Ch 3: Classifiers using scikt-learn

- topics
  - popular algorithms for classification such as logistic regression, support vector machines, and decision trees
  - using the scikit-learn machine learning library
  - discussions on classifiers with linear and nonlinear decision boundaries

scikit-learn API

- combines a user-friendly and consistent interface with a highly optimized implementation of several classification algorithms.
- scikit-learn library offers
  - a large variety of learning algorithms
  - many convenient functions to preprocess data and to fine-tune and evaluate models.

scikit-learn API

```python
from sklearn import datasets
import numpy as np

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target

print('Class labels:', np.unique(y))
```

```
Class labels: [0 1 2]
```

scikit-learn API

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y)
```

scikit-learn API

Standardizing the features:

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Training a perceptron via scikit-learn

```
from sklearn.linear_model import Perceptron

ppn = Perceptron(eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
```

```
y_pred = ppn.predict(X_test_std)
print('Misclassified examples: %d' % (y_test != y_pred).sum())
```

```
Misclassified examples: 1
```

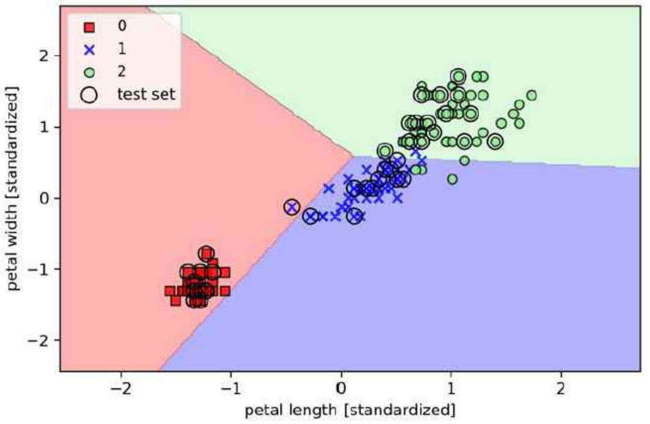
```
from sklearn.metrics import accuracy_score

print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.978
```

```
print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))
```

```
Accuracy: 0.978
```

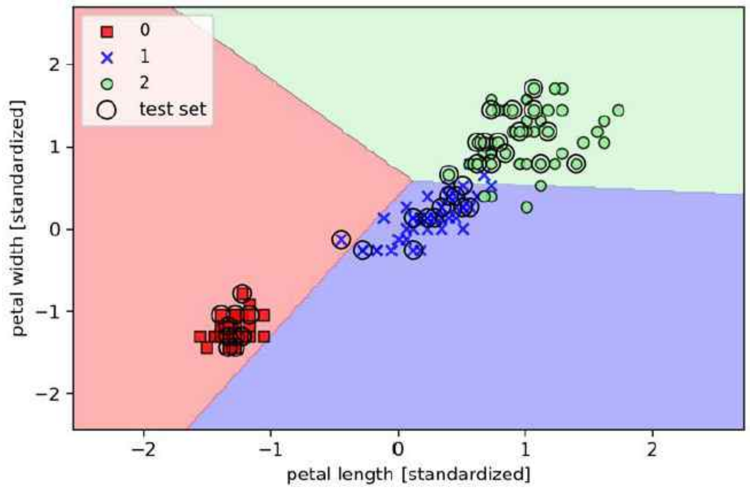
```python
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=cl,
                    edgecolor='black')

    # highlight test examples
    if test_idx:
        # plot all examples
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0],
                    X_test[:, 1],
                    c='',
                    edgecolor='black',
                    alpha=1.0,
                    linewidth=1,
                    marker='o',
                    s=100,
                    label='test set')
```



Training a perceptron model using the standardized training data:

```
X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))

plot_decision_regions(X=X_combined_std, y=y_combined,
                      classifier=ppn, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/03_01.png', dpi=300)
plt.show()
```

```
# highlight test examples
if test_idx:
    # plot all examples
    X_test, y_test = X[test_idx, :], y[test_idx]
    plt.scatter(X_test[:, 0],
                X_test[:, 1],
                c='',
                edgecolor='black',
                alpha=1.0,
                linewidth=1,
                marker='o',
                s=100,
                label='test set')
```




