Data Mining 2

Topic 04 : Hyperparameter Tuning

Lecture 02 : Hyperparameter Tuning

Dr Kieran Murphy

Department of Computing and Mathematics, WIT.
(kmurphy@wit.ie)

Spring Semester, 2022

Outline

- Standard Techniques - Grid vs Random Search
- Bayesian optimisation - hyperopt and Scikit-Optimize

Recap of where we are (were from previous lecture)

- Given a pipeline / model we "can" tune each of the hyperparameters by constructing a validation curve.

- Problems . . . (manual approach is not practical anything but small problems)
  - Can have a huge (100s – 1,000s) number of parameters.
  - A sequence of 1-dimensional searches is not the same as one d-dimensional search — interplay between hyper-parameters.

- Approaches / Techniques . . .
  - Grid Search — Systematic, regular, predetermined deterministic sample of parameter space.
  - Random Search — Non-adaptive, random sample of parameter space.
 - Bayesian Search — Adaptive, random sample of parameter space.

Grid Search

Validation curves are important but examining at each parameter individually is time consuming and does not take into account interactions between parameters — in effect it is a series of 1D semi-manual searches.
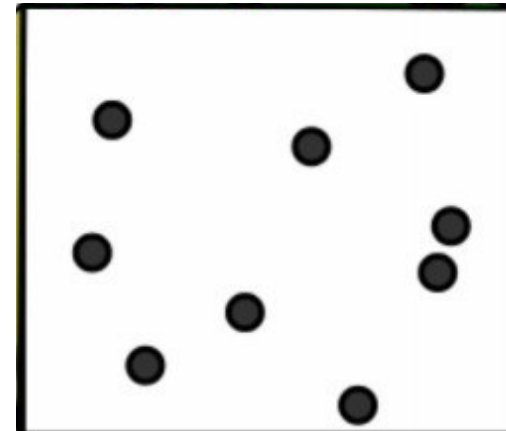
> ### Grid Search
> Try all available parameters combinations, 1 by 1, and choose the one with the best cross validation results.
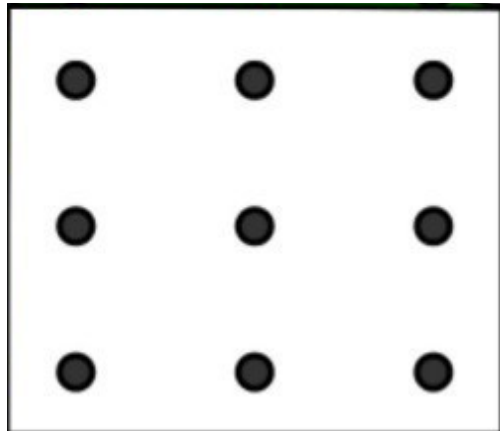
Grid Search   Random Search

Grid Search (`sklearn.model_selection.GridSearchCV`)

> ### Grid Search
> Try all available parameters combinations, 1 by 1, and choose the one with the best cross validation results.

Drawbacks

- Still very slow — trying ALL combinations of ALL parameters with no modification of search based on results found to date.

- ALL combinations $\implies$ every additional hyperparameter to vary multiplies the number of iterations you need to complete.

- It can work only with discrete values.

  If the global optimum is on `n_estimators=550`, but you are doing `GridSearchCV` from 100 to 1000 with step 100, you will never reach the optimal point.

Stratagies

- You need know / guess the approximate region of the optimum to start.

- To mitigate against drawbacks, do multiple lower dimensional grid searches, or repeat search with narrower grids and smaller step sizes.

Grid Search Example 1   I

```python
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', SVC(random_state=SEED))])

param_range = np.logspace(-4,4,10)
param_grid = [{
    'clf__gamma': param_range,
    'clf__C': param_range,
    'clf__kernel': ['rbf']
}]
gs = GridSearchCV(estimator=pipeline,
    return_train_score=True,
    param_grid=param_grid, scoring='accuracy', cv=10, n_jobs=-1)
```

- Switched from LogisticRegression to SVM — more hyperparameters to tweak.
- How many parameter combinations were generated/used?

Grid Search Example 1 II

To perform grid search, we call the `fit` method as usual ...

```
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9757971014492753
{'clf__C': 2.782559402207126, 'clf__gamma': 0.046415888336127774, 'clf__kernel': 'rbf'}
```

- CV score improved from 0.971 with default parameter values to 0.978.

