Centro de Astropartículas y Física de Altas Energías
Universidad Zaragoza

Università degli Studi di Padova
Dipartimento di Fisica e Astronomia "Galileo Galilei"

Machine Learning for global fits

4th COMCHA School
14-04-2026

Jorge Alda jorge.aldagallo@unipd.it
Università degli Studi di Padova & CAPA

Resources for this tutorial

- Lecture Notes: arXiv:2604.07520 [hep-ph]
- Code repository: Jorge-Alda/comcha_tutorial

```
git clone https://github.com/Jorge-Alda/comcha_tutorial.git
cd comcha_tutorial
conda env create --file=environment.yml
conda activate comcha-tutorial-globalfits
python -m ipykernel install --user --name comcha-tutorial-globalfits
jupyter lab
```

- Run in Binder
- Run in Google Colab (requires Google account and copying to your Drive).
- Training dataset

Contents

- The likelihood function for global fits
- Active learning to generate training datasets
- Regression trees
- Explainable models
- Sampling the posterior distribution
- Application: the $B^{\pm} \to K^{\pm}\nu\bar{\nu}$ excess at Belle II

The likelihood function

Global fits

We want to build models that accurately describe nature.

Models produce quantitative predictions for observables given parameters $\theta \in \Theta$.

Global fits aim to find a consistent set of parameters that describes simultaneously all the experimental data.

"All models are wrong, but some are useful"
– George Box

"Without data, you're just another person with an opinion"
– W. Edwards Deming

Likelihood function

The likelihood function is the main statistic quantity of interest for global fits:

$$\mathscr{L}(\theta) = p(\text{data}|\theta)\,, \qquad \chi^2(\theta) \equiv -2\log\mathscr{L}(\theta)\,.$$

Example: Gaussian uncorrelated observables

$$\mathscr{L}(\theta) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - x_i(\theta))^2}{2\sigma_i^2}\right), \qquad \chi^2(\theta) = \sum_{i=1}^{N} \frac{(y_i - x_i(\theta))^2}{\sigma_i^2}\,.$$

- Parameter estimation: Maximum Likelihood Estimators (MLE):

$$\hat{\theta} = \arg\max \mathscr{L}(\theta) = \arg\min \chi^2(\theta)\,.$$

- Hypothesis testing: Wilks' theorem:

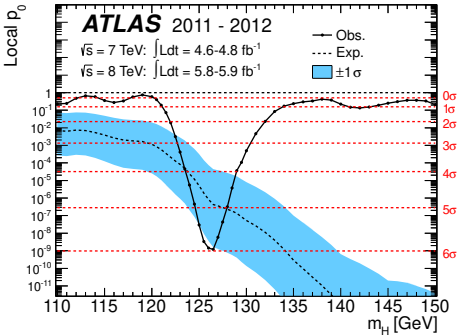
$$\lambda = -2\log\frac{\mathscr{L}(\hat{\theta}_0)}{\mathscr{L}(\hat{\theta}_1)} = \chi^2(\hat{\theta}_0) - \chi^2(\hat{\theta}_1)$$

  p-values and significance (σ) for exclusion or discovery:

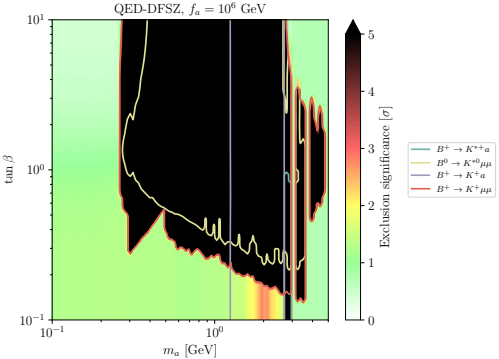
$$p = \int_{\lambda}^{\infty} f_{\chi^2}(t;\nu)\; dt\,, \qquad Z = \Phi^{-1}(1-p)\,.$$

Hypothesis testing

Discovery (e.g. Higgs boson)



Exclusion (e.g. axion models)



Likelihood function: Bayes' theorem

$$p(\theta|\mathrm{data}) \propto p(\mathrm{data}|\theta)p(\theta) = \mathscr{L}(\theta)p(\theta)\,.$$

- Proportionality constant ($p(\mathrm{data})$) is usually irrelevant.
- $p(\theta)$ is the prior probability distribution. It contains our knowledge (or ignorance) of the parameters before confronting with experimental data:
  - Uninformative priors (e.g. uniform distribution in the allowed range).
  - Informative priors (e.g. Gaussian distribution with expected value and uncertainty).
- $p(\theta|\mathrm{data})$ is the posterior probability distribution, and contains our updated knowledge after confronting with the data.

