Member-only story

Proxy SHAP: Speed Up Explainability with Simpler Models

A Practical Guide to Efficient SHAP Computation

Marco Cerliani · Follow
Published in Towards Data Science · 6 min read · 5 days ago

Photo by Joel Muniz on Unsplash

Data scientists love doing experiments, training models, and making their hands dirty with data. At the beginning of a project, enthusiasm is at the top, but when things become complicated or too time-consuming, looking for simpler solutions is a real must.

There may be situations where business stakeholders ask to make changes to the underlying solution logic or to make further adjustments/trials while trying to improve performance and maintain a good explicative level of the predictive algorithms involved. Identifying possible bottlenecks in the code implementation, which may lead to additional complexity and delays in delivering the final product, is crucial.

Imagine being a data scientist and having the task of developing a predictive model. We have all that we need easily at our disposal and after a while, we are ready to present to the business people our fancy predictive solutions built on thousands of features and millions of records that achieve astonishing performances.

The business stakeholders are fascinated by our presentation and understand the technology's potential, but they added a request. They want to know how the model takes its decisions. Nothing easier we may think...

> *Let's import shap in our code. Add some fancy colorful plots and come back to the business people for the glory.*

When we are back to our PC, the first thing we try is to get the feature contributions (SHAP values) for each prediction sample but we notice that the SHAP computation is very slow when thousands of features and millions of records are involved. This may be a serious problem for our predictive solution being adopted in real time to make predictions and provide explanations. We urgently need to find a solution to speed up SHAP values

computation avoiding lowering model performances and expensive refactoring of our excellent work.

In this post, we propose a methodology that produces reliable SHAP values according to the knowledge learned by our model and is fast enough not to make the final user wait forever.

SHAP times as a function of model complexity

Everyone can encounter the need to make SHAP fast in their data science journey, but what makes SHAP so slow? This is an interesting question that we investigate empirically.

Given three different gradient-boosting predictive algorithms (Xgboost, LightGBM, and CatBoost), we track the times taken to retrieve SHAP varying three parameters: the data size, the three depths, and the number of estimators/iterations involved during the training phase.

	boosting	catboost	lightgbm	xgboost
data_size	iters			
1000	50	2.625551	1.262080	3.359591
	100	5.862500	2.896282	6.347808
	150	10.190148	4.698679	8.734616
5000	50	2.863789	6.313984	16.777241
	100	5.827972	14.730874	31.047446
	150	9.964898	22.941437	44.204207
10000	50	3.242032	12.949972	34.063649
	100	6.280196	29.619308	62.155219
	150	10.480905	45.927272	88.065559

Shap time performances varying data sizes and boosting iterations [Image by the author]

	boosting	catboost	lightgbm	xgboost
data_size	depth			
1000	4	0.092956	0.125212	0.238494
	6	0.226274	0.700163	1.404701
	8	1.869806	4.070031	5.803602
	10	22.715229	6.913982	17.142556
5000	4	0.423238	0.584308	1.185771
	6	0.558382	3.766703	7.140334
	8	2.221162	20.596160	29.228120
	10	21.672762	33.701224	85.150968
10000	4	0.838926	1.188322	2.379913
	6	1.046879	7.049983	14.151993
	8	2.663142	41.450261	58.176688
	10	22.121897	68.306836	171.003975

Shap time performances varying data sizes and depths [Image by the author]

As we can imagine, the more data we use the more time the process lasts. The computation time is also linearly dependent on the number of estimators/iterations while being exponentially related to the depth size of the decision trees involved in the boosting.

Given the above observations, how can we make the process faster without compromising the SHAP insights obtained by our best model?

