Polynomial regression

How is polynomial regression model trained?

Step 1: Apply polynomial transformation on the feature matrix.

Step 2: Learn linear regression model (via normal equation or SGD) on the transformed feature matrix.

Implementation tips: Make use of pipeline construct for polynomial transformation followed by linear regression estimator.

Set up polynomial regression model with normal equation

```python
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

# Two steps:
# 1. Polynomial features of desired degree (here degree=2)
# 2. Linear regression
poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2))),
                ('linear_regression', LinearRegression())])

# Train with feature matrix X_train and label vector y_train
poly_model.fit(X_train, y_train)
```

Set up polynomial regression model with SGD

```python
from sklearn.linear_model import SGDRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2))),
                ('sgd_regression', SGDRegressor())])
poly_model.fit(X_train, y_train)
```

Notice that there is a single line code change in two code snippets.

How to use only interaction features for polynomial regression?

```python
from sklearn.preprocessing import PolynomialFeatures
poly_transform = PolynomialFeatures(degree=2, interaction_only=True)
```

$[x_1, x_2]$ is transformed to $[1, x_1, x_2, x_1 x_2]$.

Note that $[x_1^2, x_2^2]$ are excluded.

Hyperparameter tuning (HPT)

How to recognize hyperparameters in any sklearn estimator?

- Hyper-parameters are parameters that are not directly learnt within estimators.
- In `sklearn`, they are passed as arguments to the constructor of the estimator classes.

For example,

- degree in PolynomialFeatures
- learning rate in SGDRegressor

How to set these hyperparameters?

Select hyperparameters that results in the best cross validation scores.

Hyper parameter search consists of

- an estimator (regressor or classifier);
- a parameter space;
- a method for searching or sampling candidates;
- a cross-validation scheme; and
- a score function.

Two generic HPT approaches implemented in sklearn are:

- **GridSearchCV** exhaustively considers all parameter combinations for specified values.

```
param_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']}
 ]
```

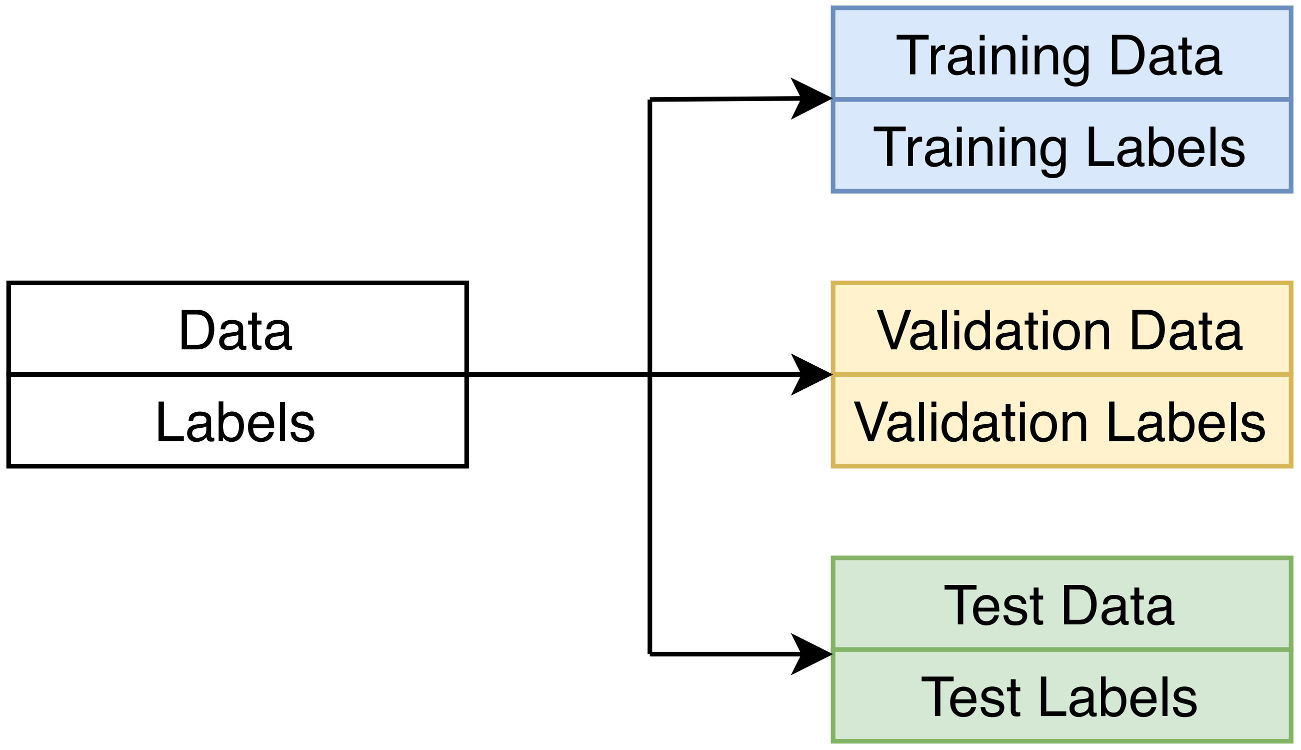
- **RandomizedSearchCV** samples a given number of candidate values from a parameter space with a specified distribution.

```
param_dist = {
     "average": [True, False],
     "l1_ratio": stats.uniform(0, 1),
     "alpha": loguniform(1e-4, 1e0),
}
```

