminal book**



The
Kaggle Book

Data analysis and machine learning for competitive data science

Foreword by:
Anthony Goldbloom
Founder & CEO, Kaggle

Konrad Banachewicz
Luca Massaron

Copyrighted Material

Packt>

Agenda

Overview

- When you start competing on Kaggle, it doesn’t take long to realize that you cannot win with a single, well-devised model; **you need to ensemble multiple models**.
- **Next, you will immediately wonder how to set up a working ensemble.**
- There are few guides around, and more is left to Kaggle’s lore than to scientific papers.

Winning Kaggle competition

- The point here is that if ensembling is the key to winning in Kaggle competitions, in the real world it is associated with complexity, poor maintainability, difficult reproducibility, and hidden technical costs for little advantage.
- Often, the small boost that can move you from the lower ranks to the top of the leaderboard really doesn’t matter for real-world applications because the costs overshadow the advantages.
- However, that doesn’t mean that ensembling is not being used at all in the real world.
- In a limited form, such as averaging and mixing a few diverse models, ensembling allows us to create models that can solve many data science problems in a more effective and efficient way.

Ensembling as a team

- Ensembling in Kaggle is not only a way to gain extra predictive performance, but it is also a teaming strategy.
- When you are working with other teammates, putting together everyone’s contributions produces a result that often performs better than individual efforts, and can also help to organize the work of the team by structuring everyone’s efforts toward a clear goal.
- In fact, when work is performed in different time zones and under different constraints for each participant, collaborative techniques like pair coding are clearly not feasible.
- One team member may be subject to constraints due to office hours, another due to studying and examinations, and so on.

Why ensembling?

- Teams in a competition often don’t have the chance to, and do not necessarily have to, synchronize and align all participants on the same tasks.
- Moreover, the skills within a team may also differ.
- A good ensembling strategy shared among a team means that individuals can keep working based on their own routines and styles, yet still contribute to the success of the group.
- Therefore, even different skills may become an advantage when using ensembling techniques based on diversity of predictions.

What this chapter is about?

- In this chapter, we will start from the ensembling techniques that you already know, because they are embedded in algorithms such as random forests and gradient boosting, and then progress to ensembling techniques for multiple models such as averaging, blending, and stacking. We will provide you with some theory, some practice, and also some code examples you can use as templates when building your own solutions on Kaggle.

Agenda

- We will cover these topics:
  - A brief introduction to ensemble algorithms
  - Averaging models into an ensemble
  - Blending models using a meta-model
  - Stacking models
  - Creating complex stacking and blending solution

Example

- https://www.kaggle.com/code/amrmahmoud123/1-guide-to-ensembling-methods/notebook

A brief introduction to ensemble algorithms

- The idea that ensembles of models can outperform single ones is not a recent one.
- We can trace it back to *Sir Francis Galton*, who was alive in Victorian Britain.
- He figured out that, in order to guess the weight of an ox at a county fair, it was more useful to take an average from a host of more or less educated estimates from a crowd than having a single carefully devised estimate from an expert.

Breiman

- In 1996, *Leo Breiman* formalized the idea of using multiple models combined into a more predictive one by illustrating the **bagging** technique (also called the "bootstrap aggregating" procedure) that later led to the development of the even more effective **random forests** algorithms. In the period that followed, other ensembling techniques such as **gradient boosting** and **stacking** were also presented, thus completing the range of ensemble methods that we use today.

References

- For random forests, read Breiman, L. Bagging predictors. Machine learning 24.2 – 1996: 123-140.
- If you want to know how boosting originally worked in more detail, read Freund, Y. and Schapire, R.E. Experiments with a new boosting algorithm. icml. Vol. 96 – 1996, and Friedman, J. H. Greedy function approximation: a gradient boosting machine. Annals of Statistics (2001): 1189-1232.
- As for stacking, refer to Ting, K. M. and Witten, I. H. Stacking bagged and dagged models, 1997, for a first formal draft of the technique.

Stacking 1/

- The first basic strategies for ensembling predictors in Kaggle competitions were taken directly from bagging and random forest strategies for classification and regression. They involved making an average of various predictions and were thus named **averaging** techniques.
- These approaches quickly emerged from the very first Kaggle competitions held over 11 years ago also because of the pre-Kaggle Netflix competition, where strategies based on the average of the results of dif- ferent models dominated the scene.
- Given their success, basic ensembling techniques based on averaging set a standard for many competitions to come, and they are still quite useful and valid even today for scoring more highly on the leaderboard.

Stacking 2/

- Stacking, which is more complex and computationally demanding, emerged a bit later, when problems in competitions become more complex and the struggle between participants fiercer.
- Just as the random forest approach has inspired averaging different predictions, boosting heavily inspired stacking approaches.
- In boosting, by sequentially re-processing information, your learning algorithm can model problems in a better and more complete way.
- In fact, in gradient boosting, sequential decision trees are built in order to model the part of data that previous iterations are unable to grasp.
- This idea is reprised in stacking ensembles, where you stack the results of previous models and re-process them in order to gain an increase in predictive performance

Averaging models into an ensemble

- In order to introduce the averaging ensembling technique better, let’s quickly revise all the strat- egies devised by Leo Breiman for ensembling. His work represented a milestone for ensembling strategies, and what he found out at the time still works fairly well in a wide range of problems

- Breiman explored all these possibilities in order to figure out if there was a way to reduce the variance of error in powerful models that tended to overfit the training data too much, such as decision trees.
- Conceptually, he discovered that ensembling effectiveness was based on three elements: how we deal with the **sampling of training cases**, how we **build the models**, and, finally, how we **combine the different models** obtained.

- As for the sampling, the approaches tested and found were:
  - **Pasting**, where a number of models are built using subsamples (sampling without re- placements) of the examples (the data rows)
  - **Bagging**, where a number of models are built using random selections of bootstrapped examples (sampling with replacement)
  - **Random subspaces**, where a number of models are built using subsamples (sampling without replacements) of the features (the data columns)
  - **Random patches**, an approach similar to bagging, except features are also sampled when each model is selected, as in random subspaces

- The reason we sample instead of using the same information is because, by subsampling cases and features, we create models that are all relevant to the same problem while each being different from the others. This difference also applies to the way each model overfits the sample; we expect all the models to grasp the useful, generalizable information from the data *in the same way*, and deal with the noise that is not useful for making predictions *in a different way*. Hence, variation in modeling reduces the variation in predictions, because errors tend to cancel each other out.

