# Machine learning workshops for material scientists

## Lecture 6: Supervised linear model with Python

October 31, 2022

**Sira Sriswasdi, PhD**

- Research Affairs
- Center of Excellence in Computational Molecular Biology (CMB)
- Center for Artificial Intelligence in Medicine (CU-AIM)

## Today’s goals

- Quick recap of linear models
- Get to know Python library
- Practice on Colab

## Linear and logistic regression

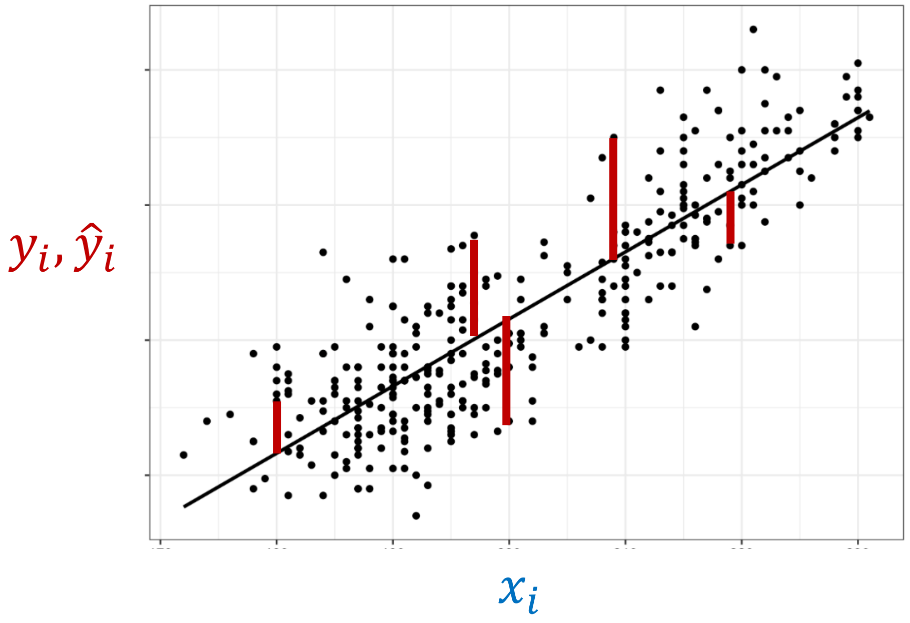
## Linear and logistic regression key points

- Predictions are made via linear combination of input features
    - $\hat{y} = b_0 + b_1 x_1 + \cdots + b_n x_n$
    - $\log\left(\frac{\hat{y}}{1-\hat{y}}\right) = b_0 + b_1 x_1 + \cdots + b_n x_n$

- Primary objectives:
    - MSE: $\frac{1}{n}\sum(y - \hat{y})^2$
    - Log-Likelihood, Cross-Entropy: $\sum y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$

- Regularization
    - LASSO: objective + $\alpha \sum |b_i|$ or Ridge: Objective + $\alpha \sum b_i^2$

## Linear regression



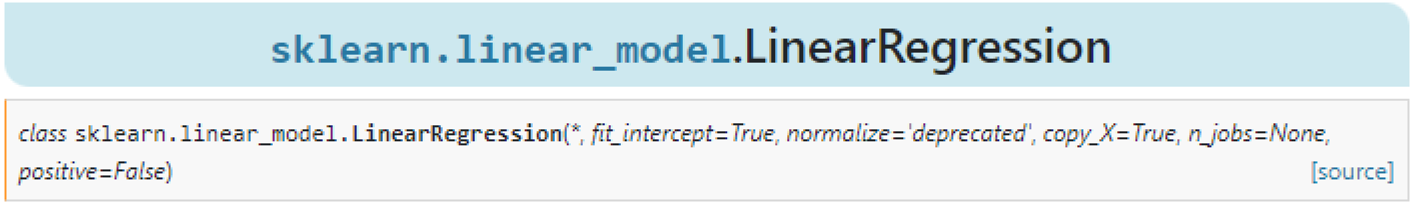
https://sixsigmadsi.com/glossary/simple-linear-regression/

- $\hat{y} = b_0 + b_1 x_1 + \cdots + b_n x_n$
- Minimize MSE: $\frac{1}{n}\sum(y - \hat{y})^2$

- MSE objective is related to assumption that errors are normally distributed and centered at zero