What are the differences between grid and randomized search?

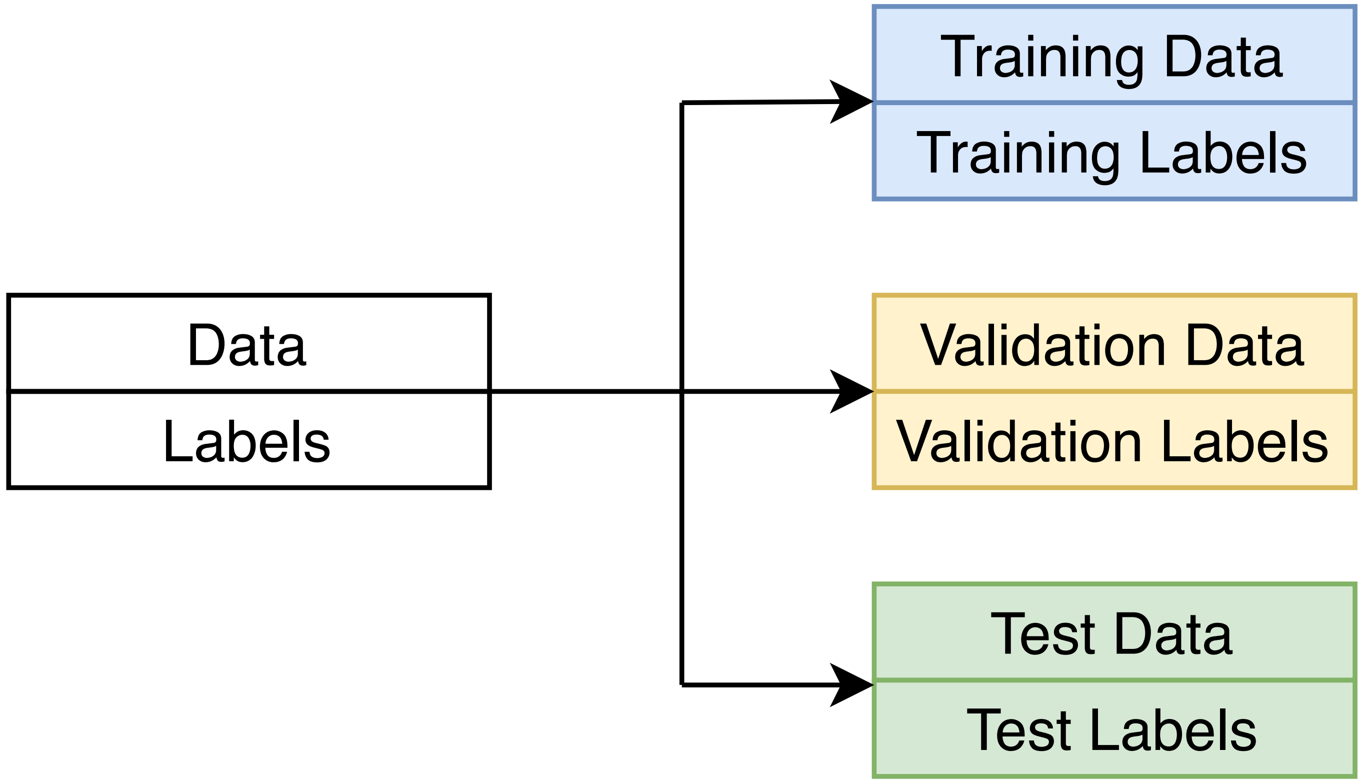
Grid search	Randomized search
Specifies exact values of parameters in grid	Specifies distributions of parameter values and values are sampled from those distributions.
	Computational budget can be chosen independent of number of parameters and their possible values.
	The budget is chosen in terms of the number of sampled candidates or the number of training iterations. Specified in n_iter argument

What data split is recommended for HPT?