•If this variation is so useful, then the next step should not just be to modify the *data* the model learns from, but also *the model itself*. We have two main approaches for the models:

- Ensembles of the same type of models
- Ensembles of different models

- Interestingly, ensembling in one way or the other doesn’t help too much if the models that we are putting together are too different in predictive power. The point here is that you get an ad- vantage if you put together models that are able to correctly guess the same type of predictions, so they can smooth out their errors when averaging the predictions that they get wrong. If you are ensembling models with performances that are too different, you will soon find out that there is no point because the net effect will be negative: as you are not smoothing your incorrect predictions, you are also degrading the correct ones.

- This is an important limit of averaging: it can use a set of different models (for instance, because they are trained using different samples and features) only if they are similar in predictive power. To take an example, a linear regression and a *k*-nearest neighbor algorithm have different ways of modeling a problem and capturing signals from data; thanks to the (distinct) characteristic functional forms at their cores, these algorithms can grasp different predictive nuances from the data and perform better on specific parts of their predictive tasks, but you cannot really take advantage of that when using averaging. By contrast, the different ways algorithms have to capture signals is something that stacking actually can leverage, because it can take the best results from each algorithm.

- Based on this, we can summarize that, for an ensemble based on averaging (averaging the results of multiple models) to be effective, it should be:
  - Built on models that are trained on different samples
  - Built on models that use different subsamples from the available features
  - Composed of models similar in predictive power

•Technically, this implies that the models' predictions should be as uncorrelated as possible while performing at the same level of accuracy on prediction tasks.

•Now that we have discussed the opportunities and limitations of averaging multiple machine learning models, we are finally going to delve into the technical details. There are three ways to average multiple classification or regression models:

- Majority voting, using the most frequent classification among multiple models (only for classification models)
- Averaging values or probabilities
- Using a weighted average of values or probabilities

- In the next few sections, we will discuss each approach in detail in the context of Kaggle com- petitions.

Majority voting

- Producing different models by varying the examples, features, and models we use in the ensemble (if they are comparable in predictive power, as we discussed before) requires a certain computa- tional effort, but it doesn't require you to build a data processing pipeline that is all that different from what you would set up when using a single model.

•In this pipeline, you just need to collect different test predictions, keeping track of the models used, how you sampled examples or features when training, the hyperparameters that you used, and the resulting cross-validation performance.

•If the competition requires you to predict a class, you can use **majority voting**; that is, for each prediction, you take the class most frequently predicted by your models. This works for both binary predictions and multi-class predictions, because it presumes that there are sometimes errors in your models, but that they can guess correctly most of the time. Majority voting is used as an "error correction procedure," discarding noise and keeping meaningful signals.

•In our first simple example, we demonstrate how majority voting works. We start by creating our example dataset. Using the `make_classification` function from Scikit-learn, we generate a *Madelon*-like dataset.

Madelon dataset

- The original Madelon was an artificial dataset containing data points grouped in clusters placed on the vertices of some dimensional hypercube and randomly labeled. It comprised a few informative features, mixed with irrelevant and repeated ones (to create multicollinearity between features) and it has a certain amount of injected random noise. Ideated by *Isabelle Guyon* (one of the creators of the SVM algorithm) for the *NIPS 2003 Feature Selection Challenge*, the Madelon dataset is the model example of a challenging artificial dataset for a competition. Even some Kaggle competitions were inspired by it: https://www.kaggle.com/c/overfitting and the more recent https://www.kaggle.com/c/dont-overfit-ii.

Code

- We will use this recreation of the Madelon dataset throughout this chapter as a basis for testing ensembling techniques:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import
train_test_split

X, y = make_classification(n_samples=5000, n_features=50,
          n_informative=10, n_redundant=25,
          n_repeated=15, n_clusters_per_class=5,
          flip_y=0.05, class_sep=0.5, random_state=0)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
     test_size=0.33, random_state=0)
```

After splitting it into a training and a test set, we proceed by instantiating our learning algorithms. We will just use three base algorithms: SVMs, random forests, and *k*-nearest neighbors classi- fiers, with default hyperparameters for demonstration purposes. You can try changing them or increasing their number:

Code 2/

```
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import log_loss, roc_auc_score,
accuracy_score


model_1 = SVC(probability=True, random_state=0)
model_2 = RandomForestClassifier(random_state=0)
model_3 = KNeighborsClassifier()
```

- The following step is just to train each model on the training set:

```
model_1.fit(X_train, y_train)
model_2.fit(X_train, y_train)
model_3.fit(X_train, y_train)
```

Testing

- At this point, we need to predict on the test set for each model and ensemble all these predictions using majority voting. To do this, we will be using the `mode` function from SciPy:

```
import numpy as np
from scipy.stats import mode

preds = np.stack([model_1.predict(X_test),
       model_2.predict(X_test),
       model_3.predict(X_test)]).T

max_voting = np.apply_along_axis(mode, 1, preds)[:,0]
```

Code

- First, we check the accuracy for each single model:

```
for i, model in enumerate(['SVC', 'RF ', 'KNN']):
  acc = accuracy_score(y_true=y_test, y_pred=preds[:, i])
  print(f"Accuracy for model {model} is: {acc:0.3f}")
```

- We see that the three models have similar performance, around **0.8**. Now it is time to check the majority voting ensemble

