COSC 3337 : Data Science I

N. Rizk

College of Natural and Applied Sciences

Department of Computer Science

University of Houston

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_i = w_i - \eta \frac{d}{dw_i}(loss(w) + regularizer(w, b))$$

$$w_j = w_j + \eta \sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta \lambda w_j$$

Finding the minimum

You're blindfolded, but you can see out of the bottom of the blindfold to the ground right by your feet. I drop you off somewhere and tell you that you're in a convex shaped valley and escape is at the bottom/minimum. How do you get out?

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
    - pick a dimension
    - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_j = w_j - h\frac{d}{dw_j}loss(w)$$

Some maths

$$\frac{d}{dw_j} loss = \frac{d}{dw_j} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b))$$

$$= \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b)) \frac{d}{dw_j} - y_i(w \cdot x_i + b)$$

$$= \sum_{i=1}^{n} - y_i x_{ij} \exp(-y_i(w \cdot x_i + b))$$

Gradient descent

- How to minimize $\min_{\sigma,\alpha} R(\alpha,\sigma)$?

Most approaches use the following method:

1. Set $\sigma = (1, ., 1)$
2. Compute $\alpha^* = \arg\min_\alpha R(\alpha, \sigma)$
3. Compute $\sigma^* = \sigma - \lambda \nabla_\sigma R(\alpha^*, \sigma)$
4. Set $\sigma \leftarrow \sigma^*$ and go back to 2.

Would it make sense to perform just a gradient step here too?

Gradient step in $[0,1]^n$.

Gradient descent summary

Many algorithms can be turned into embedded methods for feature selections by using the following approach:

1. Choose an objective function that measure how well the model returned by the algorithm performs
2. Differentiate this objective function according to the σ parameter
3. Performs a gradient descent on σ. At each iteration, rerun the initial learning algorithm to compute its solution on the new scaled feature space.
4. Stop when no more changes (or early stopping, etc.)
5. Threshold values to get list of features and retrain algorithm on the subset of features.

Difference from add/remove approach is the search strategy. It still uses the inner structure of the learning model but it scales features rather than selecting them.

Gradient descent

Advantage of this approach:

- can be done for non-linear systems (e.g. SVM with Gaussian kernels)
- can mix the search for features with the search for an optimal regularization parameters and/or other kernel parameters.

Drawback:

- heavy computations
- back to gradient based machine algorithms (early stopping, initialization, etc.)

(all training observations utilized in each iteration)

Stochastic Gradient Descent

Solution :one observation per iterations

(SGD) is a simple (select samples not all data) yet very efficient approach to discriminative learning of linear classifiers under convex loss functions such as (linear)

- Support vector machine
- And logistic regression

Advantage /Disadvantage

- **The advantages of Stochastic Gradient Descent are:**
- Efficiency.
- Ease of implementation (lots of opportunities for code tuning).
- **The disadvantages of Stochastic Gradient Descent include:**
- SGD requires a number of hyperparameters such as the regularization parameter and the number of iterations.
- SGD is sensitive to feature scaling.

GD vs SGD

Both algorithms minimize or maximize a **cost-function** by iteratively adjusting the hypothesis function's parameters, by multiplying the gradient ∇J by the learning rate η, and adding it to the parameter vector.

The only (algorithmic) difference is that each algorithm optimizes a different **cost-function**.

1. **Gradient Descent's** *cost-function* iterates over **ALL** training samples:
   a. $J(a,b) = \frac{1}{n}\sum_{i=1}^{n}(y_{i,actual} - y_{i,predicted})^2$
2. **Stochastic Gradient Descent's** *cost-function* only accounts for **ONE** training sample, chosen at random:
   a. $J(a,b) = (y_{i,actual} - y_{i,predicted})^2$

cost functions

- Cost functions referred to by different names: loss function, or error function, or scoring function.
- Consider linear regression, where we choose mean squared error (MSE) as our cost function. Our goal is to find a way to minimize the MSE.
- Our final goal, however, is to use a cost function so we can learn something from our data.