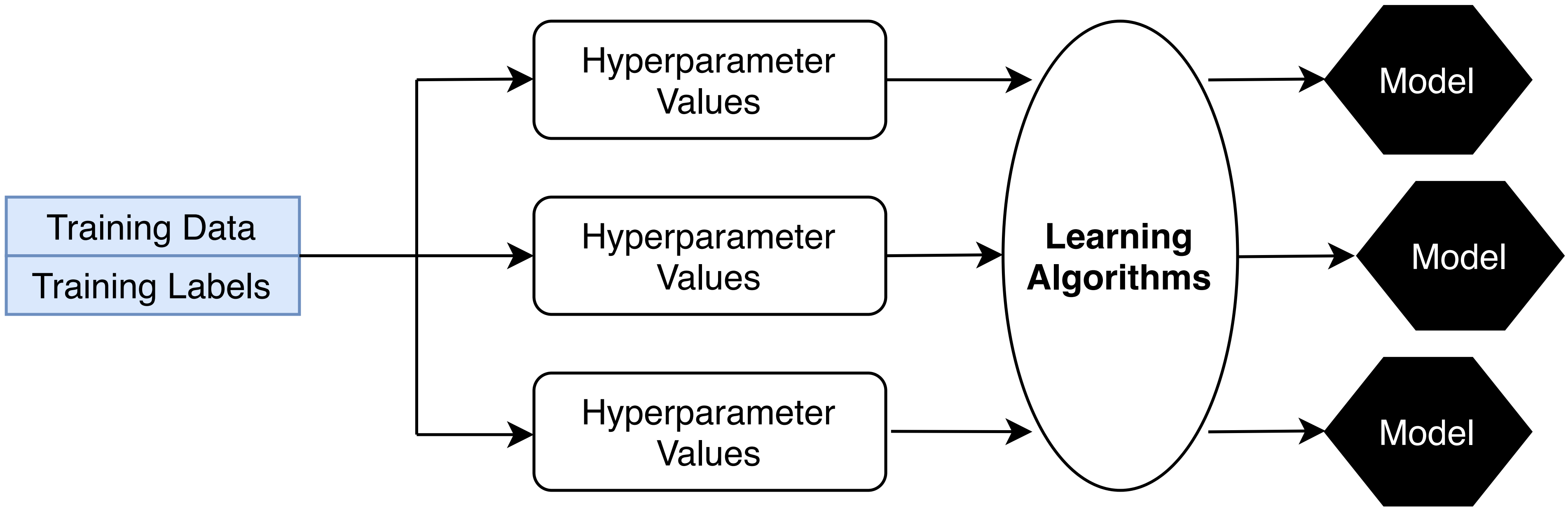


What are the steps in HPT?

STEP 1: Divide training data into training, validation and test sets.



STEP 2: For each combination of hyper-parameter values learn a model with training set.

