Data Representation in Scikit-Learn

Machine learning is about creating models from data. It is important then to understand how data can be represented in order to be understood by the computer. The best way to think about data within Scikit-Learn is in terms of tables of data.

Data as table

A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements.

Features matrix

The table layout makes the information to be represented as a two-dimensional numerical array or matrix, which is called the *features matrix*. By convention, this features matrix is often stored in a variable named X and is assumed to be two-dimensional, with shape [n_samples, n_features], and is most often contained in a NumPy array or a Pandas DataFrame. The samples (rows) always refer to the individual objects described by the dataset. The sample might be, a person, a document, an image, a sound file, a video, or anything else that can be described with a set of quantitative measurements.
The features (columns) refer to the distinct observations that describe each sample in a quantitative manner. Features are generally real-valued, but may be Boolean or discrete-valued in some cases.

Target array

In addition to the feature matrix X, you also need to work with a *label* or *target* array that by convention is called *y*. This target array is usually one dimensional, with length n_samples, and can be contained in a NumPy array or Pandas Series. It may have continuous numerical values, or discrete classes/labels. While some Scikit-Learn estimators do handle multiple target values in the form of a two-dimensional target array, you will be working with the a one-dimensional target array.

Scikit-Learn's Estimator API

Every machine learning algorithm in Scikit-Learn is implemented via the Estimator API, which provides a consistent interface for a wide range of machine learning applications.

Basics of the API

The following are the steps commonly used in the Scikit-Learn estimator API:

i). Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
ii). Arrange data into a features matrix and target vector
iii). Choose model hyperparameters by instantiating the class with desired values.
iv). Fit the model to your data by calling the *fit()* method of the model instance.
v). Apply the model to new data. For supervised learning, you predict labels for unknown data using the predict() method.
vi). Check the results of model fitting to know whether the model is satisfactory

Simple linear regression

Linear regression involves coming up with a straight-line fit to data. A straight-line fit is a model of the form $y = ax + b$ where a is commonly known as the *slope*, and b is commonly known as the *intercept*. The simplest model is the one that uses ***ordinary least squares*** (OLS), to finds the parameters w and b that minimize the *mean squared error* between predictions and the true regression targets, y, on the training set. The mean squared error is the sum of the squared differences between the predictions and the true values. Therefore this model is sometimes known as *ordinary least squares*

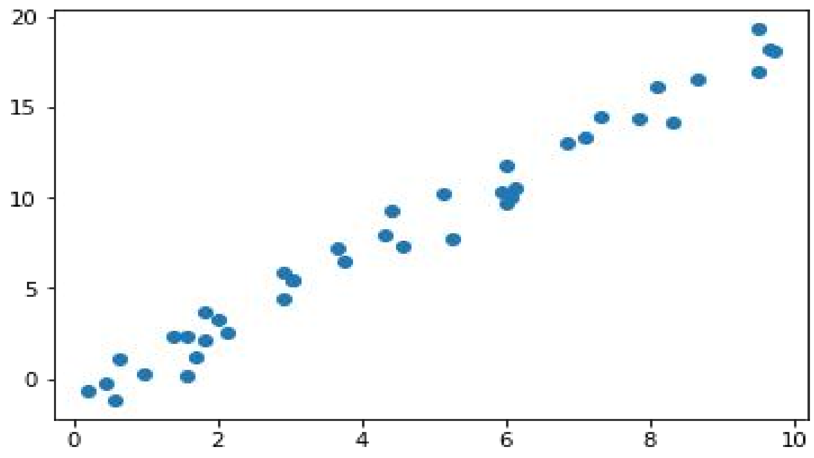
Example

Consider a simple linear regression that involves fitting a line to x, y data. Use the following data sample:

```
In[1]: import matplotlib.pyplot as plt
```

```
import numpy as np
rng = np.random.RandomState(42)
x = 10 * rng.rand(40)
y = 2 * x - 1 + rng.randn(40)
plt.scatter(x, y);
```

```
Out[1]:
```