Building Proxy SHAP

Our trained model, which we assume to be a CatBoost, is at our disposal and its performances satisfy us. We spent time and energy optimizing it. Repeating the whole training/optimization procedure using fewer iterations and depth dimensions could waste time since this would most likely worsen predictive performances.

```
import catboost as ctb
from sklearn import model_selection

CV = model_selection.KFold(5, shuffle=False)

model = model_selection.RandomizedSearchCV(
    ctb.CatBoostRegressor(verbose=0, thread_count=-1, random_state=123),
    {'n_estimators': stats.randint(1, 300),
     'depth': [4, 6, 8, 10]},
    random_state=123, n_iter=20, refit=True,
    cv=CV, scoring='neg_mean_absolute_error'
).fit(X_train, y_train)
```

What about training a new lighter model optimized to emulate the SHAP contributions of our original model? The idea sounds interesting. Let's give it a try.

> *The dataset used to obtain the following results is the "California Housing" directly available in scikit-learn (under an open BSD license) and originally available here [Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions, Statistics and Probability Letters, 33 (1997) 291–297].*

As a first step, we need to retrieve the SHAP values for some training samples. They will serve as a "ground truth" set for our lighter model.

```
class ShapCatBoostRegressor(ctb.CatBoostRegressor):
    def predict_shap(self, X):
        return self.get_feature_importance(
            ctb.Pool(X), type='ShapValues'
        )

ref_shap_val = model_selection.cross_val_predict(
    ShapCatBoostRegressor(
        **model.best_params_,
        verbose=0, thread_count=-1, random_state=123
    ),
    X_train, y_train,
    method='predict_shap',
    cv=CV
```

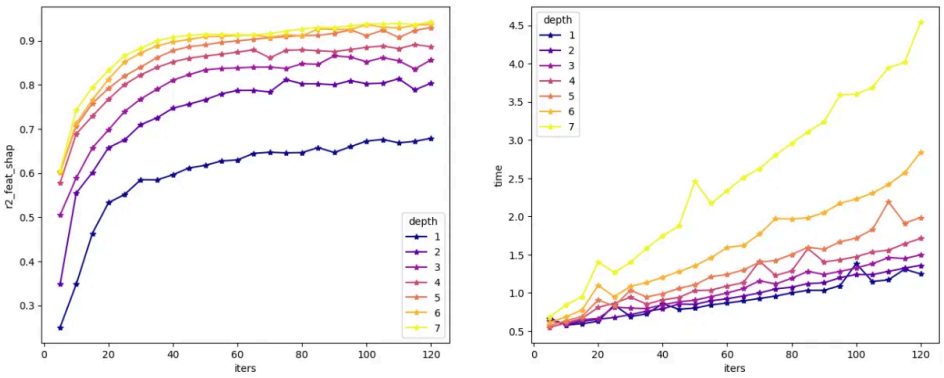
```
)

shap_feat_importance = np.abs(ref_shap_val[:,:-1]).mean(0)
shap_feat_importance /= shap_feat_importance.sum()
```

Secondly, we train smaller models varying the depths and the number of iterations. We are searching for a new model that reproduces the SHAP values faster and reliably. For each combination of depth and iteration, the SHAP values are retrieved and compared with the original one produced by our deeper model. The goodness of SHAP values is measured using standard error metrics like R2. In this way, we end up having an error score for each feature, measured as the difference between the original SHAP values and the approximated ones. A final and unique score is obtained as the mean of features' R2 (the same can be done sample-wise). More weight is assigned to error metrics of the features which are more important for our original model.