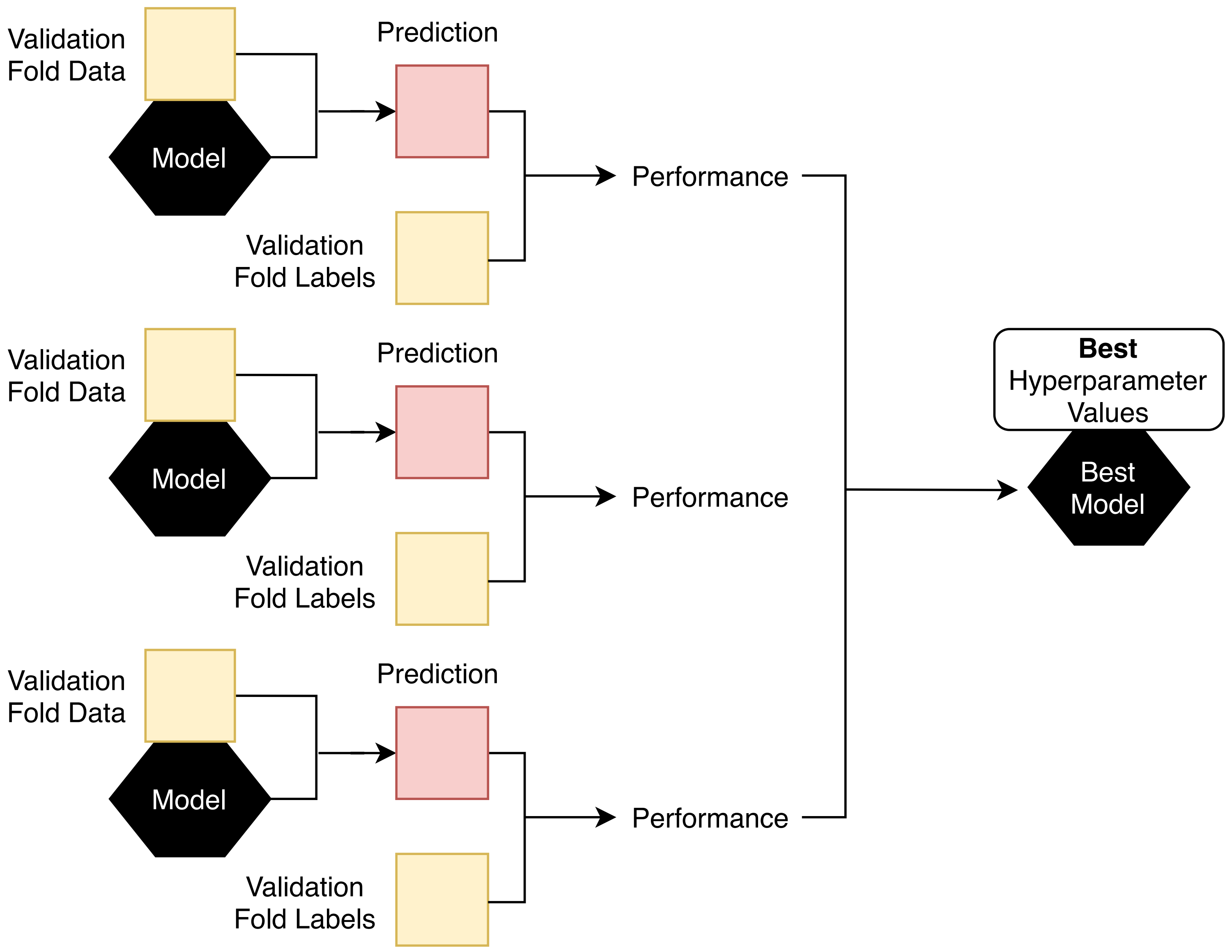


This step creates multiple models.

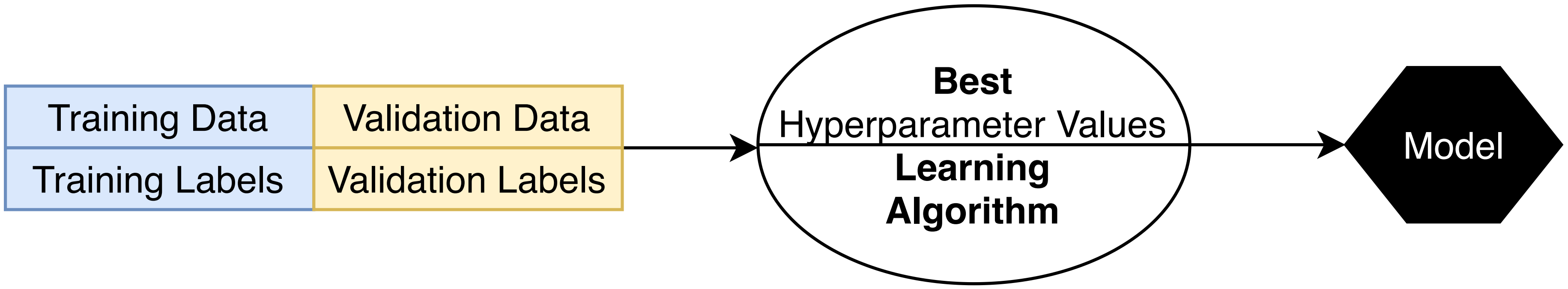
Tips

- This step can be be run in parallel by setting n_jobs = -1.
- Some parameter combinations may cause failure in fitting one or more folds of data. This may cause the search to fail. Set error_score = 0 (or np.NaN) to set score for the problematic fold to 0 and complete the search.