After obtaining the posterior, the expected value of any quantity $f(\theta)$ is

$$\langle f(\theta)\rangle = \int f(\theta)p(\theta|\mathrm{data})\,d\theta\,.$$

Challenges

In many real-world cases, the likelihood is hard to compute...

- Many uncorrelated non-Gaussian observables.
- Predictions are very costly to compute.

...and its applications (MLE, posterior sampling) require lots of evaluations!

Idea: Train a Machine Learning surrogate model to approximate the log-likelihood function by a more efficient function.

Active Learning

Active Learning

The first problem is generating samples from the likelihood function in an efficient way to train the surrogate;

Active Learning: A ML algorithm decides iteratively which are the points more interesting to grow the dataset

- Exploitation: Add points with largest $\mathscr{L}$ (according to current knowledge).
- Exploration: Add points whose prediction have large uncertainty (gain knowledge).

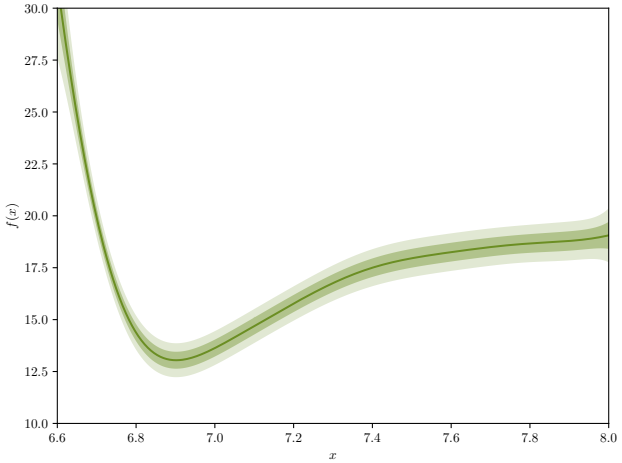
Gaussian process

ML with estimation of uncertainty

Limit of multivariate Gaussian in infinite variables

Random function distributed as

$$f(x) \sim \mathcal{GP}(\mu(x), k(x, x')) \, .$$



Using gpflow

```
import gpflow

gp = gpflow.models.GPR(
    data=(X, y[:, None]),
    kernel=gpflow.kernels.SquaredExponential()
)
opt = gpflow.optimizers.Scipy()
opt.minimize(gp.training_loss, gp.trainable_variables)

central, sigma = gp.predict_y(X_new)
central = central.numpy().flatten()
sigma = sigma.numpy().flatten()
```

Expected Improvement

If the current best point is x^*, the prediction improvement for x is

$$I(x) = \frac{s(f(x^*) - \mu(x)) - \delta}{\sigma(x)},$$

δ controls exploitation-exploration trade-off ($\delta = 0$ exploitation only, large δ increases exploration).

Expected Improvement

$$EI(x) = \sigma(x)[I(x)\Phi(I(x)) + \varphi(I(x))],$$

Φ and φ are CDF and PDF of Gaussian distribution.

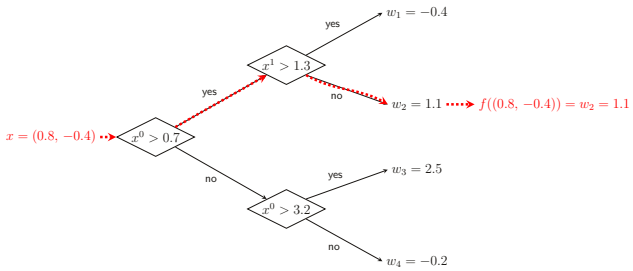
Active learning with EI

1. Start with a small sample of random points (e.g. Latin hypercube sampling), and relatively large δ to ensure proper exploration.
2. At each step, train the GP.
3. Generate candidate points, and calculate their EI.
4. For the point with largest EI, compute its log-likelihood, and add to the sample.
5. Gradually decrease δ to increase exploitation.

