Lectures 6: Class demo

Contents

- Imports
- Exhaustive grid search: `sklearn.model_selection.GridSearchCV`
- Randomized hyperparameter search
- (Optional) Fancier methods

Imports

```
import os
import sys

sys.path.append(os.path.join(os.path.abspath(".."), (".."), "code"))

import matplotlib.pyplot as plt
import mglearn
import numpy as np
import pandas as pd
from plotting_functions import *
from sklearn.dummy import DummyClassifier
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score, cross_validate, train_tes
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from utils import *

%matplotlib inline
pd.set_option("display.max_colwidth", 200)
DATA_DIR = os.path.join(os.path.abspath(".."), (".."), "data/")
```

```
from sklearn import set_config

set_config(display="diagram")
```

Let's look at an example of tuning `max_depth` of the `DecisionTreeClassifier` on the Spotify dataset.

```
spotify_df = pd.read_csv(DATA_DIR + "spotify.csv", index_col=0)
X_spotify = spotify_df.drop(columns=["target", "artist"])
y_spotify = spotify_df["target"]
X_spotify.head()
```

	acousticness	danceability	duration_ms	energy	instrumentalness	key	live
0	0.0102	0.833	204600	0.434	0.021900	2	0.
1	0.1990	0.743	326933	0.359	0.006110	1	0.
2	0.0344	0.838	185707	0.412	0.000234	2	0.
3	0.6040	0.494	199413	0.338	0.510000	5	0.(
4	0.1800	0.678	392893	0.561	0.512000	5	0.4

```
X_train, X_test, y_train, y_test = train_test_split(
    X_spotify, y_spotify, test_size=0.2, random_state=123
)
```

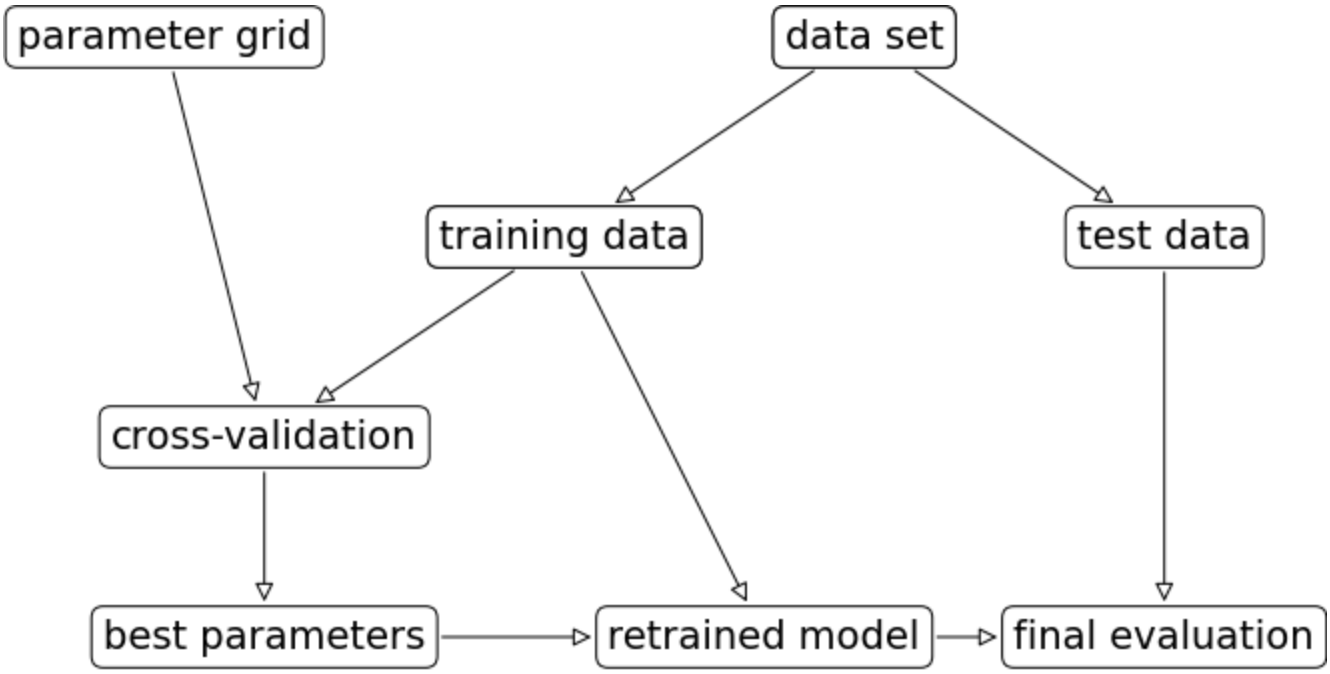
```
numeric_feats = ['acousticness', 'danceability', 'energy',
                 'instrumentalness', 'liveness', 'loudness',
                 'speechiness', 'tempo', 'valence']
categorical_feats = ['time_signature', 'key']
passthrough_feats = ['mode']
text_feat = "song_title"
```

```

```

```
from sklearn.compose import make_column_transformer
from sklearn.feature_extraction.text import CountVectorizer

preprocessor = make_column_transformer(
    (StandardScaler(), numeric_feats),
    (OneHotEncoder(handle_unknown = "ignore"), categorical_feats),
    ("passthrough", passthrough_feats),
    (CountVectorizer(max_features=100, stop_words="english"), text_feat)
)

svc_pipe = make_pipeline(preprocessor, SVC)
```

What's the general approach for model selection?

```
mglearn.plots.plot_grid_search_overview()
```



Hyperparameter optimization is so common that sklearn includes two classes to automate these steps.

- Exhaustive grid search: `sklearn.model_selection.GridSearchCV`
- Randomized search: `sklearn.model_selection.RandomizedSearchCV`

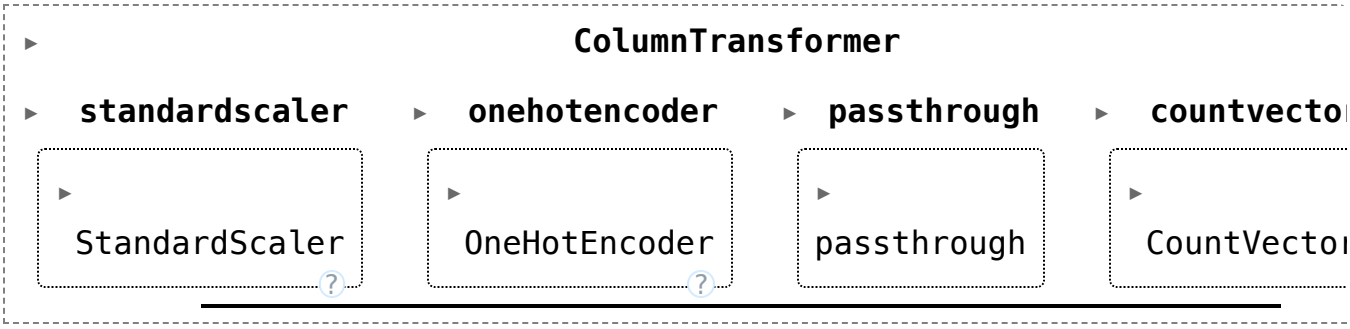
The "CV" stands for cross-validation; these methods have built-in cross-validation.

Exhaustive grid search: `sklearn.model_selection.GridSearchCV`

- For `GridSearchCV` we need
  - an instantiated model or a pipeline
  - a parameter grid: A user specifies a set of values for each hyperparameter.
  - other optional arguments

The method considers product of the sets and evaluates each combination one by one.

