

Unlocking the power of gradient-boosted trees using LightGBM

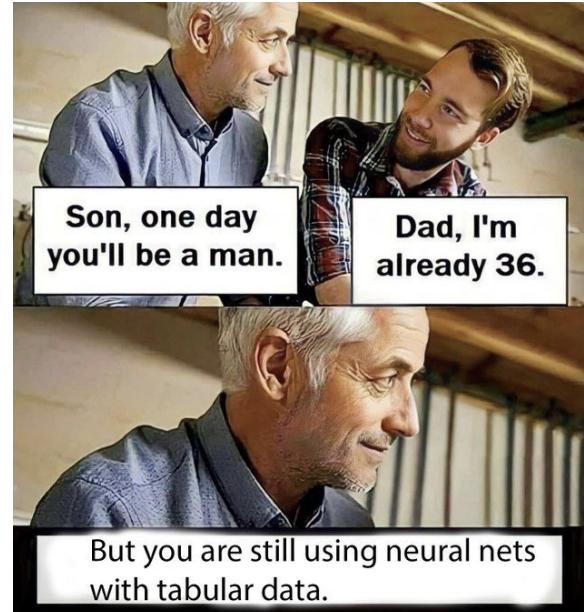
Pedro Tabacof
PyData London 2022

LightGBM: the swiss-army knife for tabular ML

Objectives:

1. Convince you to start using LightGBM or XGBoost for tabular/structured problems
2. Give a taste of its more interesting features and explain how they work

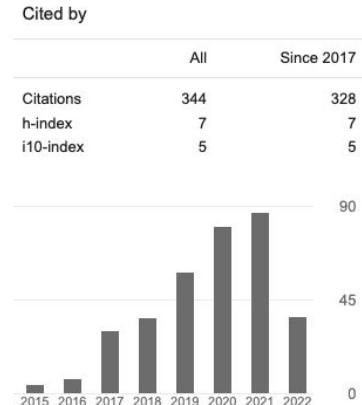
I will not go over "unstructured problems" such as document or image classification, speech-to-text, language translation, image creation, etc.



[@tunguz](#)

Personal introduction

- Originally from Brazil, now based in Dublin
- Staff data scientist at a mobile gaming company (Wildlife Studios)
- Five years of experience with applied machine learning (antifraud, credit risk, churn, lifetime value, and marketing attribution)
- Master's degree on Deep Learning (variational autoencoders), but never actually used DL to solve any business problem!



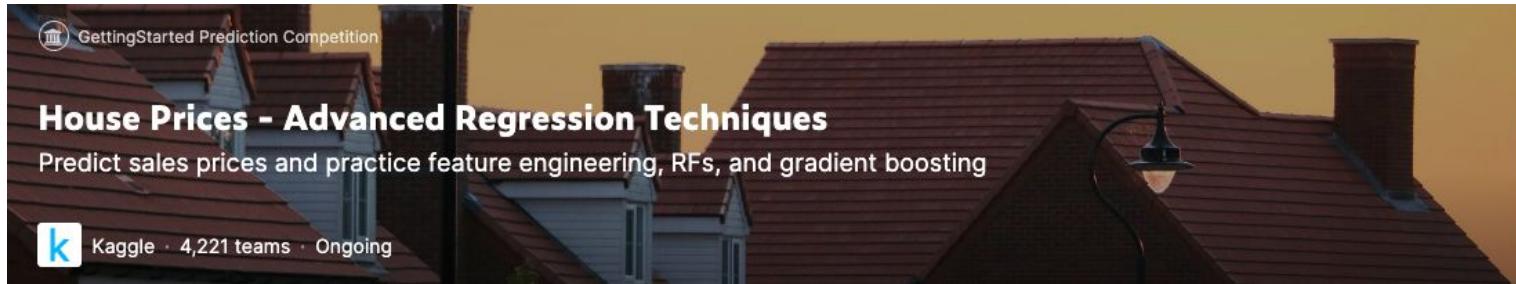
Agenda

- Motivating example
- How gradient-boosted trees work
- What are the important tuning parameters
- Handling missing values and categorical features
- The loss function menu, including custom losses
- Early stopping, sample weights, and other cool features
- Model interpretability

Motivating example: Kaggle house prices prediction

Standard regression problem:

- Predict house prices with 80 features available of different types
- Only 3k samples in total, half for training and half for testing
- Evaluated on the RMSE of the log-price
- Not real-life but instructive (*Kaggle is never real-life anyway!*)



Trivial LightGBM baseline

In 18 lines we can read the data, train a LightGBM model with no tuning or feature engineering whatsoever, score the test set and have a submission ready.

