

MTH 221

Fundamentals of Machine Learning

Batuhan Bardak

Lecture 5: Linear Regression & Gradient Descent & Logistic Regression

Date: 31.10.2023

MLE

MAP

Naive Bayes*

Recap on Naive Bayes

Setting: binary classification w/ dataset $\{x_i, y_i\}_{i=1}^n, (x_i, y_i) \sim P$, where $x \in \mathbb{R}^d, y \in \{-1, 1\}$

Goal: estimate $P(y|x)$

We take a generative modeling approach here:

$$P(y|x) \propto P(x|y)P(y)$$

Estimate $P(x|y)$ & $P(y)$ from data
(hence generative modeling)

(Discriminative modeling: directly estimate $P(y|x)$)

logistic reg.

Recap on Naive Bayes

Estimate $P(y)$ from data:

$$P(y|x) \propto P(x|y)P(y)$$

Estimate $\underline{P(y)}$ is easy:

$$P(y=1) \approx \frac{\sum_{i=1}^n \mathbf{1}(y_i = 1)}{n}$$

10 times 7 heads

$$P(y=\text{heads}) \approx \frac{\sum^n (y_i = \text{heads})}{10}$$

Recap on Naive Bayes

$$P(y|x) \propto P(x|y)P(y)$$

Estimate $P(x|y)$ from data:

$$x_{\text{test}} = \underbrace{\text{such-}x}_{\sim \sim}$$

Estimate $P(x|y)$ is not easy:

x can be high-dimensional, e.g., d is large!

There may not be repetitions in $\{x_i\}_{i=1}^n$!

A hand-drawn grid diagram consisting of a vertical column of 5 horizontal lines and a horizontal row of 5 vertical lines, forming a 4x4 grid of 16 small squares. To the right of the grid, there are two labels: "Yes" above a square and "No" below it, with a dashed vertical line extending downwards from the grid.

					Yes	
					No	

Recap on Naive Bayes

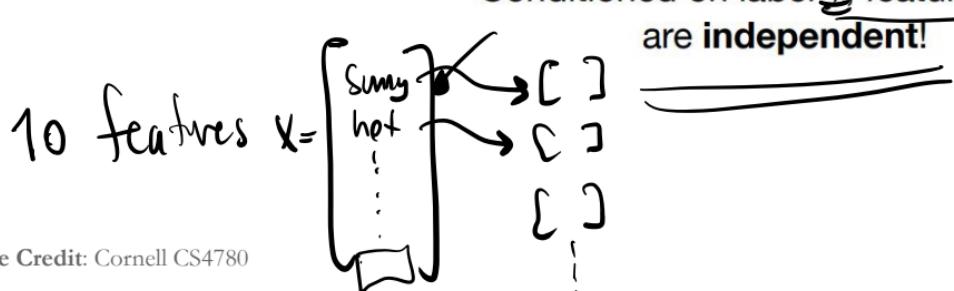
The key assumption in Naive Bayes

The Naive Bayes assumption:

$$P(x|y) = \prod_{\alpha=1}^d P(x[\alpha]|y)$$

\log
[ε — —]

Conditioned on label y, feature values
are independent!



Recap on Naive Bayes

assumption
=

Once estimated $P(y)$ and $P(x|y)$, we can make prediction:

$$P(y|x) \propto P(x|y)P(y)$$

naive

In test time, given x :

$$\hat{y} = \arg \max_y P(y|x)$$

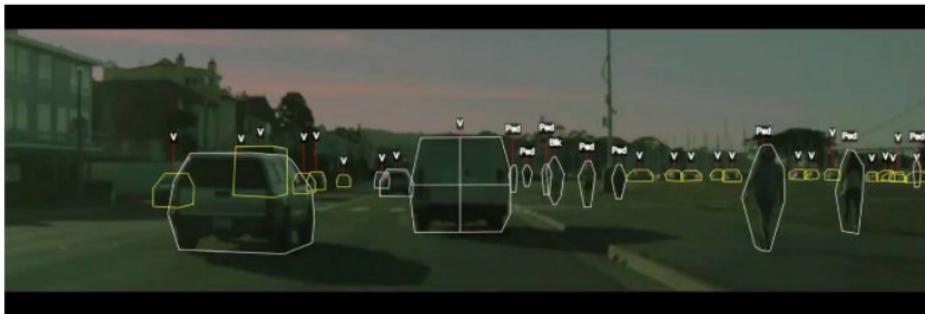
$$= \arg \max_y P(x|y)P(y) = \arg \max_y \left(\prod_{\alpha=1}^d P(x[\alpha]|y) \right) P(y)$$

Plan for today

- Linear Regression → Regression
- Gradient Descent // → Optimization
- Logistic Regression → Classification

Recall: Supervised Learning / unsupervised learning.

The most common approach to machine learning is supervised learning.



1. First, we collect a dataset of labeled training examples.
2. We train a model to output accurate predictions on this dataset.
3. When the model sees new, similar data, it will also be accurate.

do not need
"Y"
"forget"
"label"

• Reinforcement

A Supervised Learning Dataset: Notation

We say that a training dataset of size n (e.g., n patients) is a set

$$\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, \dots, n\}$$

Each $x^{(i)}$ denotes an input (e.g., the measurements for patient i), and each $y^{(i)} \in \mathcal{Y}$ is a target (e.g., the diabetes risk).

Together, $(x^{(i)}, y^{(i)})$ form a *training example*.

A Supervised Learning Dataset: Notation

We can look at the diabetes dataset in this form.

scikit-learn

```
In [11]: # Load the diabetes dataset  
diabetes_X, diabetes_y = diabetes.data, diabetes.target  
  
# Print part of the dataset  
diabetes_X.head()
```

10 features.

Out[11]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641

S

Training Dataset: Inputs

More precisely, an input $x^{(i)} \in \mathcal{X}$ is a d -dimensional vector of the form

$$\text{patient. i} \quad x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_d^{(i)} \end{bmatrix} \hookrightarrow 10$$

For example, it could be the measurements the values of the d features for patient i .

The set \mathcal{X} is called the feature space. Often, we have, $\mathcal{X} = \underline{\mathbb{R}^d}$

Training Dataset: Inputs

Let's look at data for one patient.

```
In [12]: diabetes_X.iloc[0]
```

```
Out[12]: age    0.038076  
          sex   0.050680  
          bmi   0.061696  
          bp    0.021872  
          s1   -0.044223  
          s2   -0.034821  
          s3   -0.043401  
          s4   -0.002592  
          s5    0.019908  
          s6   -0.017646  
          Name: 0, dtype: float64
```

pandas
→ Series

Training Dataset: Attributes

We refer to the numerical variables describing the patient as *attributes*. Examples of attributes include:

- The age of a patient. ✓
- The patient's gender. ✓✓
- The patient's BMI. ✓

Note that these attributes in the above example have been mean-centered at zero and rescaled to have a variance of one.

~~Attributes~~ Features: Discrete vs. Continuous

Age	height	Age * weight
10	80	$\frac{10 \times 80}{500}$

Features can be either discrete or continuous. We will see later that they may be handled differently by ML algorithms.

Features: Discrete vs. Continuous

The BMI feature that we have seen earlier is an example of a continuous feature.

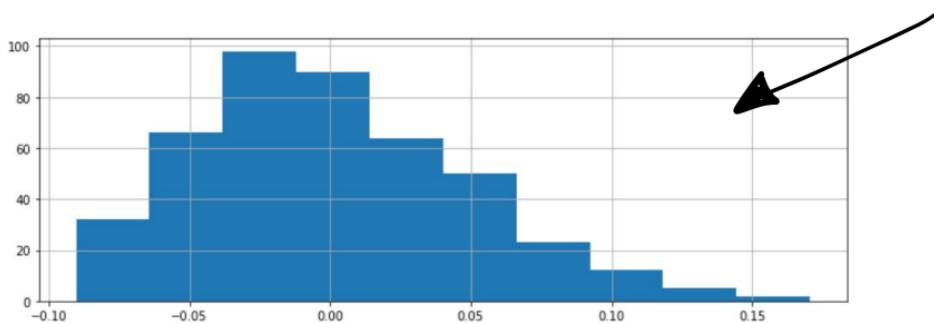
We can visualize its distribution.

Features can
be handled
differently

In [14]: `diabetes_X.loc[:, 'bmi'].hist()`

Out[14]: `<AxesSubplot:>`

may be handled



Features: Discrete vs. Continuous

Other features take on one of a finite number of discrete values. The `sex` column is an example of a categorical feature.

In this example, the dataset has been pre-processed such that the two values happen to be `0.05068012` and `-0.04464164`.