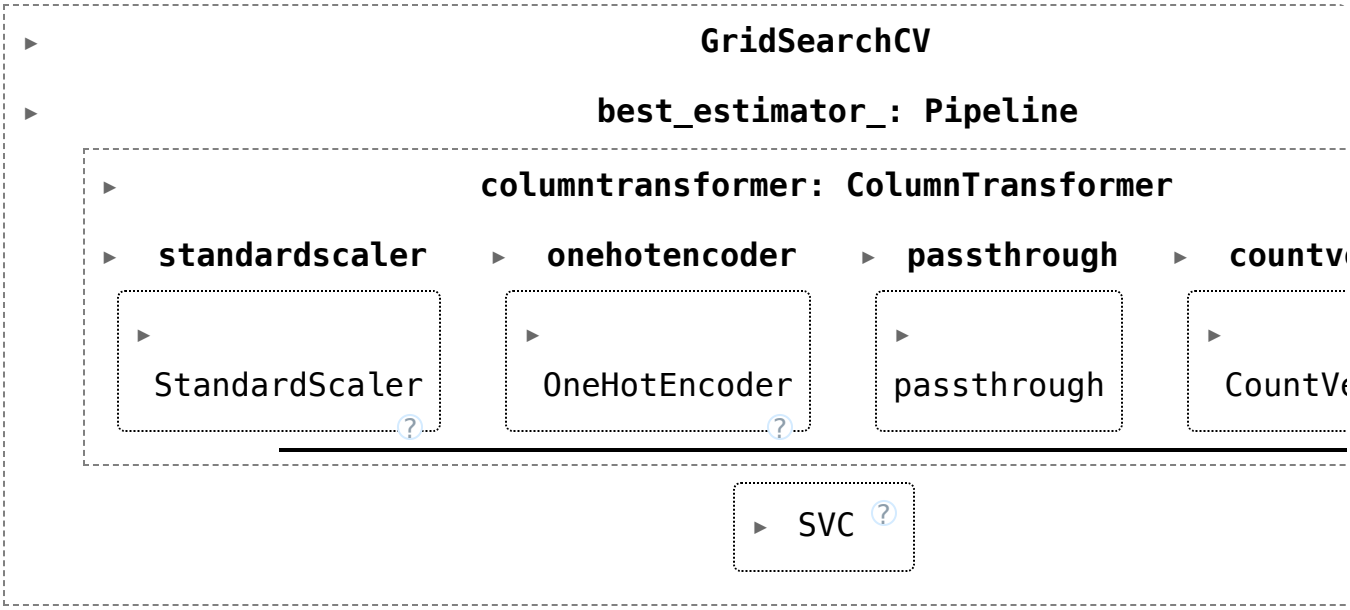
```
preprocessor.fit(X_train)
```



```
from sklearn.model_selection import GridSearchCV

pipe_svm = make_pipeline(preprocessor, SVC())

param_grid = {
    "columntransformer__countvectorizer__max_features": [100, 200, 400, 800, 1
    "svc__gamma": [0.001, 0.01, 0.1, 1.0, 10, 100],
    "svc__C": [0.001, 0.01, 0.1, 1.0, 10, 100],
}

# Create a grid search object
gs = GridSearchCV(pipe_svm,
                  param_grid = param_grid,
                  n_jobs=-1,
                  return_train_score=True
                 )
```

The `GridSearchCV` object above behaves like a classifier. We can call `fit`, `predict` or `score` on it.

```
# Carry out the search
gs.fit(X_train, y_train)
```



Fitting the `GridSearchCV` object

- Searches for the best hyperparameter values
- You can access the best score and the best hyperparameters using `best_score_` and `best_params_` attributes, respectively.

```
# Get the best score
gs.best_score_
```

```
np.float64(0.7395977155164125)
```

```
# Get the best hyperparameter values
gs.best_params_
```

```
{'columntransformer__countvectorizer__max_features': 1000,
 'svc__C': 1.0,
 'svc__gamma': 0.1}
```

- It is often helpful to visualize results of all cross-validation experiments.
- You can access this information using `cv_results_` attribute of a fitted `GridSearchCV` object.

```
results = pd.DataFrame(gs.cv_results_)
results.T
```

mean_fit_time	
std_fit_time	
mean_score_time	
std_score_time	
param_columntransformer__countvectorizer__max_features	
param_svc__C	
param_svc__gamma	
params	{'columntransformer__c 100, 'svc__(
split0_test_score	
split1_test_score	
split2_test_score	
split3_test_score	
split4_test_score	
mean_test_score	
std_test_score	
rank_test_score	
split0_train_score	
split1_train_score	
split2_train_score	
split3_train_score	
split4_train_score	
mean_train_score	
std_train_score	

23 rows × 216 columns

```
results = (
    pd.DataFrame(gs.cv_results_).set_index("rank_test_score").sort_index()
)
display(results.T)
```

rank_test_score	
mean_fit_time	
std_fit_time	
mean_score_time	
std_score_time	
param_columntransformer__countvectorizer__max_features	
param_svc__C	
param_svc__gamma	
params	{'columntransformer__cc 1000, 's
split0_test_score	
split1_test_score	
split2_test_score	
split3_test_score	
split4_test_score	
mean_test_score	
std_test_score	
split0_train_score	
split1_train_score	
split2_train_score	
split3_train_score	
split4_train_score	
mean_train_score	
std_train_score	

22 rows × 216 columns

Let's only look at the most relevant rows.

```
pd.DataFrame(gs.cv_results_)[
    [
        "mean_test_score",
        "param_columntransformer__countvectorizer__max_features",
        "param_svc__gamma",
        "param_svc__C",
        "mean_fit_time",
        "rank_test_score",
    ]
].set_index("rank_test_score").sort_index().T
```

rank_test_score	1	
mean_test_score	0.739598	0.73
param_columntransformer__countvectorizer__max_features	1000.000000	2000.00
param_svc__gamma	0.100000	0.10
param_svc__C	1.000000	1.00
mean_fit_time	0.072101	0.07

5 rows × 216 columns

- Other than searching for best hyperparameter values, `GridSearchCV` also fits a new model on the whole training set with the parameters that yielded the best results.
- So we can conveniently call `score` on the test set with a fitted `GridSearchCV` object.

```
# Get the test scores

gs.score(X_test, y_test)
```

```
0.7574257425742574
```

Why are `best_score_` and the score above different?

`n_jobs=-1`

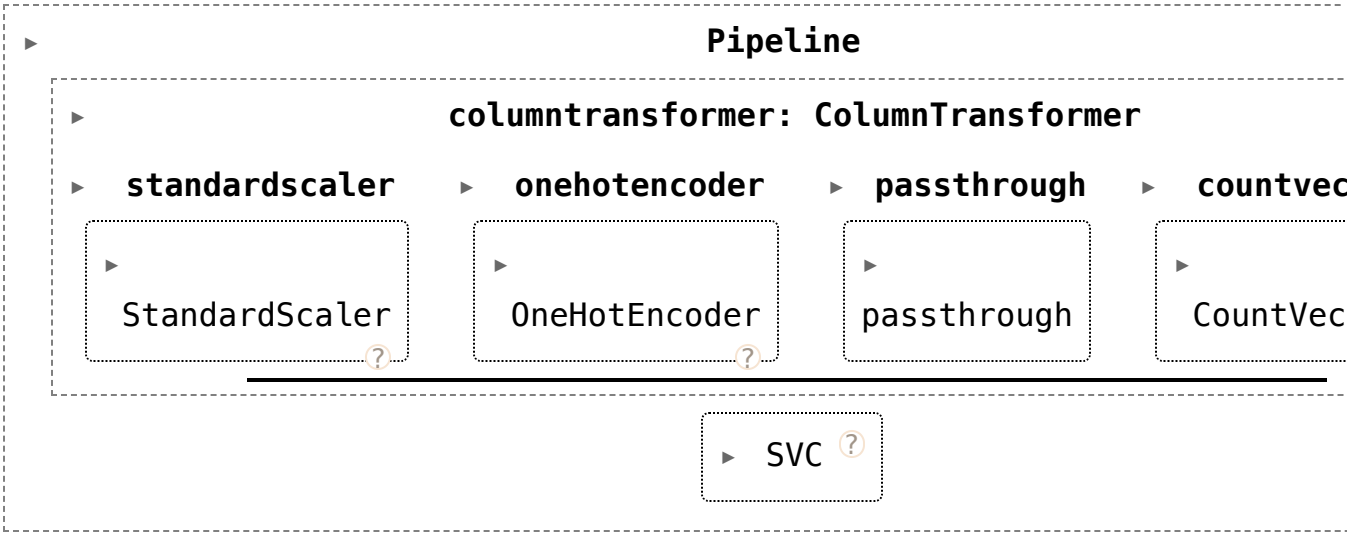
- Note the `n_jobs=-1` above.

