L2 – Supervised Learning I

- Classification and Regression
- Generalization, Overfitting and Underfitting
- Supervised Machine Learning Algorithms
  - K-Nearest Neighbors
  - Linear Models
  - Naïve Bayes Classifiers

Classification and Regression

- Classification: to predict a class label for an input
  - Binary classification for distinguishing between two classes

  (e.g., if this email is spam?)
  - Multiclass classification for more then two classes

  (e.g., given text to detect its language type from a predefined list)
- Regression: to predict a continuous / floating-point number
  - E.g., predicting a person's annual income from their education, their age, and where they live.
- How to distinguish between classification and regression?

Generalization

- Being able to make accurate predictions on unseen data, it is named as **generalize** from the training set to the test set.
  - We usually build a model can make accurate prediction on the training set (which may go wrong on some cases in test set)
  - With very complex models, can always be accurate on training

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

To predict if a customer will buy a boat

A data scientist makes a rule:
Customer older than 45 & has less than 3 kids or not divorced

Or who are 66, 52, 53 or 58 years old

Accurate but does it make sense?

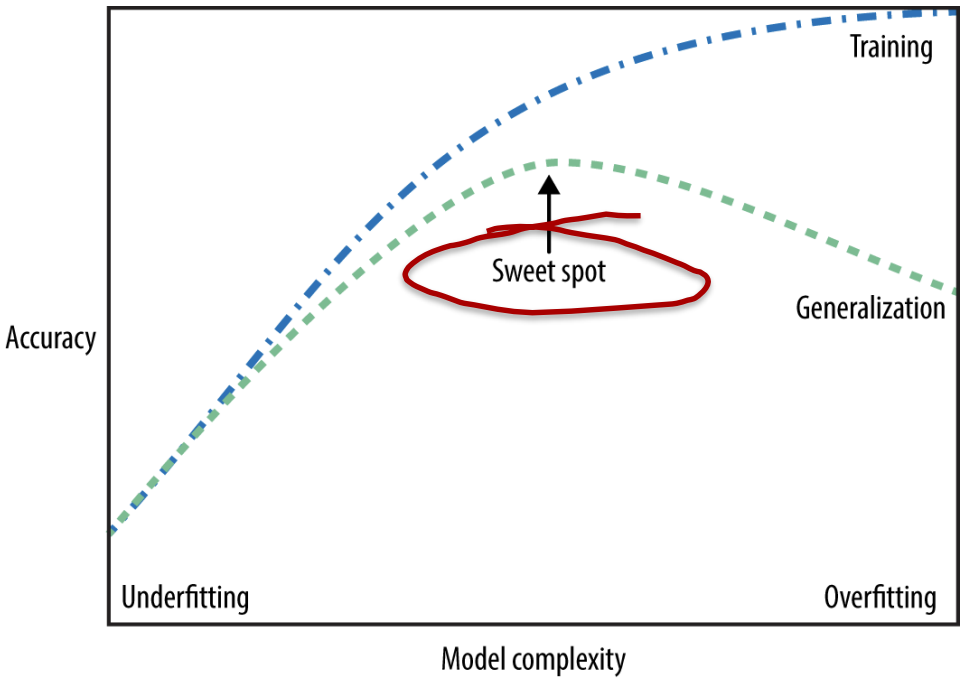
Overfitting vs. Underfitting

- We need to predict accurately on new data
  - 100% accurate on a training set has less meaning
  - The only measure of whether an algorithm will perform well on new data is the evaluation on the test set.
- We expect simple models to generalize better to new data
  - **Overfitting**: building a model too complex for the amount of info.

  (e.g., as what the data scientist proposed above)
  - **Underfitting**: choosing too simple a model do badly even on training data

  (e.g., define a rule such as: “everybody who owns a house buys boat”; you might not be able to capture all the aspects in the data)

- There is trade-off between overfitting and underfitting
  - Model complexity is intimately tied to the variation of inputs
  - The larger variety of data points your date set contains, the more complex a model you can use without overfitting
- Never under-estimate the power of more data



Considering again the rule of data scientist as:
Customer older than 45 & has less than 3 kids or not divorced

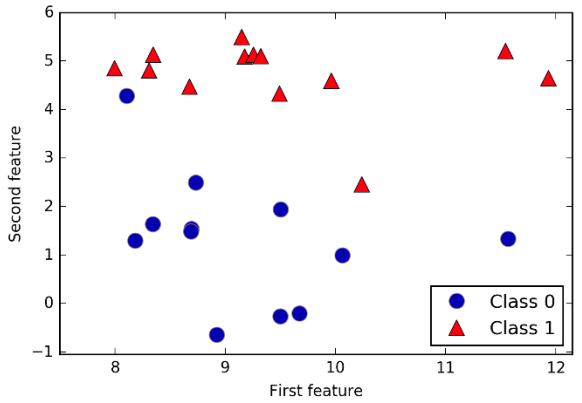
If we saw 10,000 more rows of customer data, and all of them compile with the rule

We would be much more likely to believe this to be a good rule than when it was developed using only the 12 rows in the table above

Sample Datasets for Studying Supervised Learning Algorithms

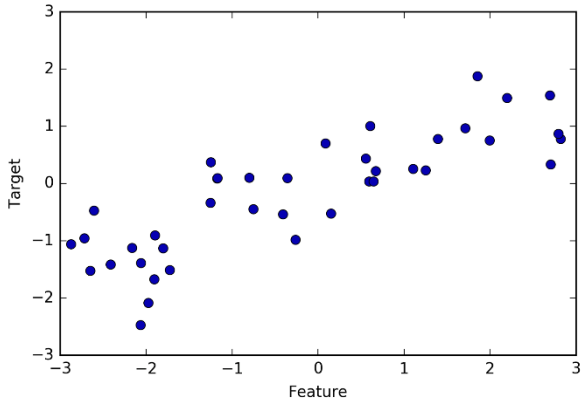
- forget dataset: a synthetic two-class classification dataset

```
# generate dataset
X, y = mglearn.datasets.make_forge()
# plot dataset
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Class 0", "Class 1"], loc=4)
plt.xlabel("First feature")        plt.ylabel("Second feature")
print("X.shape: {}".format(X.shape))
```