- GridSearchCV with option `return_train_score=True`, returns results of each combination — determine effect of hyperparameters to refine grid search.
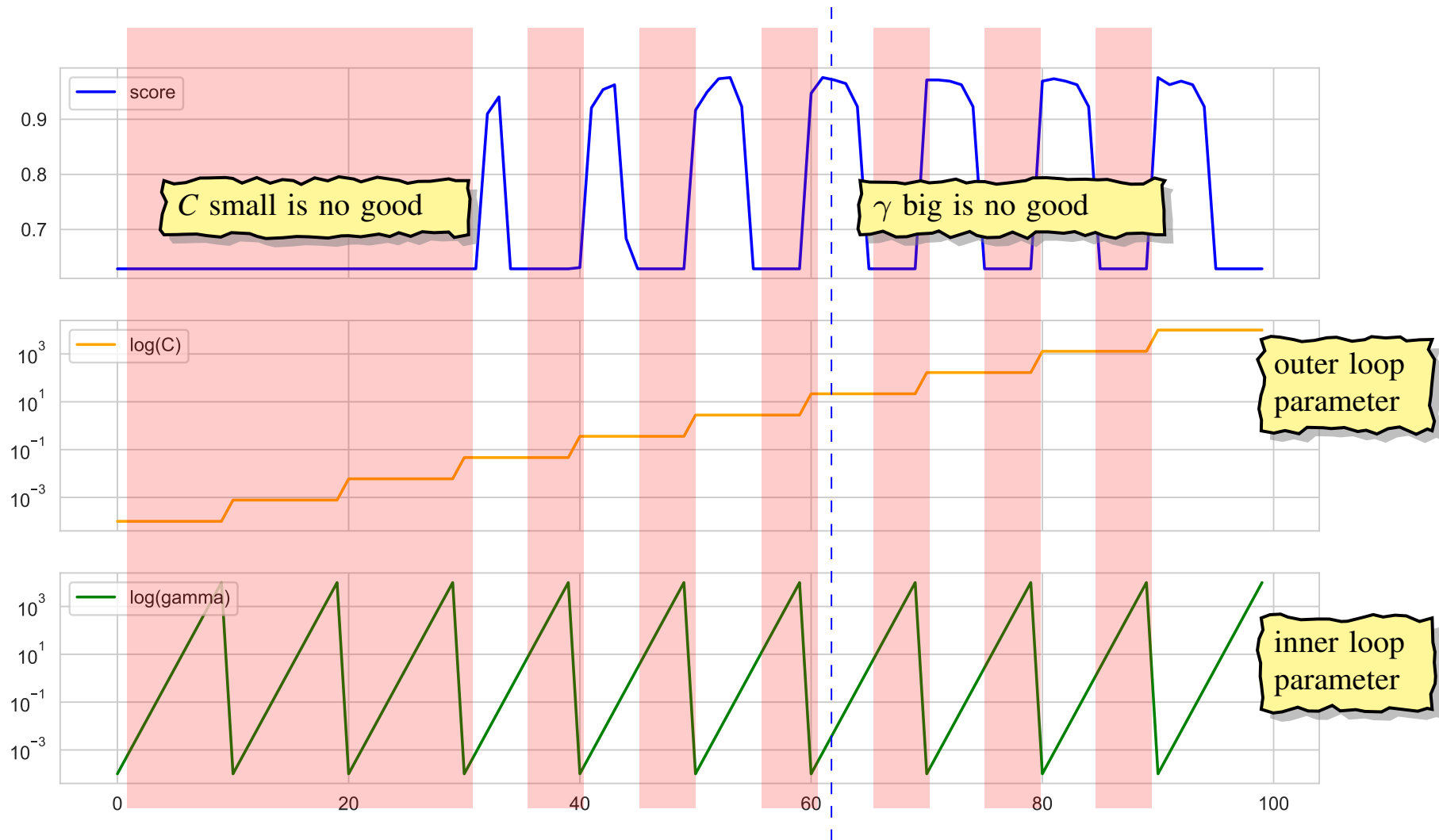
```
df_gs = pd.DataFrame(np.transpose([
    gs.cv_results_["mean_test_score"],
    gs.cv_results_["param_clf__C"].data,
    gs.cv_results_["param_clf__gamma"].data]),
    columns=['score', 'C', 'gamma'])

df_gs.plot(subplots=True,figsize=(12, 8))
plt.savefig("gs__svm__C_gamma.pdf", bbox_inches="tight")
plt.show()
```

