# Regression and Classification with Regularization
# Ch6-ISLP; Ch2-IMLP

Zhongjian Lin
University of Georgia

September 10, 2024

# Linear Regression

- Linear regression is a useful tool for predicting a quantitative response.
- It has been around for a long time and is the topic of innumerable textbooks. Though it may seem somewhat dull compared to some of the more modern statistical approaches is still a useful and widely used statistical learning method.
- It serves as a good jumping-off point for newer approaches: as we will see in later chapters, many fancy statistical learning approaches can be seen as generalizations or extensions of linear regression.
- The importance of having a good understanding of linear regression before studying more complex learning methods cannot be overstated.

# Prediction Accuracy

We discuss some ways in which the simple linear model can be improved, by replacing plain least squares fitting with some alternative fitting procedures. As we will see, alternative fitting procedures can yield better *prediction accuracy* and *model interpretability*.

## Prediction Accuracy

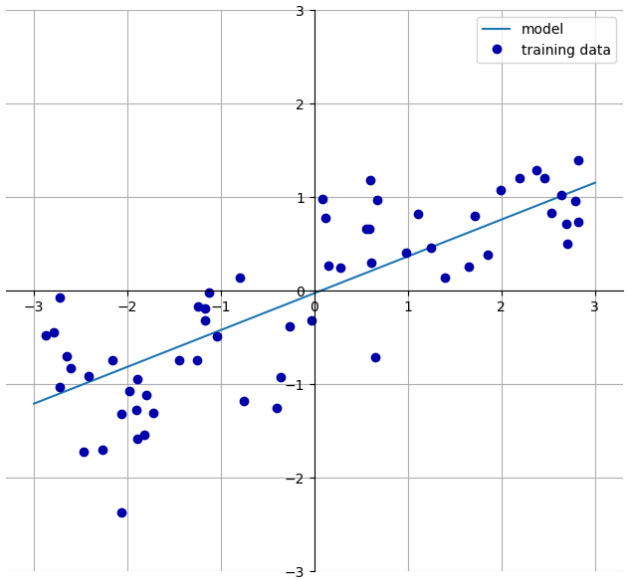
Provided that the true relationship between the response and the predictors is approximately linear, the least squares estimates will have low bias. If $n \gg p$, then the least squares estimates tend to also have low variance, and hence will perform well on test observations. However, if $n$ is not much larger than $p$, then there can be a lot of variability in the least squares fit, resulting in overfitting and consequently poor predictions on future observations. And if $p > n$, then there is no longer a unique least squares coefficient estimate. Each of these least squares solutions gives zero error on the training data, but typically very poor test set performance due to extremely high variance. By constraining or shrinking the estimated coefficients, we can often substantially reduce the variance at the cost of a negligible increase in bias. This can lead to substantial improvements in the accuracy with which we can predict the response for test observations.

## Model Interpretability

It is often the case that some or many of the variables used in a multiple regression model are in fact not associated with the response. Including such irrelevant variables leads to unnecessary complexity in the resulting model. By removing these variables—that is, by setting the corresponding coefficient estimates to zero—we can obtain a model that is more easily interpreted. Now least squares is extremely unlikely to yield any coefficient estimates that are exactly zero. We see some approaches for automatically performing feature selection or variable selection—that is, for excluding irrelevant variables from a multiple regression model.

# Linear Models

```
import mglearn
mglearn.plots.plot_linear_regression_wave()
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
lr = LinearRegression().fit(X_train, y_train)
print("lr.coef_: %s" % lr.coef_)
print("lr.intercept_: %s" % lr.intercept_)
```



# Model Performance

Let's look at the training set and test set performance:

```
print("training set score: %f" % lr.score(X_train, y_train))
print("test set score: %f" % lr.score(X_test, y_test))
training set score: 0.670089
test set score: 0.659337
```

An $R^2$ of around .66 is not very good, but we can see that the score on training and test set are very close together. This means we are likely underfitting, not overfitting. For this one-dimensional dataset, there is little danger of overfitting, as the model is very simple (or restricted). However, with higher dimensional datasets (meaning a large number of features), linear models become more powerful, and there is a higher chance of overfitting.