But how does it perform?

```
import pandas as pd
import lightgbm as lgbm
import numpy as np

train = pd.read_csv("house-prices-advanced-regression-techniques/train.csv")
train["train"] = True

test = pd.read_csv("house-prices-advanced-regression-techniques/test.csv")
test["train"] = False
test["SalePrice"] = np.nan

df = pd.concat((train, test))

cat_features = [c for c, dt in df.dtypes.items() if dt == 'object']
df[cat_features] = df[cat_features].astype("category")

train_X = df[df.train].drop(["train", "SalePrice"], axis=1)
train_y = np.log(df[df.train]["SalePrice"])

reg = lgbm.LGBMRegressor()
reg.fit(train_X, train_y)

LGBMRegressor()

test_X = df[~df.train].drop(["train", "SalePrice"], axis=1)

df.loc[~df.train, "SalePrice"] = np.exp(reg.predict(test_X))

df.loc[~df.train, ["Id", "SalePrice"]].to_csv("submission.csv", index=False)
```

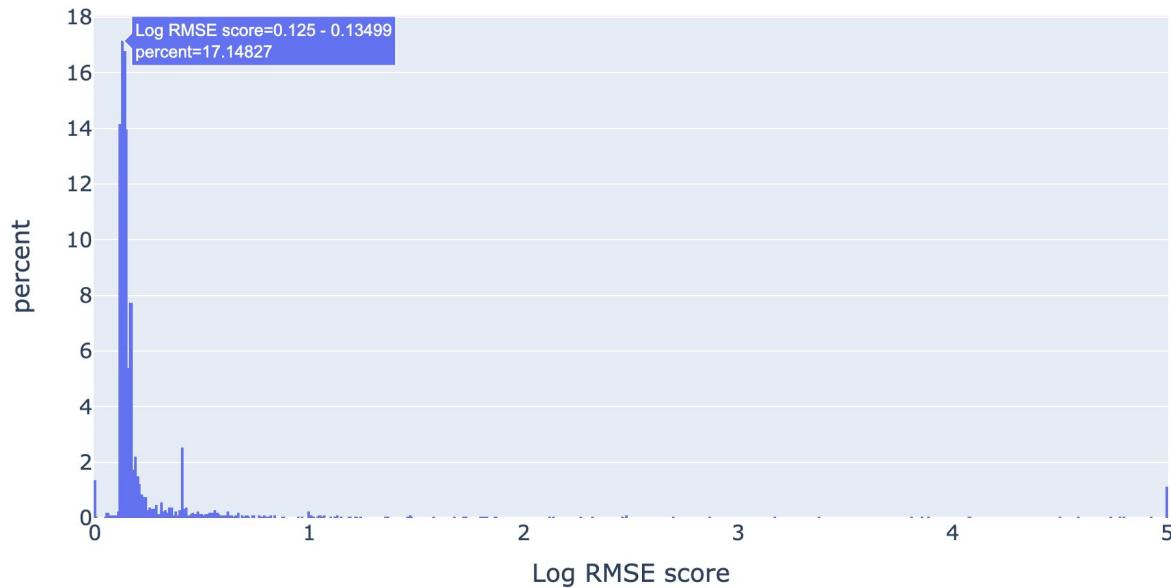
But how does it perform?

Not great, not terrible:

0.13 ~~roentgen~~ log RMSE (top 30% of the submissions).

Top solutions are usually a heavily tuned and engineered ensemble of diverse models.