- Hyperparameter optimization can be done *in parallel* for each of the configurations.
- This is very useful when scaling up to large numbers of machines in the cloud.
- When you set `n_jobs=-1`, it means that you want to use all available CPU cores for the task.

The `__` syntax

- Above: we have a nesting of transformers.
- We can access the parameters of the "inner" objects by using __ to go "deeper":
- `svc__gamma`: the `gamma` of the `svc` of the pipeline
- `svc__C`: the `C` of the `svc` of the pipeline
- `columntransformer__countvectorizer__max_features`: the `max_features` hyperparameter of `CountVectorizer` in the column transformer `preprocessor`.

```
pipe_svm
```



Range of `C`

- Note the exponential range for `C`. This is quite common. Using this exponential range allows you to explore a wide range of values efficiently.
- There is no point trying $C = \{1, 2, 3 \ldots, 100\}$ because $C = 1, 2, 3$ are too similar to each other.
- Often we're trying to find an order of magnitude, e.g. $C = \{0.01, 0.1, 1, 10, 100\}$.

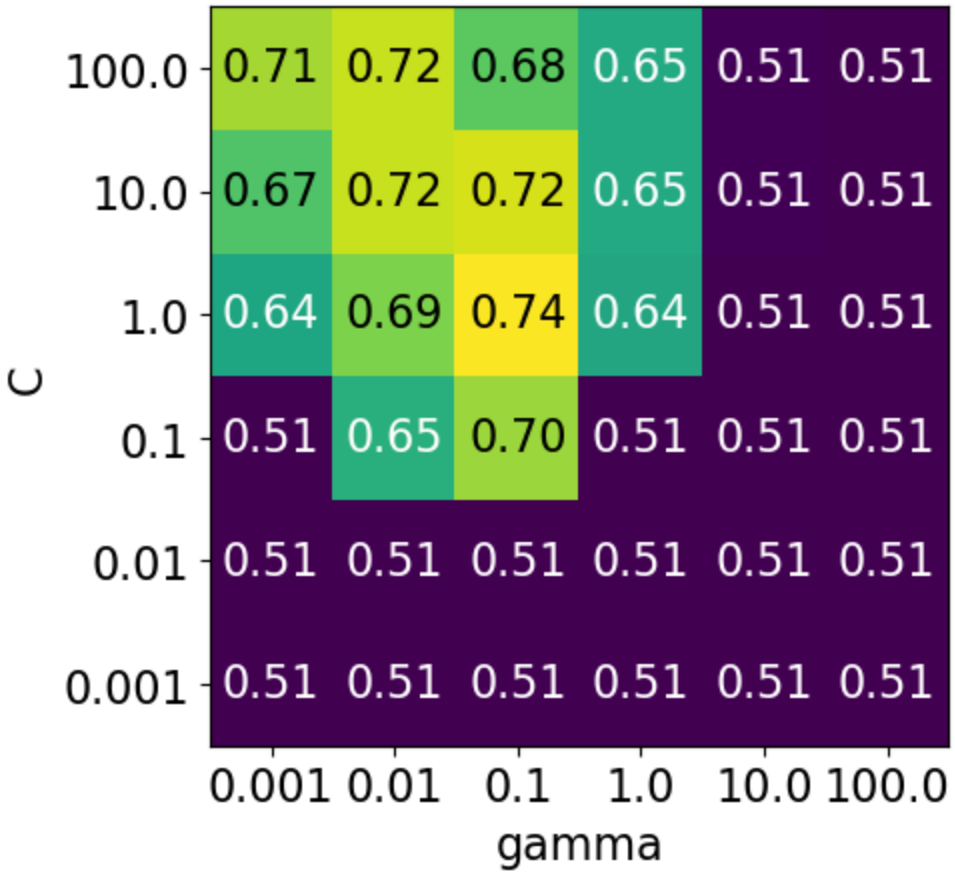
- We can also write that as $C = \{10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}\}$.
- Or, in other words, C values to try are 10^n for $n = -2, -1, 0, 1, 2$ which is basically what we have above.

Visualizing the parameter grid as a heatmap

```
def display_heatmap(param_grid, pipe, X_train, y_train):
    grid_search = GridSearchCV(
        pipe, param_grid, cv=5, n_jobs=-1, return_train_score=True
    )
    grid_search.fit(X_train, y_train)
    results = pd.DataFrame(grid_search.cv_results_)
    scores = np.array(results.mean_test_score).reshape(6, 6)

    # plot the mean cross-validation scores
    my_heatmap(
        scores,
        xlabel="gamma",
        xticklabels=param_grid["svc__gamma"],
        ylabel="C",
        yticklabels=param_grid["svc__C"],
        cmap="viridis",
    );
```

- Note that the range we pick for the parameters play an important role in hyperparameter optimization.
- For example, consider the following grid and the corresponding results.

```
param_grid1 = {
    "svc__gamma": 10.0**np.arange(-3, 3, 1),
    "svc__C": 10.0**np.arange(-3, 3, 1)
}
display_heatmap(param_grid1, pipe_svm, X_train, y_train)
```



- Each point in the heat map corresponds to one run of cross-validation, with a particular setting
- Colour encodes cross-validation accuracy.
  - Lighter colour means high accuracy
  - Darker colour means low accuracy
- SVC is quite sensitive to hyperparameter settings.
- Adjusting hyperparameters can change the accuracy from 0.51 to 0.74!

Bad range for hyperparameters

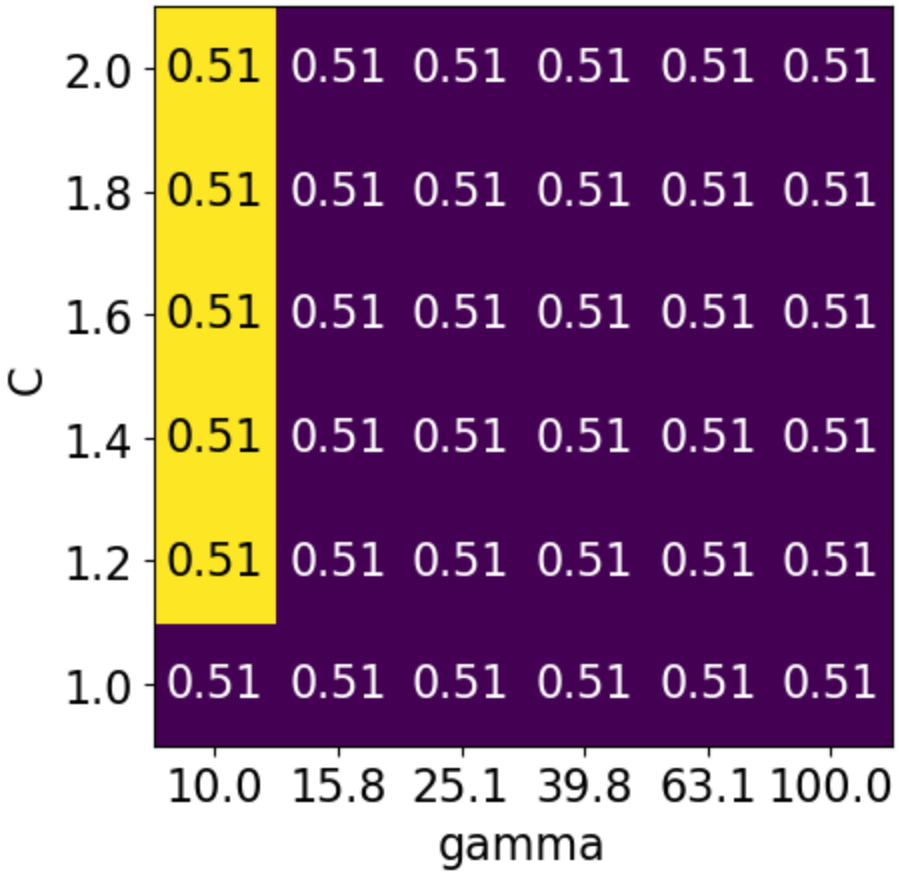
```
np.logspace(1, 2, 6)
```

```
array([ 10.        ,  15.84893192,  25.11886432,  39.81071706,
        63.09573445, 100.        ])
```

```
np.linspace(1, 2, 6)
```

```
array([1. , 1.2, 1.4, 1.6, 1.8, 2. ])
```

```
param_grid2 = {"svc__gamma": np.round(np.logspace(1, 2, 6), 1), "svc__C": np.l
display_heatmap(param_grid2, pipe_svm, X_train, y_train)
```