# Model with high dimension

Let's take a look at how LinearRegression performs on a more complex dataset, like the Boston Housing dataset. Remember that this dataset has 506 samples and 105 derived features. We load the dataset and split it into a training and a test set. Then we build the linear regression model as before:

```
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: %f" % lr.score(X_train, y_train))
print("test set score: %f" % lr.score(X_test, y_test))
training set score: 0.952353
test set score: 0.605775
```

This is a clear sign of overfitting, and therefore we should try to find a model that allows us to control complexity. One of the most commonly used alternatives to standard linear regression is Ridge regression, which we will look into next.

# Ridge Regression

Ridge regression is also a linear model for regression, so the formula it uses to make predictions is still Formula (1), as for ordinary least squares. In Ridge regression,the coefficients $\beta$ are chosen not only so that they predict well on the training data, but there is an additional constraint. We also want the magnitude of coefficients to be as small as possible; in other words, all entries of $\beta$ should be close to 0.

$$\hat{\beta} = \arg\min_{\beta} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda(\beta^T \beta) \tag{1}$$

Intuitively, this means each feature should have as little effect on the outcome as possible (which translates to having a small slope), while still predicting well. This constraint is an example of what is called regularization (L2). Regularization means explicitly restricting a model to avoid overfitting.

# Ridge Regression

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train, y_train)
print("training set score: %f" % ridge.score(X_train, y_train))
print("test set score: %f" % ridge.score(X_test, y_test))
training set score: 0.885797
test set score: 0.752768
```

As you can see, the training set score of Ridge is lower than for LinearRegression, while the test set score is higher. This is consistent with our expectation. With linear regression, we were overfitting to our data. Ridge is a more restricted model, so we are less likely to overfit. A less complex model means worse performance on the training set, but better generalization. As we are only interested in generalization performance, we should choose the Ridge model over the LinearRegression model.

# Ridge Regression

The Ridge model makes a trade-off between the simplicity of the model (near zero coefficients) and its performance on the training set. How much importance the model places on simplicity versus training set performance can be specified by the user, using the alpha parameter. Above, we used the default parameter alpha=1.0. Increasing alpha forces coefficients to move more towards zero, which decreases training set performance, but might help generalization.

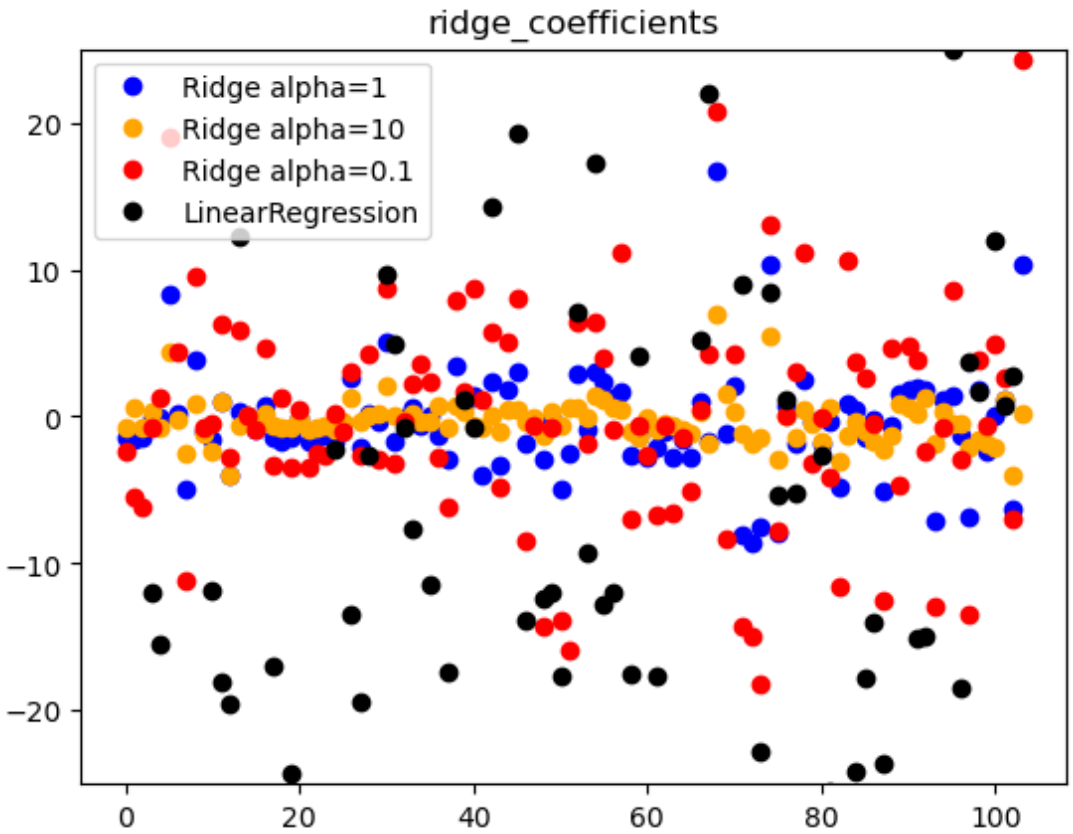
```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("training set score: %f" % ridge10.score(X_train, y_train))
print("test set score: %f" % ridge10.score(X_test, y_test))
training set score: 0.788279
test set score: 0.635941
```

For very small values of alpha, coefficients are barely restricted at all, and we end up with a model that resembles LinearRegression.

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("training set score: %f" % ridge01.score(X_train, y_train))
print("test set score: %f" % ridge01.score(X_test, y_test))
training set score: 0.928227
test set score: 0.772207
```