Regression trees

Regression trees

A regression tree partitions the parameter space in disjoint regions, and assigns a value ("leaf") to each one.



Individual trees are weak learners, but an ensemble of regression trees can learn any arbitrary function.

Boosted regression trees: XGBoost

The prediction is computed as the sum of contributions from each tree in the ensemble.

The splittings and leaves are learnt via supervised learning (we have to provide a dataset with values of the parameters and their corresponding log-likelihood).

Learning proceeds by minimizing an objective function, composed by loss function (how well does the model agree with training data) and regularization (penalizes more complex trees).

Each tree is trained sequentially: the n-th tree learns the difference between the training data and the prediction of the ensemble of $n-1$ trees.

- Early stopping: The training process is stopped when adding new trees no longer improves certain metric in the validation dataset: reduces overfitting.

XGBoost hyperparameters

- **Number of estimators:** Number of trees contained in the ensemble. Typically of order of a few hundreds.
- **Maximum depth:** Maximum number of levels that a tree can have. Large values are more likely to overfit. Typical values are between 5 and 10.
- **Learning rate (η):** Controls the weight of each tree in the ensemble. Small values reduce overfitting. Typical values are between 0.01 and 0.3.
- **Minimum split loss (γ):** Minimum loss reduction needed to create a partition on a leaf node. Larger values lead to more conservative algorithms.
- **Regularization parameters (α and λ):** Penalize large weights, again preventing overfitting.
- **Sampling parameters (subsample, colsample_bytree, colsample_bylevel, colsample_bynode):** Proportion of the training data used to train. Subsample controls the proportion of rows used, while the colsample parameters controls the proportion of columns.

Using xgboost(I): Loading the data

```
import xgboost as xgb
import pandas as pd
from sklearn.model_selection import train_test_split

# Load data
df = pd.read_csv("train_data.csv")
pars = ["p1", "p2", "p3"]
X = df[pars]
y = df["y"]

# Create training and validation datasets
X_tr, X_v, y_tr, y_v = train_test_split(X, y, test_size=0.2)
d_tr = xgb.DMatrix(X_tr.values, label=y_tr.values)
d_v = xgb.DMatrix(X_v.values, label=y_v.values)
```

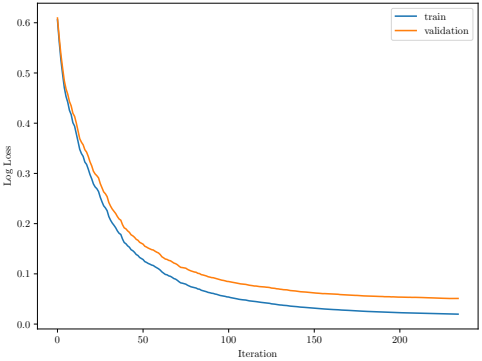
Using xgboost(II): Training

```
# Train the model
xgb_params = {
    'objective': 'reg:squarederror',
    'eval_metric': ['rmse', 'mae'],
    'max_depth': 6,
    'learning_rate': 0.1,
    'gamma': 0,
    'nthread': -1, # Use all available CPU cores
}
model_regr = xgb.train(
    xgb_params, d_tr,
    num_boost_round=5000,
    evals=[(d_tr, 'train'), (d_v, 'eval')],
    early_stopping_rounds=5,
    verbose_eval=False
)
# Wrap prediction in a friendlier format
predict_point = lambda x: model_regr.predict(xgb.DMatrix([x,]))
```

Using xgboost (III): Visualizing the overfitting

```
import matplotlib.pyplot as plt

eval_results = {}
model_regr = xgb.train(
    xgb_params, d_tr, num_boost_round=5000,
    evals=[(d_tr, 'train'), (d_v, 'eval')],
    evals_result=eval_results
    early_stopping_rounds=5,
    verbose_eval=False
)
for metric in ['rmse', 'mae']:
    plt.plot(eval_results['train'][metric], label='train')
    plt.plot(eval_results['eval'][metric], label='validation')
    plt.xlabel('Iteration')
    plt.ylabel(metric)
    plt.legend()
    plt.show()
```