```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=cl,
                    edgecolor='black')

    # highlight test examples
    if test_idx:
        # plot all examples
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0],
                    X_test[:, 1],
                    c='',
                    edgecolor='black',
                    alpha=1.0,
                    linewidth=1,
                    marker='o',
                    s=100,
                    label='test set')
```

- Three flower classes cannot be perfectly separated by a linear decision boundary.

Logistic Regression

- a classification model that is very easy to implement and performs very well on linearly separable classes
- widely used
- idea behind logistic regression
  - odds in favor of a particular event
  - odds = $p/(1 - p)$ where p = prob. of an event that we want to predict
  - *logit* function: domain ~ 0 to 1, range ~ entire real-number

$p(y = 1|\mathbf{x})$: conditional prob. that a particular example belongs to class 1 given its features, $\mathbf{x}$.

  - predicts the prob., p, that a certain example belongs to a particular class.

$$\boldsymbol{Z} = \boldsymbol{w}^T\boldsymbol{x} = w_0x_0 + w_1x_1 + \cdots + w_mx_m$$

Logistic Regression

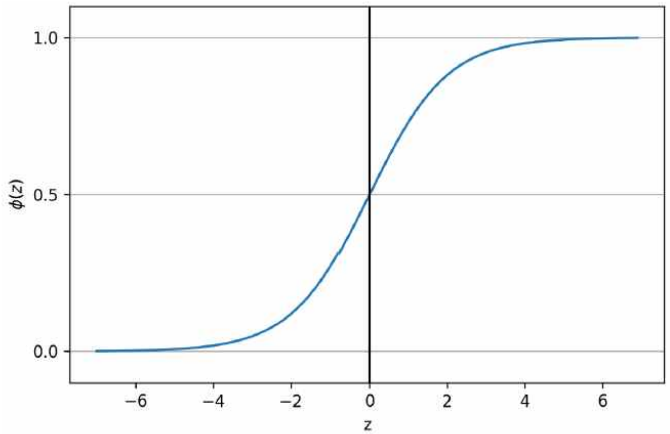
- From

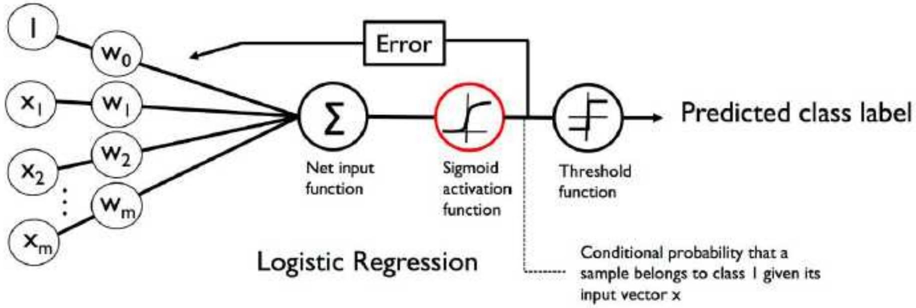
$$logit(p(y=1|\boldsymbol{x})) = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \sum_{i=0}^{m} w_ix_i = \boldsymbol{w}^T\boldsymbol{x}$$

-

- where $Z = \boldsymbol{w}^T\boldsymbol{x} = w_0x_0 + w_1x_1 + \cdots + w_mx_m$

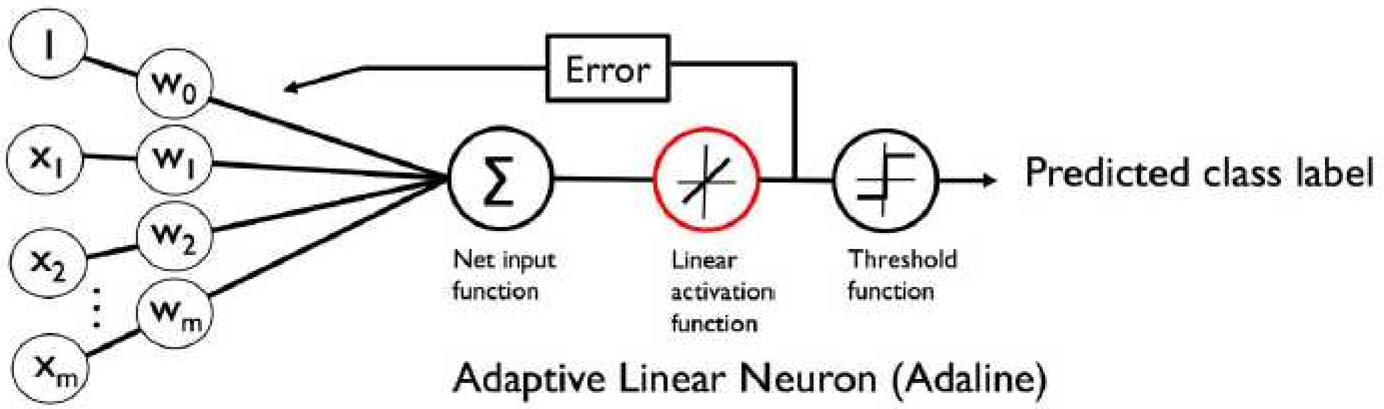
linear combination of weights and features
w_0 ~ bias, x_0 = 1

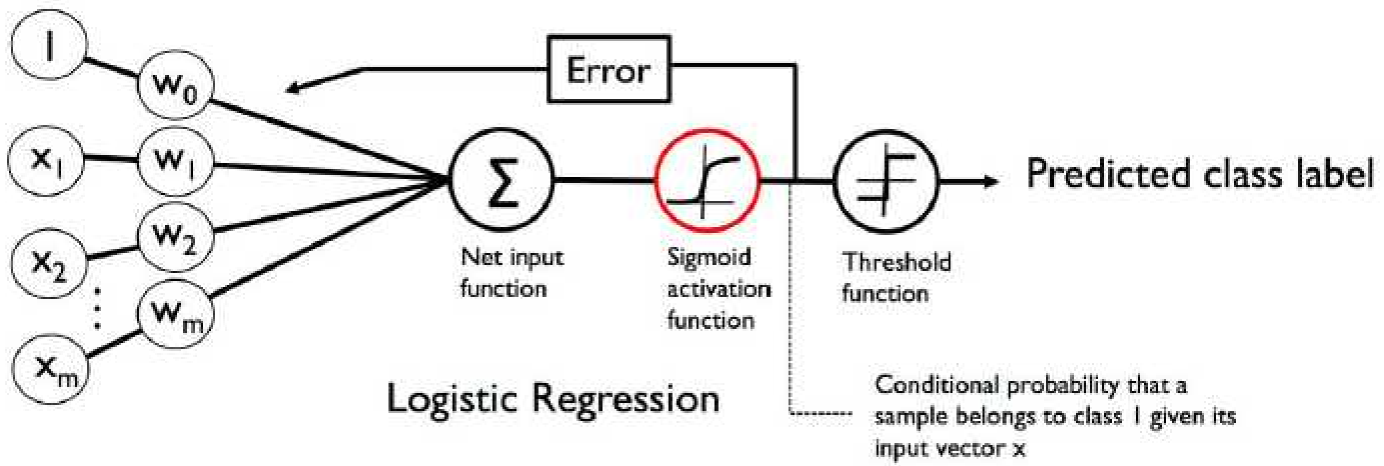