# Ridge Coefficient and Model Tradeoff

We can also get a more qualitative insight into how the alpha parameter changes the model by inspecting the coef_ attribute of models with different values of alpha. A higher alpha means a more restricted model, so we expect that the entries of coef_ have smaller magnitude for a high value of alpha than for a low value of alpha.



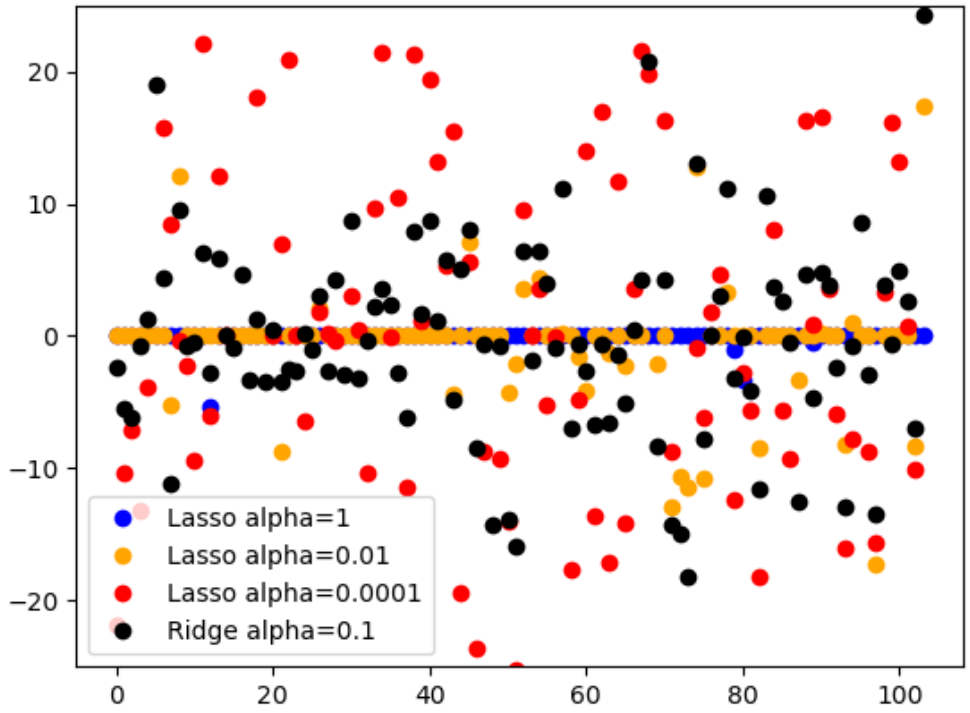
An alternative to Ridge for regularizing linear regression is the Lasso (least absolute shrinkage and selection operator). The lasso also restricts coefficients to be close to zero, similarly to Ridge regression, but in a slightly different way, called "L1" regularization.

$$\hat{\beta} = \arg\min_{\beta} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda \sum_{k=0}^{p} |\beta_k| \tag{2}$$

The consequence of l1 regularization is that when using the Lasso, some coefficients are exactly zero. This means some features are entirely ignored by the model. This can be seen as a form of automatic feature selection. Having some coefficients be exactly zero often makes a model easier to interpret, and can reveal the most important features of your model.

```
from sklearn.linear_model import Lasso
lasso = Lasso().fit(X_train, y_train)
print("training set score: %f" % lasso.score(X_train, y_train))
print("test set score: %f" % lasso.score(X_test, y_test))
print("number of features used: %d" % np.sum(lasso.coef_ != 0))
training set score: 0.293238
test set score: 0.209375
number of features used: 4
```

We find that it only used four of the 105 features. Above, we used the default of alpha=1.0. To diminish underfitting, let's try decreasing alpha.