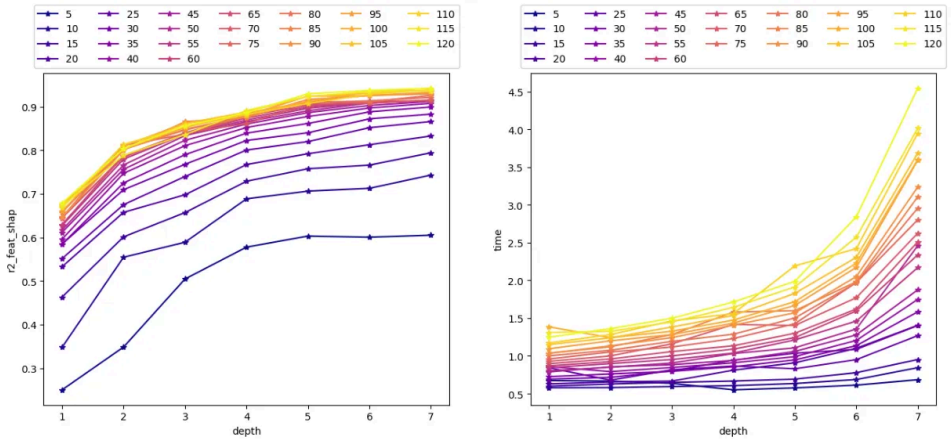
```
import datetime
import itertools

param_combi = {'iters': range(5,125,5), 'depth': range(1,8)}
for i,d in itertools.product(*param_combi.values()):

    start_time = datetime.datetime.now()
    shap_val = model_selection.cross_val_predict(
        ShapCatBoostRegressor(
            n_estimators=i, depth=d,
            verbose=0, thread_count=-1, random_state=123
        ),
        X_train, y_train,
        method='predict_shap',
        cv=CV
    )
    end_time = datetime.datetime.now()
    delta = (end_time - start_time).total_seconds()

    result.append({
        "time": delta,
        "iters": i,
        "depth": d,
        "r2_feat_shap": np.average(
```

```
                metrics.r2_score(
                    ref_shap_val[:,:-1], shap_val[:,:-1],
                    multioutput='raw_values'
                ).round(3)
            weights=shap_feat_importance
        ),
        "r2_sample_shap": metrics.r2_score(
            ref_shap_val[:,:-1].sum(1), shap_val[:,:-1].sum(1)
        ),
    })

result = pd.DataFrame(result)
```

Inspecting the results of the search process this is what we can observe:

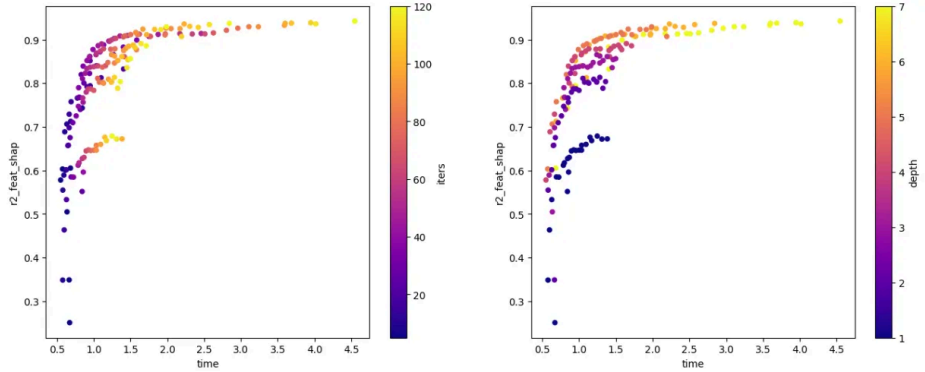


On the left, R2 Shap feature scores vs boosting iterations. On the right, time for Shap computation vs boosting iterations [Image by the author]



On the left, R2 Shap feature scores vs boosting depths. On the right, time for Shap computation vs boosting depths [Image by the author]

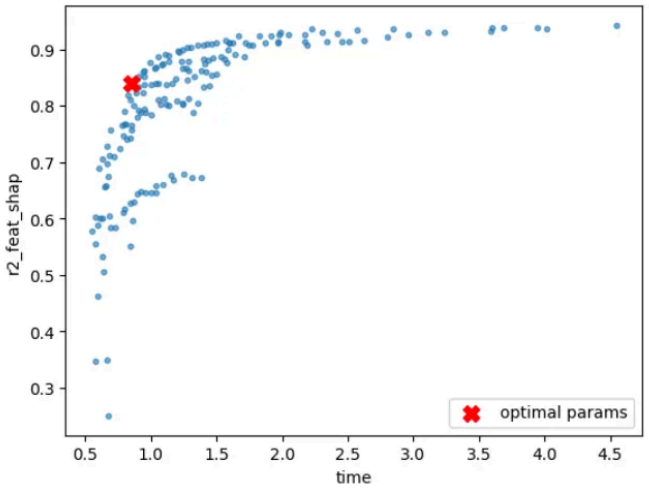
As the number of iterations or depth increases, the time taken to compute SHAP values increases and the accuracy (R2) of SHAP approximation gets higher. A result that is not surprising but confirms our initial hypothesis.



