EXPERT INSIGHT

The Kaggle Workbook

Self-learning exercises and valuable insights for Kaggle data science competitions

Konrad Banachewicz
Luca Massaron

packt

The Kaggle Workbook

Early Access Publication: The Kaggle Workbook

Early Access Production Reference: B19265

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80461-121-0

www.packt.com

Table of Contents

1. The Kaggle Workbook: Self-learning exercises and valuable insights for Kaggle data science competitions
2. 01 The most renowned tabular competition: Porto Seguro's Safe Driver Prediction
    I. Join our book community on Discord
    II. Understanding the competition and the data
    III. Understanding the Evaluation Metric
    IV. Examining the top solution's ideas from Michael Jahrer
    V. Building a LightGBM submission
    VI. Setting up a Denoising Auto-encoder and a DNN
    VII. Ensembling the results
    VIII. Summary
3. 2 The Makridakis Competitions: M5 on Kaggle for Accuracy and Uncertainty
    I. Join our book community on Discord
    II. Understanding the competition and the data
    III. Understanding the Evaluation Metric
    IV. Examining the 4th place solution's ideas from Monsaraida
    V. Computing predictions for specific dates and time horizons
    VI. Assembling public and private predictions
    VII. Summary
4. 03 Cassava Leaf Disease competition
    I. Join our book community on Discord
    II. Understanding the data and metrics
    III. Building a baseline model
    IV. Learnings from top solutions
        i. Pretraining
        ii. TTA
        iii. Transformers
        iv. Ensembling
    V. Summary

The Kaggle Workbook: Self-learning exercises and valuable insights for Kaggle data science competitions

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: The most renown tabular competition: Porto Seguro's Safe Driver Prediction
2. Chapter 2: The Makridakis competitions: M5 on Kaggle for accuracy and uncertainty
3. Chapter 3: Vision competition: Cassava Leaf Disease classification
4. Chapter 4: NLP competition: Quora insincere questions classification

01 The most renowned tabular competition: Porto Seguro's Safe Driver Prediction

Join our book community on Discord

https://packt.link/EarlyAccessCommunity

Learning how to perform top on the leaderboard on any Kaggle competition requires patience, diligence and many attempts in order to learn the best way to compete and achieve top results. For this reason, we have thought of a workbook that can help you build faster those skills by leading you to try some Kaggle competitions of the past and to learn how to make top competitions for them by reading discussions, reusing notebooks, engineering features and training various models.

We start in the book from one of the most renowned tabular competitions, Porto Seguro's Safe Driver Prediction. In this competition, you are asked to solve a common problem in insurance, to figure out who is going to have an auto insurance claim in the next year. Such information is useful in order to increase the insurance fee to drivers more likely to have a claim and to lower it to those less likely to.

In illustrating the key insight and technicalities necessary for cracking this competition we will show you the necessary code and ask you to study and answer about topics to be found on the Kaggle book itself. Therefore, without much more ado, let's start immediately this new learning path of yours.

In this chapter, you will learn:

- How to tune and train a LightGBM model.
- How to build a denoising auto-encoder and how to use it to feed a neural network.
- How to effectively blend models that are quite different from each other.

Understanding the competition and the data

Porto Seguro is the third largest insurance company in Brazil (it operates in Brazil and Uruguay), offering car insurance coverage as well as many other insurance products. Having used analytical methods and machine learning for the past 20 years in order to tailor their prices and make auto insurance coverage more accessible to more drivers. In order to explore new ways to achieve their task, they sponsored a competition (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction), expecting Kagglers to come up with new and better methods of solving some of their core analytical problems.

The competition aims at having Kagglers build a model that predicts the probability that a driver will initiate an auto insurance claim in the next year, which is a quite common kind of task (the sponsor mentions it as a "classical challenge for insurance"). For doing so, the sponsor provided a train and test sets and the competition appears ideal for anyone since the dataset is not very large and it seems very well prepared.

As stated in the page of the competition devoted to presenting the data (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/data):

"features that belong to similar groupings are tagged as such in the feature names (e.g., ind, reg, car, calc). In addition, feature names include the postfix bin to indicate binary features and cat to indicate categorical features. Features without these designations are either continuous or ordinal. Values of -1 indicate that the feature was missing from the observation. The target column signifies whether or not a claim was filed for that policy holder".

The data preparation for the competition had been quite carefully not to leak any information and, though, maintaining secrecy about the meaning of the features, it is quite clear to refer the different used tags to specific kind of features commonly used insurance modeling:

- ind refers to "individual characteristics"
- car refer to "cars characteristics"
- calc to "calculated features"
- reg to "regional/geographic features"

As for the individual features, there has been much speculation during the competition. See for instance https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/41489 or https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/41488 both by Raddar or again the attempts to attribute the feature to Porto Seguro's online quote form https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/41057. In spite of all such efforts, in the end the meaning of the features has remained a mystery up until now.

The interesting fact about this competition is that:

1. the data is real world, though the features are anonymous
2. the data is really very well prepared, without leakages of any sort (no magic features here)
3. the test set not only holds the same categorical levels of the train test, and it also seems to be from the same distribution, though Yuya Yamamoto argues that pre-processing the data with t-SNE leads to a failing adversarial validation test (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/44784) .

   As a first exercise, referring to the contents and the code on the Kaggle Book related to adversarial validation (starting from page 179), prove that train and test data most probably originated from the same data distribution.

An interesting post by Tilii (Mensur Dlakic, associate Professor at Montana State University: https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/42197) demonstrates using tSNE that "there are many people who are very similar in terms of their insurance parameters, yet some of them will file a claim and others will not". What Tilii mentions is quite typical of what happens in insurance where to certain priors (insurance parameters) there is sticked the same probability of something happening but that event will happen or not based on how long we observe the situation.

Take for instance IoT and telematic data in insurance. It is quite common to analyze a driver's behavior in order to predict if she or he will file a claim in the future. If your observation period is too short (for instance one year as in the case of this competition), it may happen that even very bad drivers won't have a claim because it is a matter of not too high probability of the outcome that can become a reality only after a certain amount of time. Similar ideas are discussed by Andy Harless (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/42735) which argues instead that the real task of the competition is to guess *"the value of a latent continuous variable that determines which drivers are more likely to have accidents"* because actually *"making a claim is not a characteristic of a driver; it's a result of chance"*.

Understanding the Evaluation Metric

The metric used in the competition is the "normalized Gini coefficient" (named for the similar Gini coefficient/index used in Economics), which has been previously used in another competition, the Allstate Claim Prediction Challenge (https://www.kaggle.com/competitions/ClaimPredictionChallenge). From that competition, we can get a very clear explanation about what this metric is about:

When you submit an entry, the observations are sorted from "largest prediction" to "smallest prediction". This is the only step where your predictions come into play, so only the order determined by your predictions matters. Visualize the observations arranged from left to right, with the largest predictions on the left. We then move from left to right, asking "In the leftmost x% of the data, how much of the actual observed loss have you accumulated?" With no model, you can expect to accumulate 10% of the loss in 10% of the predictions, so no model (or a "null" model) achieves a straight line. We call the area between your curve and this straight line the Gini coefficient.

There is a maximum achievable area for a "perfect" model. We will use the normalized Gini coefficient by dividing the Gini coefficient of your model by the Gini coefficient of the perfect model.

Another good explanation is provided in the competition by the notebook by Kilian Batzner: https://www.kaggle.com/code/batzner/gini-coefficient-an-intuitive-explanation that, though means of plots and toy examples tries to give a sense to a not so common metric but in the actuarial departments of insurance companies.

> In chapter 5 of the Kaggle Book (pages 95 onward), we explained how to deal with competition metrics, especially if they are new and generally unknown. As an exercise, can you find out how many competitions on Kaggle have used the normalized Gini coefficient as an evaluation metric?

The metric can be approximated by the Mann–Whitney U non-parametric statistical test and by the ROC-AUC score because it approximately corresponds to 2 * ROC-AUC - 1. Hence, maximizing the ROC-AUC is the same as maximizing the Normalized Gini coefficient (for a reference see the “Relation to other statistical measures” in the Wikipedia entry: https://en.wikipedia.org/wiki/Gini_coefficient).

The metric can also be approximately expressed as the covariance of scaled prediction rank and scaled target value, resulting in a more understandable rank association measure (see Dmitriy Guller: https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/40576)

From the point of view of the objective function, you can optimize for the binary logloss (as you would do in a classification problem). Nor ROC-AUC nor the normalized Gini coefficient are differentiable and they may be used only for metric evaluation on the validation set (for instance for early stopping or for reducing the learning rate in a neural network). However, optimizing for the logloss doesn’t always improve the ROC-AUC and the normalized Gini coefficients. There is actually a differentiable ROC-AUC approximation (Calders, Toon, and Szymon Jaroszewicz. "Efficient AUC optimization for classification." European conference on principles of data mining and knowledge discovery. Springer, Berlin, Heidelberg, 2007 https://link.springer.com/content/pdf/10.1007/978-3-540-74976-9_8.pdf). However, it seems that it is not necessary to use anything different from logloss as objective function and ROC-AUC or normalized Gini coefficient as evaluation metric in the competition.

There are actually a few Python implementations among the notebooks. We have used here and we suggest the work by CPMP (https://www.kaggle.com/code/cpmpml/extremely-fast-gini-computation/notebook) that uses Numba for speeding up computations: it is both exact and fast.

Examining the top solution’s ideas from Michael Jahrer

Michael Jahrer (https://www.kaggle.com/mjahrer, competitions grandmaster and one of the winners of the Netflix Prize in the team "BellKor's Pragmatic Chaos"), has led for long time and by a fair margin

the public leaderboard during the competition and, when finally the private one has been disclosed, has been declared the winner.

Shortly after, in the discussion forum, he published a short summary of his solution that has become a reference for many Kagglers because of his smart usage of denoising autoencoders and neural networks (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/44629). Although Michael hasn't accompanied his post by any Python code regarding his solution (he quoted it as an "old school" and "low level" one, being directly written in C++/CUDA with no Python), his writing is quite rich in references to what models he has used as well to theirs hyper-parameters and architectures.

First, Michael explains that his solution is composed of a blend of six models (one LightGBM model and five neural networks). Moreover, since no advantage could be gained by weighting the contributions of each model to the blend (as well as doing linear and non-linear stacking) likely because of overfitting, he states that he resorted to just a plain blend of models (an arithmetic mean) that have been built from different seeds.

Such insight makes the task much easier for us in order to replicate his approach, also because he also mentions that just having blend together the LightGBM's results with one from the neural networks he built would have been enough to guarantee the first place in the competition. That will limit our exercise work to two good single models instead of a host of them. In addition, he mentioned having done little data processing, but dropping some columns and one-hot encoding categorical features.

Building a LightGBM submission

Our exercise starts by working out a solution based on LightGBM. You can find the code already set for execution at the Kaggle Notebook at this address: https://www.kaggle.com/code/lucamassaron/workbook-lgb. Although we made the code readily available, we instead suggest you to type or copy the code directly from the book and execute it cell by cell, understanding what each line of code does and you could furthermore personalise the solution and have it perform even better.

We start by importing key packages (Numpy, Pandas, Optuna for hyper-parameter optimization, LightGBM and some utility functions). We also define a configuration class and instantiate it. We will discuss the parameters defined into the configuration class during the exploration of the code as we progress. What is important to remark here is that by using a class containing all your parameters it will be easier for you to modify them in a consistent way along the code. In the heat of a competition, it is easy to forget to update a parameter that it is referred to in multiple places in the code and it is always difficult to set the parameters when they are dispersed among cells and functions. A configuration class can save you a lot of effort and spare you mistakes along the way.

```
import numpy as np
import pandas as pd
import optuna
import lightgbm as lgb
from path import Path
from sklearn.model_selection import StratifiedKFold
class Config:
    input_path = Path('../input/porto-seguro-safe-driver-prediction')
    optuna_lgb = False
    n_estimators = 1500
    early_stopping_round = 150
    cv_folds = 5
    random_state = 0
    params = {'objective': 'binary',
              'boosting_type': 'gbdt',
              'learning_rate': 0.01,
              'max_bin': 25,
```

```
                'num_leaves': 31,
                'min_child_samples': 1500,
                'colsample_bytree': 0.7,
                'subsample_freq': 1,
                'subsample': 0.7,
                'reg_alpha': 1.0,
                'reg_lambda': 1.0,
                'verbosity': 0,
                'random_state': 0}

config = Config()
```

The next step requires importing train, test and the sample submission datasets. By doing so by pandas csv reading function, we also set the index of the uploaded dataframes to the identifier (the 'id' column) of each data example.

Since features that belong to similar groupings are tagged (using ind, reg, car, calc tags in their labels) and also binary and categorical features are easy to locate (they use the bin and cat, respectively, tags in their labels), we can enumerate them and record them into lists.

```
train = pd.read_csv(config.input_path / 'train.csv', index_col='id')
test = pd.read_csv(config.input_path / 'test.csv', index_col='id')
submission = pd.read_csv(config.input_path / 'sample_submission.csv', index_col='id')
calc_features = [feat for feat in train.columns if "_calc" in feat]
cat_features = [feat for feat in train.columns if "_cat" in feat]
```

Then, we just extract the target (a binary target of 0s and 1s) and remove it from the train dataset.

```
target = train["target"]
train = train.drop("target", axis="columns")
```

At this point, as pointed out by Michael Jahrer, we can drop the calc features. This idea has recurred a lot during the competition (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/41970), especially in notebooks, both because it could be empirically verified that dropping them improved the public leaderboard score, both because they performed poorly in gradient boosting models (their importance is always below the average). We can argue that, since they are engineered features, they do not contain new information in respect of their origin features but they just add noise to any model trained comprising them.

```
train = train.drop(calc_features, axis="columns")
test = test.drop(calc_features, axis="columns")
```

> During the competition, Tilii has tested feature elimination using Boruta (https://github.com/scikit-learn-contrib/boruta_py). You can find his kernel here: https://www.kaggle.com/code/tilii7/boruta-feature-elimination/notebook. As you can check, there is no calc_feature considered as a confirmed feature by Boruta.
>
> > Exercise: based on the suggestions provided in the Kaggle Book at page 220 ("Using feature importance to evaluate your work"), as an exercise, code your own feature selection notebook for this competition and check what features should be kept and what should be discarded.