# Ridge and LASSO

- In practice, Ridge regression is usually the first choice between these two models.
- However, if you have a large amount of features and expect only a few of them to be important, Lasso might be a better choice.
- Similarly, if you would like to have a model that is easy to interpret, Lasso will provide a model that is easier to understand, as it will select only a subset of the input features.

The classification model with linear index looks as follows:

$$y = 1\{\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \varepsilon > 0\}. \tag{3}$$

The formula looks very similar to the one for linear regression, but instead of just returning the weighted sum of the features, we threshold the predicted value at zero. If the function was smaller than zero, we predict the class 0, if it was larger than zero, we predict the class 1. This prediction rule is common to all linear models for classification. Again, there are many different ways to find the coefficients $\hat{\beta} = (\hat{\beta}_0, \cdots, \hat{\beta}_p)$.

# Algorithm

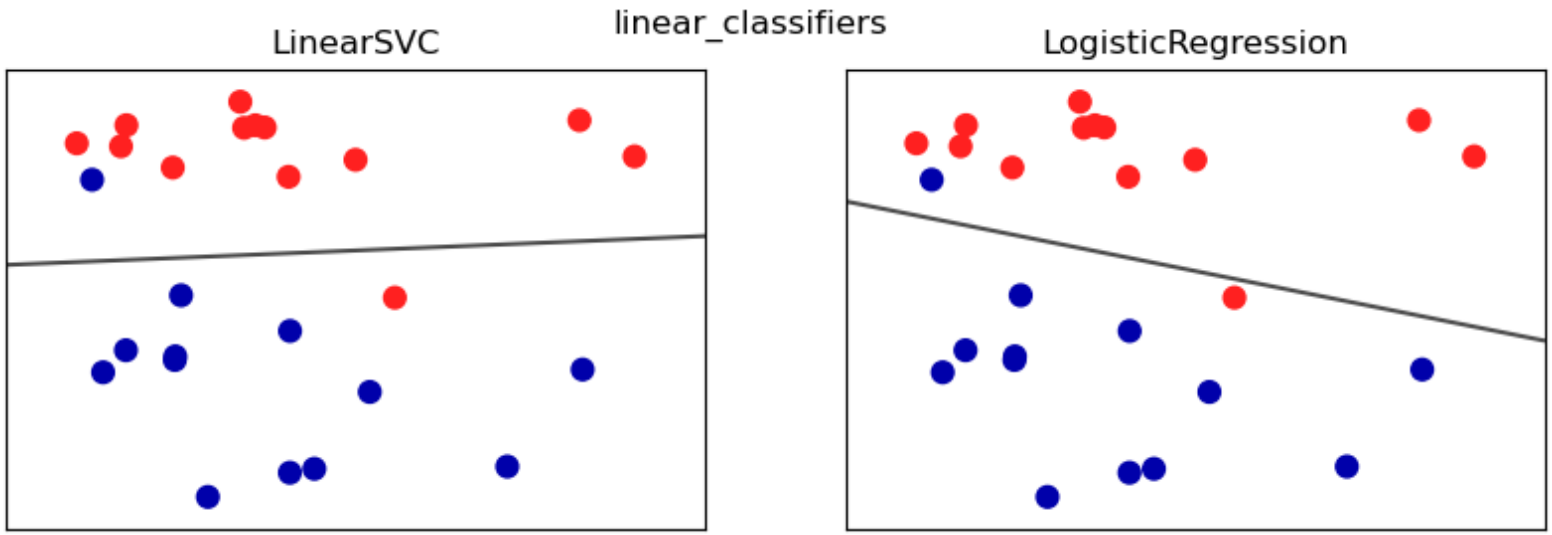
There are many algorithms for learning linear models. These algorithms all differ in the following two ways:

1. How they measure how well a particular combination of coefficients and intercept fits the training data.
2. If and what kind of regularization they use.