- wave dataset: a single input feature with a continuous target variable (or response)

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Feature")
plt.ylabel("Target")
```



- Simple & low-dimensional datasets
  - Can be easily visualized
  - However, any intuition derived from datasets with few features (low-dimensional datasets) might not hold in datasets with many features (high-dimensional datasets)
- Wisconsin Breast Cancer dataset (benign vs. malignant)

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): \n{}".format(cancer.keys()))
```

- Boston Housing dataset (to predict the median value of homes in different regions in the 1970s)

```
from sklearn.datasets import load_boston
boston = load_boston()
print("Data shape: {}".format(boston.data.shape))

X, y = mglearn.datasets.load_extended_boston()
print("X.shape: {}".format(X.shape))
```

We can look at all products (also called interactions) between 13 features:

i.e., not only consider crime rate and highway accessibility as features but also the product of crime rate and highway accessibility as features.

k-Nearest Neighbors

- Simplest machine learning algorithm
  - Building the model consists of only storing the training dataset
  - Algorithm finds the closest data points (by different dist. metrics)
- Simplest version: consider exactly one nearest neighbor

mglearn.plots.plot_knn_classification(n_neighbors=1)

- Voting version: consider more than one neighbors (specified by parameter “n_neighbors”)

mglearn.plots.plot_knn_classification(n_neighbors=3)

  - Voting can also be applied to multi-class classification
  - We count how many neighbors belong to each class and predict the most common class

- We test the kNN classifier on the forge dataset
  - First, split our data into a training and a test set
  - Next, import and instantiate the class
  - Third, fit the classifier using the training set (i.e., storing the set)
  - Last, call the predict method and evaluate the generalize

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)

clf.fit(X_train, y_train)

print("Test set predictions: {}".format(clf.predict(X_test)))
print("Test set accuracy: {:.2f}".format(clf.score(X_test, y_test)))
```

- Analyzing kNN classifier
  - A best way is to visualize the decision boundary

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))
for n_neighbors, ax in zip([1, 3, 9], axes):
  # the fit method returns the object self, so we can instantiate
  # and fit in one line
  clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
  mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
  mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
  ax.set_title("{} neighbor(s)".format(n_neighbors))
  ax.set_xlabel("feature 0")
  ax.set_ylabel("feature 1")
axes[0].legend(loc=3)
```

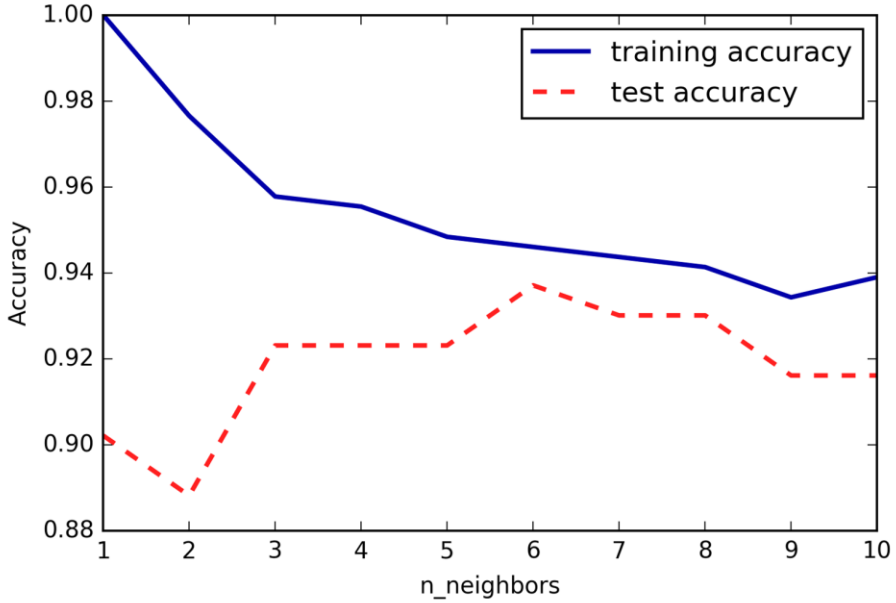
  - Considering more and more neighbors leads to a smoother decision boundary (i.e., lower model complexity)
  - Using fewer neighbors corresponds to high model complexity

- We can now study and confirm the connection between model complexity and generalization (on real-world data)

```
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=66)

training_accuracy = []                    test_accuracy = []

neighbors_settings = range(1, 11)     # try n_neighbors from 1 to 10

for n_neighbors in neighbors_settings:
        # build the model
        clf = KNeighborsClassifier(n_neighbors=n_neighbors)
        clf.fit(X_train, y_train)
        # record training set accuracy
        training_accuracy.append(clf.score(X_train, y_train))
        # record generalization accuracy
        test_accuracy.append(clf.score(X_test, y_test))
```

```
plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
```

- While real-world plots are rarely very smooth, we can still recognize some of the characteristics of overfitting and underfitting
- When more neighbors are considered, the model becomes simpler and the training accuracy drops.



k-Neighbors Regression

- There is also a regression variant of the kNN algorithm
  - Using single neighbor, the prediction gives the nearest neighbor's value as target.
  - Using more neighbors, the prediction is the average or mean.

```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

- The kNN algorithm for regression is implemented in the KNeighborsRegressor class in scikit-learn