- Data transformation (e.g., log) can be beneficial

## Linear regression in Python

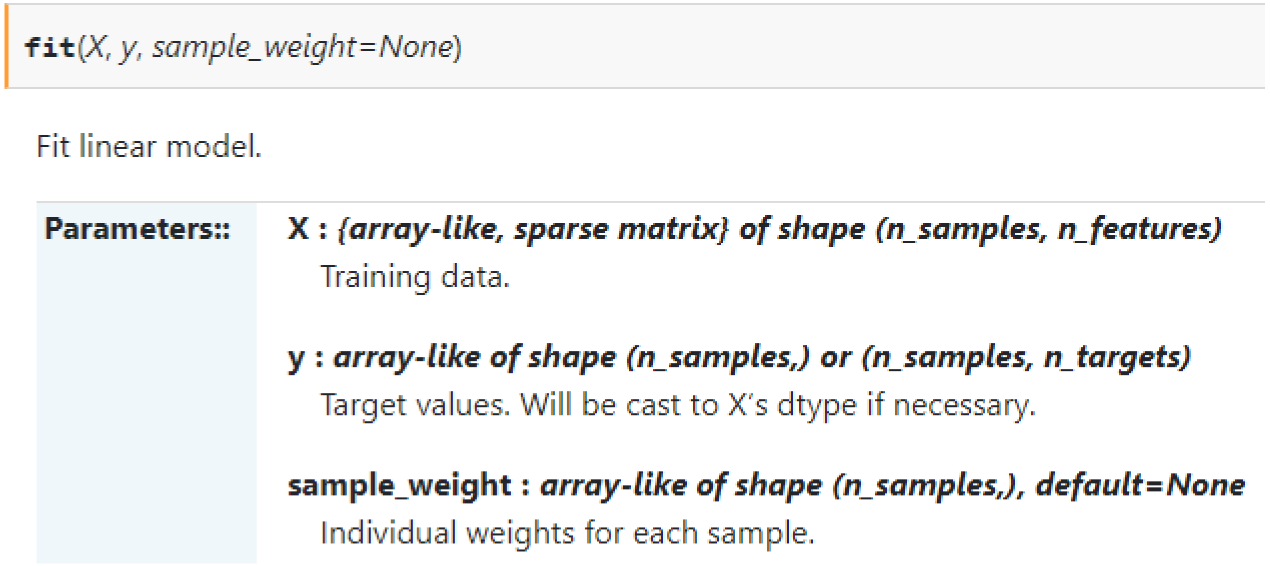


**sklearn.linear_model.LinearRegression**

*class* sklearn.linear_model.**LinearRegression**(**, fit_intercept=True, normalize='deprecated', copy_X=True, n_jobs=None, positive=False*) [source]

```
from sklearn.linear_model import LinearRegression

linear = LinearRegression().fit(input_data, label)
prediction = linear.predict(new_data)

## array of fitted coefficients ($b_1$,$b_2$,…,$b_n$) for input features
linear.coef_

## y-intercept ($b_0$)
linear.intercept_
```

## Sample weighting



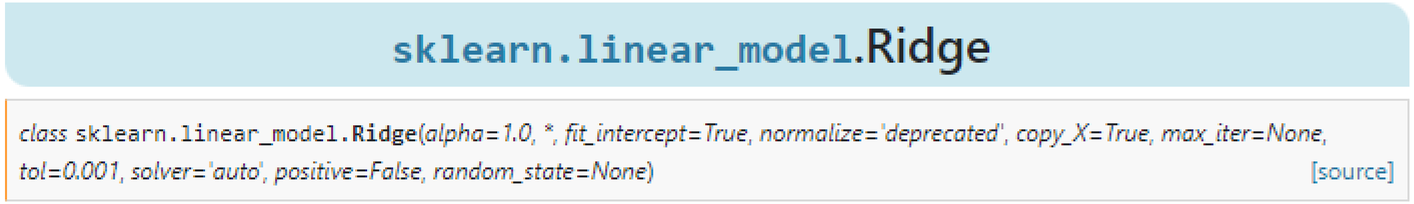
**fit**(*X, y, sample_weight=None*)

Fit linear model.

| | |
|---|---|
| **Parameters::** | **X : *{array-like, sparse matrix} of shape (n_samples, n_features)*** <br> Training data. <br><br> **y : *array-like of shape (n_samples,) or (n_samples, n_targets)*** <br> Target values. Will be cast to X's dtype if necessary. <br><br> **sample_weight : *array-like of shape (n_samples,), default=None*** <br> Individual weights for each sample. |

- Add weight to the calculation of objective: $\sum w_i(y_i - \hat{y_i})^2$
- Large weight → focus on fitting those data points

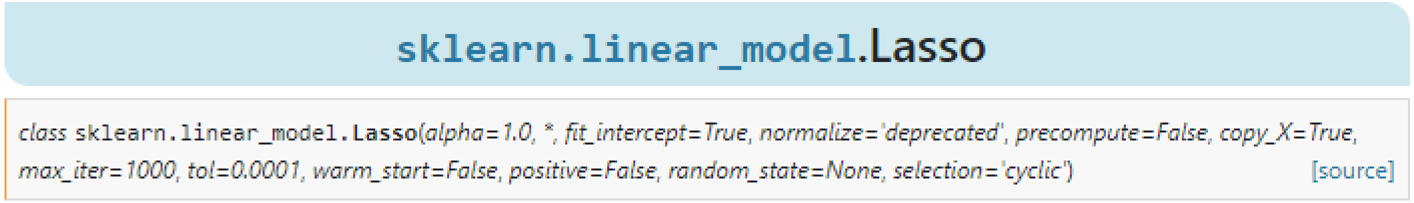
## Ridge and LASSO in Python



sklearn.linear_model.Ridge

```
class sklearn.linear_model.Ridge(alpha=1.0, *, fit_intercept=True, normalize='deprecated', copy_X=True, max_iter=None,
tol=0.001, solver='auto', positive=False, random_state=None)
```

[source]



sklearn.linear_model.Lasso

```
class sklearn.linear_model.Lasso(alpha=1.0, *, fit_intercept=True, normalize='deprecated', precompute=False, copy_X=True,
max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic')
```

[source]

- alpha = regularization strength
  - Larger alpha → try to keep coefficients small
  - Small alpha → focus more on prediction performance

## Ridge and LASSO usage

```
from sklearn.linear_model import Ridge, Lasso

## Ridge model
ridge = Ridge(alpha = 10, random_state = 25)
ridge.fit(input_data, label)

## LASSO model
lasso = Lasso(alpha = 10, random_state = 25)
lasso.fit(input_data, label)

## make predictions
prediction = ridge.predict(new_data)
```