Different range for hyperparameters yields better results!

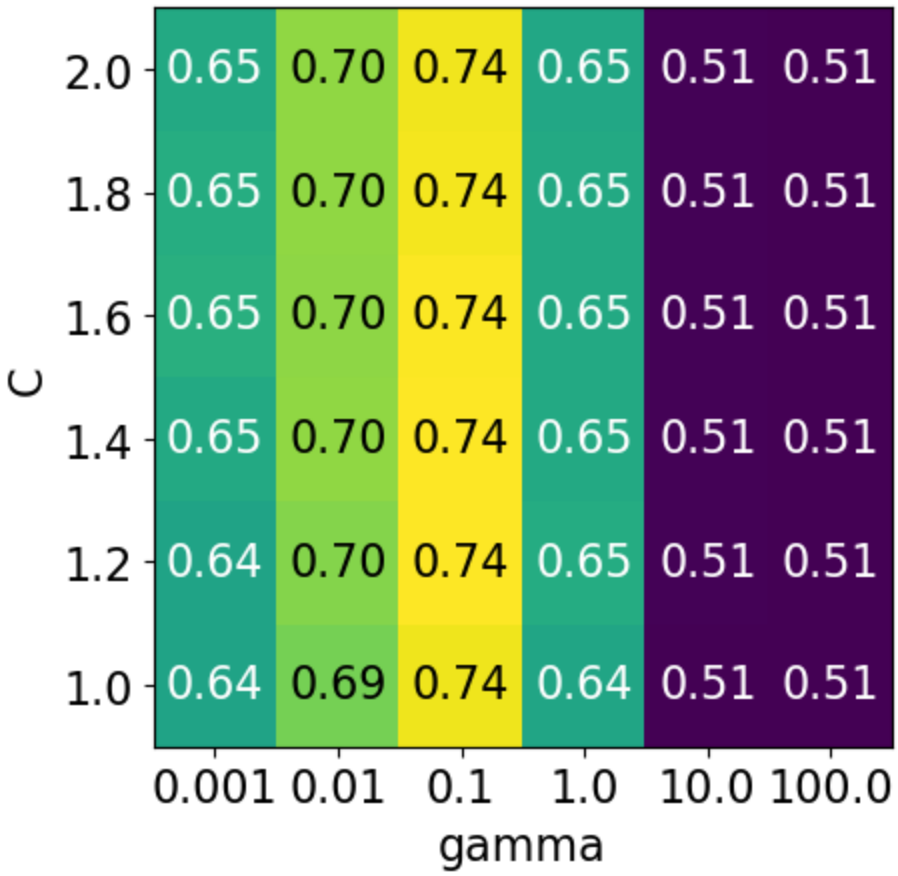
```
np.logspace(-3, 2, 6)
```

```
array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02])
```

```
np.linspace(1, 2, 6)
```

```
array([1. , 1.2, 1.4, 1.6, 1.8, 2. ])
```

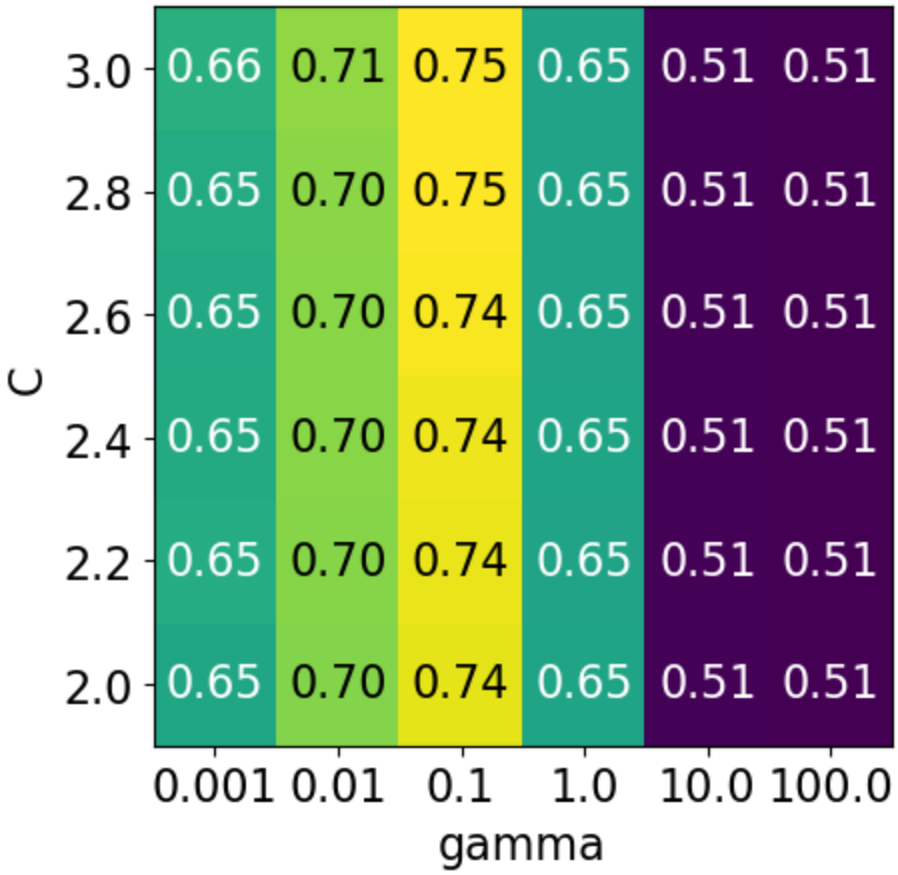
```
param_grid3 = {"svc__gamma": np.logspace(-3, 2, 6), "svc__C": np.linspace(1, 2
display_heatmap(param_grid3, pipe_svm, X_train, y_train)
```



It seems like we are getting even better cross-validation results with `C` = 2.0 and `gamma` = 0.1

How about exploring different values of `C` close to 2.0?

```
param_grid4 = {"svc__gamma": np.logspace(-3, 2, 6), "svc__C": np.linspace(2, 3
display_heatmap(param_grid4, pipe_svm, X_train, y_train)
```



That's good! We are finding some more options for `C` where the accuracy is 0.75. The tricky part is we do not know in advance what range of hyperparameters might work the best for the given problem, model, and the dataset.