Grid Search Example 1 ($C = 10$, $\gamma = 0.01$) III
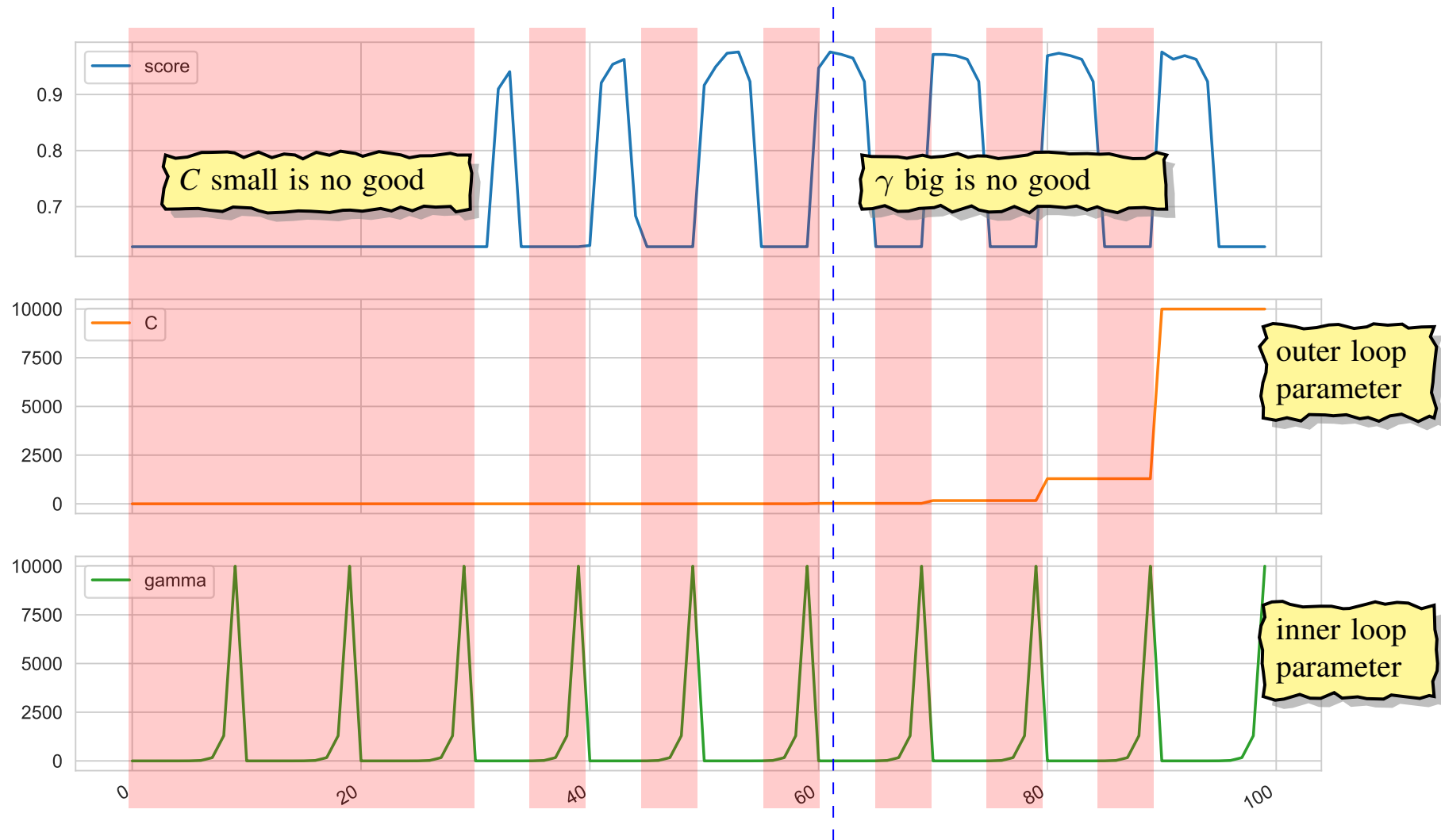
Grid Search Example 1 ($C = 10$, $\gamma = 0.01$) (Using log scales) III

Grid Search Example 2

GridSearch can accept a list of dictionaries so that incompatible hyperparameter combinations can still be searched. For example, different SVM kernels have different parameters:

```python
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', SVC(random_state=SEED))
])

from sklearn.model_selection import GridSearchCV
param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
param_grid = [
    {
        'clf__C': param_range,
        'clf__kernel': ['linear']
    },
    {
        'clf__gamma': param_range,
        'clf__C': param_range,
        'clf__kernel': ['rbf']
    }
]
```

```
0.9780676328502415
{'clf__C': 545.5594781168514, 'clf__gamma': 0.0001}
```
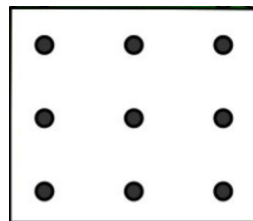
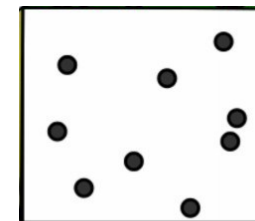- How many hyperparameter combinations are now generated?