## Logistic regression

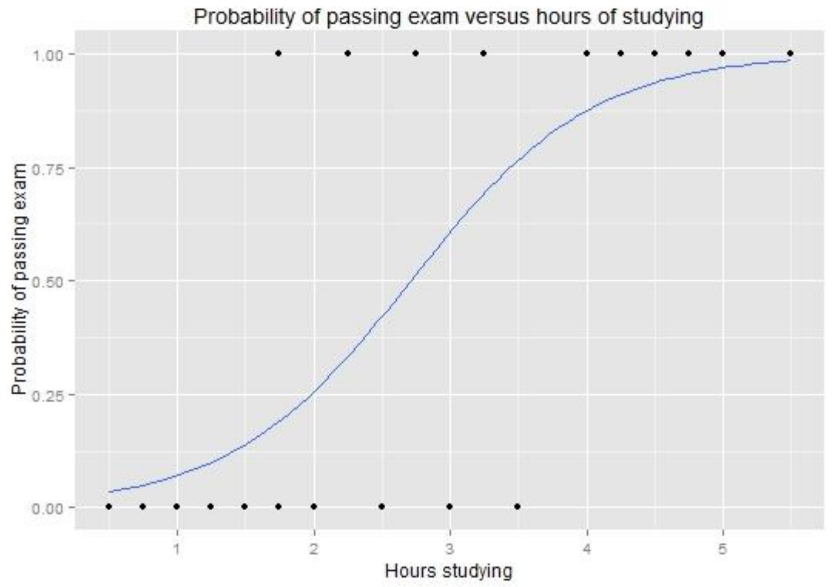


Image from Wikipedia

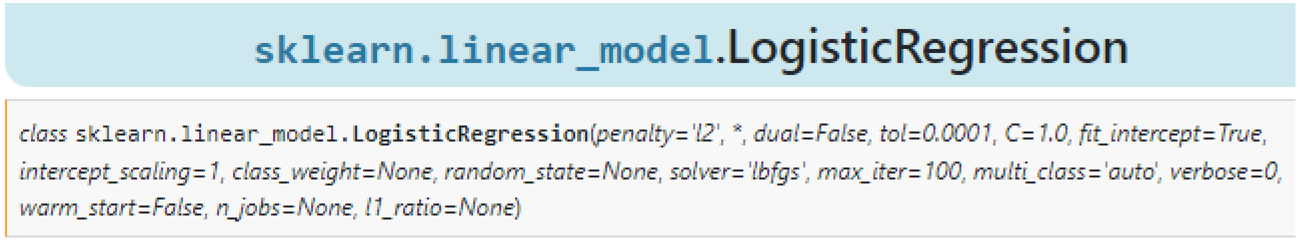
- Transform linear regression output into probability via log-odd function
- $\log\left(\frac{\hat{y}}{1-\hat{y}}\right) = b_0 + b_1 x_1 + \cdots + b_n x_n$

- Minimize Cross-Entropy:
  - $\sum y \log(\hat{y}) + (1-y)\log(1-\hat{y})$

- Data transformation (e.g., log) can be beneficial

- Well-calibrated (interpretable output)

## Logistic regression in Python



sklearn.linear_model.LogisticRegression

```
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)
```

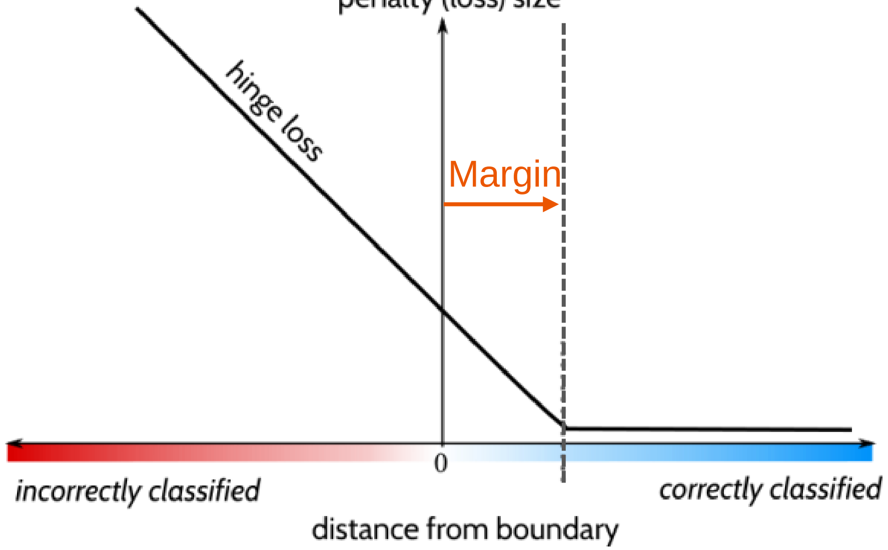
- penalty = type or regularization (l1 = LASSO, l2 = Ridge)
- C = inverse of regularization strength (1 / alpha)
- solver = optimization algorithm to be used
  - Not all support LASSO regularization → change to “liblinear”

## Logistic regression usage

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(penalty = ‘l2’, C = 0.01, random_state = 25)
logreg.fit(input_data, label)