```
max_voting_accuray = accuracy_score(y_true=y_test, y_pred=max_voting)
print(f"Accuracy for majority voting is: {max_voting_accuray:0.3f}")
```

- The voting ensemble is actually more accurate: **0.817**, because it managed to put together the correct signals from the majority.
- For multilabel problems (when you can predict multiple classes), you can just pick the classes that are predicted above a certain number of times, assuming a relevance threshold that indicates that a prediction for a class is signal, not noise. For instance, if you have five models, you could set this threshold to 3, which means if a class is predicted by at least three models, then the prediction should be considered correct.

What about regression

- In regression problems, as well as when you are predicting probabilities, you cannot actually use majority voting. Majority voting works exclusively with class ownership. Instead, when you have to predict numbers, you need to combine the results numerically. In this case, resorting to an **average** or a **weighted average** will provide you the right way to combine predictions.

Averaging of model predictions

- When averaging your predictions from different models in a competition, you can consider all your predictions as having potentially the same predictive power and use the arithmetic mean to derive an average value.

For regression

- Aside from the arithmetic mean, we have also found it quite effective to use:
  - The **geometric mean**: This is where you multiply the *n* submissions, then you take the
  - *1/n*th power of the resulting product.
  - The **logarithmic mean**: Analogous to the geometric mean, you take the logarithm of your submission, average them together, and take the exponentiation of the resulting mean.
  - The **harmonic mean**: Where you take the arithmetic mean of the reciprocals of your submissions, then you take the reciprocal of the resulting mean.
  - The **mean of powers**: Where you take the average of the *n*th power of the submissions,
  - then you take the *1/n*th power of the resulting average.

- The simple arithmetic average is always quite effective and basically a no-brainer that works more often than expected. Sometimes, variants such as the geometric mean or the harmonic mean may work better.

- Continuing with the previous example, we will now try to figure out what kind of mean works best when we switch to **ROC-AUC** as our evaluation metric. To begin with, we evaluate the per- formances of each single model:

Code

```python
proba = np.stack([model_1.predict_proba(X_test)[:, 1],
                  model_2.predict_proba(X_test)[:, 1],
                  model_3.predict_proba(X_test)[:, 1]]).T

for i, model in enumerate(['SVC', 'RF ', 'KNN']):
  ras = roc_auc_score(y_true=y_test, y_score=proba[:, i])
  print(f"ROC-AUC for model {model} is: {ras:0.5f}")
```

- The results give us a range from **0.875** to **0.881**.
- Our first test is performed using the arithmetic mean:

```
arithmetic = proba.mean(axis=1)

ras = roc_auc_score(y_true=y_test,
      y_score=arithmetic)

print(f"Mean averaging ROC-AUC is: {ras:0.5f}")
```

- The resulting ROC-AUC score is decisively better than the single performances: **0.90192**. We also test if the geometric, harmonic, or logarithmic mean, or the mean of powers, can outperform the plain mean:

Different types of mean

```
geometric = proba.prod(axis=1)**(1/3)

ras = roc_auc_score(y_true=y_test, y_score=geometric) print(f"Geometric averaging ROC-AUC is:
{ras:0.5f}")


harmonic = 1 / np.mean(1. / (proba + 0.00001), axis=1) ras = roc_auc_score(y_true=y_test,
y_score=harmonic) print(f"Geometric averaging ROC-AUC is: {ras:0.5f}")


n = 3
mean_of_powers = np.mean(proba**n, axis=1)**(1/n)

ras = roc_auc_score(y_true=y_test, y_score=mean_of_powers) print(f"Mean of powers averaging ROC-AUC
is: {ras:0.5f}")


logarithmic = np.expm1(np.mean(np.log1p(proba), axis=1)) ras = roc_auc_score(y_true=y_test,
y_score=logarithmic) print(f"Logarithmic averaging ROC-AUC is: {ras:0.5f}")
```

- Running the code will tell us that none of them can. In this case, the arithmetic mean is the best choice for ensembling. What actually works better than the simple mean, in almost all cases, is putting some *prior knowledge* into the way you combine the numbers. This happens when you weight your models in the mean calculation.

Weighted averages

- When weighting your models, you need to find an empirical way to figure out the right weights. A common method, though very prone to adaptive overfitting, is to test different combinations on the public leaderboard until you find the combination that scores the best. Of course, that won't ensure that you score the same on the private leaderboard. Here, the principle is to weight what works better. However, as we have discussed at length, very often the feedback from the public leaderboard cannot be trusted because of important differences with the private test data. Yet, you *can* use your cross-validation scores or out-of-fold ones (the latter will be discussed along with stacking in a later section). In fact, another viable strategy is to use weights that are **proportional to the models' cross-validation performances**.

- Although it is a bit counterintuitive, another very effective method is weighting the submissions **inversely proportionally to their covariances**. In fact, since we are striving to cancel errors by averaging, averaging based on the unique variance of each submission allows us to weight more heavily the predictions that are less correlated and more diverse, more effectively reducing the variance of the estimates.

Using correlatin

- In the next example, we will first create a **correlation matrix** of our predicted probabilities, and then we proceed by:

1. Removing the one values on the diagonal and replacing them with zeroes
2. Averaging the correlation matrix by row to obtain a vector
3. Taking the reciprocal of each row sum
4. Normalizing their sum to 1.0
5. Using the resulting weighting vector in a matrix multiplication of our predicted proba- bilities

Code

```
cormat = np.corrcoef(proba.T)
np.fill_diagonal(cormat, 0.0)
W = 1 / np.mean(cormat, axis=1)
W = W / sum(W) # normalizing to sum==1.0
weighted = proba.dot(W)
ras = roc_auc_score(y_true=y_test,
y_score=weighted) print(f"Weighted averaging
ROC-AUC is: {ras:0.5f}")
```

Result

- The resulting ROC-AUC of **0.90206** is slightly better than the plain average. Giving more impor- tance to more uncorrelated predictions is an ensembling strategy that is often successful. Even if it only provides slight improvements, this could suffice to turn the competition to your advantage.

Averaging in your cross-validation strategy

- As we have covered, averaging doesn’t require you to build any special complex pipelines, only a certain number of typical data pipelines that create the models you are going to average, either using the same weights for all predictions or some empirically found weights. The only way to test it is to run a submission on the public leaderboard, thus risking adaptive fitting because your evaluation of the averaging will solely be based on the response from Kaggle.

- Before testing directly on the leaderboard, though, you may also test at training time by running the averaging operations on the validation fold (the fold that you are not using for training your model). This will provide you with less biased feedback than that from the leaderboard. In the following code, you can find an example of how a cross-validation prediction is arranged:

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=5, shuffle=True, random_state=0)
scores = list()

 for k, (train_index, test_index) in
 enumerate(kf.split(X_train)):
 model_1.fit(X_train[train_index, :], y_train[train_index])
 model_2.fit(X_train[train_index, :], y_train[train_index])
 model_3.fit(X_train[train_index, :], y_train[train_index])
```

```
proba = np.stack(
  [model_1.predict_proba(X_train[test_index, :])[:, 1],
   model_2.predict_proba(X_train[test_index, :])[:, 1],
   model_3.predict_proba(X_train[test_index, :])[:,
   1]]).T

arithmetic = proba.mean(axis=1)
ras = roc_auc_score(y_true=y_train[test_index],
                    y_score=arithmetic)
  scores.append(ras)
  print(f"FOLD {k} Mean averaging ROC-AUC is: {ras:0.5f}")

print(f"CV Mean averaging ROC-AUC is:
{np.mean(scores):0.5f}")
```

Relying on the results of a cross-validation as in the code above can help you evaluate which averaging strategy is more promising, without testing directly on the public leaderboard.