With this data, you can use the steps outlined for the Scikit-Learn estimator API above:

i). **Choose a class of model.**
In Scikit-Learn, every class of model is represented by a Python class. E.g., for a simple linear regression model, you import the linear regression class as shown below:

```
In[2]: from sklearn.linear_model import LinearRegression
```

ii). **Arrange data into a features matrix and target vector.**
Scikit-Learn data representation, requires a two-dimensional features matrix and a one-dimensional target array. The target variable y is in the correct form, but you need to reshape data x into two-dimensional array i.e. make it a matrix of size [n_samples, n_features].

```
In[4]: X = x[:, np.newaxis]
X.shape
Out[4]: (40, 1)
```

iii). **Choose model hyper parameters.**
Depending on the model class you are working with, you might need to consider one or more of the following options:

- Would you like the model to be normalized? Use **`normalize`** (False by default) that decides whether to normalize the input variables (True) or not (False).
- Would we like to pre-process the features to add model flexibility?
- What degree of regularization would we like to use in the model?

These are important choices that must be made *once the model class is selected*. These choices are often represented as *hyperparameters*, or parameters that must be set before the model is fit to data. This is done when you choose hyperparameters by passing values at model instantiation. For instance, you can specify you would like to fit the intercept using the fit_intercept hyperparameter:

```
In[3]: model = LinearRegression(fit_intercept=True)
Out[3]: LinearRegression()
```

iv). Fit the model to your data.

Apply the model to data. This can be done with the fit() method of the model as follows:

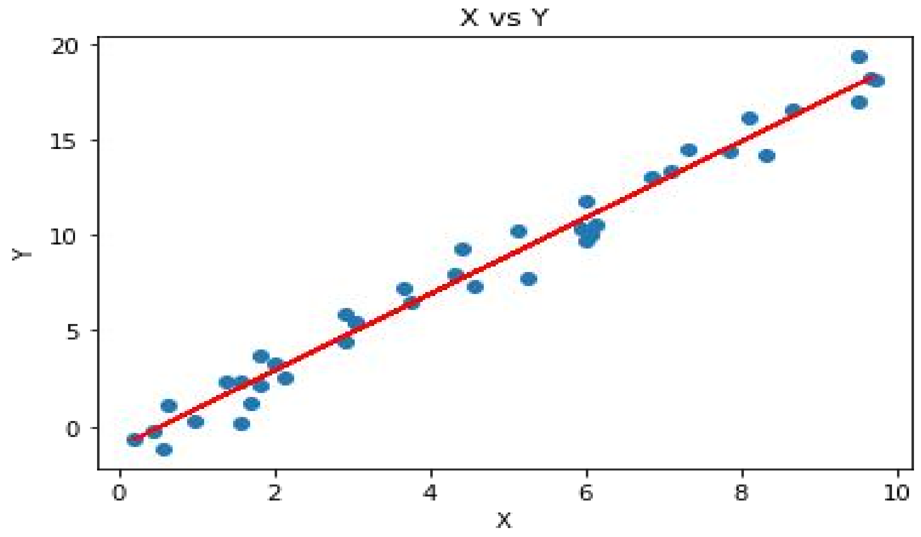
```
In[5]: model.fit(X, y)
Out[5]:
LinearRegression()
```

The fit() command causes a number of model-dependent internal computations to take place, and the results of these computations are stored in model specific attributes. In this linear model, we have the following:

```
In[6]: model.coef_
Out[6]: array([2.0133901])

In[7]: model.intercept_
Out[7]: -1.139509001948376
```

These two parameters represent the slope and intercept of the simple linear fit to the data.
Based on the two parameters, you can visualize the results by plotting the best fit line against the raw data as shown below:

```
In[8]: plt.title("X vs Y")
plt.scatter(x, y);
plt.plot(x, (model.coef_*x)+(model.intercept_),color='red')
plt.xlabel("X")
plt.ylabel("Y")

Out[8]:
```



v). Predict labels for unknown data.

Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In Scikit-Learn, this can done by using the predict() method. In this example, the "new data" will be a grid of *x* values, and we will need to know what *y* values the model predicts:

```
In[9]: xfit = np.linspace(-1, 11, 40)
```

Change the *x* values into a [n_samples, n_features] features matrix format and which can be feed to the model as shown below:

```
In[10]: Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
```

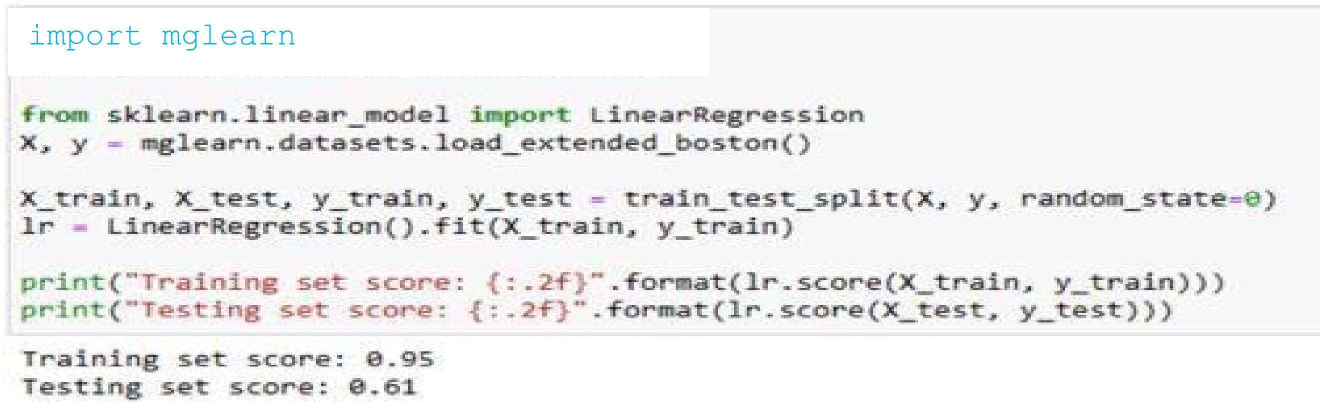
vi). Check the results of model fitting to know whether the model is satisfactory
Typically one evaluates the efficacy of the model by comparing its results to some known baseline.

Model Performance (Optimization)
The Goodness of fit determines how the line of regression fits the set of observations. This is achieved by using *R-squared method (R^2),* which is a statistical method that determines the goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of explained variance. It measures the strength of the relationship between the dependent and independent variables on a scale of 0-100%. It is computed by using the score()function.

```
In[11]:
r_sq = model.score(X, y)
print('coefficient of determination:', r_sq)

Out[11]:
coefficient of determination: 0.9780949186629684
```

Notice that linear models perform better with higher-dimensional datasets which reduces the effect of underfitting but exposes it to overfitting as shown in the following example

Example 2
Create a model using the extended Boston dataset and test the accuracy of training and testing dataset



```
import mglearn

from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)

print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Testing set score: {:.2f}".format(lr.score(X_test, y_test)))

Training set score: 0.95
Testing set score: 0.61
```

Although the training scores are high, the model performs poorly on the test set, a clear sign of overfitting. We have limited control over this because they do not have parameters that we can adjust. One of the most commonly used alternatives to standard linear regression is *ridge regression*.

Ridge Regression
The formula it uses to make predictions is the same one used for ordinary least squares (OLS). However, in ridge regression, the coefficients (*w*) have an additional goal other than accurate predictions. We want each feature to have little effect on the outcome ($\Delta Y \approx 0$) hence gradient (*w*) should be as small as possible. This constraint is an example of what is called *regularization* and is useful in restricting a model to avoid overfitting. The particular kind used by ridge regression is known as L2 regularization and gives us a parameter that we can control.
Generally, the Training and Test scores converge more for Ridge Regression compared to Linear Regression hence reducing overfitting. Generally, it is better to have lower training accuracy and better test accuracy for generalization.