## coefficients
logreg.coef_
logreg.intercept_

## predict classes
pred_class = logreg.predict(new_data)

## predict raw probabilities
pred_prob = logreg.predict_proba(new_data)
```

## Support vector machine

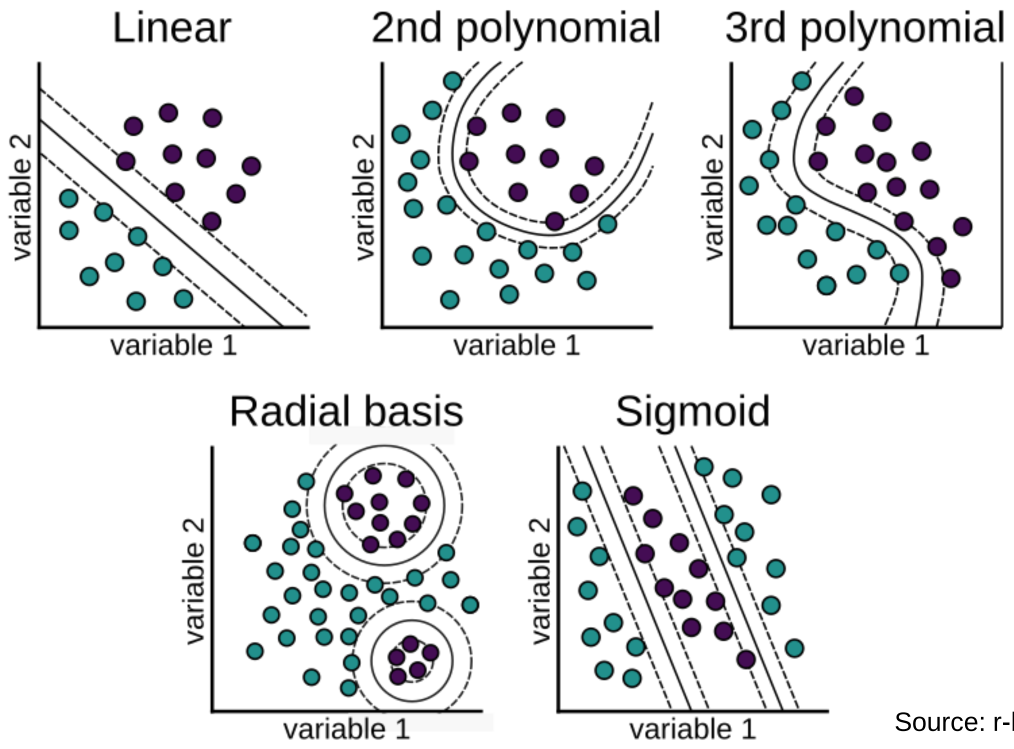
## Support vector machine key points



Source: towarddatasciences.com

- Hinge loss
    - Penalize correctly classified data points that lie too close to the decision boundary
    - Do not penalize small regression error
- Generalize to non-linear feature engineering via kernel
    - Radial basis
    - Polynomial
- Does not output class probabilities

## Kernels

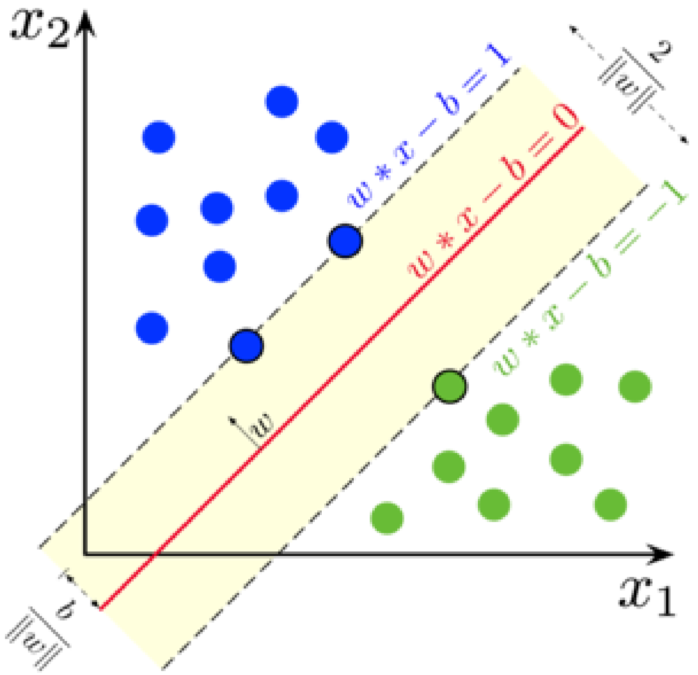


- Radial basis operates like nearest neighbor algorithm
  - Require large training data to be effective

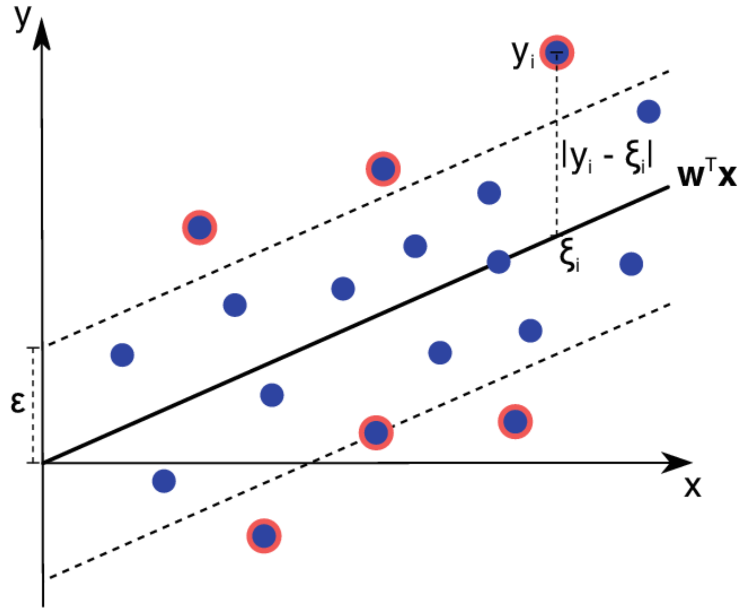
- Polynomial can work well when there are interactions between features
  - Terms $x_i x_j$ are produced

Source: r-bloggers.com

## Support vector classifier versus regressor

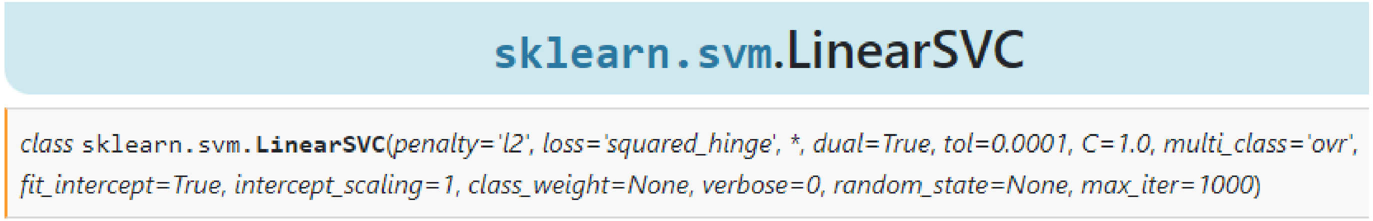


Source: wikipedia.com



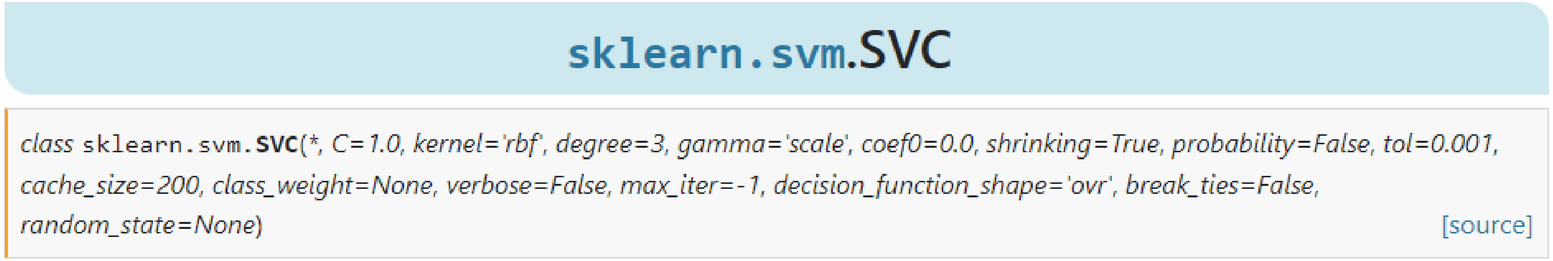
Rosenbaum, L. et al. J of Cheminformatics 5:33 (2013)

## Support vector classifier in Python



sklearn.svm.LinearSVC