Correcting averaging for ROC-AUC evaluations

- If your task will be evaluated on the ROC-AUC score, simply averaging your results may not suffice. This is because different models may have adopted different optimization strategies and their outputs may be deeply different.
- A solution could be to calibrate the models, a type of post-pro- cessing we previously discussed in *Competition Tasks and Metrics*, but this obviously takes further time and computational effort.

- In these cases, the straightforward solution would be to convert output probabilities into ranks and just average the ranks (or make a weighted average of them). Using a min-max scaler ap- proach, you simply convert each model’s estimates into the range 0-1 and then proceed with averaging the predictions. That will effectively convert your model’s probabilistic output into ranks that can be compared:

```
from sklearn.preprocessing import MinMaxScaler

proba = np.stack(
     [model_1.predict_proba(X_train)[:, 1],
      model_2.predict_proba(X_train)[:, 1],
      model_3.predict_proba(X_train)[:, 1]]).T


arithmetic = MinMaxScaler().fit_transform(proba).mean(axis=1) ras =
roc_auc_score(y_true=y_test, y_score=arithmetic) print(f"Mean averaging
ROC-AUC is: {ras:0.5f}")
```

- This approach works perfectly when you are directly handling the test predictions. If, instead, you are working and trying to average results during cross-validation, you may encounter problems because the prediction range of your training data may differ from the range of your test predic- tions. In this case, you can solve the problem by training a calibration model (see **probability calibration** on Scikit-learn (`https://scikit-learn.org/stable/modules/calibration.html`) and *Chapter 5*), converting predictions into true, comparable probabilities for each of your models.

Blending models using a meta-model

- The Netflix competition (which we discussed at length in *Chapter 1*) didn’t just demonstrate that averaging would be advantageous for difficult problems in a data science competition; it also brought about the idea that you can use a model to average your models’ results more effectively. The winning team, BigChaos, in their paper (Töscher, A., Jahrer, M., and Bell, R.M. *The BigChaos Solution to the Netflix Grand Prize*. Netflix prize documentation – 2009) made many mentions of **blending**, and provided many hints about its effectiveness and the way it works.

- In a few words, blending is kind of a weighted averaging procedure where the weights used to combine the predictions are estimated by way of a holdout set and a meta-model trained on it. A **meta-model** is simply a machine learning algorithm that learns from the output of other ma- chine learning models. Usually, a meta-learner is a linear model (but sometimes it can also be a non-linear one; more on that in the next section), but you can actually use whatever you want, with some risks that we will discuss.

- The procedure for obtaining a blending is straightforward:

1. Before starting to build all your models, you randomly extract a holdout sample from the training data (in a team, you should all use the same holdout). Usually, the holdout is about 10% of the available data; however, depending on circumstances (for instance, the number of examples in your training data, stratifications), it could be less as well as more. As always in sampling, you may enforce stratification in order to ensure sampling representativeness, and you can test using adversarial validation that the sample really matches the distribution in the rest of the training set.

2. Train all your models on the remaining training data.
3. Predict on the holdout and on the test data.
4. Use the holdout predictions as training data in a meta-learner and reuse the meta-learner model to compute the final test predictions using the test predictions from your models. Alternatively, you can use the meta-learner to figure out the selection of predictors and their weights that should be used in a weighted average.

- There a quite a few advantages and disadvantages to such a procedure. Let’s start with the ad- vantages. First, it is easy to implement; you just have to figure out what the holdout sample is. In addition, using a meta-learning algorithm ensures you will find the best weights without testing on the public leaderboard.

- In terms of weaknesses, sometimes, depending on sample size and the type of models you use, reducing the number of training examples may increase the variance of the predictions of your estimators. Moreover, even if you take great care over how you sample your holdout, you may still fall into adaptive overfitting, that is, finding weights that suit the holdout but are not generaliz- able, especially if you use a meta-learner that is too complex. Finally, using a holdout for testing purposes has the same limitations as the training and test split we discussed in the chapter on model validation: you won’t have a reliable estimate if the sample size of the holdout is too small or if, for some reason, your sampling is not representative.

Best practices for blending

- In blending, the kind of meta-learner you use can make a great difference. The most common choices are to use a linear model or a non-linear one. Among linear models, linear or logistic regressions are the preferred ones. Using a regularized model also helps to discard models that are not useful (L1 regularization) or reduce the influence of less useful ones (L2 regularization). One limit to using these kinds of meta-learners is that they may assign some models a negative contribution, as you will be able to see from the value of the coefficient in the model. When you encounter this situation, the model is usually overfitting, since all models should be contributing positively to the building of the ensemble (or, at worst, not contributing at all). The most recent versions of Scikit-learn allow you to impose only positive weights and to remove the intercept. These constraints act as a regularizer and prevent overfitting.

- Non-linear models as meta-learners are less common because they tend to overfit in regression and binary classification problems, but they often shine in multiclass and multilabel classifica- tion problems since they can model the complex relationships between the classes present. They also generally perform better if, aside from the models' predictions, you also provide them with the *original features*, since they can spot any useful interactions that help them correctly select which models to trust more.

- In our next example, we first try blending using a linear model (a logistic regression), then a non-linear approach (a random forest). We start by splitting the training set into a training part for the blend elements and a holdout for the meta-learner. Afterward, we fit the models on the trainable part and predict on the holdout.

