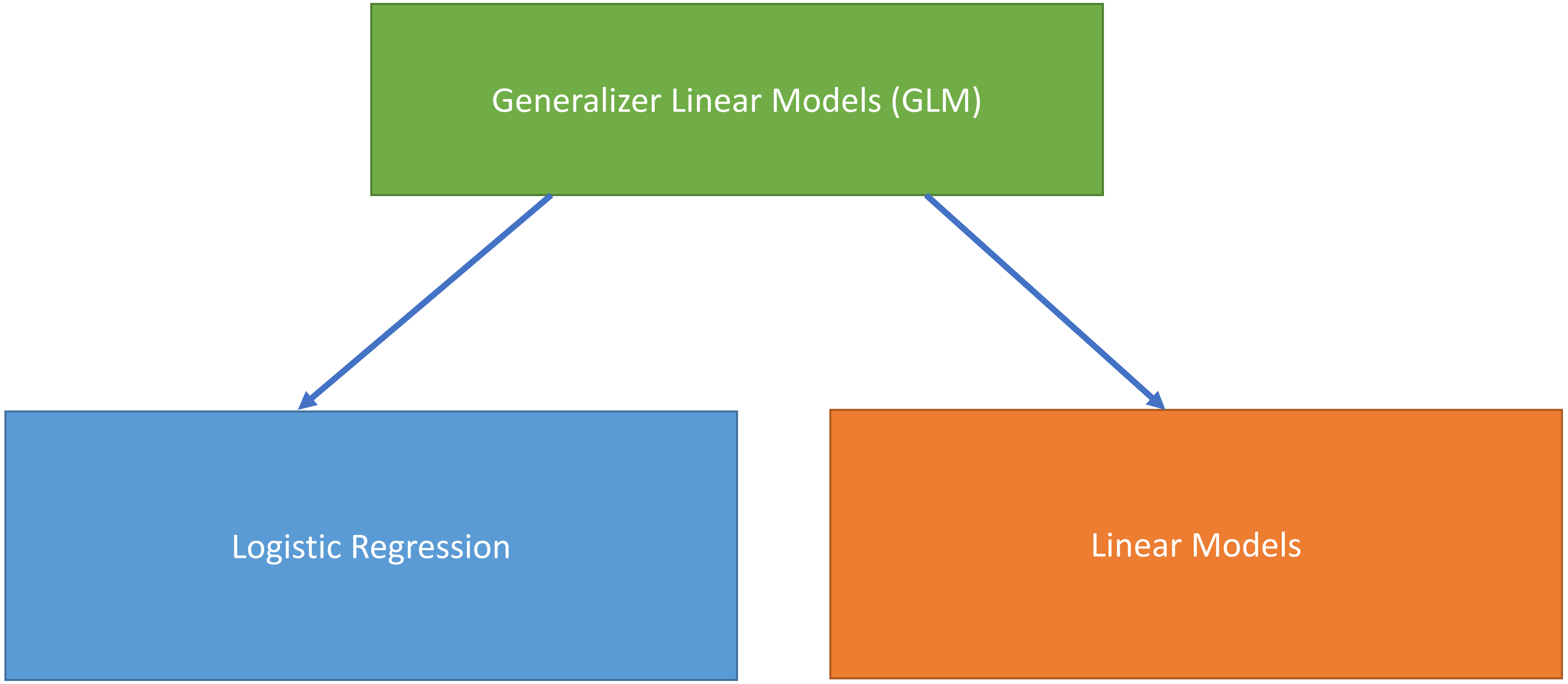
# Abdul Qayyum

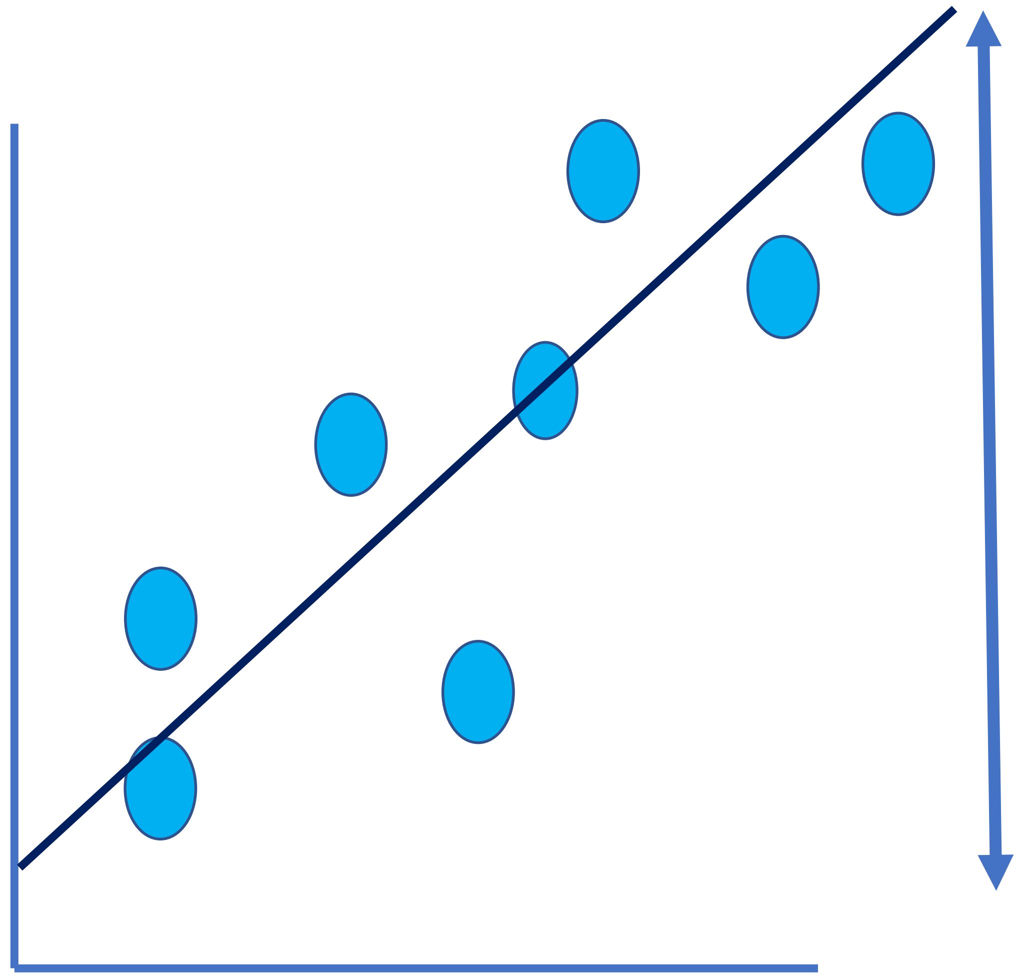
## Topic: Regression Models

## Linear and Logistic Regression



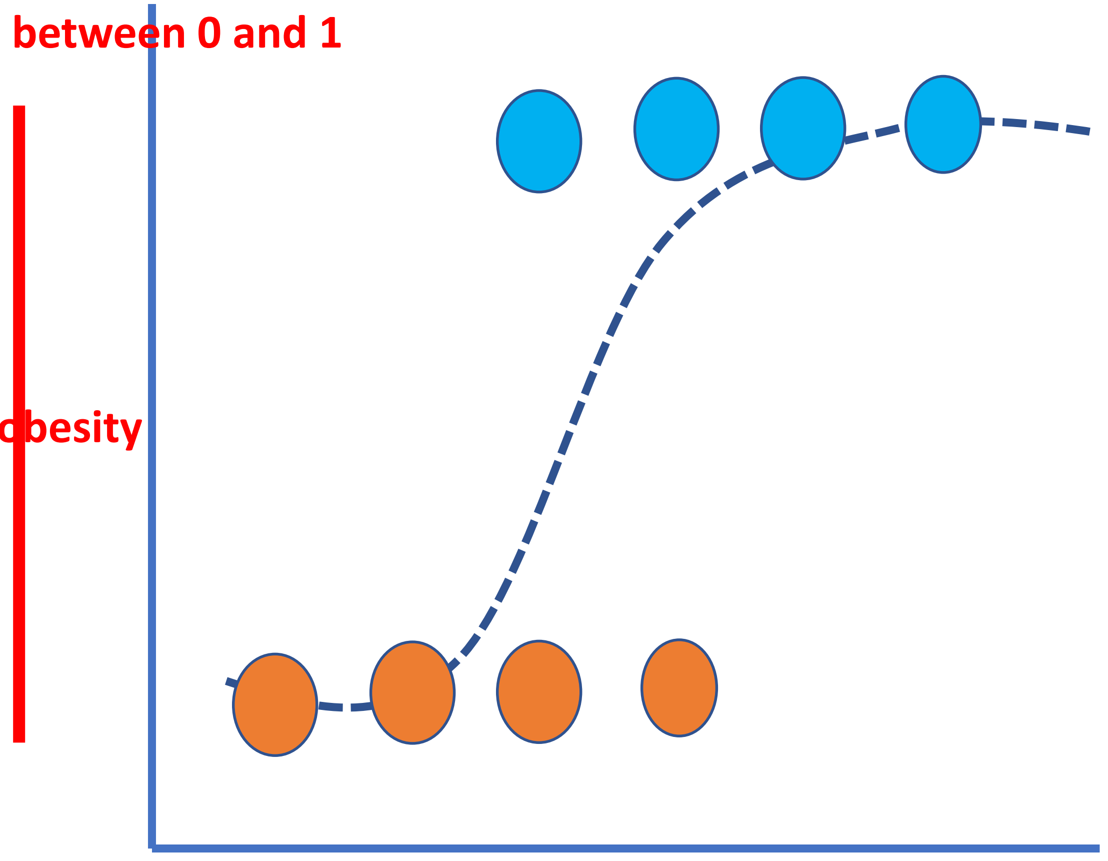
# Linear and Logistic Regression

With linear regression, the values on the y-axis can in theory be a number



With logistic regression the y-axis is confined to probability values between 0 and 1

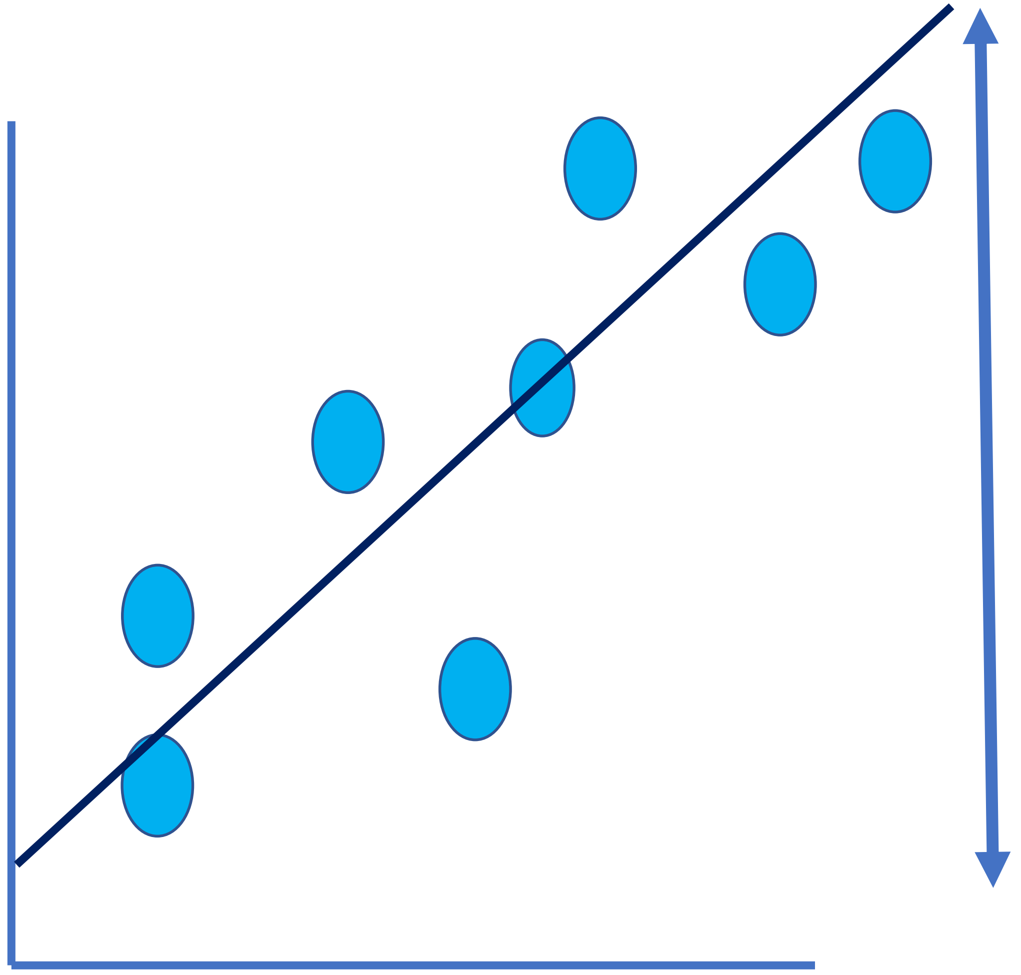
Probability of obesity



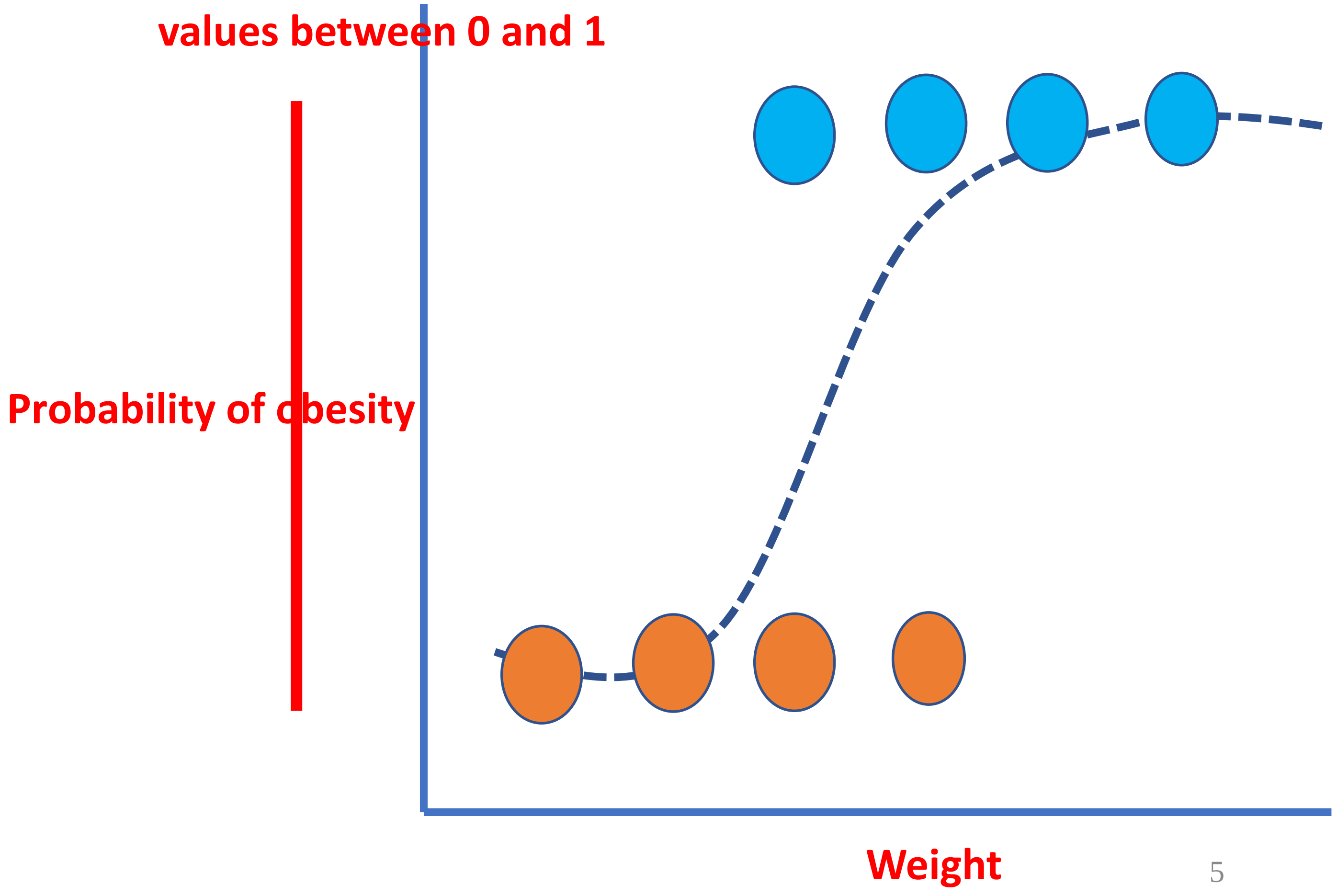
Weight

# Linear and Logistic Regression

With linear regression, the values on the y-axis can in theory be a number

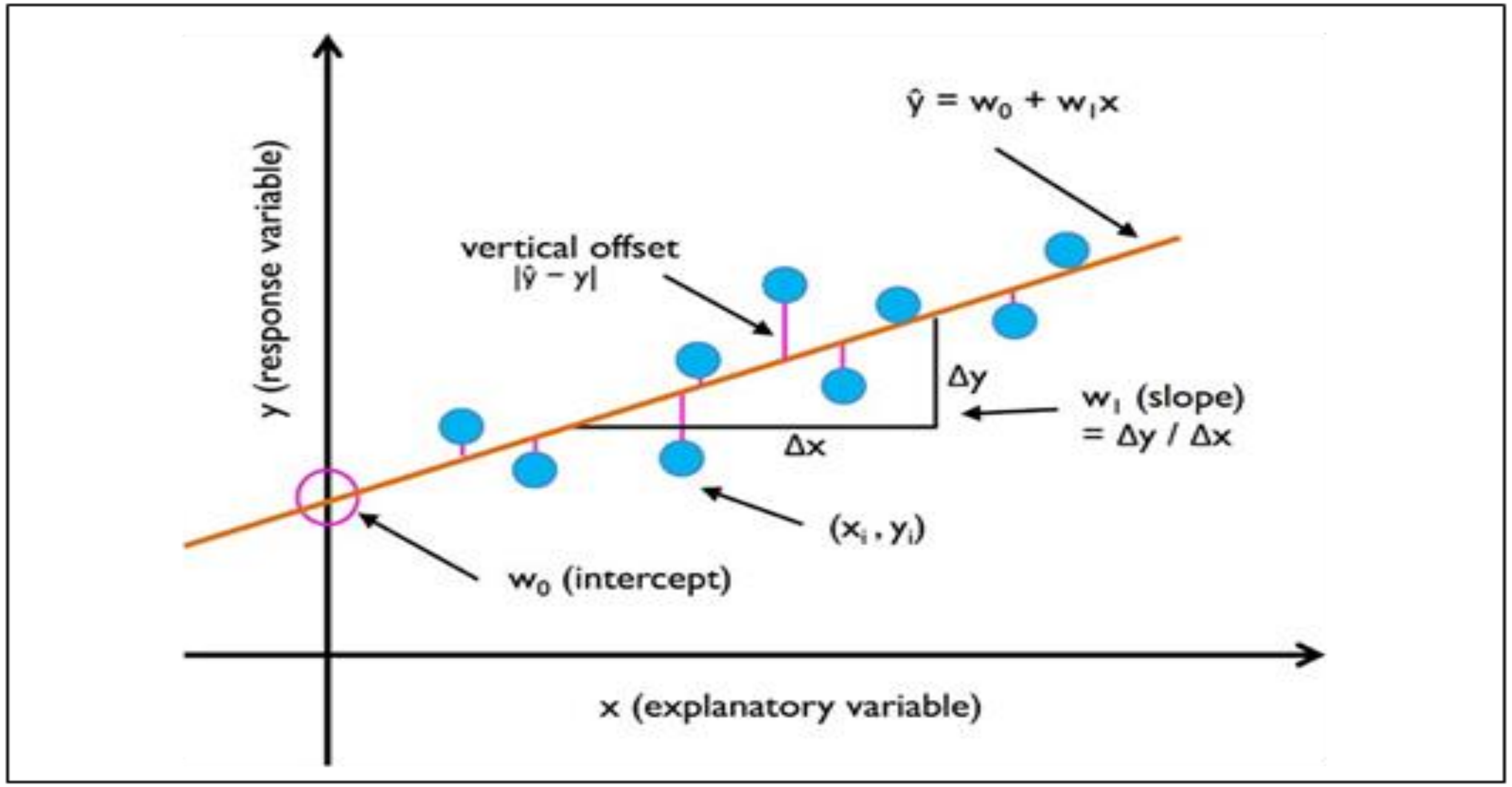


With logistic regression the y-axis is confined to probability values between 0 and 1