```
class sklearn.svm.LinearSVC(penalty='l2', loss='squared_hinge', *, dual=True, tol=0.0001, C=1.0, multi_class='ovr', fit_intercept=True, intercept_scaling=1, class_weight=None, verbose=0, random_state=None, max_iter=1000)
```



sklearn.svm.SVC

```
class sklearn.svm.SVC(*, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', break_ties=False, random_state=None)
```

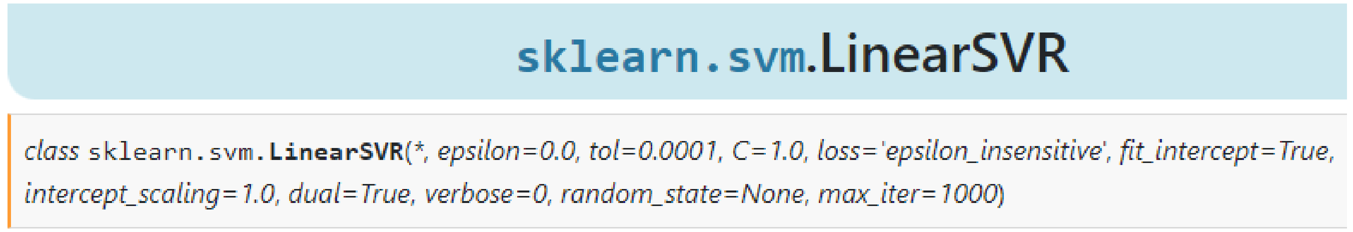
[source]

- C = inverse of regularization strength
- kernel, degree = set kernel for non-linear SVM
- probability = make the model estimate probability (empirically from cross-validation), enable predict_proba

## Support vector classifier usage

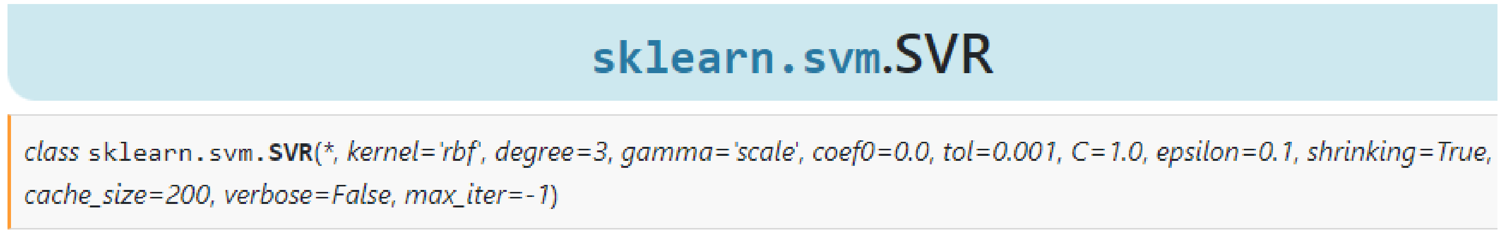
```
from sklearn.svm import LinearSVC, SVC

## linear kernel
linearsvc = LinearSVC(C = 0.01, random_state = 25)
linearsvc.fit(input_data, label)
prediction = linearsvc.predict(new_data)
linearsvc.coef_

## non-linear kernel
svc = SVC(C = 0.01, kernel = 'poly', degree = 2, random_state = 25)
svc.fit(input_data, label)
prediction = svc.predict(new_data)

## get all support vectors (data points near decision boundary)
svc.support_vectors_
```

## Support vector regressor in Python



### sklearn.svm.LinearSVR

```
class sklearn.svm.LinearSVR(*, epsilon=0.0, tol=0.0001, C=1.0, loss='epsilon_insensitive', fit_intercept=True,
intercept_scaling=1.0, dual=True, verbose=0, random_state=None, max_iter=1000)
```



### sklearn.svm.SVR