Features can
differently

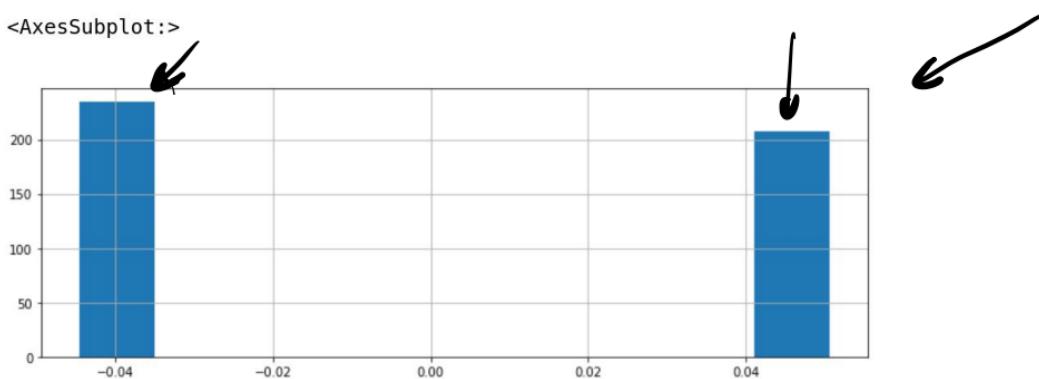
In

Out

```
In [15]: print(diabetes_X.loc[:, 'sex'].unique())
diabetes_X.loc[:, 'sex'].hist()
```

[0.05068012 -0.04464164]

Out[15]: <AxesSubplot:>

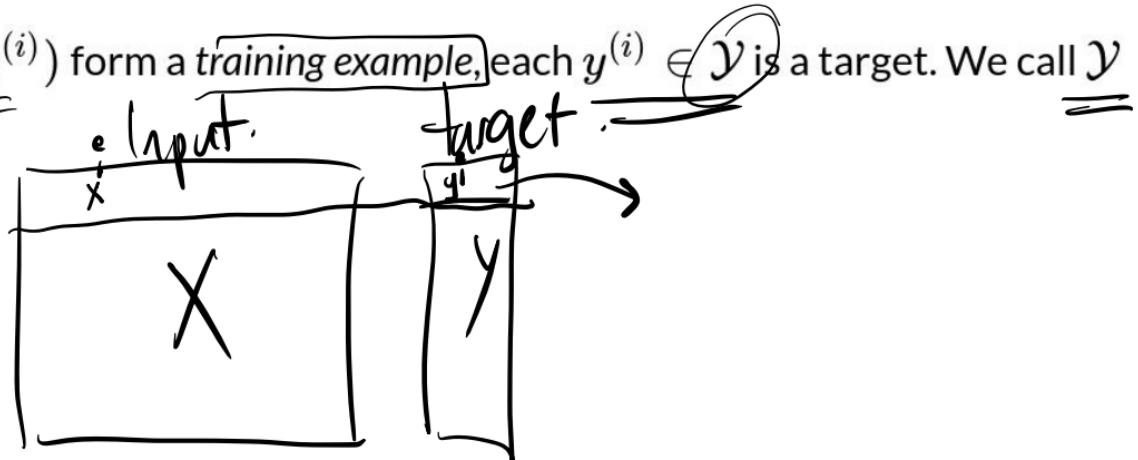


be handled

Training Dataset: Targets

For each patient, we are interested in predicting a quantity of interest, the target. In our example, this is the patient's diabetes risk.

Formally, when $(x^{(i)}, y^{(i)})$ form a training example, each $y^{(i)} \in \mathcal{Y}$ is a target. We call \mathcal{Y} the target space.



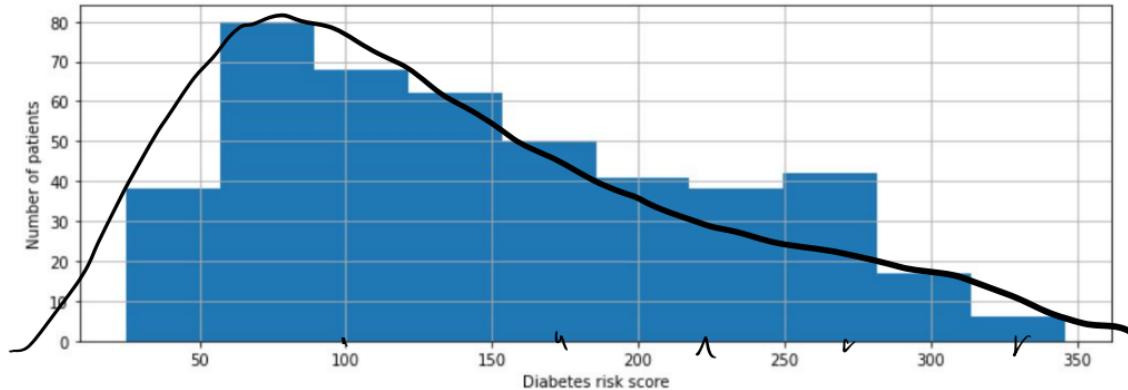
Training Dataset: Targets

```
In [16]: plt.xlabel('Diabetes risk score')
plt.ylabel('Number of patients')
diabetes_y.hist()
```

For example

```
Out[16]: <AxesSubplot:xlabel='Diabetes risk score', ylabel='Number of patients'>
```

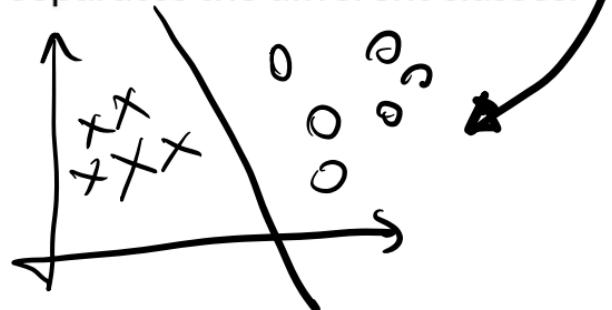
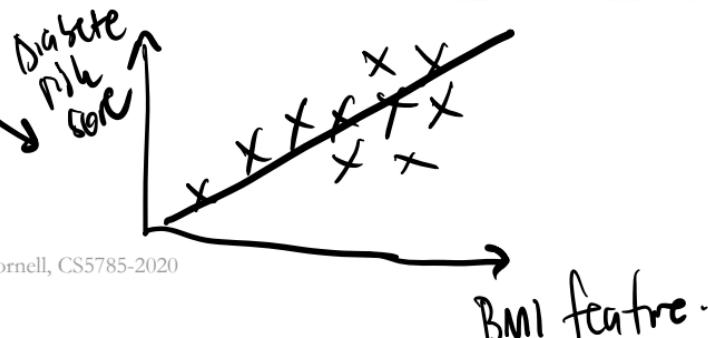
For the target



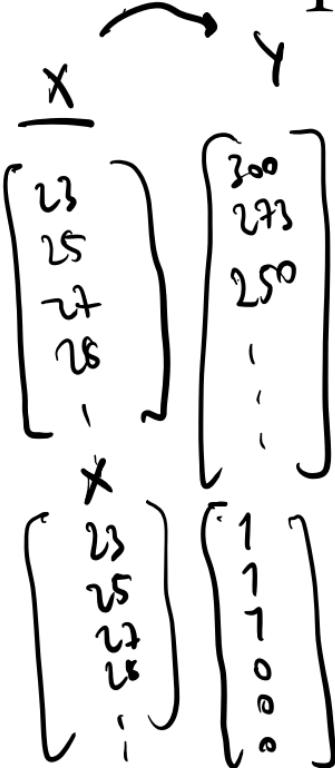
Targets: Regression vs. Classification

We distinguish between two broad types of supervised learning problems that differ in the form of the target variable.

1. **Regression:** The target variable y is continuous. We are fitting a curve in a high-dimensional feature space that approximates the shape of the dataset.
2. **Classification:** The target variable y is discrete. Each discrete value corresponds to a class and we are looking for a hyperplane that separates the different classes.



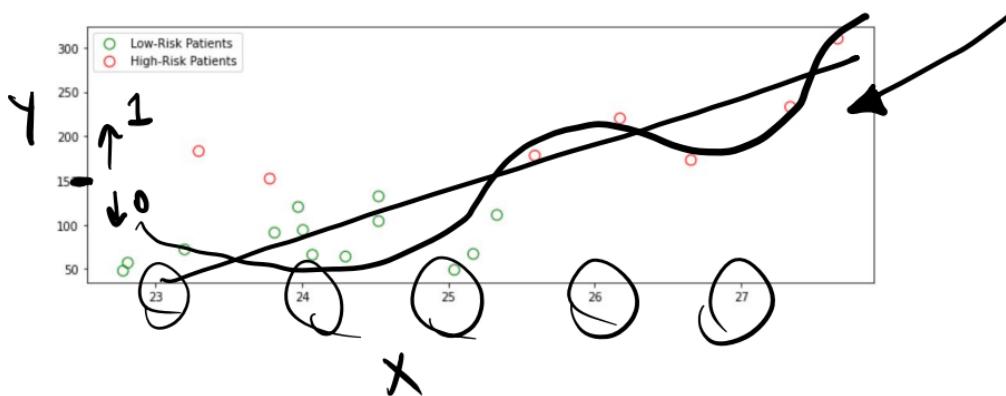
Targets: Regression vs. Classification



We can easily turn our earlier regression example into classification by discretizing the diabetes risk scores into high or low.

```
In [17]: # Discretize the targets  
diabetes_y_train_discr = np.digitize(diabetes_y_train, bins=[150])  
  
# Visualize it  
plt.scatter(diabetes_X_train[diabetes_y_train_discr==0], diabetes_y_train[diabetes_y_train_discr==0], marker='o', s=80, facecolors='none', edgecolors='g')  
plt.scatter(diabetes_X_train[diabetes_y_train_discr==1], diabetes_y_train[diabetes_y_train_discr==1], marker='o', s=80, facecolors='none', edgecolors='r')  
plt.legend(['Low-Risk Patients', 'High-Risk Patients'])
```