```
from sklearn.neighbors import KNeighborsRegressor
X, y = mglearn.datasets.make_wave(n_samples=40)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
reg = KNeighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
print("Test set predictions:\n{}".format(reg.predict(X_test)))
```

- We can also evaluate the model using the score method
  - For regression, returns the R2 score between [0, 1]
  - 1 stands for a perfect prediction; 0 corresponds to a constant model that just predicts the mean of the training set responses.

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# create 1,000 data points, evenly spaced between -3 and 3
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
        # make predictions using 1, 3, or 9 neighbors
        reg = KNeighborsRegressor(n_neighbors=n_neighbors)
        reg.fit(X_train, y_train)
        ax.plot(line, reg.predict(line))
        ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
        ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)
        ax.set_title("{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(n_neighbors,
reg.score(X_train, y_train),reg.score(X_test, y_test)))
        ax.set_xlabel("Feature")
        ax.set_ylabel("Target")
axes[0].legend(["Model predictions", "Training data/target", "Test data/target"], loc="best")
```

Analysis of kNN Classifier

- Two important parameters:
  - Number of neighbors
  - How you measure distance between data points
- Drawbacks:
  - When the training set is very large, prediction can be slow
  - Often does not perform well on datasets with many features (hundreds or more), and it does particularly badly with datasets where most features are 0 most of the time (so-called sparse datasets).

Linear Models

- Make a prediction using a linear function of input features
  - For regression, the general prediction formula

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

  - For a data set with a single feature

$$\hat{y} = w[0] * x[0] + b$$

mglearn.plots.plot_linear_regression_wave()

- Linear models for regression (Link of Explanation)
  - A linear for a single feature (i.e., all the fine details are lost)
  - A plane when using two features
  - A hyperplane in higher dimensions when using more features

(For dataset with many features, linear models can be very powerful)

- Linear regression finds the parameters $\boldsymbol{w}$ and $\boldsymbol{b}$ that minimize the mean squared error between predictions and true regression targets, $\boldsymbol{y}$, on the training set.
  - Linear regression has no parameters, which is a benefit
  - It also has no way to control model complexity

```
from sklearn.linear_model import LinearRegression

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)
```

- The "slope" parameters ($\boldsymbol{w}$) also called weights/coefficients
- The offset or intercept ($\boldsymbol{b}$) is stored in intercept_ attribute

```
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

- Score on the wave dataset

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

  - An R2 of around 0.66 is not very good, but close scores on the training and test sets.
  - This means likely underfitting but not overfitting.

- Score on the extended Boston Housing dataset

```
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("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

  - Discrepancy between performance on the training set and the test set is a clear sign of overfitting
  - Try an alternative as ridge regression (also a linear model)

Ridge Regression

- A method of regularization of ill-possed problems (Link)
  - The coefficients (w) are chosen not only for well predicting but also to fit an additional constraint (i.e., min $||w||$)
  - Intuitively, this means each feature should have as little effect on the outcome as possible (which translates to a small slope)
- Regularization: explicitly restrict a model to avoid overfitting
  - With linear regression, we were overfitting our data
  - Ridge is a more restricted model (i.e., less likely to overfitting)

```
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("Test set score: {:.2f}".format(ridge.score(X_test, y_test)))
```

Less complex models means Better generalization

- How much importance the model places on simplicity vs. training set performance
  - Can be controlled by the parameter alpha (default value 1.0)
  - Increasing alpha forces coefficients to move more toward zero, thus might help generalization

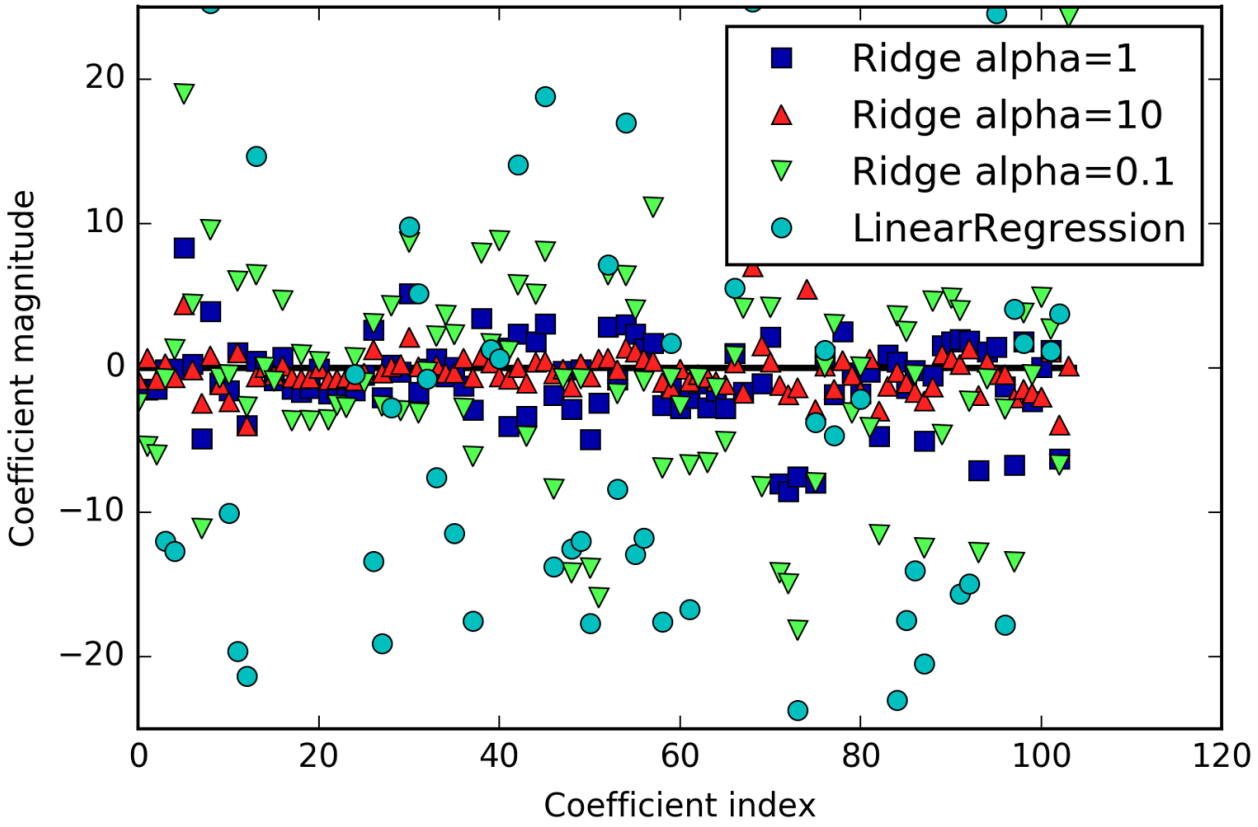
```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge10.score(X_test, y_test)))
```