Code

```
from sklearn.preprocessing import StandardScaler

X_blend, X_holdout, y_blend, y_holdout =
train_test_split(X_train, y_ train,
test_size=0.25, random_state=0)
```

Code

```
model_1.fit(X_blend,            y_blend)
model_2.fit(X_blend,            y_blend)
model_3.fit(X_blend, y_blend)

proba = np.stack([model_1.predict_proba(X_holdout)[:, 1],
      model_2.predict_proba(X_holdout)[:, 1],
      model_3.predict_proba(X_holdout)[:, 1]]).T
scaler = StandardScaler()
proba = scaler.fit_transform(proba)
```

- We can now train our linear meta-learner using the probabilities predicted on the holdout:

```
from sklearn.linear_model import LogisticRegression


blender =
LogisticRegression(solver='liblinear')
blender.fit(proba, y_holdout)

print(blender.coef_)
```

Result

- The resulting coefficients are:
- `[[0.78911314 0.47202077 0.75115854]]`

- By looking at the coefficients, we can figure out which model contributes more to the meta-en- semble. However, remember that coefficients also rescale probabilities when they are not well calibrated, so a larger coefficient for a model may not imply that it is the most important one. If you want to figure out the role of each model in the blend by looking at coefficients, you first have to rescale them by standardization (in our code example, this has been done using Scikit-learn’s `StandardScaler`).

- Our output shows us that the SVC and *k*-nearest neighbors models are weighted more in the blend than the random forest one; their coefficients are almost equivalent and both are larger than the random forest coefficient.
- Once the meta-model is trained, we just predict on our test data and check its performance:

- test_proba = np.stack([model_1.predict_proba(X_test)[:, 1],
    - model_2.predict_proba(X_test)[:, 1],

      model_3.predict_proba(X_test)[:, 1]]).T

Code

```
blending = blender.predict_proba(test_proba)[:, 1] ras = roc_auc_score(y_true=y_test,
y_score=blending)
print(f"ROC-AUC for linear blending {model} is: {ras:0.5f}")
```

We can try the same thing using a non-linear meta-learner, such as a random forest, for instance:

```
blender = RandomForestClassifier() blender.fit(proba, y_holdout)


test_proba = np.stack([model_1.predict_proba(X_test)[:, 1],

             model_2.predict_proba(X_test)[:, 1], model_3.predict_proba(X_test)[:,
             1]]).T


blending = blender.predict_proba(test_proba)[:, 1] ras =
roc_auc_score(y_true=y_test, y_score=blending)
print(f"ROC-AUC for non-linear blending {model} is: {ras:0.5f}")
```

- An alternative to using a linear or non-linear model as a meta-learner is provided by the **ensemble selection** technique formalized by *Caruana*, *Niculescu-Mizil*, *Crew*, and *Ksikes*.
- If you are interested in more details, read their famous paper: Caruana, R., Nicules- cu-Mizil, A., Crew, G., and Ksikes, A. *Ensemble selection from libraries of models* (Pro- ceedings of the Twenty-First International Conference on Machine Learning, 2004).

- The ensemble selection is actually a weighted average, so it could simply be considered analogous to a linear combination. However, it is a constrained linear combination (because it is part of a hill-climbing optimization) that will also make a selection of models and apply only positive weights to the predictions. All this minimizes the risk of overfitting and ensures a more compact solution, because the solution will involve a model selection. From this perspective, ensemble selection is recommended in all problems where the risk of overfitting is high (for instance, be- cause the training cases are few in number or the models are too complex) and in real-world applications because of its simpler yet effective solution.

- When using a meta-learner, you are depending on the optimization of its own cost function, which may differ from the metric adopted for the competition. Another great advantage of en- semble selection is that it can be optimized to any evaluation function, so it is mostly suggested when the metric for the competition is different from the canon of those typically optimized in machine learning models.

- Implementing ensemble selection requires the following steps, as described in the paper men- tioned previously:

Steps

1. Start with your trained models and a holdout sample.
2. Test all your models on the holdout sample and, based on the evaluation metric, retain the most effective in a selection (the **ensemble selection**).
3. Then, keep on testing other models that could be added to the one(s) in the ensemble selection so that the average of the proposed selection improves over the previous one. You can either do this with replacement or without. Without replacement, you only put a model into the selection ensemble once; in this case, the procedure is just like a simple average after a forward selection. (In a forward selection, you iteratively add to a solution the model that improves the performance the most, until adding further models no longer improves the performance.) With replacement, you can put a model into the selection multiple times, thus resembling a weighted average.
4. When you cannot get any further improvement, stop and use the ensemble selection.

Code

- Here is a simple code example of an ensemble selection. We start by deriving a holdout sample and a training selection from our original training data. We fit the models and obtain the predictions on our holdout, as previously seen when blending with a meta-learner:

Code

- `X_blend, X_holdout, y_blend, y_holdout = train_test_split (X_train, y_train, test_size=0.5, random_state=0)`

- `model_1.fit(X_blend, y_blend) model_2.fit(X_blend, y_blend) model_3.fit(X_blend, y_blend)`

- `proba = np.stack([model_1.predict_proba(X_holdout)[:, 1],`
  - `model_2.predict_proba(X_holdout)[:, 1], model_3.predict_proba(X_holdout)[:, 1]]).T`

- In the next code snippet, the ensembling is created through a series of iterations. At each itera- tion, we try adding all the models in turn to the present ensemble and check if they improve the model. If any of these additions outperforms the previous ensemble on the holdout sample, the ensemble is updated and the bar is raised to the present level of performance.