Random Search

> ## Random Search
>
> Try random hyperparameter combinations and choose the one with the best cross validation results.

<div align="center">

Grid Search   Random Search

</div>

Advantages

- On every step random search varies all parameters $\Rightarrow$ doesn't spend time on meaningless parameters.
- On average finds near optimal parameters much faster than Grid search.
- It is not limited by grid when optimising continuous parameters.

Disadvantages

- It may not find the global optimal parameter on a grid.
- All steps are independent. Does not use information about the results gathered to inform search.

Random Search I

```
pipeline = Pipeline([
    ('scl', StandardScaler()),
    ('clf', SVC(kernel='rbf', random_state=SEED))
])

from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_range = np.logspace(-4,4,20)

param_grid = {'clf__C': param_range,
              'clf__gamma': param_range}

rs = RandomizedSearchCV(estimator=pipeline,
    param_distributions=param_grid,
    n_iter = 100, random_state=SEED,
    return_train_score=True,
    scoring='accuracy', cv=10, n_jobs=-1)
```

- This is the same setup as used in GridSearch but doubled the number of parameter values in both C and γ, so search space is 400 points (not 100). Sampling 100 points.

Random Search Example   II

To perform grid search, we call the `fit` method as usual ...

6
```python
rs.fit(X_train, y_train)
print(rs.best_score_)
print(rs.best_params_)
```

```
0.9780676328502415
{'clf__gamma': 0.0001, 'clf__C': 545.5594781168514}
```

- Sampling half the number of points (50) in a search space 4 times larger, RandomSearch outperforms GridSearch.

7
```python
df_rs = pd.DataFrame(np.transpose([
    rs.cv_results_["mean_test_score"],
    rs.cv_results_["param_clf__C"].data,
    rs.cv_results_["param_clf__gamma"].data]),
    columns=['score', 'C', 'gamma'])

df_rs.plot(subplots=True,figsize=(12, 8))
plt.savefig("rs__example_1.pdf", bbox_inches="tight")
plt.show()
```