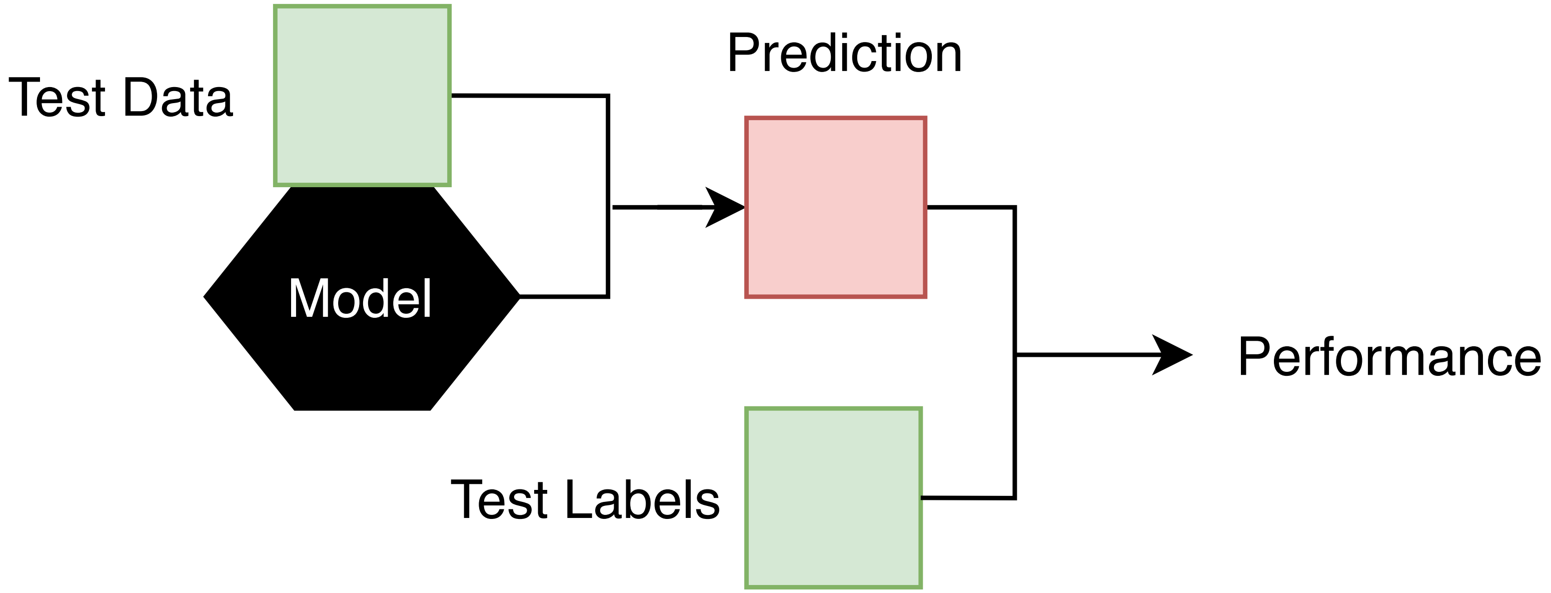
STEP 3: Evaluate performance of each model with validation set and select a model with the best evaluation score.



STEP 4: Retrain model with the best hyper-parameter settings on training and validation set combined.



STEP 5: Evaluate the model performance on the test set.



Note that the test set was not used in hyper-parameter search and model retraining .

What are some of model specific HPT available for regression tasks?

- Some models can fit data for a range of values of some parameter almost as efficiently as fitting the estimator for a single value of the parameter.
- This feature can be leveraged to perform more efficient cross-validation used for model selection of this parameter.

  - linear_model.LassoCV
  - linear_model.RidgeCV
  - linear_model.ElasticNetCV

How to determine degree of polynomial regression with grid search?

```python
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import POlynomialFeatures
from sklearn.linear_model import SGDRegressor

param_grid = [
    {'poly__degree': [2, 3, 4, 5, 6, 7, 8, 9]}
  ]

pipeline = Pipeline(steps=[('poly', PolynomialFeatures()),
                           ('sgd', SGDRegressor())])

grid_search = GridSearchCV(pipeline, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(x_train.reshape(-1, 1), y_train)
```

Regularization

How to perform ridge regularization with specific regularization rate?

[Option #1]

Step 1: Instantiate object of Ridge estimator

Step 2: Set parameter alpha to the required regularization rate.

```python
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1e-3)
```

fit, score, predict work exactly like other linear regression estimators

[Option #2]

Step 1: Instantiate object of SGDRegressor estimator

Step 2: Set parameter alpha to the required regularization rate and penalty = l2.

```python
from sklearn.linear_model import SGDRegressor
sgd = SGDRegressor(alpha=1e-3, penalty='l2')
```

How to search the best regularization parameter for ridge?

[Option #1]

Search for the best regularization rate with built-in cross validation in RidgeCV estimator.

[Option #2]

Use cross validation with Ridge or SVDRegressor to search for best regularization.