- If no addition can improve the ensemble, the loop is stopped and the composition of the ensemble is reported back:

Code

```
iterations = 100

proba = np.stack([model_1.predict_proba(X_holdout)[:, 1],
                  model_2.predict_proba(X_holdout)[:, 1],
                  model_3.predict_proba(X_holdout)[:, 1]]).T

baseline = 0.5
print(f"starting baseline is {baseline:0.5f}")

models = []
```

For loop

- for i in range(iterations): challengers = list()
- for j in range(proba.shape[1]):
  - new_proba = np.stack(proba[:, models + [j]]) score = roc_auc_score(y_true=y_holdout,
    - y_score=np.mean(new_proba, axis=1)) challengers.append([score, j])
  - challengers = sorted(challengers, key=lambda x: x[0], reverse=True)
- best_score, best_model = challengers[0] if best_score > baseline:
    - print(f"Adding model_{best_model+1} to the ensemble", end=': ')
    - print(f"ROC-AUC increases score to {best_score:0.5f}") models.append(best_model)
  - baseline = best_score else:
- print("Cannot improve further - Stopping")

•Finally, we count how many times each model has been inserted into the average and we calculate the weights for our averaging on the test set:

```
from collections import Counter freqs =
Counter(models)


weights = {key: freq/len(models) for key,
 freq in freqs.items()}
print(weights)
```

- You can make the procedure more sophisticated in various ways. Since this approach may overfit, especially at the initial stages, you could start from a randomly initialized ensemble set or, as the authors suggest, you may already be starting with the *n* best performing models in the set (you decide the value of *n*, as a hyperparameter). Another variation involves applying sampling to the set of models that can enter the selection at each iteration; in other words, you randomly exclude some models from being picked. Not only will this inject randomness into the process but it will also prevent specific models from dominating the selection.

Stacking models together

•**Stacking** was first mentioned in *David Wolpert*'s paper (*Wolpert, D. H. Stacked generalization*. Neural networks 5.2 – 1992), but it took years before the idea become widely accepted and common (only with release 0.22 in December 2019, for instance, has Scikit-learn implemented a stacking wrapper). This was due principally to the Netflix competition first, and to Kaggle competitions afterward.

•In stacking, you always have a meta-learner. This time, however, it is not trained on a holdout, but on the entire training set, thanks to the **out-of-fold (OOF)** prediction strategy. We already discussed this strategy in *Chapter 6*, *Designing Good Validation*. In OOF prediction, you start from a replicable *k*-fold cross-validation split. *Replicable* means that, by recording the cases in each training and testing sets at each round or by reproducibility assured by a random seed, you can replicate the same validation scheme for each model you need to be part of the stacking ensemble.

What about Netflix competition?

- In the Netflix competition, stacking and blending were often used interchangeably, though the actual method devised by Wolpert originally implied leveraging a scheme based on *k*-fold cross-validation, not a holdout set. In fact, the core idea in stacking is not to reduce the variance, as in averaging; it is mostly to reduce the bias, because it is expected that each model involved in the stacking will grasp a part of the infor- mation present in the data, to be recomposed in the final meta-learner.

- Let’s remind ourselves of how OOF predictions on the training data work. When testing a model, at each round of the validation you train a model on part of the training data and you validate on another part that is held out from the training.

- By recording the validation predictions and then reordering them to reconstruct the ordering of the original training cases, you will obtain a prediction of your model on the very same training set that you have used. However, as you have used multiple models and each model has predicted on cases it didn’t use for training, you should not have any overfitting effects on your training set predictions.

- Having obtained OOF predictions for all your models, you can proceed to build a meta-learn- er that predicts your target based on the OOF predictions (first-level predictions), or you can keep on producing further OOF predictions on top of your previous OOF predictions (second- or higher-level predictions), thus creating multiple stacking layers. This is compatible with an idea presented by Wolpert himself: by using multiple meta-learners, you are actually imitating the structure of a fully connected feedforward neural network without backpropagation, where the weights are optimally calculated in order to maximize the predictive performance at the level of each layer separately. From a practical point of view, stacking multiple layers has proven very effective and works very well for complex problems where single algorithms are unable to obtain the best results.

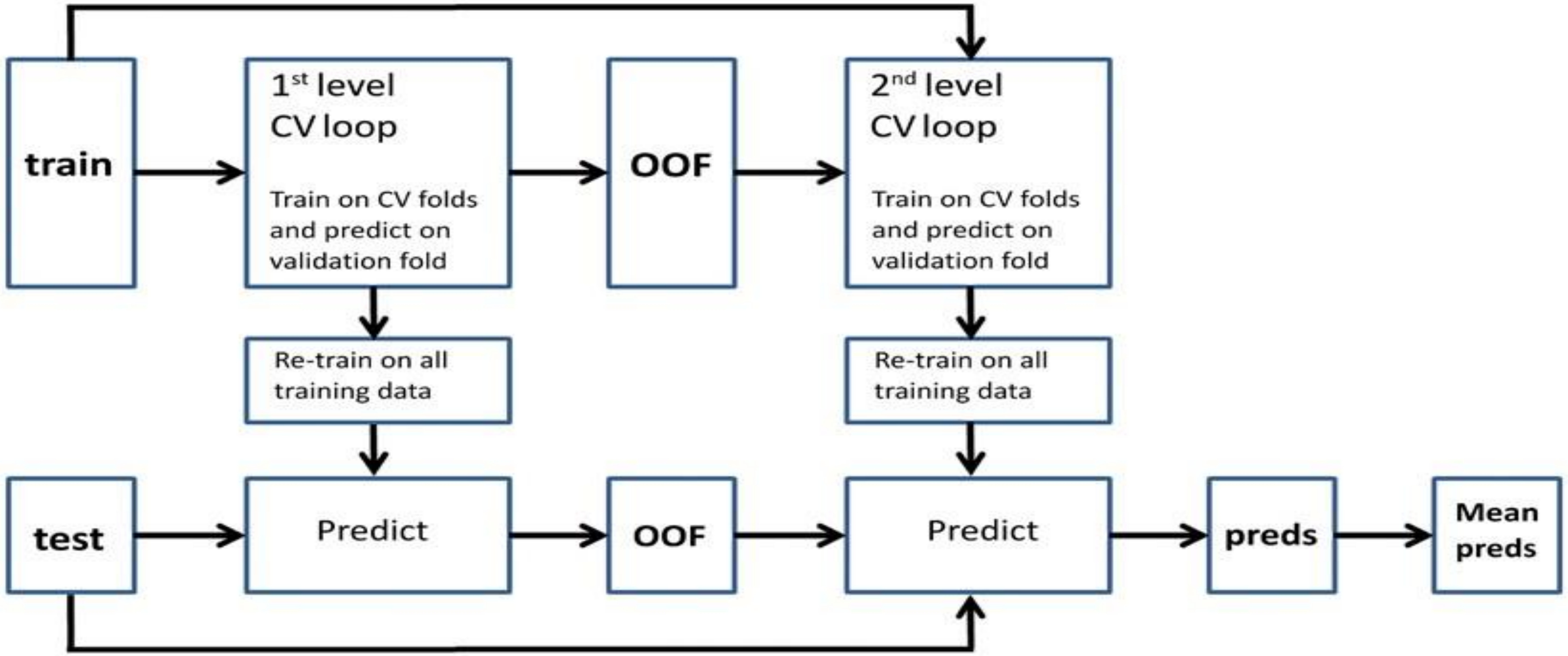
- Moreover, one interesting aspect of stacking is that you don’t need models of comparable predic- tive power, as in averaging and often in blending. In fact, even worse-performing models may be effective as part of a stacking ensemble. A *k*-nearest neighbors model may not be comparable to a gradient boosting solution, but when you use its OOF predictions for stacking it may contribute positively and increase the predictive performance of the ensemble.