  - Decreasing alpha allow coefficients to be less restricted, which can end up with (alpha = 0 as the standard linear regression)

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

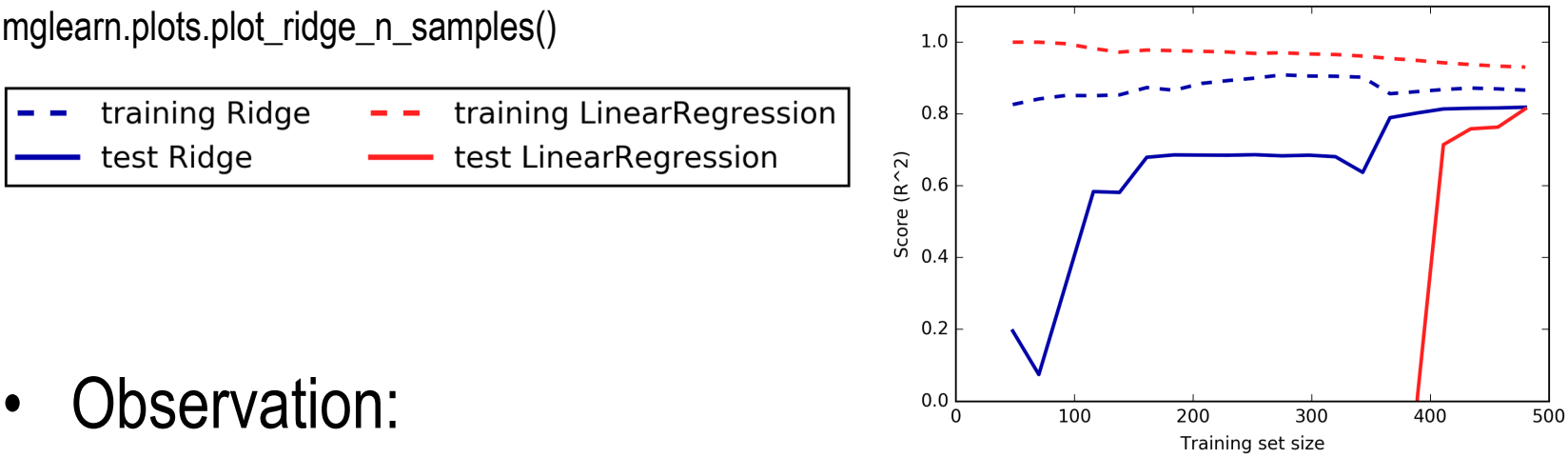
- Plot of resultant coefficients with different alpha values (see the figure below)

```
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=0.1")
plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```



- Another way to understand the influence of regularization is to fix a value of alpha but vary the amount of training data

mglearn.plots.plot_ridge_n_samples()



- Observation:
  - The training score is higher than the test score for all cases
  - The training score of ridge is lower than the linear regression
  - The test score for ridge is better (particularly for small subsets)
- Lesson:
  - With enough data, regularization becomes less important
  - Interesting decrease in training performance for linear regression (if more data is added, it is harder to overfit or memorize the data)

Lasso

- An alternative to Ridge but with L1 regularization
  - Consequence: some coefficients are exactly zero.

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

- Observation:
  - Lasso does quite badly with default parameter alpha = 1.0
  - This indicates that we are underfitting
  - Try decrease *alpha*; and also increase the default *max_iter*

```
# we increase the default setting of "max_iter",
# otherwise the model would warn us that we should increase max_iter.
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
```

- If we set alpha too low, again we remove the effect of regularization and overfitting (similar to LinearRegression)

```
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("Test set score: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso00001.coef_ != 0)))