- Grid search
- Randomized search

How to perform ridge regularization in polynomial regression?

Set up a pipeline of polynomial transformation followed by the ridge regressor.

```python
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2)),
                ('ridge', Ridge(alpha=1e-3))])
poly_model.fit(X_train, y_train)
```

Instead of Ridge, we can use SGDRegressor, as shown on previous slide, to get equivalent formulation.

How to perform lasso regularization with specific regularization rate?

[Option #1]

Step 1: Instantiate object of Lasso estimator

Step 2: Set parameter alpha to the required regularization rate.

```python
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1e-3)
```

fit, score, predict work exactly like other linear regression estimators

[Option #2]

Step 1: Instantiate object of SGDRegressor estimator

Step 2: Set parameter alpha to the required regularization rate and penalty = l1.

```python
from sklearn.linear_model import SGDRegressor
sgd = SGDRegressor(alpha=1e-3, penalty='l1')
```

How to search the best regularization parameter for lasso regularization?

[Option #1]

Search for the best regularization rate with built-in cross validation in LassoCV estimator.

[Option #2]

Use cross validation with Lasso or SVDRegressor to search for best regularization.

- Grid search
- Randomized search

How to perform lasso regularization in polynomial regression?

Set up a pipeline of polynomial transformation followed by the lasso regressor.

```python
from sklearn.linear_model import Lasso
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2)),
                ('lasso', Lasso(alpha=1e-3))])
poly_model.fit(X_train, y_train)
```

Instead of Lasso, we can use SGDRegressor to get equivalent formulation.

How to perform both lasso and ridge regularization in polynomial regression?

Set up a pipeline of polynomial transformation followed by the SGDRegressor with `penalty = 'elasticnet'`

```python
from sklearn.linear_model import Lasso
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2)),
                ('elasticnet', SGDRegressor(penalty='elasticnet',
                                            l1_ratio=0.3))])
poly_model.fit(X_train, y_train)
```

Remember elasticnet is a convex combination of L1 (Lasso) and L2 (Ridge) regularization.

- In this example, we have set l1_ratio to 0.3, which means l2_ratio = 1- l1_ratio = 0.7. L2 takes higher weightage in this formulation.

Summary

How to implement

- Different regression models: standard linear regression and polynomial regression.
- Regularization.
- Model evaluation through different error metrics and scores derived from them.
- Cross validation - different iterators
- Hyperparameter tuning via grid search and randomized search.

Appendix - More Details

Introduction

In this module, we will be covering the implementation aspects of models of linear regression.

First we will learn linear regression models with:

- Normal equation, which estimates model parameter with a closed-form solution.

- Iterative optimization approach of gradient descent and its variants namely batch, mini-batch and stochastic gradient descent.

Further, we will study the implementation of the polynomial regression model, that is capable of modelling non-linear relationships between features and labels.

- Since the polynomial regression uses more parameters (due to polynomial representation of the input), it is more prone to overfitting.
- We will study how to detect overfitting with learning curves and use of regularization to mitigate the risk of overfitting.

Recap

Let's recall components of Linear regression

Training Data

(features, label) or (X, y), where label y is a real number.

Model

The label is obtained by a linear combination (or weighted sum) of the input features and a *bias* (or *intercept*) term. The model $h_{\mathbf{w}}$ is given by

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

where,

- $\hat{y}$ is the predicted value.
- $\mathbf{X}$ is a feature vector $\{x_1, x_2, \ldots, x_m\}$ for a given example with m features in total.
  - i-th feature: x_i
- Weight or parameter vector $\mathbf{w}$ includes bias term too: $\{w_0, w_1, w_2, \ldots, w_m\}$
- w_i is i-the model parameter associated with i-the feature.
- $h_{\mathbf{w}}$ is a model with parameter vector $\mathbf{w}$.

Loss function

The model parameters $\mathbf{w}$ are learnt such that the square of difference between the actual and the predicted values is minimized.

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} \left(\hat{y}^{(i)} - y^{(i)} \right)^2$$

$$J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} \left(\mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

Optimization

1. Normal equation
2. Iterative optimization with gradient descent: full batch, mini-batch or stochastic.

Evaluation measure

1. Mean squared error
2. Root mean squared error

Implementing with sklearn

Normal equation

sklearn provides LinearRegression estimator for weight vector estimation via normal equation

```python
from sklearn.linear_model import LinearRegression
```

As like other estimator object, it implements fit method that takes dataset as an input along with any other hyperparameters and returns estimated weight vector.

```python
lin_reg = LinearRegression(normalize=True)
lin_reg.fit(X_train, y_train)
```

It's a good practice to scale or normalize features. We can set normalize flag to True for normalizing the input features.