Categorical features are instead one-hot encoded. Since we want to retrain their labels, and since the same levels are present both in train and test (the result of a careful train/test split between the two arranged by the Porto Seguro team), instead of the usual Scikit-Learn OneHotEncoder (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) we use the pandas get_dummies function (https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html). Since the pandas function may produce different encodings if the features and their levels differ from train to test set, we assert a check on the one hot encoding resulting in the same for both.

```
train = pd.get_dummies(train, columns=cat_features)
test = pd.get_dummies(test, columns=cat_features)
```

```
assert((train.columns==test.columns).all())
```

After one hot encoding the categorical features we have completed doing all the data processing. We proceed to define our evaluation metric, the Normalized Gini Coefficient, as previously discussed. Since we are going to use a LightGBM model, we have to add a suitable wrapper (`gini_lgb`) in order to return to the GBM algorithm the evaluation of the training and the validation sets in a form that could works with it (see: https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.Booster.html?highlight=higher_better#lightgbm.Booster.eval - "Each evaluation function should accept two parameters: preds, eval_data, and return `(eval_name, eval_result, is_higher_better)` or list of such tuples").

```
from numba import jit
@jit
def eval_gini(y_true, y_pred):
    y_true = np.asarray(y_true)
    y_true = y_true[np.argsort(y_pred)]
    ntrue = 0
    gini = 0
    delta = 0
    n = len(y_true)
    for i in range(n-1, -1, -1):
        y_i = y_true[i]
        ntrue += y_i
        gini += y_i * delta
        delta += 1 - y_i
    gini = 1 - 2 * gini / (ntrue * (n - ntrue))
    return gini
def gini_lgb(y_true, y_pred):
    eval_name = 'normalized_gini_coef'
    eval_result = eval_gini(y_true, y_pred)
    is_higher_better = True
    return eval_name, eval_result, is_higher_better
```

As for the training parameters, we found that the parameters suggested by Michael Jahrer in his post (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/44629) work perfectly. You may also try to come up with the same parameters or with similarly performing ones by performing a search by optuna (https://optuna.org/) if you set the `optuna_lgb` flag to True in the config class. Here the optimization tries to find the best values for key parameters such as the learning rate and the regularization parameters, based on a 5-fold cross-validation test on training data. In order to speed up things, early stopping on the validation itself is taken into account (which, we are aware, could actually advantage some values that can overfit better the validation fold - a good alternative could be to remove the early stopping callback and keep a fixed number of rounds for the training).

```
if config.optuna_lgb:

    def objective(trial):
        params = {
    'learning_rate': trial.suggest_float("learning_rate", 0.01, 1.0),
    'num_leaves': trial.suggest_int("num_leaves", 3, 255),
    'min_child_samples': trial.suggest_int("min_child_samples",
                                           3, 3000),
    'colsample_bytree': trial.suggest_float("colsample_bytree",
                                            0.1, 1.0),
    'subsample_freq': trial.suggest_int("subsample_freq", 0, 10),
    'subsample': trial.suggest_float("subsample", 0.1, 1.0),
    'reg_alpha': trial.suggest_loguniform("reg_alpha", 1e-9, 10.0),
    'reg_lambda': trial.suggest_loguniform("reg_lambda", 1e-9, 10.0),
        }

        score = list()
        skf = StratifiedKFold(n_splits=config.cv_folds, shuffle=True,
                              random_state=config.random_state)
        for train_idx, valid_idx in skf.split(train, target):
            X_train = train.iloc[train_idx]
            y_train = target.iloc[train_idx]
```

```
            X_valid = train.iloc[valid_idx]
            y_valid = target.iloc[valid_idx]
            model = lgb.LGBMClassifier(**params,
                                       n_estimators=1500,
                                       early_stopping_round=150,
                                       force_row_wise=True)
            callbacks=[lgb.early_stopping(stopping_rounds=150,
                                          verbose=False)]
            model.fit(X_train, y_train,
                      eval_set=[(X_valid, y_valid)],
                      eval_metric=gini_lgb, callbacks=callbacks)

            score.append(
                model.best_score_['valid_0']['normalized_gini_coef'])
        return np.mean(score)
    study = optuna.create_study(direction="maximize")
    study.optimize(objective, n_trials=300)
    print("Best Gini Normalized Score", study.best_value)
    print("Best parameters", study.best_params)

    params = {'objective': 'binary',
              'boosting_type': 'gbdt',
              'verbosity': 0,
              'random_state': 0}

    params.update(study.best_params)

else:
    params = config.params
```

During the competition, Tilii has tested feature elimination using Boruta (https://github.com/scikit-learn-contrib/boruta_py). You can find his kernel here: https://www.kaggle.com/code/tilii7/boruta-feature-elimination/notebook. As you can check, there is no calc_feature considered as a confirmed feature by Boruta.

> In the Kaggle Book, we explain about both hyper-parameter optimization (pages 241 onward) and provide some key hyper-parameters for the LightGBM model. As an exercise, try to improve the hyper-parameter search by reducing or increasing the explored parameters and by trying alternative methods such as the random search or the halving search from Scikit-Learn (pages 245-246).

Once we have our best parameters (or we simply try Jahrer's ones), we are ready to train and predict. Our strategy, as suggested by the best solution, is to train a model on each cross-validation folds and use that fold to contribute to an average of test predictions. The snippet of code will produce both the test predictions and the out of fold predictions on the train set, later useful for figuring out how to ensemble the results.

```
preds = np.zeros(len(test))
oof = np.zeros(len(train))
metric_evaluations = list()
skf = StratifiedKFold(n_splits=config.cv_folds, shuffle=True, random_state=config.random_stat
for idx, (train_idx, valid_idx) in enumerate(skf.split(train,
                                                       target)):
    print(f"CV fold {idx}")
    X_train, y_train = train.iloc[train_idx], target.iloc[train_idx]
    X_valid, y_valid = train.iloc[valid_idx], target.iloc[valid_idx]

    model = lgb.LGBMClassifier(**params,
                               n_estimators=config.n_estimators,
                    early_stopping_round=config.early_stopping_round,
                               force_row_wise=True)

    callbacks=[lgb.early_stopping(stopping_rounds=150),
               lgb.log_evaluation(period=100, show_stdv=False)]

    model.fit(X_train, y_train,
```

```
            eval_set=[(X_valid, y_valid)],
            eval_metric=gini_lgb, callbacks=callbacks)
    metric_evaluations.append(
              model.best_score_['valid_0']['normalized_gini_coef'])
    preds += (model.predict_proba(test,
            num_iteration=model.best_iteration_)[:,1]
            / skf.n_splits)
    oof[valid_idx] = model.predict_proba(X_valid,
                   num_iteration=model.best_iteration_)[:,1]
```

The model training shouldn't take too long. In the end you can get reported the Normalized Gini Coefficient obtained during the cross-validation procedure.

```
print(f"LightGBM CV normalized Gini coefficient:
        {np.mean(metric_evaluations):0.3f}
        ({np.std(metric_evaluations):0.3f})")
```

The results are quite encouraging because the average score is 0.289 and the standard deviation of the values is quite small.

```
LightGBM CV Gini Normalized Score: 0.289 (0.015)
```

All that is left is to save the out-of-fold and the test predictions as a submission and to verify the results on the public and private leaderboards.

```
submission['target'] = preds
submission.to_csv('lgb_submission.csv')
oofs = pd.DataFrame({'id':train_index, 'target':oof})
oofs.to_csv('dnn_oof.csv', index=False)
```

The obtained public score should be around 0.28442. The associated private score is about 0.29121, placing you in the 30th position in the final leaderboard. A quite good result, but we still have to blend it with a different model, a neural network.

Bagging the training set (i.e. taking multiple bootstraps of the training data and training multiple models based on the bootstraps) should increase the performance, though, as Michael Jahrer himself noted in his post, not all that much.

Setting up a Denoising Auto-encoder and a DNN

The next step is not to set up a denoising auto-encoder (DAE) and a neural network that can learn and predict from it. You can find the running code at this notebook: https://www.kaggle.com/lucamassaron/workbook-dae . The notebook can be run in GPU mode (speedier), but it can also run in CPU one with some slight modifications.

> You can read more about denoising auto-encoders as being used in Kaggle competitions in the Kaggle Book, at pages 226 and following.

Actually there are no examples around reproducing Michael Jahrer's approach in the competition using DAEs, but we tooked example from a working TensorFlow implementation in another competition by OsciiArt (https://www.kaggle.com/code/osciiart/denoising-autoencoder).

Here we start by importing all the necessary packages, especially TensorFlow and Keras. Since we are going to create multiple neural networks, we point out to TensorFlow not to use all the GPU memory available by using the experimental `set_memory_growth` command. Such will avoid having memory overflow problems along the way. We also record the Leaky Relu activation as a custom one, so we can just mention it as an activation by a string in Keras layers.

```
import numpy as np
import pandas as pd
```

```
from matplotlib import pyplot as plt
from path import Path
import gc
import optuna
from sklearn.model_selection import StratifiedKFold
from scipy.special import erfinv
import tensorflow as tf
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
from tensorflow import keras
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Dense, BatchNormalization, Dropout
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.regularizers import l2
from tensorflow.keras.metrics import AUC
from tensorflow.keras.utils import get_custom_objects
from tensorflow.keras.layers import Activation, LeakyReLU
get_custom_objects().update({'leaky-relu': Activation(LeakyReLU(alpha=0.2))})
```

Related to our intention of creating multiple neural networks without running out of memory, we also define a simple function for cleaning the memory in GPU and removing models that are no longer needed.

```
def gpu_cleanup(objects):
    if objects:
        del(objects)
    K.clear_session()
    gc.collect()
```

We also reconfigure the Config class in order to take into account multiple parameters related to the denoising auto-encoder and the neural network. As previously stated about the LightGBM, having all the parameters in a single place renders simpler modifying them in a consistent way.

```
class Config:
    input_path = Path('../input/porto-seguro-safe-driver-prediction')
    dae_batch_size = 128
    dae_num_epoch = 50
    dae_architecture = [1500, 1500, 1500]
    reuse_autoencoder = False
    batch_size = 128
    num_epoch = 150
    units = [64, 32]
    input_dropout=0.06
    dropout=0.08
    regL2=0.09
    activation='selu'

    cv_folds = 5
    nas = False
    random_state = 0

config = Config()
```

As previously, we load the datasets and proceed processing the features by removing the calc features and one hot encoding the categorical ones. We leave missing cases valued at -1, as Michael Jahrer pointed out in his solution.

```
train = pd.read_csv(config.input_path / 'train.csv', index_col='id')
test = pd.read_csv(config.input_path / 'test.csv', index_col='id')
submission = pd.read_csv(config.input_path / 'sample_submission.csv', index_col='id')
calc_features = [feat for feat in train.columns if "_calc" in feat]
cat_features = [feat for feat in train.columns if "_cat" in feat]
target = train["target"]
train = train.drop("target", axis="columns")
train = train.drop(calc_features, axis="columns")
test = test.drop(calc_features, axis="columns")
```

```
train = pd.get_dummies(train, columns=cat_features)
test = pd.get_dummies(test, columns=cat_features)
assert((train.columns==test.columns).all())
```

However, differently from our previous approach, we have to rescale all the features that are not binary or one hot-encoded categorical. Rescaling will allow the optimization algorithm of both the auto-encoder and the neural network to converge to a good solution faster because it will have to work with values set into a comparable and predefined range. Instead of using statistical normalisation, GaussRank is a procedure that also allows the modification of the distribution of transformed variables into a Gaussian one.

As also stated in some papers, such as in the Batch Normalization paper: https://arxiv.org/pdf/1502.03167.pdf, neural networks perform even better if you provide them a Gaussian input. Accordingly to this NVidia blog post, https://developer.nvidia.com/blog/gauss-rank-transformation-is-100x-faster-with-rapids-and-cupy/, GaussRank works most of the times but when features are already normally distributed or are extremely asymmetrical (in such cases applying the transformation may lead to worsened performances).

```
print("Applying GaussRank to columns: ", end='')
to_normalize = list()
for k, col in enumerate(train.columns):
    if '_bin' not in col and '_cat' not in col and '_missing' not in col:
        to_normalize.append(col)
print(to_normalize)
def to_gauss(x): return np.sqrt(2) * erfinv(x)
def normalize(data, norm_cols):
    n = data.shape[0]
    for col in norm_cols:
        sorted_idx = data[col].sort_values().index.tolist()
        uniform = np.linspace(start=-0.99, stop=0.99, num=n)
        normal = to_gauss(uniform)
        normalized_col = pd.Series(index=sorted_idx, data=normal)
        data[col] = normalized_col
    return data
train = normalize(train, to_normalize)
test = normalize(test, to_normalize)
```

We can apply the GaussRank transformation separately on the train and test features on all the numeric features of our dataset:

```
Applying GaussRank to columns: ['ps_ind_01', 'ps_ind_03', 'ps_ind_14', 'ps_ind_15', 'ps_reg_(
```

When normalising the features, we simply turn our data into a NumPy array of float32 values, the ideal input for a GPU.

```
features = train.columns
train_index = train.index
test_index = test.index
train = train.values.astype(np.float32)
test = test.values.astype(np.float32)
```

Next, we just prepare some useful functions such as the evaluation function, the normalized Gini coefficient and a plotting function helpful representing a Keras model history of fitting on both training and validation sets.

```
def plot_keras_history(history, measures):
    rows = len(measures) // 2 + len(measures) % 2
    fig, panels = plt.subplots(rows, 2, figsize=(15, 5))
    plt.subplots_adjust(top = 0.99, bottom=0.01,
                        hspace=0.4, wspace=0.2)
    try:
        panels = [item for sublist in panels for item in sublist]
    except:
        pass
```

```
    for k, measure in enumerate(measures):
        panel = panels[k]
        panel.set_title(measure + ' history')
        panel.plot(history.epoch, history.history[measure],
                   label="Train "+measure)
        try:
            panel.plot(history.epoch,
                       history.history["val_"+measure],
                       label="Validation "+measure)
        except:
            pass
        panel.set(xlabel='epochs', ylabel=measure)
        panel.legend()

    plt.show(fig)
from numba import jit
@jit
def eval_gini(y_true, y_pred):
    y_true = np.asarray(y_true)
    y_true = y_true[np.argsort(y_pred)]
    ntrue = 0
    gini = 0
    delta = 0
    n = len(y_true)
    for i in range(n-1, -1, -1):
        y_i = y_true[i]
        ntrue += y_i
        gini += y_i * delta
        delta += 1 - y_i
    gini = 1 - 2 * gini / (ntrue * (n - ntrue))
    return gini
```

The next functions are actually a bit more complex and more related to the functioning of both the denoising auto-encoder and the supervised neural network. The `batch_generator` is a function that will create a generator providing shuffled chunks of the data based on a batch size. It isn't actually used as a stand-alone generator but as part of a more complex batch generator that we are soon going to describe, the `mixup_generator`.

```
def batch_generator(x, batch_size, shuffle=True, random_state=None):
    batch_index = 0
    n = x.shape[0]
    while True:
        if batch_index == 0:
            index_array = np.arange(n)
            if shuffle:
                np.random.seed(seed=random_state)
                index_array = np.random.permutation(n)
        current_index = (batch_index * batch_size) % n
        if n >= current_index + batch_size:
            current_batch_size = batch_size
            batch_index += 1
        else:
            current_batch_size = n - current_index
            batch_index = 0
        batch = x[index_array[current_index: current_index + current_batch_size]]
        yield batch
```

The `mixup_generator`, another generator, returns batches of data whose values have been partially swapped in order to create some noise and augment the data in order for the denoising auto-encoder (DAE) not to overfit the train set. It works based on a swap rate, fixed at 15% of features as suggested by Michael Jahrer.

The function generates two distinct batches of data, one to be released to the model and another to be used as a source for the value to be swapped in the batch to be released. Based on a random choice, whose base probability is the swap rate, at each batch a certain number of features is decided to be swapped between the two batches.