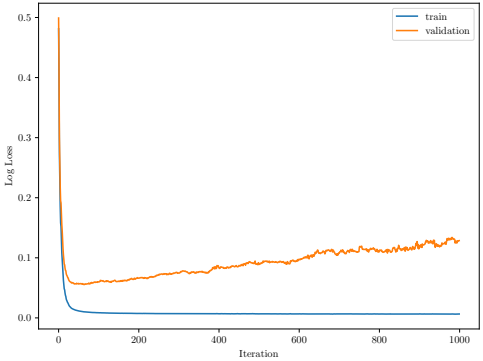
Overfitting



✓ No overfitting

Metric in validation improves like in training dataset

Model is generalizing



✗ Overfitting

Metric improves only in training dataset, not in validation

Model is learning "by heart" its training

Tuning the hyperparameters

Choosing the right hyperparameters is fundamental to achieve a good convergence of the model while avoiding overfitting.

Common strategies:

- Cross-validation (CV): The dataset is divided into several subsets. For each candidate choice of hyperparameters, some subsets are used for training and others for validation (k-fold CV, stratified CV, leave-one-out CV,...).
  For example, implemented by `sklearn.model_selection.GridSearchCV`
- Bayesian optimization, implemented by `optuna`.

Hyperparameter tuning with optuna

```python
import optuna
from sklearn.metrics import mean_absolute_error
def optuna_obj(trial):
    optuna_params = {
        'reg_alpha': trial.suggest_float('reg_alpha', 0, 1.2),
        'reg_lambda': trial.suggest_float('reg_lambda', 0, 1.2),
    }
    params = xgb_params | optuna_params
    model_regr = xgb.train(
        params, d_tr, num_boost_round=5000,
        evals=[(d_tr, 'train'), (d_v, 'eval')],
        early_stopping_rounds=5,
    )
    predictions = model_regr.predict(d_v)
    return mean_absolute_error(y_v, predictions)
study = optuna.create_study(direction='minimize')
study.optimize(optuna_obj, n_trials=100, n_jobs=-1)
xgb_params |= study.best_params
```

Storing and compiling xgboost models

xgboost can save and load models in two formats: JSON (text) and UBJSON (binary). Binary format results in smaller files and less processing time.
Additionally, we can compile the model into a C library. Great speed-up in evaluation!

```
import tl2cgen
import treelite.frontend
# Save to UBJSON
model_regr.save_model('xgb_regressor.ubj')
# Compile to C library
tree_reg = treelite.frontend.from_xgboost(model_regr)
tl2cgen.export_lib(
    tree_reg,
    toolchain='gcc', libpath='xgb_regressor.so', params={'parallel_comp': 4}
)
```

⚠ Toolchain and the file extension of the library depend on your OS!

Explainable models

Explaining the surrogate model

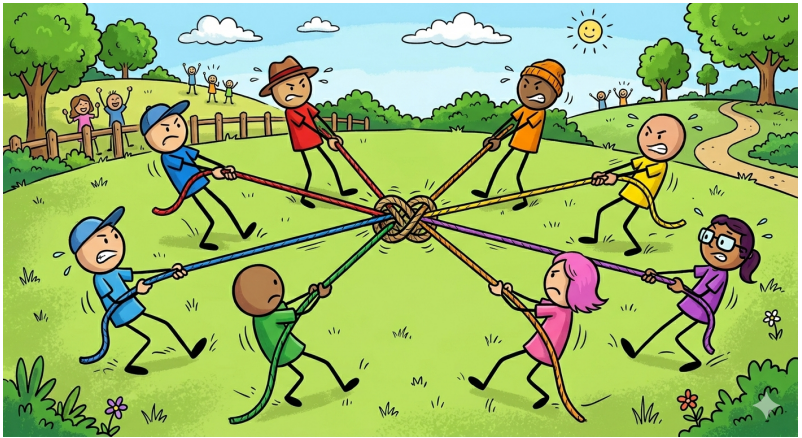
SHAP values gauge the importance of each parameter in the predictions of the ML surrogate.

The original concept, Shapley values, comes from cooperative game theory, and is the answer to the question "How much did every player contribute to the final outcome?" (or "how should the reward be fairly distributed amongst the players?")