The two most common linear classification algorithms are logistic regression, implemented in *linear_model.LogisticRegression* and linear support vector machines (linear SVMs), implemented in *svm.LinearSVC*.

# Logistic and Support Vector Classification

```
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))
plt.suptitle("linear_classifiers")
for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
  clf = model.fit(X, y)
  mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5, ax=ax, alpha=.7)
  ax.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm2)
  ax.set_title("%s" % clf.__class__.__name__)
```



# Optimization (MLE)

When we fit the model to the training data, the algorithm chooses $\beta$ such that:

$$\hat{\beta} = \arg\max_{\beta} L(\beta)$$

where

$$L(\beta) \equiv \sum_{i=1}^{n} y_i \cdot \log[P(y_i = 1|X_i)] + (1 - y_i) \cdot \Big( \log[1 - P(y_i = 1|X_i)] \Big)$$

is the log-likelihood function.

**Regularized Logistic Regression**

$$\hat{\beta} = \arg\max_{\beta} L(\beta) + C\beta'\beta$$

# Logistic Regression on Breast Cancer

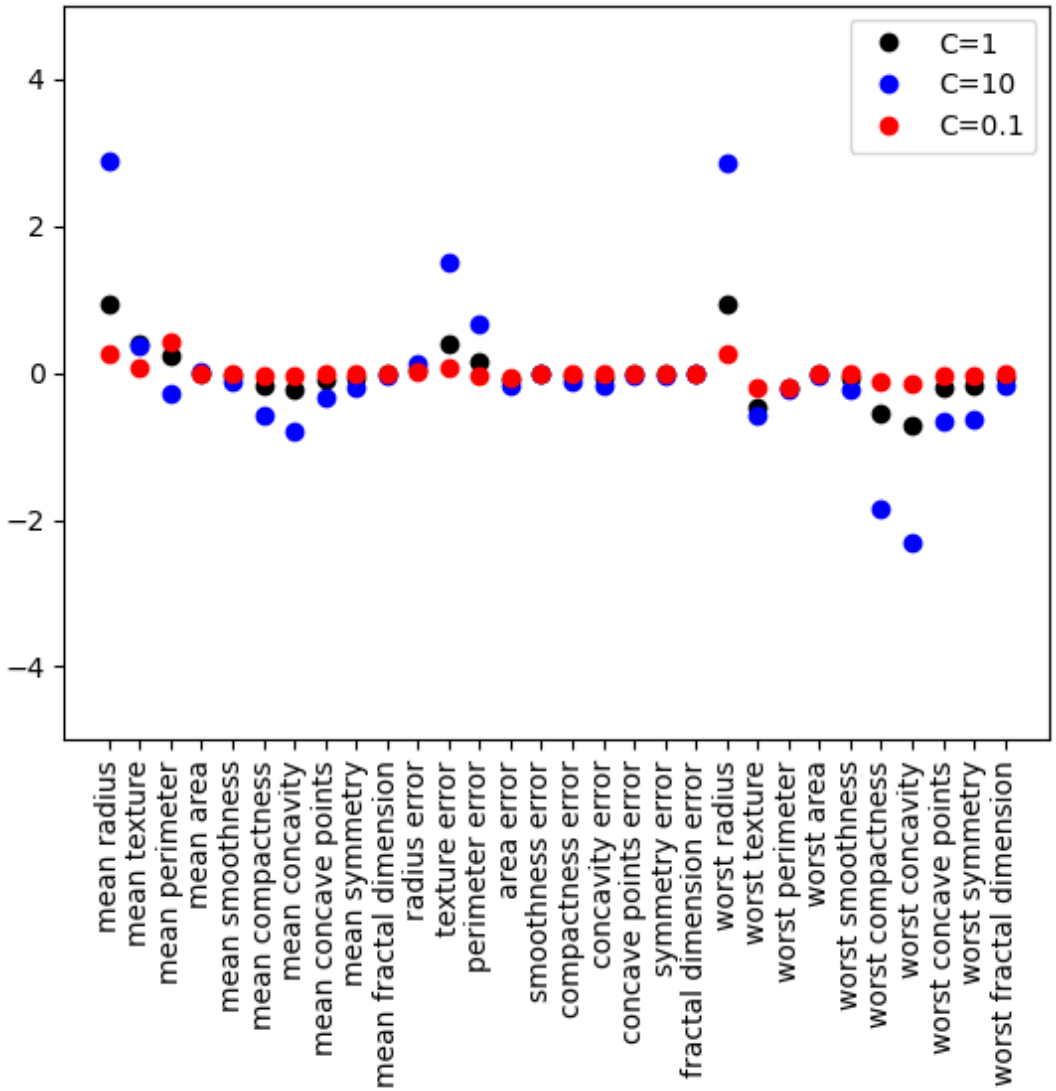
```
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logisticregression = LogisticRegression().fit(X_train, y_train)
print("training set score: %f" % logisticregression.score(X_train, y_train))
print("test set score: %f" % logisticregression.score(X_test, y_test))
training set score: 0.948357
test set score: 0.958042
```