Such assures that the DAE cannot always rely on the same features (since they can be randomly swapped from time to time) but it has to concentrate on the whole of the features (something similar to dropout in a certain sense). in order to find relations between them and reconstruct correctly the data at the end of the process.

```
def mixup_generator(X, batch_size, swaprate=0.15, shuffle=True, random_state=None):
    if random_state is None:
        random_state = np.randint(0, 999)
    num_features = X.shape[1]
    num_swaps = int(num_features * swaprate)
    generator_a = batch_generator(X, batch_size, shuffle,
                                  random_state)
    generator_b = batch_generator(X, batch_size, shuffle,
                                  random_state + 1)
    while True:
        batch = next(generator_a)
        mixed_batch = batch.copy()
        effective_batch_size = batch.shape[0]
        alternative_batch = next(generator_b)
        assert((batch != alternative_batch).any())
        for i in range(effective_batch_size):
            swap_idx = np.random.choice(num_features, num_swaps,
                                        replace=False)
            mixed_batch[i, swap_idx] = alternative_batch[i, swap_idx]
        yield (mixed_batch, batch)
```

The `get_DAE` is the function that builds the denoising auto-encoder. It accepts a parameter for defining the architecture which in our case has been set to three layers of 1500 nodes each (as suggested from Michael Jahrer's solution). The first layer should act as an encoder, the second is a bottleneck layer ideally containing the latent features capable of expressing the information in the data, the last layer is a decoding layer, capable of reconstructing the initial input data. The three layers have a relu activation function, no bias and each one is followed by a batch normalization layer. The final output with the reconstructed input data has a linear activation. The training is provided using an adam optimizer with standard settings (the optimised cost function is the mean squared error - mse).

```
def get_DAE(X, architecture=[1500, 1500, 1500]):
    features = X.shape[1]
    inputs = Input((features,))
    for i, nodes in enumerate(architecture):
        layer = Dense(nodes, activation='relu',
                      use_bias=False, name=f"code_{i+1}")
        if i==0:
            x = layer(inputs)
        else:
            x = layer(x)
        x = BatchNormalization()(x)
    outputs = Dense(features, activation='linear')(x)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='mse',
                  metrics=['mse', 'mae'])
    return model
```

The `extract_dae_features` function is reported here only for educational purposes. The function helps in the extraction of the values of specific layers of the trained denoising auto-encoder. The extraction works by building a new model combining the DAE input layer and the desired output one. A simple predict will then extract the values we need (the predict also allows fixing the preferred batch size in order to fit any memory requirement).

In the case of the competition, given the number of observations and the number of the features to be taken out from the auto-encoder, if we were to use this function, the resulting dense matrix would be too large to be handled by the memory of a Kaggle notebook. For this reason our strategy won't be to transform the original data into the auto-encoder node values of the bottleneck layer but to fuse the

auto-encoder with its frozen layers up to the bottleneck with the supervised neural network, as we will be discussing soon.

```
def extract_dae_features(autoencoder, X, layers=[3]):
    data = []
    for layer in layers:
        if layer==0:
            data.append(X)
        else:
            get_layer_output = Model([autoencoder.layers[0].input],
                                     [autoencoder.layers[layer].output])
            layer_output = get_layer_output.predict(X,
                                                    batch_size=128)
            data.append(layer_output)
    data = np.hstack(data)
    return data
```

To complete the work with the DAE, we have a final function wrapping all the previous ones into an unsupervised training procedure (at least partially unsupervised since there is an early stop monitor set on a validation set). The function sets up the mix-up generator, creates the denoising auto-encoder architecture and then trains it, monitoring its fit on a validation set for an early stop if there are signs of overfitting. Finally, before returning the trained DAE, it plots a graph of the training and validation fit and it stores the model on disk.

Even if we try to fix a seed on this model, contrary to the LightGBM model, the results are extremely variable and they may influence the final ensemble results. Though the result will be a high scoring one, it may land higher or lower on the private leaderboard, though the results obtained on the public one are very correlated to the public leaderboard and it will be easy for you to always pick up the best final submission based on its public results.

```
def autoencoder_fitting(X_train, X_valid, filename='dae',
                        random_state=None, suppress_output=False):
    if suppress_output:
        verbose = 0
    else:
        verbose = 2
        print("Fitting a denoising autoencoder")
    tf.random.set_seed(seed=random_state)
    generator = mixup_generator(X_train,
                                batch_size=config.dae_batch_size,
                                swaprate=0.15,
                                random_state=config.random_state)

    dae = get_DAE(X_train, architecture=config.dae_architecture)
    steps_per_epoch = np.ceil(X_train.shape[0] /
                              config.dae_batch_size)
    early_stopping = EarlyStopping(monitor='val_mse',
                                   mode='min',
                                   patience=5,
                                   restore_best_weights=True,
                                   verbose=0)
    history = dae.fit(generator,
                      steps_per_epoch=steps_per_epoch,
                      epochs=config.dae_num_epoch,
                      validation_data=(X_valid, X_valid),
                      callbacks=[early_stopping],
                      verbose=verbose)
    if not suppress_output: plot_keras_history(history,
                                               measures=['mse', 'mae'])
    dae.save(filename)
    return dae
```

Having dealt with the DAE, we take the chance also to define the supervised neural model down the line that should predict our claim expectations. As a first step, we define a function to define a single layer of the work:

- Random normal initialization, since empirically it has been found to converge to better results in this problem
- A dense layer with L2 regularization and parametrable activation function
- An excludable and tunable dropout

Here is the code for creating the dense blocks:

```
def dense_blocks(x, units, activation, regL2, dropout):
    kernel_initializer = keras.initializers.RandomNormal(mean=0.0,
                                stddev=0.1, seed=config.random_state)
    for k, layer_units in enumerate(units):
        if regL2 > 0:
            x = Dense(layer_units, activation=activation,
                      kernel_initializer=kernel_initializer,
                      kernel_regularizer=l2(regL2))(x)
        else:
            x = Dense(layer_units,
                      kernel_initializer=kernel_initializer,
                      activation=activation)(x)
        if dropout > 0:
            x = Dropout(dropout)(x)
    return x
```

As you may have already noticed, the function defining the single layer is quite customizable. The same goes for the wrapper architecture function, taking inputs for the number of layers and units in them, dropout probabilities, regularization and activation type. The idea is to be able to run a Neural Architecture Search (NAS) and figure out what configuration should perform better in our problem.

As a final note on the function, among the inputs it is required to provide the trained DAE because its inputs are used as the neural network model inputs while its first layers are connected to the DAE's outputs. In such a way we are de facto concatenating the two models into one (the DAE weights are frozen anyway and not trainable, though). This solution has been devised in order to avoid having to transform all your training data but only the single batches that the neural network is processing, thus saving memory in the system.

```
def dnn_model(dae, units=[4500, 1000, 1000],
            input_dropout=0.1, dropout=0.5,
            regL2=0.05,
            activation='relu'):

    inputs = dae.get_layer("code_2").output
    if input_dropout > 0:
        x = Dropout(input_dropout)(inputs)
    else:
        x = tf.keras.layers.Layer()(inputs)
    x = dense_blocks(x, units, activation, regL2, dropout)
    outputs = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=dae.input, outputs=outputs)
    model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
                loss=keras.losses.binary_crossentropy,
                metrics=[AUC(name='auc')])
    return model
```

We conclude with a wrapper for the training process, including all the steps in order to train the entire pipeline on a cross-validation fold.

```
def model_fitting(X_train, y_train, X_valid, y_valid, autoencoder,
                  filename, random_state=None, suppress_output=False):
        if suppress_output:
            verbose = 0
        else:
            verbose = 2
            print("Fitting model")
        early_stopping = EarlyStopping(monitor='val_auc',
                                       mode='max',
```

```
                                patience=10,
                                restore_best_weights=True,
                                verbose=0)
        rlrop = ReduceLROnPlateau(monitor='val_auc',
                                  mode='max',
                                  patience=2,
                                  factor=0.75,
                                  verbose=0)

        tf.random.set_seed(seed=random_state)
        model = dnn_model(autoencoder,
                    units=config.units,
                    input_dropout=config.input_dropout,
                    dropout=config.dropout,
                    regL2=config.regL2,
                    activation=config.activation)

        history = model.fit(X_train, y_train,
                            epochs=config.num_epoch,
                            batch_size=config.batch_size,
                            validation_data=(X_valid, y_valid),
                            callbacks=[early_stopping, rlrop],
                            shuffle=True,
                            verbose=verbose)
        model.save(filename)

        if not suppress_output:
            plot_keras_history(history, measures=['loss', 'auc'])
        return model, history
```

Since our DAE implementation is surely different from Jahrer's, although the idea behind is the same, we cannot rely completely on his indications on the architecture of the supervised neural network and we have to look for the ideal ones as we have been looking for the best hyper-parameters in the LightGBM model. Using Optuna and leveraging the multiple parameters that we set open for configuring the network's architecture, we can run this code snippet for some hours and get an idea about what could work better.

In our experiments we that:

- we should use a two layer network with few nodes, 64 and 32 respectively.
- input dropout, dropout between layers and some L2 regularization do help.
- It is better to use the SELU activation function.

Here is the code snippet for running the entire optimization experiments:

```
if config.nas is True:
    def evaluate():
        metric_evaluations = list()
        skf = StratifiedKFold(n_splits=config.cv_folds, shuffle=True, random_state=config.rar
        for k, (train_idx, valid_idx) in enumerate(skf.split(train, target)):

            X_train, y_train = train[train_idx, :], target[train_idx]
            X_valid, y_valid = train[valid_idx, :], target[valid_idx]
            if config.reuse_autoencoder:
                autoencoder = load_model(f"./dae_fold_{k}")
            else:
                autoencoder = autoencoder_fitting(X_train, X_valid,
                                                  filename=f'./dae_fold_{k}',
                                                  random_state=config.random_state,
                                                  suppress_output=True)

            model, _ = model_fitting(X_train, y_train, X_valid, y_valid,
                                     autoencoder=autoencoder,
                                     filename=f"dnn_model_fold_{k}",
                                     random_state=config.random_state,
                                     suppress_output=True)
```

```
            val_preds = model.predict(X_valid, batch_size=128, verbose=0)
            best_score = eval_gini(y_true=y_valid, y_pred=np.ravel(val_preds))
            metric_evaluations.append(best_score)

            gpu_cleanup([autoencoder, model])

        return np.mean(metric_evaluations)
    def objective(trial):
        params = {
                'first_layer': trial.suggest_categorical("first_layer", [8, 16, 32, 64, 128,
                'second_layer': trial.suggest_categorical("second_layer", [0, 8, 16, 32, 64,
                'third_layer': trial.suggest_categorical("third_layer", [0, 8, 16, 32, 64, 12
                'input_dropout': trial.suggest_float("input_dropout", 0.0, 0.5),
                'dropout': trial.suggest_float("dropout", 0.0, 0.5),
                'regL2': trial.suggest_uniform("regL2", 0.0, 0.1),
                'activation': trial.suggest_categorical("activation", ['relu', 'leaky-relu',
        }
        config.units = [nodes for nodes in [params['first_layer'], params['second_layer'], pa
        config.input_dropout = params['input_dropout']
        config.dropout = params['dropout']
        config.regL2 = params['regL2']
        config.activation = params['activation']

        return evaluate()
    study = optuna.create_study(direction="maximize")
    study.optimize(objective, n_trials=60)
    print("Best Gini Normalized Score", study.best_value)
    print("Best parameters", study.best_params)
    config.units = [nodes for nodes in [study.best_params['first_layer'], study.best_params['
    config.input_dropout = study.best_params['input_dropout']
    config.dropout = study.best_params['dropout']
    config.regL2 = study.best_params['regL2']
    config.activation = study.best_params['activation']
```

If you are looking for more information about more on Neural Architecture Search (NAS), you can have a look at the Kaggle Book, at pages 276 onward. In the case of the DAE and the supervised neural network, it is critical to look for the best architecture since we are implementing something surely different from Michael Jahrer solution.

As an exercise, try to improve the hyper-parameter search by using KerasTuner (to be found on pages 285 onward in the Kaggle Book), a fast solution for optimizing neural networks that has seen the important contribution of François Chollet, the creator of Keras.

Having finally set everything ready, we are set to start the training. In about one hour, on a Kaggle notebook with GPU, you can obtain complete test and out-of-fold predictions.

```
preds = np.zeros(len(test))
oof = np.zeros(len(train))
metric_evaluations = list()
skf = StratifiedKFold(n_splits=config.cv_folds, shuffle=True, random_state=config.random_stat
for k, (train_idx, valid_idx) in enumerate(skf.split(train, target)):
    print(f"CV fold {k}")

    X_train, y_train = train[train_idx, :], target[train_idx]
    X_valid, y_valid = train[valid_idx, :], target[valid_idx]
    if config.reuse_autoencoder:
        print("restoring previously trained dae")
        autoencoder = load_model(f"./dae_fold_{k}")
    else:
        autoencoder = autoencoder_fitting(X_train, X_valid,
                                          filename=f'./dae_fold_{k}',
                                          random_state=config.random_state)

    model, history = model_fitting(X_train, y_train, X_valid, y_valid,
                                   autoencoder=autoencoder,
                                   filename=f"dnn_model_fold_{k}",
                                   random_state=config.random_state)
```

```
    val_preds = model.predict(X_valid, batch_size=128)
    best_score = eval_gini(y_true=y_valid,
                           y_pred=np.ravel(val_preds))
    best_epoch = np.argmax(history.history['val_auc']) + 1
    print(f"[best epoch is {best_epoch}]\tvalidation_0-gini_dnn: {best_score:0.5f}\n")

    metric_evaluations.append(best_score)
    preds += (model.predict(test, batch_size=128).ravel() /
              skf.n_splits)
    oof[valid_idx] = model.predict(X_valid, batch_size=128).ravel()
    gpu_cleanup([autoencoder, model])
```

As done with the LighGBM model, we can get an idea of the results by looking at the average fold normalized Gini coefficient.

```
print(f"DNN CV normalized Gini coefficient: {np.mean(metric_evaluations):0.3f} ({np.std(metri
```

The results won't be quite in line with what previously obtained using the LightGBM.

```
DNN CV Gini Normalized Score: 0.276 (0.015)
```

Producing the submission and submitting it will result in a public score of about 0.27737 and a private score of about 0.28471 (results may vary wildly as we previously mentioned), not quite a high score.

```
submission['target'] = preds
submission.to_csv('dnn_submission.csv')
oofs = pd.DataFrame({'id':train_index, 'target':oof})
oofs.to_csv('dnn_oof.csv', index=False)
```

The scarce results from the neural network seem to go by the adage that neural networks underperform in tabular problems. As Kagglers, anyway, we know that all models are useful for a successful placing on the leaderboard, we just need to figure out how to best use them. Surely, a neural network feed with an auto-encoder has worked out a solution less affected by noise in data and elaborated the information in a different way than a GBM.

