Logistic Regression

- The biggest disadvantage of the Perceptron is that it never converges if the classes are not perfectly linearly separable.
- We will now take a look at another simple yet more powerful algorithm for **linear and binary classification** problems: logistic regression.
    - Note that, in spite of its name, logistic regression is a model for classification, not regression.
- The Logistic regression is a linear model for binary classification that **can be extended to multiclass classification** via the OvR technique.

- Logistic regression is a binary classifier: the output variable Y has two possible values 0 and 1
- Logistic regression is a probabilistic model: its goal is to model the probability of the positive class (i.e., the class that we want to predict), typically class 1.
    - The term positive event does not necessarily mean good, but refers to the event that we want to predict, for example, the probability that a patient has a certain disease; we can think of the positive event as class label $y = 1$.
- Consider a single input observation $\mathbf{x}$, which we will represent by a vector of features $[x_1, x_2, \dots, x_n]$, we want to know the probability that this observation $\mathbf{x}$ belongs to the positive class 1, $P(Y = 1|x)$.

- To explain the idea behind logistic regression as a probabilistic model, let's first introduce the **odds ratio**, which is the odds in favor of a particular event.
- The odds ratio can be written as $\frac{p}{(1-p)}$, where p stands for the probability of the positive event.
- We can then further define the **logit function**, which is simply the logarithm of the odds ratio (log-odds):

$$logit(p) = \log \frac{p}{(1-p)}$$

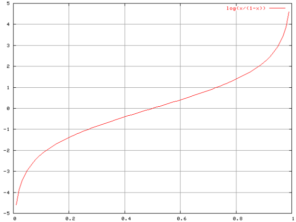
- The aim of the logistic regression algorithm is to compute:

$$logit(p(y=1|\mathbf{x}))$$

- The logit function takes input values in $[0, 1]$ and transforms them to values over the entire real number range, which we can use to express a linear relationship between feature values and the log-odds:

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

where $p(y = 1|\mathbf{x})$ is the conditional probability that a particular sample belongs to class 1 given its features $\mathbf{x}$.



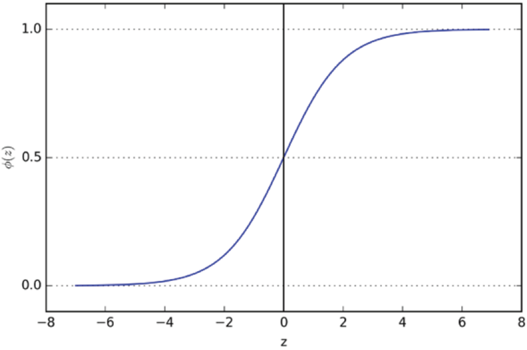
- What we are actually interested in is predicting the probability that a certain sample belongs to a particular class, which is the inverse form of the logit function: the logistic function, sometimes simply abbreviated as **sigmoid function** due to its characteristic S-shape.

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

where z is the net input, that is, the linear combination of weights and sample features and can be calculated as

$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

Logistic Regression
Probabilistic Models



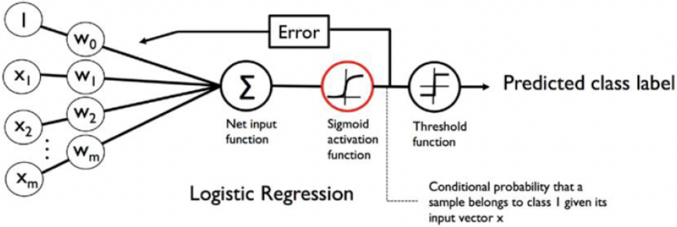
- Sigmoid function takes real number values as input and transforms them to values in the range [0, 1] with an intercept at $\phi(z) = 0.5$.

- The predicted probability can then simply be converted into a binary outcome via a quantizer (unit step function):

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

- If we look at the preceding sigmoid plot, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



- In Adaline, we used the identity function $\phi(z) = z$ as the activation function. In logistic regression, this activation function simply becomes the sigmoid function that we defined earlier.