R2 Shap feature scores and times of Shap computation for all parameters combinations [Image by the author]

The model that best approximates the SHAP calculation is the one with the best trade-off between accuracy and time. More weight can be assigned to accuracy or time depending on what we would like to prioritize.

```
def distance(time, r2, w_time=0.1, w_r2=0.9):
    return ((time - result.time.min()) **2) *w_time + \
            ((result.r2_feat_shap.max() - r2) **2) *w_r2

result['distance'] = result.apply(
    lambda x: distance(x.time, x.r2_feat_shap), axis=1
)
result = result.sort_values('distance')

proxy_model = ctb.CatBoostRegressor(
    n_estimators=result.head(1).iters.squeeze(),
    depth=result.head(1).depth.squeeze(),
    verbose=0, thread_count=-1, random_state=123,
).fit(X_train, y_train)
```



Optimal parameter combination [Image by the author]

Passing from around 3 seconds to around 3 microseconds, we register a clear benefit in terms of time taken to compute SHAP values on a test set of 10k samples.

```
%%time
proxy_shap = proxy_model.get_feature_importance(
    ctb.Pool(X_test), type='ShapValues'
)
proxy_shap.shape
```

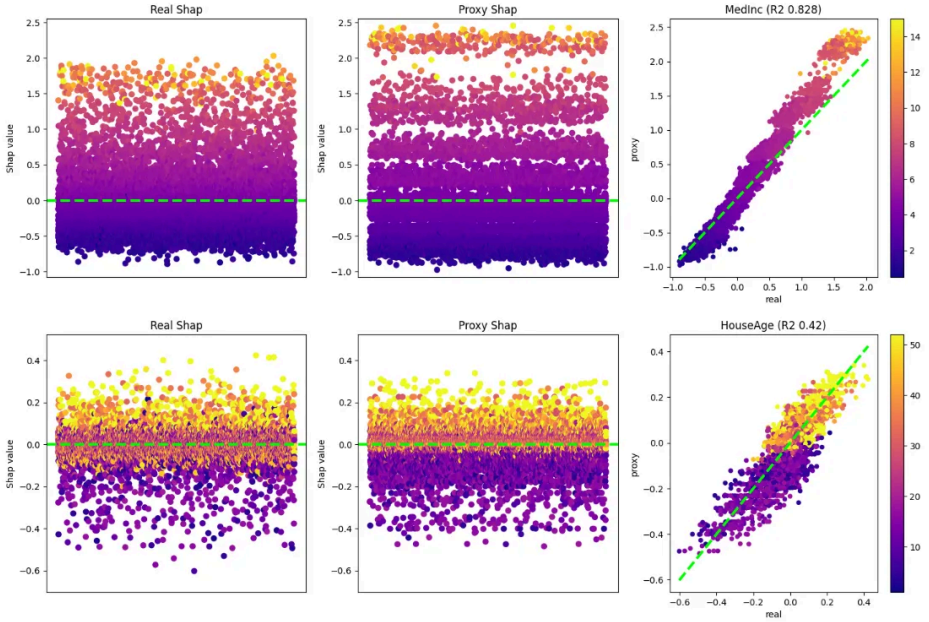
```
CPU times: user 1.22 s, sys: 32.5 ms, total: 1.25 s
Wall time: 356 ms
(10000, 9)
```

```
%%time
best_shap = model.best_estimator_.get_feature_importance(
    ctb.Pool(X_test), type='ShapValues'
)
best_shap.shape
```