> **Note**
>
> `GridSearchCV` allows the param_grid to be a list of dictionaries. Sometimes some hyperparameters are applicable only for certain models. For example, in the context of `SVC`, `C` and `gamma` are applicable when the kernel is `rbf` whereas only `C` is applicable for `kernel="linear"`.

Problems with exhaustive grid search

- Required number of models to evaluate grows exponentially with the dimensionally of the configuration space.
- Example: Suppose you have
  - 5 hyperparameters
  - 10 different values for each hyperparameter

- You'll be evaluating $10^5 = 100,000$ models! That is you'll be calling `cross_validate` 100,000 times!
- Exhaustive search may become infeasible fairly quickly.
- Other options?

Randomized hyperparameter search

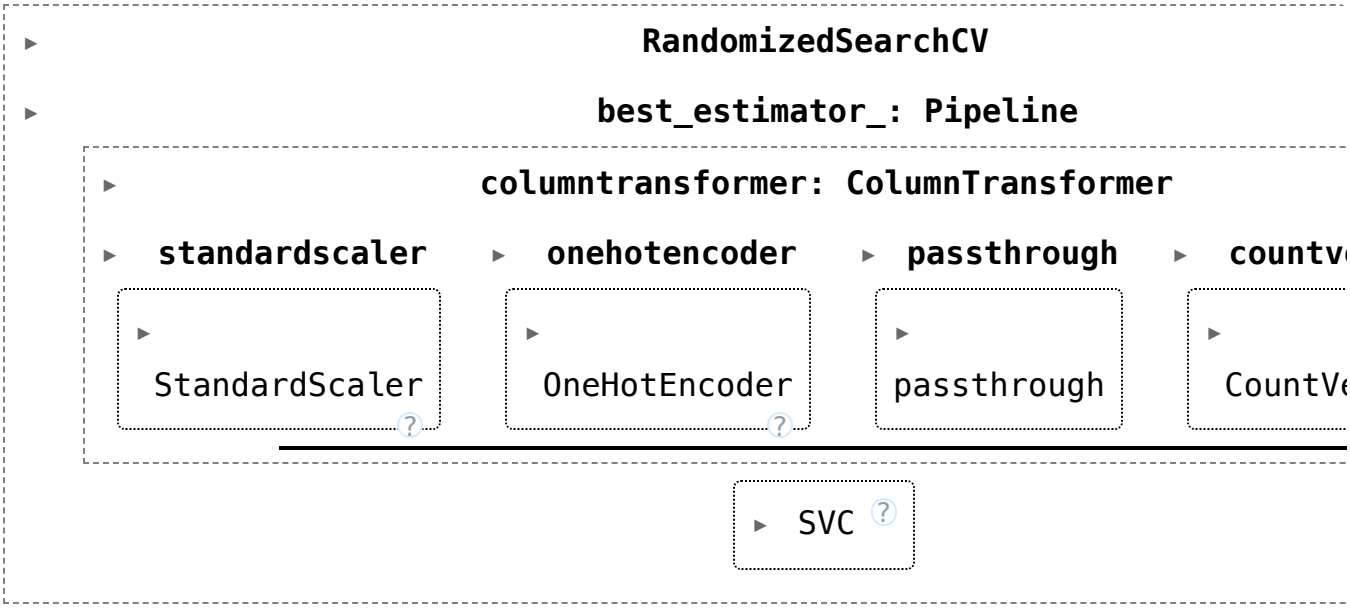
- Randomized hyperparameter optimization
  - **`sklearn.model_selection.RandomizedSearchCV`**
- Samples configurations at random until certain budget (e.g., time) is exhausted

```
from sklearn.model_selection import RandomizedSearchCV

param_grid = {
    "columntransformer__countvectorizer__max_features": [100, 200, 400, 800, 1
    "svc__gamma": [0.001, 0.01, 0.1, 1.0, 10, 100],
    "svc__C": np.linspace(2, 3, 6),
}

print("Grid size: %d" % (np.prod(list(map(len, param_grid.values())))))
param_grid
```

```
Grid size: 216
```

```
{'columntransformer__countvectorizer__max_features': [100,
  200,
  400,
  800,
  1000,
  2000],
 'svc__gamma': [0.001, 0.01, 0.1, 1.0, 10, 100],
 'svc__C': array([2. , 2.2, 2.4, 2.6, 2.8, 3. ])}
```

```
# Create a random search object
random_search = RandomizedSearchCV(pipe_svm,
                   param_distributions = param_grid,
                   n_iter=100,
                   n_jobs=-1,
                   return_train_score=True)

# Carry out the search
random_search.fit(X_train, y_train)
```




```
pd.DataFrame(random_search.cv_results_)[
    [
        "mean_test_score",
        "param_columntransformer__countvectorizer__max_features",
        "param_svc__gamma",
        "param_svc__C",
        "mean_fit_time",
        "rank_test_score",
    ]
].set_index("rank_test_score").sort_index().T
```

rank_test_score	1	
mean_test_score	0.744565	0.7433
param_columntransformer__countvectorizer__max_features	400.000000	800.0000
param_svc__gamma	0.100000	0.1000
param_svc__C	3.000000	2.6000
mean_fit_time	0.070555	0.0895

5 rows × 100 columns

`n_iter`

- Note the `n_iter`, we didn't need this for `GridSearchCV`.
- Larger `n_iter` will take longer but it'll do more searching.
  - Remember you still need to multiply by number of folds!
- I have set the `random_state` for reproducibility but you don't have to do it.

Passing probability distributions to random search

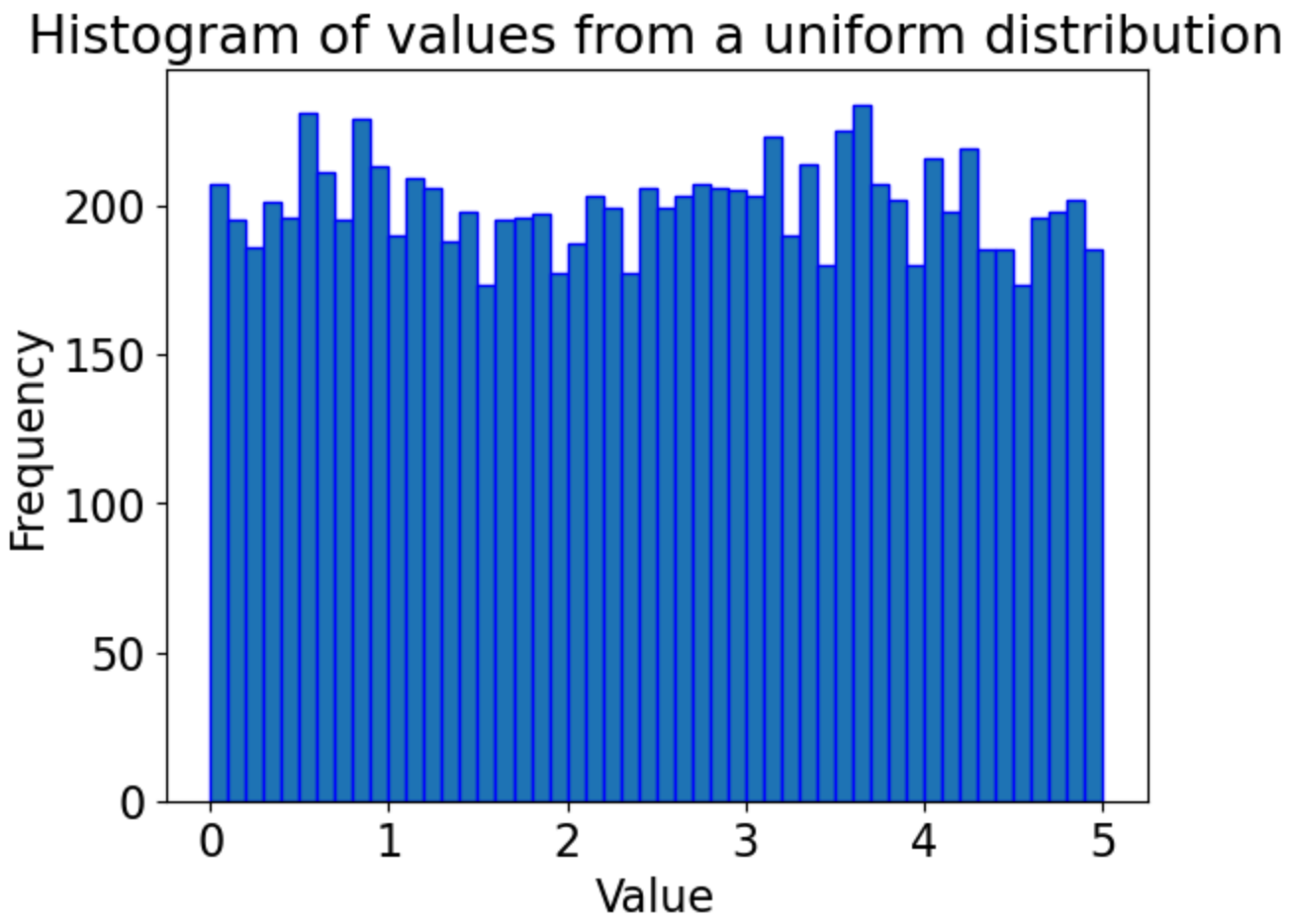
Another thing we can do is give probability distributions to draw from:

```
from scipy.stats import expon, lognorm, loguniform, randint, uniform, norm, ra
```

```
np.random.seed(123)
def plot_distribution(y, bins, dist_name="uniform", x_label='Value', y_label="
    plt.hist(y, bins=bins, edgecolor='blue')
    plt.title('Histogram of values from a {} distribution'.format(dist_name))
    plt.xlabel('Value')
    plt.ylabel('Frequency')
    plt.show()
```