The default value of $C = 1$ provides quite good performance, with 95% accuracy on both the training and the test set. As training and test set performance are very close, it is likely that we are underfitting. Let's try to increase C to fit a more flexible model.

```
logisticregression10 = LogisticRegression(C=10).fit(X_train, y_train)
print("training set score: %f" % logisticregression100.score(X_train, y_train))
print("test set score: %f" % logisticregression100.score(X_test, y_test))
training set score: 0.957746
test set score: 0.965035
```

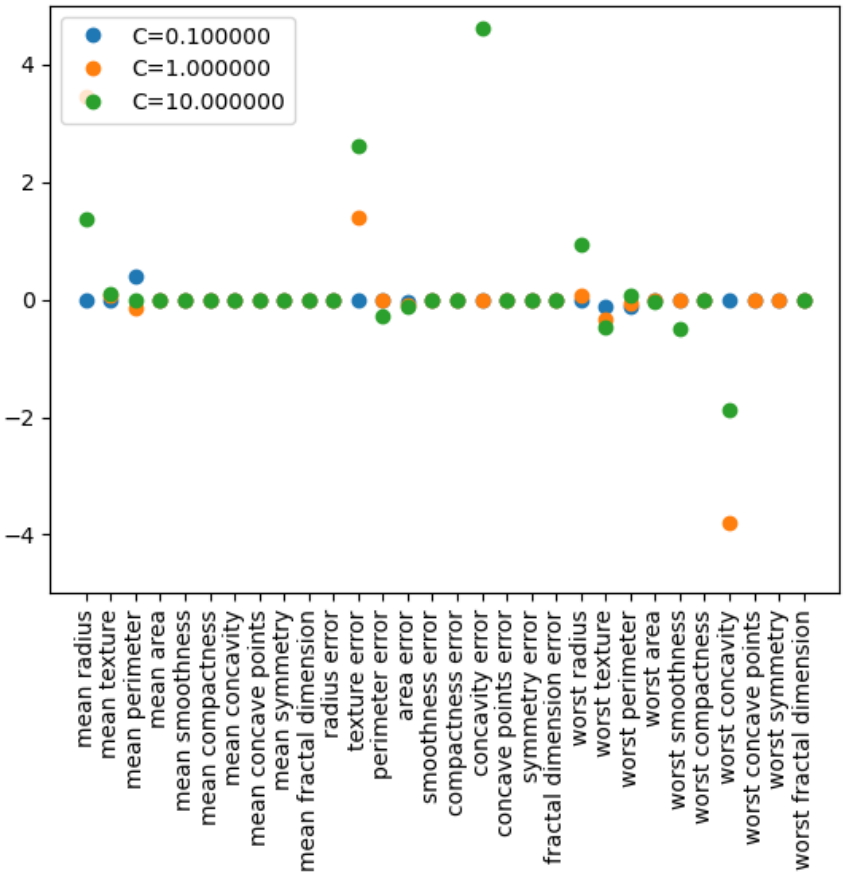
# Logistic and Support Vector Classification