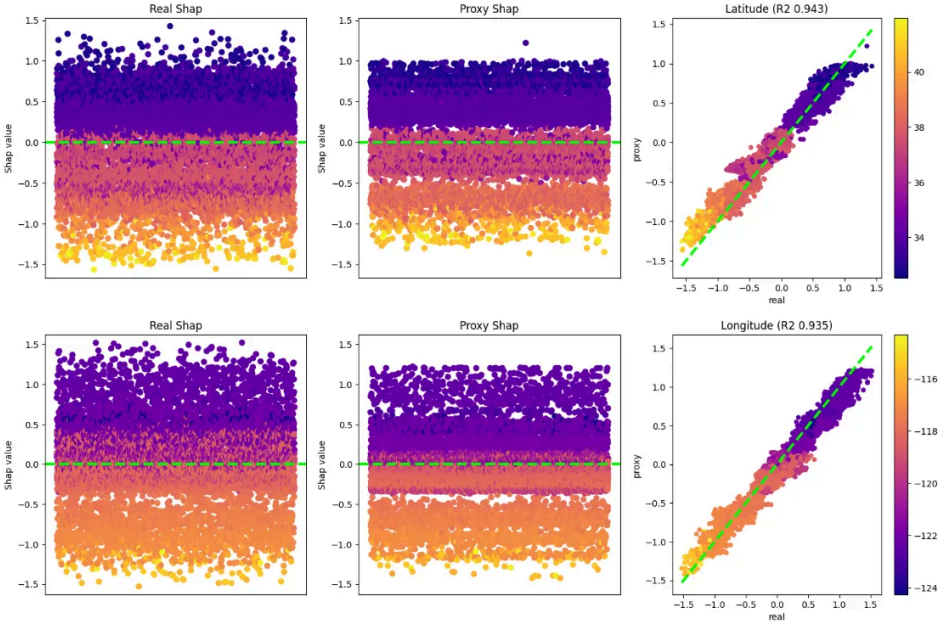
```
CPU times: user 11.9 s, sys: 97 ms, total: 12 s
Wall time: 3.11 s
(10000, 9)
```

Time taken to compute shap [Image by the author]

A final comparison, between the original model and the proxy model, shows also how well the proxy SHAP values can approximate the real values on unseen data, especially on the most important features.



Original vs Proxy Shap value comparison on unseen data [Image by the author]



Original vs Proxy Shap value comparison on unseen data [Image by the author]

Summary

In this post, we proposed a methodology to compute reliable SHAP values by using simpler and lighter models. The approach consists of training a lighter model to emulate the SHAP contributions of an original and heavy model. By optimizing parameters such as depth and iterations, we achieved a balance between speed and accuracy. This enables significantly reduced computation time while maintaining reliable SHAP value approximations, offering a practical solution for real-time predictive applications.

. . .

CHECK MY GITHUB REPO

Keep in touch: Linkedin

Machine Learning | Data Science | Explainable Ai | Shapley Values | Hands On Tutorials

Written by Marco Cerliani

6.9K Followers · Writer for Towards Data Science

Follow

Statistician Hacker & Data Scientist

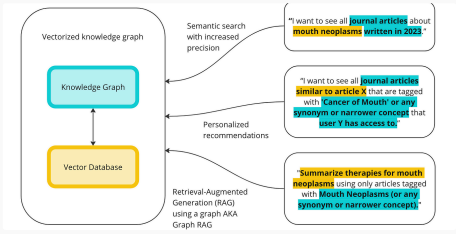
More from Marco Cerliani and Towards Data Science

Marco Cerliani in Towards Data Science

Time2Vec for Time Series features encoding

Learn a valuable representation of time for your Machine Learning Model

Sep 25, 2019 458 6

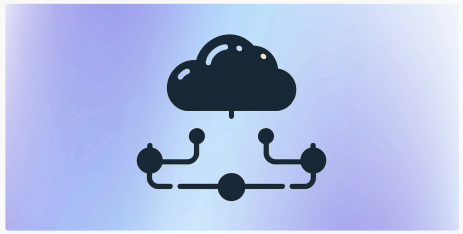


Steve Hedden in Towards Data Science

How to Implement Graph RAG Using Knowledge Graphs and...