Competition score distribution:

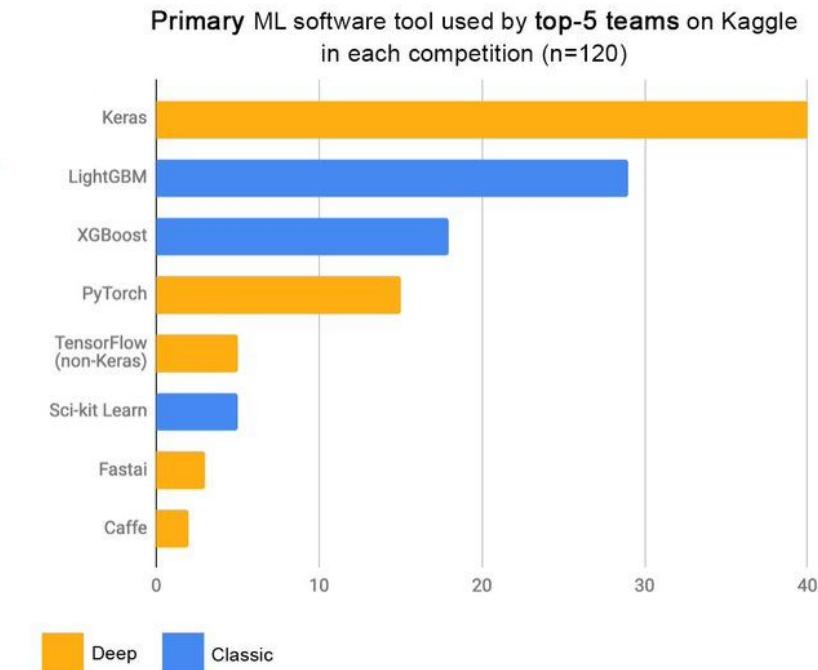


Kaggle gradient-boosted trees performance

Gradient-boosted trees (LightGBM, XGBoost, Catboost) dominate structured/tabular data competitions.

Neural networks dominate unstructured data competitions.

Not only can they give you a reasonable baseline with few lines of code, they can also get you to the top!



The LightGBM magic

The dataset is quite complex, even though it is small:

Overview Alerts 176 Reproduction

Dataset statistics		Variable types	
Number of variables	83	Numeric	39
Number of observations	2919	Categorical	42
Missing cells	15424	Boolean	2
Missing cells (%)	6.4%		
Duplicate rows	0		
Duplicate rows (%)	0.0%		
Total size in memory	1.8 MiB		
Average record size in memory	657.0 B		



Where are the standard scalers?

The missing value imputer?

The categorical encoders?

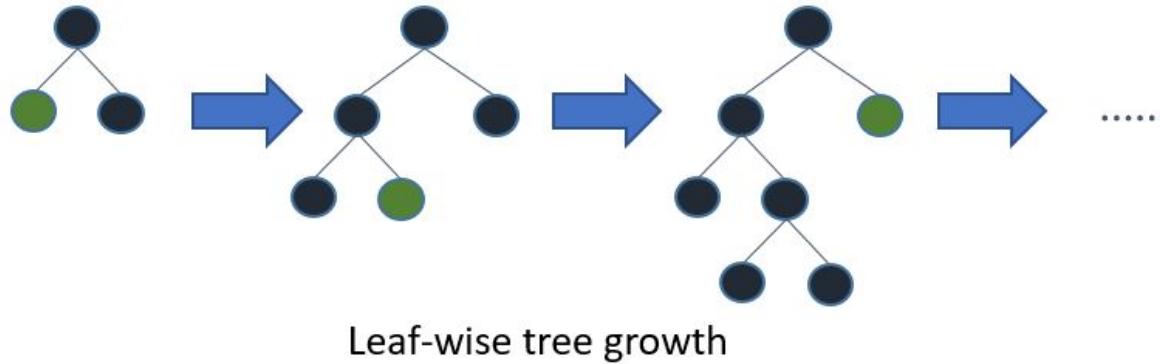
The parameter tuning?

The feature selection?

How does LightGBM handle all that under the hood?

How does gradient-boosting works?

First, let's start with a single regression tree:



LightGBM grows trees leaf-wise. It will choose the leaf with max delta loss to grow. This is based on the *gradient* of the loss: which threshold best splits the sum of gradients between each child? LightGBM provides a fast approximation using histograms of the gradients (i.e. by mapping to bins).

How does gradient-boosting works?