Let's look at coefficients estimate with different $C$'s



## Logistic Regression with $L_1$ regularization

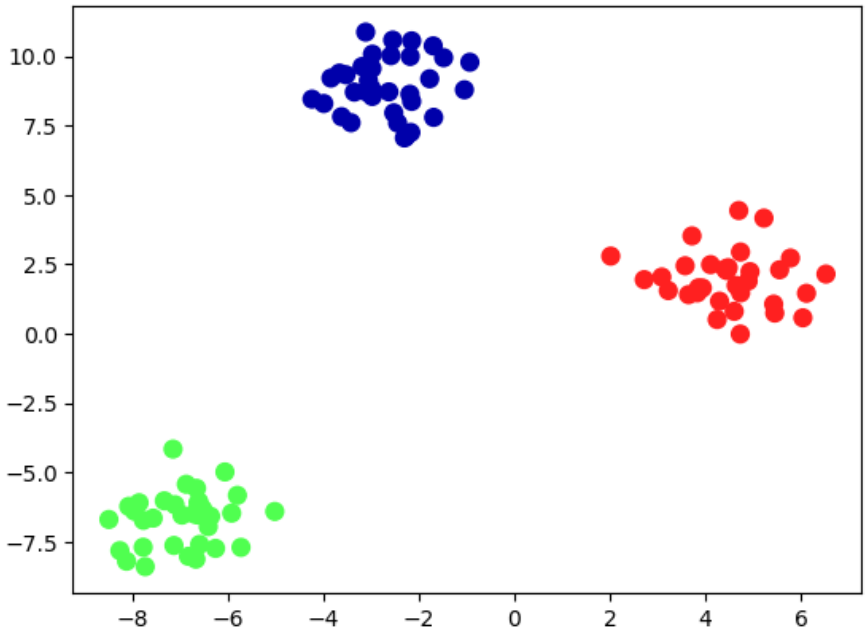
$$\hat{\beta} = \arg\max_{\beta} L(\beta) + C|\beta|$$



# Multiclass Classification

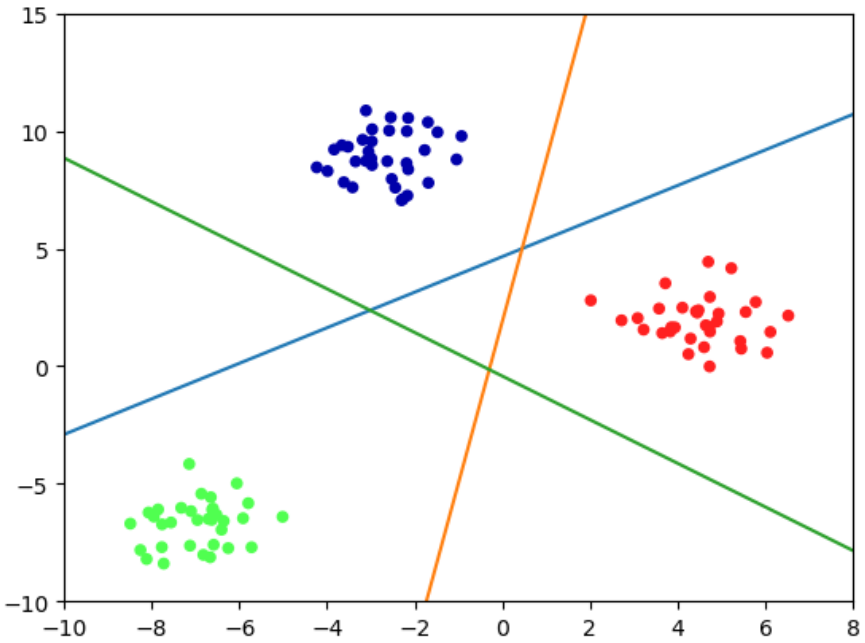
A common technique to extend a binary classification algorithm to a multi-class classification algorithm is the one-vs-rest approach. In the one-vs-rest approach, a binary model is learned for each class, which tries to separate this class from all of the other classes, resulting in as many binary models as there are classes. Linear support vector classifier serves to do the classification.

```
from sklearn.datasets import make_blobs
X, y = make_blobs(random_state=42)
plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm3)
```



# Support Vector Classifier

```
linear_svm = LinearSVC().fit(X, y)
print(linear_svm.coef_.shape)
print(linear_svm.intercept_.shape)
plt.scatter(X[:, 0], X[:, 1], c=y, s=60, cmap=mglearn.cm3)
line = np.linspace(-15, 15)
for coef, intercept in zip(linear_svm.coef_, linear_svm.intercept_):
plt.plot(line, -(line * coef[0] + intercept) / coef[1])
plt.ylim(-10, 15)
plt.xlim(-10, 8)
```



# Support Vector Classifier