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("Coefficient index")
plt.ylabel("Coefficient magnitude")
```

  - In practice, ridge regression is usually the first choice
  - For an expectation with smaller amount of feature, use Lasso
  - Other option: the ElasticNet class of scikit-learn (L1 & L2)

Linear Model for Classification

- A prediction is made using the following formula

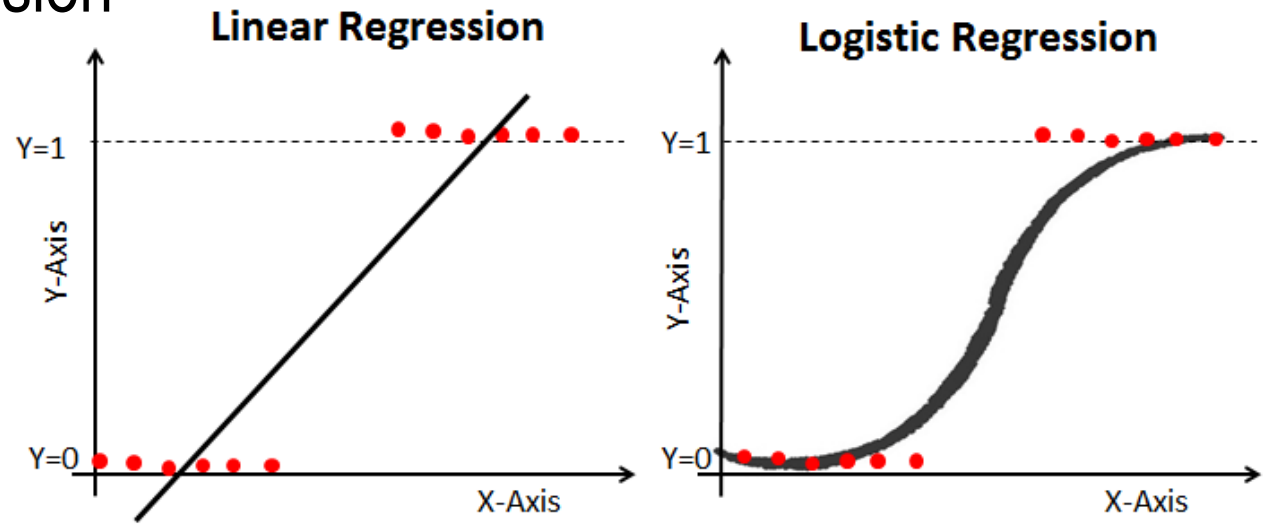
$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b > 0$$

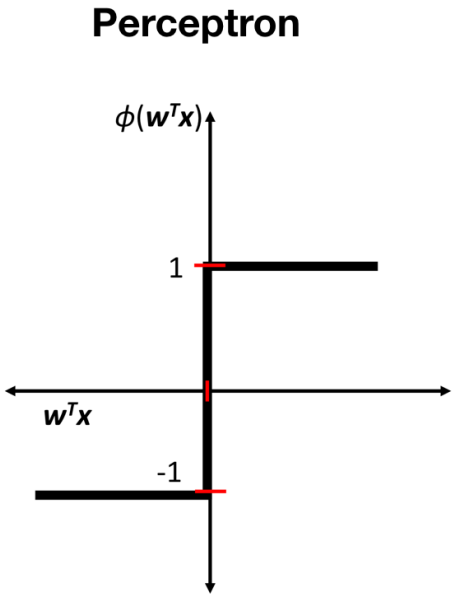
  - Two classes: the class +1 and the class -1
  - For linear models for classification, the decision boundary is a linear function of the input.
  - Two common linear classification:
    - Logistic regression – linear_model.LogisticRegression
    - Linear support vector machines (line SVMs) – svm.LinearSVC
  - Algorithms mainly differ in two ways:
    - Different loss functions (in many case, of little significance)
    - If and what kind of regularization (more important for generalization)
  - We apply both classifiers to the forge dataset below.

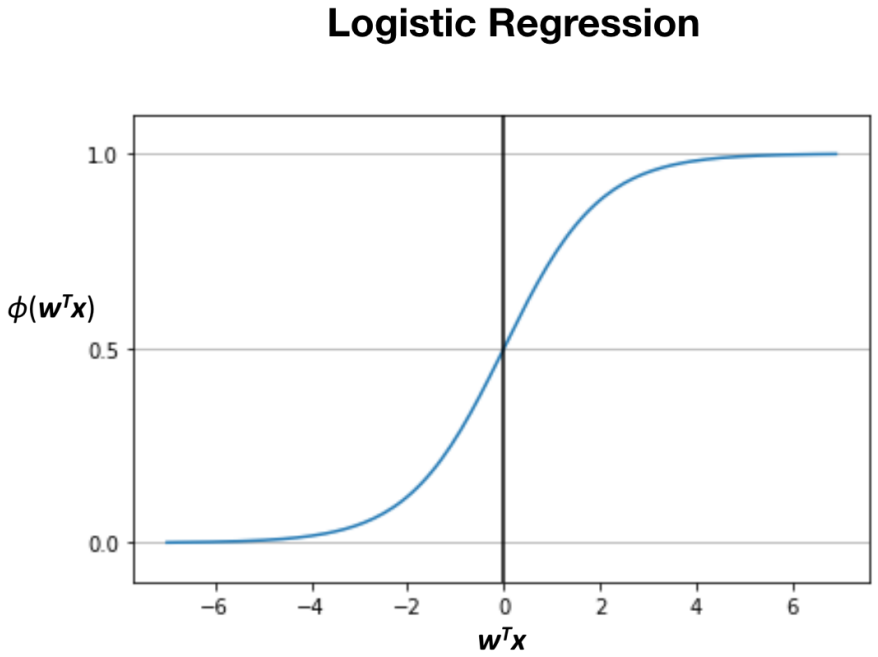
- LogisticRegression can provide the probability

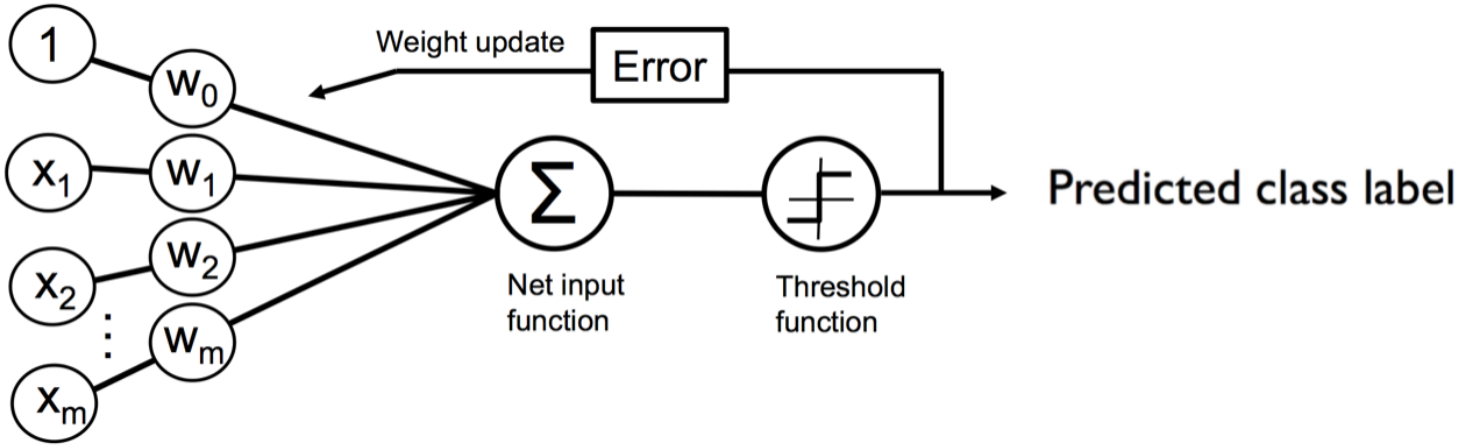
$$z = w^T x$$

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

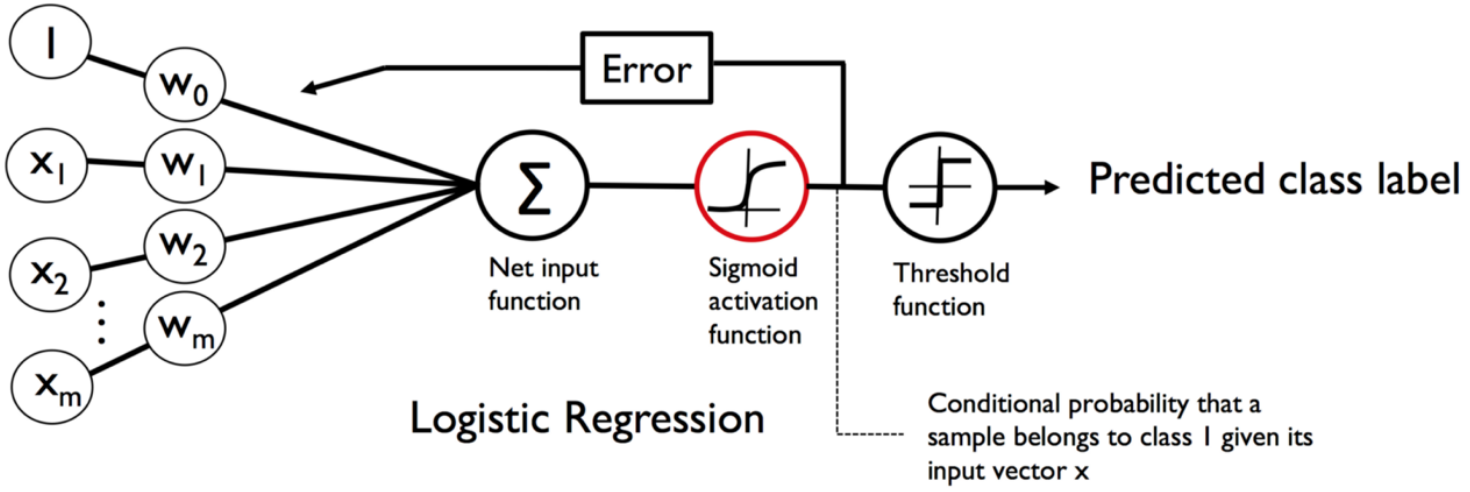








Perceptron



Logistic Regression