```
class sklearn.svm.SVR(*, kernel='rbf', degree=3, gamma='scale', coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True,
cache_size=200, verbose=False, max_iter=-1)
```

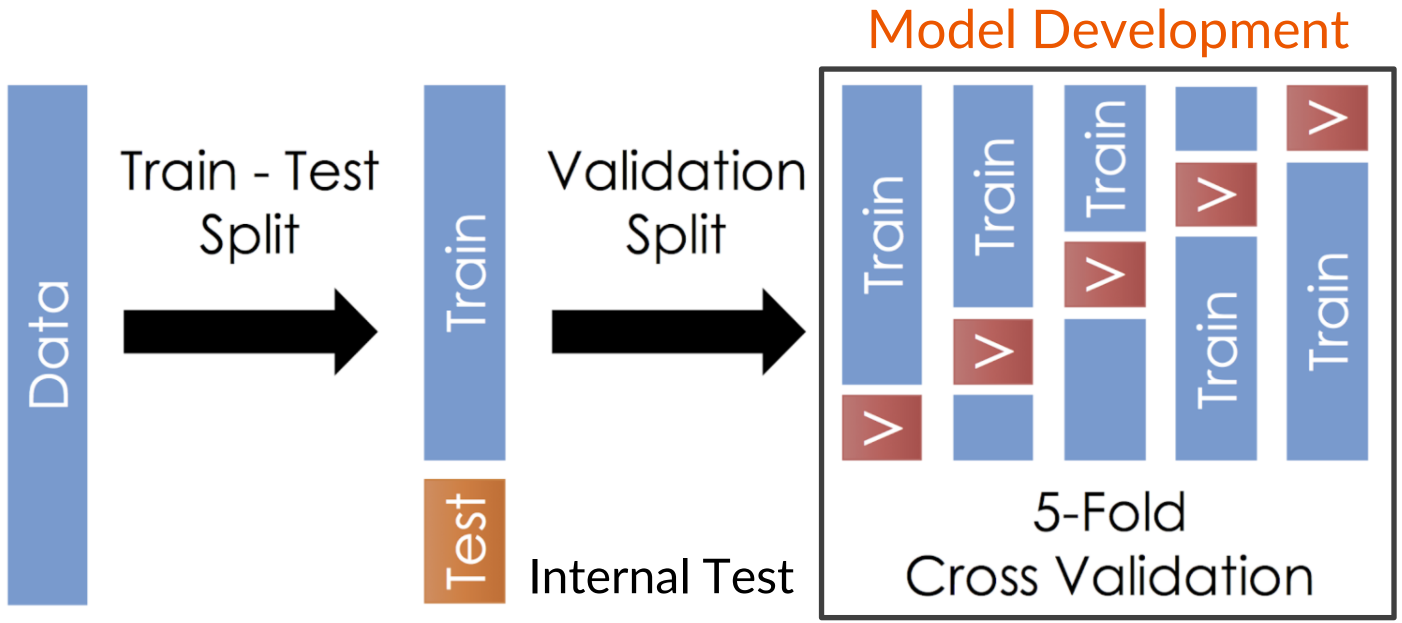
- C = inverse of regularization strength
- kernel, degree = set kernel for non-linear SVM
- epsilon = maximum regression error to not penalize

## Support vector regressor usage

```
from sklearn.svm import LinearSVR, SVR

## linear kernel
linearsvr = LinearSVR(C = 0.01, epsilon = 0.5, random_state = 25)
linearsvr.fit(input_data, label)
prediction = linearsvr.predict(new_data)
linearsvr.coef_

## non-linear kernel
svr = SVR(C = 0.01, kernel = ‘poly’, degree = 2, epsilon = 0.5, random_state = 25)
svr.fit(input_data, label)
prediction = svr.predict(new_data)

## get all support vectors (data points near decision boundary)
svr.support_vectors_
```

## Data splitting

## Train-Val-Test



Source: medium.com

## Data splitting tips

- **Test or no test:** can you dedicate enough sample to test set?

- Label distribution on validation and test set
  - Stratified versus 1:1

- Cross-validation versus bootstrapping
  - CV ensures that all samples will be used equally
  - Bootstrapping provides more control on label distribution

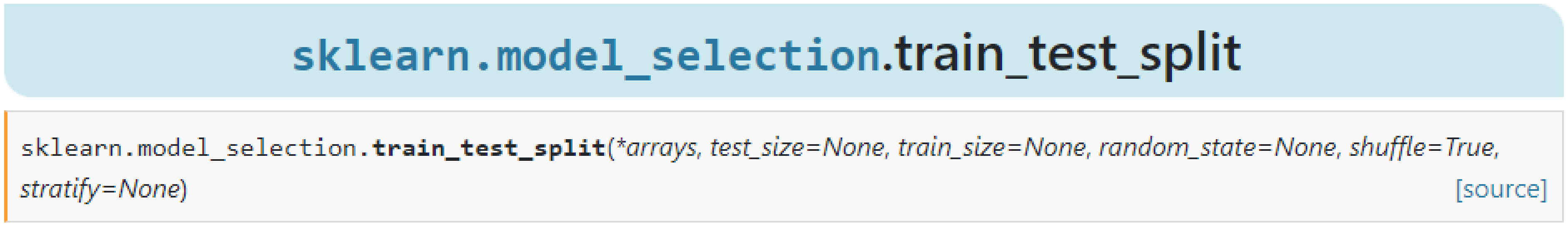
- Monitor performances on each split
  - Mean and SD

## Data splitting examples

- **Example 1**: 223 negative, 77 positive
  - **Test**: 31 negative, 27 positive
  - **Validation**: 25 negative, 25 positive
  - **Training**: 167 negative, 25 positive
  - Repeat training-validation split using bootstrapping

- **Example 2**: 48 negative, 23 positive
  - 2-fold cross-validation: 24 negative, 11 positive
  - Perform only logistic regression model
  - Limited to discussion of feature importance

## Data splitting in Python

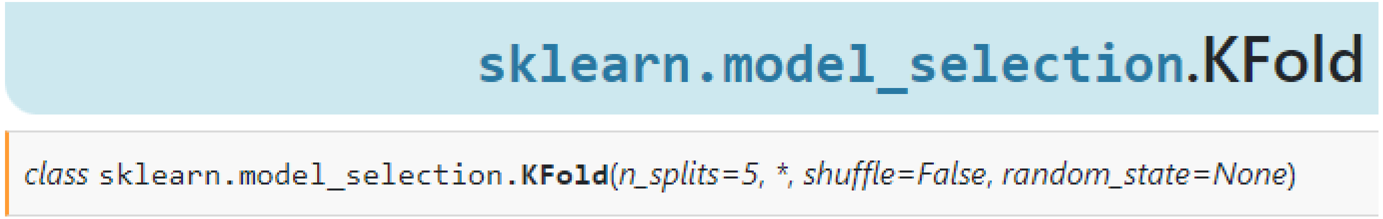


sklearn.model_selection.train_test_split

sklearn.model_selection.**train_test_split**(**arrays*, *test_size=None*, *train_size=None*, *random_state=None*, *shuffle=True*, *stratify=None*) [source]

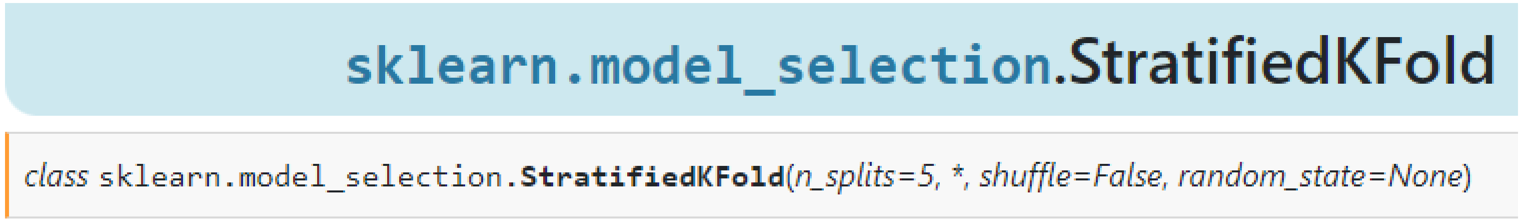
```
from sklearn.model_selection import train_test_split

