Introduction

In this module, we will be covering 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 and
- Iterative optimization approach of gradient descent and its variants namely batch, mini-batch and stochastic gradient descent. Here the model parameters are obtained by minimizing the loss function over training data.

Further we will study implementation of polynomial regression model, that is capable of modeling non-linear relationship between features and labels. It mainly transforms the input features with a polynomial transformation of certain degree and then uses the linear regression model.

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.

Let's start with essential imports.

```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

Recap

Let's recall the components of linear regression:

1. **Training data**: (features, label) or $(\mathbf{X}, y)$, where label y is a real number.

2. **Model**: The label is obtained by linear combination (or weighted sum) of the input features and a *bias* (or *intercept*) term.

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

   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
   - 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 $\mathbf{w}$.

3. **Loss function**: The model parameters $\mathbf{w}$ are learnt such the difference between the actual and the predicted values is minimized.

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

4. **Optimization**
   - Normal equation
   - Iterative optimization with gradient descent: full batch, mini-batch or stochastic

5. **Evaluation measure**
   - Mean squared error (MSE)
   - Root mean squared error (RMSE)

Implementation

```
#@title [Toy data for demonstration.]
# Randomly sample 1000 points.
num_samples = 1000
```

[Toy data for demonstration.]

```
X = 2 * np.random.rand(num_samples, 1)

# Obtain y = 4 + 3*x + noise.  Noise is randomly sampled.
y = 4 + 3 * X + np.random.randn(num_samples, 1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-3342f3588662> in <module>()
      2 # Randomly sample 1000 points.
      3 num_samples = 1000
----> 4 X = 2 * np.random.rand(num_samples, 1)
      5 
      6 # Obtain y = 4 + 3*x + noise.  Noise is randomly sampled.

NameError: name 'np' is not defined
```

SEARCH STACK OVERFLOW

```
#@title [Data exploration: Shapes]
print ("Shape of feature matrix:", X.shape)
print ("Shape of label vector:", y.shape)
```

[Data exploration: Shapes]

```
Shape of feature matrix: (1000, 1)
Shape of label vector: (1000, 1)
```

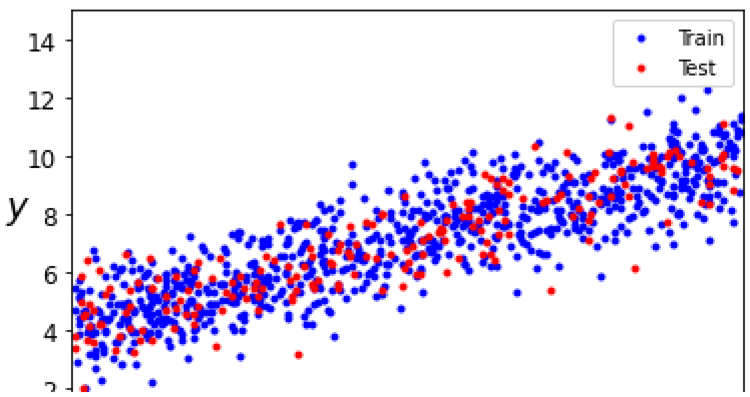
```
#@title [Preprocessing: Train-test split]
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

print ("Shape of training feature matrix:", X_train.shape)
print ("Shape of training label vector:", y_train.shape)
print ("Shape of test feature matrix:", X_test.shape)
print ("Shape of test label vector:", y_test.shape)
```

[Preprocessing: Train-test split]

```
Shape of training feature matrix: (800, 1)
Shape of training label vector: (800, 1)
Shape of test feature matrix: (200, 1)
Shape of test label vector: (200, 1)
```

```
#@title [Data exploration: Visualization]
plt.plot(X_train, y_train, "b.", label="Train")
plt.plot(X_test, y_test, "r.", label="Test")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.legend()
plt.show()
```

[Data exploration: Visualization]



Weight vector estimation via normal equation

Here we make use of estimator/predictor class `LinearRegression`. As you may recall from sklearn introduction session, the estimators have `fit` method that takes dataset as an input along with any other hyperparameters and returns estimated parameters.