- Let's consider a single observation x, we can define the output of the sigmoid function as the probability of particular sample to belong to the positive class (i.e, class 1):

$$\phi(z) = p(y = 1|x) = \hat{y} \text{ and } 1 - \phi(z) = p(y = 0|x) = 1 - \hat{y}$$

- We can express the probability $p(y|x)$ that our classifier produces for one observation as the following:

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

- Let's consider the function L that we want to maximise when we build a logistic regression model as:

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

where $P(\mathbf{y}|\mathbf{x}; \mathbf{w})$ is the probability that our classifier produces given its features $\mathbf{x}$ parameterized by the weights $\mathbf{w}$.

Logistic Regression
Utility Function

- Mathematically,

$$L(\mathbf{w})=P(\mathbf{y}|\mathbf{x};\mathbf{w})=\prod_{i=1}^{n} P(y^{(i)}|x^{(i)};\mathbf{w})=\prod_{i=1}^{n}(\phi(z^{(i)})^{y^{(i)}}(1-\phi(z^{(i)})^{1-y^{(i)}})$$

- We perform the $logL(\mathbf{w})$, since, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:

$$l(\mathbf{w}) = \log L(\mathbf{w}) = \sum_{i=1}^{n}\left[y^{(i)} \log\left(\phi(z^{(i)})\right) + (1-y^{(i)}) \log\left(1 - \phi(z^{(i)})\right)\right]$$

$$\log_a(b \cdot c) = \log_a(b) + \log_a(c)$$

$$\log_a\left(b^c\right) = c \log_a(b)$$

- We could use an optimization algorithm such as gradient ascent to maximize the log-likelihood function. Alternatively, let's rewrite the log-likelihood as a cost function, J, that can be minimized using **gradient descent**:

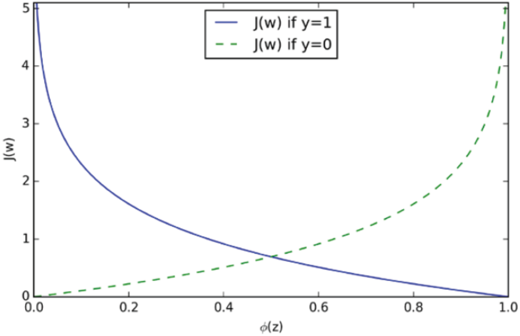
$$J(\mathbf{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi(z^{(i)})\right) - (1 - y^{(i)}) \log\left(1 - \phi(z^{(i)})\right) \right]$$

- To get a better grasp on this cost function, let's take a look at the cost that we calculate for one single-sample instance:

$$J(\phi(z); y; \mathbf{w}) = -y \log(\phi(z)) - (1 - y) \log(1 - \phi(z))$$

- Looking at the preceding equation, we can see that:

$$J(\phi(z); y; \mathbf{w}) = \begin{cases} -\log(\phi(z)) & \text{if } y = 1 \\ -\log(1 - \phi(z)) & \text{if } y = 0 \end{cases}$$



- The cost approaches 0 (plain blue line) if we correctly predict that a sample belongs to class 1. Similarly, we can see on the y axis that the cost also approaches 0 if we correctly predict $y = 0$ (dashed line). However, if the prediction is wrong, the cost goes towards infinity: we penalize wrong predictions with an increasingly larger cost.

- If we were to implement logistic regression ourselves, we could simply substitute the cost function J in our Adaline implementation with the new cost function:

$$J(\mathbf{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi(z^{(i)})\right) - (1 - y^{(i)}) \log\left(1 - \phi(z^{(i)})\right) \right]$$

- Also, we need to swap the linear activation function with the sigmoid activation and change the threshold function to return class labels 0 and 1 instead of -1 and 1.

```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) Sum-of-squares cost function value in each
        epoch.
    """
```

```
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_samples, n_features])
        Training vectors, where n_samples is the number of
        samples and n_features is the number of features.
    y : array-like, shape = [n_samples] 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
```

```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)
```