```
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(), LogisticRegression()], axes):
        clf = model.fit(X, y)
        mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5, ax=ax, alpha=.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()
```

- Two models come up with similar boundaries
  - Both can be further controlled by the strength of regularization, C
    - Applying an L2 regularization
    - High value of C correspond to less regularization

```
mglearn.plots.plot_linear_svc_regularization()
```

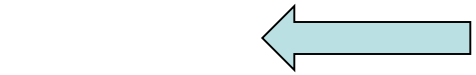
- Analyze LogisticRegression on the Breast Cancer dataset

```
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)
logreg = LogisticRegression(solver='lbfgs', max_iter=10000).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

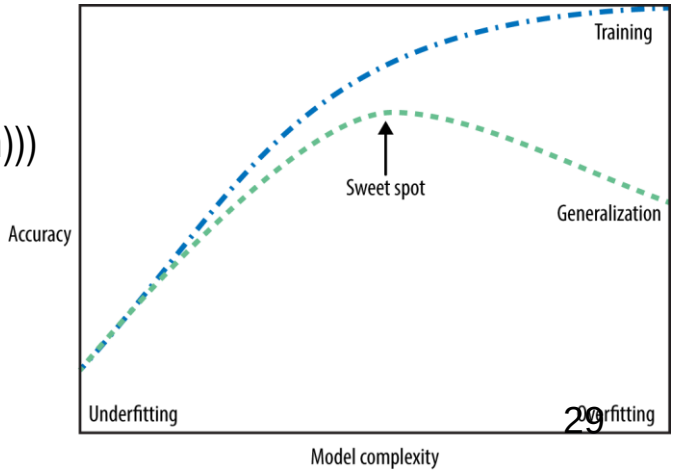
Default value of C = 1.0

```
logreg100 = LogisticRegression(C=100, solver='lbfgs', max_iter=10000).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg100.score(X_test, y_test)))
```