- When you have trained all the stacking layers, it is time to predict. As far as producing the pre- dictions used at various stacking stages, it is important to note that you have two ways to do this. The original Wolpert paper suggests re-training your models on all your training data and then using those re-trained models for predicting on the test set. In practice, many Kagglers don’t retrain, but directly use the models created for each fold and make multiple predictions on the test set that are averaged at the end.

- In our experience, stacking is generally more effective with complete re-training on all available data before predicting on the test set when you are using a low number of *k*-folds. In these cases, the sample consistency may really make a difference in the quality of the prediction because training on less data means getting more variance in the estimates. As we discussed in *Chapter 6*, when creating OOF predictions it is always better to use a high number of folds, between 10 to 20. This limits the number of examples that are held out, and, without re-training on all the data, you can simply use the average of predictions obtained from the cross-validation trained models for obtaining your prediction on the test set.

- In our next example, for illustrative purposes, we only have five CV folds and the results are stacked twice. In the diagram below, you can follow how the data and the models move between different stages of the stacking process:

2-layer stacking process with final averaging


train
1st level CV loop
Train on CV folds and predict on validation fold
OOF
2nd level CV loop
Train on CV folds and predict on validation fold
Re-train on all training data
Re-train on all training data
test
Predict
OOF
Predict
preds
Mean preds

- Notice that:
  - Training data is fed to both levels of the stacking (OOF predictions at the second level of the stacking are joined with the training data)
  - After obtaining OOF predictions from the CV loops, models are re-trained on the entire training dataset
  - The final predictions are a simple average of all the predictions obtained by the stacked predictors

- Let’s now take a look at the code to understand how this diagram translates into Python com- mands, starting with the first level of training:

Code

- `from sklearn.model_selection import KFold`

- `kf = KFold(n_splits=5, shuffle=True,`
  `random_state=0) scores = list()`

- `first_lvl_oof = np.zeros((len(X_train), 3))`

- fist_lvl_preds = np.zeros((len(X_test), 3))

  - for k, (train_index, val_index) in enumerate(kf.split(X_train)):
    model_1.fit(X_train[train_index, :], y_train[train_index]) first_lvl_oof[val_index, 0]
    = model_1.predict_proba(
    -                   X_train[val_index, :])[:, 1]

  - model_2.fit(X_train[train_index, :], y_train[train_index]) first_lvl_oof[val_index,
    1] = model_2.predict_proba(
    -                   X_train[val_index, :])[:, 1]

  - model_3.fit(X_train[train_index, :], y_train[train_index]) first_lvl_oof[val_index,
    2] = model_3.predict_proba(
-                   X_train[val_index, :])[:, 1]

After the first layer, we retrain on the full dataset:

- `model_1.fit(X_train, y_train)`
- `fist_lvl_preds[:, 0] = model_1.predict_proba(X_test)[:, 1]`

- `model_2.fit(X_train, y_train)`
- `fist_lvl_preds[:, 1] = model_2.predict_proba(X_test)[:, 1]`

- `model_3.fit(X_train, y_train)`
- `fist_lvl_preds[:, 2] = model_3.predict_proba(X_test)[:, 1]`

Second stacking

- In the second stacking, we will reuse the same models as those in the first layer, adding the stacked OOF predictions to the existing variables:

```
second_lvl_oof = np.zeros((len(X_train), 3)) second_lvl_preds =
np.zeros((len(X_test), 3))


 for k, (train_index, val_index) in enumerate(kf.split(X_train))

   skip_X_train = np.hstack([X_train, first_lvl_oof])

   model_1.fit(skip_X_train[train_index, :], y_train[train_index])

   second_lvl_oof[val_index, 0] = model_1.predict_proba(
            skip_X_train[val_index, :])[:, 1]
```

```
model_2.fit(skip_X_train[train_index, :],
  y_train[train_index])
second_lvl_oof[val_index, 1] = model_2.predict_proba(
   skip_X_train[val_index, :])[:, 1]

model_3.fit(skip_X_train[train_index, :],
    y_train[train_index])
second_lvl_oof[val_index, 2] = model_3.predict_proba(
    skip_X_train[val_index, :])[:, 1]
```

Retraining

- Again, we retrain on the full data for the second layer:
- `skip_X_test = np.hstack([X_test, fist_lvl_preds])`

- `model_1.fit(skip_X_train, y_train)`
- `second_lvl_preds[:, 0] = model_1.predict_proba(skip_X_test)[:, 1]`

- `model_2.fit(skip_X_train, y_train)`
- `second_lvl_preds[:, 1] = model_2.predict_proba(skip_X_test)[:, 1]`

- `model_3.fit(skip_X_train, y_train)`
- `second_lvl_preds[:, 2] = model_3.predict_proba(skip_X_test)[:, 1]`

- The stacking is concluded by averaging all the stacked OOF results from the second layer:
- `arithmetic = second_lvl_preds.mean(axis=1)`
- `ras = roc_auc_score(y_true=y_test,`

  `y_score=arithmetic) scores.append(ras)`
- `print(f"Stacking ROC-AUC is: {ras:0.5f}")`

Result

- The resulting ROC-AUC score is about **0.90424**, which is better than previous blending and averaging attempts on the same data and models.