Ensembling the results

Now, having two models, what's left is to mix them together and see if we can improve the results. As suggested by Jahrer we go straight for a blend of them, but we do not limit ourselves to producing just an average of the two (since our approach in the end has slightly differed from Jahrer's one) but we will also try to get optimal weights for the blend. We start importing the out-of-fold predictions and having our evaluation function ready.

```
import pandas as pd
import numpy as np
from numba import jit
@jit
def eval_gini(y_true, y_pred):
    y_true = np.asarray(y_true)
    y_true = y_true[np.argsort(y_pred)]
    ntrue = 0
    gini = 0
    delta = 0
    n = len(y_true)
    for i in range(n-1, -1, -1):
        y_i = y_true[i]
        ntrue += y_i
        gini += y_i * delta
        delta += 1 - y_i
    gini = 1 - 2 * gini / (ntrue * (n - ntrue))
    return gini
lgb_oof = pd.read_csv("../input/workbook-lgb/lgb_oof.csv")
dnn_oof = pd.read_csv("../input/workbook-dae/dnn_oof.csv")
target = pd.read_csv("../input/porto-seguro-safe-driver-prediction/train.csv", usecols=['id',
```

Once done, we turn the out-of-fold predictions of the LightGBM and the neural network into ranks since the normalized Gini coefficient is sensible to rankings (as would be a ROC-AUC evaluation).

```
lgb_oof_ranks = (lgb_oof.target.rank() / len(lgb_oof))
dnn_oof_ranks = (dnn_oof.target.rank() / len(dnn_oof))
```

Now we just test if, by combining the two models using different weights we can get a better evaluation on the out-of-fold data.

```
baseline = eval_gini(y_true=target.target, y_pred=lgb_oof_ranks)
print(f"starting from a oof lgb baseline {baseline:0.5f}\n")
best_alpha = 1.0
for alpha in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]:
    ensemble = alpha * lgb_oof_ranks + (1.0 - alpha) * dnn_oof_ranks
    score = eval_gini(y_true=target.target, y_pred=ensemble)
    print(f"lgd={alpha:0.1f} dnn={(1.0 - alpha):0.1f} -> {score:0.5f}")

    if score > baseline:
        baseline = score
        best_alpha = alpha

print(f"\nBest alpha is {best_alpha:0.1f}")
```

When ready, by running the snippet we can get interesting results:

```
starting from a oof lgb baseline 0.28850
lgd=0.1 dnn=0.9 -> 0.27352
lgd=0.2 dnn=0.8 -> 0.27744
lgd=0.3 dnn=0.7 -> 0.28084
lgd=0.4 dnn=0.6 -> 0.28368
lgd=0.5 dnn=0.5 -> 0.28595
lgd=0.6 dnn=0.4 -> 0.28763
lgd=0.7 dnn=0.3 -> 0.28873
lgd=0.8 dnn=0.2 -> 0.28923
lgd=0.9 dnn=0.1 -> 0.28916
Best alpha is 0.8
```

It seems that blending using a strong weight (0.8) on the LightGBM model and a weaker one (0.2) on the neural network may lead to an over performing model. We immediately try this hypothesis by setting a blend with the same weights for the models and the ideal weights that we have found out.

```
lgb_submission = pd.read_csv("../input/workbook-lgb/lgb_submission.csv")
dnn_submission = pd.read_csv("../input/workbook-dae/dnn_submission.csv")
submission = pd.read_csv(
"../input/porto-seguro-safe-driver-prediction/sample_submission.csv")
```

First we try the equal weights solution:

```
lgb_ranks = (lgb_submission.target.rank() / len(lgb_submission))
dnn_ranks = (dnn_submission.target.rank() / len(dnn_submission))
submission.target = lgb_ranks * 0.5 + dnn_ranks * 0.5
submission.to_csv("equal_blend_rank.csv", index=False)
```

It leads to a public score of 0.28393 and a private score of 0.29093, which is around 50th position in the final leaderboard, a bit far from our expectations. Now let's try using the weights the out-of-fold predictions helped us to find:

```
lgb_ranks = (lgb_submission.target.rank() / len(lgb_submission))
dnn_ranks = (dnn_submission.target.rank() / len(dnn_submission))
submission.target = lgb_ranks * best_alpha +  dnn_ranks * (1.0 - best_alpha)
submission.to_csv("blend_rank.csv", index=False)
```

Here the results lead to a public score of 0.28502 and a private score of 0.29192 that turns out to be around the seventh position in the final leaderboard. A much better result indeed, because the LightGBM is a good one but it is probably missing some nuances in the data that can be provided as a

favorable correction by adding some information from the neural network trained on the denoised data.

> As pointed out by CPMP in his solution (https://www.kaggle.com/competitions/porto-seguro-safe-driver-prediction/discussion/44614), depending on how to build your cross-validation, you can experience a "huge variation of Gini scores among folds". For this reason, CPMP suggests to decrease the variance of the estimates by using many different seeds for multiple cross-validations and averaging the results.
>
> > Exercise: as an exercise, try to modify the code we used in order to create more stable predictions, especially for the denoising auto-encoder.

Summary

In this first chapter, you have dealt with a classical tabular competition. By reading the notebooks and discussions of the competition, we have come up with a simple solution involving just two models to be easily blended. In particular, we have offered an example on how to use a denoising auto-encoder in order to produce an alternative data processing particularly useful when operating with neural networks for tabular data. By understanding and replicating solutions on past competitions, you can quickly build up your core competencies on Kaggle competitions and be quickly able to perform consistently and highly in more recent competitions and challenges.

In the next chapter, we will explore another tabular competition from Kaggle, this time revolving about a complex prediction problem with time series.

2 The Makridakis Competitions: M5 on Kaggle for Accuracy and Uncertainty

Since 1982, Spyros Makridakis (https://mofc.unic.ac.cy/dr-spyros-makridakis/https://mofc.unic.ac.cy/dr-spyros-makridakis/) has involved groups of researchers from all over the world in forecasting challenges, called M competitions, in order to conduct comparisons of the efficacy of existing and new forecasting methods against different forecasting problems. For this reason, M competitions have always been completely open to both academics and practitioners. The competitions are probably the most cited and referenced event in the forecasting community and they have always highlighted the changing state of the art in forecasting methods. Each previous M competition has provided both researchers and practitioners not only with useful data to train and test their forecasting tools, but also with a series of discoveries and approaches that are revolutionizing the way forecasting are done.

The recent M5 competition (the M6 is just running as this chapter is being written) has been held on Kaggle and it has proved particularly significant in remarking the usefulness of gradient boosting methods when trying to solve a host of volume forecasts of retail products. In this chapter, focusing on the accuracy track, we deal with a time series problem from Kaggle competitions, and by replicating one of the top, yet simplest and most clear solutions, we intend to provide our readers with code and ideas to successfully handle any future forecasting competition that may appear on Kaggle.

- Apart from the competition pages, we found a lot of information regarding the competition and its dynamics in the following papers from the International Journal of Forecasting:
- Makridakis, Spyros, Evangelos Spiliotis, and Vassilios Assimakopoulos. *The M5 competition: Background, organization, and implementation*. International Journal of Forecasting (2021).
- Makridakis, Spyros, Evangelos Spiliotis, and Vassilios Assimakopoulos. "M5 accuracy competition: Results, findings, and conclusions." International Journal of Forecasting (2022).
- Makridakis, Spyros, et al. "The M5 Uncertainty competition: Results, findings and conclusions." International Journal of Forecasting (2021).

Understanding the competition and the data

The competition ran from March to June 2020 and over 7,000 participants took part in it on Kaggle. The organizers arranged it into two separated tracks, one for point-wise prediction (accuracy track) and another one for estimating reliable values at different confidence intervals (uncertainty track)

Walmart provided the data. It consisted of 42,840 daily sales time series of items hierarchically arranged into departments, categories, and stores spread in three U.S. states (the series are somewhat correlated each other). Along with the sales, Walmart also provided accompanying information (exogenous variables, usually not often provided in forecasting problems) such as the prices of items, some calendar information, associated promotions or presence of other events affecting the sales.

Apart from Kaggle, the data is available, together with the datasets from previous M competition, at the address https://forecasters.org/resources/time-series-data/.

One interesting aspect of the competitions is that it dealt with consumer goods sales both fast moving and slow moving with many examples of the latest presenting intermittent sales (sales are often zero but for some rare cases). Intermittent series, though common in many industries, are still a challenging case in forecasting for many practitioners.

The competition timeline has been arranged in two parts. In the first, from the beginning of March 2020 to June 1st, competitors could train models on the range of days up to day 1,913 and score their submission on the public test set (ranging from day 1,914 to 1,941). After that date, until the end of the competition on July 1st, the public test set was made available as part of the training set, allowing participants to tune their models in order to predict from day 1,942 to 1969 (a time windows of 28 days, i.e. four weeks). In that period, submissions were not scored on the leaderboard.

The ratio behind such an arrangement of the competition was to allow teams initially to test their models on the leaderboard and to have grounds to share their best performing methods in notebooks and discussions. After the first phase, the organizers wanted to avoid having the leaderboard used for overfitting purposes or hyperparameter tuning of the models and they wanted to resemble a forecasting situation, as it would happen in the real world. In addition, the requirement to choose only one submission as the final one mirrored the same necessity for realism (in the real world you cannot use two distinct models predictions and choose the one that suits you the best afterwards).

As for as the data, we mentioned that the data has been provided by Walmart and it represents the USA market: it originated from 10 stores in California, Wisconsin and Texas. Specifically, the data it is made up by the sales of 3,049 products, organized into three categories (hobbies, food, and household) that can be divided furthermore into 7 departments each. Such hierarchical structure is certainly a challenge because you can model sale dynamics at the level of USA market, state market, single store, product category, category department and finally specific product. All these levels can also combine as different aggregates, which are something required to be predicted in the second track, the uncertainty track:

Level id	Level description	Aggregation level	Number of series
1	All products, aggregated for all stores and states	Total	1
2	All products, aggregated for each state	State	3
3	All products, aggregated for each store	Store	10
4	All products, aggregated for each category	Category	3
5	All products, aggregated for each department	Department	7
6	All products, aggregated for each state and category	State-Category	9
7	All products, aggregated for each state and department	State-Department	21
8	All products, aggregated for each store and category	Store-Category	30
9	All products, aggregated for each store and department	Store-Department	70
10	Each product, aggregated for all stores/states	Product	3,049
11	Each product, aggregated for each state	Product-State	9,147
12	Each product, aggregated for each store	Product-Store	30,490
		Total	42,840

From the point of view of time, the granularity is daily sales record and the covered the period spanning from 29 January 2011 to 19 June 2016 which equals to 1,969 days in total, 1,913 for training,

28 for validation – public leaderboard – 28 for test – private leaderboard. A forecasting horizon of 28 days is actually recognized in the retail sector as the proper horizon for handling stocks and re-ordering operations for most goods.

Let's examine the different data you receive for the competition. You get `sales_train_evaluation.csv`, `sell_prices.csv` and `calendar.csv`. The one keeping the time series is `sales_train_evaluation.csv`. It is composed of fields that act as identifiers (`item_id`, `dept_id`, `cat_id`, `store_id`, and `state_id`) and columns from `d_1` to `d_1941` representing the sales of those days:

	id	item_id	dept_id	cat_id	store_id	state_id	d_1	d_2	d_3	d_4	...	d_1932	d_1933	d_1934	d_1935
0	HOBBIES_1_001_CA_1_evaluation	HOBBIES_1_001	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	2	4	0	0
1	HOBBIES_1_002_CA_1_evaluation	HOBBIES_1_002	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	0	1	2	1
2	HOBBIES_1_003_CA_1_evaluation	HOBBIES_1_003	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	1	0	2	0
3	HOBBIES_1_004_CA_1_evaluation	HOBBIES_1_004	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	1	1	0	4
4	HOBBIES_1_005_CA_1_evaluation	HOBBIES_1_005	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	0	0	0	2

Figure 2.1: The sales_train_evaluation.csv data

`sell_prices.csv` contains instead information about the price of the items. The difficulty here is in joining the `wm_yr_wk` (the id of the week) with the columns in the training data:

	store_id	item_id	wm_yr_wk	sell_price
0	CA_1	HOBBIES_1_001	11325	9.58
1	CA_1	HOBBIES_1_001	11326	9.58
2	CA_1	HOBBIES_1_001	11327	8.26
3	CA_1	HOBBIES_1_001	11328	8.26
4	CA_1	HOBBIES_1_001	11329	8.26

Figure 2.2: The sell_prices.csv data

The last file, `calendar.csv`, contains data relative to events that could have affected the sales:

	date	wm_yr_wk	weekday	wday	month	year	d	event_name_1	event_type_1	event_name_2	event_type_2	snap_CA	snap_TX	snap_WI
0	2011-01-29	11101	Saturday	1	1	2011	d_1	NaN	NaN	NaN	NaN	0	0	0
1	2011-01-30	11101	Sunday	2	1	2011	d_2	NaN	NaN	NaN	NaN	0	0	0
2	2011-01-31	11101	Monday	3	1	2011	d_3	NaN	NaN	NaN	NaN	0	0	0
3	2011-02-01	11101	Tuesday	4	2	2011	d_4	NaN	NaN	NaN	NaN	1	1	0
4	2011-02-02	11101	Wednesday	5	2	2011	d_5	NaN	NaN	NaN	NaN	1	0	1

Figure 2.3: The calendar.csv data

Again, the main difficulty seems in joining the data to the columns in the training table. Anyway, here you can get an easy key to connect columns (the d field) with the `wm_yr_wk`. In addition, in the table we have represented different events that may have occurred on particular days as well as SNAP days that are special days when the nutrition assistance benefits called the Supplement Nutrition Assistance Program (SNAP) can be used.

Understanding the Evaluation Metric

The accuracy competition introduced a new evaluation metric: Weighted Root Mean Squared Scaled Error (WRMSSE). The metric evaluates the deviation of the of the point forecasts around the mean of the realized values of the series being predicted:

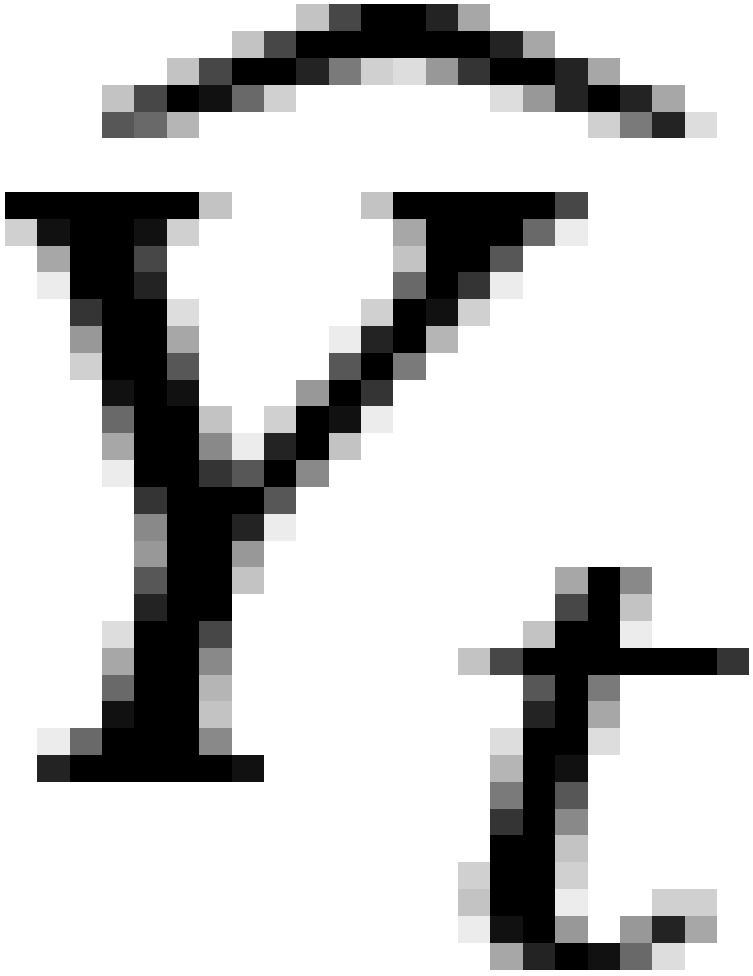
$$\boldsymbol{RMSSE} = \sqrt{\frac{1}{h}\frac{\sum_{t=n+1}^{n+h}\left(Y_t - \widehat{Y}_t\right)^2}{\frac{1}{n-1}\sum_{t=2}^{n}\left(Y_t - Y_{t-1}\right)^2}},$$

where:

n is the length of the training sample

h is the forecasting horizon (in our case it is h=28)

Y_t is the sales value at time t,



is the predicted value at time *t*

In the competition guidelines (https://mofc.unic.ac.cy/m5-competition/), in regard of WRMSSE, it is remarked that:

- The denominator of RMSSE is computed only for the time-periods for which the examined product(s) are actively sold, i.e., the periods following the first non-zero demand observed for the series under evaluation
- The measure is scale independent, meaning that it can be effectively used to compare forecasts across series with different scales.
- In contrast to other measures, it can be safely computed as it does not rely on divisions with values that could be equal or close to zero (e.g. as done in percentage errors when $Y_t = 0$ or relative errors when the error of the benchmark used for scaling is zero).
- The measure penalizes positive and negative forecast errors, as well as large and small forecasts, equally, thus being symmetric.