```
from sklearn.linear_model import LinearRegression

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

```
    LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=True)
```

> It's a good practice to scale or normalize features. In case of this particular problem where we have a single feature, it does not matter. However, for the sake of uniformity, we set flag `normalize` to `True` for normalizing the input features.

> While using the `LinearRegression` class object for estimating the model parameters, we do not need to add a dummy column of 1's. It is automatically added by setting `fit_intercept` parameter in the constructor to `True` by default. In case, you do not want to learn the bias, you can choose to set it to `False`.

It's a good idea to check what more features `LinearRegression` estimator/predictor offers from its help and [sklearn pages on `LinearRegression`].

```
?LinearRegression
```

The following class methods are of our primary interests:

- `fit`: Fits linear model for a given training set.
- `predict`: Predicts label for a new example based on the learnt model.
- `score`: Returns R^2 of the linear regression model.

The coefficient of determination R^2 is defined as

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

where,

- u is the residual sum of squares and is calculated as
$$u = (\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})$$
- and v is the total sum of square. Let, $\hat{y}_{mean} = \frac{1}{n}(\mathbf{X}\mathbf{w})$, then v is calculated as
$$v = (\mathbf{y} - \hat{y}_{mean})^T(\mathbf{y} - \hat{y}_{mean})$$

Note that

- 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

- Since we generated data with the following equation: $y = 4 + 3x_1 + noise$, we know actual values of the parameters: $w_0 = 4$ and $w_1 = 3$.
- We can compare these values with the estimated values. Note that we do not have this luxury in real world problems, since we do not have access to the data generation process and its parameters.
- There we rely on the evaluation metrics to assess quality of the model.

> We can obtain them by accessing the following class variables of `LinearRegression` estimator:
>
> - The intercept weight w_0 can be obtained via `intercept_` class variable.
> - The weights corresponding to the features can be obtained via `coef_` class variable.

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

```
    (array([4.18985729]), array([[2.91204789]]))
```

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 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 or predictor `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`.

Let's look at some key functionalities of this estimator.

Loss function

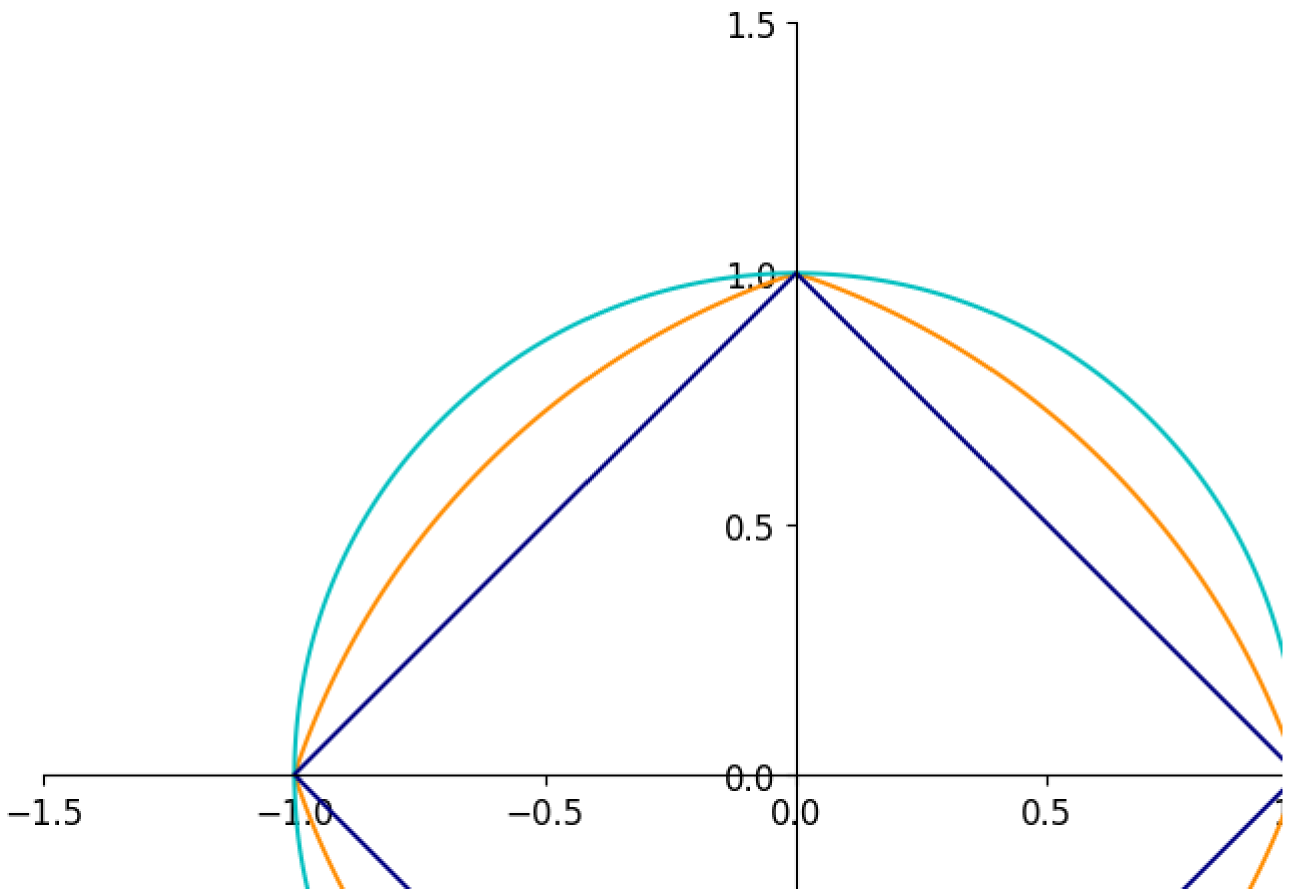
- Can be set via the loss parameter.
- `SGDRegressor` supports the following loss functions:
    - `loss="squared_error"` : Ordinary least squares,
    - `loss="huber"` : Huber loss for robust regression