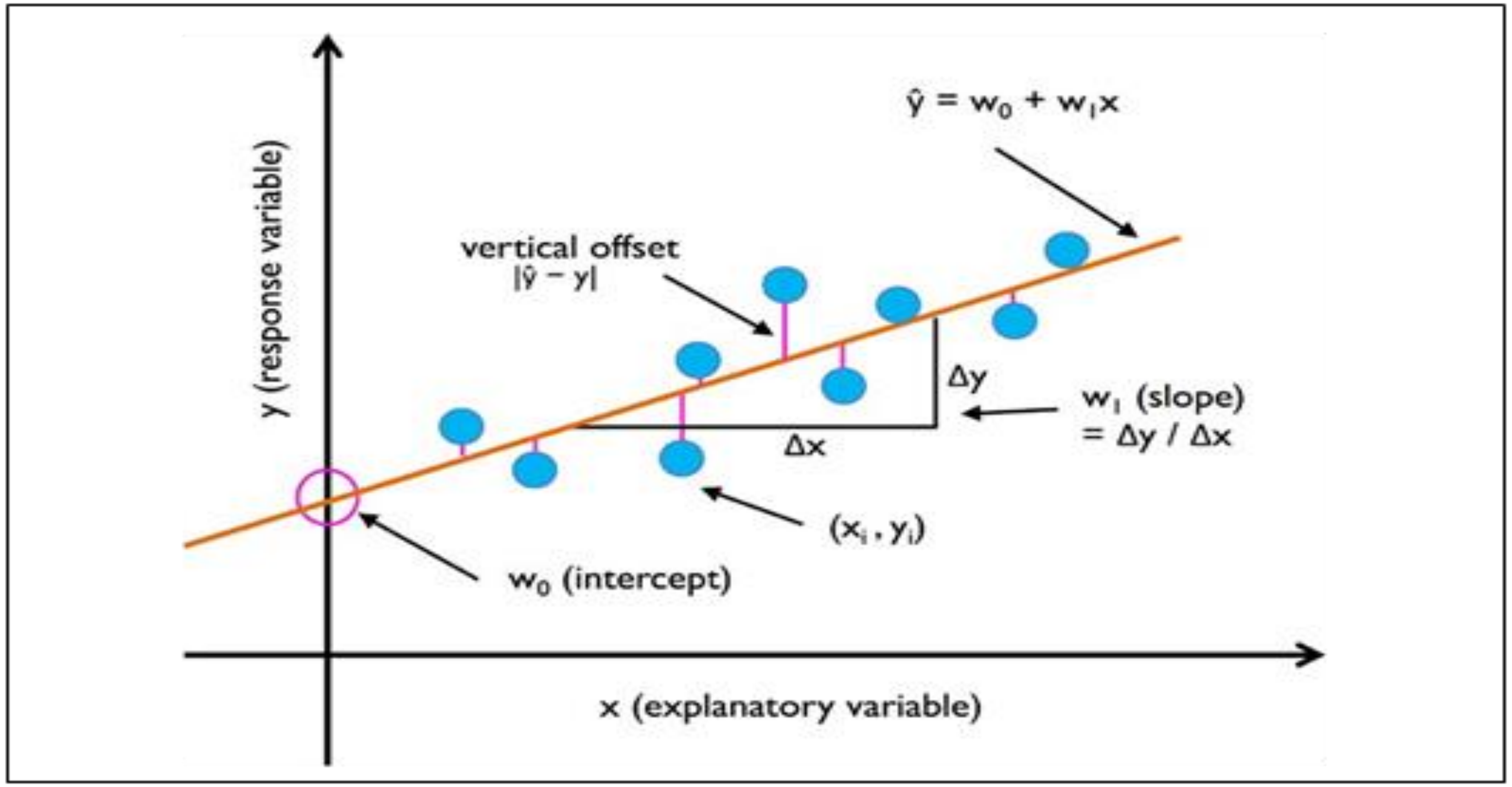
## Linear Regression

- Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the training examples, as shown in the following figure:



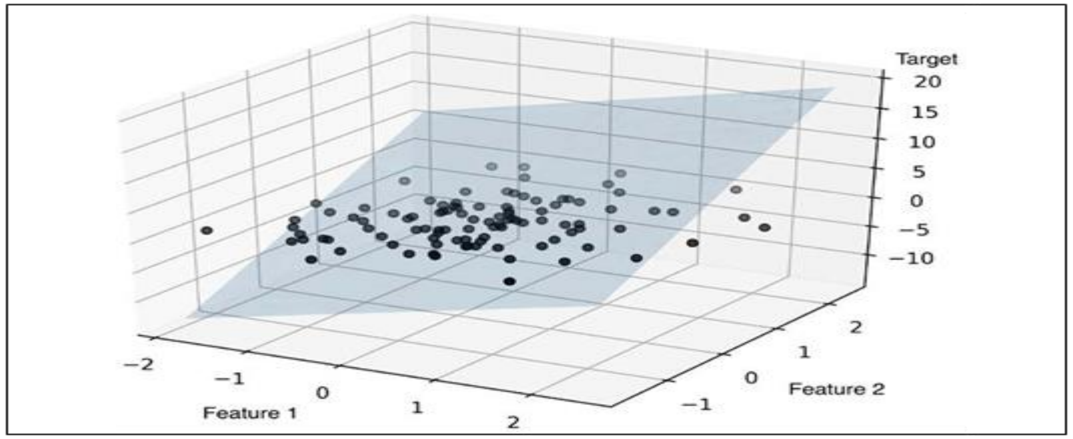
# Linear Regression

- This best-fitting line is also called the regression line, and the vertical lines from the regression line to the training examples are the so-called offsets or residuals—the errors of our prediction.



# Multiple Linear Regression

- **Multiple linear regression**
- The previous section introduced simple linear regression, a special case of linear regression with one explanatory variable. Of course, we can also generalize the linear regression model to multiple explanatory variables; this process is called multiple linear regression:
- $z = w_0\ x_0 + w_1\ x_1 + \ldots + w_m\ x_m = \sum_{j=0}^{m} \boldsymbol{w_j}\ \boldsymbol{x_j} = \boldsymbol{w^T x}$
- Here, $w0$ is the y axis intercept with $x0 = 1$.
- The following figure shows how the two-dimensional, fitted hyperplane of a
- multiple linear regression model with two features could look:



## Multiple Linear Regression

- **Multiple linear regression**
- As you can see, visualizations of multiple linear regression hyperplanes in a
- three-dimensional scatterplot are already challenging to interpret when looking at static figures. Since we have no good means of visualizing hyperplanes with two dimensions in a scatterplot (multiple linear regression models fit to datasets with three or more features), the examples and visualizations in this chapter will mainly focus on the univariate case, using simple linear regression.
- However, simple and multiple linear regression are based on the same concepts and the same evaluation techniques; the code implementations that we will discuss in this chapter are also compatible with both types of regression model.

# Implementation of Regression Models

- **Implementing an ordinary least squares linear regression model**
- At the beginning of this chapter, it was mentioned that linear regression can be understood as obtaining the best-fitting straight line through the examples of our training data. However, we have neither defined the term best-fitting nor have we discussed the different techniques of fitting such a model.
- **The ordinary least squares (OLS) method (sometimes also called linear least squares) to estimate the parameters of the linear regression line that minimizes the sum of the squared vertical distances (residuals or errors) to the training examples.**

- **Solving regression for regression parameters with gradient descent**
- Also, we defined a cost function, J(w), which we minimized to learn the weights via optimization algorithms, such as gradient descent (GD) and stochastic gradient descent (SGD).

## Implementation of Regression Models

- **Solving regression for regression parameters with gradient descent**
- Also, we defined a cost function, J(w), which we minimized to learn the weights via optimization algorithms, such as gradient descent (GD) and stochastic gradient descent (SGD).
- The cost function is the sum of squared errors (SSE), which is identical to the cost function that we use for OLS:
- $J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
- Here,ˆ is the predicted valueˆ = $\boldsymbol{w}^T\boldsymbol{x}$

## Evaluating the performance of Linear Regression Models

# Implementation of Regression Models

- **Evaluating the performance of linear regression models**
- Residual plots are a commonly used graphical tool for diagnosing regression models. They can help to detect nonlinearity and outliers, and check whether the errors are randomly distributed.
- Another useful quantitative measure of a model's performance is the so-called mean squared error (MSE), which is simply the averaged value of the SSE cost that we minimized to fit the linear regression model. The MSE is useful for comparing different regression models or for tuning their parameters via grid search and cross- validation, as it normalizes the SSE by the sample size:
- $MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$

Let's compute the MSE of our training and test predictions:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
...            mean_squared_error(y_train, y_train_pred),
...            mean_squared_error(y_test, y_test_pred))) MSE
train: 19.958, test: 27.196
```

# Implementation of Regression Models

- **Evaluating the performance of linear regression models**
- $MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
- You can see that the MSE on the training dataset is 19.96, and the MSE on the test dataset is much larger, with a value of 27.20, which is an indicator that our model is overfitting the training data in this case. However, please be aware that the MSE is unbounded in contrast to the classification accuracy, for example. In other words, the interpretation of the MSE depends on the dataset and feature scaling

```
Let's compute the MSE of our training and test predictions:
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
...           mean_squared_error(y_train, y_train_pred),
...           mean_squared_error(y_test, y_test_pred))) MSE
train: 19.958, test: 27.196
```

# Implementation of Regression Models

- **Evaluating the performance of linear regression models**
- Thus, it may sometimes be more useful to report the coefficient of determination ($R^2$), which can be understood as a standardized version of the MSE, for better interpretability of the model's performance. Or, in other words, $R^2$ is the fraction of response variance that is captured by the model. The $R^2$ value is defined as:
- $R^2 = 1 - \frac{SSE}{SST}$
- Here, SSE is the sum of squared errors and SST is the total sum of squares:
- SSE$= \sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2$
- In other words, SST is simply the variance of the response.
- Let's quickly show that $R^2$ is indeed just a rescaled version of the MSE:
- $R^2 = 1 - \frac{SSE}{SST}$
- $= \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}-\hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}-\mu_y\right)^2}$
- $=1-\frac{MSE}{Var(y)}$
- For the training dataset, the $R^2$ is bounded between 0 and 1, but it can become negative for the

# Implementation of Regression Models

- **Evaluating the performance of linear regression models**
- $R^2 = 1 - \frac{SSE}{SST}$
- $= \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}$
- $=1-\frac{MSE}{Var(y)}$
- For the training dataset, the $R^2$ is bounded between 0 and 1, but it can become negative for the test dataset. If $R^2$ = 1, the model fits the data perfectly with a corresponding MSE = 0.

we can compute by $R^2$ executing the following code:

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 train: %.3f, test: %.3f' %
...          (r2_score(y_train, y_train_pred),
...          r2_score(y_test, y_test_pred))) R^2 train: 0.765, test:
0.673
```

## Evaluating the performance of Linear Regression Models (**Regularized Methods for Regression**)

# Implementation of Regression Models

- **Using regularized methods for regression**
- *regularization is one approach to tackling the problem of overfitting by adding additional information, and thereby shrinking the parameter values of the model to induce a penalty against complexity. The most popular approaches to regularized linear regression are the so-called* ***Ridge Regression, least absolute shrinkage and selection operator (LASSO), and elastic Net.***
- Ridge Regression is an L2 penalized model where we simply add the squared sum of the weights to our least-squares cost function:
- $J_{Ridge}(\mathrm{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\|w\|_2^2$
- Here:
- $L_2 : \lambda\|w\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2$
- By increasing the value of hyperparameter $\lambda$ , we increase the regularization strength and thereby shrink the weights of our model. Please note that we don't regularize the intercept term, $w0$.

# Implementation of Regression Models

- **Using regularized methods for regression**
- An alternative approach that can lead to sparse models is LASSO. Depending on the regularization strength, certain weights can become zero, which also makes LASSO useful as a supervised feature selection technique:
- $J_{LASSO}(\mathrm{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\|w\|_1$
- Here, the L1 penalty for LASSO is defined as the sum of the absolute magnitudes of
- the model weights, as follows:
- $L_1: \lambda\|w\|_1 = \lambda \sum_{j=1}^{m}\left|w_j\right|$
- However, a limitation of LASSO is that it selects at most n features if m > n, where n is the number of training examples. This may be undesirable in certain applications of feature selection. In practice, however, this property of LASSO is often an advantage because it avoids saturated models
- Saturation of a model occurs if the number of training examples is equal to the number of features, which is a form
- of overparameterization. As a consequence, a saturated model can always fit the training data perfectly but is merely a form of interpolation and thus is not expected to generalize well.

## Implementation of Regression Models

- **Using regularized methods for regression**
- A compromise between Ridge Regression and LASSO is elastic net, which has an L1 penalty to generate sparsity and an L2 penalty such that it can be used for selecting more than n features if m > n:
- $J_{ElasticNet}(\mathrm{w})=\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}-\hat{y}^{(i)}\right)^2+\lambda_1\sum_{j=1}^{m}w_j^2+\lambda_2\sum_{j=1}^{m}\left|w_j\right|$
- Those regularized regression models are all available via scikit-learn, and their usage is similar to the regular regression model except that we have to specify the regularization strength via the parameter $\lambda$, for example, optimized via k-fold cross- validation.

A Ridge Regression model can be initialized via:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Note that the regularization strength is regulated by the parameter alpha, which is similar to the parameter $\lambda$. Likewise, we can initialize a LASSO regressor from the linear_model submodule:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

## Implementation of Regression Models

- **Turning a linear regression model into a curve – polynomial regression**
- In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

- $y = w_o + w_1 x + w_2 x^2 + \ldots . + w_d x^d$
- Here, d denotes the degree of the polynomial. Although we can use polynomial regression to model a nonlinear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients, w.
- We will see how we can add such polynomial terms to an existing dataset conveniently and fit a polynomial regression model.

# Implementation of Regression Models

- **Turning a linear regression model into a curve – polynomial regression**
- We will now learn how to use the PolynomialFeatures transformer class from scikit-learn to add a quadratic term (d = 2) to a simple regression problem with one explanatory variable. Then, we will compare the polynomial to the linear fit by following these steps:
- 1. Add a second-degree polynomial term:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>>       X        =        np.array([ 258.0, 270.0,      294.0,    320.0,    342.0,
...                         368.0, 396.0,       446.0,    480.0,    586.0])\
...                         [:, np.newaxis]
>>>       y        =        np.array([ 236.4, 234.4,      252.8,    298.6,    314.2,
...                         342.2, 360.8,       368.0,    391.2,    390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

# Implementation of Regression Models

- **Turning a linear regression model into a curve – polynomial regression**
- 2. Fit a simple linear regression model for comparison:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

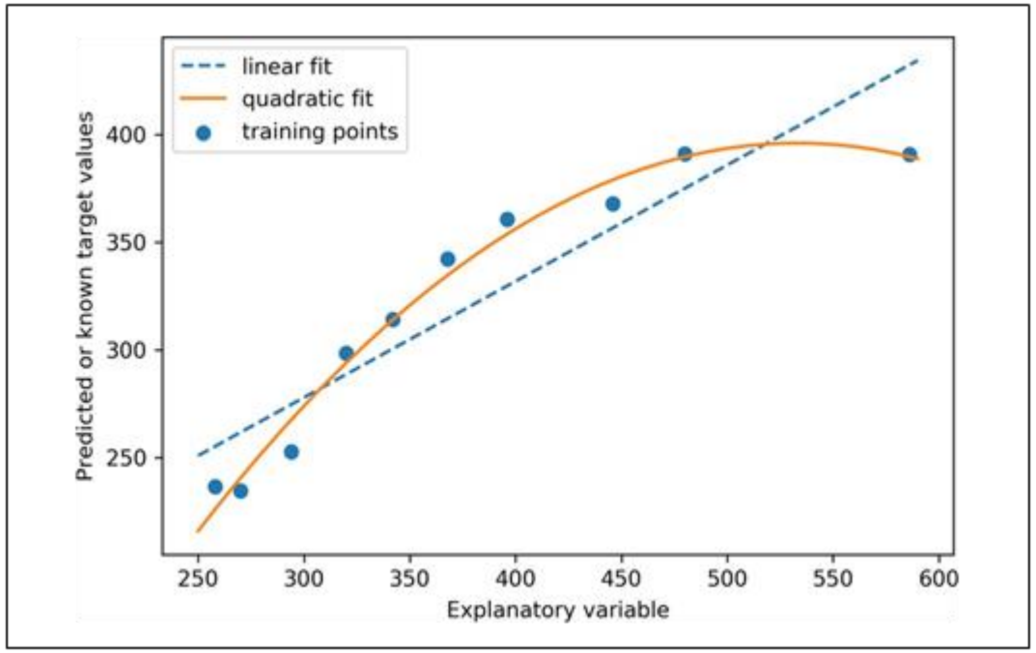
- 3. Fit a multiple regression model on the transformed features for polynomial regression:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