Using gradient ascent for linear classifiers

Key idea :

1. Define a linear classifier (logistic regression)
2. Define an objective function (likelihood)
3. Optimize it with gradient descent to learn parameters
4. Predict the class with highest probability under the model

Create a learning process

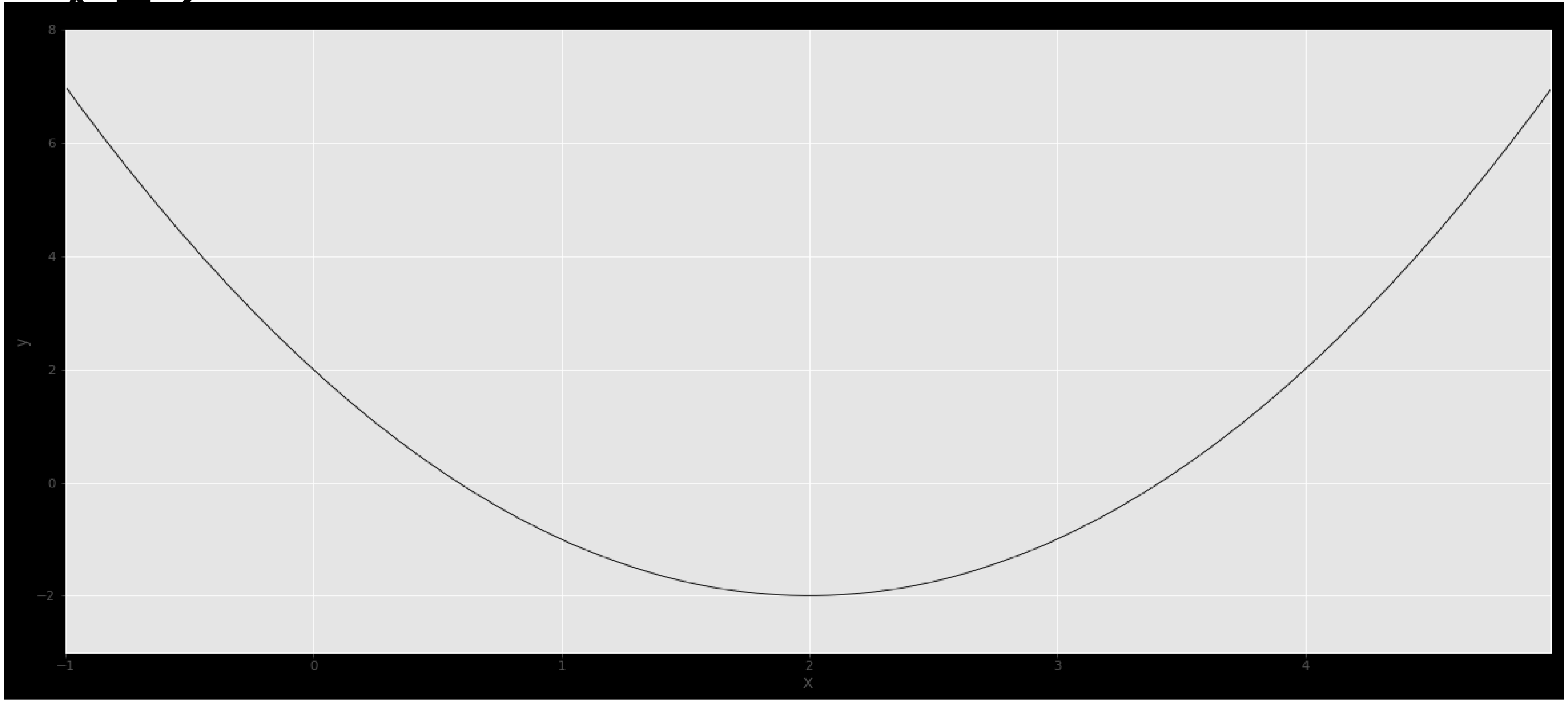
we’re going to be implementing gradient descent to create a learning process with feedback.

Each time—each step really—we receive some new information, we’re going to make some updates to our estimated parameter which move towards an optimal combination of parameters.

We get these estimates using our cost

# Finding the minimum of the function
y = x^2 - 4x + 2

dy/dx = 0 = 2*x - 4 # This is our cost function
x = 2



How would we find the solution using gradient descent?