## random split
X_train, X_test, y_train, y_test = train_test_split(input_data, label, test_size = 0.25,
random_state = 25)

## stratified split
X_train, X_test, y_train, y_test = train_test_split(input_data, label, test_size = 0.25,
stratify = label, random_state = 25)
```

## Cross-validation in Python



sklearn.model_selection.KFold

```
class sklearn.model_selection.KFold(n_splits=5, *, shuffle=False, random_state=None)
```



sklearn.model_selection.StratifiedKFold

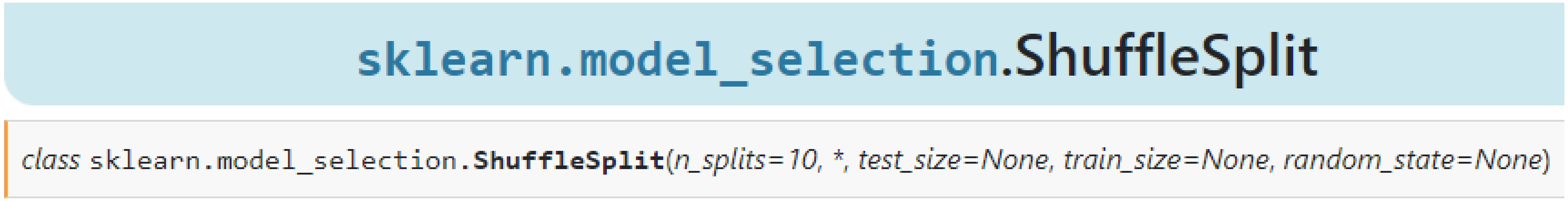
```
class sklearn.model_selection.StratifiedKFold(n_splits=5, *, shuffle=False, random_state=None)
```

- n_splits = number of folds
- Fixed random state and set shuffle = True (otherwise will get consecutive)
- Stratification will always be done based on the label (y)

## Cross-validation usage

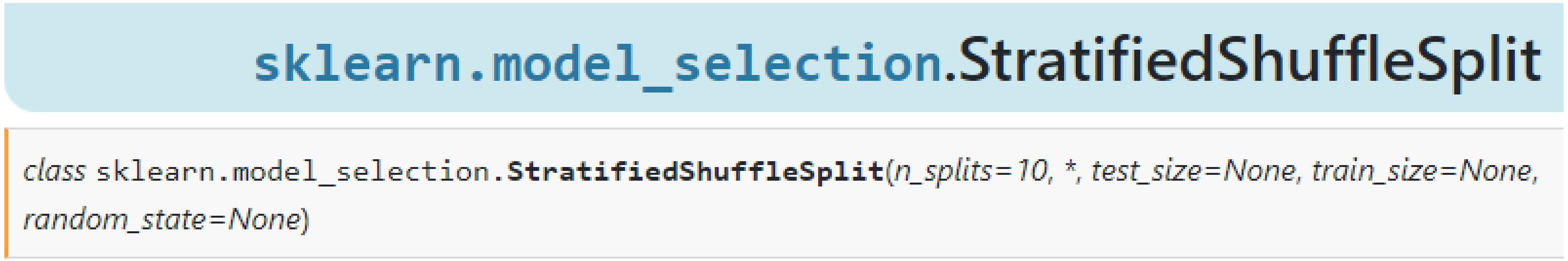
```
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits = 10, shuffle = True, random_state = 25)

## a fitted PCA can be used to transform any new data
for train_index, test_index in skf.split(input_data, label):
    X_train = input_data[train_index, :]
    y_train = label[train_index]

    X_test = input_data[test_index, :]
    y_test = label[test_index]

    ## model development
```

## Bootstrapping in Python



sklearn.model_selection.ShuffleSplit

```
class sklearn.model_selection.ShuffleSplit(n_splits=10, *, test_size=None, train_size=None, random_state=None)
```



sklearn.model_selection.StratifiedShuffleSplit

```
class sklearn.model_selection.StratifiedShuffleSplit(n_splits=10, *, test_size=None, train_size=None,
random_state=None)
```

- n_splits = number of bootstrap samples
- Controllable sizes of both train and test splits
- Stratification will be done based on the label (y)

## Bootstrapping usage

```
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(n_splits = 100, train_size = 0.5, test_size = 0.3,
random_state = 25)

## a fitted PCA can be used to transform any new data
for train_index, test_index in sss.split(input_data, label):
    X_train = input_data[train_index, :]
    y_train = label[train_index]

    X_test = input_data[test_index, :]
    y_test = label[test_index]