> Based on what we have learnt so far, we will use `loss="squared_error"`.

```
#@title [Penalty visualization]
# This code is taken from [sklearn documentation](https://scikit-learn.org/stable/auto_exa
import numpy as np
import matplotlib.pyplot as plt
```

[Penalty visualization]

```
l1_color = "navy"
l2_color = "c"
elastic_net_color = "darkorange"

line = np.linspace(-1.5, 1.5, 1001)
xx, yy = np.meshgrid(line, line)

l2 = xx ** 2 + yy ** 2
l1 = np.abs(xx) + np.abs(yy)
rho = 0.5
elastic_net = rho * l1 + (1 - rho) * l2

plt.figure(figsize=(8, 8), dpi=100)
ax = plt.gca()

elastic_net_contour = plt.contour(xx, yy, elastic_net, levels=[1],
                                  colors=elastic_net_color)
l2_contour = plt.contour(xx, yy, l2, levels=[1], colors=l2_color)
l1_contour = plt.contour(xx, yy, l1, levels=[1], colors=l1_color)
ax.set_aspect("equal")
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_color('none')

plt.clabel(elastic_net_contour, inline=1, fontsize=18,
           fmt={1.0: 'elastic-net'}, manual=[(-1, -1)])
plt.clabel(l2_contour, inline=1, fontsize=18,
           fmt={1.0: 'L2'}, manual=[(-1, -1)])
plt.clabel(l1_contour, inline=1, fontsize=18,
           fmt={1.0: 'L1'}, manual=[(-1, -1)])

plt.tight_layout()
plt.show()
```



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`.

We have studied ridge regression (`penalty="l2"`) and lasso regression (`penalty="l1"`). The elastic net penalty is a combination of ridge and lasso penalties. It solves some deficiencies of the L1 penalty in the presence of highly correlated attributes. The parameter `l1_ratio` controls the convex combination of L1 and L2 penalty.

Learning rate

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

- **[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.

- [**constant**]For a constant learning rate use `learning_rate='constant'` and use η_0 to specify the learning rate.
- [**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 algorithm does not stop.
    - The algorithm stops when the learning rate goes below 1e-6.
- [**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).