Out[17]: <matplotlib.legend.Legend at 0x125ffc240>

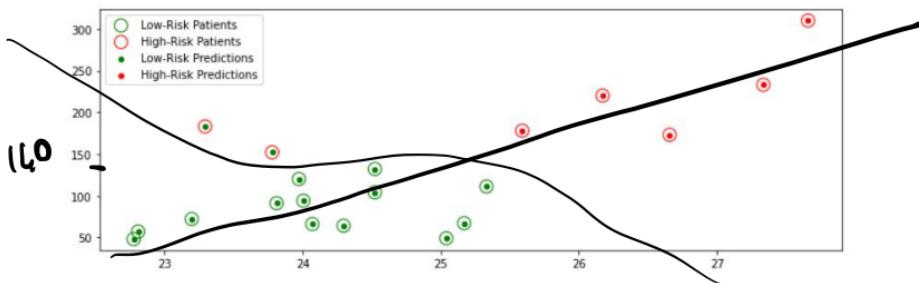


Let's try to generate predictions for this dataset.

```
In [18]: # Create logistic regression object (note: this is actually a classification algorithm!)
clf = linear_model.LogisticRegression()
# Train the model using the training sets
clf.fit(diabetes_X_train, diabetes_y_train_discr)
# Make predictions on the training set
diabetes_y_train_pred = clf.predict()

# Visualize it
plt.scatter(diabetes_X_train[diabetes_y_train_discr==0], diabetes_y_train[diabetes_y_train_discr==0], marker='o', s=140, facecolors='none', edgecolors='g')
plt.scatter(diabetes_X_train[diabetes_y_train_discr==1], diabetes_y_train[diabetes_y_train_discr==1], marker='o', s=140, facecolors='none', edgecolors='r')
plt.scatter(diabetes_X_train[diabetes_y_train_pred==0], diabetes_y_train[diabetes_y_train_pred==0], color='g', s=20)
plt.scatter(diabetes_X_train[diabetes_y_train_pred==1], diabetes_y_train[diabetes_y_train_pred==1], color='r', s=20)
plt.legend(['Low-Risk Patients', 'High-Risk Patients', 'Low-Risk Predictions', 'High-Risk Predictions'])
```

Out[18]: <matplotlib.legend.Legend at 0x11847d320>



Components of Supervised Learning

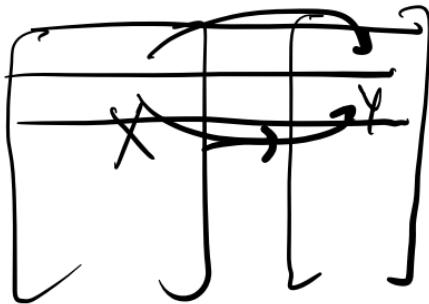
We can also define the high-level structure of a supervised learning algorithm as consisting of three components:

- A **model class**: the set of possible models we consider.
- An **objective function**, which defines how good a model is.
- An **optimizer**, which finds the best predictive model in the model class according to the objective function

Model: Notation

We'll say that a model is a function

that maps inputs $x \in \mathcal{X}$ to targets $y \in \mathcal{Y}$.



Often, models have *parameters* $\theta \in \Theta$ living in a set Θ . We will then write the model as

$f_\theta : \mathcal{X} \rightarrow \mathcal{Y}$
to denote that it's parametrized by θ .

Model Class: Notation

Formally, the model class is a set

$$\mathcal{M} \subseteq \{f \mid f : \mathcal{X} \rightarrow \mathcal{Y}\}$$

of possible models that map input features to targets.

When the models f_θ are parametrized by *parameters* $\theta \in \Theta$ living in some set Θ . Thus we can also write

$$\mathcal{M} = \{f_\theta \mid f : \mathcal{X} \rightarrow \mathcal{Y}; \theta \in \Theta\}.$$

Model Class: Example

One simple approach is to assume that x and y are related by a linear model of the form

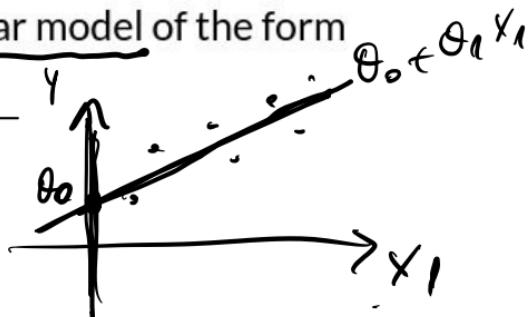
$$y = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \dots + \theta_d \cdot x_d$$

where x is a featurized output and y is the target.

The θ_j are the *parameters* of the model.

$$y = \theta_0 + \theta_1 x_1 + \dots$$

↑ ↑ ↗ ↗



Objectives: Notation

To capture this intuition, we define an *objective function* (also called a *loss function*)
 $\rightarrow J(f) : \mathcal{M} \rightarrow [0, \infty)$,

which describes the extent to which f "fits" the data

$$\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, \dots, n\}.$$



When f is parametrized by $\theta \in \Theta$, the objective becomes a function
 $J(\theta) : \Theta \rightarrow [0, \infty)$.

Objective: Examples

What would some possible objective functions? We will see many, but here are a few examples:

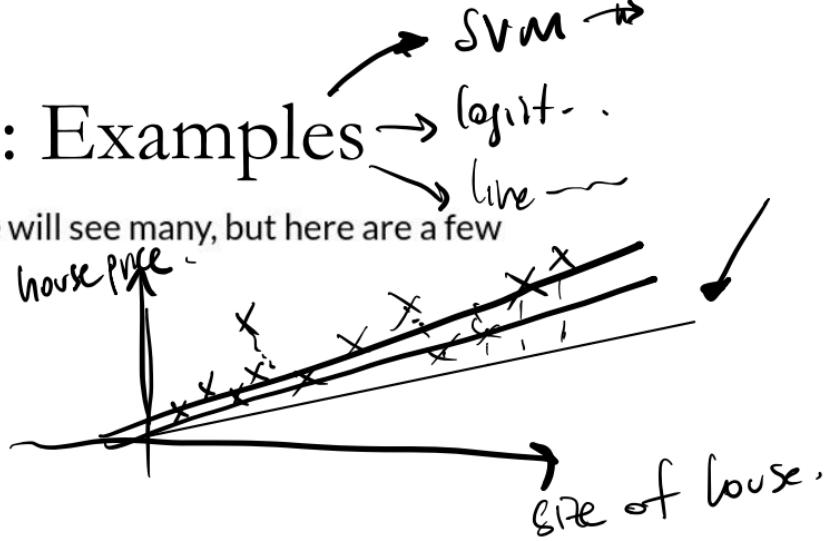
- Mean squared error:

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^n \left(\underbrace{f_\theta(x^{(i)})}_{\hat{y}^{(i)}} - \underbrace{y^{(i)}}_{=} \right)^2$$

- Absolute (L1) error:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \left| \underbrace{f_\theta(x^{(i)})}_{\hat{y}^{(i)}} - \underbrace{y^{(i)}}_{=} \right|$$

These are defined for a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, \dots, n\}$.



Objective: Examples

What would some possible objective functions? We will see many, but here are a few

```
In [60]: from sklearn.metrics import mean_squared_error, mean_absolute_error  
y1 = np.array([1, 2, 3, 4])  
y2 = np.array([-1, 1, 3, 5])  
  
print('Mean squared error: %.2f' % mean_squared_error(y1, y2))  
print('Mean absolute error: %.2f' % mean_absolute_error(y1, y2))
```

Mean squared error: 1.50

Mean absolute error: 1.00

$$\text{MSE} = \frac{1}{4} \left[(1 - 1)^2 + (2 - 1)^2 + (3 - 3)^2 + (4 - 5)^2 \right]$$
$$= \frac{1}{4} \left[0 + 1 + 0 + 1 \right] = \frac{2}{4} = 0.5$$

3. Optimizer: Notation

At a high-level an optimizer takes an objective J and a model class \mathcal{M} and finds a model $f \in \mathcal{M}$ with the smallest value of the objective J .

$$\min_{f \in \mathcal{M}} J(f)$$


Intuitively, this is the function that bests "fits" the data on the training dataset.

When f is parametrized by $\theta \in \Theta$, the optimizer minimizes a function $J(\theta)$ over all $\theta \in \Theta$.