- Let's break this down mathematically, as we're going to be estimating a parameter θ which we will substitute for x. θ is the value we're going to update after every step and will tell us what the current value of x is through minimization process. As θ converges to the minimum using our cost function.
- However, we don't always know were to start θ on our cost function so we take a guess. It starts at this guessed point somewhere along the cost function, and descends towards the actual value.
- That is the descent, in gradient descent.
- We are also going to introduce a variable called α which is out learning rate.
- The learning rate tells our cost function how fast to move toward its goal

$$y = x^2 - 4x + 2$$

$$\frac{dy}{dx} = 2x - 4$$

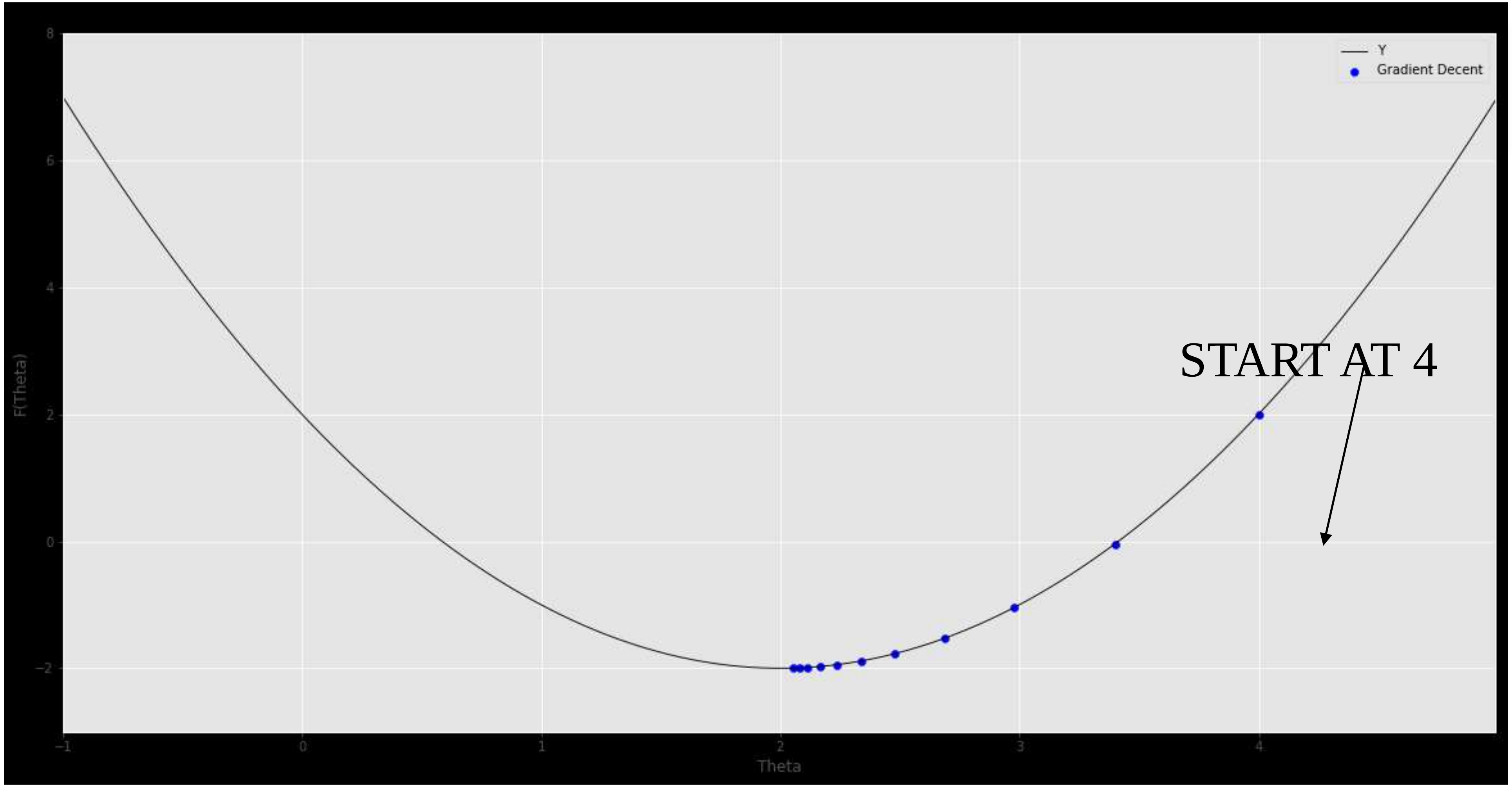
$$\alpha = learning\ rate$$

$$\theta = parameter\ to\ estimate$$

$$\frac{dy}{d\theta} = 2\theta - 4$$

$$\theta_i := \theta_i - \alpha \frac{dy}{d\theta_i}$$

```
def func_y(x):