SHAP values offer local explanations for each point of parameter space.

- Global explanations: average importance, SHAP dependence.

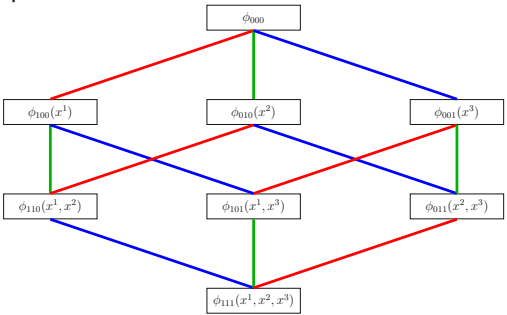
SHAP values: an analogy



Each parameter is a player of the tug-of-war game. SHAP values measure the strength of each player, and the prediction is the sum of all of them.

Computation of SHAP values

If the original model has k parameters, then train 2^k models with all possible "coalitions" of parameters:



The marginal contribution of the parameter x^1 is the difference between models trained on x^1 and models not trained on that parameter: connected by red edges.

SHAP values are the weighted average of marginal contributions.

For generic ML models, it is very costly to train 2^k models even for moderate k.

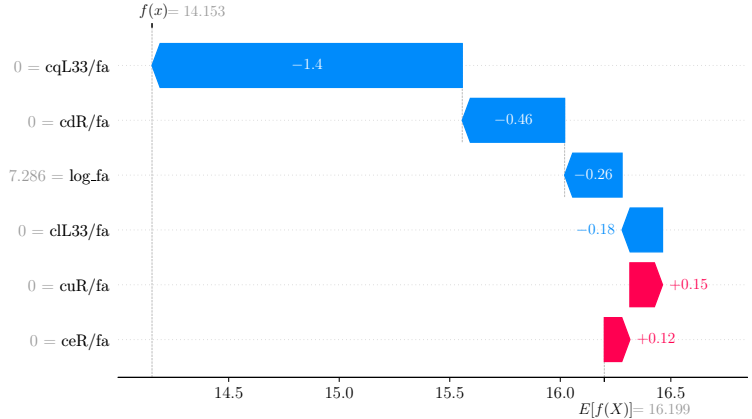
For tree-based models, very efficient approximation TreeSHAP, based only on the topology of each tree: no need to train additional models.

Using xgboost

```
import shap

explainer = shap.TreeExplainer(model_regr)

# Local explanation of point x0
print(explainer(x0))
shap.waterfall_plot(explainer(x0)[0])

# Global explanation of the dataset
shap_values = explainer.shap_values(dataset)
shap.summary_plot(shap_values, dataset, plot_type='bar')
shap.dependence_plot(parameter, shap_values, dataset)
```

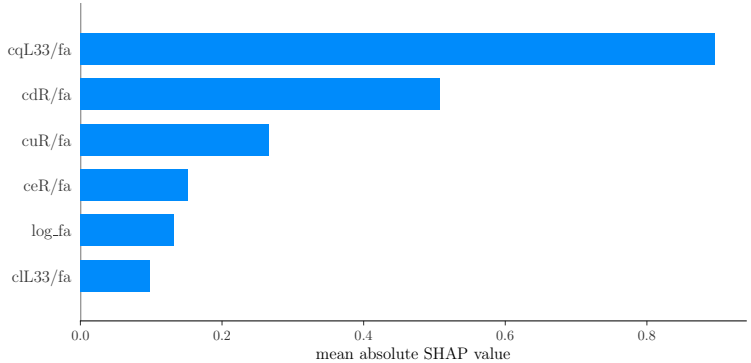
Waterfall plots

Importance of each parameter to explain the prediction of a single point



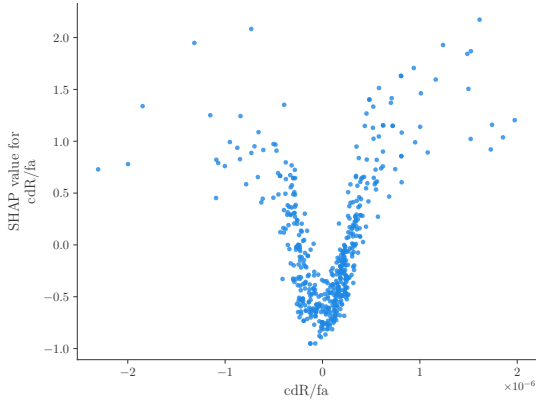
Summary plot

Average importance of each feature for global explanations



Dependence plot

Scatter plot of parameter vs SHAP value



Where the points are vertically concentrated: functional dependence of the parameter.
Where the points are spread: Interaction with other parameters.