Stacking variations

- The main variations on stacking involve changing how test data is processed across the layers, whether to use only stacked OOF predictions or also the original features in all the stacking layers, what model to use as the last one, and various tricks in order to prevent overfitting.

Tips

- **Optimization may or may not be used.** Some solutions do not care too much about optimizing single models; others optimize only the last layers; others optimize on the first layers. Based on our experiences, optimization of single models is important and we prefer to do it as early as possible in our stacking ensemble.

- **Models can differ at the different stacking layers, or the same sequence of models can be repeated at every stacking layer.** Here we don’t have a general rule, as it really depends on the problem. The kind of models that are more effective may vary according to the problem. As a general suggestion, putting together gradient boosting solutions and neural networks has never disappointed us.

- **At the first level of the stacking procedure, just create as many models are possible.** For instance, you can try a regression model if your problem is a classification one, and vice versa. You can also use different models with different hyperparameter settings, thus avoiding too much extensive optimization because the stacking will decide for you. If you are using neural networks, just changing the random initialization seed could suffice to create a diverse bag of models. You can also try models using different feature engineering and even use unsupervised learning (like *Mike Kim* did when he used t-SNE dimensions in a solution of his: https://www.kaggle.com/c/otto-group-product-classification-challenge/discussion/14295). The idea is that the selection of all such contributions is done during the second level of the stacking. This means that, at that point, you do not have to experiment any further and you just need to focus on a narrower set of better-perform- ing models. By applying stacking, you can re-use all your experiments and let the stacking decide for you to what degree you should use something in your modeling pipeline.

- Some stacking implementations take on all the features or a selection of them to further stages, reminiscent of skip layers in neural networks. We have noticed that bringing in features at later stages in the stacking can improve your results, but be careful: it also brings in more noise and risk of overfitting.

- Ideally, your OOF predictions should be made from cross-validation schemes with a high number of folds, in other words, between 10 to 20, but we have also seen solutions working with a lower number, such as 5 folds.
- For each fold, bagging the data (resampling with repetition) multiple times for the same model and then averaging all the results from the model (OOF predictions and test pre- dictions) helps to avoid overfitting and produces better results in the end.

- **Beware of early stopping in stacking.** Using it directly on the validation fold may cause a certain degree of overfitting, which may or may not be mitigated in the end by the stacking procedure. We suggest you play it safe and always apply early stopping based on a validation sample from your training folds, not your validation one.

- The possibilities are endless. Once you have grasped the basic concept of this ensembling tech- nique, all you need is to apply your creativity to the problem at hand. We will discuss this key concept in the final section of this chapter, where we will look at a stacking solution for a Kaggle competition.

Creating complex stacking and blending solutions

- At this point in the chapter, you may be wondering to what extent you should apply the tech- niques we have been discussing. In theory, you could use all the ensembling techniques we have presented in any competition on Kaggle, not just tabular ones, but you have to consider a few limiting factors:

- Sometimes, datasets are massive, and training a single model takes a long time.
- In image recognition competitions, you are limited to using deep learning methods.
- Even if you can manage to stack models in a deep learning competition, you have a limited choice for stacking different models. Since you are restricted to deep learning solutions, you can only vary small design aspects of the networks and some hyperparameters (or sometimes just the initialization seed) without degrading the performance. In the end, given the same type of models and more similarities than differences in the architectures, the predictions will tend to be too similar and more correlated than they should be, lim- iting the effectiveness of ensembling.

- Under these conditions, complex stacking regimes are usually not feasible. By contrast, averaging and blending are usually possible when you have large datasets.

- In earlier competitions, as well as in all recent tabular competitions, complex stacking and blend- ing solutions ruled the day. To give you an idea of the complexity and creativity that needs to be put into stacking for a competition, in this last section we will discuss the solution provided by *Gilberto Titericz* (https://www.kaggle.com/titericz) and *Stanislav Semenov* (https://www. kaggle.com/stasg7) to the *Otto Group Product Classification Challenge* (https://www.kaggle. com/c/otto-group-product-classification-challenge). The competition was held in 2015 and its task required classifying over 200,000 products into 9 distinct classes based on 93 features

3 levels stakcing

- The solution proposed by Gilberto and Stanislav comprised three levels:
  1. On the first level, there were 33 models. All the models used quite different algorithms, apart from a cluster of *k*-nearest neighbors where only the *k* parameter varied. They also used unsupervised t-SNE. In addition, they engineered eight features based on dimen- sionality manipulation (computations performed on distances from nearest neighbors and clusters) and on row statistics (the number of non-zero elements in each row). All the OOF predictions and features were passed to the second level.
  2. On the second level, they started optimizing hyperparameters and doing model selec- tion and bagging (they created multiple versions of the same model by resampling and averaged the results for each model). In the end, they had only three models that they re-trained on all the data: an XGBoost, an AdaBoost, and a neural network.
  3. On the third level, they prepared a weighted average of the results by first doing a geo- metric mean of XGBoost and the neural network and then averaging it with the AdaBoost.

- We can learn a lot from this solution, and not just limited to this competition. Aside from the complexity (on the second level, the number of times they resampled was in the order of hundreds for each model), it is noticeable that there are multiple variations on the schemes we discussed in this chapter. Creativity and trial and error clearly dominate the solution. This is quite typical of many Kaggle competitions, where the problems are seldom the same from one competition to another and each solution is unique and not easily repeatable.

- Many AutoML engines, such as **AutoGluon**, more or less explicitly try to take inspiration from such procedures in order to offer a predefined series of automated steps that can ensure you a top result by stacking and blending.

Summary

- In this chapter, we discussed how ensembling multiple solutions works and proposed some basic code examples you can use to start building your own solutions. We started from the ideas that power model ensembles such as random forests and gradient boosting. Then, we moved on to explore the different ensembling approaches, from the simple averaging of test submissions to meta-modeling across multiple layers of stacked models.
- As we discussed at the end, ensembling is more an art form based on some shared common practices. When we explored a successful complex stacking regime that won a Kaggle competition, we were amazed by how the combinations were tailored to the data and the problem itself. You cannot just take a stacking, replicate it on another problem, and hope that it will be the best solution. You can only follow guidelines and find the best solution consisting of averaging/stacking/ blending of diverse models yourself, through lots of experimentation and computational effort.