Optimizer: Notation

We will use the a quadratic function as our running example for an objective J .

```
In [2]: import numpy as np  
import matplotlib.pyplot as plt  
plt.rcParams['figure.figsize'] = [8, 4]
```

```
In [3]: def quadratic_function(theta):  
    """The cost function, J(theta)."""  
    return 0.5*(2*theta-1)**2
```

$$\frac{1}{2} \star (2 \times \theta - 1)^2$$

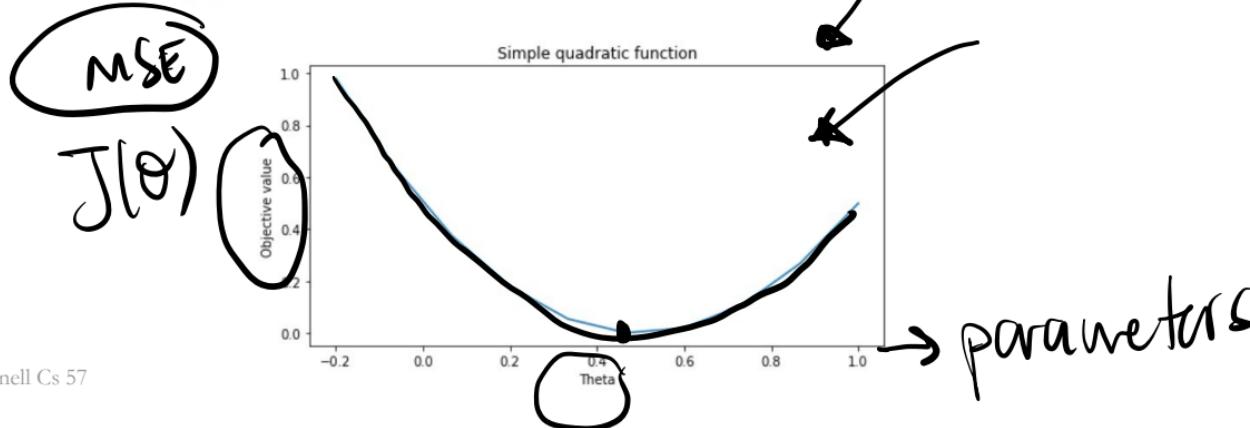
Optimizer: Notation

$J(\theta)$

We can visualize it.

```
In [4]: # First construct a grid of theta1 parameter pairs and their corresponding  
# cost function values.  
thetas = np.linspace(-0.2,1,10)  
f_vals = quadratic_function(thetas[:,np.newaxis])  
  
plt.plot(thetas, f_vals)  
plt.xlabel('Theta')  
plt.ylabel('Objective value')  
plt.title('Simple quadratic function')
```

Out[4]: Text(0.5, 1.0, 'Simple quadratic function')



Example Supervised ML Problem

Let's start with a simple example of a supervised learning problem: predicting diabetes risk.

Suppose we have a dataset of diabetes patients.

- For each patient we have access to measurements from their medical record and an estimate of diabetes risk.
- We are interested in understanding how the measurements affect an individual's diabetes risk.

A Supervised Learning Dataset

Let's return to our example: predicting diabetes risk. What would a dataset look like?

We will use the UCI Diabetes Dataset; it's a toy dataset that's often used to demonstrate machine learning algorithms.

- For each patient we have access to a measurement of their body mass index (BMI) and a quantitative diabetes risk score (from 0-400).
- We are interested in understanding how BMI affects an individual's diabetes risk.

A Supervised Learning Dataset

Let's ret

We will
machine

- F
- E
- V

In [2]:

```
import numpy as np
import pandas as pd
from sklearn import datasets

# Load the diabetes dataset
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True, as_frame=True)

# Use only the BMI feature
diabetes_X = diabetes_X.loc[:, ['bmi']]

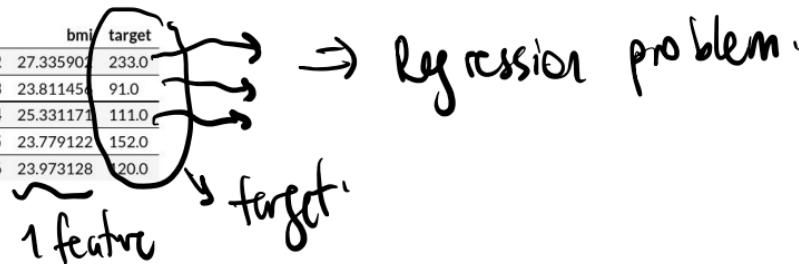
# The BMI is zero-centered and normalized; we recenter it for ease of presentation
diabetes_X = diabetes_X * 30 + 25

# Collect 20 data points
diabetes_X_train = diabetes_X.iloc[-20:]
diabetes_y_train = diabetes_y.iloc[-20:]

# Display some of the data points
pd.concat([diabetes_X_train, diabetes_y_train], axis=1).head()
```

Out[2]:

	bmi	target
422	27.335901	233.0
423	23.811450	91.0
424	25.331171	111.0
425	23.779122	152.0
426	23.973128	120.0



ok like?

demonstrate

; index

diabetes risk.

A Supervised Learning Dataset

We can also visualize this two-dimensional dataset.

Let's ret

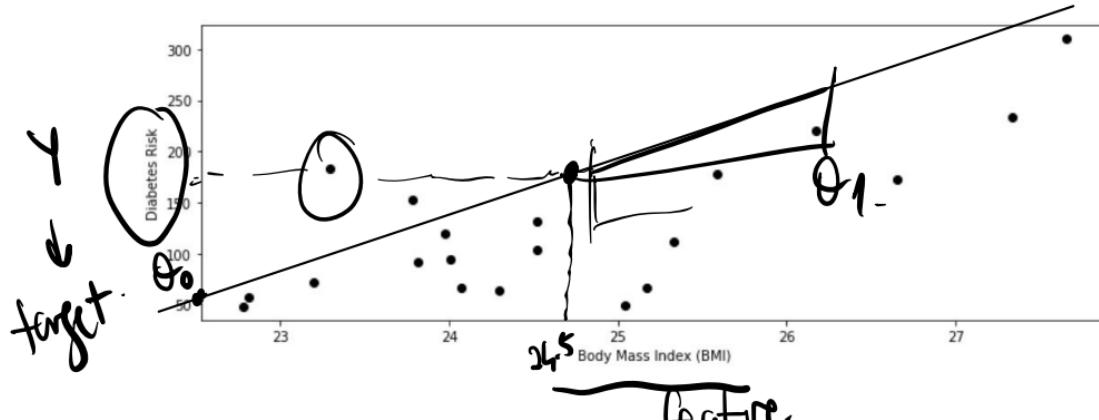
We will
machine

- F
- E
- V

```
In [3]: %matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

plt.scatter(diabetes_X_train, diabetes_y_train, color='black')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
```

Out[3]: Text(0, 0.5, 'Diabetes Risk')



Slide Credit:

k like?

nonstrate

index

abetes risk.

A Supervised Learning Algorithm

What is the relationship between BMI and diabetes risk?

We could assume that risk is a linear function of BMI. In other words, for some unknown $\theta_0, \theta_1 \in \mathbb{R}$, we have

= =

$$y = \theta_1 \cdot x + \theta_0,$$

where x is the BMI (also called the dependent variable), and y is the diabetes risk score (the independent variable).

target

Note that θ_1, θ_0 are the slope and the intercept of the line relates x to y . We call them *parameters*.

A Supervised Learning Algorithm

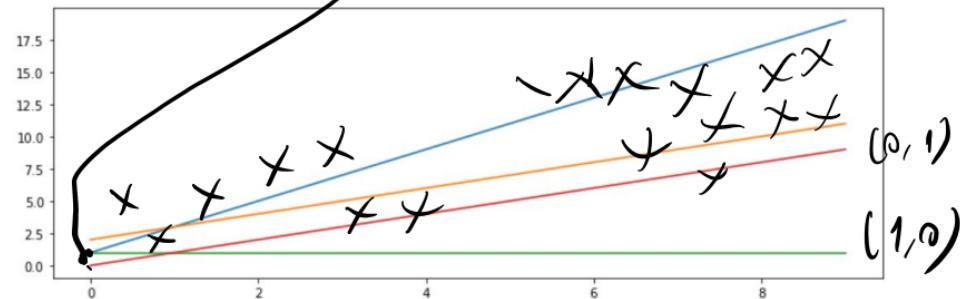
We can visualize this for a few values of θ_1, θ_0 .

In [4]:

```
theta_list = [(1, 2), (2, 1), (-1, 0), (0, 1)]  
for theta0, theta1 in theta_list:  
    x = np.arange(10)  
    y = theta1 * x + theta0  
    plt.plot(x,y)
```