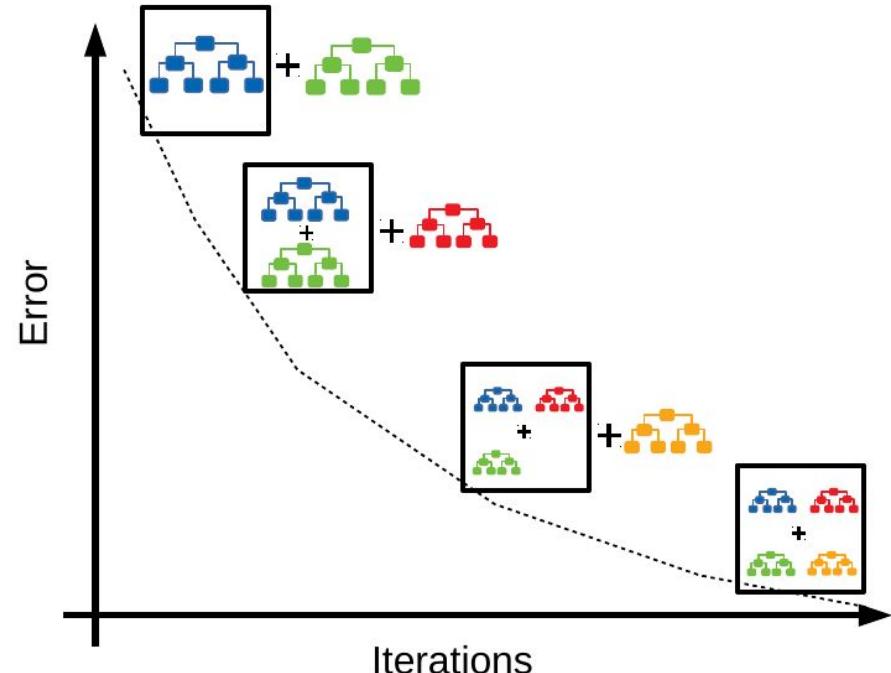
Second, let's ensemble them:

Each new tree tries to predict the residual (error at the current iteration).

Each tree's prediction is multiplied by the learning_rate to reduce overfitting.

The final prediction is:

```
initial_prediction +  
learning_rate*predicted_residual_1  
+ ... +  
learning_rate*predicted_residual_N
```



What are some parameters you can tune?

The most important parameters are the following:

- `n_estimators`: use between 100 and 1000
- `learning_rate` : use between 0.01 and 0.1
- `max_depth`: use between 5 and 20
- `num_leaves`: use default
- `bagging_fraction` or `feature_fraction`: to make it more like a RF
- `objective`: more on the choices here later

You can always tune them with grid search, random search, Bayesian optimization or early stopping for the number of trees (more on this later).

Missing values: standard answer

Whenever I interview data science candidates, I always ask how they would handle missing values in their dataset. A typical answer is structured like this:

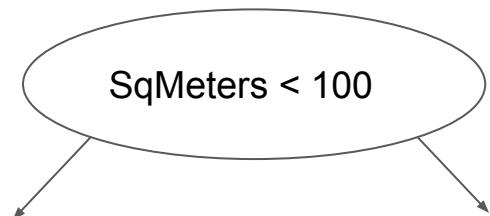
- Numerical features: use mean or median
- Categorical features: use unknown class

Sometimes I hear about creating a model to impute the missing values, but *I almost never hear that some models handle them natively!*

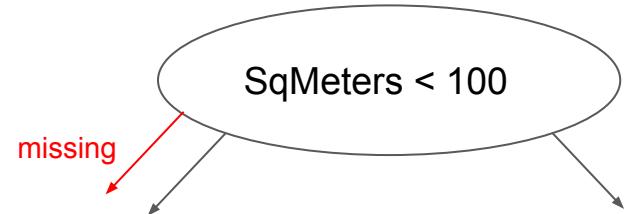
Missing values: LightGBM answer

Missing value splits are learned separately on each node:

Say we've a node that splits on Square Meters, below 100 it goes left and above it goes right:



If a missing value comes in the training set, it will create a separate missing split that minimizes the loss:



Missing values: disaster averted!

"Hey, my credit score on your model is 1000 (perfect), I'm pretty sure it is wrong as I'm a terrible credit card payer!"

-Business analyst to me at some fintech company

The model was indeed wrong: a large set of important features were missing in production for a small set of users, but they were never missing in training.

The default missing values behaviour was leading to low risk predictions.

Never accept missing in production if there were none in training.

Categorical encoding: standard answer

Typical answers:

- **Label** encoding: categories become numbers like 0, 1, 2, 3, 4, ...
(surprisingly good for trees!)
- **One-hot** encoding: categories become a set of binary features like 000, 001, 010, ... (not great for trees unless the cardinality is low)
- **Target** encoding: average target value by category (powerful but risky)
- **Frequency** encoding: category frequency (might be a useful complement)

Categorical encoding: LightGBM answer

LightGBM uses histogram splits for categorical features as well:

1. It bins the categories together based on their loss gradient
2. This creates a histogram which is then sorted according to the objective
3. It finds the best split on the sorted histogram

This process is not so unlike target encoding, which explains its usefulness.