# Implementation of Regression Models

- **Turning a linear regression model into a curve – polynomial regression**
- 4. Plot the results:

```
>>> plt.scatter(X, y, label='Training points')
>>> plt.plot(X_fit, y_lin_fit,
...             label='Linear fit', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...             label='Quadratic fit')
>>> plt.xlabel('Explanatory variable')
>>> plt.ylabel('Predicted or known target values')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```



In the resulting plot, you can see that the polynomial fit captures the relationship between the response and explanatory variables much better than the linear fit:

# Implementation of Regression Models

- **Turning a linear regression model into a curve – polynomial regression**
- However, you should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting.
- Thus, in practice, it is always recommended to evaluate the performance of the model on a separate test dataset to estimate the generalization performance.
- In addition, polynomial features are not always the best choice for modeling nonlinear relationships.
- For instance, my perception is that this relationship between the two variables looks quite similar to an exponential function:
- $f(x) = e{-}x$

- Since the natural logarithm of an exponential function is a straight line, I assume that such a log-transformation can be usefully applied here:

- $\log(f(x)) = -x$

# Implementation of Regression Models

- **Log transformation of features**
- However, you should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting.
- Thus, in practice, it is always recommended to evaluate the performance of the model on a separate test dataset to estimate the generalization performance.
- In addition, polynomial features are not always the best choice for modeling nonlinear relationships.
- For instance, my perception is that this relationship between the two variables looks quite similar to an exponential function:
- $f(x) = e{-}x$

- Since the natural logarithm of an exponential function is a straight line, I assume that such a log-transformation can be usefully applied here:

- $\log(f(x)) = -x$

# Implementation of Regression Models

- **Log transformation of features**
- Let's test this hypothesis by executing the following code:

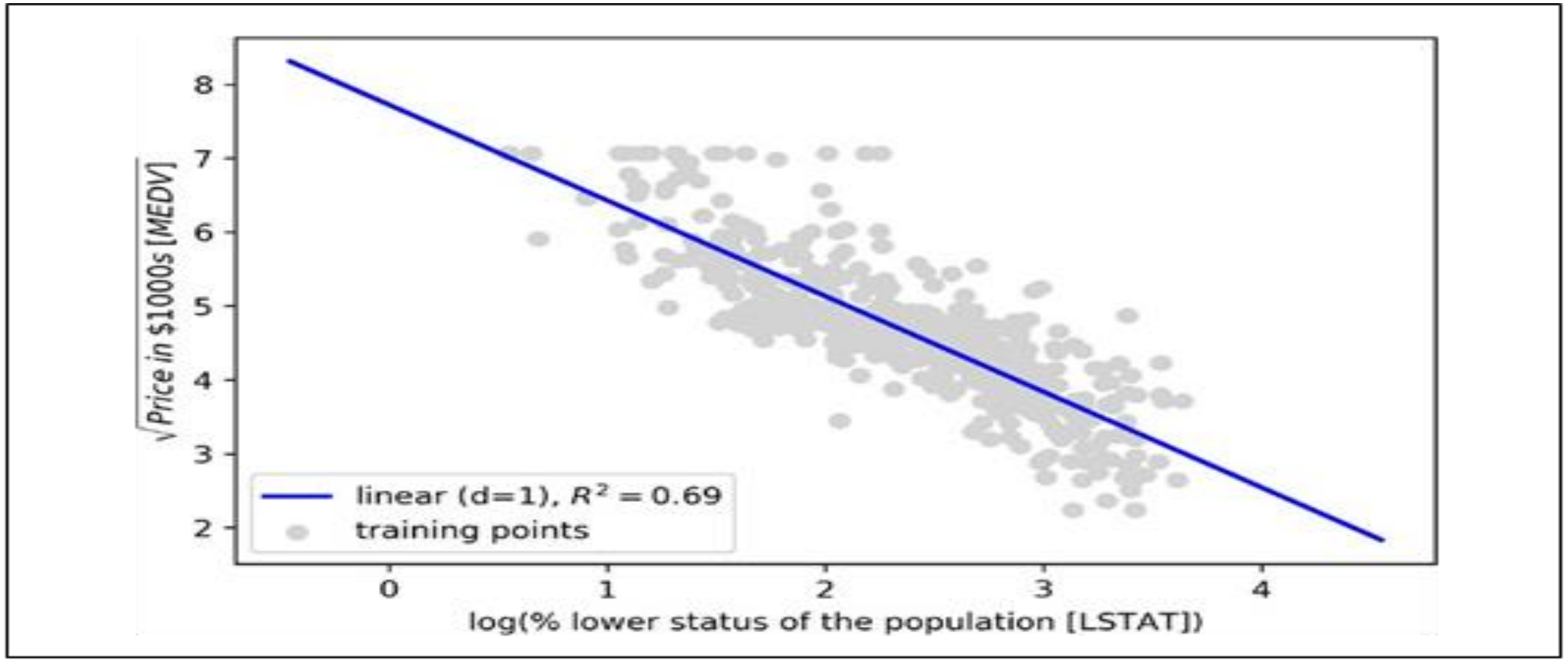
```
>>> # transform features
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)
>>>
>>> # fit features
>>> X_fit = np.arange(X_log.min()-1,
...             X_log.max()+1, 1)[:, np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))
```

```
>>> # plot results
>>> plt.scatter(X_log, y_sqrt,
...             label='Training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...             label='Linear (d=1), $R^2=%.2f$' % linear_r2,
...             color='blue',
...             lw=2)

>>> plt.xlabel('log(% lower status of the population
[LSTAT])')
>>> plt.ylabel('$\sqrt{Price \; in \; \$1000s \; [MEDV]}$')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

# Implementation of Regression Models

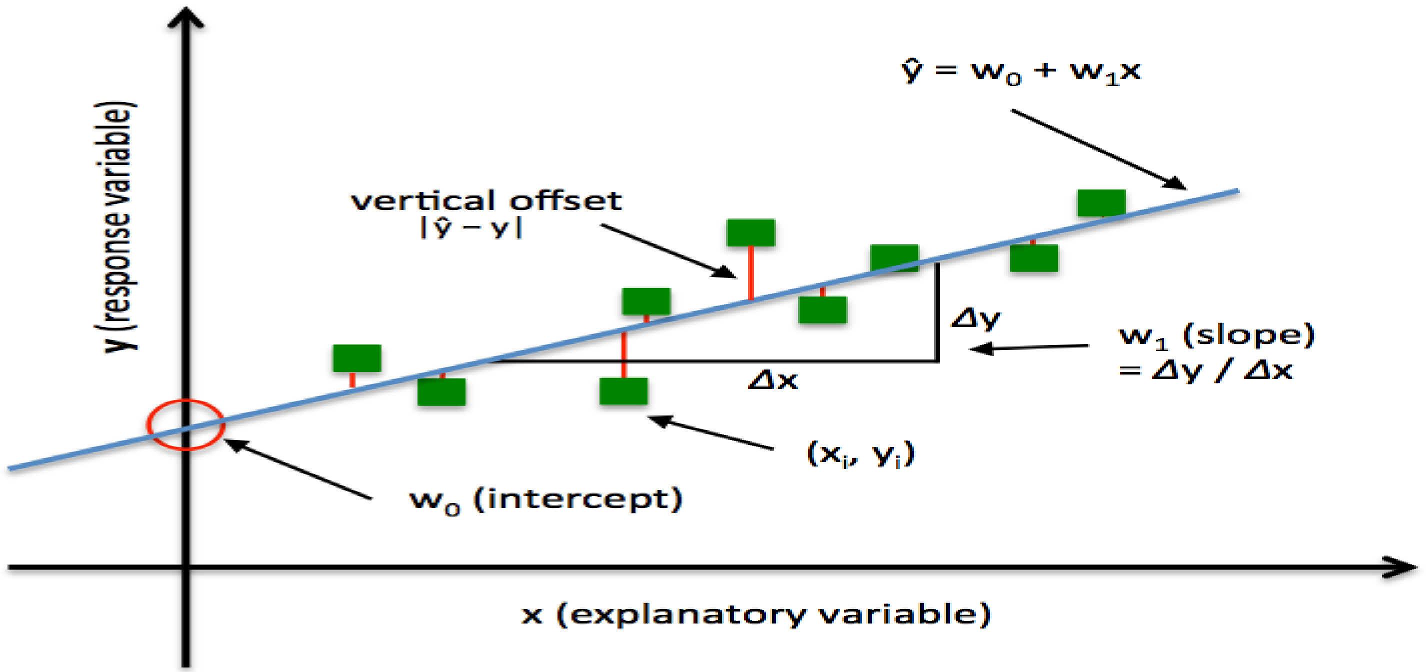
- **Log transformation of features**
- After transforming the explanatory onto the log space and taking the square root of the target variables, we were able to capture the relationship between the two variables with a linear regression line that seems to fit the data better ($R2$ = 0.69)
- than any of the previous polynomial feature transformations**:**



# Implementation of Regression Models

- **Log transformation of features**
- **LinearRegression**
- **A implementation of Ordinary Least Squares simple and multiple linear regression.**

- **from mlxtend.regressor import LinearRegression**



# Implementation of Regression Models

- **A implementation of Ordinary Least Squares simple and multiple linear regression.**
- In Ordinary Least Squares (OLS) Linear Regression, our goal is to find the line (or hyperplane) that minimizes the vertical offsets.
- Or in other words, we define the best-fitting line as the line that minimizes the sum of squared errors (SSE) or mean squared error (MSE) between our target variable (y) and our predicted output over all samples i in our dataset of size n.
- $SSE = \sum_i \left(target^{(i)} - output^{(i)}\right)^2$
- MSE=$\frac{1}{n}$xSSE
- Now, LinearRegression implements a linear regression model for performing ordinary least squares regression using one of the following five approaches:

**Normal Equations**
**QR Decomposition Method**
**SVD (Singular Value Decomposition) method**
**Gradient Descent**
**Stochastic Gradient Descent**

# Implementation of Regression Models

- **Normal Equations (closed-form solution)**
- The closed-form solution should be preferred for "smaller" datasets where calculating (a "costly") matrix inverse is not a concern.
- For very large datasets, or datasets where the inverse of [XTX] may not exist (the matrix is non-invertible or singular, e.g., in case of perfect multicollinearity), the QR, SVD or gradient descent approaches are to be preferred.
- *The linear function (linear regression model) is defined as:*
- y= $w_0\, x_0 + w_1\, x_1 + \ldots + w_m\, x_m = \sum_{\boldsymbol{j=0}}^{\boldsymbol{m}} \boldsymbol{w_j}\, \boldsymbol{x_j} = \boldsymbol{w^T x}$
- where y is the response variable, x is an m-dimensional sample vector, and w is the weight vector (vector of coefficients). Note that w0 represents the y-axis intercept of the model and therefore x0=1.
- Using the closed-form solution (normal equation), we compute the weights of the model as follows:
- $w = (X^T X)^{-1}\, X^T$y

# Implementation of Regression Models

- **Stable OLS via QR Factorization**
- Using the closed-form solution (normal equation), we compute the weights of the model as follows:
- The QR decomposition method offers a more numerically stable alternative to the closed-form, analytical solution based on the "normal equations," and it can be used to compute the inverse of large matrices more efficiently.
- QR decomposition method decomposes given matrix into two matrices for which an inverse can be easily obtained. For instance, a given matrix $X \in R^{n \times m}$
- the QR decomposition into two matrices is:
- $X = QR$
- Where
- $Q \in R^{n \times m}$ is an orthonormal matrix, such that $Q^T Q = QQ^T = \mathrm{I}$. The second matrix $R \in R^{m \times m}$ is an upper triangular matrix.
- The weight parameters of the ordinary least squares regression model can then be computed as follows [1]:
- $w = R^{-1}\, Q^T \mathrm{y}$

## Implementation of Regression Models

- **Stable OLS via Singular Value Decomposition**
- Another alternative way for obtaining the OLS model weights in a numerically stable fashion is by Singular Value Decomposition (SVD), which is defined as:
- $X = U\sum V^T$
- for a given matrix X.
- Then, it can be shown that the pseudo-inverse of X, $X^+$ ,can be obtained as follows [1]:
- $X^+ = U\sum^+ V^T$
- Note that while Σ is the diagonal matrix consisting of singular values of X, $\sum^+$ is the diagonal matrix consisting of the reciprocals of the singular values.
- The model weights can then be computed as follows:
- $w = X^+ y$
- Please note that this OLS method is computationally most inefficient. However, it is a useful approach when the direct method (normal equations) or QR factorization cannot be applied or the normal equations (via $X^T X$) are ill-conditioned [3].

# Implementation of Regression Models

- **Gradient Descent (GD) and Stochastic Gradient Descent (SGD)**
- Random shuffling is implemented as:
- **for one or more epochs**
- **randomly shuffle samples in the training set**
- **for training sample i**
- **compute gradients and perform weight updates**

References
[1] Chapter 3, page 55, Linear Methods for Regression. Trevor Hastie; Robert Tibshirani; Jerome Friedman (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd ed.). New York: Springer. (ISBN 978–0–387–84858–7)
[2] G. Strang, Linear Algebra and Its Applications, 2nd Ed., Orlando, FL, Academic Press, Inc., 1980, pp. 139-142.
[3] Douglas Wilhelm Harder. Numerical Analysis for Engineering. Section 4.8, ill-conditioned Matrices

## Implementation of Regression Models

- **Example 1 - Closed Form Solution**

```
import numpy as np
import matplotlib.pyplot as plt
from mlxtend.regressor
import LinearRegression
X = np.array([ 1.0, 2.1, 3.6, 4.2, 6])[:, np.newaxis]
y = np.array([ 1.0, 2.0, 3.0, 4.0, 5.0])
ne_lr = LinearRegression() ne_lr.fit(X, y)
print('Intercept: %.2f' % ne_lr.b_)
print('Slope: %.2f' % ne_lr.w_[0])
def lin_regplot(X, y, model):
plt.scatter(X, y, c='blue’)
plt.plot(X, model.predict(X), color='red’)
return lin_regplot(X, y, ne_lr)
plt.show()
```

# Implementation of Regression Models

- **Example 2 - QR decomposition method**

```
import numpy as np
 import matplotlib.pyplot as plt
from mlxtend.regressor
import LinearRegression
X = np.array([ 1.0, 2.1, 3.6, 4.2, 6])[:, np.newaxis]
y = np.array([ 1.0, 2.0, 3.0, 4.0, 5.0])
qr_lr = LinearRegression(method='qr’)
qr_lr.fit(X, y) print('Intercept: %.2f' % qr_lr.b_)
print('Slope: %.2f' % qr_lr.w_[0])
def lin_regplot(X, y, model):
plt.scatter(X, y, c='blue’)
plt.plot(X, model.predict(X), color='red’)
return lin_regplot(X, y, qr_lr)
 plt.show()
```

## Implementation of Regression Models

- **Example 3 - SVD method**

```
import numpy as np
 import matplotlib.pyplot as plt
 from mlxtend.regressor
import LinearRegression
X = np.array([ 1.0, 2.1, 3.6, 4.2, 6])[:, np.newaxis]
y = np.array([ 1.0, 2.0, 3.0, 4.0, 5.0])
svd_lr = LinearRegression(method='svd’)
svd_lr.fit(X, y) print('Intercept: %.2f' %svd_lr.b_)
 print('Slope: %.2f' % svd_lr.w_[0])
def lin_regplot(X, y, model):
 plt.scatter(X, y, c='blue’)
plt.plot(X, model.predict(X), color='red’)
return lin_regplot(X, y, svd_lr)
plt.show()
```

# Implementation of Regression Models

- **Example 4 - Gradient Descent**

```
import numpy as np
import matplotlib.pyplot as plt
from mlxtend.regressor
import LinearRegression
X = np.array([ 1.0, 2.1, 3.6, 4.2, 6])[:, np.newaxis]
y = np.array([ 1.0, 2.0, 3.0, 4.0, 5.0])
gd_lr = LinearRegression(method='sgd', eta=0.005, epochs=100, minibatches=1,
random_seed=123, print_progress=3)
 gd_lr.fit(X, y) print('Intercept: %.2f' % gd_lr.b_)
print('Slope: %.2f' % gd_lr.w_)
def lin_regplot(X, y, model):
plt.scatter(X, y, c='blue')
plt.plot(X, model.predict(X), color='red')
return lin_regplot(X, y, gd_lr)
plt.show()
```

# Implementation of Regression Models

- **Example 4 - Gradient Descent**

```
import numpy as np
import matplotlib.pyplot as plt
from mlxtend.regressor
import LinearRegression
X = np.array([ 1.0, 2.1, 3.6, 4.2, 6])[:, np.newaxis]
y = np.array([ 1.0, 2.0, 3.0, 4.0, 5.0])
gd_lr = LinearRegression(method='sgd', eta=0.005, epochs=100,
minibatches=1, random_seed=123, print_progress=3)
 gd_lr.fit(X, y) print('Intercept: %.2f' % gd_lr.b_)
print('Slope: %.2f' % gd_lr.w_)
def lin_regplot(X, y, model):
plt.scatter(X, y, c='blue’)
plt.plot(X, model.predict(X), color='red’)
return lin_regplot(X, y, gd_lr)
plt.show()
```

***# Visualizing the cost to check for convergence and plotting the linear model:***

```
plt.plot(range(1, gd_lr.epochs+1),
gd_lr.cost_)
plt.xlabel('Epochs’)
plt.ylabel('Cost’)
plt.ylim([0, 0.2])
plt.tight_layout()
plt.show()
```

## Implementation of Regression Models

- **Example 5 - Stochastic Gradient Descent**

```
import numpy as np
import matplotlib.pyplot as plt
from mlxtend.regressor
import LinearRegression
X = np.array([ 1.0, 2.1, 3.6, 4.2, 6])[:, np.newaxis]
y = np.array([ 1.0, 2.0, 3.0, 4.0, 5.0])
sgd_lr = LinearRegression(method='sgd', eta=0.01, epochs=100,
random_seed=0, minibatches=len(y))
sgd_lr.fit(X, y)
print('Intercept: %.2f' % sgd_lr.w_)
print('Slope: %.2f' % sgd_lr.b_)
def lin_regplot(X, y, model):
plt.scatter(X, y, c='blue’)
plt.plot(X, model.predict(X), color='red’)
return lin_regplot(X, y, sgd_lr)
plt.show()
```

```
plt.plot(range(1, sgd_lr.epochs+1),
sgd_lr.cost_)
 plt.xlabel('Epochs’)
plt.ylabel('Cost’)
plt.ylim([0, 0.2])
plt.tight_layout()
plt.show()
```

## Implementation of Regression Models

- **Example 6 - Stochastic Gradient Descent with Minibatches**

```
import numpy as np
import matplotlib.pyplot as plt
from mlxtend.regressor
import LinearRegression
X = np.array([ 1.0, 2.1, 3.6, 4.2, 6])[:, np.newaxis]
y = np.array([ 1.0, 2.0, 3.0, 4.0, 5.0])
sgd_lr = LinearRegression(method='sgd', eta=0.01, epochs=100,
random_seed=0, minibatches=3)
sgd_lr.fit(X, y)
 print('Intercept: %.2f' % sgd_lr.b_)
print('Slope: %.2f' % sgd_lr.w_)
def lin_regplot(X, y, model):
plt.scatter(X, y, c='blue’)
plt.plot(X, model.predict(X), color='red’)
return lin_regplot(X, y, sgd_lr)
plt.show()
```

```
plt.plot(range(1, sgd_lr.epochs+1),
sgd_lr.cost_)
 plt.xlabel('Epochs’)
plt.ylabel('Cost’)
 plt.ylim([0, 0.2])
plt.tight_layout()
 plt.show()
```

## Logistic Regression

## Logistic Regression

- Logistic regressions ability to provide probabilities and classify new samples using continuous and discrete measurements makes it a popular machine learning method
- One big difference between linear regression and logistic regression is how the line is fit to the data
- With linear regression, we fit the line using “least squares” in other words,
- We find the line that minimizes the sum of the squares of these residuals.
- We also use the residuals to calculate the R2 and to compare simple models to complicated models
- Logistic regression doesn’t have the same concept of a residual, so it cant use least squares and it cant calculate R2
- Instead it uses something called “maximum likelihood”
- Logistic regression can be use to classify the samples and it can use different types of data to do that classification and it can also be used to asses what variables are useful for classifying samples.