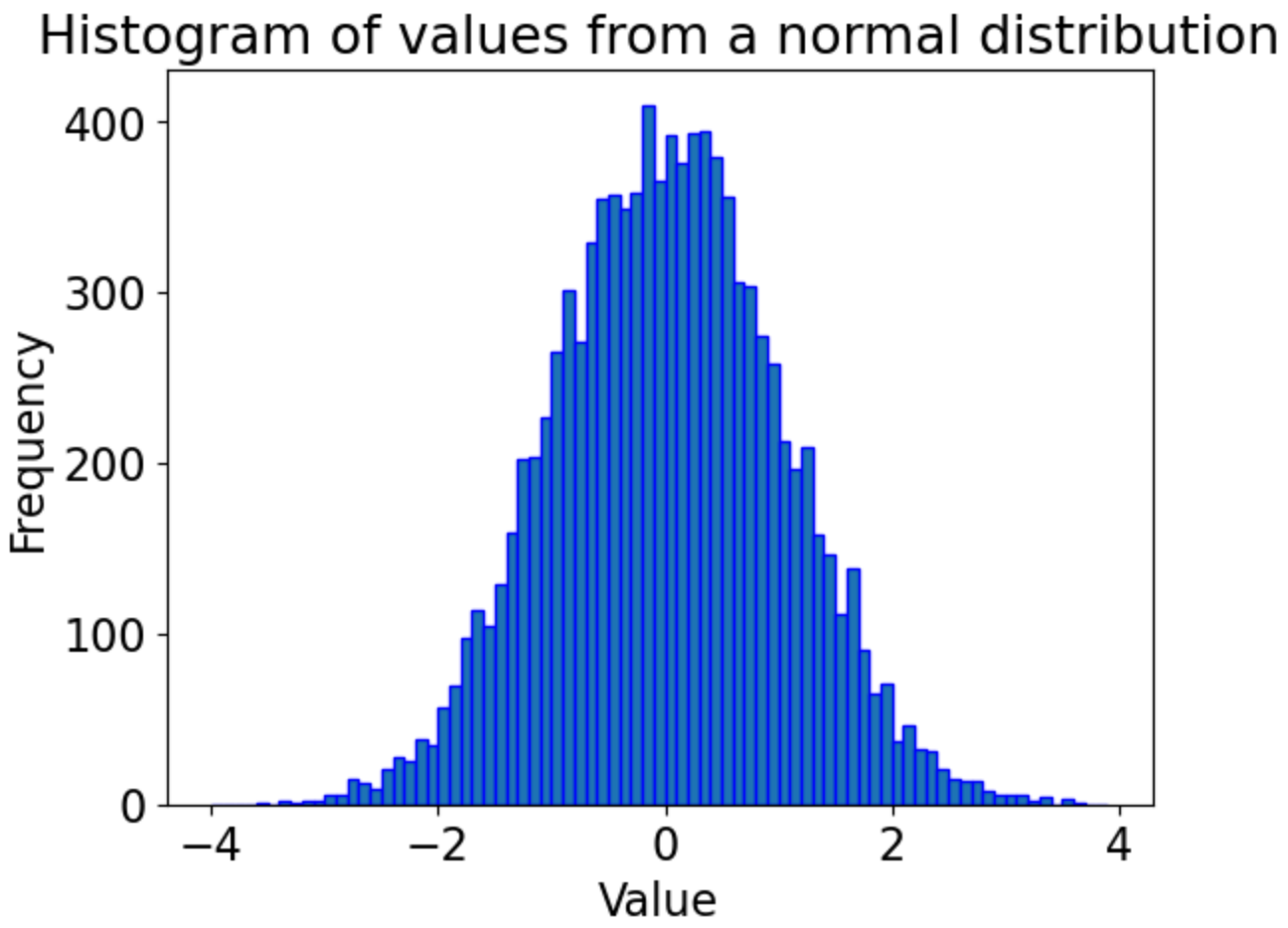
Uniform distribution

```
# Generate random values from a uniform distribution
y = uniform.rvs(0, 5, size=10000)

# Creating bins within the range of the data
bins = np.arange(0, 5.1, 0.1)  # Bins from 0 to 5, in increments of 0.1

plot_distribution(y, bins, "uniform")
```