A good explanation of the underlying workings of this is provided by this post from Alexander Soare (https://www.kaggle.com/alexandersoare): https://www.kaggle.com/competitions/m5-forecasting-accuracy/discussion/148273. After having transformed the evaluation metric, Alexandre attributes improving performances to improving the ratio between the error in the predictions and the day-to-day variation of sales values. If the error is the same as the daily variations (ratio=1), it is likely that the model is not much better than a random guess based on historical variations. If your error is less than that, it is score in a quadratic way (because of the square root) as it approaches zero. Consequently, a WRMSSE of 0.5 corresponds to a ratio of 0.7 and a WRMSSE of 0.25 corresponds to a ratio of 0.5.

During the competition, many attempts have been made at using the metric not only for evaluation besides the leaderboard but also as an objective function. First of all, the Tweedie loss (implemented both in XGBoost and LightGBM) worked quite well for the problem because it could handle the skewed distributions of sales for most products (a lot of them also had intermittent sales and that is also handled finely by the Tweedie loss). The Poisson and Gamma distributions can be considered extreme cases of the Tweedie distribution: based on the parameter power, p, with p=1 you get a Poisson distribution and with p=2 a Gamma one. Such power parameter is actually the glue that keeps the mean and the variance of the distribution connected by the formula variance = k*mean**p. Using a power value between 1 and 2, you actually get a mix of Poisson and Gamma distributions which can fit very well the competition problem. Most of the Kagglers involved in the competition and using a GBM solution, actually have resorted to Tweedie loss.

In spite of Tweedie success, some other Kagglers, however, found interesting ways to implement an objective loss more similar to WRMSSE for their models:

* Martin Kovacevic Buvinic with his asymmetric loss:
https://www.kaggle.com/code/ragnar123/simple-lgbm-groupkfold-cv/notebook

* Timetraveller using PyTorch Autograd to get gradient and hessian for any differentiable continuous loss function to be implemented in LighGBM: https://www.kaggle.com/competitions/m5-forecasting-accuracy/discussion/152837

Examining the 4th place solution's ideas from Monsaraida

There are many solutions available for the competition, to be mostly found on the competition Kaggle discussions pages. The top five methods of both challenges have also been gathered and published (but one because of proprietary rights) by the competition organizers themselves: https://github.com/Mcompetitions/M5-methods (by the way, reproducing the results of the winning submissions was a prerequisite for the collection of a competition prize).

Noticeably, all the Kagglers that have placed in the higher ranks of the competitions have used, as their unique model type or in blended/stacked in ensembles, LightGBM because of its lesser memory usage and speed of computations that gave it an advantage in the competition because of the large amount of times series to process and predict. But there are also other reasons for its success. Contrary to classical methods based on ARIMA, it doesn't require relying on the analysis of auto-correlation and in specifically figuring out the parameters for each single series in the problem. In addition, contrary to methods based on deep learning, it doesn't require looking for improving complicated neural architectures or tuning large number of hyperparameters. The strength of the gradient boosting methods in time series problems (for extension of every other gradient boosting algorithm, such as for instance XGBoost) is to rely on feature engineering, creating the right number of features based on time lags, moving averages and averages from groupings of the series attributes. Then choosing the right objective function and doing some hyper-parameter tuning will suffice to obtain excellent results when the time series are enough long (for shorter series, classical statistical methods such as ARIMA or exponential smoothing are still the recommended choice).

Another advantage of LightGBM and XGBoost against deep learning solutions in the competition was the Tweedie loss, not requiring any feature scaling (deep learning networks are particularly sensible to the scaling you use) and the speed of training that allowed faster iterations while testing feature engineering.

Among all these available solutions, we found the one proposed by Monsaraida (Masanori Miyahara), a Japanese computer scientist, the most interesting one. He has proposed a simple and straightforward solution that has ranked four on the private leaderboard with a score of 0.53583. The solution uses just general features without prior selection (such as sales statistics, calendar, prices, and identifiers). Moreover, it uses a limited number of models of the same kind, using LightGBM gradient boosting, without resorting to any kind of blending, recursive modelling when predictions feed other hierarchically related predictions or multipliers that is choosing constants to fit the test set better. Here is a scheme taken from his presentation solution presentation to the M (https://github.com/Mcompetitions/M5-methods/tree/master/Code%20of%20Winning%20Methods/A4) where it can be noted that he treats each of the ten stores by each of the four weeks to be looked into the future that in the end corresponds to producing 40 models:

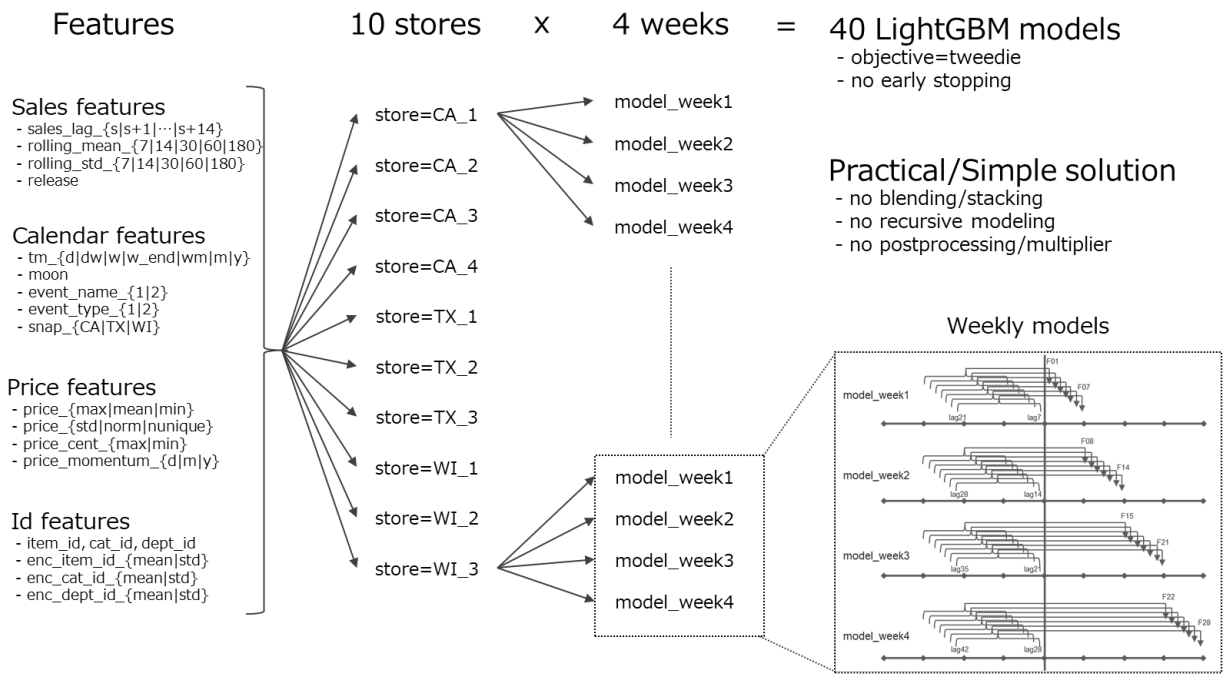


Figure 2.4: explanation by Monsaraida about the structure of his solution

Given that Monsaraida has kept his solution simple and practical, like in a real-world forecasting project, in this chapter, we will try to replicate his example by refactoring his code in order to run in Kaggle notebooks (we will handle the memory and the running time limitations by splitting the code into multiple notebooks). In this way, we intend to provide the readers with a simple and effective way, based on gradient boosting, to approach forecasting problems.

Computing predictions for specific dates and time horizons

The plan for replicating Monsaraida's solution is to create a notebook customizable by input parameters in order to produce the necessary processed data for train and test and the LightGBM models for predictions. The models, given data in the past, will be trained to learn to predict values in a

specific number of days in the future. The best results can be obtained by having each model to learn to predict the values in a specific week range in the future. Since we have to predict up to 28 days ahead in the future, we need a model predicting from day +1 to day +7 in the future, then another one able to predict from day +8 to day +14, another from day +15 to +21 and finally another last one capable of handling predictions from day +22 to day +28. We will need a Kaggle notebook for each of these time ranges, thus we need four notebooks. Each of these notebooks will be trained to predict that future time span for each of the ten stores part of the competitions. In total, each notebook will produce ten models. All together, the notebooks will then produce forty models covering all the future range and all the stores.

Since we need to predict both for the public leaderboard and for the private one, it is necessary to repeat this process twice, stopping training at day 1,913 (predicting days from 1,914 to 1,941) for the public test set submission and at day 1,941 (predicting days from 1,942 50 1,969) for the private one.

Given the current limitations for running Kaggle notebooks based on CPU, all these eight notebooks can be run in parallel (all the process taking almost 6 hours and a half). Each notebook can be distinguishable by others by its name, containing the parameters' values relative to the last training day and the look-ahead horizon in days. An example of one of these notebooks can be found at: https://www.kaggle.com/code/lucamassaron/m5-train-day-1941-horizon-7.

Let's now examine together how the code has been arranged and what we can learn from Monsaraida's solution.

We simply start by importing the necessary packages. You can just notice how, apart from NumPy and pandas, the only data science specialized package is LightGBM. You may also notice that we are going to use gc (garbage collection): that's because we need to limit the amount of used memory by the script, and we frequently just collect and recycle the unused memory. As part of this strategy, we also frequently store away on disk models and data structures, instead of keeping them in-memory:

```
import numpy as np
import pandas as pd
import os
import random
import math
from decimal import Decimal as dec
import datetime
import time
import gc
import lightgbm as lgb
import pickle
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
```

As part of the strategy to limit the memory usage, we resort to the function to reduce pandas DataFrame described in the Kaggle book and initially developed by Arjan Groen during the Zillow competition (read the discussion https://www.kaggle.com/competitions/tabular-playground-series-dec-2021/discussion/291844):

```
def reduce_mem_usage(df, verbose=True):
    numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
    start_mem = df.memory_usage().sum() / 1024**2
    for col in df.columns:
        col_type = df[col].dtypes
        if col_type in numerics:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                    df[col] = df[col].astype(np.int16)
```

```
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)
    end_mem = df.memory_usage().sum() / 1024**2
    if verbose: print('Mem. usage decreased to {:5.2f} Mb ({:.1f}% reduction)'.format(end_mem
    return df
```

We keep on defining functions for this solution, because it helps splitting the solution into smaller parts and because it is easier to clean up all the used variables when you just return from a function (you keep only what you saved to disk, and you returned from the function). Our next function helps us to load all the data available and compress it:

```
def load_data():
    train_df = reduce_mem_usage(pd.read_csv("../input/m5-forecasting-accuracy/sales_train_eva
    prices_df = reduce_mem_usage(pd.read_csv("../input/m5-forecasting-accuracy/sell_prices.cs
    calendar_df = reduce_mem_usage(pd.read_csv("../input/m5-forecasting-accuracy/calendar.csv
    submission_df = reduce_mem_usage(pd.read_csv("../input/m5-forecasting-accuracy/sample_sub
    return train_df, prices_df, calendar_df, submission_df
train_df, prices_df, calendar_df, submission_df = load_data()
```

After preparing the code to retrieve the data relative to prices, volumes, and calendar information, we proceed to prepare the first processing function that will have the role to create a basic table of information having `item_id`, `dept_id`, `cat_id`, `state_id` and `store_id` as row keys, a day column and values column containing the volumes. This is achieved starting from rows having all the days' data columns by using the pandas command melt (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html). The command takes as reference the index of the DataFrame and then picks all the remaining features, placing their name on a column and their value on another one (`var_name` and `value_name` parameters help you define the name of these new columns). In this way, you can unfold a row representing the sales series of a certain item in a certain store into multiple rows each one representing a single day. The fact that the positional order of the unfolded columns is preserved guarantees that now your time series spans on the vertical axis (you can therefore apply furthermore transformations on it, such as moving means).

To give you an idea of what is happening, here is the `train_df` before the transformation with `pd.melt`. Notice how the volumes of the distinct days are column features:

```
train_df.head()
```

	id	item_id	dept_id	cat_id	store_id	state_id	d_1	d_2	d_3	d_4	...	d_1932	d_1933	d_1934	d_1935
0	HOBBIES_1_001_CA_1_evaluation	HOBBIES_1_001	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	2	4	0	0
1	HOBBIES_1_002_CA_1_evaluation	HOBBIES_1_002	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	0	1	2	1
2	HOBBIES_1_003_CA_1_evaluation	HOBBIES_1_003	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	1	0	2	0
3	HOBBIES_1_004_CA_1_evaluation	HOBBIES_1_004	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	1	1	0	4
4	HOBBIES_1_005_CA_1_evaluation	HOBBIES_1_005	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...	0	0	0	2

5 rows × 1947 columns

Figure 2.5: The training DataFrame

After the transformation, you obtain a `grid_df` where the days have been distributed on separated days:

```
grid_df.head()
```

	id	item_id	dept_id	cat_id	store_id	state_id	d	sales
0	HOBBIES_1_001_CA_1_evaluation	HOBBIES_1_001	HOBBIES_1	HOBBIES	CA_1	CA	d_1	0
1	HOBBIES_1_002_CA_1_evaluation	HOBBIES_1_002	HOBBIES_1	HOBBIES	CA_1	CA	d_1	0
2	HOBBIES_1_003_CA_1_evaluation	HOBBIES_1_003	HOBBIES_1	HOBBIES	CA_1	CA	d_1	0
3	HOBBIES_1_004_CA_1_evaluation	HOBBIES_1_004	HOBBIES_1	HOBBIES	CA_1	CA	d_1	0
4	HOBBIES_1_005_CA_1_evaluation	HOBBIES_1_005	HOBBIES_1	HOBBIES	CA_1	CA	d_1	0

Figure 2.5: Applying pd.melt to the training DataFrame

The feature d contains the reference to the columns that are not part of the index, in essence, all the features from `d_1` to `d_1935`. By simply removing the 'd_' prefix from its values and converting them to integers, you now have a day feature.

Apart from this, the code snippet also separates a holdout of the rows (your validation set) based on the time from the training ones. On the training part it will also add the rows necessary for your predictions based on the predict horizon you provide.