Example
The following is an example of Ridge model creation. Three models are created with different values of alpha and in each case the model accuracy is tested

```
# Ridge Regression for Boston dataset

from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge.score(X_train, y_train)))
print("Testing set score: {:.2f}".format(ridge.score(X_test, y_test)))
```

```
Training set score: 0.89
Testing set score: 0.75
```

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Testing set score: {:.2f}".format(ridge10.score(X_test, y_test)))
```

```
Training set score: 0.79
Testing set score: 0.64
```

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Testing set score: {:.2f}".format(ridge01.score(X_test, y_test)))
```

```
Training set score: 0.93
Testing set score: 0.77
```

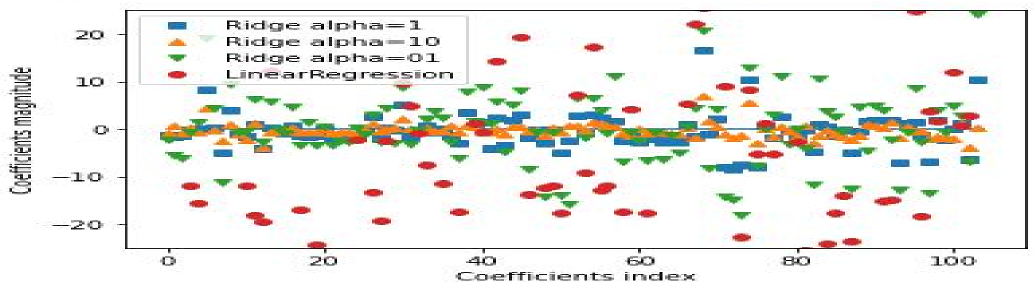
From the example, we can see that the alpha parameter allows us to control the gradient thus increasing the training set performance.

NB: The best alpha setting depends on the particular dataset we are using. Although Ridge regression usually tests better, with enough data points, linear regression can also perform well. With sufficient training data, regularization becomes less important and the performance of both methods becomes similar.

Ridge Regularization Impact
To analyse how regularization impacts magnitude of coefficients for the different algorithms (Linear regression, and Ridge) we can plot them. Here, the x-axis enumerates the different features while the y-axis shows the magnitude of the coefficients, coef_. For a large alpha, (alpha=10) the coefficients have small magnitudes while small ones have very large magnitudes. The best is option is the one that approaches zero without reverting to zero for most, hence alpha=0.1. Linear regression which has no regularization has magnitudes so large that they are outside of the chart.

```
plt.plot(ridge.coef_, 's', label="Ridge alpha=1")
plt.plot(ridge10.coef_, '^', label="Ridge alpha=10")
plt.plot(ridge01.coef_, 'v', label="Ridge alpha=01")

plt.plot(lr.coef_,'o', label="LinearRegression")
plt.xlabel("Coefficients index")
plt.ylabel("Coefficients magnitude")
plt.hlines(0,0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x27a5ce732e0>
```



NB: the magnitude of the coefficients never becomes zero with L2 regularization (Euclidean Distance Approach). Hence all features contribute to the prediction

Lasso Regression (Least Absolute Shrinkage and Selection Operator)
Like Ridge it restricts coefficients to be close to zero, using L1 regularization, which penalizes the sum of the absolute values of the coefficients. In this approach some coefficients can become *zero* hence their features are entirely ignored by the model. In this way it behaves as a form of automatic feature selection and can be used to reveal the most important features of the model.
Applying it to the to the extended Boston Housing dataset, the following observations can be made:

- The training and test sets are poor before regularization (underfitting), It only used 4 of the 105 features with the default alpha=1.0.
- To reduce underfitting, we introduce a small alpha (the maximum number of iterations have to be increased when you do this, otherwise the lines will fail to converge because the training stopped to early). When we do this more features are selected hence increasing performance.
- However, if we set alpha too low, we remove the effect of regularization (behaviour similar to LinearRegression) and end up overfitting.

```
# Lasso for Boston dataset
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Testing set score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))
```

```
Training set score: 0.29
Testing set score: 0.21
Number of features used: 4
```

```
#Lets change the alpha
#This requires us to increase the number of iterations too

lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Testing set score: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso001.coef_ != 0)))
```

```
Training set score: 0.90
Testing set score: 0.77
Number of features used: 33
```