The output of sigmoid function is interpreted as the prob that a particular example belongs to class 1, $\phi(z) = p(y = 1|\mathbf{x};\ \mathbf{w})$.

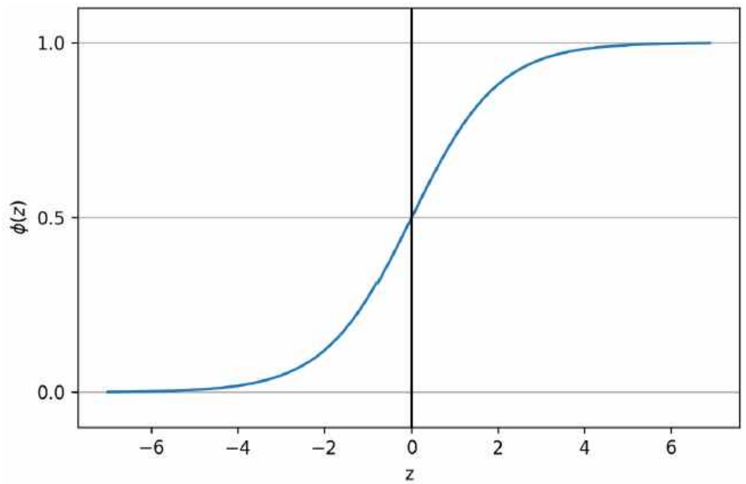
Logistic Regression





Logistic Regression

- The predicted probability is converted to a binary outcome



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

$$J(\mathbf{w}) = \sum_i \frac{1}{2}\left(\phi(z^{(i)}) - y^{(i)}\right)^2$$

Updating weights

- Likelihood

$$L(\mathbf{w}) = P(\mathbf{y} \mid \mathbf{x}; \mathbf{w}) =$$

Notice

Recall:
The output of sigmoid function is interpreted as the prob that a particular example belongs to class 1, $\phi(z) = p(y = 1|\mathbf{x};\ \mathbf{w})$.

- log-likelihood

$$\ell(\mathbf{w}) =$$

to be maximized!

- cost function, J(w) =

$$J(\mathbf{w}) = -\sum_i y^{(i)} \log\left(\phi(z^{(i)})\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi(z^{(i)})\right)$$

to be minimized!

Updating weights

partial derivative of log-likelihood

- $$\frac{\partial}{\partial w_j}\ell(w) = \frac{\partial}{\partial w_j}\sum_{i=1}^{n}\left[y^{(i)}\log(\phi(z^{(i)})) + (1-\phi(z^{(i)}))^{1-y^{(i)}}\right]$$
$$= \sum_{i=1}^{n}\underline{\frac{\partial}{\partial w_j}\left[y^{(i)}\log(\phi(z^{(i)})) + (1-\phi(z^{(i)}))^{1-y^{(i)}}\right]}$$

- $$\frac{\partial}{\partial w_j}\left[y^{(i)}\log(\phi(z^{(i)})) + (1-\phi(z^{(i)}))^{1-y^{(i)}}\right]$$
$$= \left(y^{(i)}\frac{1}{\phi(z^{(i)})} - (1-y^{(i)})\frac{1}{1-\phi(z^{(i)})}\right)\frac{\partial}{\partial w_j}\phi(z^{(i)})$$

removing superscript (i)

$$= \underline{\left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial}{\partial w_j}\phi(z)}$$

$$= \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\phi(z)(1-\phi(z))\frac{\partial}{\partial w_j}z$$
$$= \left(y(1-\phi(z)) - (1-y)\phi(z)\right)x_j$$
$$= \underline{(y-\phi(z))x_j}$$

- $$w_j := w_j + \eta\sum_{i=1}^{n}\underline{\left(y^{(i)} - \phi(z^{(i)})\right)x_j^{(i)}}$$

-

Updating weights

- maximizing log-likelihood is equal to minimizing the cost function J(w)
- gradient descent update rule

$$\Delta w_j =$$

)

Converting Adaline implementation to algorithm for logistic regresson

```python
class LogisticRegressionGD(object):
    """Logistic Regression Classifier using gradient descent.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.


    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    cost_ : list
      Logistic cost function value in each epoch.

    """
```

```python
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, z):
        """Compute logistic sigmoid activation"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, 0)
        # equivalent to:
        # return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

```python
    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the number of examples and
          n_features is the number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()

            # note that we compute the logistic `cost` now
            # instead of the sum of squared errors cost
            cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
            self.cost_.append(cost)
        return self
```

Logistic regression

- Consider only Iris-setosa and Iris-versicolor flowers (classes 0 and 1)

```
X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]

lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset,
         y_train_01_subset)

plot_decision_regions(X=X_train_01_subset,
                      y=y_train_01_subset,
                      classifier=lrgd)

plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/03_05.png', dpi=300)
plt.show()
```