    y = x**2 - 4*x + 2
    return y
def gradient_descent(previous_x, learning_rate, epoch):

    # To fill with values
    x_gd = []
    y_gd = []

    x_gd.append(previous_x)
    y_gd.append(func_y(previous_x))
    # begin the loops to update x and y with out cost function
    for i in range(epoch):
        current_x = previous_x - learning_rate * (2*previous_x - 4)
        x_gd.append(current_x)
        y_gd.append(func_y(current_x))
        # update previous_x
        previous_x = current_x
return x_gd, y_gd
# Initialize x0 and learning rate
x0 = 4 # Our first 'guess' at what theta could be
learning_rate = 0.15 # Alpha
epoch = 10 # Number of tries
```

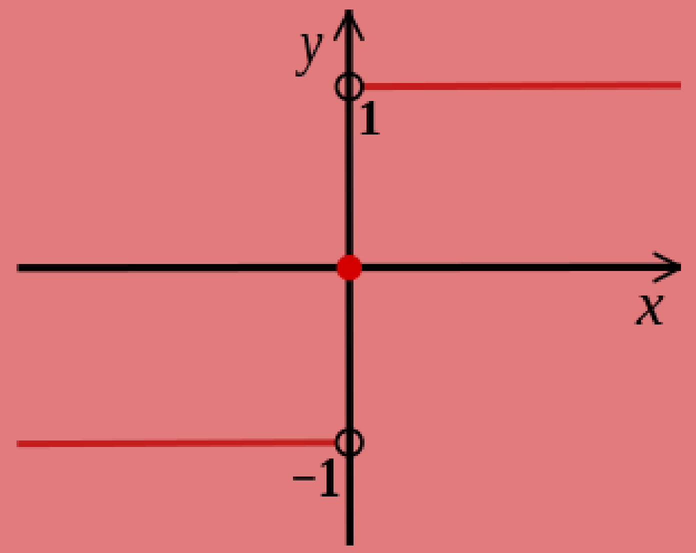


Y
Gradient Decent
F(Theta)
Theta
START AT 4

Using gradient ascent for linear classifiers

This decision function isn't differentiable:

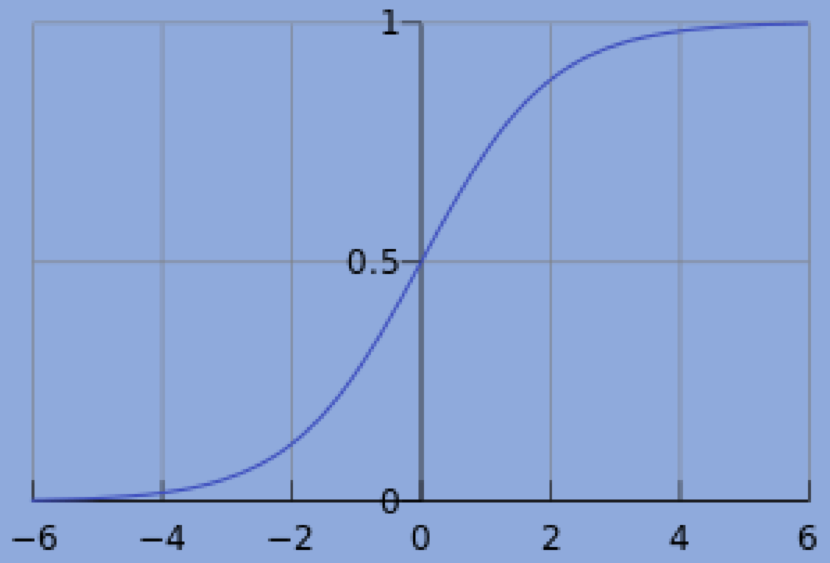
$$h(\mathbf{x}) = \mathrm{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



sign(x)

Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$

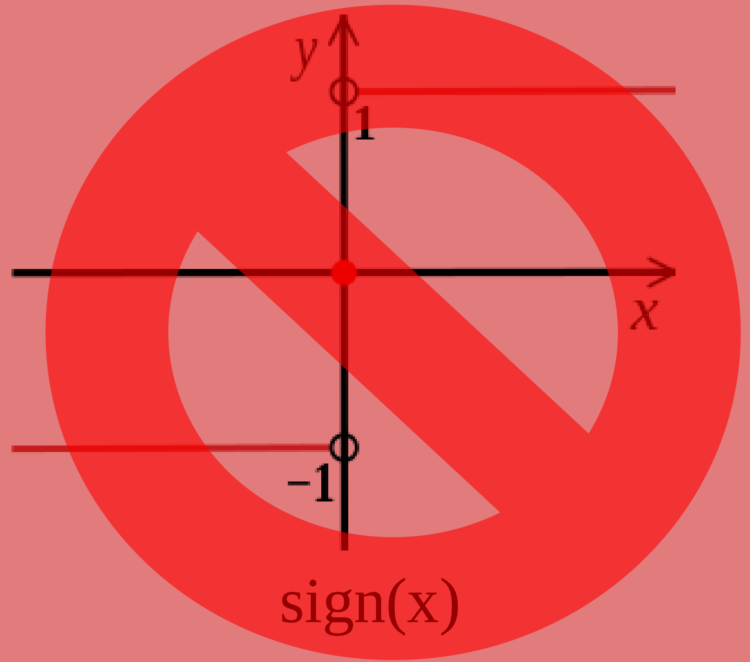


$$\mathrm{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

Using gradient ascent for linear classifiers

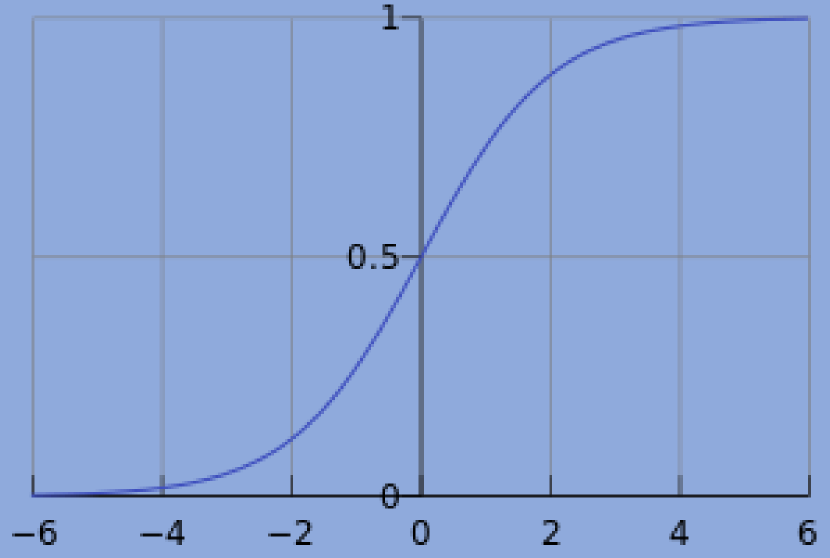
This decision function isn't differentiable:

$$h(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$



$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

Logistic Regression

Data: Inputs are continuous vectors of length K. Outputs are discrete.

$$\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N} \text{ where } \mathbf{x} \in \mathbb{R}^K \text{ and } y \in \{0, 1\}$$

Model: Logistic function applied to dot product of parameters with input vector.

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$

Learning: finds the parameters that minimize some objective function.

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$

Prediction: Output is the most probable class.

$$\hat{y} = \underset{y \in \{0,1\}}{\operatorname{argmax}}\, p_{\boldsymbol{\theta}}(y|\mathbf{x})$$

Speeding logistic regression using SGD

The class **SGDClassifier** implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties for classification.

As other classifiers, SGD has to be fitted with two arrays: an array X of size [n_samples, n_features] holding the training samples, and an array Y of size [n_samples] holding the target values (class labels) for the training samples:

```
from sklearn.linear_model import
SGDClassifier
X = [[0., 0.], [1., 1.]]
y = [0, 1]
clf = SGDClassifier(loss="hinge",
penalty="l2", max_iter=5)
clf.fit(X, y)
```

```
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0,
fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge',
max_iter=5,
       n_iter=None, n_iter_no_change=5, n_jobs=None,
penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False)
```