Random Search Example ($C = 10000, \gamma = 0.0002636$)   III
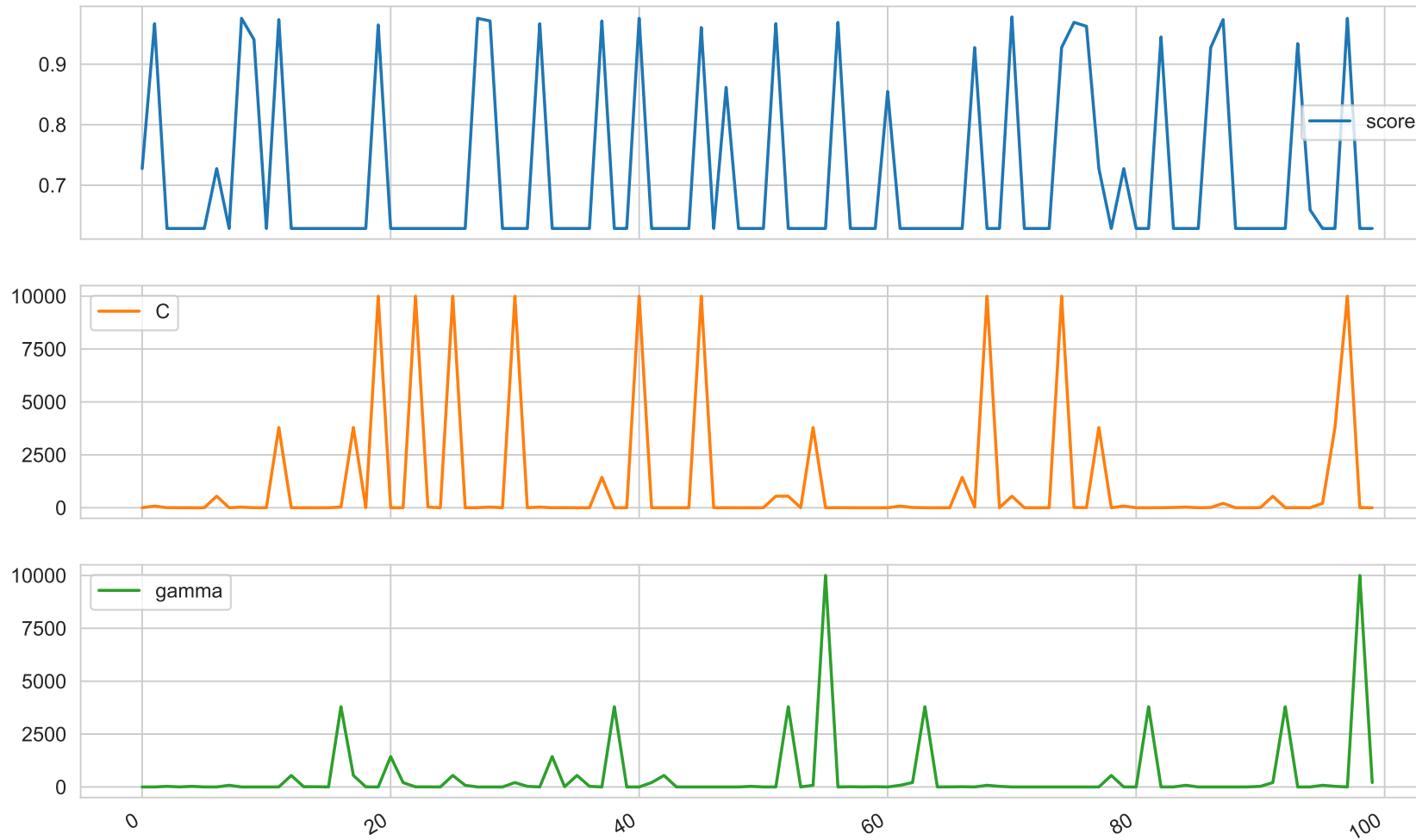
Every step is completely random. $\Rightarrow$ does not spent time on useless parameters, but does not use the information gathered on the first steps to improve outcomes of the latter ones.

The Problem

> Both GridSearch and RandomSearch ignore information gained during the search, by using such information can searches improve?

- What information would be useful?
  - Function value
  - Function derivative (gradient) — gives direction in which the function is decreasing (locally).
- Why should we do this?
  - It seems reasonable that lower function values would be clustered — so focus search near best function values found to date.
  - Making multiple decisions based on local information could locate global minimum.
- What can go wrong?
  - Multiple local minimum — can get stuck in a side valley.
    - There will be situations in which we should make decisions against current function value information — see Simulated Annealing.
  - Gradient could be flat (so no direction to go in), or worse almost flat (some algorithms become unstable), or non-existent.

Bayesian Optimisation

> **Sequential model-based optimisation (SMBO)**
> (also known as Bayesian optimisation) is a general technique for function optimisation that uses information from earlier function evaluations to refine the optimisation search — with the aim of minimising number of function calls.

- For sufficiently complex functions, SMBO are some of the most call-efficient (in terms of function evaluations) optimisation methods currently available.

- Compared with standard optimisation strategies such as conjugate gradient descent methods, model-based optimisation algorithms invest more time between function evaluations in order to reduce the number of function evaluations overall.

- Takes advantage of smoothness without analytic gradient.

- Supports real-valued, discrete, and conditional variables.

- Performs relatively will in high dimensions —- hundreds of variables, even with budget of just a few hundred function evaluations, i.e., number of function calls is $\mathcal{O}(\text{dimensions})$ not $\mathcal{O}(e^{\text{dimensions}})$.