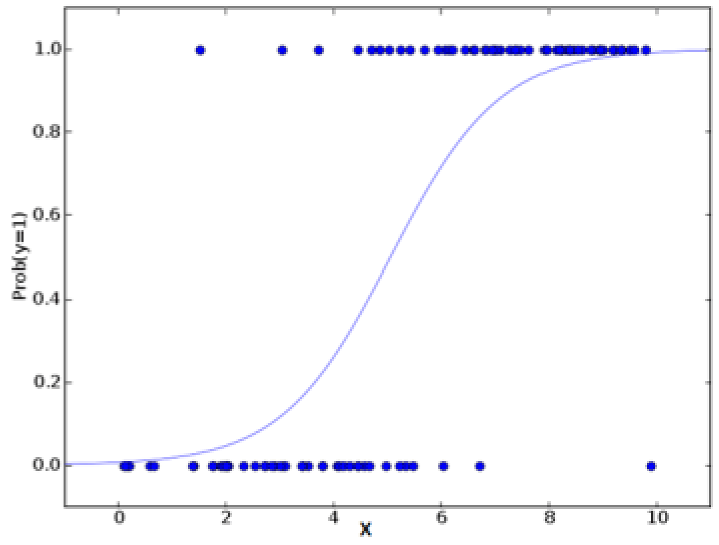
# Logistic Regression

- Logistic Regression is one of the basic and popular algorithm to solve a classification problem. It is named as ‘Logistic Regression’, because it’s underlying technique is quite the same as Linear Regression. The term “Logistic” is taken from the Logit function that is used in this method of classification.
- Logistic Regression is one of the basic and popular algorithm to solve a classification problem. It is named as ‘Logistic Regression’, because it’s underlying technique is quite the same as Linear Regression. The term “Logistic” is taken from the **Logit function** that is used in this method of classification.
- An explanation of logistic regression can begin with an explanation of the standard logistic function. The logistic function is a Sigmoid function, which takes any real value between zero and one. It is defined as

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$$

And if we plot it, the graph will be S curve,

# Logistic Regression



Let’s consider t as linear function in a univariate regression model.

$$t = \beta_0 + \beta_1 x$$

# Logistic Regression

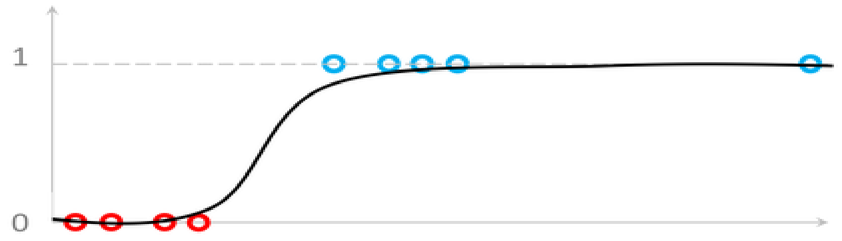
Let’s consider t as linear function in a univariate regression model.

$$t = \beta_0 + \beta_1 x$$

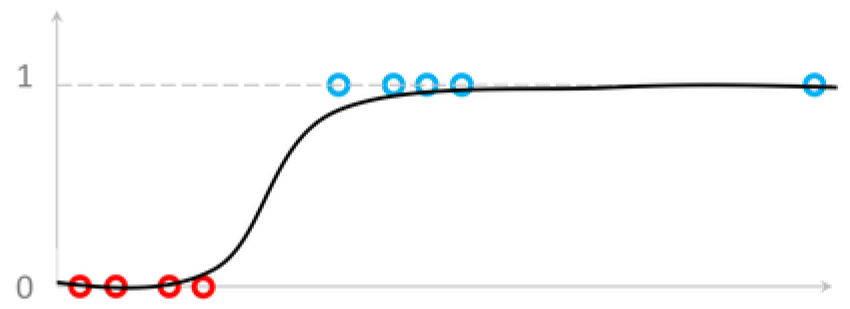
So the Logistic Equation will become

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

Now, when logistic regression model come across an outlier, it will take care of it.



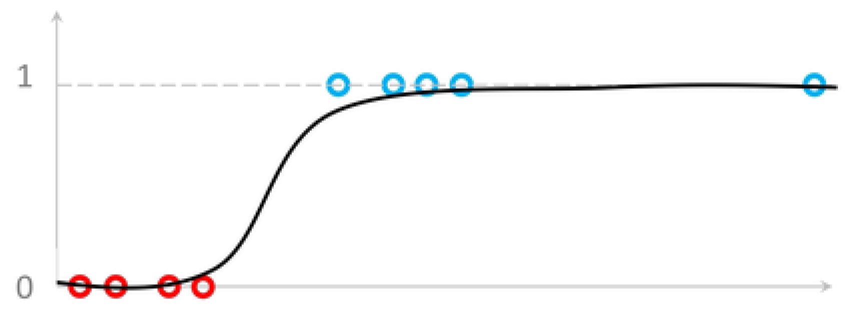
## Logistic Regression



But sometime it will shift its y axis to left or right depending on outliers positions.

What is Decision Boundary?
Decision boundary helps to differentiate probabilities into positive class and negative class.
Linear Decision Boundary

# Logistic Regression

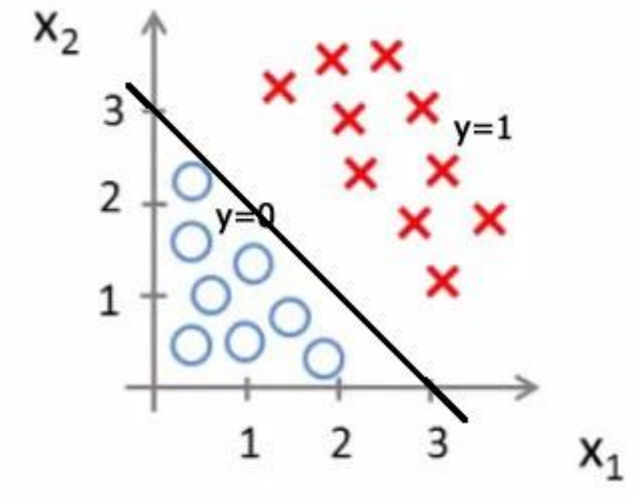


For y = 1, Equation of line would be $x_1 + x_2 >= 3$

For y = 0, Equation of line would be $x_1 + x_2 < 3$

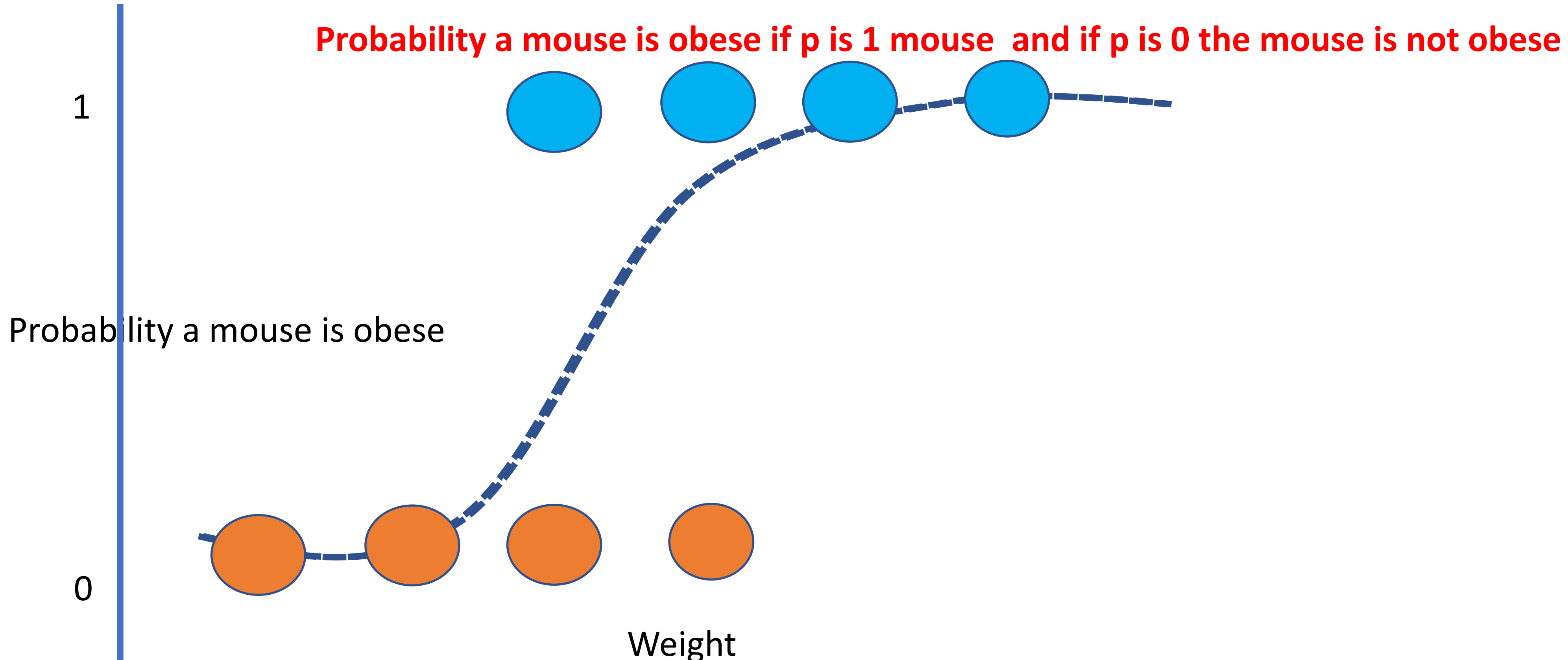
But sometime it will shift its y axis to left or right depending on outliers positions.

What is Decision Boundary?
Decision boundary helps to differentiate probabilities into positive class and negative class.
Linear Decision Boundary

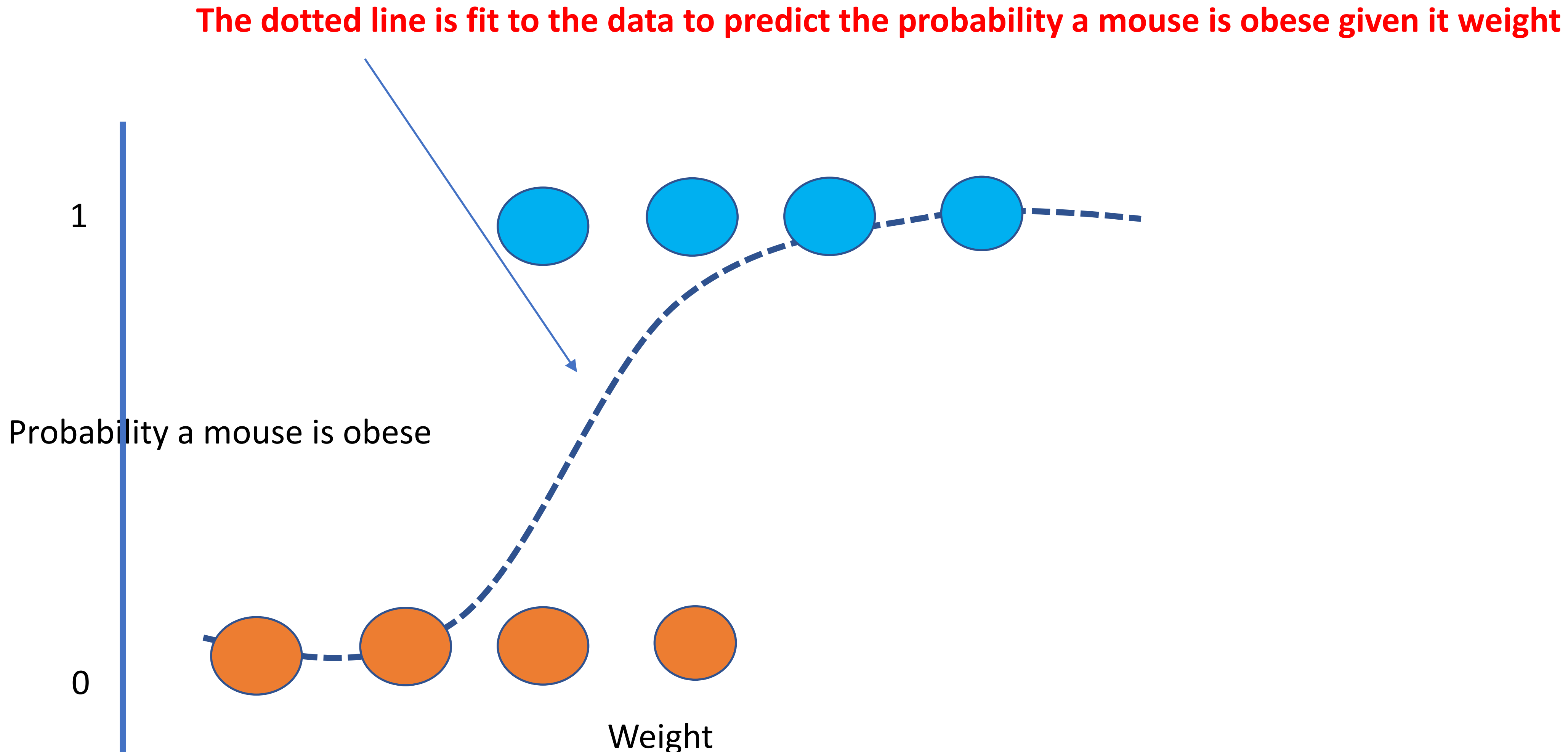


# Logistic Regression

- Logistic regressions ability to provide probabilities and classify new samples using continuous and discrete measurements makes it a popular machine learning method
- One big difference between linear regression and logistic regression is how the line is fit to the data