A Step-by-Step Tutorial on Implementing Retrieval-Augmented Generation (RAG),...

Sep 6 1.2K 12

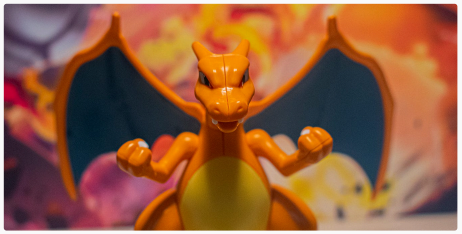


Aparna Dhinakaran in Towards Data Science

Navigating the New Types of LLM Agents and Architectures

The failure of ReAct agents gives way to a new generation of agents— and possibilities

Aug 30 1.4K 10



Marco Cerliani in Towards Data Science

How to Improve Recursive Time Series Forecasting

Simple yet Effective Evolutions of Recursive Method without the need for Deep Learning

Jul 20, 2022 70

See all from Marco Cerliani

See all from Towards Data Science

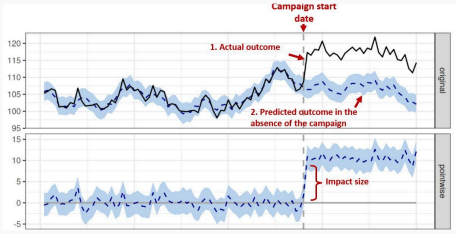
Recommended from Medium

Aparna Dhinakaran in Towards Data Science

Choosing Between LLM Agent Frameworks

The tradeoffs between building bespoke code-based agents and the major agent...

6d ago 1.98K 20



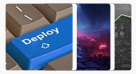
Robson Tigre

When and how to apply causal inference in time series

Intuition, step-by-step script, and limitations of the CausalImpact and CausalArima...

Sep 18 89 2

Lists



Predictive Modeling w/ Python

20 stories · 1554 saves

Practical Guides to Machine Learning

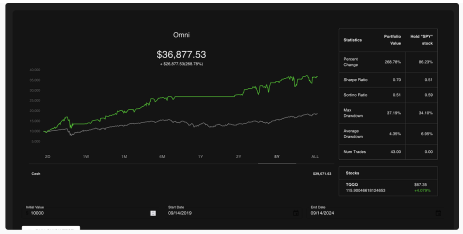
10 stories · 1886 saves

Natural Language Processing

1720 stories · 1296 saves

data science and AI

40 stories · 248 saves

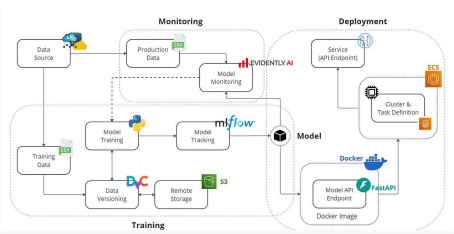


Austin Starks in DataDrivenInvestor

I used OpenAI's o1 model to develop a trading strategy. It is...

It literally took one try. I was shocked.

Sep 15 2.1K 65



Prasad Mahamulkar

Machine Learning Operations (MLOps) For Beginners

End-to-end Project Implementation

Aug 29 1K 6

Thomas Reid in AI Advances

DuckDB Version 1.1.0 is out now.

All the good stuff from the latest release

Sep 18 308 1

Bhavik Patel in Product Coalition

Analysing Tech Layoffs: Which Roles Were Hit Hardest?

In the aftermath of tech layoffs, it would appear that engineering was hit hardest, but...

Sep 17 596 13

See more recommendations