Stopping criteria

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

- With `early_stopping=True`,
    - The input data is split into a training set and a validation set based on `validation_fraction` parameter.
    - The model is fitted on the training set, and the stopping criterion is based on the prediction score (using the `score` method) computed on the validation set.
- 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`, and the algorithm stops in any case after a maximum number of iteration `max_iter`.

A few more `SGDRegressor` parameters of our interest are as follows:

- `fit_intercept` : Whether the intercept should be estimated or not.
- `shuffle` : Whether or not the training data should be shuffled after each epoch.

SGD variations

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

- ASGD performs the same updates as the regular SGD, but instead of using the last value of the coefficients as the `coef_` attribute (i.e. the values of the last update), `coef_` is set instead to the average value of the coefficients across all updates.
- The same is done for the `intercept_` attribute.

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

Tip

> When using Gradient Descent, ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

Tip

When using Stochastic Gradient Descent, the training instances must be independent and identically distributed (IID), to ensure that the parameters get pulled towards the global optimum, on average.

> A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch).

If you do not do this, for example if the instances are sorted by label, then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

```
from sklearn.linear_model import SGDRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

# Always scale the input. The most convenient way is to use a pipeline.
reg = make_pipeline(StandardScaler(),
                    SGDRegressor())
reg.fit(X_train, y_train.ravel())
```

```
Pipeline(memory=None,
         steps=[('standardscaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('sgdregressor',
                 SGDRegressor(alpha=0.0001, average=False, early_stopping=False,
                              epsilon=0.1, eta0=0.01, fit_intercept=True,
                              l1_ratio=0.15, learning_rate='invscaling',
                              loss='squared_loss', max_iter=1000,
                              n_iter_no_change=5, penalty='l2', power_t=0.25,
                              random_state=None, shuffle=True, tol=0.001,
                              validation_fraction=0.1, verbose=0,
                              warm_start=False))],
         verbose=False)
```

It's a good idea to check what more feature `SGDRegressor` estimator/predictor offers from its help and [sklearn pages on `SGDRegression`]()

```
?SGDRegressor
```

The following class methods are of our primary interests:

- `fit` : Fits linear model for a given training set with SGD.
- `predict` : Predicts label for a new examples using the linear model.
- `score` : Returns coefficient of determination of the prediction.

Model inspection

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

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

```
reg[1].intercept_, reg[1].coef_
```

```
(array([7.07510998]), array([1.71136209]))
```

Additionally we can obtain information on the training run with the two indicators:

- `n_iter_` stores the actual number of SGD iterations before reaching the stopping criterion.
- `t_` stores the number of weight updates performed during the training. This is equal to `n_iter_ * n_samples`.

```
reg[1].n_iter_, reg[1].t_
```

```
(10, 8001.0)
```

Practical tips

- SGD is sensitive to feature scaling, so it is highly recommended to scale your data.
  - For example, scale each attribute on the input vector $\mathbf{X}$ to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1.
  - Note that the same scaling must be applied to the test vector to obtain meaningful results.
  - If your attributes have an intrinsic scale (e.g. word frequencies or indicator features) scaling is not needed.
- Finding a reasonable regularization term α is best done using automatic hyper-parameter search, e.g. `GridSearchCV` or `RandomizedSearchCV`, usually in the range `10.0**-np.arange(1,7)`.
- Empirically, `sklearn` team found that SGD converges after observing approximately 10^6 training samples. Thus, a reasonable first guess for the number of iterations is `max_iter = np.ceil(10**6/n)`, where `n` is the number of training examples.
- If you apply SGD to features extracted using PCA we found that it is often wise to scale the feature values by some constant c such that the average L2 norm of the training data equals one.
- `sklearn` documentation reports that averaged SGD works best with a larger number of features and a higher η_0

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 transformation + Linear Regression

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

```
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))
```

```
     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.]]
```

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 $[1, x_1, x_2, x_1x_2]$.

```
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, interaction_only=True)
print ("\n\nAfter transformation: \n", poly.fit_transform(X))
```