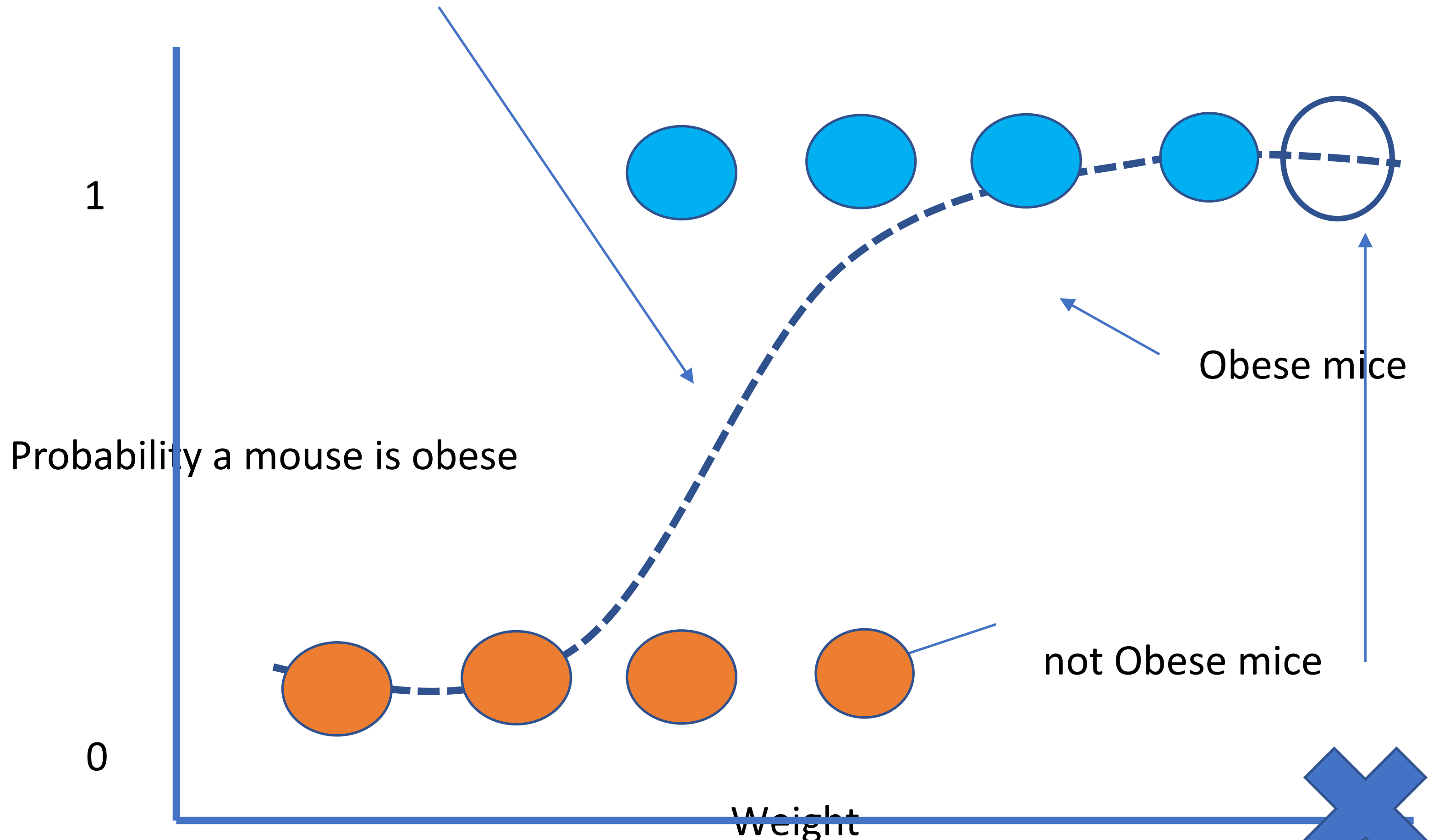


# Logistic Regression

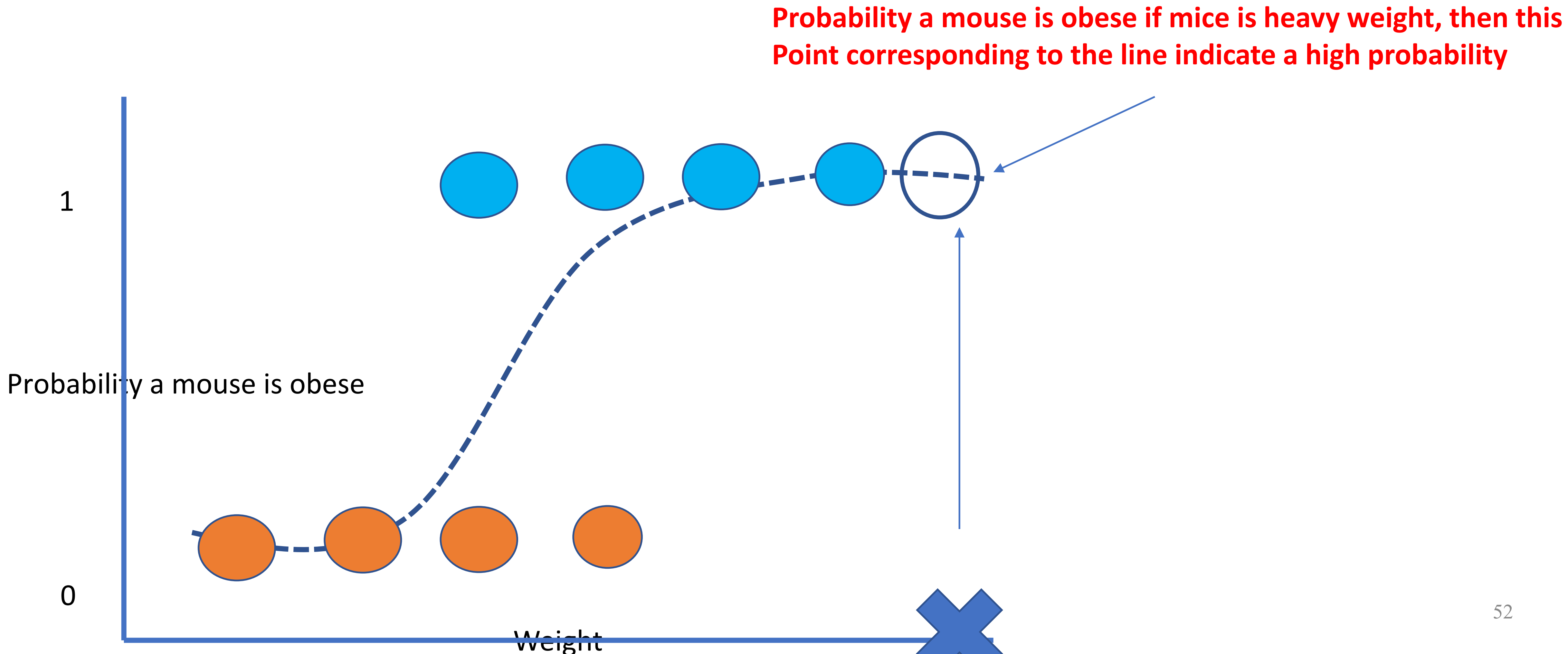


# Logistic Regression

**The dotted line is fit to the data to predict the probability a mouse is obese given it weight**

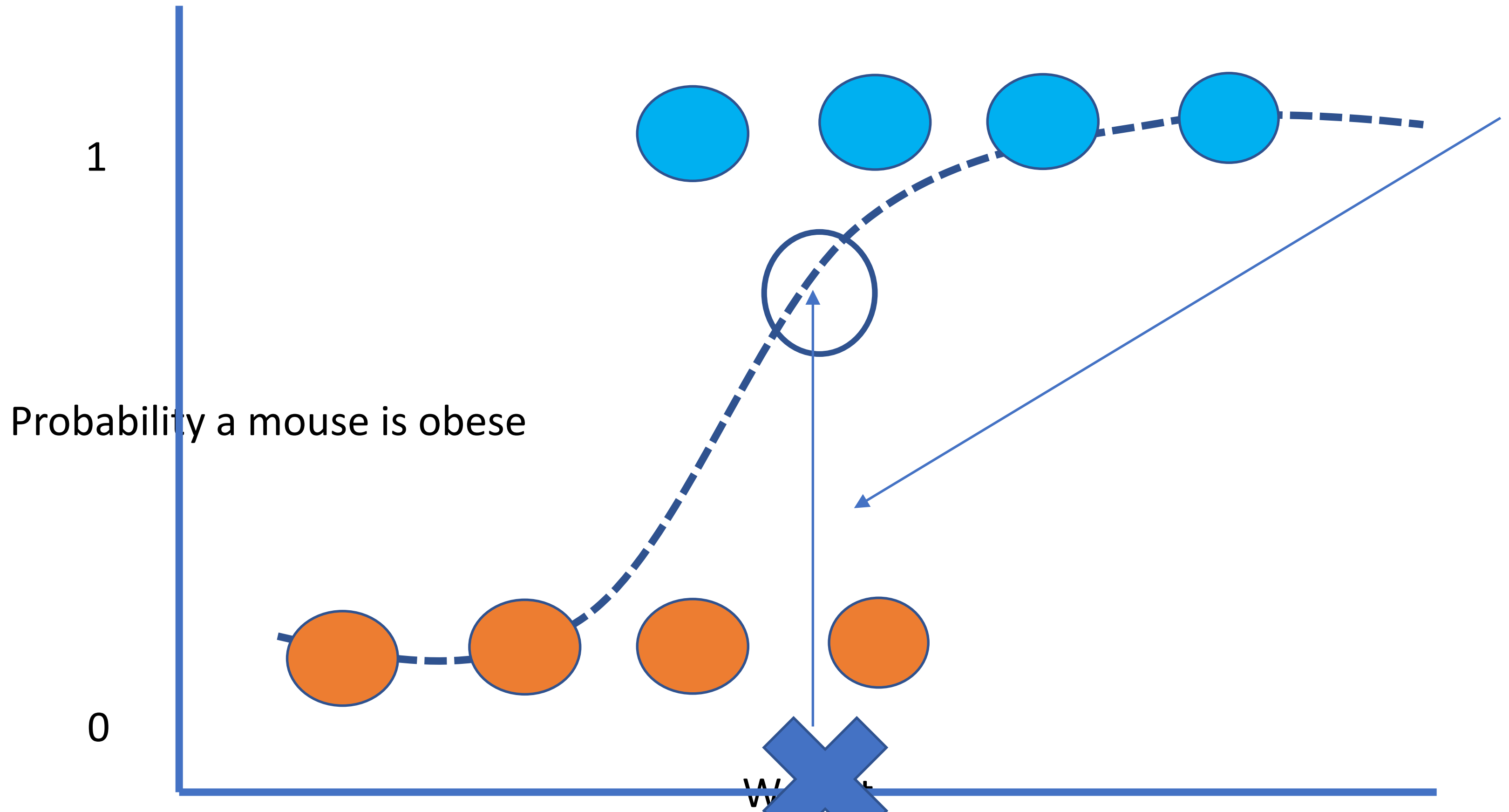


# Logistic Regression



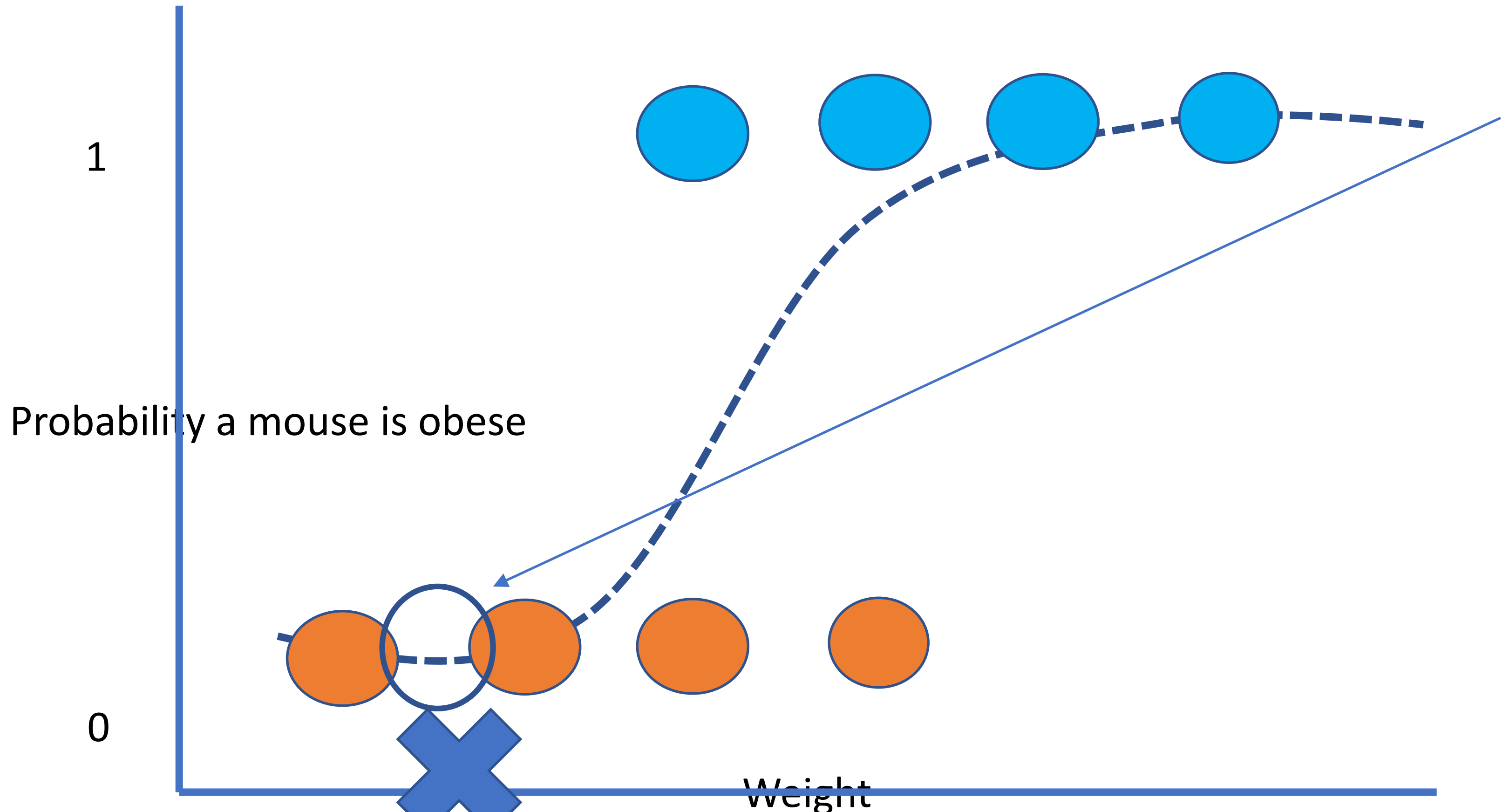
# Logistic Regression

**Probability a mouse is obese if mice is intermediate weight, then thi**
**Point corresponding to the line indicate a intermediate probability**



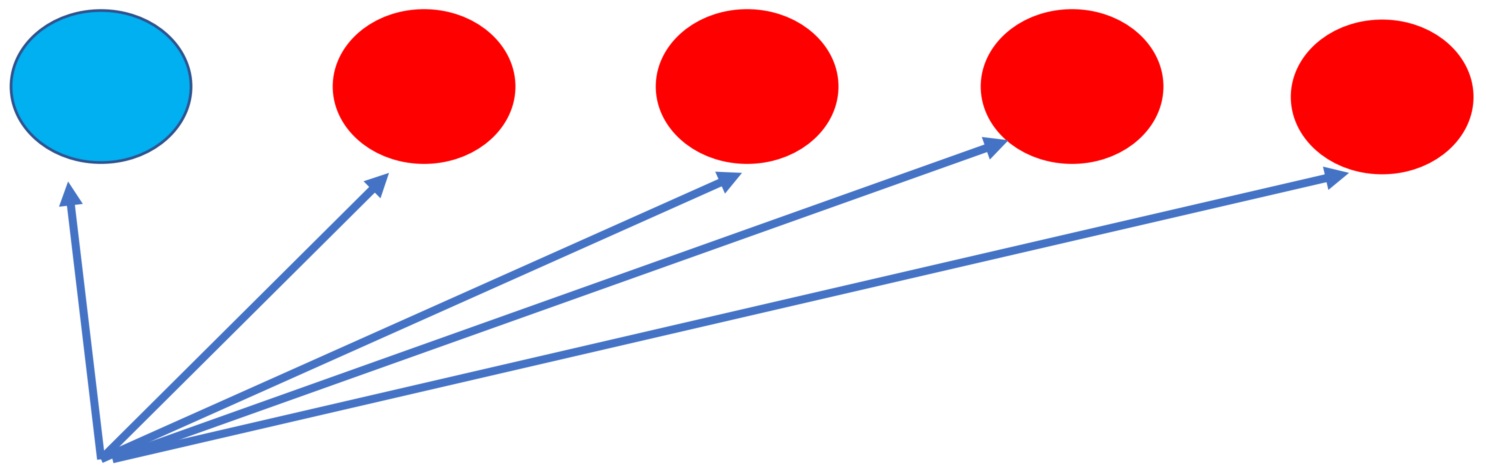
# Logistic Regression

**If mice is low weight, then this**
**Point corresponding to the line indicate a low probability,**
**mice is not obese**



## Logistic Regression

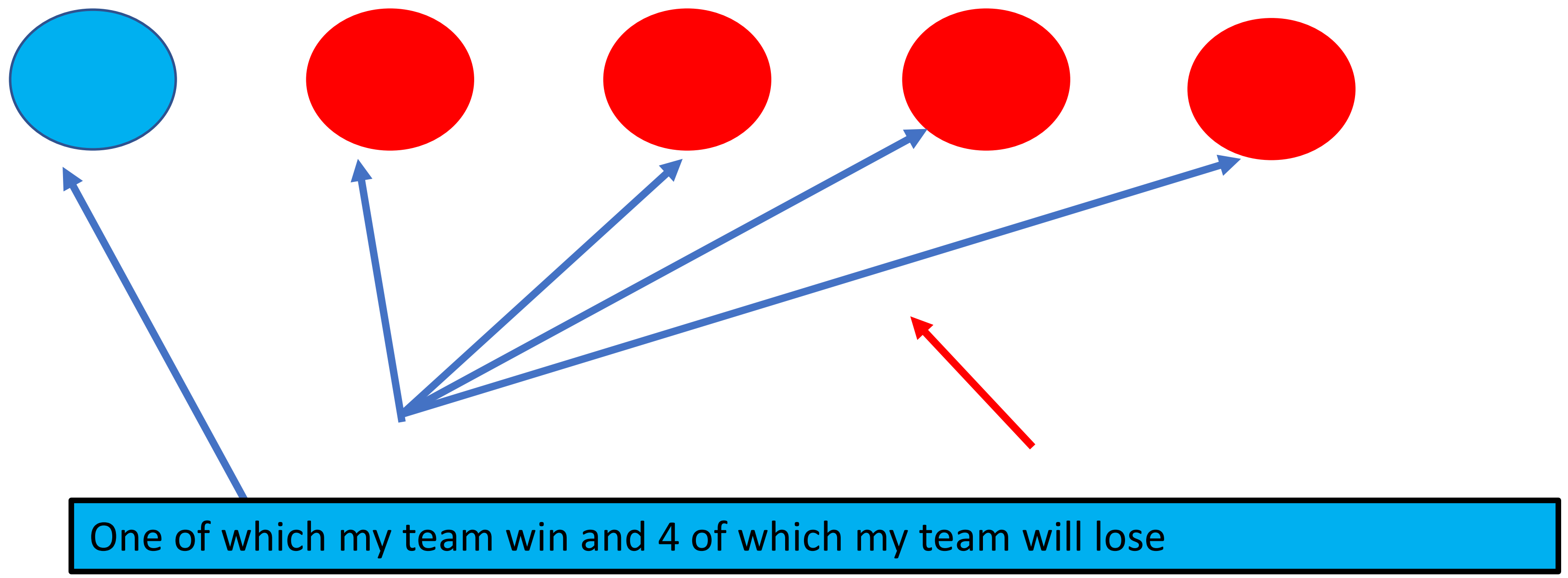
**You might say that the odds in favor of my team winning the game are 1 to 4:**



Visually, we have 5 total games

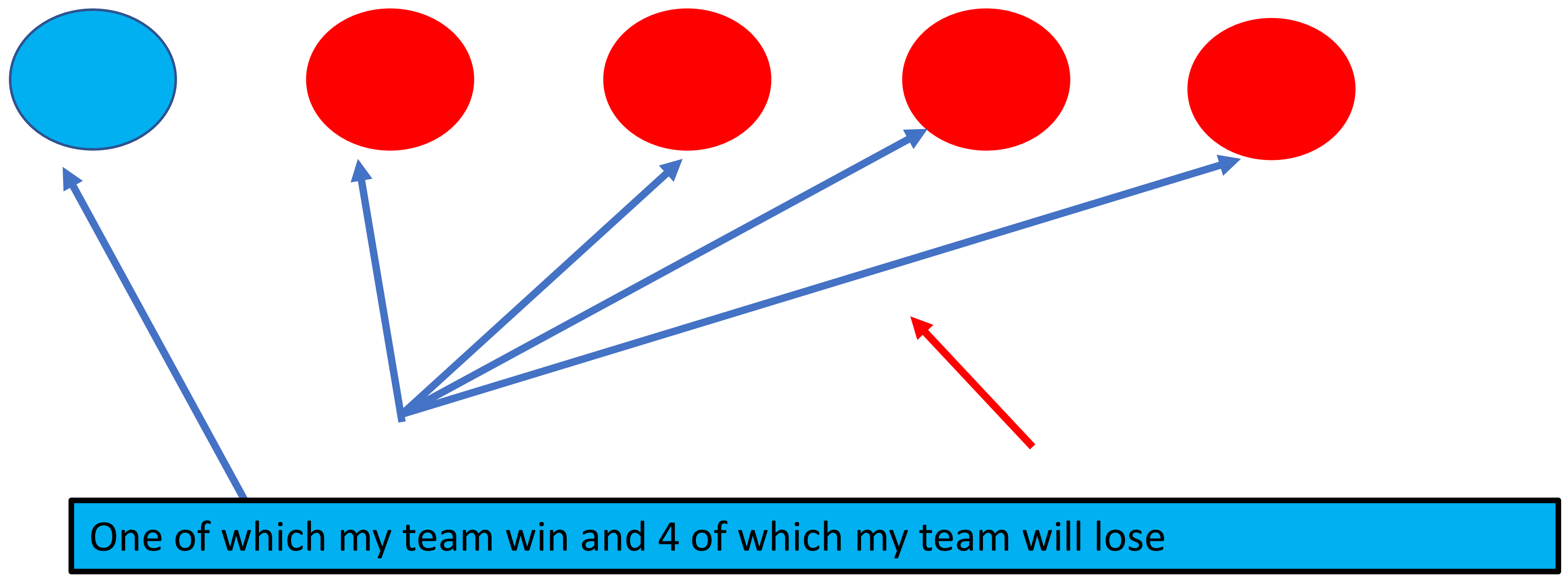
# Logistic Regression

You might say that the odds in favor of my team winning the game are 1 to 4:


One of which my team win and 4 of which my team will lose

# Logistic Regression

**You might say that the odds in favor of my team winning the game are 1 to 4:**



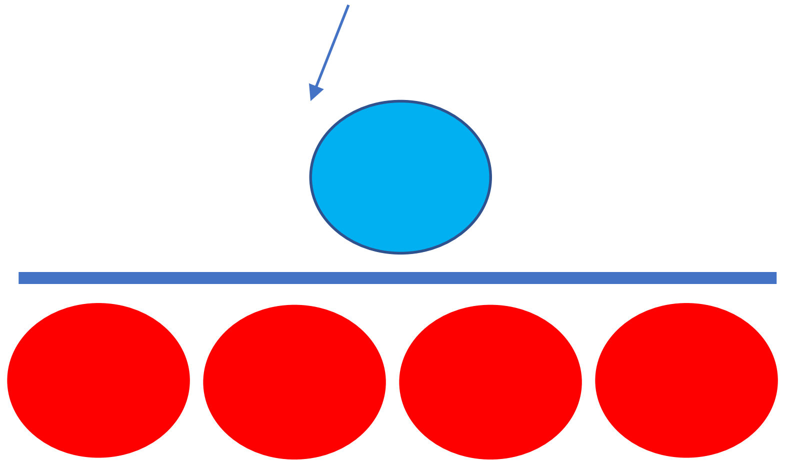
**You might So the odds are 1 and 4 and we can write this as a fraction: 1/4**

# Logistic Regression

**You might say that the odds in favor of my team winning the game are 1 to 4:**

**You might  So the odds are 1 and 4 and we can write this as a fraction: 1/4**

**Visually, we have the 1 game my team wins**



**… divided by the 4 games that my team loses**

## Logistic Regression

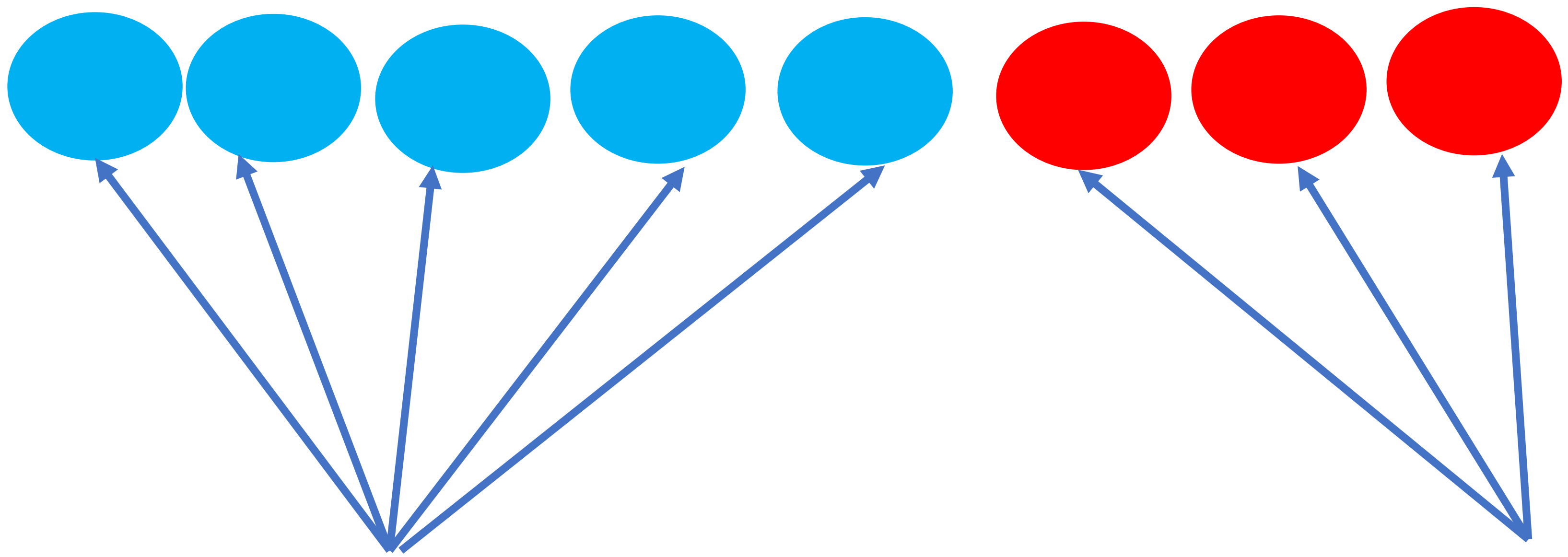
You might say that the odds in favor of my team winning the game are 1 to 4:

We see that the odds are 0.25 that my team will win the game

You might So the odds are 1 and 4 and we can write this as a fraction: ¼=0.25

# Logistic Regression

Another example: You might say that the odds in favor of my team winning the game are 5 to 3:



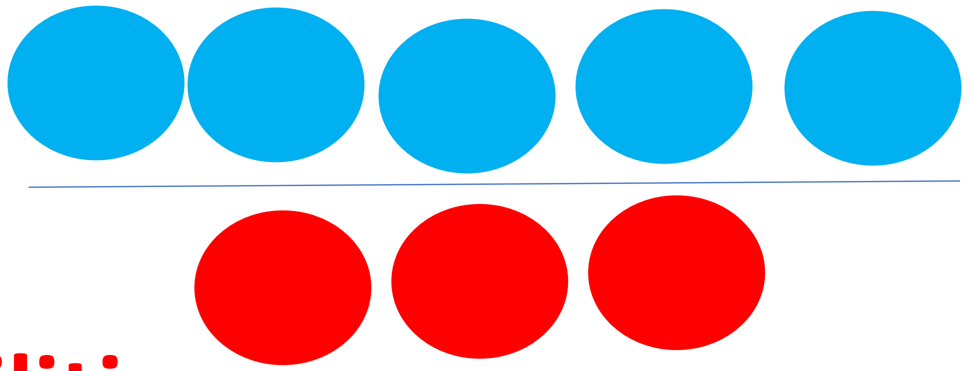
5 of which my team will win

and 3 of which my team will loss

# Logistic Regression

**Another example: You might say that the odds in favor of my team winning the game are 5 to 3:**

**You might  So the odds are 5 and 3 and we can write this as a fraction: 5/3=0.25**

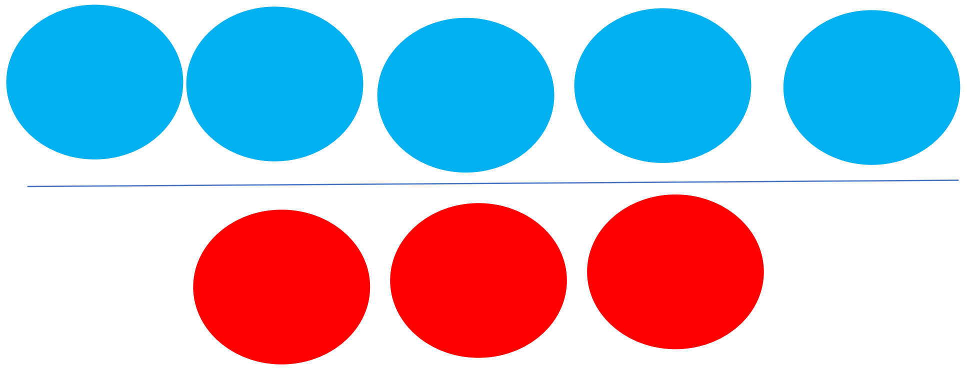


**odds are not probabilities**

# Logistic Regression

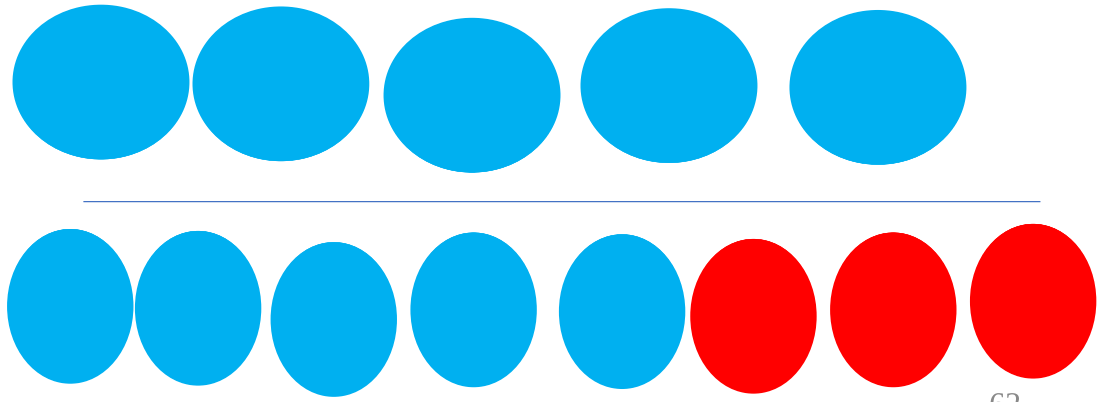
Odds are the ratio of something happing(my team winning)

To something not happing(my team not winning)



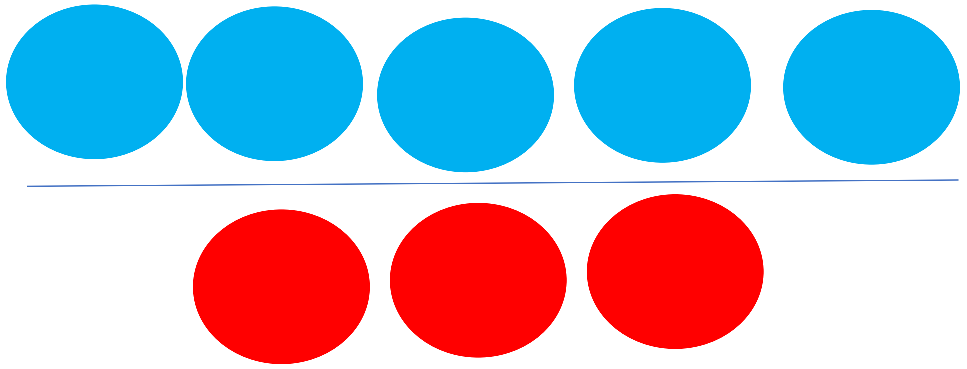
Probabilty is the the ratio of something happing(my team winning)

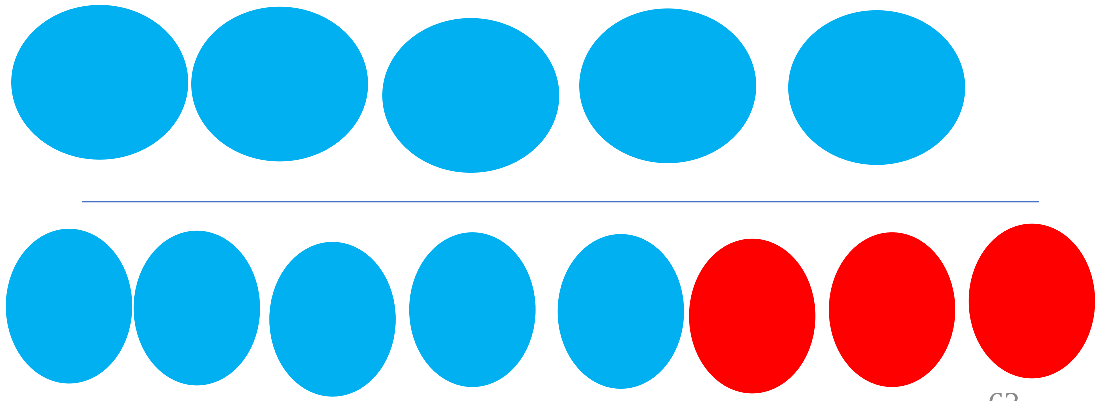
To wvwything that could happen (my team winning and losing)



## Logistic Regression

The odds in favor of my team winning the game are 5 to 3

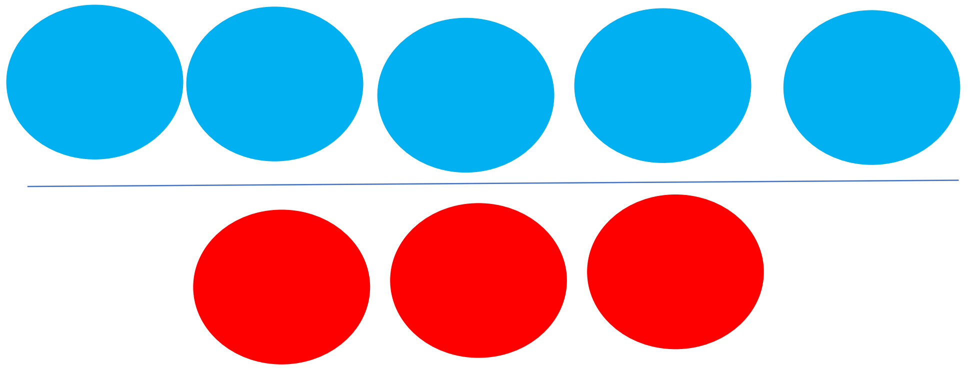




however, the probability of my team wining is the number of games They win(5) divided by the total number of games they play(8).

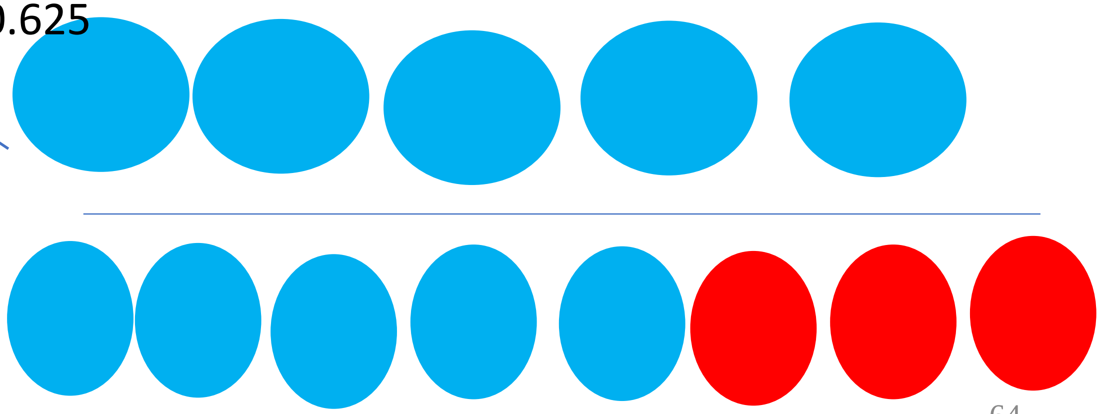
# Logistic Regression

The odds in favor of my team winning the game are 5 to 3



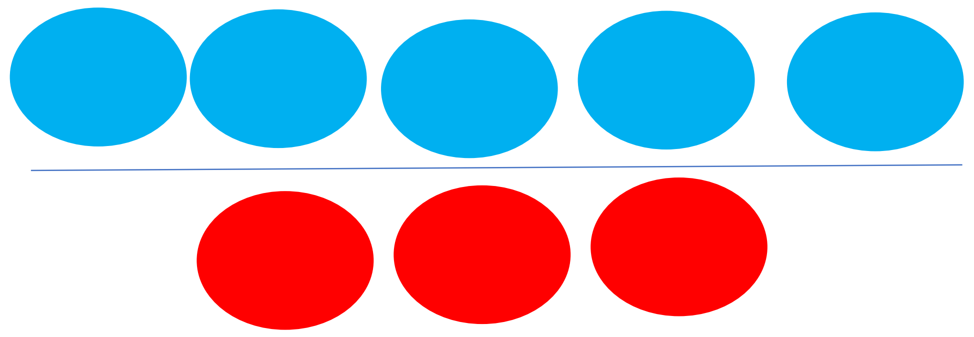
$$\frac{5}{3} = 1.7$$

$$\frac{5}{8} = 0.625$$



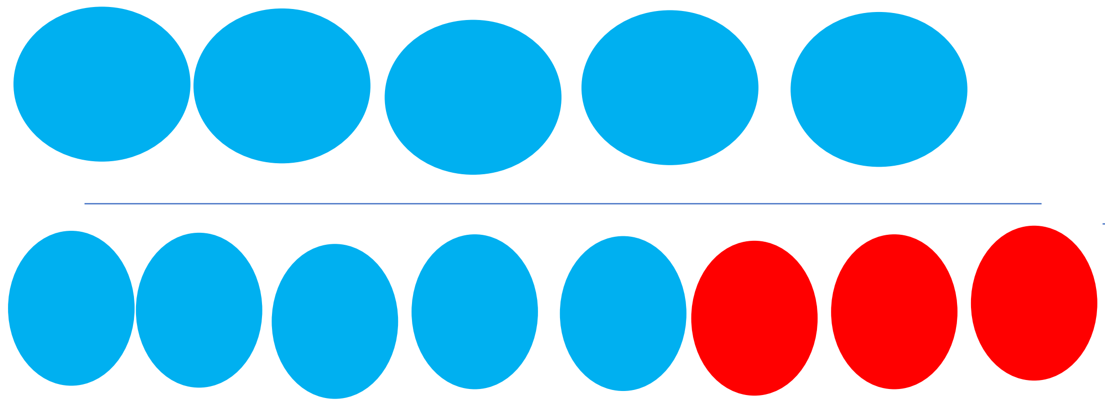
however, the probability of my team wining is the number of games They win(5) divided by the total number of games they play(8).

# Logistic Regression



$$\frac{5}{3} = 1.7$$

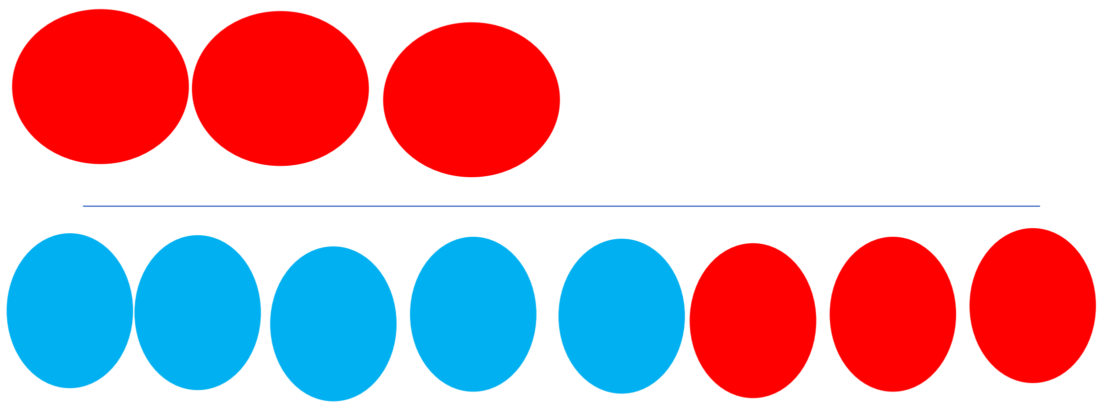
Odds of wining is 1.7



$$\frac{5}{8} = 0.625$$

probabilty of wining is 0.625

# Logistic Regression



$$\frac{3}{8} = 0.375$$

$$\frac{3}{8}$$

probability of losing is 0.375

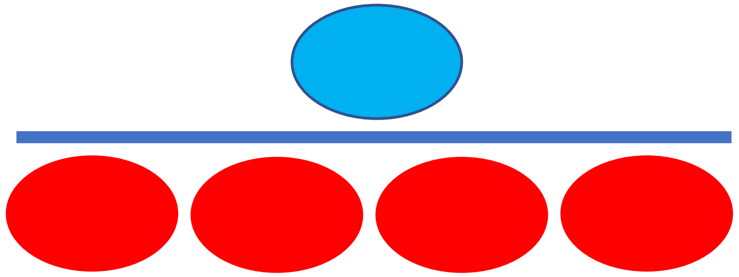
$$\frac{\text{ratio of the probability of winning}}{\text{To the probability of losing}}$$

$$\frac{\text{ratio of the probability of winning}}{\text{To (1-the probability of winning)}} = \frac{5/8}{3/8} = \frac{5}{3}$$

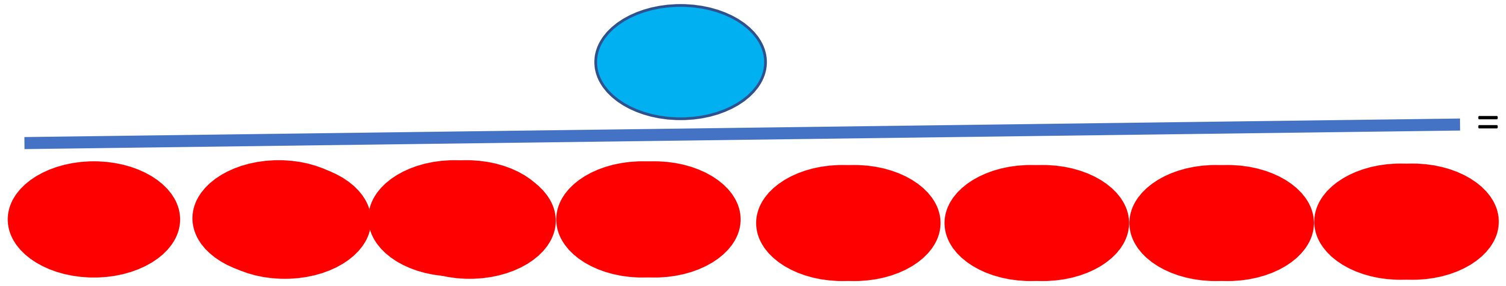
# Logistic Regression

$$\frac{\text{ratio of the probability of winning}}{\text{To (1-the probability of winning)}} = \frac{p}{1-p}$$