$$(\theta_0, \theta_1) (\theta_0, \theta_1) ==$$

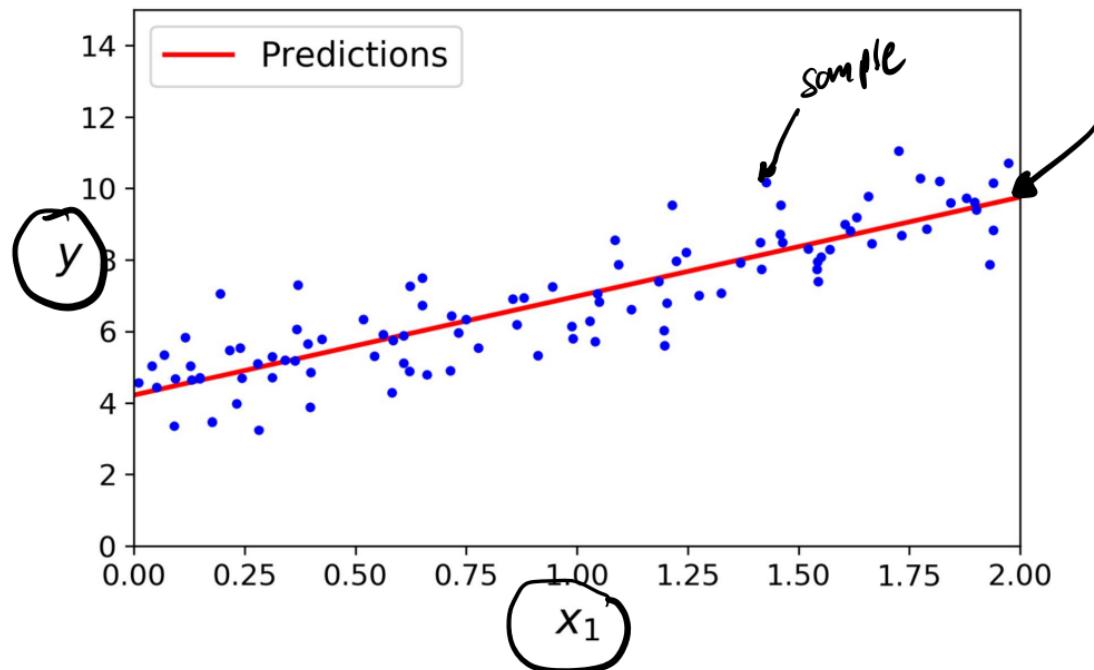
$$y = \theta_1 x + \theta_0$$



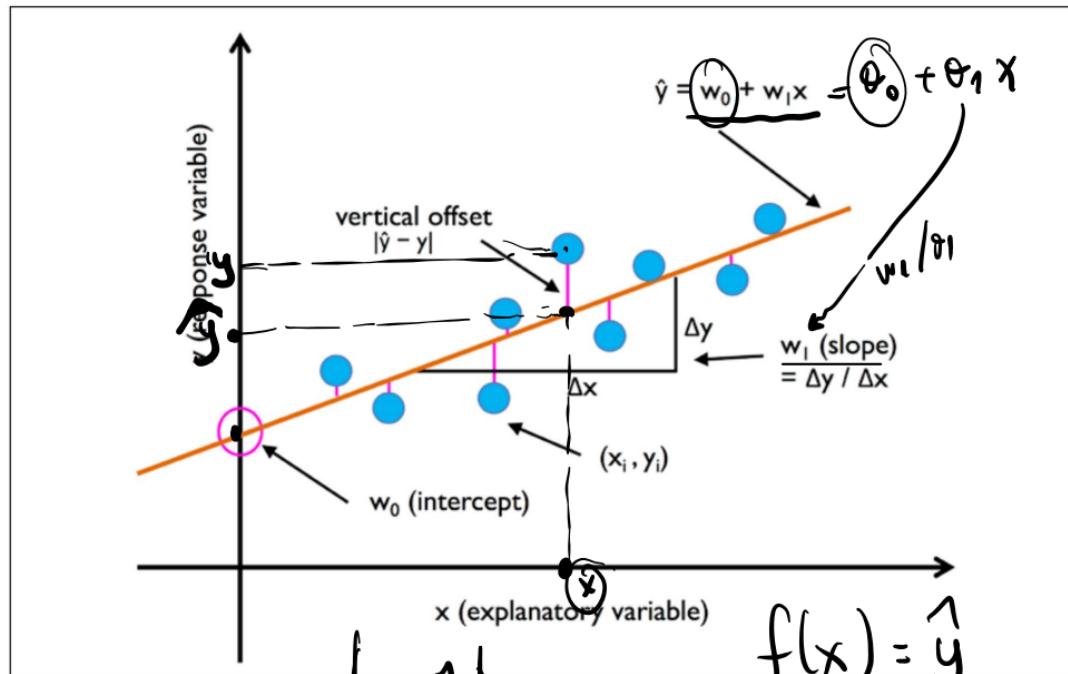
$$\theta_0=0$$

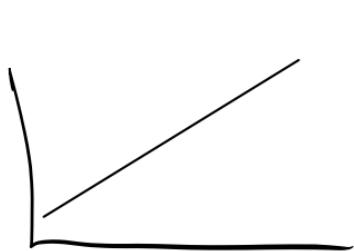
,

Linear Regression



Linear Regression





$$y = \underline{\theta_0} + \underline{\theta_1}x$$

Linear Regression

Assuming that x, y follow the above linear relationship, the goal of the **supervised learning algorithm** is to find a good set of parameters consistent with the data.

We will see many algorithms for this task. For now, let's call the `sklearn.linear_model` library to find a θ_1, θ_0 that fit the data well.

Linear Regression

Assum
learni

We wi
skle

```
In [6]: from sklearn import linear_model
from sklearn.metrics import mean_squared_error

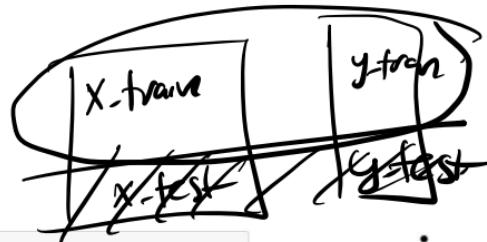
# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)

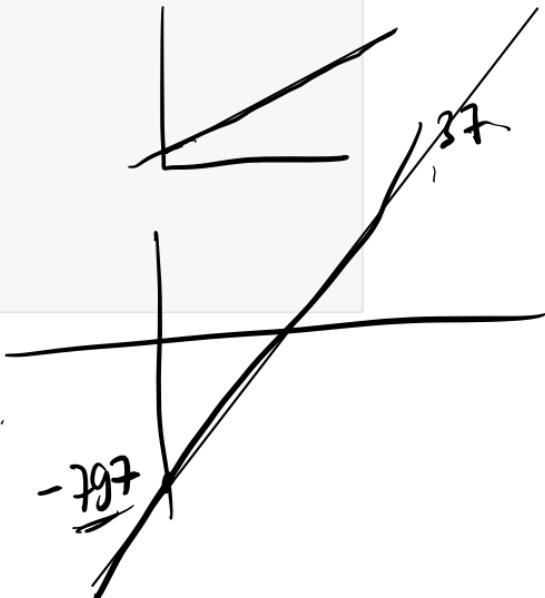
# Make predictions on the training set
diabetes_y_train_pred = regr.predict(diabetes_X_train)

# The coefficients
print('Slope (theta1): \t', regr.coef_[0])
print('Intercept (theta0): \t', regr.intercept_)
```

Slope (theta1): 37.378842160517664
Intercept (theta0): -797.0817390342369



ervised
ata.



Linear Regression

The supervised learning algorithm gave us a pair of parameters θ_1^*, θ_0^* . These define the *predictive model* f^* , defined as

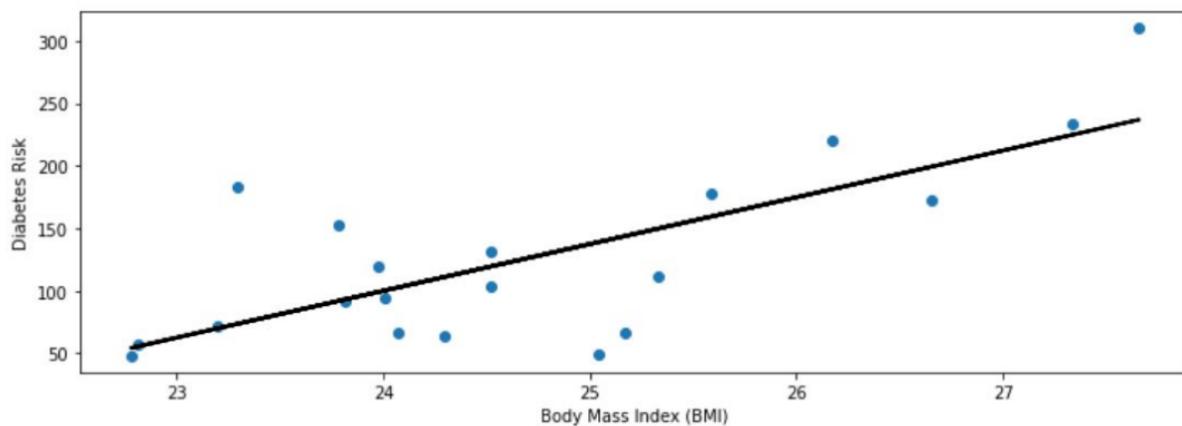
$$f(x) = \theta_1^* \cdot x + \theta_0^*,$$

where again x is the BMI, and y is the diabetes risk score.

Linear Regression

```
In [7]: plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(diabetes_X_train, diabetes_y_train)
plt.plot(diabetes_X_train, diabetes_y_train_pred, color='black', linewidth=2)
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x1253f9240>]
```



Predictions Using Supervised Learning

Given a new dataset of patients with a known BMI, we can use this model to estimate their diabetes risk.