Decrease the value of C

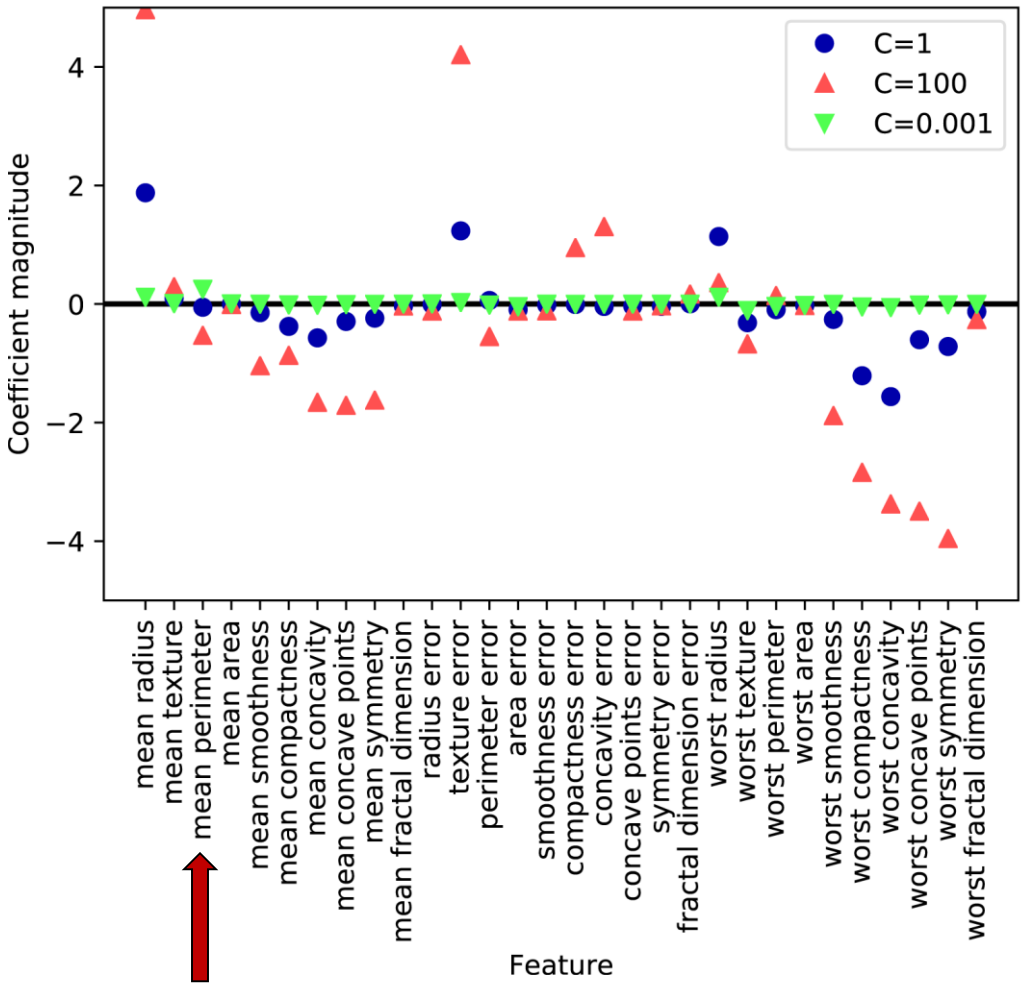


```
logreg001 = LogisticRegression(C=0.01,
                solver='lbfgs', max_iter=10000).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg001.score(X_test, y_test)))
```



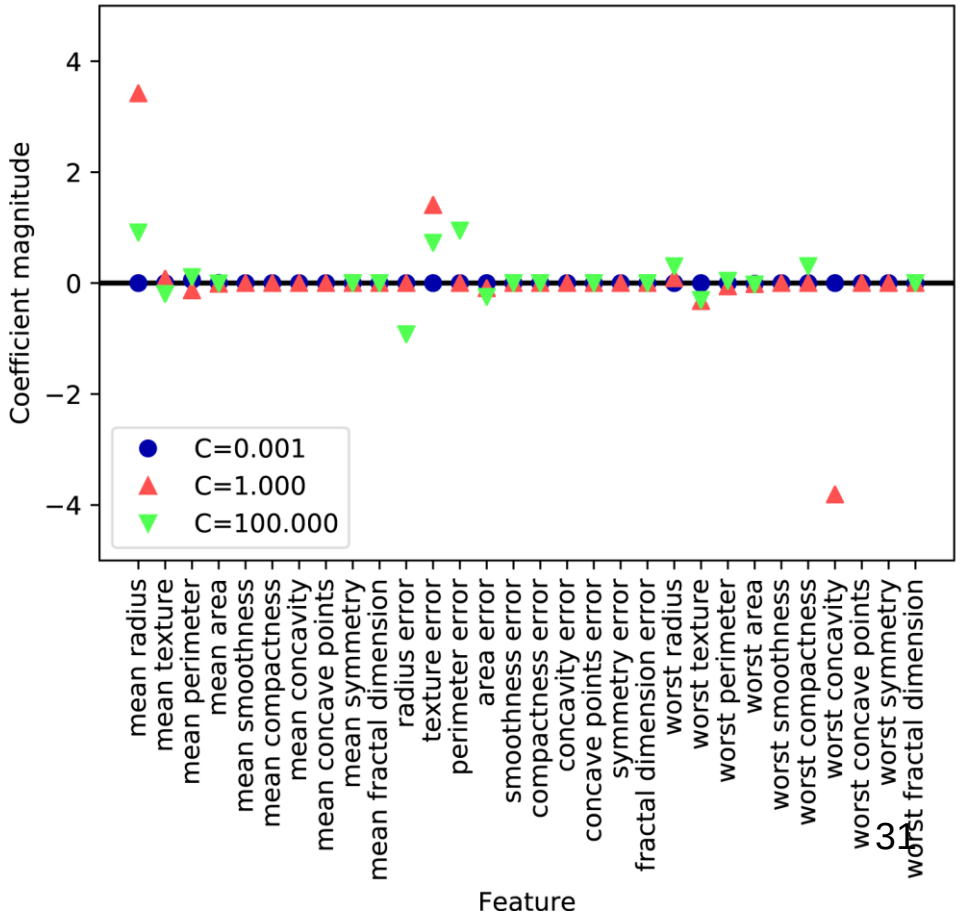
- Look at the coefficients learned by using diff. regularization para. C

```
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("Feature")
plt.ylabel("Coefficient magnitude")
plt.legend()
```



- Desire a more interpretable model, using L1 regularization

```
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
  lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
  print("Training accuracy of l1 logreg with C={:.3f}: {:.2f}".format(C, lr_l1.score(X_train, y_train)))
  print("Test accuracy of l1 logreg with C={:.3f}: {:.2f}".format(C, lr_l1.score(X_test, y_test)))
  plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))

plt.xticks(range(cancer.data.shape[1]),
           cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Feature")
plt.ylabel("Coefficient magnitude")
plt.ylim(-5, 5)
plt.legend(loc=3)
```