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


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

More information on `PolynomialFeatures` can be obtained from the help documentation:

```
?PolynomialFeatures
```

We can combine the polynomial transformation and linear regression through `Pipeline` construct:

```
import numpy as np

from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler

model = Pipeline([('poly', PolynomialFeatures(degree=3)),
                  ('linear', LinearRegression(fit_intercept=False))])

# fit to an order-3 polynomial data
x = np.arange(5)
y = 3 - 2 * x + x ** 2 - x ** 3

model = model.fit(x[:, np.newaxis], y)
model.named_steps['linear'].coef_
```

```
     array([ 3., -2.,  1., -1.])
```

The linear model trained on polynomial features is able to exactly recover the input polynomial coefficients.

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 has the following important parameters:

1. `alpha` which is the regularization rate. [Note that the notation is different from what we used in MLT. We use α for learning rate.]
2. It uses a variety of solvers. Refer to ['sklearn ridge regression documentation page']. We will use `auto` that chooses the solver automatically based on the type of the training data.

```
from sklearn.linear_model import Ridge
reg = Ridge(alpha=.5)
reg.fit([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
print ("Weight vector: ", reg.coef_)
print ("Intercept/bias: ", reg.intercept_)
```

```
     Weight vector:  [0.34545455 0.34545455]
     Intercept/bias:  0.13636363636363638
```

As we discussed in MLT, we need to select an appropriate regularization rate via cross validation.

> `RidgeCV` estimator implements ridge regression with cross validation for regularization rate.

```
# This is formatted as code
```

RidgeCV parameters

The following are some important parameters in `RidgeCV`

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. The options are:
    - `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.

For more details on parameters of `RidgeCV`, let's look at the documentation

```
?RidgeCV
```

```
    Object `RidgeCV` not found.
```

Example code snippet

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import RidgeCV
X, y = load_diabetes(return_X_y=True)
clf = RidgeCV(alphas=[1e-3, 1e-2, 1e-1, 1]).fit(X, y)
clf.score(X, y)
```

```
    0.5166287840315831
```

Model inspection

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

- alpha_ provides the estimated regularization parameter.

```
print ('Weight vector: ', clf.coef_)
print ('w[0]: ', clf.intercept_)
print ('Best regularization parameter: ', clf.alpha_)
```

```
Weight vector:  [  -7.19945679 -234.55293001  520.58313622  320.52335582 -380.6070656
   150.48375154  -78.59123221  130.31305868  592.34958662   71.1337681 ]
w[0]:  152.13348416289645
Best regularization parameter:  0.01
```

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:

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

`Lasso`

Let's first look at the documentation of `Lasso`

```
from sklearn.linear_model import Lasso
?Lasso
```

Some important parameters are:

- `alpha` is the regularization rate whose default value is 1.
- `selection` can be chosen from {`cyclic`, `random`}.
  - `cyclic` is the default choice - it updates weights by iterating over features sequentially in every iteration.
  - `random` selection updates a random weight in every iteration. This leads to significantly faster convergence especially when `tol` is higher than 1e-4.

It's output is same as other regression methods like `LinearRegression` or `Ridge`.

`LassoLars`

Now let's look at the documentation of `LarsLasso`

```
from sklearn.linear_model import LassoLars
?LassoLars
```

It has `alpha` parameter for setting the regularization rate apart from other parameters like `fit_intercept`.

```
from sklearn import linear_model
reg = linear_model.LassoLars(alpha=0.01, normalize=False)
reg.fit([[-1, 1], [0, 0], [1, 1]], [-1, 0, -1])
LassoLars(alpha=0.01, normalize=False)
print(reg.coef_)
print(reg.alphas_)
print(reg.active_)
print (reg.coef_path_)
```

```
     [ 0.    -0.955]
     [0.22222222 0.01      ]
     [1]
     [[ 0.     0.   ]
      [ 0.    -0.955]]
```

The values of the regularization parameter is found via:

1. `LassoCV`
2. `LassoLarsCV`

How to perform model selection?

- CV strategies
- Learning curves