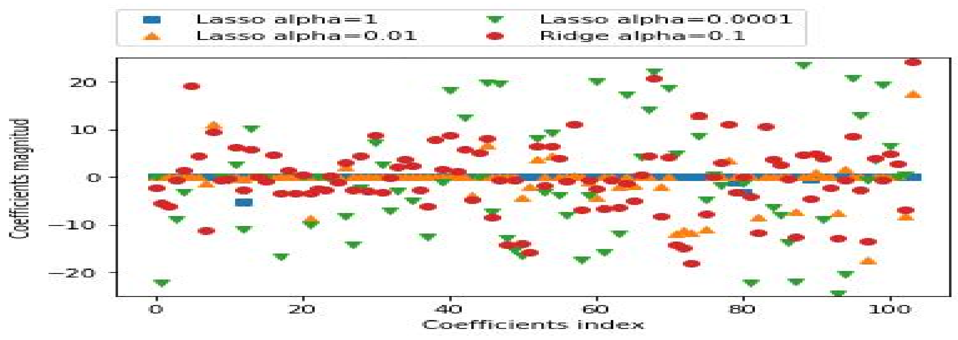
```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Testing set score: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso00001.coef_ != 0)))
```

```
Training set score: 0.95
Testing set score: 0.64
Number of features used: 96
```

Lasso Regularization Impact
For alpha=1, most of the coefficients are zero (not ideal). However, ddecreasing alpha to 0.01, increases the magnitude hence offers the best option of the three as shown below:

```
plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0001")

plt.plot(ridge01.coef_,'o', label="Ridge alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Coefficients index")
plt.ylabel("Coefficients magnitud")
```

```
Text(0, 0.5, 'Coefficients magnitud')
```



For comparison, the best Ridge solution is included. We note that for the best Ridge model (alpha=0.1) the coefficient magnitudes are large (non zero) unlike for the best lasso model (alpha=0.01) hence the lasso is suitable for feature selection. Recall the accuracy levels for the two were similar hence only feature importance would lead you to select the later.

Linear Classification Models
Linear models are also used for classification. Predications are made using the following formula:

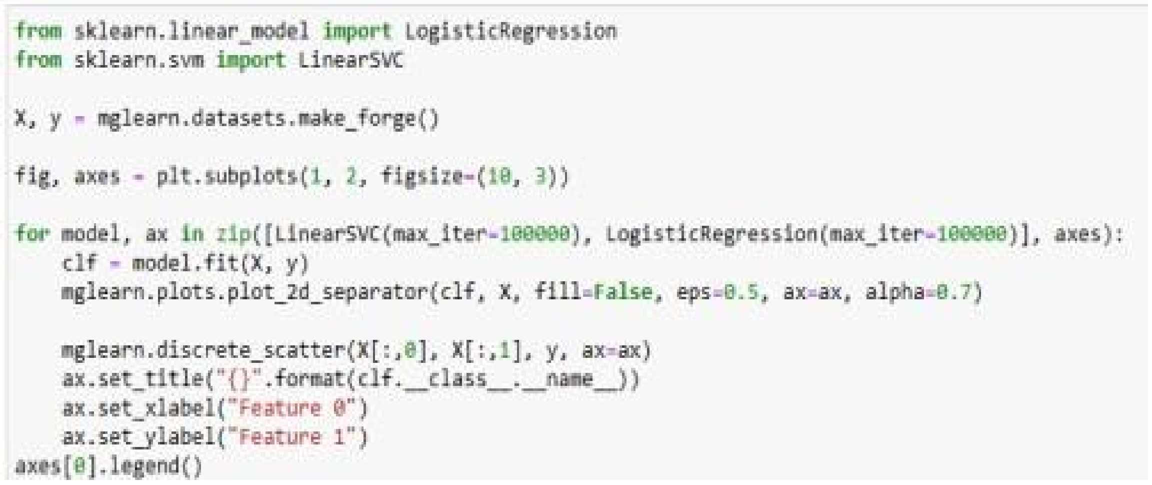
$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$$