Sequential model-based optimisation (in HyperOpt)
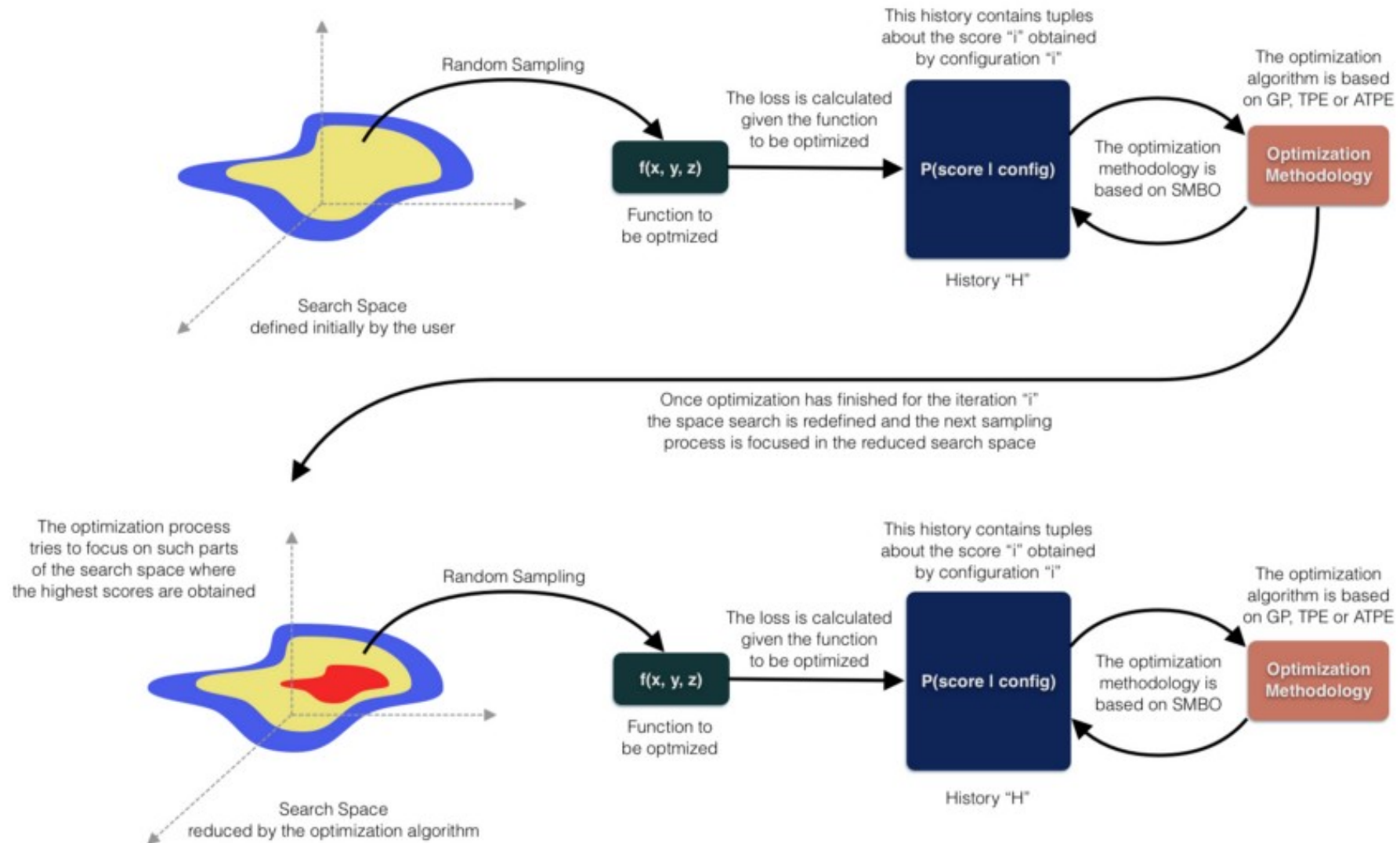
Hyperopt

> Hyperopt: Distributed Asynchronous Hyper-parameter Optimisation

- Python library for optimising over awkward search spaces with real-valued, discrete, and conditional dimensions.

  github.com/hyperopt/hyperopt

- Install (Anaconda) or use pip

  `conda install -cy conda-forge hyperopt`
- Supports parallel evaluation
  - Uses MongoDB to share results of functions evaluations..

> hyperopt-sklearn

- Hyperopt-based model selection algorithms in scikit-learn

  github.com/hyperopt/hyperopt-sklearn

- Install using pip.

  Purpose of hyperopt is not to always to completely optimise the search space but simply to do better than people.

  — J Bergstra, SciPy 2013

Resources

- Tune Hyperparameters for Classification Machine Learning Algorithms
  `machinelearningmastery.com/`
  `hyperparameters-for-classification-machine-learning-algorithms/`
  List of important hyper-parameters for skilearn classifiers

- Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms
  `conference.scipy.org/proceedings/scipy2013/pdfs/bergstra_hyperopt.pdf`
  SCIPY Conference 2013 Paper

- A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning
  `towardsdatascience.com/`
  `a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machin`
  Relatively high level comparison of hyper-parameter tuning approaches.