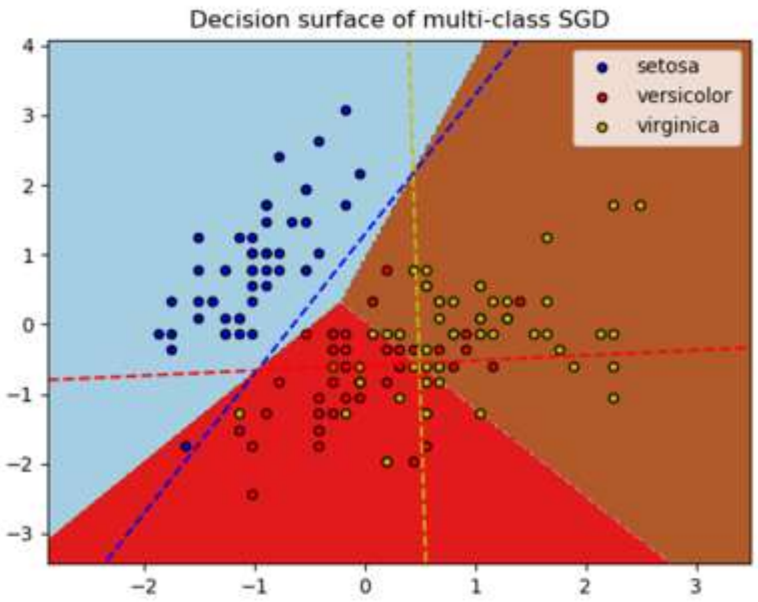
After being fitted, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])

array([1])

#SGD fits a linear model to the training data. The
member coef_ holds the model parameters:

>>> clf.coef_

array([[9.9..., 9.9...]])
#Member intercept_ holds the intercept

>>> clf.intercept_

array([-9.9...])
```

decision

To get the signed distance to the hyperplane use SGDClassifier.decision_function:

```
>>> clf.decision_function([[2., 2.]])
array([29.6...])
```



The concrete loss function can be set via the loss parameter.
SGDClassifier supports the following loss functions:

- loss="hinge": (soft-margin) linear Support Vector Machine,
- loss="modified_huber": smoothed hinge loss,
- loss="log": logistic regression,

Using loss="log" or loss="modified_huber" enables the predict_proba method, which gives a vector of probability estimates per sample :

```
>>> clf = SGDClassifier(loss="log", max_iter=5).fit(X, y)
>>> clf.predict_proba([[1., 1.]])
array([[0.00..., 0.99...]])
```

The concrete penalty can be set via the penalty parameter. SGD supports the following penalties:

- penalty="l2": L2 norm penalty on coef_.
- penalty="l1": L1 norm penalty on coef_.
- penalty="elasticnet": Convex combination of L2 and L1; (1 - l1_ratio) * L2 + l1_ratio * L1.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.linear_model import SGDClassifier
# import some data to play with
iris = datasets.load_iris()
# we only take the first two features. We could
# avoid this ugly slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target
colors = "bry"

# shuffle
idx = np.arange(X.shape[0])
np.random.seed(13)
np.random.shuffle(idx)
X = X[idx]
y = y[idx]
# standardize
mean = X.mean(axis=0)
std = X.std(axis=0)
X = (X - mean) / std
h = .02  # step size in the mesh
clf = SGDClassifier(alpha=0.001, max_iter=100).fit(X, y)
```

```python
# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:,
0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:,
1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min,
x_max, h),
                     np.arange(y_min,
y_max, h))
```

```
# Plot the decision boundary. For that,
we will assign a color to each
# point in the mesh [x_min,
x_max]x[y_min, y_max].
Z = clf.predict(np.c_[xx.ravel(),
yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z,
cmap=plt.cm.Paired)
plt.axis('tight')
```

```
# Plot also the training points
for i, color in zip(clf.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color,
label=iris.target_names[i],
                cmap=plt.cm.Paired, edgecolor='black', s=20)
plt.title("Decision surface of multi-class SGD")
plt.axis('tight')

# Plot the three one-against-all classifiers
xmin, xmax = plt.xlim()
ymin, ymax = plt.ylim()
coef = clf.coef_
intercept = clf.intercept_

def plot_hyperplane(c, color):
    def line(x0):
   return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]

    plt.plot([xmin, xmax], [line(xmin), line(xmax)],
             ls="--", color=color)

for i, color in zip(clf.classes_, colors):
    plot_hyperplane(i, color)
plt.legend()
plt.show()
```