Given a new \hat{x} , we can output a predicted \hat{y}' as

$$\hat{y}' = f(\hat{x}') = \theta_1^* \cdot \hat{x}' + \theta_0.$$

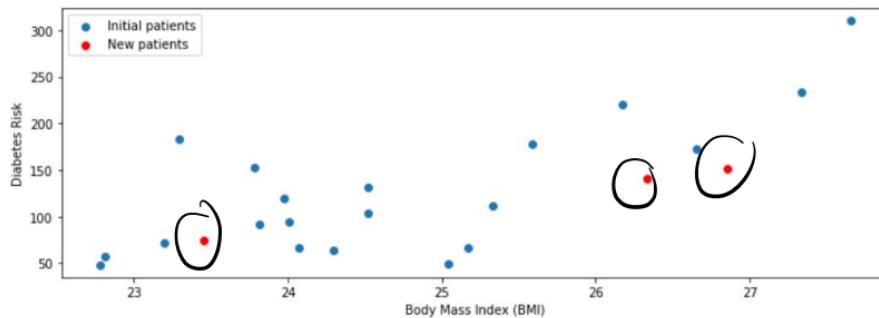


Let's start by loading more data. We will load three new patients (shown in red below) that we haven't seen before.

```
In [8]: # Collect 3 data points
diabetes_X_train = diabetes_X.iloc[:3]
diabetes_y_train = diabetes_y.iloc[:3]

plt.scatter(diabetes_X_train, diabetes_y_train)
plt.scatter(diabetes_X_test, diabetes_y_test, color='red')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.legend(['Initial patients', 'New patients'])
```

Out[8]: <matplotlib.legend.Legend at 0x1259cd390>

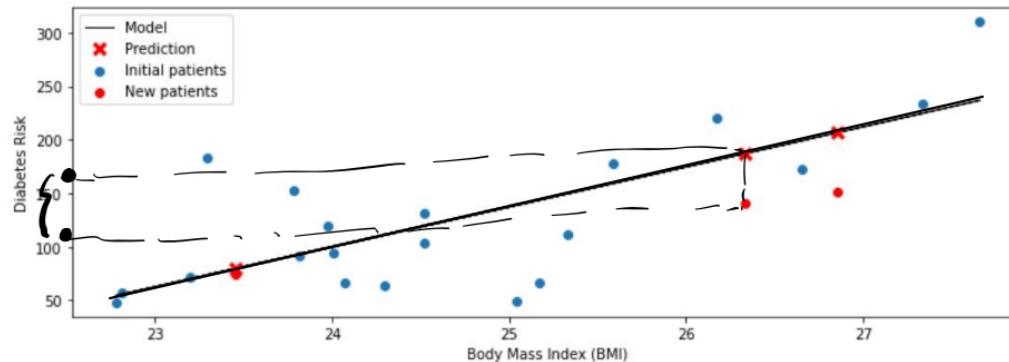


Our linear model provides an estimate of the diabetes risk for these patients.

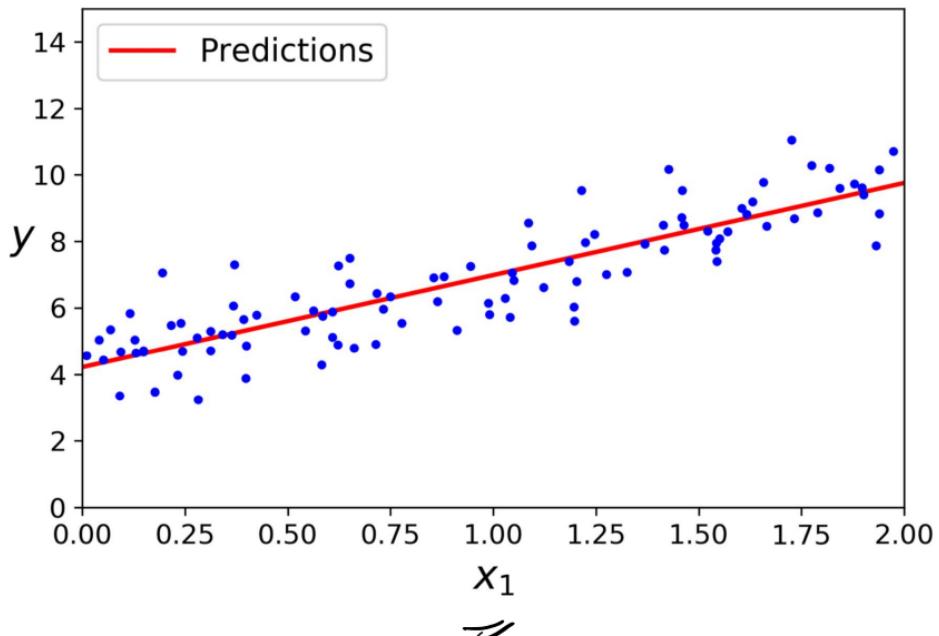
```
In [9]: # generate predictions on the new patients
diabetes_y_test_pred = regr.predict(diabetes_X_test)

# visualize the results
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(diabetes_X_train, diabetes_y_train)
plt.scatter(diabetes_X_test, diabetes_y_test, color='red', marker='o')
plt.plot(diabetes_X_train, diabetes_y_train_pred, color='black', linewidth=1)
plt.plot(diabetes_X_test, diabetes_y_test_pred, 'x', color='red', mew=3, marker
size=8)
plt.legend(['Model', 'Prediction', 'Initial patients', 'New patients'])
```

Out[9]: <matplotlib.legend.Legend at 0x125bfb048>



Linear Regression



$$\hat{y} = \underbrace{w^T \mathbf{x}}_{\text{wavy bracket}} + b = \sum_{i=1}^p w_i x_i + b$$
$$= \underbrace{w_1 x_1 + w_2 x_2 + w_3 x_3 + \dots}_{\text{wavy bracket}} + b$$
$$\theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_0$$

Linear Model

Recall that a linear model has the form

$$y = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \dots + \theta_d \cdot x_d$$

where $x \in \mathbb{R}^d$ is a vector of features and y is the target. The θ_j are the *parameters* of the model.

By using the notation $x_0 = 1$, we can represent the model in a vectorized form

$$f_{\theta}(x) = \sum_{j=0}^d \theta_j \cdot x_j = \theta^\top x.$$

Linear Model

Let's define our model in Python.

```
In [26]: def f(X, theta):
    """The linear model we are trying to fit.

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional data matrix

    Returns:
    y_pred (np.array): n-dimensional vector of predicted targets
    """
    return X.dot(theta)
```

$$f = X \cdot \theta^T$$

$$\begin{aligned} f &= \underline{\underline{w^T X}} \\ &= \underline{\underline{\theta^T X}} \end{aligned}$$

An Objective: Mean Squared Error

$$f = \boxed{\theta^\top x = \hat{y}}$$

We pick θ to minimize the mean squared error (MSE). Slight variants of this objective are also known as the residual sum of squares (RSS) or the sum of squared residuals (SSR).

$$\min_{\theta} J(\theta) = \frac{1}{2n} \sum_{i=1}^n (y^{(i)} - \underline{\theta^\top x^{(i)}})^2$$

In other words, we are looking for the best compromise in θ over all the data points.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

An Objective: Mean Squared Error

Let's implement mean squared error.

```
In [27]: def mean_squared_error(theta, X, y):
    """The cost function, J, describing the goodness of fit.
```

Parameters:

theta (np.array): d-dimensional vector of parameters

X (np.array): (n,d)-dimensional design matrix

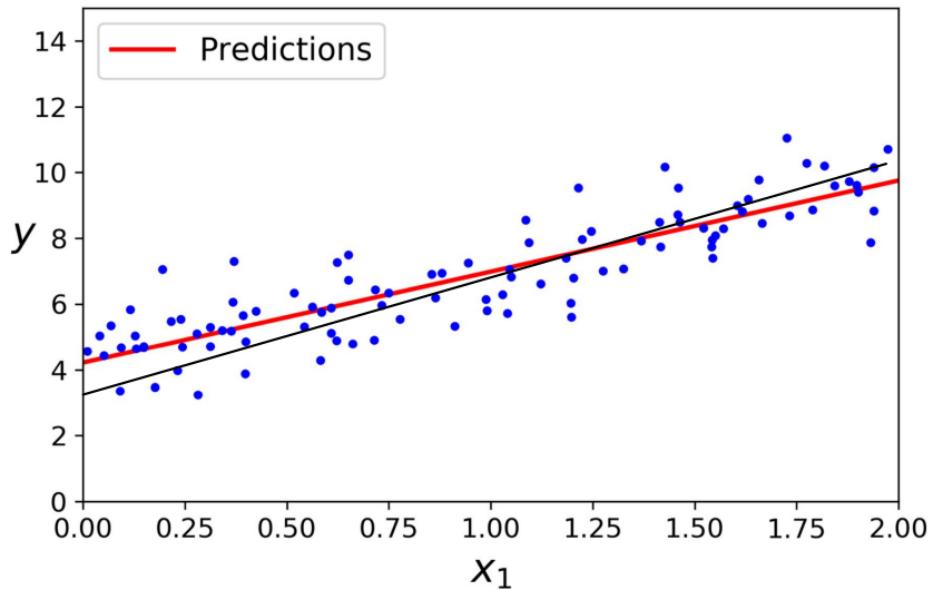
y (np.array): n-dimensional vector of targets