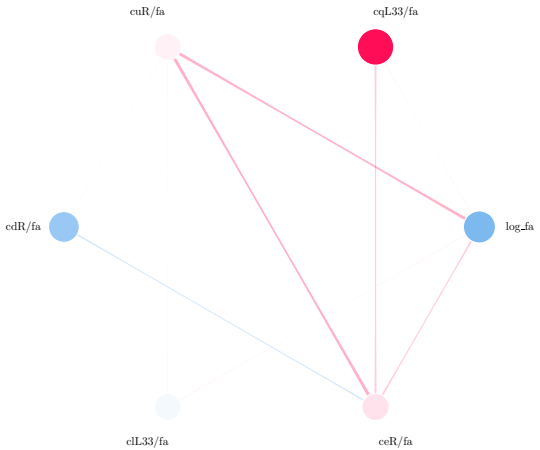
Interactions between parameters

shapiq generalizes SHAP values to capture also the importance of pairs, triplets, etc of parameters.

```
import shapiq

explainer_iq = shapiq.TreeExplainer(model_regr, max_order=2)
inter_values = explainer_iq.explain(x0)
print(inter_values)
shapiq.network_plot(inter_values)
```

Network plot

Explanation of the prediction of a single point using first and second order SHAP values



Sampling the posterior

Sampling from the posterior

The goal is to obtain a sample of points in parameter space following the probability distribution

$$p(\theta|\text{data}) \propto \mathscr{L}(\theta)p(\theta)\,.$$

For uninformative priors, uniform distribution might run into numerically stability issues, specially in narrow regions. As an alternative, use softer boundaries, e.g. sigmoids

$$\log p(\theta) = -\log\left(1+\exp\frac{a-\theta}{w}\right) - \log\left(1+\exp\frac{\theta-b}{w}\right)\,.$$

Markov chain Monte Carlo

The MCMC algorithm produces a chain of parameter points $\theta_1 \to \theta_2 \to \ldots \to \theta_n$.

The probability to go from θ_k to θ_{k+1} depends only on θ_k, and not on the previous points in the chain (Markov process).

A candidate new point in the chain is accepted or rejected with certain probability, so that the chain asymptotically reproduces the posterior distribution.

Drawback: the points in the chain are correlated: reduced number of "effective" samples (autocorrelation time τ).

Affine-Invariant ensemble MCMC

Algorithm implemented by `emcee`

- Pros: works for non-differentiable log-posteriors, not affected by scale of parameters.
- Cons: efficiency decreases with dimensionality of parameter space d.

The algorithm uses an ensemble of chains that are grown simultaneously. The proposal for the $(k+1)$-th point of the chain p is determined from the position of the chains p and q at the same step:

$$\theta^p_{k+1} = \theta^p_k + \gamma(\theta^q_k - \theta^p_k)\,.$$

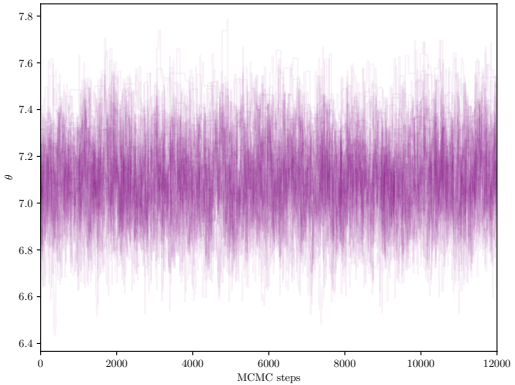
with stretch factor γ distributed as

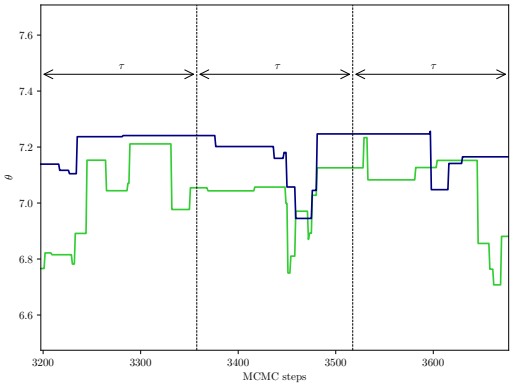
$$p(\gamma) = \begin{cases} \gamma^{-1/2} & a^{-1} \leq \gamma \leq a, \\ 0 & \text{otherwise.} \end{cases}$$