Here is the function that creates our basic feature template. As input, it takes the `train_df` DataFrame, it expects the day the train ends and the predict horizon (the number of days you want to predict in the future):

```
def generate_base_grid(train_df, end_train_day_x, predict_horizon):
    index_columns = ['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id']
    grid_df = pd.melt(train_df, id_vars=index_columns, var_name='d', value_name='sales')
    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df['d_org'] = grid_df['d']
    grid_df['d'] = grid_df['d'].apply(lambda x: x[2:]).astype(np.int16)
    time_mask = (grid_df['d'] > end_train_day_x) &  (grid_df['d'] <= end_train_day_x + predic
    holdout_df = grid_df.loc[time_mask, ["id", "d", "sales"]].reset_index(drop=True)
    holdout_df.to_feather(f"holdout_df_{end_train_day_x}_to_{end_train_day_x + predict_horizc
    del(holdout_df)
    gc.collect()
    grid_df = grid_df[grid_df['d'] <= end_train_day_x]
    grid_df['d'] = grid_df['d_org']
    grid_df = grid_df.drop('d_org', axis=1)
    add_grid = pd.DataFrame()
    for i in range(predict_horizon):
        temp_df = train_df[index_columns]
        temp_df = temp_df.drop_duplicates()
        temp_df['d'] = 'd_' + str(end_train_day_x + i + 1)
        temp_df['sales'] = np.nan
        add_grid = pd.concat([add_grid, temp_df])

    grid_df = pd.concat([grid_df, add_grid])
    grid_df = grid_df.reset_index(drop=True)

    for col in index_columns:
        grid_df[col] = grid_df[col].astype('category')

    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df.to_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horizon}.fea
    del(grid_df)
    gc.collect()
```

After handling the function to create the basic feature template, we prepare a merge function for pandas DataFrames that will help to save memory space and avoid memory errors when handling large sets of data. Given two DataFrame, df1 and df2 and the set of foreign keys we need them to be merged,

the function applies a left outer join between df1 and df2 without creating a new merged object but simply expanding the existent df1 DataFrame.

The function works first by extracting the foreign keys from df1, then merging the extracted keys with df2. In this way, the function creates a new DataFrame, called `merged_gf`, which is ordered as df1. At this point, we just assign the `merged_gf` columsn to df1. Internally, df1 will pick the reference to the internal data structures from `merged_gf`. Such an approach helps minimizing the memory usage because only the necessary used data is created at any time (there are no duplicates that can fill-up the memory). When the function returns df1, `merged_gf` is cancelled but for the data now used by df1.

Here is the code for this utility function:

```
def merge_by_concat(df1, df2, merge_on):
    merged_gf = df1[merge_on]
    merged_gf = merged_gf.merge(df2, on=merge_on, how='left')
    new_columns = [col for col in list(merged_gf)
                   if col not in merge_on]
    df1[new_columns] = merged_gf[new_columns]
    return df1
```

After this necessary step, we proceed to program a new function to process the data. This time we handle the prices data, a set of data containing the prices of each item by each store for all the weeks. Since it is important to figure out if we are talking about a new product appearing in a store or not, the function picks the first date of price availability (using the `wm_yr_wk` feature in the price table, representing the id of the week) and it copies it to our feature template.

Here is the code for processing the release dates:

```
def calc_release_week(prices_df, end_train_day_x, predict_horizon):
    index_columns = ['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id']

    grid_df = pd.read_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horiz

    release_df = prices_df.groupby(['store_id', 'item_id'])['wm_yr_wk'].agg(['min']).reset_ir
    release_df.columns = ['store_id', 'item_id', 'release']

    grid_df = merge_by_concat(grid_df, release_df, ['store_id', 'item_id'])

    del release_df
    grid_df = reduce_mem_usage(grid_df, verbose=False)
    gc.collect()

    grid_df = merge_by_concat(grid_df, calendar_df[['wm_yr_wk', 'd']], ['d'])
    grid_df = grid_df.reset_index(drop=True)
    grid_df['release'] = grid_df['release'] - grid_df['release'].min()
    grid_df['release'] = grid_df['release'].astype(np.int16)

    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df.to_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horizon}.fea
    del(grid_df)
    gc.collect()
```

After having handles the day of the product appearance in a store, we definitely proceed to deal with the prices. In regard of each item, by each shop, we prepare basic price features telling:

- the actual price (normalized by the maximum)
- the maximum price
- the minimum price
- the mean price
- the standard deviation of the price
- the number of different prices the item has taken
- the number of items in the store with the same price

Besides these basic descriptive statistics of prices, we also add some features to describe their dynamics for each item in a store based on different time granularities:

- the day momentum, i.e. the ratio before the actual price and its price the previous day
- the month momentum, i.e. the ratio before the actual price and its average price the same month
- the year momentum, i.e. the ratio before the actual price and its average price the same year

Here we use two interesting and essential pandas methods for time series feature processing:

* shift : that can move the index forward or backward by n steps (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.shift.html)

* transform: that applied to a group by, fills a like-index feature with the transformed values (https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.transform.html)

In addition, the decimal part of the price is processed as a feature, in order to reveal a situation when the item is sold at psychological pricing thresholds (e.g. $19.99 or £2.98 – see this discussion: https://www.kaggle.com/competitions/m5-forecasting-accuracy/discussion/145011). The function `math.modf` (https://docs.python.org/3.8/library/math.html#math.modf) helps in doing so because it splits any floating-point number into the fractional and integer parts (a two-item tuple).

Finally, the resulting table is saved onto disk.

Here is the function doing all the feature engineering on prices:

```
def generate_grid_price(prices_df, calendar_df, end_train_day_x, predict_horizon):
    grid_df = pd.read_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horiz
    prices_df['price_max'] = prices_df.groupby(['store_id', 'item_id'])['sell_price'].transfo
    prices_df['price_min'] = prices_df.groupby(['store_id', 'item_id'])['sell_price'].transfo
    prices_df['price_std'] = prices_df.groupby(['store_id', 'item_id'])['sell_price'].transfo
    prices_df['price_mean'] = prices_df.groupby(['store_id', 'item_id'])['sell_price'].transf
    prices_df['price_norm'] = prices_df['sell_price'] / prices_df['price_max']
    prices_df['price_nunique'] = prices_df.groupby(['store_id', 'item_id'])['sell_price'].tra
    prices_df['item_nunique'] = prices_df.groupby(['store_id', 'sell_price'])['item_id'].trar
    calendar_prices = calendar_df[['wm_yr_wk', 'month', 'year']]
    calendar_prices = calendar_prices.drop_duplicates(subset=['wm_yr_wk'])
    prices_df = prices_df.merge(calendar_prices[['wm_yr_wk', 'month', 'year']], on=['wm_yr_wk

    del calendar_prices
    gc.collect()

    prices_df['price_momentum'] = prices_df['sell_price'] / prices_df.groupby(['store_id', 'i
        'sell_price'].transform(lambda x: x.shift(1))
    prices_df['price_momentum_m'] = prices_df['sell_price'] / prices_df.groupby(['store_id',
        'sell_price'].transform('mean')
    prices_df['price_momentum_y'] = prices_df['sell_price'] / prices_df.groupby(['store_id',
        'sell_price'].transform('mean')
    prices_df['sell_price_cent'] = [math.modf(p)[0] for p in prices_df['sell_price']]
    prices_df['price_max_cent'] = [math.modf(p)[0] for p in prices_df['price_max']]
    prices_df['price_min_cent'] = [math.modf(p)[0] for p in prices_df['price_min']]
    del prices_df['month'], prices_df['year']
    prices_df = reduce_mem_usage(prices_df, verbose=False)
    gc.collect()

    original_columns = list(grid_df)
    grid_df = grid_df.merge(prices_df, on=['store_id', 'item_id', 'wm_yr_wk'], how='left')
    del(prices_df)
    gc.collect()

    keep_columns = [col for col in list(grid_df) if col not in original_columns]
    grid_df = grid_df[['id', 'd'] + keep_columns]
    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df.to_feather(f"grid_price_{end_train_day_x}_to_{end_train_day_x + predict_horizon}.
```

```
    del(grid_df)
    gc.collect()
```

The next function computes the moon phase, giving back one of its eight phases (from new moon to waning crescent). Although moon phases shouldn't directly influence any sales (weather conditions instead do, but we have no weather information in the data), they represent a periodic cycle of 29 and a half days which can well suit periodic shopping behaviors. There is an interesting discussion, with different hypothesis regarding why moon phases may work as a predictor, in this competition post: https://www.kaggle.com/competitions/m5-forecasting-accuracy/discussion/154776:

```
def get_moon_phase(d):  # 0=new, 4=full; 4 days/phase
    diff = datetime.datetime.strptime(d, '%Y-%m-%d') - datetime.datetime(2001, 1, 1)
    days = dec(diff.days) + (dec(diff.seconds) / dec(86400))
    lunations = dec("0.20439731") + (days * dec("0.03386319269"))
    phase_index = math.floor((lunations % dec(1) * dec(8)) + dec('0.5'))
    return int(phase_index) & 7
```

The moon phase function is part of a general function for creating time-based features. The function takes the calendar dataset information and places it among the features. Such information contains events and their type as well as indication of the SNAP periods (a nutrition assistance benefit called the Supplement Nutrition Assistance Program – hence SNAP – to help lower-income families) that could drive furthermore sales of basic goods. The function also generates numeric features such as the day, the month, the year, the day of the week, the week in the month, if it is an end of week. Here is the code:

```
def generate_grid_calendar(calendar_df, end_train_day_x, predict_horizon):

    grid_df = pd.read_feather(
                f"grid_df_{end_train_day_x}_to_{end_train_day_x +
                predict_horizon}.feather")
    grid_df = grid_df[['id', 'd']]
    gc.collect()
    calendar_df['moon'] = calendar_df.date.apply(get_moon_phase)
    # Merge calendar partly
    icols = ['date',
             'd',
             'event_name_1',
             'event_type_1',
             'event_name_2',
             'event_type_2',
             'snap_CA',
             'snap_TX',
             'snap_WI',
             'moon',
             ]
    grid_df = grid_df.merge(calendar_df[icols], on=['d'], how='left')
    icols = ['event_name_1',
             'event_type_1',
             'event_name_2',
             'event_type_2',
             'snap_CA',
             'snap_TX',
             'snap_WI']

    for col in icols:
        grid_df[col] = grid_df[col].astype('category')
    grid_df['date'] = pd.to_datetime(grid_df['date'])
    grid_df['tm_d'] = grid_df['date'].dt.day.astype(np.int8)
    grid_df['tm_w'] = grid_df['date'].dt.isocalendar().week.astype(np.int8)
    grid_df['tm_m'] = grid_df['date'].dt.month.astype(np.int8)
    grid_df['tm_y'] = grid_df['date'].dt.year
    grid_df['tm_y'] = (grid_df['tm_y'] - grid_df['tm_y'].min()).astype(np.int8)
    grid_df['tm_wm'] = grid_df['tm_d'].apply(lambda x: math.ceil(x / 7)).astype(np.int8)
    grid_df['tm_dw'] = grid_df['date'].dt.dayofweek.astype(np.int8)
    grid_df['tm_w_end'] = (grid_df['tm_dw'] >= 5).astype(np.int8)

    del(grid_df['date'])
```

```
    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df.to_feather(f"grid_calendar_{end_train_day_x}_to_{end_train_day_x + predict_horizo

    del(grid_df)
    del(calendar_df)
    gc.collect()
```

The following function instead just removes the `wm_yr_wk` feature and transforms the d (day) feature into a numeric. It is a necessary step for the following feature transformation functions.

```
def modify_grid_base(end_train_day_x, predict_horizon):
    grid_df = pd.read_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horiz
    grid_df['d'] = grid_df['d'].apply(lambda x: x[2:]).astype(np.int16)
    del grid_df['wm_yr_wk']

    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df.to_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horizon}.fea

    del(grid_df)
    gc.collect()
```

Our last two feature creation functions will generate more sophisticated feature engineering for time series. The first function will produce both lagged sales and their moving averages. Fist, using the shift method (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.shift.html) will generate a range of lagged sales up to 15 days in the past. Then using shift in conjunction with rolling (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rolling.html) will create moving means with windows 7, 14, 30, 60 and 180 days.

The shift command is necessary because it will allow moving the index so that you will always consider available data for your calculations. Hence, if your prediction horizon goes up to seven days, the calculations will consider only the data available seven days before. Then the rolling command will create a moving window observations that can be summarized (in this case by the mean). Having a mean over a period (the moving window) and following its evolutions will help you to detect better any changes in trends because patterns not repeating across the time windows will be leveled off. This a common strategy in time series analysis to remove noise and non-interesting patters. For instance, with a rolling mean of seven days you will cancel all the daily patterns and just represent what happens to your sales on a weekly basis.

> Can you experiment with different moving average windows? Also trying different strategies may help. For instance, by exploring the Tabular Playground of January 2022 (https://www.kaggle.com/competitions/tabular-playground-series-jan-2022) devoted to time series you may find furthermore idea since most solutions are built using Gradient Boosting.

Here is the code to generate the lag and rolling mean features:

```
def generate_lag_feature(end_train_day_x, predict_horizon):
    grid_df = pd.read_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horiz
    grid_df = grid_df[['id', 'd', 'sales']]

    num_lag_day_list = []
    num_lag_day = 15
    for col in range(predict_horizon, predict_horizon + num_lag_day):
        num_lag_day_list.append(col)

    grid_df = grid_df.assign(**{
        '{}_lag_{}'.format(col, l): grid_df.groupby(['id'])['sales'].transform(lambda x: x.sh
        for l in num_lag_day_list
    })
    for col in list(grid_df):
        if 'lag' in col:
            grid_df[col] = grid_df[col].astype(np.float16)
    num_rolling_day_list = [7, 14, 30, 60, 180]
    for num_rolling_day in num_rolling_day_list:
```

```
        grid_df['rolling_mean_' + str(num_rolling_day)] = grid_df.groupby(['id'])['sales'].tr
            lambda x: x.shift(predict_horizon).rolling(num_rolling_day).mean()).astype(np.flo
        grid_df['rolling_std_' + str(num_rolling_day)] = grid_df.groupby(['id'])['sales'].tra
            lambda x: x.shift(predict_horizon).rolling(num_rolling_day).std()).astype(np.floa
    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df.to_feather(f"lag_feature_{end_train_day_x}_to_{end_train_day_x + predict_horizon}

    del(grid_df)
    gc.collect()
```

As for as the second advanced feature engineering function, it is an encoding function, taking specific groupings of variables among state, store, category, department, and sold item and representing their mean and standard deviation. Such embeddings are time independent (time is not part of the grouping) and they have the role to help the training algorithm to distinguish how items, categories, and stores (and their combinations) differentiate among themselves.

> The proposed embeddings are quite easy to compute using target encoding, as described in *The Kaggle Book* on page 216, can you obtain better results and how?