For a model specialized in categorical data, check out [Catboost](#).

Loss functions menu

LightGBM offers ~15 different loss functions:

- Regression
 - L2: mean squared error (default, recovers the mean value)
 - L1: mean absolute error (good for outliers, recovers the median value*)
 - MAPE: mean absolute percentage error (good for time series)
 - Quantile: predict quantiles (might be used for prediction intervals)
 - Tweedie and gamma (good for skewed problems)
 - Poisson (good for count data)
- Classification
 - Logloss for binary classification (logloss leads to calibrated probabilities)
 - Multiclass and cross-entropy for multi-class problems
- Ranking (lambdaRank)
- Survival analysis: 😞 but XGBoost offers some possibilities

*see this awesome blog post on MSE vs MAE vs log-transform for skewed data problems: [Honey, I shrunk the target variable](#)

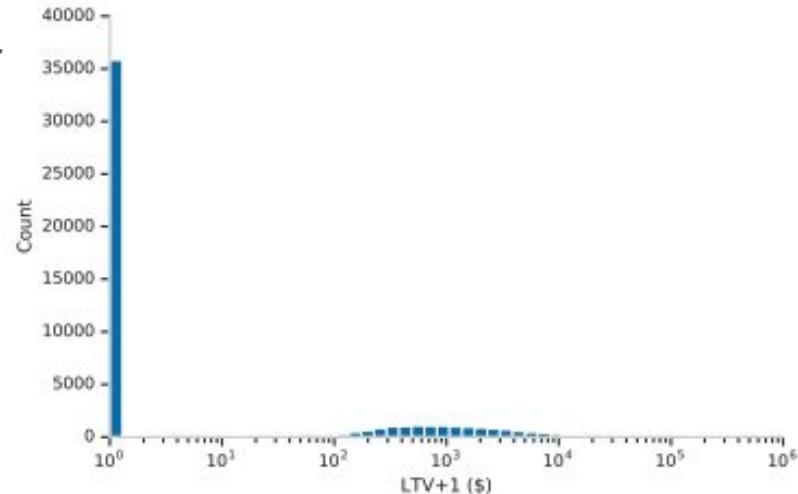
Lifetime value (LTV) modelling example

LTV is bimodal and right-skewed: most users don't spend a cent, some spend thousands.

First attempt: two-stage model, one classifier for paying users, one regressor to predict paying user value.

Second attempt: zero-inflated lognormal neural network.

Best solution: LightGBM with the Tweedie loss.

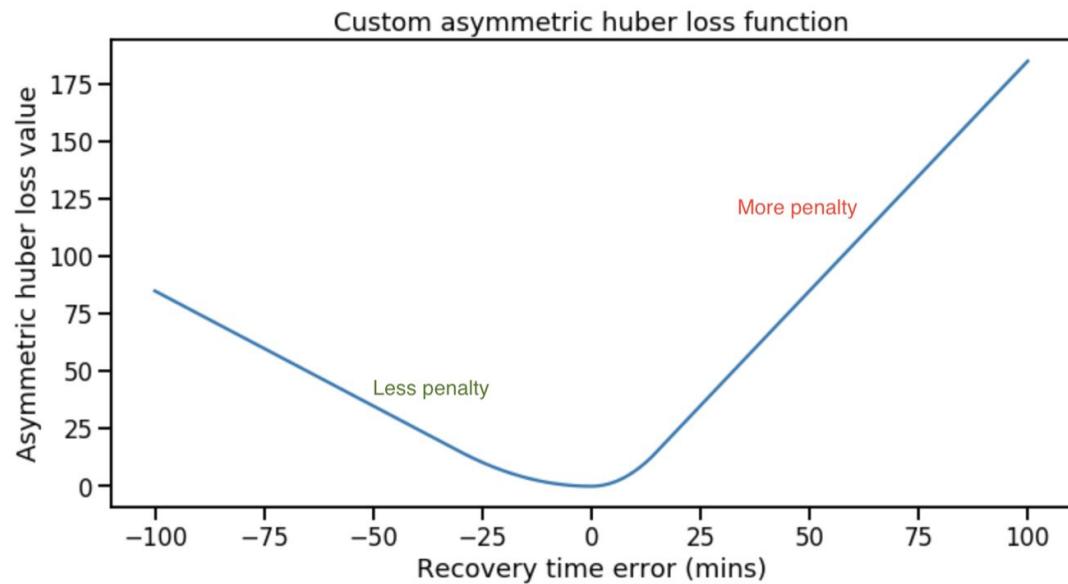


[A Deep Probabilistic Model for Customer Lifetime Value Prediction](#)