Accepted with probability

$$p_T(\theta^p_{k+1}|\theta^p_k) = \min\left[1, \gamma^{d-1}\frac{p(\theta^p_{k+1}|\text{data})}{p(\theta^p_k|\text{data})}\right]\,.$$

MCMC chains and autocorrelation





Using emcee

```python
import emcee

n_walkers = 20
def log_posterior(x):
    # chi2 and log_priors defined elsewhere
    return -0.5 * chi2(x) + log_priors(x)

sampler = emcee.EnsembleSampler(n_walkers, n_pars, log_posterior)

_ = sampler.run_mcmc(
    initial_guess, # list of n_walker guesses, close to minimum
    n_mcmc_steps
)

print(sampler.acceptance_fraction) # Fraction of points accepted
print(sampler.get_autocorr_time()) # Autocorrelation time for each parameter
chain = sampler.get_chain(flat=True)
```

Inference with MCMC

Since the points of the chain are distributed according to the posterior, the expected value of any function of the parameter points is given by

$$\langle f(\theta) \rangle = \frac{1}{N_{\text{chain}}} \sum_{\theta \in \text{chain}} f(\theta)\,.$$

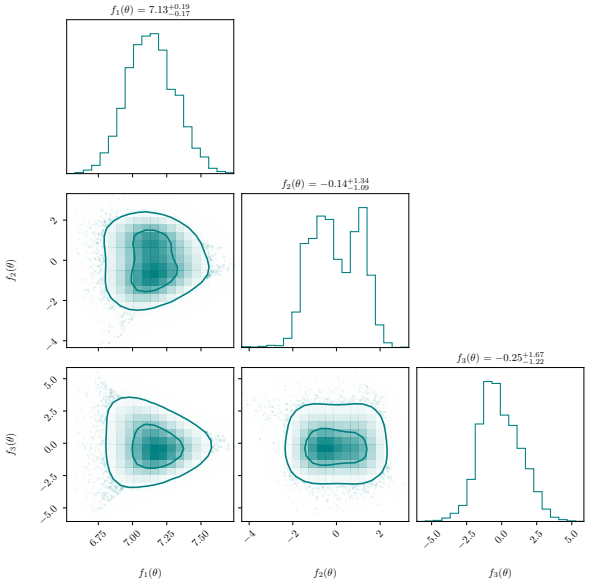
The 1-D and 2-D posterior distribution of the parameters or functions of parameters are usually presented in triangle/corner plots, for example using the library `corner`.

Using chains and corner plots

```
mean_x0 = np.mean(chain[:,0])
variance_x0 = np.mean((chain[:,0] - mean_x0)**2)
```

```
import corner

f1_chain = [f1(x) for x in chain]
f2_chain = [f2(x) for x in chain]
sampled_values = np.vstack([f1_chain, f2_chain]).T

_ = corner.corner(
    sampled_values,
    levels = [0.393, 0.864], # 1sigma and 2sigma in 2D
    smooth = 1.5,
    show_titles = True
)
```

Corner plots



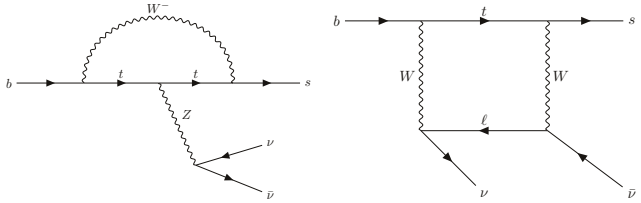
Other architectures

- Neural networks instead of regression trees: Differentiable surrogates, more flexible, but more sensible to hyperparameter tuning and difficult to interpret.
- Differentiable likelihoods (JAX, PyTorch).
- Normalizing flows: learn directly the posterior distribution.
- Hamiltonian MCMC (NUTS): more scalable to higher dimensionality, but requires gradients of likelihood.