The code works by grouping the features, computing their descriptive statistic (the mean or the standard deviation in our case) and then applying the results to the dataset using the transform (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transform.html) method that we discussed before:

```
def generate_target_encoding_feature(end_train_day_x, predict_horizon):
    grid_df = pd.read_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + predict_horiz

    grid_df.loc[grid_df['d'] > (end_train_day_x - predict_horizon), 'sales'] = np.nan
    base_cols = list(grid_df)
    icols = [
        ['state_id'],
        ['store_id'],
        ['cat_id'],
        ['dept_id'],
        ['state_id', 'cat_id'],
        ['state_id', 'dept_id'],
        ['store_id', 'cat_id'],
        ['store_id', 'dept_id'],
        ['item_id'],
        ['item_id', 'state_id'],
        ['item_id', 'store_id']
    ]
    for col in icols:
        col_name = '_' + '_'.join(col) + '_'
        grid_df['enc' + col_name + 'mean'] = grid_df.groupby(col)['sales'].transform('mean').
            np.float16)
        grid_df['enc' + col_name + 'std'] = grid_df.groupby(col)['sales'].transform('std').as
            np.float16)
    keep_cols = [col for col in list(grid_df) if col not in base_cols]
    grid_df = grid_df[['id', 'd'] + keep_cols]
    grid_df = reduce_mem_usage(grid_df, verbose=False)
    grid_df.to_feather(f"target_encoding_{end_train_day_x}_to_{end_train_day_x + predict_hori

    del(grid_df)
    gc.collect()
```

Having completed the feature engineering part, we now proceed to put together all the files we have stored away on disk while generating the features. The following function just loads the different datasets of basic features, price features, calendar features, lag/rolling and embedded features, and concatenate all together. The code then filters only the rows relative to a specific shop to be saved as a separated dataset. Such an approach matches the strategy of having a model trained on a specific store aimed at predicting for a specific time interval:

```
def assemble_grid_by_store(train_df, end_train_day_x, predict_horizon):
    grid_df = pd.concat([pd.read_feather(f"grid_df_{end_train_day_x}_to_{end_train_day_x + pr
```

```
                    pd.read_feather(f"grid_price_{end_train_day_x}_to_{end_train_day_x + pre
                    pd.read_feather(f"grid_calendar_{end_train_day_x}_to_{end_train_day_x +
                    axis=1)
    gc.collect()
    store_id_set_list = list(train_df['store_id'].unique())
    index_store = dict()
    for store_id in store_id_set_list:
        extract = grid_df[grid_df['store_id'] == store_id]
        index_store[store_id] = extract.index.to_numpy()
        extract = extract.reset_index(drop=True)
        extract.to_feather(f"grid_full_store_{store_id}_{end_train_day_x}_to_{end_train_day_x
    del(grid_df)
    gc.collect()

    mean_features = [
        'enc_cat_id_mean', 'enc_cat_id_std',
        'enc_dept_id_mean', 'enc_dept_id_std',
        'enc_item_id_mean', 'enc_item_id_std'
        ]
    df2 = pd.read_feather(f"target_encoding_{end_train_day_x}_to_{end_train_day_x + predict_h
    for store_id in store_id_set_list:
        df = pd.read_feather(f"grid_full_store_{store_id}_{end_train_day_x}_to_{end_train_day
        df = pd.concat([df, df2[df2.index.isin(index_store[store_id])].reset_index(drop=True)
        df.to_feather(f"grid_full_store_{store_id}_{end_train_day_x}_to_{end_train_day_x + pr
    del(df2)
    gc.collect()

    df3 = pd.read_feather(f"lag_feature_{end_train_day_x}_to_{end_train_day_x + predict_horiz
    for store_id in store_id_set_list:
        df = pd.read_feather(f"grid_full_store_{store_id}_{end_train_day_x}_to_{end_train_day
        df = pd.concat([df, df3[df3.index.isin(index_store[store_id])].reset_index(drop=True)
        df.to_feather(f"grid_full_store_{store_id}_{end_train_day_x}_to_{end_train_day_x + pr
    del(df3)
    del(store_id_set_list)
    gc.collect()
```

The following function, instead, just further processes the selection from the previous one, by removing unused features and reordering the columns and it returns the data for a model to be trained on:

```
def load_grid_by_store(end_train_day_x, predict_horizon, store_id):
    df = pd.read_feather(f"grid_full_store_{store_id}_{end_train_day_x}_to_{end_train_day_x +

    remove_features = ['id', 'state_id', 'store_id', 'date', 'wm_yr_wk', 'd', 'sales']
    enable_features = [col for col in list(df) if col not in remove_features]
    df = df[['id', 'd', 'sales'] + enable_features]
    df = reduce_mem_usage(df, verbose=False)
    gc.collect()

    return df, enable_features
```

Finally, we can now deal with the training phase. The following code snippet starts by defining the training parameters as explicated by Monsaraida being the most effective on the problem. For training time reasons, we just modified the boosting type, choosing using goss instead of gbdt because that can really speed up training without much loss in terms of performance. A good speed-up to the model is also provided by the subsample parameter and the feature fraction: at each learning step of the gradient boosting only half of the examples and half of the features will be considered.

> Also compiling LightGBM on your machine with the right compiling options may increase your speed as explained in this interesting competition discussion: https://www.kaggle.com/competitions/m5-forecasting-accuracy/discussion/148273

The Tweedie loss, with a power value of 1.1 (hence with an underlying distribution quite similar to Poisson) seems particularly effective in modeling intermittent series (where zero sales prevail). The used metric is just the root mean squared error (there is no necessity to use a custom metric for representing the competition metric). We also use the `force_row_wise` parameter in order to save

memory in the Kaggle notebook. All the other parameters are exactly the ones presented by Monsaraida in his solution (apart from the subsampling parameter that has been disabled because of its incompatibility with the goss boosting type).

> In what other Kaggle competition the Tweedie loss has proven useful? Can you find useful discussions about this loss and its usage in Meta Kaggle by exploring the ForumTopics and ForumMessages tables (https://www.kaggle.com/datasets/kaggle/meta-kaggle)?

After defining the training parameters, we just iterate over the stores, each time uploading the training data of a single store and training the LightGBM model. Each model is the pickled. We also extract feature importance from each model in order to consolidate it into a file and then aggregate it resulting into having for each feature the mean importance across all the stores for that prediction horizon.

Here is the complete function for training all the models for a specific prediction horizon:

```
def train(train_df, seed, end_train_day_x, predict_horizon):

    lgb_params = {
            'boosting_type': 'goss',
            'objective': 'tweedie',
            'tweedie_variance_power': 1.1,
            'metric': 'rmse',
             #'subsample': 0.5,
             #'subsample_freq': 1,
            'learning_rate': 0.03,
            'num_leaves': 2 ** 11 - 1,
            'min_data_in_leaf': 2 ** 12 - 1,
            'feature_fraction': 0.5,
            'max_bin': 100,
            'boost_from_average': False,
            'num_boost_round': 1400,
            'verbose': -1,
            'num_threads': os.cpu_count(),
            'force_row_wise': True,
        }
    random.seed(seed)
    np.random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)

    lgb_params['seed'] = seed
    store_id_set_list = list(train_df['store_id'].unique())
    print(f"training stores: {store_id_set_list}")

    feature_importance_all_df = pd.DataFrame()
    for store_index, store_id in enumerate(store_id_set_list):
        print(f'now training {store_id} store')
        grid_df, enable_features = load_grid_by_store(end_train_day_x, predict_horizon, store
        train_mask = grid_df['d'] <= end_train_day_x
        valid_mask = train_mask & (grid_df['d'] > (end_train_day_x - predict_horizon))
        preds_mask = grid_df['d'] > (end_train_day_x - 100)
        train_data = lgb.Dataset(grid_df[train_mask][enable_features],
                                 label=grid_df[train_mask]['sales'])
        valid_data = lgb.Dataset(grid_df[valid_mask][enable_features],
                                 label=grid_df[valid_mask]['sales'])
        # Saving part of the dataset for later predictions
        # Removing features that we need to calculate recursively
        grid_df = grid_df[preds_mask].reset_index(drop=True)
        grid_df.to_feather(f'test_{store_id}_{predict_horizon}.feather')
        del(grid_df)
        gc.collect()

        estimator = lgb.train(lgb_params,
                              train_data,
                              valid_sets=[valid_data],
                              callbacks=[lgb.log_evaluation(period=100, show_stdv=False)],
                              )
        model_name = str(f'lgb_model_{store_id}_{predict_horizon}.bin')
```

```
        feature_importance_store_df = pd.DataFrame(sorted(zip(enable_features, estimator.feat
                                                   columns=['feature_name', 'importance'])
        feature_importance_store_df = feature_importance_store_df.sort_values('importance', a
        feature_importance_store_df['store_id'] = store_id
        feature_importance_store_df.to_csv(f'feature_importance_{store_id}_{predict_horizon}.
        feature_importance_all_df = pd.concat([feature_importance_all_df, feature_importance_
        pickle.dump(estimator, open(model_name, 'wb'))
        del([train_data, valid_data, estimator])
        gc.collect()
    feature_importance_all_df.to_csv(f'feature_importance_all_{predict_horizon}.csv', index=F
    feature_importance_agg_df = feature_importance_all_df.groupby(
        'feature_name')['importance'].agg(['mean', 'std']).reset_index()
    feature_importance_agg_df.columns = ['feature_name', 'importance_mean', 'importance_std']
    feature_importance_agg_df = feature_importance_agg_df.sort_values('importance_mean', asce
    feature_importance_agg_df.to_csv(f'feature_importance_agg_{predict_horizon}.csv', index=F
```

With the last function prepared, we got all the necessary code up for our pipeline to work. For the function wrapping the whole operations together, we need the input datasets (the time series dataset, the price dataset, the calendar information) together with the last training day (1,913 for predicting on the public leaderboard, 1,941 for the private one) and the predict horizon (which could be 7, 14, 21 or 28 days).

```
def train_pipeline(train_df, prices_df, calendar_df,
                   end_train_day_x_list, prediction_horizon_list):

    for end_train_day_x in end_train_day_x_list:

        for predict_horizon in prediction_horizon_list:

            print(f"end training point day: {end_train_day_x} - prediction horizon: {predict_
            # Data preparation
            generate_base_grid(train_df, end_train_day_x, predict_horizon)
            calc_release_week(prices_df, end_train_day_x, predict_horizon)
            generate_grid_price(prices_df, calendar_df, end_train_day_x, predict_horizon)
            generate_grid_calendar(calendar_df, end_train_day_x, predict_horizon)
            modify_grid_base(end_train_day_x, predict_horizon)
            generate_lag_feature(end_train_day_x, predict_horizon)
            generate_target_encoding_feature(end_train_day_x, predict_horizon)
            assemble_grid_by_store(train_df, end_train_day_x, predict_horizon)
            # Modelling
            train(train_df, seed, end_train_day_x, predict_horizon)
```

Since Kaggle notebook have a limited running time and a limited amount of both memory and disk space, our suggested strategy is to replicate four notebooks with the code hereby presented and train them with different prediction horizon parameters. Using the same name for the notebooks but for a part containing the value of the prediction parameter will help gathering and handling the models later as external datasets in another notebook.

Here is the first notebook, m5-train-day-1941-horizon-7 (https://www.kaggle.com/code/lucamassaron/m5-train-day-1941-horizon-7):

```
end_train_day_x_list = [1941]
prediction_horizon_list = [7]
seed = 42
train_pipeline(train_df, prices_df, calendar_df, end_train_day_x_list, prediction_horizon_lis
```

The second notebook, m5-train-day-1941-horizon-14 (https://www.kaggle.com/code/lucamassaron/m5-train-day-1941-horizon-14):

```
end_train_day_x_list = [1941]
prediction_horizon_list = [14]
seed = 42
train_pipeline(train_df, prices_df, calendar_df, end_train_day_x_list, prediction_horizon_lis
```

The third notebook, m5-train-day-1941-horizon-21 (https://www.kaggle.com/code/lucamassaron/m5-train-day-1941-horizon-21):

```
end_train_day_x_list = [1941]
prediction_horizon_list = [21]
seed = 42
train_pipeline(train_df, prices_df, calendar_df, end_train_day_x_list, prediction_horizon_lis
```

Finally the last one, m5-train-day-1941-horizon-28 (https://www.kaggle.com/code/lucamassaron/m5-train-day-1941-horizon-28):

```
end_train_day_x_list = [1941]
prediction_horizon_list = [28]
seed = 42
train_pipeline(train_df, prices_df, calendar_df, end_train_day_x_list, prediction_horizon_lis
```

If you are working on a local computer with enough disk space and memory resources, you can just run all the four prediction horizons together, by using as an input the list containing them all: [7, 14, 21, 28]. Now the last step before being able to submit our prediction is assembling the predictions.

Assembling public and private predictions

You can see an example about how we assembled the predictions for both the public and private leaderboards here:

- Public leaderboard example: https://www.kaggle.com/lucamassaron/m5-predict-public-leaderboard
- Private leaderboard example: https://www.kaggle.com/code/lucamassaron/m5-predict-private-leaderboard

What changes between the public and private submissions is just the different last training day: it determinates what days we are going to predict.

In this conclusive code snippet, after loading the necessary packages, such as LightGBM, for every end of training day, and for every prediction horizon, we recover the correct notebook with its data. Then, we iterate through all the stores and predict the sales for all the items in the time ranging from the previous prediction horizon up to the present one. In this way, every model will predict on a single week, the one it has been trained on.

```
import numpy as np
import pandas as pd
import os
import random
import math
from decimal import Decimal as dec
import datetime
import time
import gc
import lightgbm as lgb
import pickle
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
store_id_set_list = ['CA_1', 'CA_2', 'CA_3', 'CA_4', 'TX_1', 'TX_2', 'TX_3', 'WI_1', 'WI_2',
end_train_day_x_list = [1913, 1941]
prediction_horizon_list = [7, 14, 21, 28]
pred_v_all_df = list()
for end_train_day_x in end_train_day_x_list:
    previous_prediction_horizon = 0
    for prediction_horizon in prediction_horizon_list:
        notebook_name = f"../input/m5-train-day-{end_train_day_x}-horizon-{prediction_horizor
        pred_v_df = pd.DataFrame()
```

```
        for store_index, store_id in enumerate(store_id_set_list):

            model_path = str(f'{notebook_name}/lgb_model_{store_id}_{prediction_horizon}.bin'
            print(f'loading {model_path}')
            estimator = pickle.load(open(model_path, 'rb'))
            base_test = pd.read_feather(f"{notebook_name}/test_{store_id}_{prediction_horizon
            enable_features = [col for col in base_test.columns if col not in ['id', 'd', 'sa

            for predict_day in range(previous_prediction_horizon + 1, prediction_horizon + 1)
                print('[{3} -> {4}] predict {0}/{1} {2} day {5}'.format(
                store_index + 1, len(store_id_set_list), store_id,
                previous_prediction_horizon + 1, prediction_horizon, predict_day))
                mask = base_test['d'] == (end_train_day_x + predict_day)
                base_test.loc[mask, 'sales'] = estimator.predict(base_test[mask][enable_featu

            temp_v_df = base_test[
                    (base_test['d'] >= end_train_day_x + previous_prediction_horizon + 1) &
                    (base_test['d'] < end_train_day_x + prediction_horizon + 1)
                    ][['id', 'd', 'sales']]

            if len(pred_v_df)!=0:
                pred_v_df = pd.concat([pred_v_df, temp_v_df])
            else:
                pred_v_df = temp_v_df.copy()

            del(temp_v_df)
            gc.collect()

        previous_prediction_horizon = prediction_horizon
        pred_v_all_df.append(pred_v_df)

pred_v_all_df = pd.concat(pred_v_all_df)
```

When all the predictions have been gatherd, we merge them using the sample submission file as a reference, both for the required rows to be predicted and for the columns format (Kaggle expects distinct rows for items in the validation or testing periods with daily sales in progressive columns).

```
submission = pd.read_csv("../input/m5-forecasting-accuracy/sample_submission.csv")
pred_v_all_df.d = pred_v_all_df.d - end_train_day_x_list
pred_h_all_df = pred_v_all_df.pivot(index='id', columns='d', values='sales')
pred_h_all_df = pred_h_all_df.reset_index()
pred_h_all_df.columns = submission.columns
submission = submission[['id']].merge(pred_h_all_df, on=['id'], how='left').fillna(0)
submission.to_csv("m5_predictions.csv", index=False)
```

The solution can reach around 0.54907 in the private leaderboard, resulting in 12th position, in the gold medal area. Reverting back to Monsaraida's LightGBM parameters (for instance using gbdt instead of goss for the boosting parameter) should result even in higher performances (but you would need to run the code in local computer or on the Google Cloud Platform).