By default, normalize is False.

It also implements a couple of other methods:

- predict: Predicts label for a new examples based on the learnt model.
- score: Returns R^2 of the linear regression model.

Coefficient of determination R^2

$$\boxed{R^2 = \left(1 - \frac{u}{v}\right)}$$

where

- u is the residual sum of squares and is calculated as

$$u = (\mathbf{Xw} - \mathbf{y})^T(\mathbf{Xw} - \mathbf{y}\ )$$

- and v is the total sum of square. Let, $\hat{y}_{mean} = \frac{1}{n}\left(\mathbf{Xw}\right)$, then v is calculated as

$$v = (\mathbf{y} - \hat{\mathbf{y}}_{\mathbf{mean}})^T(\mathbf{y} - \hat{\mathbf{y}}_{\mathbf{mean}})$$

$$R^2 = \left(1 - \frac{u}{v}\right)$$

- The best possible score is 1.0.
- The score can be negative (because the model can be arbitrarily worse).
- A constant model that always predicts the expected value of y, disregarding the input features, would get a score of 0.0.

Model inspection

The learnt weights can be obtained by accessing the following class variables of LinearRegression estimator:

- The intercept weight w_0 *can be obtained via* `intercept_` class variable.

- The weights can be obtained via `coef_` class variable.

```
lin_reg.intercept_, lin_reg.coef_
```

Computational Complexity

- The normal equation uses the following equation to obtain

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

- This involves matrix inversion operation of feature matrix $\mathbf{X}$.

- The `LinearRegression` estimator uses SVD for this task and has the complexity of $O(m^2)$ where m is the number of features.

- This implies that if we double the number of features, the training computation grows roughly 4 times.
    - As the number of features grows large, the approach of normal equation slows down significantly.
    - These approaches are linear w.r.t. the number of training examples as long as the training set fits in the memory.
- The inference process is linear w.r.t. both the number of examples and number of features.

Weight vector estimation via SGD

- SGD is a simple yet very efficient approach of learning weight vectors in linear regression problems especially in large scale problem settings.
- SGD offers provisions for tuning the optimization process. However as a downside of this, we need to set a few hyperparameters.
- SGD is sensitive to feature scaling.

In sklearn, an estimator SGDRegressor implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties to fit linear regression models.

- SGDRegressor is well suited for regression problems with a large number of training samples (> 10,000). For learning problems with small number of training examples, sklearn user guide recommends Ridge or Lasso.

Key functionalities of SGDRegressor

Loss function:

- Can be set via the loss parameter.
- SGDRegressor supports the following loss functions.
    - `loss= 'squared error'`: Ordinary least squares,
    - `loss = 'huber'`: Huber loss for robust regression

- **Regularization**

SGD supports the following penalties:

- `Penalty = 'l2'` : L2 norm penalty on `coef_` . This is default setting.
- `penalty = 'l1'`: L1 norm penalty on `coef_`. This leads to sparse solutions.
- `penalty = 'elasticnet'`: Convex combination of L2 and L1;

`(1 - l1_ratio) * L2 + l1_ratio * L1

Learning rate

The learning rate η can be either constant or gradually decaying. There are following options for learning rate schedule specification in SGD:

1. invscaling

For regression the default learning rate schedule is inverse scaling `learning_rate = 'invscaling'`. The learning rate in t-th iteration or time step is calculated as

$$\eta^{(t)} = \frac{\eta_0}{t^{power\_t}}$$

where, η_0 and power_t are hyperparameters chosen by the user.

2. Constant

For a constant learning rate use `learning_rate = 'constant'` and use η_0 to specify the learning rate.

3. Adaptive

For an adaptively decreasing learning rate, use `learning_rate = 'adaptive'` and use η_0 to specify the starting learning rate.

- When the stopping criterion is reached, the learning rate is divided by 5, and the training loop continues.
- The algorithm stops when the learning rate goes below 10^{-6}.

4. Optimal

Used as a default setting for classification problems. The learning rate η for t-th iteration is calculated as follows:

$$\eta^{(t)} = \frac{1}{\alpha(t_0 + t)}$$

Here

- α is a regularization rate.
- t is the time step (there are a total of `n_samples*n_iter` time steps)
- t_0 is determined based on a heuristic proposed by Léon Bottou such that the expected initial updates are comparable with the expected size of the weights (this assuming that the norm of the training samples is approx. 1).