Anything above 0 is predicted as a positive class (+1) while any value below 0 is predicted as a negative class (-1). In this way it acts a binary classification. The end result is prediction using two classes called linear binary classification

Linear Binary Classification (LBC)
There are many algorithms for linear binary classification models distinguished how they measure how well coefficients fit training data and the kind of regularization used. The two most common linear binary classification algorithms are:

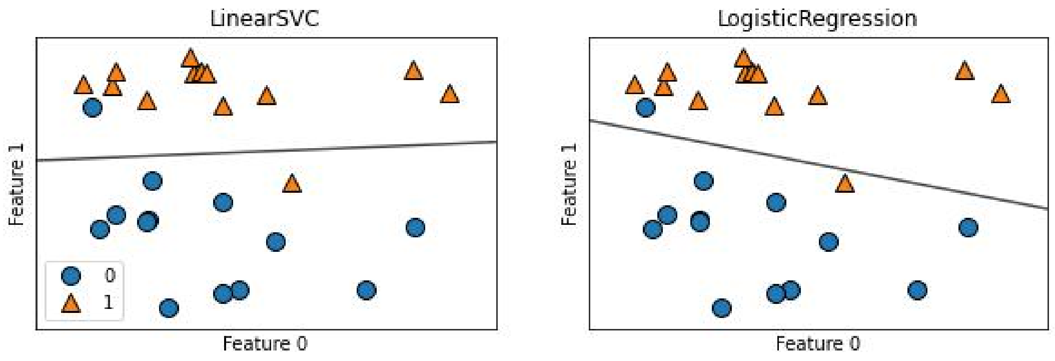
i) *Logistic Regression*,
ii) *Linear Support Vector Classification (SVC)*

When you apply these two algorithms forge dataset and you include a decision boundary for ease of analysis, you will observe that the two models have similar decision boundaries (both misclassify two data points).



```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(max_iter=100000), LogisticRegression(max_iter=100000)], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5, ax=ax, alpha=0.7)

    mglearn.discrete_scatter(X[:,0], X[:,1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
axes[0].legend()
```

Output



LBC Regularization Impact

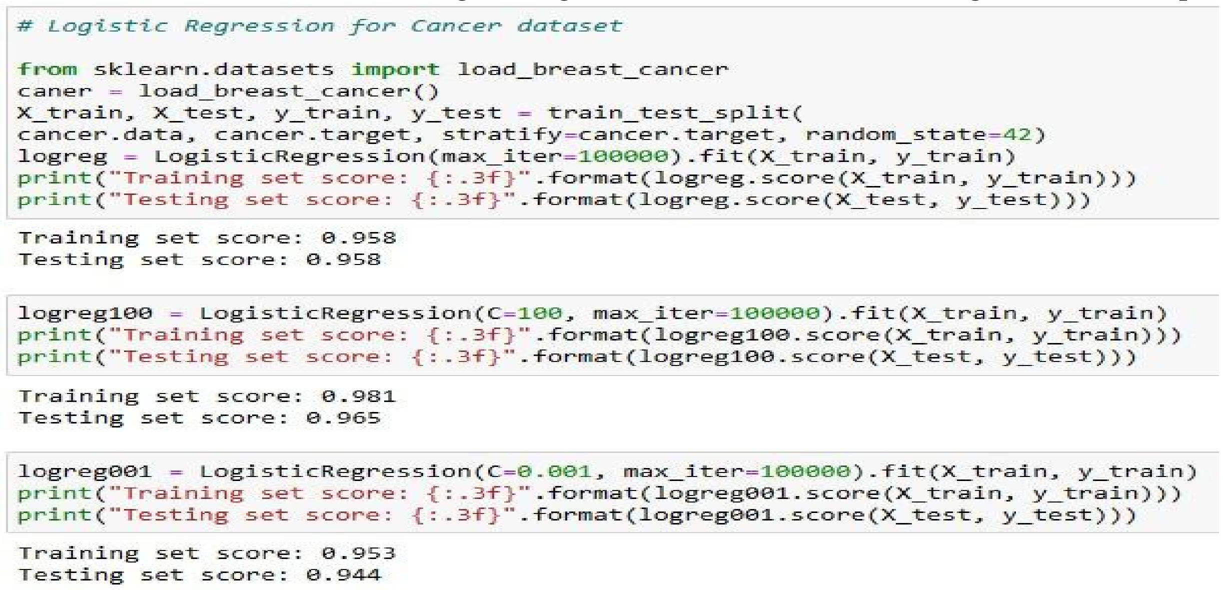
The trade-off parameter that determines the strength of the regularization is called C, and controls