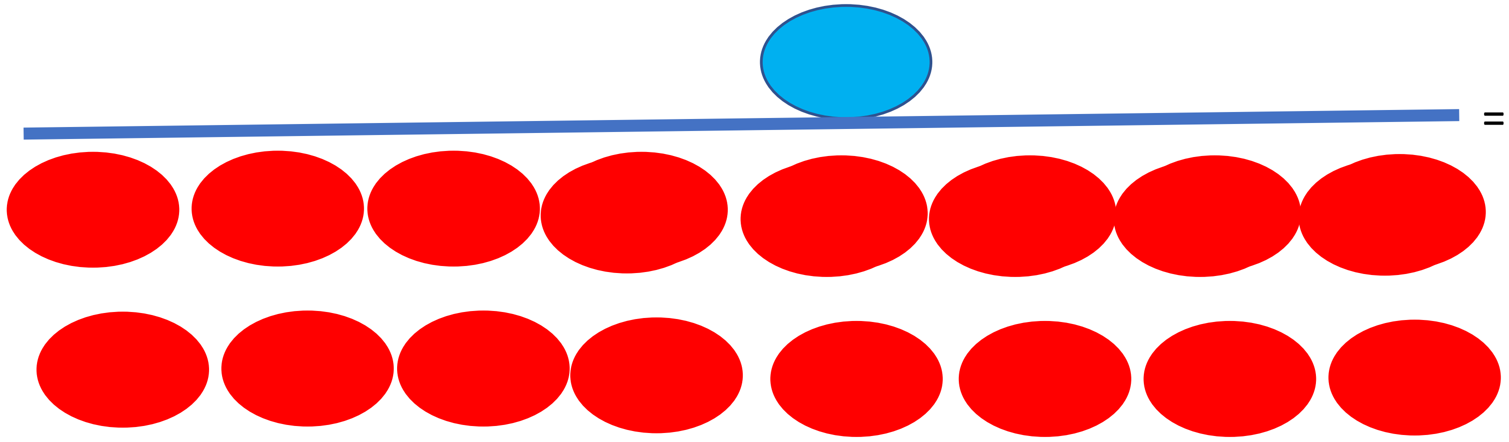
## Logistic Regression



$= \frac{1}{4}$

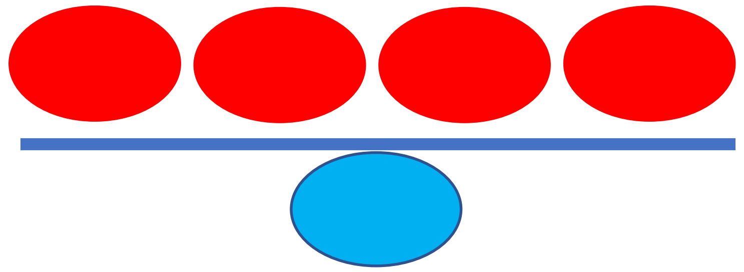


$= \frac{1}{8}$

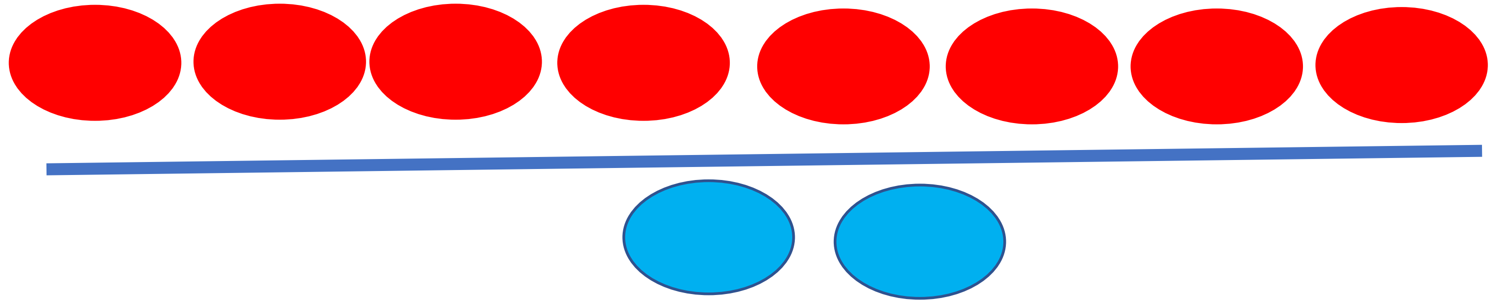


$= \frac{1}{16}$

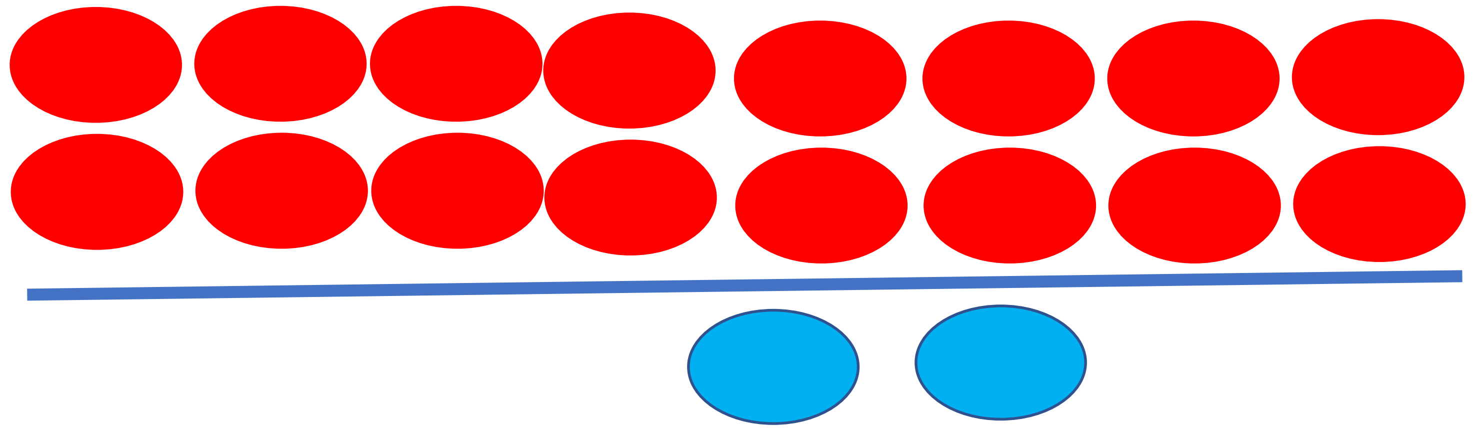
## Logistic Regression



$$= \frac{4}{1}$$



$$= \frac{8}{2}$$



$$= \frac{16}{2}$$

## Logistic Regression

- **Modeling class probabilities via logistic regression**
- Although the perceptron rule offers a nice and easy-going introduction to machine learning algorithms for classification, its biggest disadvantage is that it never converges if the classes are not perfectly linearly separable. The classification task in the previous section would be an example of such a scenario. The reason for this is that the weights are continuously being updated, since there is always at least one misclassified training example present in each epoch. Of course, you can change the learning rate and increase the number of epochs, but be warned that the perceptron will never converge on this dataset.
- To make better use of our time, 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.**

## Optimization of Objective Function for Logistic Regression

## Logistic Regression

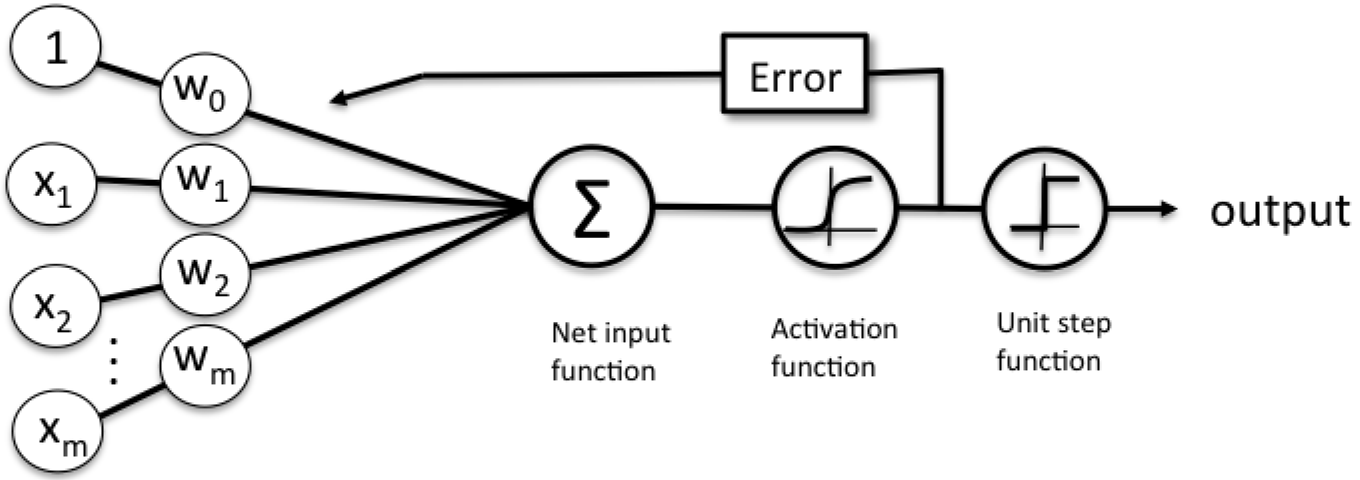
- **Logistic regression and conditional probabilities**
- Logistic regression is a classification model that is very easy to implement and performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry. Similar to the perceptron and Adaline, the logistic regression model in this chapter is also a linear model for binary classification.
- To explain the idea behind logistic regression as a probabilistic model for binary classification, let's first introduce the odds: the odds in favor of a particular event.
- The odds can be written as $p/(1-p)$
- where p stands for the probability of the positive event. 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. We can then further define the logit function, which is simply the logarithm of the odds (log-odds): $logits(p) = log\frac{p}{(1-p)}$

## Logistic Regression

- **Logistic regression and conditional probabilities**
- Note that log refers to the natural logarithm, as it is the common convention in computer science. The logit function takes input values in the range 0 to 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:
- $logits(p) = log\frac{p}{(1-p)}$

# Logistic Regression

- **Logistic Regression**
- **A logistic regression class for binary classification tasks.**



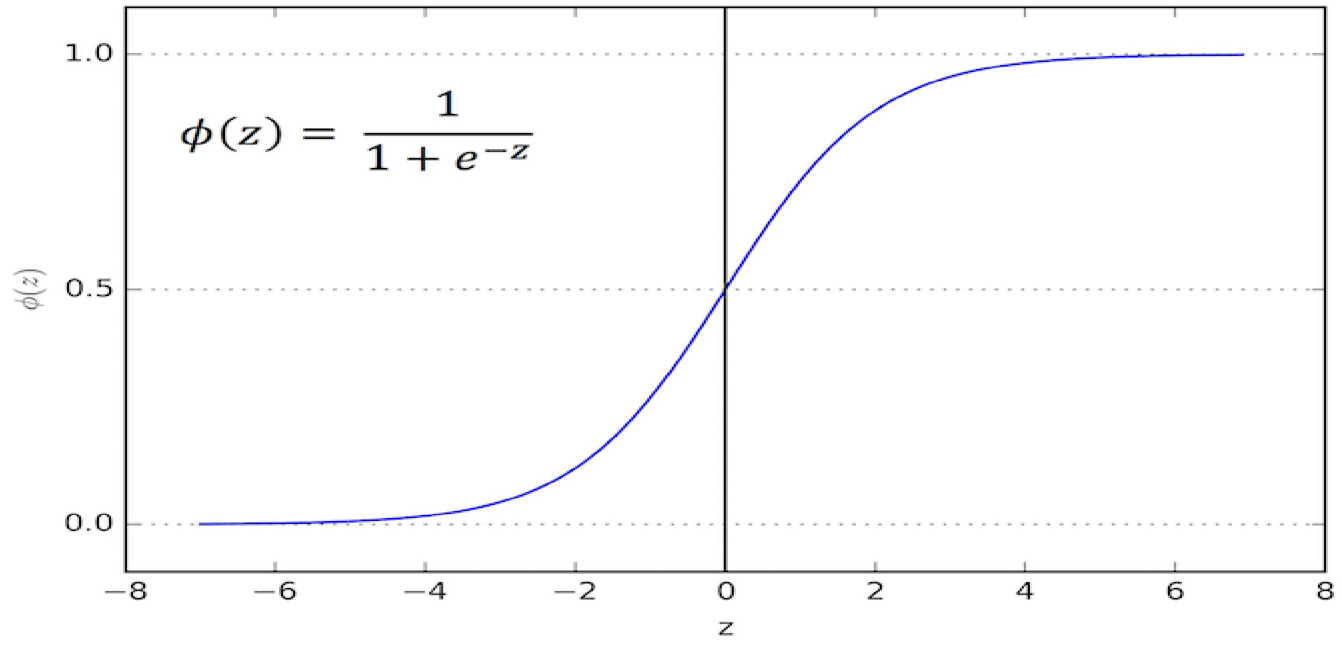
**Schematic of a logistic regression classifier.**

# Logistic Regression

- **Related to the Perceptron and 'Adaline', a Logistic Regression model is a linear model for binary classification. However, instead of minimizing a linear cost function such as the sum of squared errors (SSE) in Adaline, we minimize a sigmoid function, i.e., the logistic function:**
- $\varphi(z)=\frac{1}{1+e^{-z}}$
- where z is defined as the net input
- $z = w_0\ x_0 + w_1\ x_1 + \ldots + w_m\ x_m = \sum_{j=0}^{m} \boldsymbol{w_j\ x_j = w^T x}$
- The net input is in turn based on the logit function
- $logit\left(p\left(y = \frac{1}{x}\right)\right) = z$
- Here, p(y=1|x) is the conditional probability that a particular sample belongs to class 1 given its features x. The logit function takes inputs in the range [0, 1] and transform them to values over the entire real number range. In contrast, the logistic function takes input values over the entire real number range and transforms them to values in the range [0, 1]. In other words, the logistic function is the inverse of the logit function, and it lets us predict the conditional probability that a certain sample belongs to class 1 (or class 0).

# Logistic Regression

- Here, p(y=1|x) is the conditional probability that a particular sample belongs to class 1 given its features x. The logit function takes inputs in the range [0, 1] and transform them to values over the entire real number range. In contrast, the logistic function takes input values over the entire real number range and transforms them to values in the range [0, 1]. In other words, the logistic function is the inverse of the logit function, and it lets us predict the conditional probability that a certain sample belongs to class 1 (or class 0).



## Logistic Regression

- After model fitting, the conditional probability p(y=1|x) is converted to a binary class label via a threshold function g(·):
- $y = g(\{z\}) = \begin{cases} 1 & if \varphi(z) \geq 0.5 \\ 0 & otherwise \end{cases}$
- $Or$
- $y = g(\{z\}) = \begin{cases} 1 & if\ z \geq 0 \\ 0 & otherwise \end{cases}$

- **Objective Function -- Log-Likelihood**
- In order to parameterize a logistic regression model, we maximize the likelihood L(·) (or minimize the logistic cost function).
- We can write the likelihood as
- $L(w)=P(y\backslash \mathrm{x}; w) = \prod_{i=1}^{n} P(y^{i}\backslash x^{i}; w)= \prod_{i=1}^{n}\left(\emptyset(z^{(i)})\right)^{y^{(i)}}\left(1-\emptyset(z^{(i)})\right)^{1-y^{(i)}}$
- under the assumption that the training samples are independent of each other.

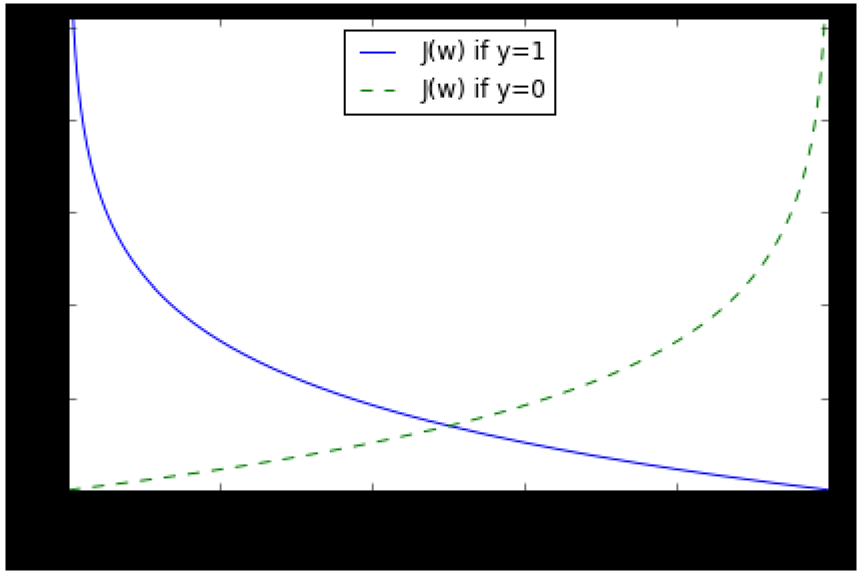
# Logistic Regression

- **Objective Function -- Log-Likelihood**
- In order to parameterize a logistic regression model, we maximize the likelihood L(·) (or minimize the logistic cost function).
- We can write the likelihood as
- $L(w)=P(y\backslash x; w) = \prod_{i=1}^{n} P(y^{i}\backslash x^{i}; w)= \prod_{i=1}^{n}\left(\phi(z^{(i)})\right)^{y^{(i)}}\left(1-\phi(z^{(i)})\right)^{1-y^{(i)}}$
- under the assumption that the training samples are independent of each other.
- In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:
- $l(w)=\mathrm{Log}(L(w))=\sum_{i=1}^{n} y^{(i)} log\ \phi(z^{(i)})+ (1-y^{(i)})log\left(1-\phi(z^{(i)})\right)$
- One advantage of taking the log is to avoid numeric underflow (and challenges with floating point math) for very small likelihoods.
- Another advantage is that we can obtain the derivative more easily, using the addition trick to rewrite the product of factors as a summation term, which we can then maximize using optimization algorithms such as gradient ascent.

# Logistic Regression

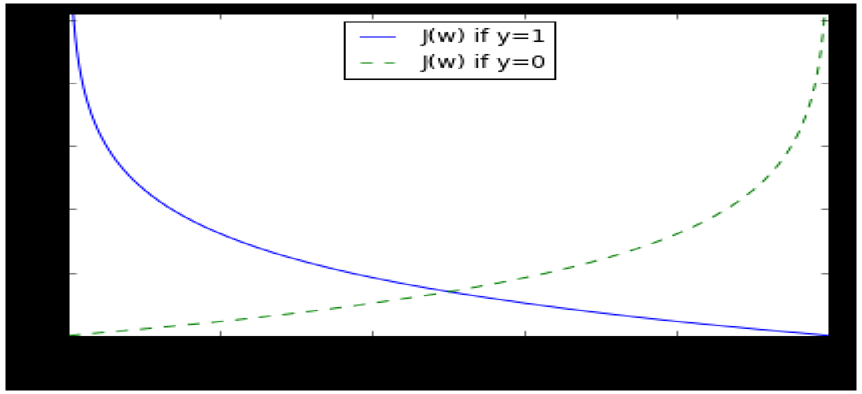
- **Objective Function -- Logistic Cost Function**
- **An alternative to maximizing the log-likelihood, we can define a cost function J(·) to be minimized; we rewrite the log-likelihood as:**
- J(w)=$\sum_{i=1}^{n} -y^{(i)} log\ \emptyset\left(z^{(i)}\right) - (1 - y^{(i)}\ ) log\ \left(1 - \emptyset(z^{(i)})\right)$

- $\begin{cases} -log\emptyset\left(z^{(i)}\right) & if\ y = 1 \\ -log\ \left(1 - \emptyset\left(z^{(i)}\right)\right) & if\ y = 0 \end{cases}$



# Logistic Regression

- J(w)=$\sum_{i=1}^{n} -y^{(i)} log\ \emptyset\left(z^{(i)}\right) - (1 - y^{(i)}\ ) log\ \left(1 - \emptyset(z^{(i)})\right)$
- $\begin{cases} -log\emptyset\left(z^{(i)}\right) & if\ y = 1 \\ -log\ \left(1 - \emptyset\left(z^{(i)}\right)\right) & if\ y = 0 \end{cases}$
- As we can see in the figure below, we penalize wrong predictions with an increasingly larger cost.



http://rasbt.github.io/mlxtend/user_guide/classifier/LogisticRegression/

# Logistic Regression

- Gradient Descent (GD) and Stochastic Gradient Descent (SGD) Optimization

- **Gradient Ascent and the log-likelihood**
- To learn the weight coefficient of a logistic regression model via gradient-based optimization, we compute the partial derivative of the log-likelihood function -- w.r.t. the jth weight -- as follows:
- J(w)=$\sum_{i=1}^{n} -y^{(i)} log\ \emptyset\left(z^{(i)}\right) - (1 - y^{(i)})log\left(1 - \emptyset(z^{(i)})\right)$
- $\frac{\partial}{\partial w_j}$l(w) $= \left(y\frac{1}{\emptyset(z)} - (1 - y)\frac{1}{1-\emptyset(z)}\right)\frac{\partial}{\partial w_j}\emptyset(z)$
- As an intermediate step, we compute the partial derivative of the sigmoid function, which will come in handy later:
- Now, in order to find the weights of the model, we take a step proportional to the positive direction of the gradient to maximize the log-likelihood.
- Furthermore, we add a coefficient, the learning rate η to the weight update:

# Logistic Regression

- Gradient Descent (GD) and Stochastic Gradient Descent (SGD) Optimization
- Now, in order to find the weights of the model, we take a step proportional to the positive direction of the gradient to maximize the log-likelihood. Futhermore, we add a coefficient, the learning rate η to the weight update:
- $w_j := w_j + \eta \frac{\partial}{\partial w_j} \mathrm{l(w)}$
- $w_j := w_j + \eta \sum_{i=1}^{n} (y^{(i)} - \emptyset(z^{(i)}) x_j^{(i)}$
- **Regularization**
- As a way to tackle overfitting, we can add additional bias to the logistic regression model via a regularization terms. Via the L2 regularization term, we reduce the complexity of the model by penalizing large weight coefficients:
- $L_2 : \frac{\lambda}{2} \|w\|_2 = \frac{\lambda}{2} \sum_{j=1}^{m} w_j^2$
- 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:

# Logistic Regression

- **Regularization**
- As a way to tackle overfitting, we can add additional bias to the logistic regression model via a regularization terms. Via the L2 regularization term, we reduce the complexity of the model by penalizing large weight coefficients:
- $L_2 : \frac{\lambda}{2} \|w\|_2 = \frac{\lambda}{2} \sum_{j=1}^{m} w_j^2$
- 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:
- $J(w) = \sum_{i=1}^{m} \left[ -y^{(i)} log \emptyset\left(z^{(i)}\right) - \ (1 - y^{(i)}\ ) log\ \left(1 - \emptyset(z^{(i)})\right) \right] + \frac{\lambda}{2} \sum_{j=1}^{m} w_j^2$
- The update rule for a single weight: $\Delta w_j = -\eta\left(\frac{\partial}{\partial w_j} l(w) + \lambda\, w_j\right)$
- $\Delta w_j = -\eta\left(\frac{\partial}{\partial w_j} l(w) + \lambda\, w_j\right) = -\eta\ \sum_{i=1}^{n} (y^{(i)}\ - \emptyset\left(z^{(i)}\right) x_j^{(i)} - \eta\lambda\, w_j$
- The simultaneous weight update:
- $w := w + \Delta w$

$\Delta w = -\eta\ (\nabla J(w) + \lambda w)$

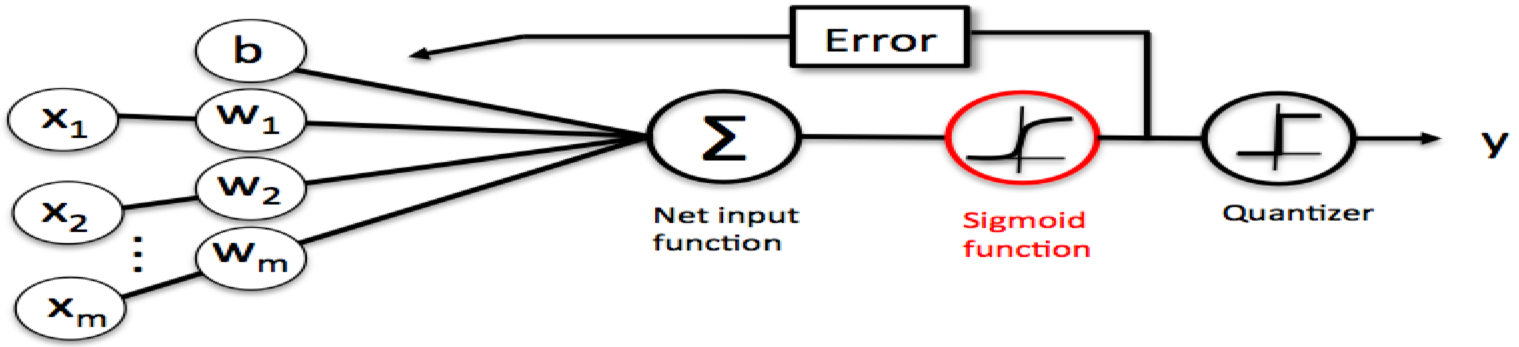
# Logistic Regression

- **Regularization**
- The goal of simple (univariate) linear regression is to model the relationship between a single feature (explanatory variable, x) and a continuous-valued target (response variable, y). The equation of a linear model with one explanatory variable is defined as follows:
- $y = w0 + w1x$
- Here, the weight, $w0$ , represents the y axis intercept and $w1$ is the weight coefficient of the explanatory variable. Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset.
- Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the training examples, as shown in the following figure:

## Softmax Regression

# Softmax Regression

- **Softmax Regression**
- **A logistic regression class for multi-class classification tasks.**
- **Overview**
- Softmax Regression (synonyms: Multinomial Logistic, Maximum Entropy Classifier, or just Multi-class Logistic Regression) is a generalization of logistic regression that we can use for multi-class classification (under the assumption that the classes are mutually exclusive). In contrast, we use the (standard) Logistic Regression model in binary classification tasks.
- Below is a schematic of a Logistic Regression model

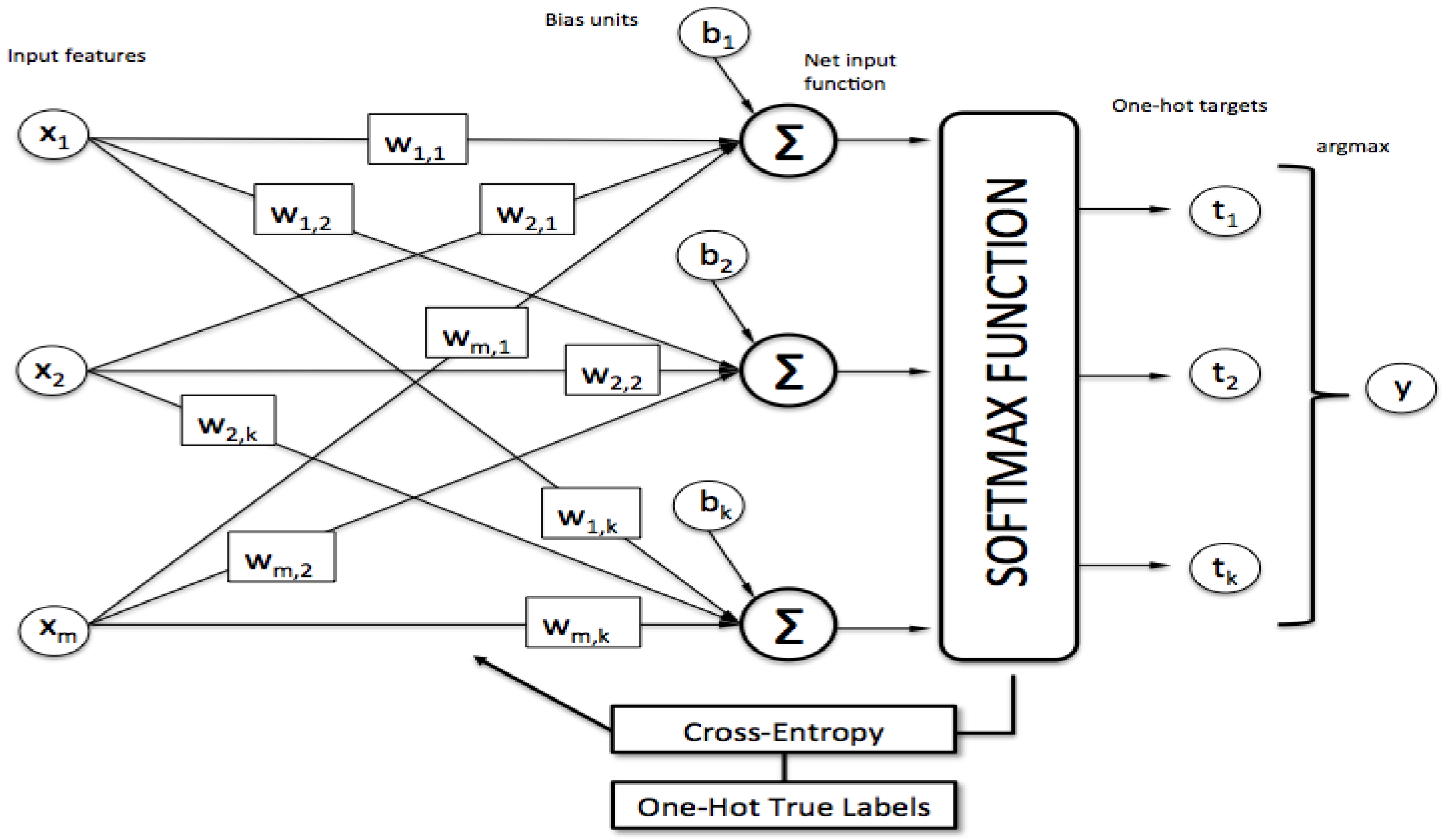


# Softmax Regression

- **Softmax Regression**
- In Softmax Regression (SMR), we replace the sigmoid logistic function by the so-called softmax function ϕsoftmax(·).
- $\boldsymbol{P(y = j \backslash z^{(i)})} = \boldsymbol{\emptyset_{softmax}(z^{(i)})} = \dfrac{\boldsymbol{e^{z^{(i)}}}}{\boldsymbol{\sum_{j=o}^{k} e^{z_k^{(i)}}}}$
- where we define the net input z as
- $\boldsymbol{z = w_1\, x_1 + \ldots + w_m\, x_m + b = \sum_{l=1}^{m} w_l x_l + b = w^T x + b}$
- (w is the weight vector, x is the feature vector of 1 training sample, and b is the bias unit.)
- Now, this softmax function computes the probability that this training sample x(i) belongs to class j given the weight and net input z(i). So, we compute the probability p(y=j|x(i);wj) for each class label in j=1,…,k..
- Note the normalization term in the denominator which causes these class probabilities to sum up to one.

# Softmax Regression

- **Softmax Regression**
- Note the normalization term in the denominator which causes these class probabilities to sum up to one.



## Softmax Regression

- **Softmax Regression**
- To illustrate the concept of softmax, let us walk through a concrete example. Let's assume we have a training set consisting of 4 samples from 3 different classes (0, 1, and 2)
- x0→class 0
- x1→class 1
- x2→class 2
- x3→class 2

```
import numpy as np

y = np.array([0, 1, 2, 2])
```

# Softmax Regression

- **Softmax Regression**
- First, we want to encode the class labels into a format that we can more easily work with; we apply one-hot encoding:

```
import numpy as np

y = np.array([0, 1, 2, 2])
```

```
y_enc = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)

print('one-hot encoding:\n', y_enc)
```

```
one-hot encoding:
 [[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  1.]]
```

A sample that belongs to class 0 (the first row) has a 1 in the first cell, a sample that belongs to class 2 has a 1 in the second cell of its row, and so forth.

# Softmax Regression

- **Softmax Regression**
- Next, let us define the feature matrix of our 4 training samples. Here, we assume that our dataset consists of 2 features;
- thus, we create a 4x2 dimensional matrix of our samples and features.
- Similarly, we create a 2x3 dimensional weight matrix (one row per feature and one column for each class).

```
X = np.array([[0.1, 0.5],
              [1.1, 2.3],
              [-1.1, -2.3],
              [-1.5, -2.5]])

W = np.array([[0.1, 0.2, 0.3],
              [0.1, 0.2, 0.3]])

bias = np.array([0.01, 0.1, 0.1])
```

```
print('Inputs X:\n', X)
print('\nWeights W:\n', W)
print('\nbias:\n', bias)
```

```
Inputs X:
 [[ 0.1  0.5]
 [ 1.1  2.3]
 [-1.1 -2.3]
 [-1.5 -2.5]]

Weights W:
 [[ 0.1  0.2  0.3]
 [ 0.1  0.2  0.3]]

bias:
 [ 0.01  0.1   0.1 ]
```

# Softmax Regression

- **Softmax Regression**
- To compute the net input, we multiply the 4x2 matrix feature matrix X with the 2x3 (n_features x n_classes) weight matrix W, which yields a 4x3 output matrix (n_samples x n_classes) to which we then add the bias unit:
- Z=XW+b.

```
X = np.array([[0.1, 0.5],
              [1.1, 2.3],
              [-1.1, -2.3],
              [-1.5, -2.5]])

W = np.array([[0.1, 0.2, 0.3],
              [0.1, 0.2, 0.3]])

bias = np.array([0.01, 0.1, 0.1])

print('Inputs X:\n', X)
print('\nWeights W:\n', W)
print('\nbias:\n', bias)
```

```
Inputs X:
 [[ 0.1  0.5]
 [ 1.1  2.3]
 [-1.1 -2.3]
 [-1.5 -2.5]]

Weights W:
 [[ 0.1  0.2  0.3]
 [ 0.1  0.2  0.3]]

bias:
 [ 0.01  0.1   0.1 ]
```

```
def net_input(X, W, b):
    return (X.dot(W) + b)

net_in = net_input(X, W, bias)
print('net input:\n', net_in)
```

```
net input:
 [[ 0.07  0.22  0.28]
 [ 0.35  0.78  1.12]
 [-0.33 -0.58 -0.92]
 [-0.39 -0.7  -1.1 ]]
```

# Softmax Regression

- **Softmax Regression**
- Now, it's time to compute the softmax activation that we discussed earlier:
- $P(y = j \backslash z^{(i)}) = \emptyset_{softmax}(z^{(i)}) = \frac{e^{z^{(i)}}}{\sum_{j=o}^{k} e^{z_k^{(i)}}}$

```
def net_input(X, W, b):
    return (X.dot(W) + b)

net_in = net_input(X, W, bias)
print('net input:\n', net_in)
```

```
def softmax(z):
    return (np.exp(z.T) / np.sum(np.exp(z), axis=1)).T

smax = softmax(net_in)
print('softmax:\n', smax)
```

```
softmax:
 [[ 0.29450637  0.34216758  0.36332605]
 [ 0.21290077  0.32728332  0.45981591]
 [ 0.42860913  0.33380113  0.23758974]
 [ 0.44941979  0.32962558  0.22095463]]
```

As we can see, the values for each sample (row) nicely sum up to 1 now. E.g., we can say that the first sample **[ 0.29450637 0.34216758 0.36332605] has a 29.45% probability to belong to class 0.**

**Now, in order to turn these probabilities back into class labels, we could simply take the argmax-index position of each row:**

[[ 0.29450637 0.34216758 **0.36332605**] -> 2
[ 0.21290077 0.32728332 **0.45981591**] -> 2
[ **0.42860913** 0.33380113 0.23758974] -> 0
[ **0.44941979** 0.32962558 0.22095463]] -> 0

# Softmax Regression

- **Softmax Regression**
- Now, it's time to compute the softmax activation that we discussed earlier:
- $P(y = j\backslash z^{(i)}) = \emptyset_{softmax}(z^{(i)}) = \frac{e^{z^{(i)}}}{\sum_{j=o}^{k} e^{z_k^{(i)}}}$

```
def softmax(z):
    return (np.exp(z.T) / np.sum(np.exp(z), axis=1)).T

smax = softmax(net_in)
print('softmax:\n', smax)
```

As we can see, our predictions are terribly wrong, since the correct class labels are [0, 1, 2, 2].

```
def to_classlabel(z):
    return z.argmax(axis=1)

print('predicted class labels: ', to_classlabel(smax))
```

```
predicted class labels:  [2 2 0 0]
```

# Softmax Regression

- **Softmax Regression**
- Now, in order to train our logistic model (e.g., via an optimization algorithm such as gradient descent), we need to define a cost function J(·) that we want to minimize::
- $\boldsymbol{J(W;b) = \frac{1}{n}\sum_{i=1}^{n} H(T_i, O_i)}$
- which is the average of all cross-entropies over our n training samples.
- The cross-entropy function is defined as
- $\boldsymbol{H(T_i, O_i) = -\sum_{i=1}^{n} T_i \cdot log(O_i)}$
- Here the T stands for "target" (i.e., the true class labels) and the O stands for output -- the computed probability via softmax; not the predicted class label.

```
def cross_entropy(output, y_target):
    return - np.sum(np.log(output) * (y_target), axis=1)

xent = cross_entropy(smax, y_enc)
print('Cross Entropy:', xent)
```

```
Cross Entropy: [ 1.22245465  1.11692907  1.43720989
1.50979788]
```

# Softmax Regression

- **Softmax Regression**
- Now, in order to train our logistic model (e.g., via an optimization algorithm such as gradient descent), we need to define a cost function J(·) that we want to minimize::
- $\boldsymbol{J(W; b) = \frac{1}{n}\sum_{i=1}^{n} H(T_i\,, O_i)}$

```
def cost(output, y_target):
    return np.mean(cross_entropy(output, y_target))

J_cost = cost(smax, y_enc)
print('Cost: ', J_cost)
```

$\Downarrow$

```
Cost:  1.32159787159
```

# Softmax Regression

- **Softmax Regression**
- In order to learn our softmax model -- determining the weight coefficients -- via gradient descent, we then need to compute the derivative
- $\boldsymbol{J(W;b)} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{H(T_i\,, O_i)}$
- $\nabla w_j\ \boldsymbol{J(W;b)}$
- I don't want to walk through the tedious details here, but this cost derivative turns out to be simply:
- $\nabla w_j\ \boldsymbol{J(W;b)} = \frac{1}{n}\sum_{i=1}^{n}\left[\boldsymbol{x^{(i)}}(\boldsymbol{O_i} - \boldsymbol{T_i})\right]$
- We can then use the cost derivate to update the weights in opposite direction of the cost gradient with learning rate η:
- $w_j := w_j$ **-** η $\nabla w_j\ \boldsymbol{J(W;b)}$
- for each class

$$j \in \{0,1, \ldots, k\}$$

note that wj is the weight vector for the class y=j), and we update the bias units

$b_j := b_j$ **-** η$\left[\frac{1}{n}\sum_{i=1}^{n}[(\boldsymbol{O_i} - \boldsymbol{T_i})]\right]$

# Softmax Regression

- **Softmax Regression**
- As a penalty against complexity, an approach to reduce the variance of our model and decrease the degree of overfitting by adding additional bias, we can further add a regularization term such as the L2 term with the regularization parameter λ:
- $\boldsymbol{J(W;b) = \frac{1}{n}\sum_{i=1}^{n} H(T_i\,, O_i)}$
- $L_2 : \lambda/2\|w\|_2^2$
- Where
- $\|w\|_2^2 = \sum_{l=0}^{m}\sum_{j=0}^{k} w_{l,j}$
- so that our cost function becomes
- $\boldsymbol{J(W;b) = \frac{1}{n}\sum_{i=1}^{n} H(T_i\,, O_i)} + \lambda/2\|w\|_2^2$

- and we define the "regularized" weight update as

$w_j := w_j$ **-** $\eta\left[\nabla w_j\, \boldsymbol{J(W;b)} + \lambda w_j\right]$

(Please note that we don't regularize the bias term.)

# Softmax Regression

- **Softmax Regression**
- **Example 1 - Gradient Descent**

```
from mlxtend.data
import iris_data
from mlxtend.plotting
import plot_decision_regions
from mlxtend.classifier
import SoftmaxRegression
import matplotlib.pyplot as plt
 # Loading Data
X, y = iris_data()
X = X[:, [0, 3]] # sepal length and petal width
# standardize
X[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
lr = SoftmaxRegression(eta=0.01, epochs=500, minibatches=1, random_seed=1, print_progress=3)
lr.fit(X, y)
plot_decision_regions(X, y, clf=lr)
plt.title('Softmax Regression - Gradient Descent') plt.show() plt.plot(range(len(lr.cost_)), lr.cost_)
plt.xlabel('Iterations') plt.ylabel('Cost') plt.show()
```

*Predicting Class Labels*

```
y_pred = lr.predict(X)
print('Last 3 Class Labels: %s' % y_pred[-3:])
```

```
Last 3 Class Labels: [2 2 2]
```

Predicting Class Probabilities

```
y_pred = lr.predict_proba(X)
print('Last 3 Class Labels:\n %s' % y_pred[-3:])
```

```
Last 3 Class Labels:
 [[ 9.18728149e-09   1.68894679e-02   9.83110523e-01]
 [  2.97052325e-11   7.26356627e-04   9.99273643e-01]
 [  1.57464093e-06   1.57779528e-01   8.42218897e-01]]
```

# Softmax Regression

- **Softmax Regression**
- **Example 2 - Stochastic Gradient Descent**

```
from mlxtend.data
import iris_data
from mlxtend.plotting
import plot_decision_regions
from mlxtend.classifier
import SoftmaxRegression
import matplotlib.pyplot as plt
 # Loading Data
X, y = iris_data()
X = X[:, [0, 3]] # sepal length and petal width
# standardize
X[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
 lr = SoftmaxRegression(eta=0.01, epochs=300, minibatches=len(y), random_seed=1)
lr.fit(X, y)
plot_decision_regions(X, y, clf=lr)
plt.title('Softmax Regression - Gradient Descent') plt.show() plt.plot(range(len(lr.cost_)), lr.cost_)
plt.xlabel('Iterations') plt.ylabel('Cost') plt.show()
```

Q&A