Application to a real-world problem

Application: the $B^{\pm} \to K^{\pm}\nu\bar{\nu}$ anomaly at Belle II

In the SM, hadron decays where the initial and final hadrons differ only on one quark of the same electric charge (FCNC decays) are predicted to be very rare. New Physics contribution are potentially large.



The experiment Belle II found a $2.7\,\sigma$ excess in the $B^{\pm} \to K^{\pm}\nu\bar{\nu}$ decay compared to the SM prediction.

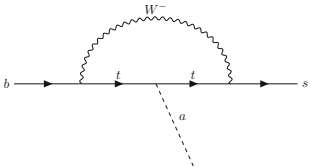
Axion-like particles

ALPs are hypothetical pseudoscalar light particles. Interaction with SM fermions:

$$\mathcal{L} = \frac{\partial_\mu a}{f_a}\left[c^{ij}_{q_L}\,\bar{q}_i\gamma^\mu q_j + c^{ij}_{u_R}\,\bar{u}_i\gamma^\mu u_j + c^{ij}_{d_R}\,\bar{d}_i\gamma^\mu d_j + c^{ij}_{\ell_R}\,\bar{\ell}_i\gamma^\mu \ell_j + c^{ij}_{e_R}\,\bar{e}_i\gamma^\mu e_j\right].$$

ALP decays into neutrinos are extremely suppressed. Instead, a long-lived ALP that escapes the Belle II detector would mimic the experimental signal (missing energy and momentum).

The excess is compatible with $m_a = 2\,\mathrm{GeV}$ and $c\tau_a \geq 80\,\mathrm{cm}$.

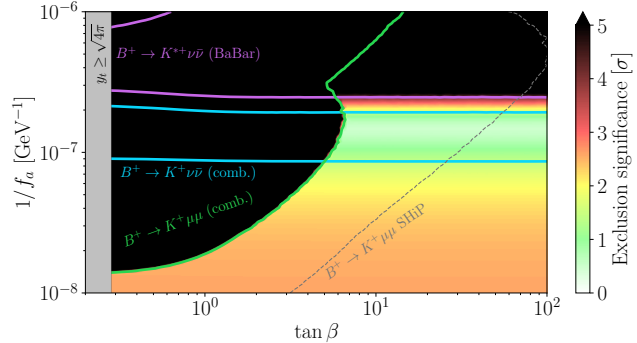


FCNC decays into ALP are mainly generated by top loops.

Problem: find values for couplings and f_a such that $B^\pm \to K^\pm a$ is large enough, but $c\tau_a$ is not too small.

ALP model for the excess

We found a non-universal DFSZ-like ALP model compatible with the data



- $c_{q_L}^{33}$, $c_{\ell_L}^{33}$ (3rd gen. couplings), c_{u_R}, c_{d_R}, c_{e_R} (universal couplings). All of them $\in [-1, 1]$.
- $f_a \approx 10^7\,\mathrm{GeV}$. Controls the size of loop effects.

For the tutorial, we'll allow these parameters to vary independently in the ranges $c_f \in [-3, 3]$ and $\log_{10} f_a \in [6, 8]$.

ALP-aca

ALP-aca: The ALP Automatic Computing Algorithm. alp-aca/alp-aca

```
pip install alpaca-alps==1.2rc1
```

- RGE evolution of couplings
- Calculation of observables and probability of decaying inside/outside/displaced vertex
- χ^2 scans
- Beautiful publication-ready plots

Some tasks for the tutorial

- Try different parameters of the hyperparameters when training the surrogate. Check which choices increase or decrease overfitting and convergence.
- Select subsets of the training data with low and moderate values of χ^2, and calculate SHAP values in them. How does the importance of each parameter change between subsets? Can you find a physical explanation?
- Identify which parameter is less important to define the surrogate model. Train a simplified surrogate model omitting this parameter, and compare the results between both of them.
- Compute expected values for other quantities, for example $c_t = c_{u_R} - c_{q_L}^{33}$, $c_{\mu\mu}(m_a)$, $c_{bs}(m_a)$.