    ## model development
```

## Hyperparameter tuning

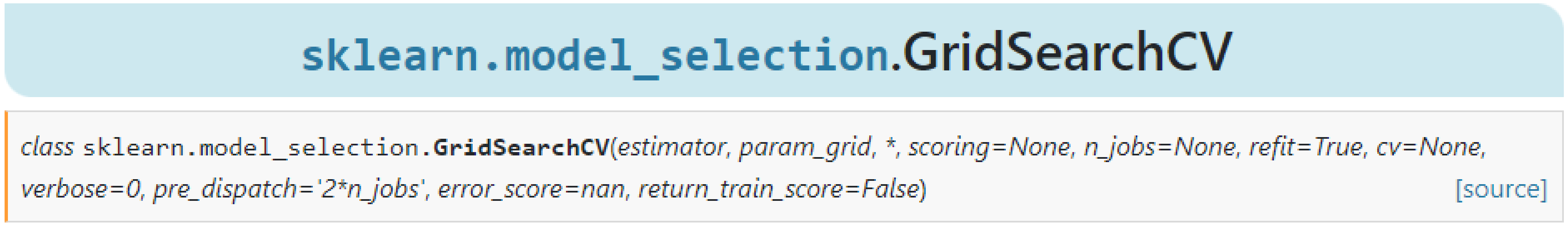
## Hyperparameter tuning tips

- Understand the impact of key parameters for each model family
  - Increase $\alpha$ in regularization $\alpha \sum b_i^2$ helps overfitting
  - Decrease $\alpha$ will instead help underfitting

- No need to tune aggressively
  - $\alpha \in \{0.0001,\ 0.01,\ 1,\ 100\}$
  - max_depth $\in \{5,\ 10,\ 20,\ \mathrm{None}\}$

- Monitor multiple metrics: AUROC, average precision, etc.
- Look for pattern in the result
  - Large enough $\alpha$ yields good performance

## Hyperparameter tuning in Python



### sklearn.model_selection.GridSearchCV

```
class sklearn.model_selection.GridSearchCV(estimator, param_grid, *, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan, return_train_score=False)
```

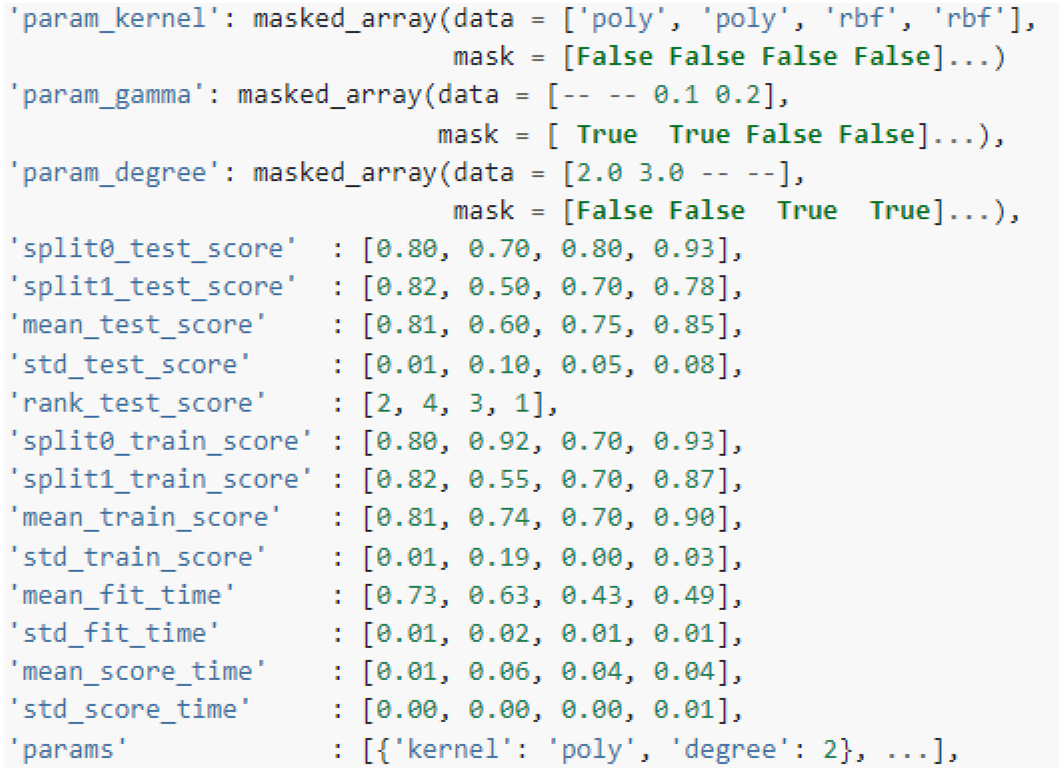
[source]

- estimator = number of directions/axes to produce
- param_grid = dictionary of hyperparameters to tune
- scoring = metrics to keep track of
- n_jobs = use multiple CPU to speed up the calculation
- refit = whether to train using the optimal hyperparameters
- cv = data splitting

# Hyperparameter tuning usage

```
## preparing data split, base model, hyperparameters
skf = StratifiedKFold(n_splits = 10, shuffle = True, random_state = 25)
logreg = LogisticRegression(solver = ‘liblinear’, max_iter = 1000, random_state = 25)
logreg_param = {‘C’: [0.0001, 0.01, 1, 100],
                ‘penalty’: [‘l1’, ‘l2’]}

## define and fit GridSearchCV
grid_logreg = GridSearchCV(logreg, logreg_param, scoring = [‘accuracy’, ‘roc_auc’,
‘average_precision’], n_jobs = 4, refit = ‘average_precision’, cv = skf)

grid_logreg.fit(input_data, label)

## dictionary containing full tuning results (performance scores, time spent, etc.)
grid_logreg.cv_results_
```

## GridSearchCV output



```
'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                             mask = [False False False False]...)
'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                            mask = [ True  True False False]...),
'param_degree': masked_array(data = [2.0 3.0 -- --],
                             mask = [False False  True  True]...),
'split0_test_score'  : [0.80, 0.70, 0.80, 0.93],
'split1_test_score'  : [0.82, 0.50, 0.70, 0.78],
'mean_test_score'    : [0.81, 0.60, 0.75, 0.85],
'std_test_score'     : [0.01, 0.10, 0.05, 0.08],
'rank_test_score'    : [2, 4, 3, 1],
'split0_train_score' : [0.80, 0.92, 0.70, 0.93],
'split1_train_score' : [0.82, 0.55, 0.70, 0.87],
'mean_train_score'   : [0.81, 0.74, 0.70, 0.90],
'std_train_score'    : [0.01, 0.19, 0.00, 0.03],
'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
'mean_score_time'    : [0.01, 0.06, 0.04, 0.04],
'std_score_time'     : [0.00, 0.00, 0.00, 0.01],
'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
```

## Any question?