"""

```
return 0.5*np.mean((y-f(X, theta))**2)
```

$$\frac{1}{2n} \sum (y - f(x, \theta))^2$$

Linear Regression



$$\hat{y} = w^T \mathbf{x} + b = \sum_{i=1}^p w_i x_i + b$$

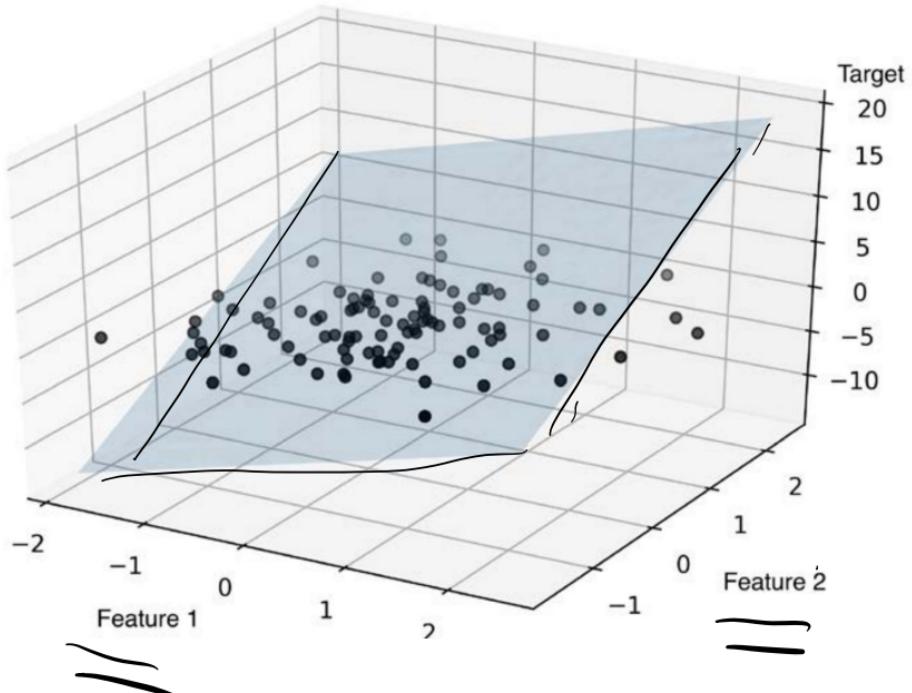
function $\Rightarrow \hat{y}$

$$\min_{w \in \mathbb{R}^p, b \in \mathbb{R}} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

• ground truth
• real value

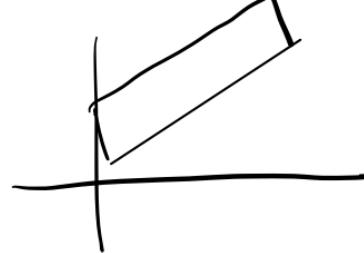
Diagram illustrating the cost function for linear regression. The cost function is shown as a sum of squared differences between the predicted values (\hat{y}_i) and the ground truth values (y_i). The optimization problem is to find the parameters w and b that minimize this cost function.

Multiple Linear Regression → $y = w_0 + w_1 X$



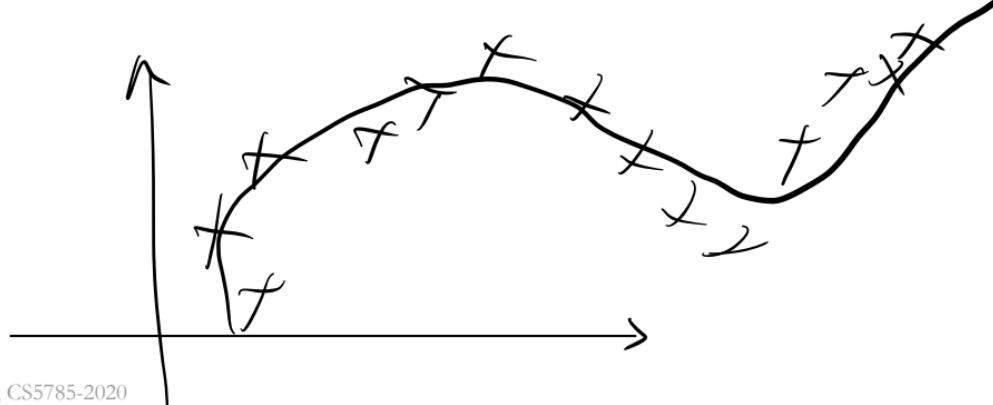
$$y = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^n w_i x_i = w^T x$$

Non Linear Least Squares



So far, we have learned about a very simple linear model. These can capture only simple linear relationships in the data. How can we use what we learned so far to model more complex relationships?

We will now see a simple approach to model complex non-linear relationships called *least squares*.



Non Linear Least Squares

Recall that a polynomial of degree p is a function of the form

$$a_p x^p + a_{p-1} x^{p-1} + \dots + a_1 x + a_0.$$

Below are some examples of polynomial functions.

Non Linear Least Squares

```
In [24]: import warnings
warnings.filterwarnings("ignore")

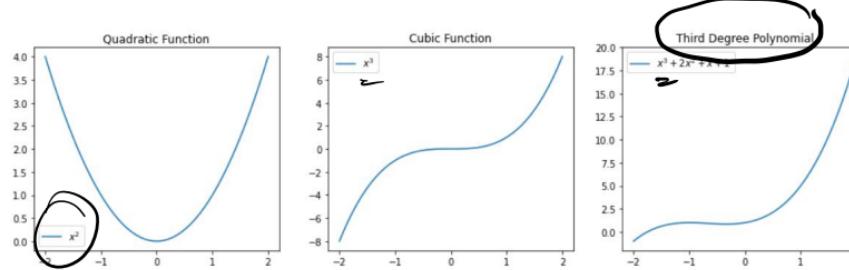
plt.figure(figsize=(16,4))
x_vars = np.linspace(-2, 2)

plt.subplot('131')
plt.title('Quadratic Function')
plt.plot(x_vars, x_vars**2)
plt.legend(["$x^2$"])

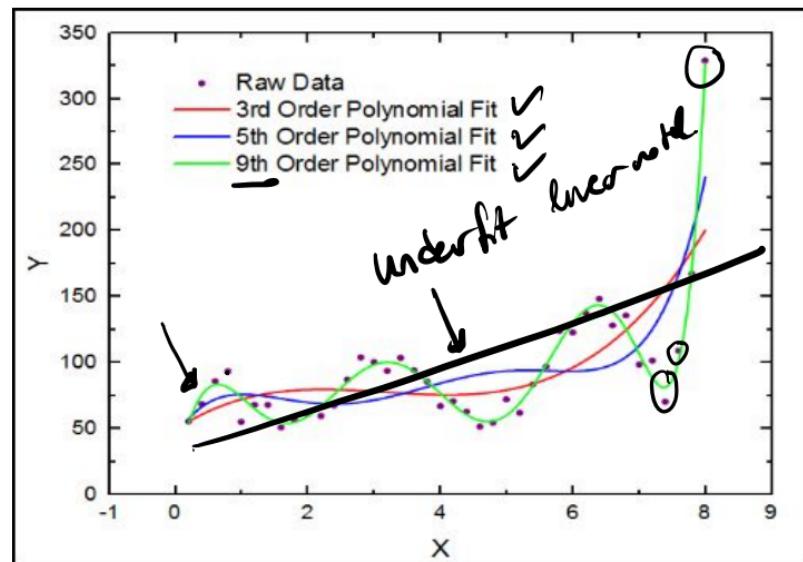
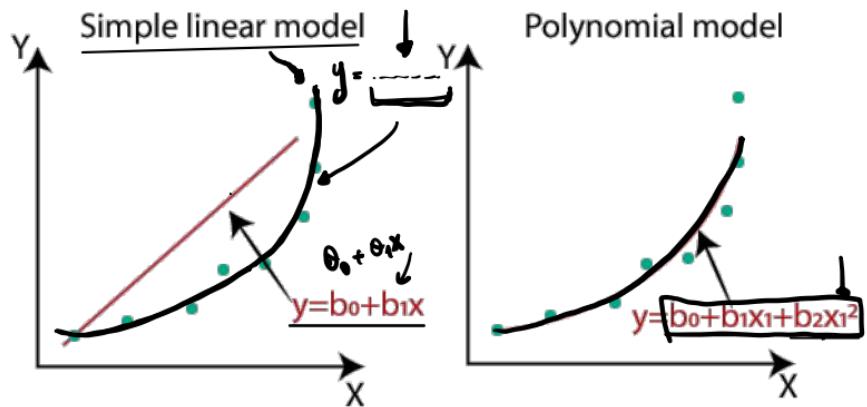
plt.subplot('132')
plt.title('Cubic Function')
plt.plot(x_vars, x_vars**3)
plt.legend(["$x^3$"])

plt.subplot('133')
plt.title('Third Degree Polynomial')
plt.plot(x_vars, x_vars**3 + 2*x_vars**2 + x_vars + 1)
plt.legend(["$x^3 + 2 x^2 + x + 1$"])
```

Out[24]: <matplotlib.legend.Legend at 0x128ed2ac8>



Polynomial Regression



Non Linear Least Squares

```
In [24]: import warnings
warnings.filterwarnings("ignore")

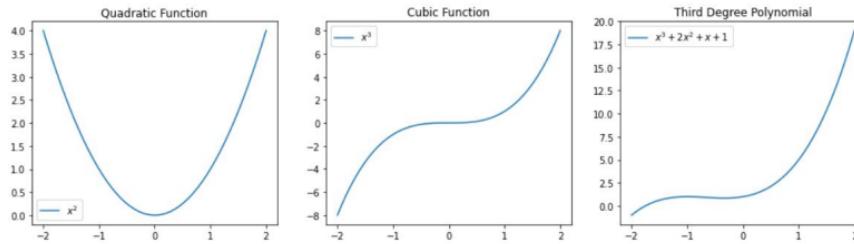
plt.figure(figsize=(16,4))
x_vars = np.linspace(-2, 2)

plt.subplot('131')
plt.title('Quadratic Function')
plt.plot(x_vars, x_vars**2)
plt.legend(["$x^2$"])

plt.subplot('132')
plt.title('Cubic Function')
plt.plot(x_vars, x_vars**3)
plt.legend(["$x^3$"])

plt.subplot('133')
plt.title('Third Degree Polynomial')
plt.plot(x_vars, x_vars**3 + 2*x_vars**2 + x_vars + 1)
plt.legend(["$x^3 + 2 x^2 + x + 1$"])
```