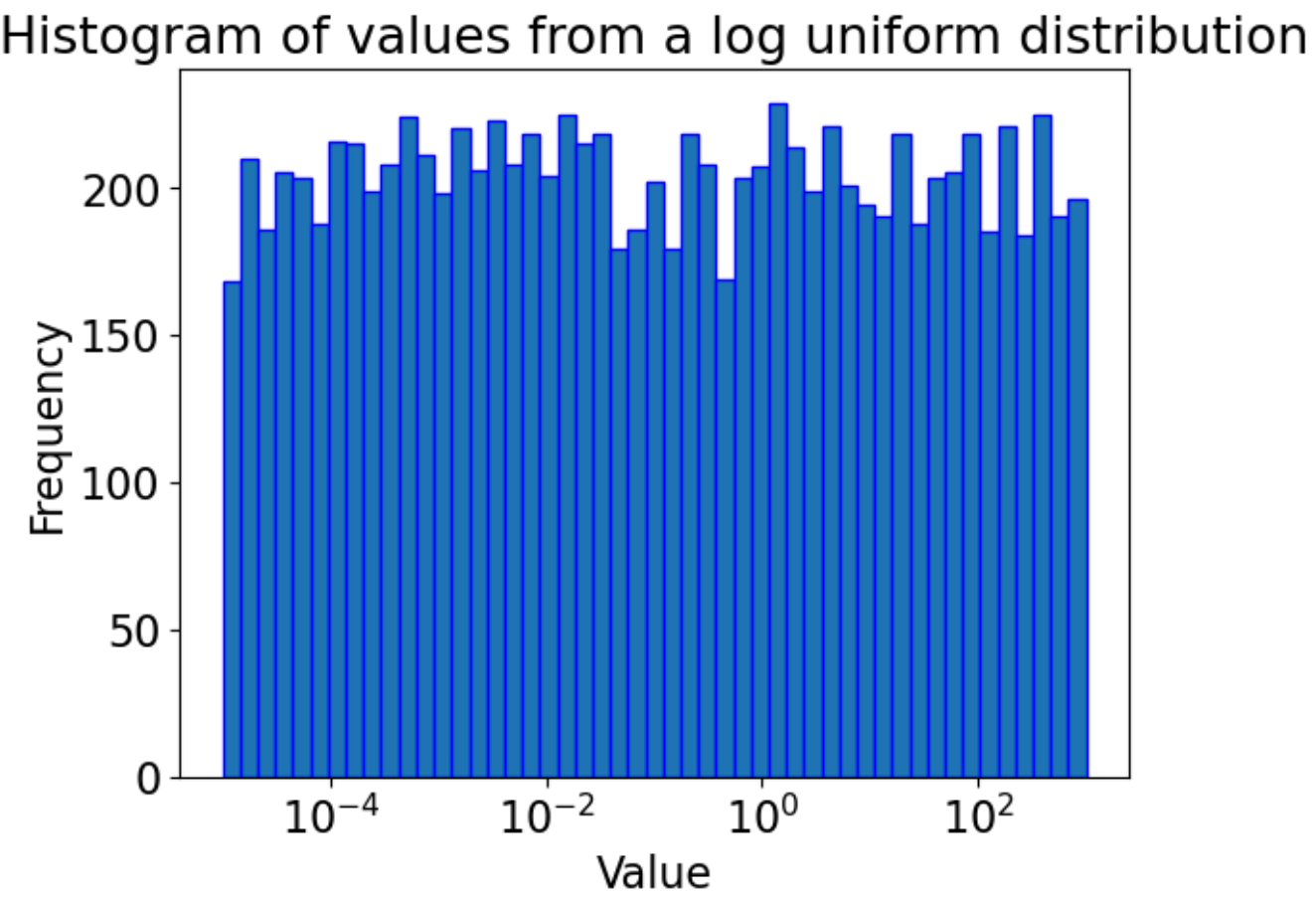


Gaussian distribution

```
y = norm.rvs(0, 1, 10000)

# Creating bins
bins = np.arange(-4, 4, 0.1)

plot_distribution(y, bins, "normal")  # Corrected distribution name
```




```
# Generate random values from a log-uniform distribution
# Using loguniform from scipy.stats, specifying a range using the exponents
y = loguniform.rvs(1e-5, 1e3, size=10000)

# Creating bins on a logarithmic scale for better visualization
bins = np.logspace(np.log10(1e-5), np.log10(1e3), 50)
plt.xscale('log')

plot_distribution(y, bins, "log uniform")  # Corrected distribution name
```

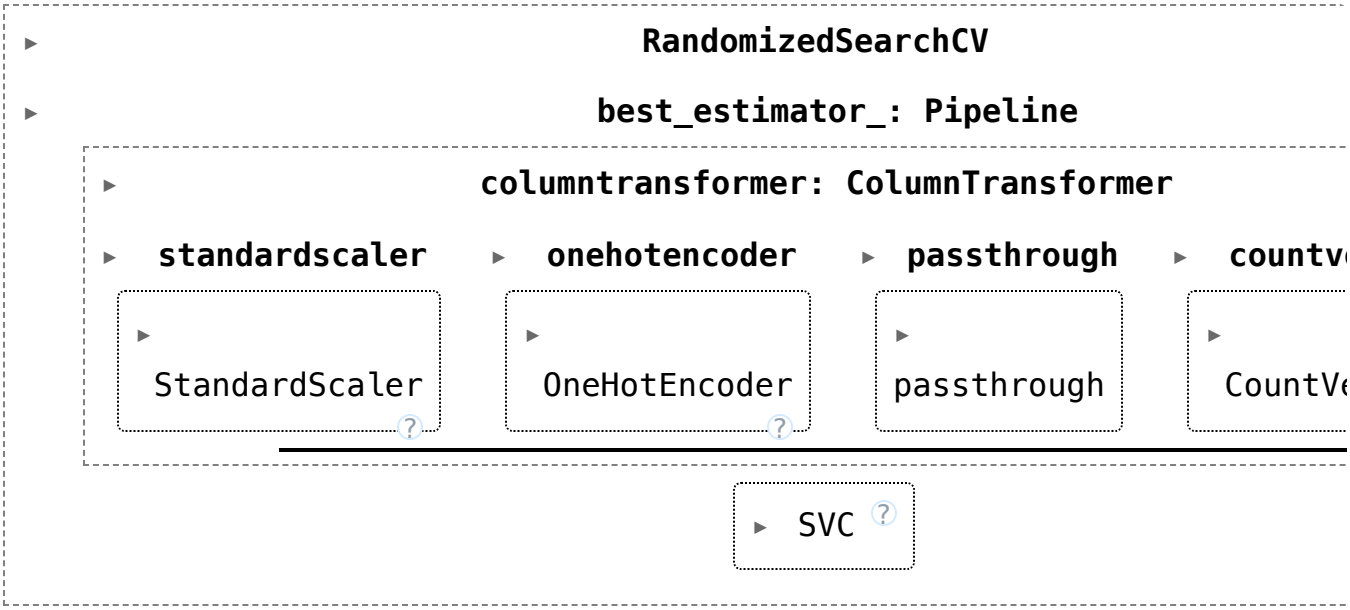



```
pipe_svm
```

Pipeline
columntransformer: ColumnTransformer
standardscaler
StandardScaler
onehotencoder
OneHotEncoder
passthrough
passthrough
countvec
CountVec
SVC

```
from scipy.stats import randint

param_dist = {
    "columntransformer__countvectorizer__max_features": randint(100, 2000),
    "svc__C": loguniform(1e-3, 1e3),
    "svc__gamma": loguniform(1e-5, 1e3),
}
```

```
# Create a random search object
random_search = RandomizedSearchCV(pipe_svm,
                   param_distributions = param_dist,
                   n_iter=100,
                   n_jobs=-1,
                   return_train_score=True)

# Carry out the search
random_search.fit(X_train, y_train)
```




```
random_search.best_score_
```

```
np.float64(0.7278214718381631)
```

```
pd.DataFrame(random_search.cv_results_)[
    [
        "mean_test_score",
        "param_columntransformer__countvectorizer__max_features",
        "param_svc__gamma",
        "param_svc__C",
        "mean_fit_time",
        "rank_test_score",
    ]
].set_index("rank_test_score").sort_index().T
```

rank_test_score	1	
mean_test_score	0.727821	0.727
param_columntransformer__countvectorizer__max_features	1619.000000	771.000
param_svc__gamma	0.123539	0.111
param_svc__C	33.944878	0.398
mean_fit_time	0.101156	0.070