Custom loss functions

You can even create your own custom loss function! You "only" need to provide the gradient and hessian.

Simple application:
Asymmetric regression
when overpredicting is
worse than underpredicting.



[Custom Loss Functions for Gradient Boosting](#)

Custom loss functions

You can even create your own custom loss function! You "only" need to provide the gradient and hessian.

Simple application:
Asymmetric regression
when overpredicting is
worse than underpredicting.

```
def custom_asymmetric_train(y_true, y_pred):  
    residual = (y_true - y_pred).astype("float")  
    grad = np.where(residual<0, -2*10.0*residual, -2*residual)  
    hess = np.where(residual<0, 2*10.0, 2.0)  
    return grad, hess  
  
def custom_asymmetric_valid(y_true, y_pred):  
    residual = (y_true - y_pred).astype("float")  
    loss = np.where(residual < 0, (residual**2)*10.0, residual**2)  
    return "custom_asymmetric_eval", np.mean(loss), False
```

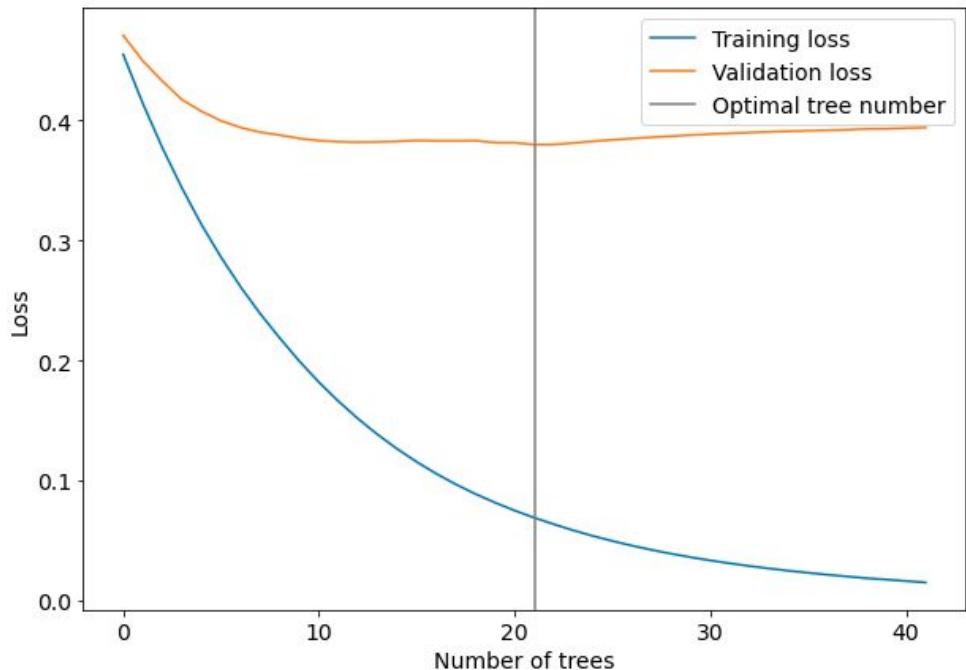
[Custom Loss Functions for Gradient Boosting](#)

Early stopping

Choose the number of iterations that minimizes the validation loss.

```
reg.fit(train_X, train_y,  
       callbacks=[lgbm.early_stopping(5)],  
       eval_set=(val_X, val_y))
```

For the Kaggle housing example, it netted a *tiny* score improvement (80 positions / 2% improvement).



[How to use early stopping in Xgboost training?](#)

Sample / class weights

Class weights (class creation): `class_weight (dict, 'balanced' or None)`

Sample weights (fit): `sample_weight (array-like of shape = [n_samples])`

Use them for imbalanced datasets instead of over/under sampling as it's more efficient and leads to similar results.