Exercise

> As an exercise, try comparing a training of LightGBM using the same number of iterations with the boosting set to gbdt instead of goss. How much is the difference in performance and training time (you may need to use a local machine or cloud one because the training may exceed the 12 hours)?

Summary

In this second chapter, we took on quite a complex time series competition, hence the easiest top solution we tried, it is actually fairly complex, and it requires coding quite a lot of processing functions. After you went through the chapter, you should have a better idea of how to process time series and have them predicted using gradient boosting. Favoring gradient boosting solutions over traditional methods, when you have enough data, as with this problem, should help you create strong solutions

for complex problems with hierarchical correlations, intermittent series and availability of covariates such as events or prices or market conditions. In the following chapters, you will tackle with even more complex Kaggle competitions, dealing with images and texts. You will be amazed at how much you can learn by recreating top-scoring solutions and understanding their inner workings.

03 Cassava Leaf Disease competition

In this chapter we will leave the domain of tabular data and focus on image processing. In order to demonstrate the steps necessary to do well in classification competitions, we will use the data from the Cassava Leaf Disease contest:

https://www.kaggle.com/competitions/cassava-leaf-disease-classification

The first thing to do upon starting a Kaggle competition is to read the descriptions properly:

> *"As the second-largest provider of carbohydrates in Africa, cassava is a key food security crop grown by smallholder farmers because it can withstand harsh conditions. At least 80% of household farms in Sub-Saharan Africa grow this starchy root, but viral diseases are major sources of poor yields. With the help of data science, it may be possible to identify common diseases so they can be treated."*

So this competition relates to an actually important real-life problem - your mileage might vary, but in general it is useful to know that.

> *"Existing methods of disease detection require farmers to solicit the help of government-funded agricultural experts to visually inspect and diagnose the plants. This suffers from being labor-intensive, low-supply and costly. As an added challenge, effective solutions for farmers must perform well under significant constraints, since African farmers may only have access to mobile-quality cameras with low-bandwidth."*

This paragraph - especially the last sentence - sets the expectations: since the data is coming from diverse sources, we are likely to have some challenges related to quality of the images and (possibly) distribution shift.

> *"Your task is to classify each cassava image into four disease categories or a fifth category indicating a healthy leaf. With your help, farmers may be able to quickly identify diseased plants, potentially saving their crops before they inflict irreparable damage."*

This bit is rather important: it specifies that this is a classification competition, and the number of classes is small (5).

With the introductory footwork out of the way, let us have a look at the data.

Understanding the data and metrics

Upon entering the "Data" tab for this competition, we see the summary of the provided data:

Dataset Description

Can you identify a problem with a cassava plant using a photo from a relatively inexpensive camera? This competition will challenge you to distinguish between several diseases that cause material harm to the food supply of many African countries. In some cases the main remedy is to burn the infected plants to prevent further spread, which can make a rapid automated turnaround quite useful to the farmers.

Files

[train/test]_images the image files. The full set of test images will only be available to your notebook when it is submitted for scoring. Expect to see roughly 15,000 images in the test set.

train.csv

- `image_id` the image file name.
- `label` the ID code for the disease.

sample_submission.csv A properly formatted sample submission, given the disclosed test set content.

- `image_id` the image file name.
- `label` the predicted ID code for the disease.

[train/test]_tfrecords the image files in tfrecord format.

label_num_to_disease_map.json The mapping between each disease code and the real disease name.

Figure 3.1: description of the Cassava competition dataset

What can we make of that?

- The data is in a fairly straightforward format, where the organisers even provided the mapping between disease names and numerical codes
- We have the data in tfrecord format, which is good news for anyone interested in using a TPU
- The provided test set is only a small subset and to be substituted with the full dataset at evaluation time. **This suggests that loading a previously trained model at evaluation time and using it for inference is a preferred strategy.**

The evaluation metric was chosen to be categorization accuracy: https://developers.google.com/machine-learning/crash-course/classification/accuracy. This metric takes discrete values as

inputs, which means potential ensembling strategies become somewhat more involved. Loss function is implemented during training to optimise a learning function and as long as we want to use methods based on gradient descent, this one needs to be continuous; evaluation metric, on the other hand, is used after training to measure overall performance and as such, can be discrete.

Exercise: without building a model, write a code to conduct a basic EDA

- Compare the cardinality of classes in our classification problem

Normally, this would also be the moment to check for distribution shift: if the images are very different in train and test set, it is something that most certainly needs to be taken into account. However, since we do not have access to the complete dataset in this case, the step is omitted - please check Chapter 1 for a discussion of adversarial validation, which is a popular technique for detecting concept drift between datasets.

Building a baseline model

We start our approach by building a baseline solution. The notebook running an end-to-end solution is available at:

While hopefully useful as a starting point for other competitions you might want to try, it is more educational to follow the flow described in this section, i.e. copying the code cell by cell, so that you can understand it better (and of course improve on it - it is called a baseline solution for a reason).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import tensorflow as tf
from tensorflow.keras import models, layers
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.optimizers import Adam

# ignoring warnings
import warnings
warnings.simplefilter("ignore")

import os, cv2, json
from PIL import Image
```

Figure 3.2: the imports needed for our baseline solution

We begin by importing the necessary packages - while personal differences in style are a natural thing, it is our opinion that gathering the imports in one place makes the code easier to maintain as the competition progresses and you move towards more elaborate solutions. In addition, we create a configuration class: a placeholder for all the parameters defining our learning process:

```
class CFG:
    # config
    WORK_DIR = '../input/cassava-leaf-disease-classification'
    BATCH_SIZE = 8
    EPOCHS = 5
    TARGET_SIZE = 512
```

Figure 3.3: configuration class for our baseline solution

The components include:

- The data folder is mostly useful if you train models outside of Kaggle sometimes (e.g. in Google Colab or on your local machine)
- BATCH_SIZE is a parameter that sometimes needs adjusting if you want to optimise your training process (or make it possible at all, for large images in constrained memory environment)
- Modifying EPOCHS is useful for debugging: start with small number of epochs to verify that your solution is running smoothly end-to-end and increase as you are moving towards a proper solution
- TARGET_SIZE defines the size to which we want to rescale our images
- NCLASSES corresponds to the number of possible classes in our classification problem

A good practice for coding a solution is to encapsulate the important bits in functions - and creating our trainable model certainly qualifies as important:

```
def create_model():
    conv_base = EfficientNetB0(include_top = False, weights = None,
                               input_shape = (CFG.TARGET_SIZE, CFG.TARGET_SIZE, 3))
    model = conv_base.output
    model = layers.GlobalAveragePooling2D()(model)
    model = layers.Dense(5, activation = "softmax")(model)
    model = models.Model(conv_base.input, model)

    model.compile(optimizer = Adam(lr = 0.001),
                  loss = "sparse_categorical_crossentropy",
                  metrics = ["acc"])
    return model
```

Figure 3.4: function to create our model

Few remarks around this step:

- While more expressive options are available, it is practical to begin with a fast model that can quickly iterated upon;

EfficientNet https://paperswithcode.com/method/efficientnet architecture fits the bill quite well
- We add a pooling layer for regularisation purposes
- Add a classification head - a Dense layer with CFG.NCLASSES indicating the number of possible results for the classifier
- Finally, we compile the model with loss and metric corresponding to the requirements for this competition.

Exercise: Examine the possible choices for loss and metric - a useful guide is https://keras.io/api/losses/ What would the other reasonable options be?

Next step is the data:

```
train_labels = pd.read_csv(os.path.join(CFG.WORK_DIR, "train.csv"))

STEPS_PER_EPOCH = len(train_labels)*0.8 / CFG.BATCH_SIZE
VALIDATION_STEPS = len(train_labels)*0.2 / CFG.BATCH_SIZE
```

```
train_labels.label = train_labels.label.astype('str')

train_datagen = ImageDataGenerator(validation_split = 0.2, preprocessing_function = None,
                                   rotation_range = 45, zoom_range = 0.2,
                                   horizontal_flip = True, vertical_flip = True,
                                   fill_mode = 'nearest', shear_range = 0.1,
                                   height_shift_range = 0.1, width_shift_range = 0.1)

train_generator = train_datagen.flow_from_dataframe(train_labels, directory = os.path.join(CFG.W
ORK_DIR, "train_images"),
                        subset = "training", x_col = "image_id",
                        y_col = "label", target_size = (CFG.TARGET_SIZE, CFG.TARGET_SIZE),
                        batch_size = CFG.BATCH_SIZE, class_mode = "sparse")


validation_datagen = ImageDataGenerator(validation_split = 0.2)

validation_generator = validation_datagen.flow_from_dataframe(train_labels,
                        directory = os.path.join(CFG.WORK_DIR, "train_images"),
                        subset = "validation", x_col = "image_id",
                        y_col = "label", target_size = (CFG.TARGET_SIZE, CFG.TARGET_SIZE),
                        batch_size = CFG.BATCH_SIZE, class_mode = "sparse")
```

Figure 3.5: setting up the data generator

Next we setup the model - straightforward, thanks to the function we defined above:

```
model = create_model()
model.summary()
```

Figure 3.6: instantiating the model

Before we proceed with training the model, we should dedicate some attention to callbacks:

```
model_save = ModelCheckpoint('./EffNetB0_512_8_best_weights.h5',
                             save_best_only = True,
                             save_weights_only = True,
                             monitor = 'val_loss',
                             mode = 'min', verbose = 1)
early_stop = EarlyStopping(monitor = 'val_loss', min_delta = 0.001,
                           patience = 5, mode = 'min', verbose = 1,
                           restore_best_weights = True)
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.3,
                              patience = 2, min_delta = 0.001,
                              mode = 'min', verbose = 1)
```

Figure 3.7: Model callbacks

Some points worth mentioning:

- ModelCheckpoint is used to ensure we keep the weights for the best model only, where the optimality is decided on the basis of the metric to monitor (validation loss in this instance)
- EarlyStopping helps us control the risk of overfitting by ensuring that the training is stopped if the validation loss does not improve (decrease) for a given number of epochs
- ReduceLROnPlateau is a schema for learning rate

Exercise: what parameters would it make sense to modify in the above setup, and which ones can be left at their default values?

With this setup, we can fit the model:

```
history = model.fit(
    train_generator,
    steps_per_epoch = STEPS_PER_EPOCH,
    epochs = CFG.EPOCHS,
    validation_data = validation_generator,
    validation_steps = VALIDATION_STEPS,
    callbacks = [model_save, early_stop, reduce_lr]
)
```

Figure 3.8: Fitting the model

Once the training is complete, we can use the model to build the prediction of image class for each image in the test set. Recall that in this competition the public (visible) test set consisted of a single

image and the size of the full one was unknown - hence the need for a slightly convoluted manner of constructing the submission dataframe

```
ss = pd.read_csv(os.path.join(CFG.WORK_DIR, "sample_submission.csv"))

preds = []

for image_id in ss.image_id:
    image = Image.open(os.path.join(CFG.WORK_DIR,  "test_images", image_id))
    image = image.resize((CFG.TARGET_SIZE, CFG.TARGET_SIZE))
    image = np.expand_dims(image, axis = 0)
    preds.append(np.argmax(model.predict(image)))

ss['label'] = preds
ss.to_csv('submission.csv', index = False)
```

Figure 3.9: Generating a submission

In this section we have demonstrated how to start to compete in a competition focused on image classification - you can use this approach to move quickly from basic EDA to a functional submission. However, a rudimentary approach like this is unlikely to produce very competitive results. For this reason, in the next section we discuss more specialised techniques that were utilised in top scoring solutions.

Learnings from top solutions

In this section we gather aspects from the top solutions that could allow us to rise above the level of the baseline solution. Keep in mind that the leaderboard (both public and private) in this competition were quite tight; this was a combination of a few factors:

- The noisy data - it was easy to get to .89 accuracy by correctly identifying large part of the train data, and then each new correct one allowed for a tiny move upward

- The metric - accuracy can be tricky to ensemble
- Limited size of the data

Pretraining

First and most obvious remedy to the issue of limited data size was pretraining: using more data. The Cassava competition was held a year before as well:

https://www.kaggle.com/competitions/cassava-disease/overview

With minimal adjustments, the data from the 2019 edition could be leveraged in the context of the current one. Several competitors addressed the topic:

- Combined 2019 + 2020 dataset in TFRecords format was released in the forum: https://www.kaggle.com/competitions/cassava-leaf-disease-classification/discussion/199131
- Winning solution from the 2019 edition served as a useful starting point: https://www.kaggle.com/competitions/cassava-leaf-disease-classification/discussion/216985
- Generating predictions on 2019 data and using the pseudo-labels to augment the dataset was reported to yield some (minor) improvements https://www.kaggle.com/competitions/cassava-leaf-disease-classification/discussion/203594

TTA

The idea behind Test Time Augmentation (TTA) is to apply different transformations to the test image: rotations, flipping and translations. This creates a few different versions of the test image and we generate a prediction for each of them. The resulting class probabilities are then averaged to get a more confident answer. An excellent demonstration of this technique is given in a notebook by

Andrew Khael: https://www.kaggle.com/code/andrewkh/test-time-augmentation-tta-worth-it

TTA was used extensively by the top solutions in the Cassava competition, an excellent example being the top3 private LB result: https://www.kaggle.com/competitions/cassava-leaf-disease-classification/discussion/221150

Transformers

While more known architectures like ResNext and EfficientNet were used a lot in the course of the competition, it was the addition of more novel ones that provided the extra edge to many competitors yearning for progress in a tightly packed leaderboard. Transformers emerged in 2017 as a revolutionary architecture for NLP (if somehow you missed the paper that started it all, here it is: https://arxiv.org/abs/1706.03762) and were such a spectacular success that inevitably many people started wondering if they could be applied to other modalities as well - vision being an obvious candidate. Aptly named Vision Transformer made one of its first appearances in a Kaggle competition in the Cassava contest. An excellent tutorial for ViT has been made public:

https://www.kaggle.com/code/abhinand05/vision-transformer-vit-tutorial-baseline

Vision Transformer was an important component of the winning solution:

https://www.kaggle.com/competitions/cassava-leaf-disease-classification/discussion/221150

Ensembling

The core idea of ensembling is very popular on Kaggle (see Chapter 9 of the Kaggle Book for a more elaborate description) and Cassava

competition was no exception. As it turned out, combining diverse architectures was very beneficial (by averaging the class probabilities): EfficientNet, ResNext and ViT are sufficiently different from each other that their predictions complement each other. Another important angle was stacking, i.e. using models in two stages: the 3rd place solution combined predictions from multiple models and then applied LightGBM as a blender

https://www.kaggle.com/competitions/cassava-leaf-disease-classification/discussion/220751

The winning solution involved a different approach (with fewer models in the final blend), but relying on the same core logic:

https://www.kaggle.com/competitions/cassava-leaf-disease-classification/discussion/221957

Summary

In this chapter, we have described how to get started with a baseline solution for an image classification competition and discussed a diverse set of possible extensions for moving to a competitive (medal) zone.