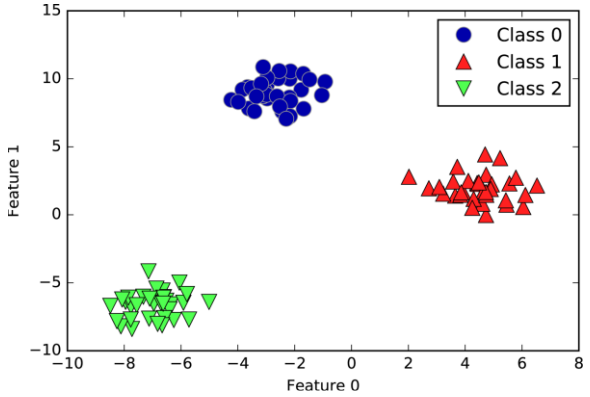


Linear Models for Multiclass Classification

- Many linear classification models are for binary only
  - Binary classifier can be extended by the one-vs.-rest approach
    - A binary model is learn for each class
    - To make a prediction, all binary classifiers are run on a test point
    - The classifier has the highest score on its single class "wins"
  - With the exception of logistic regression

- The mathematics behind multiclass logistic regression differ from the one-vs.-rest approach
  - They also result in one coefficient vector and intercept per class
  - Also choose the one with the highest score

```
from sklearn.datasets import make_blobs
X, y = make_blobs(random_state=42)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"])
```

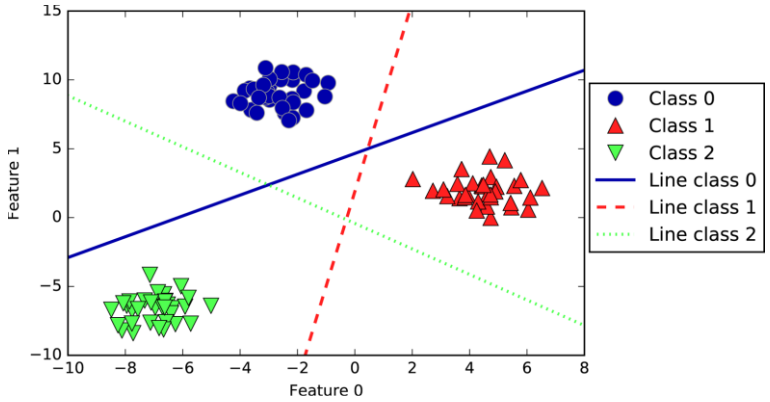


- Now we train a LinearSVC classifier on the dataset

```
linear_svm = LinearSVC().fit(X, y)    # Need to add "from sklearn.svm import LinearSVC" before this line
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

- Let's visualize the lines given by three binary classifiers

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_,
    linear_svm.intercept_, mglearn.cm3.colors):
      plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1', 'Line class 2'], loc=(1.01, 0.3))
```



- The following code shows the predictions for all regions

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
# Source code: https://github.com/amueller/mglearn/blob/master/mglearn/plot_2d_separator.py
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_, mglearn.cm3.colors):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1', 'Line class 2'], loc=(1.01, 0.3))
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

- Parameters in linear model
  - L2 regularization (default) or L1 regularization (interpretation)

alpha in the *regression* models; C in LinearSVC / LogisticRegression

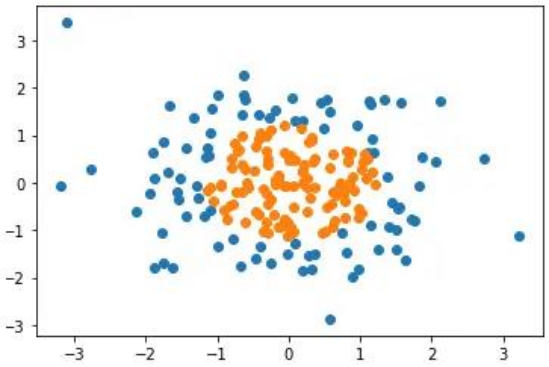
  - Solver considering to use solver='sag' option for large dataset
- Strength: Often perform well when feature # is large compared to sample # (i.e., high dimension space)

Naïve Bayes Classifiers

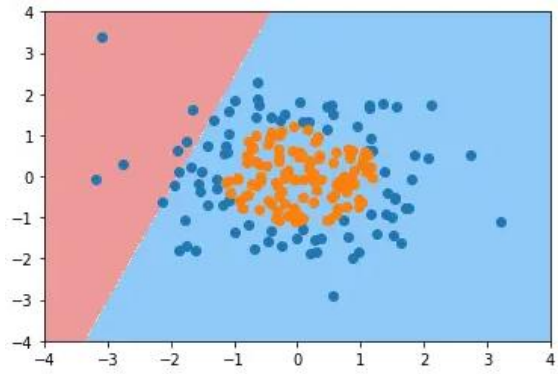
- Quite similar to the linear models discussed above
  - Faster: Bayes models learn parameters by checking features individually & collect simple per-class statistics from each feature
  - Continues data: GaussianNB (used on very high-Dim. Data)
  - Count data (integer count of sth): MultinomialNB
  - Binary data: BernoulliNB
- To make a prediction, a data point is compared to the statistics for each of the classes and the best matching class is predicted. [Link]
  - Prediction formula is in the similar form as the linear models
  - But training is even faster (good for very large datasets)

Nonlinear Logistic Regression

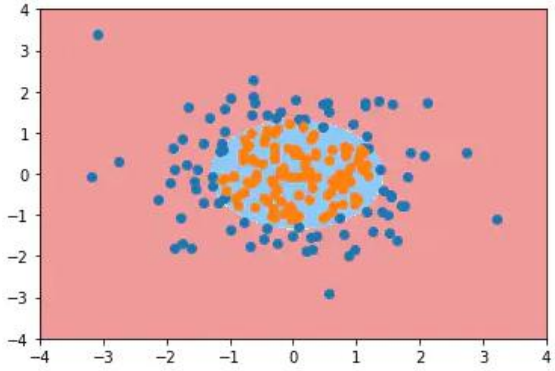
- Linear logistic regression can fail in complex situation
  - Mainly caused by the linear decision boundary



Input Samples



Linear Logistic Regression



Logistic Regression
(by quadratic polynomial)

- Solution: changing the decision boundary by polynomial [Reading Link]