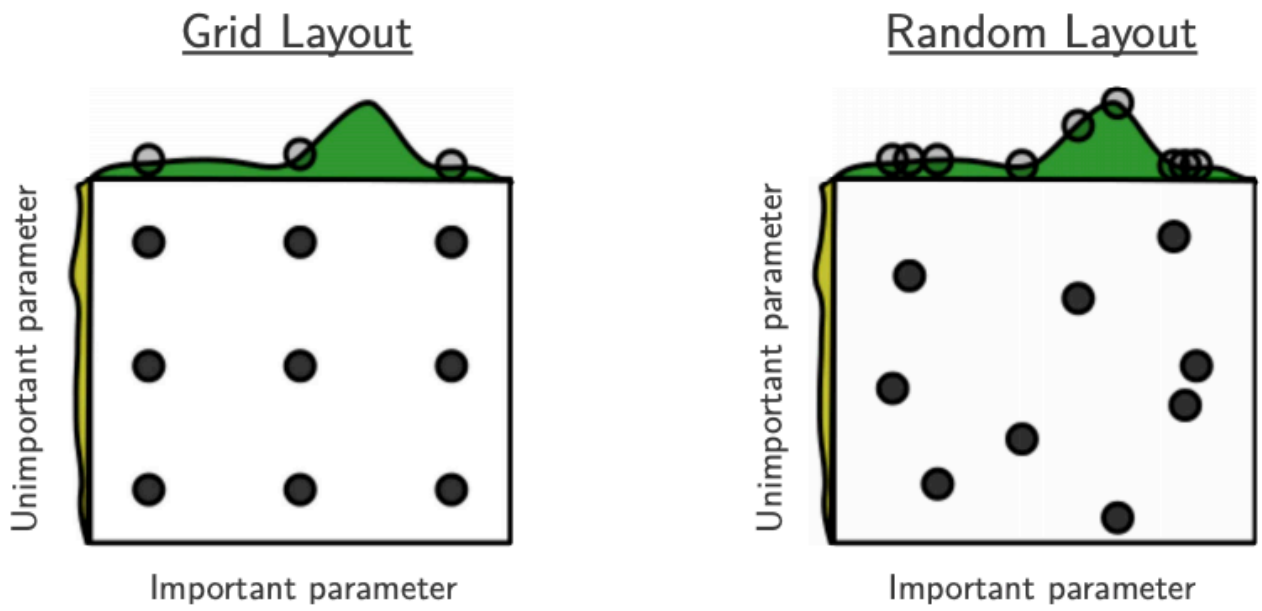
5 rows × 100 columns

- This is a bit fancy. What's nice is that you can have it concentrate more on certain values by setting the distribution.

Advantages of `RandomizedSearchCV`

- Faster compared to `GridSearchCV`.
- Adding parameters that do not influence the performance does not affect efficiency.
- Works better when some parameters are more important than others.
- In general, I recommend using `RandomizedSearchCV` rather than `GridSearchCV`.

Advantages of `RandomizedSearchCV`



Source: Bergstra and Bengio, Random Search for Hyper-Parameter Optimization, JMLR 2012.

- The yellow on the left shows how your scores are going to change when you vary the unimportant hyperparameter.
- The green on the top shows how your scores are going to change when you vary the important hyperparameter.
- You don't know in advance which hyperparameters are important for your problem.
- In the left figure, 6 of the 9 searches are useless because they are only varying the unimportant parameter.
- In the right figure, all 9 searches are useful.

(Optional) Searching for optimal parameters with successive halving¶

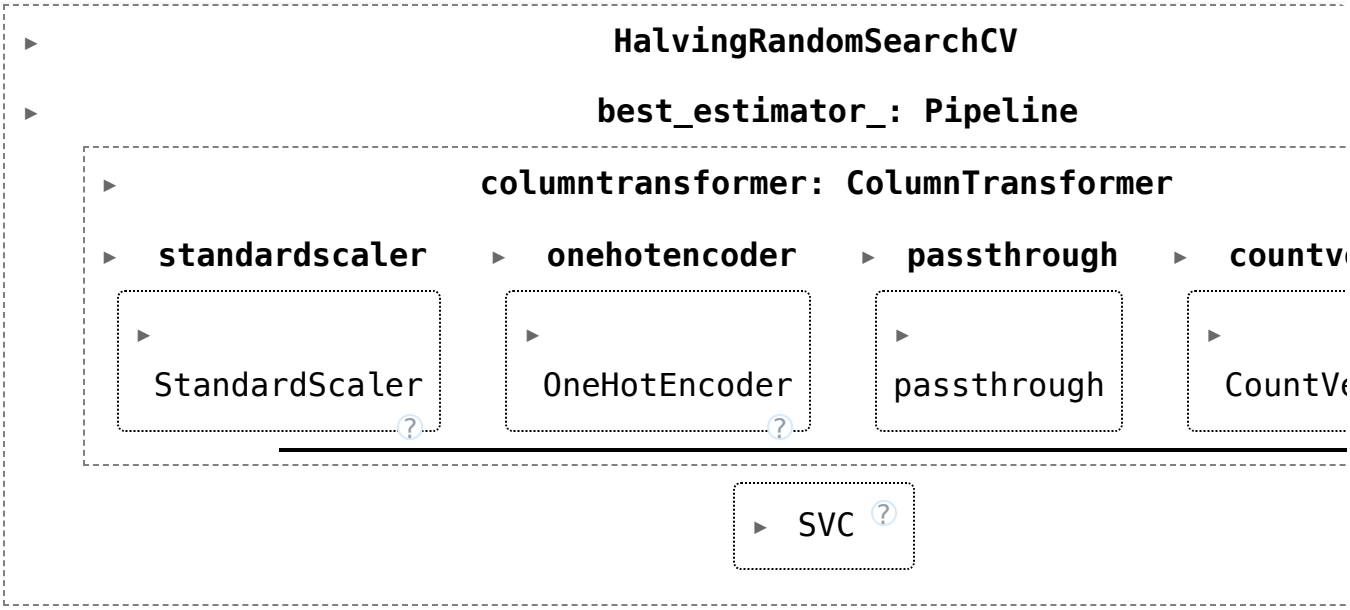
- Successive halving is an iterative selection process where all candidates (the parameter combinations) are evaluated with a small amount of resources (e.g., small amount of training data) at the first iteration.
- Checkout successive halving with grid search and random search.

```
from sklearn.experimental import enable_halving_search_cv  # noqa
from sklearn.model_selection import HalvingRandomSearchCV
```

```
rsh = HalvingRandomSearchCV(
    estimator=pipe_svm, param_distributions=param_dist, factor=2, random_state
)
rsh.fit(X_train, y_train)
```

```
/Users/kvarada/miniforge3/envs/jbook/lib/python3.12/site-packages/numpy/ma/cor
  _data = np.array(data, dtype=dtype, copy=copy,
```

```
/Users/kvarada/miniforge3/envs/jbook/lib/python3.12/site-packages/numpy/ma/cor
  _data = np.array(data, dtype=dtype, copy=copy,
```




```
results = pd.DataFrame(rsh.cv_results_)
results["params_str"] = results.params.apply(str)
results.drop_duplicates(subset=("params_str", "iter"), inplace=True)
results
```

	iter	n_resources	mean_fit_time	std_fit_time	mean_score_time	std_scor
0	0	20	0.003520	0.000498	0.001708	0.(
1	0	20	0.003105	0.000235	0.001664	0.(
2	0	20	0.003079	0.000238	0.001641	0.(
3	0	20	0.002976	0.000172	0.001607	0.(
4	0	20	0.003004	0.000137	0.001599	0.(
...	...	...	...	...	...	
155	5	640	0.010648	0.000234	0.003246	0.(
156	5	640	0.013042	0.000263	0.003551	0.(
157	5	640	0.014676	0.000117	0.003718	0.(
158	6	1280	0.037244	0.000187	0.007978	0.
159	6	1280	0.045848	0.000670	0.008781	0.(

160 rows × 26 columns

(Optional) Fancier methods

- Both `GridSearchCV` and `RandomizedSearchCV` do each trial independently.
- What if you could learn from your experience, e.g. learn that `max_depth=3` is bad?

    - That could save time because you wouldn't try combinations involving `max_depth=3` in the future.
- We can do this with `scikit-optimize`, which is a completely different package from `scikit-learn`
- It uses a technique called "model-based optimization" and we'll specifically use "Bayesian optimization".
    - In short, it uses machine learning to predict what hyperparameters will be good.
    - Machine learning on machine learning!
- This is an active research area and there are sophisticated packages for this.

Here are some examples

- hyperopt-sklearn
- auto-sklearn
- SigOptSearchCV
- TPOT
- hyperopt
- hyperband
- SMAC
- MOE
- pybo
- spearmint
- BayesOpt