Just don't use SMOTE:



At last a paper on SMOTE that confirms my experience. TL;DR SMOTE is detrimental.



Gael Varoquaux @GaelVaroquaux · Jun 10

Replies to [@JFPuget](#)

Our experience is also that SMOTE is not useful.

[@JFPuget](#)

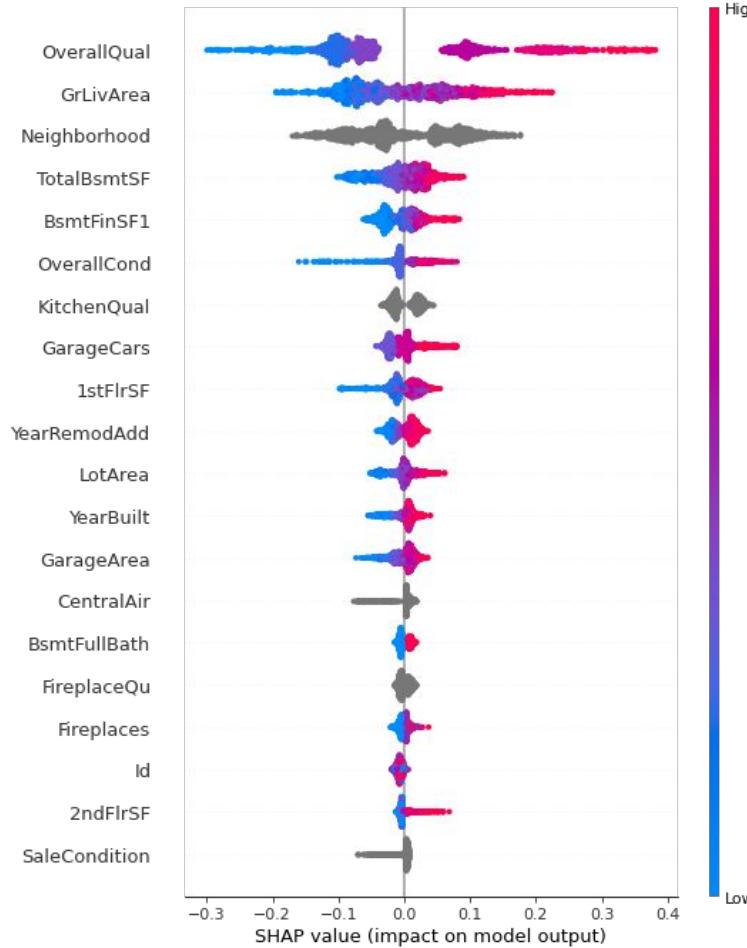
Other cool features

- **Monotonicity constraints**: force a monotonic relation between one or more features and the predictions, which is quite useful for interpretability or business constraints!
- **Random forest** "boosting" method: sometimes you just need a good old RF.
- **Distributed** computation and **GPU** support: never tried myself but XGBoost for Spark works wonders with large datasets.

Interpretability: SHAP

Use **SHAP** instead of split count or gain.

Individual examples below, overall importance on the right:



Conclusion and takeaways

- LightGBM and XGBoost provide not only a reasonable baseline with minimal tuning and engineering necessary, but they also allow you to get to the next level.
- Most of your tabular modelling needs are covered with built-in features or well-integrated libraries (e.g. Sklearn, pandas-profiling, SHAP).
- This leaves you time to focus on what really matters in applied ML:

Conclusion and takeaways

- LightGBM and XGBoost provide not only a reasonable baseline with minimal tuning and engineering necessary, but they also allow you to get to the next level.
- Most of your tabular modelling needs are covered with built-in features or well-integrated libraries (e.g. Sklearn, pandas-profiling, SHAP).
- This leaves you time to focus on what really matters in applied ML:
 - Building new features (new data sources or [feature engineering](#))
 - Proper evaluation methodology (random train-test split is never enough!)
 - Translating model results into the business bottomline (\$)
 - Automated decision making with the model outputs ("the last mile is the longest")
 - Model deployment infrastructure and [monitoring](#)

Thanks!



Bojan Tunguz
@tunguz

It takes a lot of study to learn all the necessary
Machine Learning techniques and tools.

It takes a lot of experience to learn which ones to
ignore.

Check out the notebook [here](#)

Follow me on Twitter at [@PedroTabacof](#) or add me on [Linkedin](#)