Logistic Regression

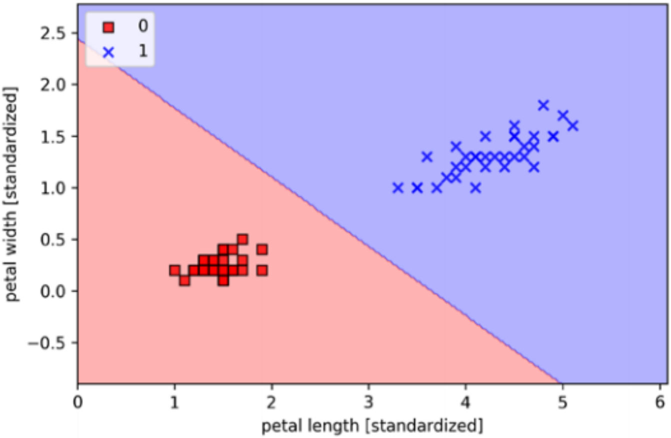
Converting an Adaline Implementation into an Algorithm for Logistic Regression

- When we fit a logistic regression model, we have to keep in mind that it only works for binary classification tasks. So, let us consider only Iris-setosa and Iris-versicolor flowers (classes 0 and 1) and check that our implementation of logistic regression works:

```
>>> 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.show()
```

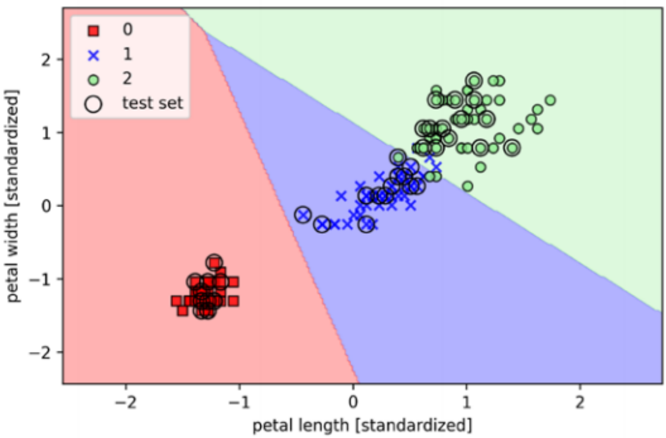
Logistic Regression

Converting an Adaline Implementation into an Algorithm for Logistic Regression



- Let's learn how to use scikit-learn's more optimized implementation of logistic regression that also supports multi-class settings off the shelf (OvR by default).

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, random_state=1)
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...     classifier=lr, test_idx=range(105, 150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```


0
1
2
test set
2
1
0
−1
−2
petal width [standardized]
−2
−1
0
1
2
petal length [standardized]

- The probability that training examples belong to a certain class can be computed using the predict_proba method. For example, we can predict the probabilities of the first three samples in the test set as follows:

```
>>> lr.predict_proba(X_test_std[:3,:])
```

- This code snippet returns the following array:

```
array([[ 3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

- The first row corresponds to the class-membership probabilities of the first instance, the second row corresponds to the class-membership probabilities of the second instance, and so forth.

- Notice that the rows sum all up to one, as expected (you can confirm this by executing `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`). The highest value in the first row is approximately 0.856, which means that the first sample belongs to class three (Iris-virginica) with a predicted probability of 85.6%.
- So, we can get the predicted class labels by identifying the largest column in each row, for example, using NumPy's `argmax` function:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

- The returned class indices are shown here (they correspond to Iris-virginica, Iris-setosa, and Iris-setosa):

```
array([2, 0, 0])
```

- The class labels we obtained from the preceding conditional probabilities is, of course, just a manual approach to calling the `predict` method directly, which we can quickly verify as follows:

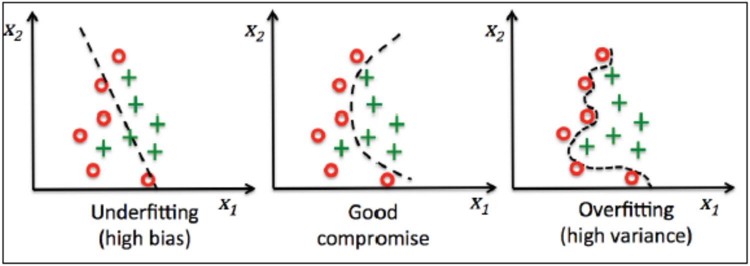
```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

- Lastly, a word of caution if you want to predict the class label of a single flower sample: sciki-learn expects a two-dimensional array as data input; thus, we have to convert a single row slice into such a format first. One way to convert a single row entry into a two-dimensional data array is to use NumPy's `reshape` method to add a new dimension, as demonstrated here:

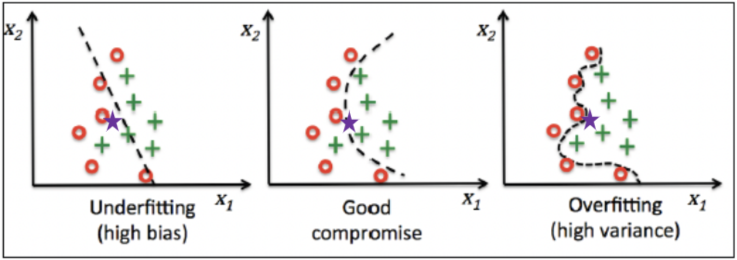
```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

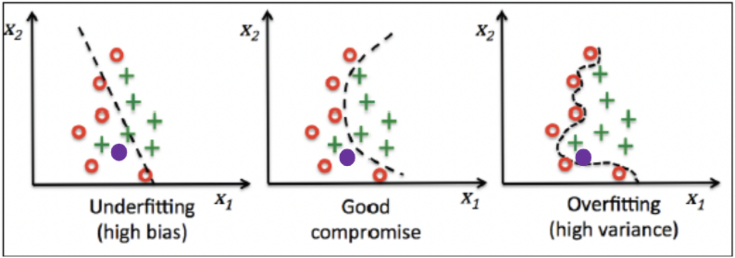
Overfitting vs Underfitting: Definitions

- **Overfitting** is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data).
- **Underfitting** means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.



Overfitting vs Underfitting: Examples





Regularization

- One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization.
- Motivation:
  - If a feature is perfectly predictive of the outcome, it will be assigned a very **high weight**.
  - If a feature will attempt to perfectly fit details of the training set could model noisy factors that just accidentally correlate with the class.
- Basic idea:
  - Regularization introduces additional information **to penalize high weights**.
  - Thus a setting of the weights that matches the training data perfectly—but uses many weights with high values to do so—will be penalized more than a setting that matches the data a little less well, but does so using smaller weights
- Regularization is a very useful method to filter out noise from data, and eventually prevent overfitting.

L2 Regularization

- The most common form of regularization is the so-called $L2$ regularization, which can be written as follows:

$$\frac{\lambda}{2}\|\mathbf{w}\|_2^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

where λ is the so-called **regularization parameter**.

Regularization in Logistic Regression

- In order to apply regularization, we just need to add the regularization term to the cost function that we defined for logistic regression to shrink the weights:

$$J(\mathbf{w})=\left[\sum_{i=1}^{n}\left(-y^{(i)}\log\left(\phi(z^{(i)})\right)+(1-y^{(i)})\right)\left(-\log\left(1-\phi(z^{(i)})\right)\right)\right]+\frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

- Via the regularization parameter λ, we can then control how well we fit the training data while keeping the weights small. By increasing the value of λ, we increase the regularization strength.
- The parameter C that is implemented for the Logistic Regression class in scikit-learn is directly related to the regularization parameter λ, which is its inverse:

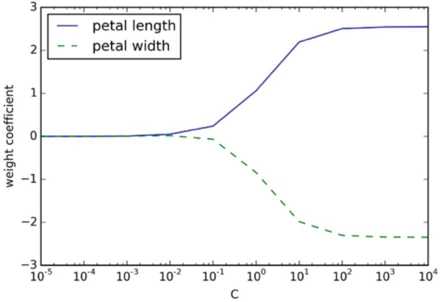
$$C = \frac{1}{\lambda}$$

- We can rewrite the regularized cost function of logistic regression as follows:

$$J(\mathbf{w})=C\Big[\sum_{i=1}^{n}\Big(-\log\Big(\phi(z^{(i)})\Big)+(1-y^{(i)})\Big)\Big(-\log\Big(1-\phi(z^{(i)})\Big)\Big)\Big]+\frac{1}{2}\|\mathbf{w}\|_2^2$$

- Consequently, decreasing the value of the inverse regularization parameter C means that we are increasing the regularization strength.

- We can visualize this consideration by plotting the $L2$ regularization path for the two weight coefficients:



- As we can see in the resulting plot, the weight coefficients shrink if we decrease the parameter C, that is, if we increase the regularization strength.

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c, random_state=1)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0], label='petal
    length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
    label='petal width')
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```