Stoping creteria

SGDRegressor provides two stopping criteria to stop the algorithm when a given level of convergence is reached:

1. With `early_stopping = True`
   - The input data is split into a training set and a validation set based on the `validation_fraction` parameter.
   - The model is fitted on the training set, and the stopping criterion is based on the prediction score (using the scoring method) computed on the validation set.

2. With `early_stopping = False`
    - The model is fitted on the entire input data and
    - The stopping criterion is based on the objective function computed on the training data.

- In both cases, the criterion is evaluated once by epoch, and the algorithm stops when the criterion does not improve `n_iter_no_change times` in a row.
- The improvement is evaluated with absolute tolerance `tol.`
- The algorithm stops in any case after a maximum number of iteration `max_iter`

SGD variation: Average SGD

SGDRegressor supports averaged SGD (ASGD). Averaging can be enabled by setting `average = True`

- ASGD performs the same updates as the regular SGD, and sets `coef_` attribute to the average value of the coefficients across all updates.
  - SGD sets `coef_` attribute to the last value of the coefficients (i.e. the values of the last update)

- The same process is followed for the `intercept_` attribute.

When using ASGD the learning rate can be larger and even constant, leading to a speed up in training.

Model inspection

We obtain the weight vector from the trained model as follows:

- `coef_` variable stores weights assigned to the features.
- `intercept_`, as name suggests, stores the intercept term.

Complexity

The major advantage of SGD is its efficiency, which is basically linear in the number of training examples.

If $\mathbf{X}$ is a matrix of size (n, m) training has a cost of $O(knp)$.

- where k is the number of iterations (epochs) and p is the average number of non-zero attributes per sample.

Recent theoretical results, however, show that the runtime to get some desired optimization accuracy does not increase as the training set size increases.

Polynomial regression

Polynomial regression

Polynomial regression = Polynomial transformation + Linear Regression

PolynomialFeatures transformer transforms an input data matrix into a new data matrix of a given degree.

Example:

```python
from sklearn.preprocessing import PolynomialFeatures
import numpy as np
X = np.arange(6).reshape(3, 2)
print ("Data matrix: \n", X)
poly = PolynomialFeatures(degree=2)
print ("\n\nAfter transformation: \n", poly.fit_transform(X))
```

Output:

```
Data matrix:
 [[0 1]
 [2 3]
 [4 5]]


After transformation:
 [[ 1.  0.  1.  0.  0.  1.]
 [ 1.  2.  3.  4.  6.  9.]
 [ 1.  4.  5. 16. 20. 25.]]
```

In the above example, the features of $\mathbf{X}$ have been transformed from $[x_1, x_2]$ to $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$, and can now be used within any linear model.

In some cases it's not necessary to include higher powers of any single feature, but only the so-called interaction features that multiply together as most distinct features. These can be gotten from 'PolynomialFeatures' with the setting `interaction_only = True`.

In this case, the features of $\mathbf{X}$ would be transformed from $[x_1, x_2]$ to $[1, x_1, x_2, x_1x_2]$.

Ridge regression

Ridge regression

Ridge regression minimizes L_2 penalized sum of squared error.

Ridge Loss = Sum of squared error + regularization_rate * penalty

- We use 'Ridge' estimator for implementing ridge regression. It takes parameter 'alpha' which is the regularization rate.
- 'RidgeCV' estimator implements ridge regression with cross validation for regularization rate.

RidgeCV parameters:

1. `alphas` is the list of regularization rates to try.
   - The regularization rate must be positive.
   - Larger values indicate stronger regularization.
2. `cv` determines the cross-validation splitting strategy.
   - `None`, to use the efficient Leave-One-Out cross-validation
   - `integer`, to specify the number of folds.
   - `CV splitter` specifies how to generate cross validation sets.
   - An iterable yielding (train, test) splits as arrays of indices.

In case of a binary or multiclass problems, for 'cv=None' or 'cv=5' (i.e. integer), 'StratifiedKFold' cross validation strategy is used. In other cases, 'KFold' cross validation strategy is used.

Model inspection

'RidgeCV' provides an additional output apart from usual outputs like `coef_` and `intercept_`:

- `alphas` provides the estimated regularization parameter.

Lasso regression

Lasso regression

Lasso uses $L1$ norm of weight vector as a penalty in linear regression loss function.

'sklearn' provides two implementations for learning weight vector in Lasso.

- 'Lasso' uses coordinated descent algorithm.
- 'LassoLars' uses least angle regression algorithm. It is adaption of forward stepwise feature selection for solving Lasso regression.