(i) the size of the coefficients w, b (low C reduces the magnitude)
(ii) the accuracy of training sets (high C tries to fit the training set as best as possible).

Just like linear regression models, linear classification models perform better with higher dimension data by avoiding overfitting.

Example

Use the breast cancer dataset and logistic regression to show how controlling the C value impacts regularization



```
# Logistic Regression for Cancer dataset

from sklearn.datasets import load_breast_cancer
caner = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression(max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Testing set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

```
Training set score: 0.958
Testing set score: 0.958
```

```
logreg100 = LogisticRegression(C=100, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Testing set score: {:.3f}".format(logreg100.score(X_test, y_test)))
```

```
Training set score: 0.981
Testing set score: 0.965
```

```
logreg001 = LogisticRegression(C=0.001, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Testing set score: {:.3f}".format(logreg001.score(X_test, y_test)))
```

```
Training set score: 0.953
Testing set score: 0.944
```

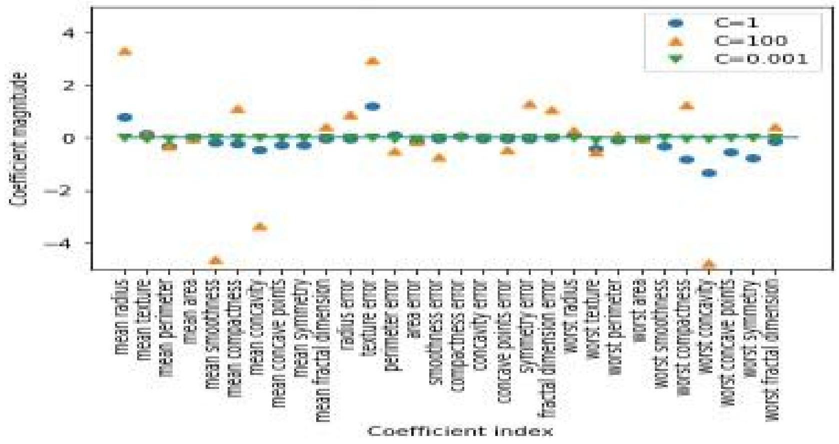
The default value of C=1 provides quite good performance, with 95% accuracy on both the training and the test set. But as training and test set performance are very close, it is likely that we have a simple model (it is using few features – thus results are the same). As we increase C (=100) the accuracy starts to differ.

Reducing the c value (C=0.001) on the other hand decreases both training and test set accuracy (possibly underfitting).

Plotting the coefficient magnitudes for the different models reveals that the default C=1 gives the best model coefficients. Stronger regularization (C=0.001) pushes coefficients more and more toward zero

```
plt.plot(logreg.coef_.T, 'o', label='C=1')
plt.plot(logreg100.coef_.T, '^', label='C=100')
plt.plot(logreg001.coef_.T, 'v', label='C=0.001')
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x27a5e217310>
```



Linear Multiclass Classification Models

These are extended from linear binary classification and a common technique is the *one-vs.-rest* approach. Here, a binary model is built for each class resulting in as many binary models as there are classes.
To make a prediction, all binary classifiers are run on a class label and the classifier that has the highest score is returned as the prediction.