Out[24]: <matplotlib.legend.Legend at 0x128ed2ac8>



```
In [25]: %matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 4]

import numpy as np
import pandas as pd
from sklearn import datasets

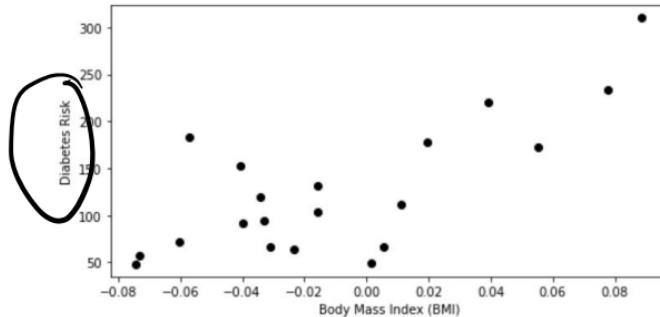
# Load the diabetes dataset
X, y = datasets.load_diabetes(return_X_y=True, as_frame=True)

# add an extra column of ones
X['one'] = 1

# Collect 20 data points
X_train = X.iloc[-20:]
y_train = y.iloc[-20:]

plt.scatter(X_train.loc[:,['bmi']], y_train, color='black')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
```

Out[25]: Text(0, 0.5, 'Diabetes Risk')



Diabetes Dataset: A Non-Linear Featurization

Let's now obtain linear features for this dataset.

```
In [26]: X_bmi = X_train.loc[:, ['bmi']]  
X_bmi_p3 = pd.concat([X_bmi, X_bmi**2, X_bmi**3], axis=1)  
X_bmi_p3.columns = ['bmi', 'bmi2', 'bmi3']  
X_bmi_p3['one'] = 1  
X_bmi_p3.head()
```

Out[26]:

	bmi	bmi2	bmi3	one
422	0.077863	0.006063	0.000472	1
423	-0.039618	0.001570	-0.000062	1
424	0.011039	0.000122	0.000001	1
425	-0.040696	0.001656	-0.000067	1
426	0.034229	0.001172	-0.000040	1

$$\theta_1 X_1 + \theta_2 X^2 + \theta_3 X^3$$

$$x_1 + x_2 + x_3 + \dots$$

Diabetes

Prediction

In [27]:

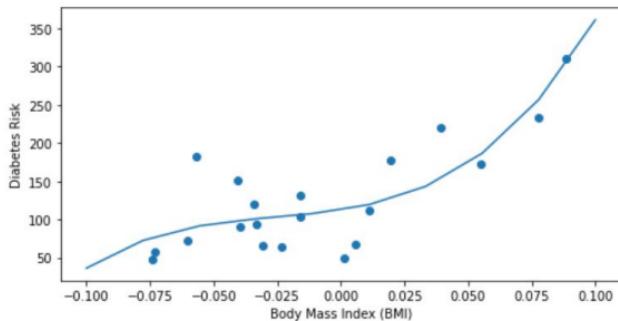
```
# Fit a linear regression
theta = np.linalg.inv(X_bmi_p3.T.dot(X_bmi_p3)).dot(X_bmi_p3.T).dot(y_train)

# Show the learned polynomial curve
x_line = np.linspace(-0.1, 0.1, 10)
x_line_p3 = np.stack([x_line, x_line**2, x_line**3, np.ones(10,)], axis=1)
y_train_pred = x_line_p3.dot(theta)

plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(X_bmi, y_train)
plt.plot(x_line, y_train_pred)
```

Out[27]: [`<matplotlib.lines.Line2D at 0x1292c99e8>`]

fit //



Evaluation Metrics for Regression

- Pearson correlation
- R-squared, or R^2
- Adjusted- R-squared
- Mean Squared Error
- Mean Absolute Error
- Root Mean Squared Error

Pearson Correlation

- Pearson's r
- Is a measure of linear correlation between two sets of data.
- It is the ratio between the covariance of two variables and the product of their standard deviations.
- The results always has a value between -1 and 1.

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}}$$

Where,

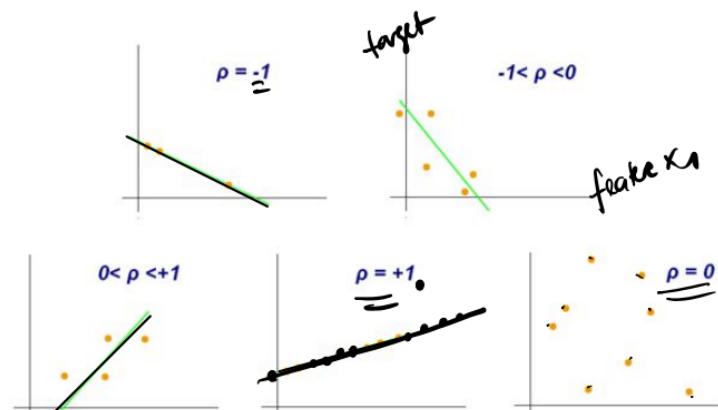
r = Pearson Correlation Coefficient

x_i = x variable samples

y_i = y variable sample

\bar{x} = mean of values in x variable

\bar{y} = mean of values in y variable



Coefficient of Determination (R^2)

- R-squared, or R^2
- R-squared ranges from 0 to 1 and measures the proportion of variation in the data that is accounted for in the model.
- It is useful mainly in explanatory uses of regression where you want to assess how well the model fits the data.
- **Note:** It does not take into consideration of overfitting problem. If your model has many independent variables, due to model is too complicated, it may fit very well to the training data but performs badly for testing data.



$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$



Adjusted R²

test error
= = =

- The disadvantage of the R_square score is while adding new features in data the R_square score starts increasing or remains constant but never decreases because it assumes that while adding more data variance of data increases.

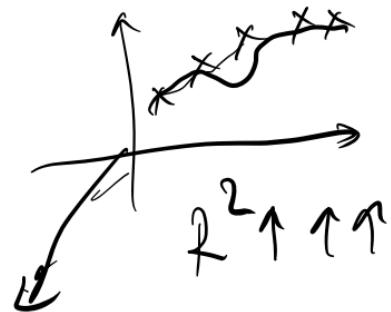
$$\text{Adjusted } R^2 = 1 - \frac{(1 - R^2)(N - 1)}{N - p - 1}$$

Where

R^2 Sample R-Squared

N Total Sample Size

p Number of independent variable



- Where where n is the total number of observations and p is the number of predictors. Adjusted R² will always be less than or equal to R².

Adjusted R²

The diagram illustrates three cases (Case 1, Case 2, Case 3) of data tables and their corresponding R_squared and Adjusted R_squared values.

Data Tables:

Case 1		Case 2			Case 3		
Var1	Y	Var1	Var2	Y	Var1	Var2	Y
x1	y1	x1	2*x1	y1	x1	2*x1+0.1	y1
x2	y2	x2	2*x2	y2	x2	2*x2	y2
x3	y3	x3	2*x3	y3	x3	2*x3 + 0.1	y3
x4	y4	x4	2*x4	y4	x4	2*x4	y4
x5	y5	x5	2*x5	y5	x5	2*x5 + 0.1	y5

R_squared and Adj_R_squared Values:

	Case 1	Case 2	Case 3
R_squared	0.985	0.985	0.987
Adj_R_squared	0.981	0.971	0.975

Annotations highlight the following points:

- A large circle encloses the entire first row of the Case 1 table.
- Arrows point from the circled "Var1" and "Y" in Case 1 to the "Var1" and "Y" in Case 2.
- Arrows point from the circled "Var1" and "Var2" in Case 2 to the "Var1" and "Var2" in Case 3.
- Arrows point from the circled "R_squared" values in Case 1 and Case 2 to the circled "Adj_R_squared" value in Case 3.
- A bracket labeled "R²" is shown above the circled "R_squared" value in Case 3.
- A bracket labeled "Adj R²" is shown above the circled "Adj_R_squared" value in Case 3.

Mean Square Error (MSE) //

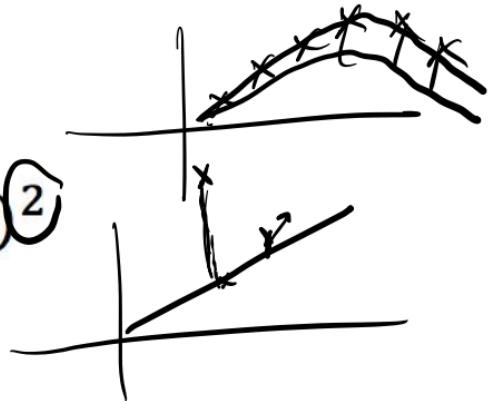
- MSE is calculated by the sum of square of prediction error which is real output minus predicted output and the divide by the number of data points.
- It is hard to interpret many insights from one single result but it gives you a real number to compare against other model results and help you select the best regression model.
- If you have outliers in the dataset then it penalizes the outliers most and the calculated MSE is bigger (-).

Lower values better.

1M\$

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

MSE → 200,000



Mean Absolute Error (MAE)

lower values are better.

- Same unit as the output variable (+)
- Robust to outliers (+)

$$MAE = \frac{1}{n} \sum \left| \underbrace{y - \hat{y}}_{\text{The absolute value of the residual}} \right|$$

Divide by the total number of data points

Predicted output value

Actual output value

Sum of

The absolute value of the residual



Root Mean Square Error (MSE)

lower.

$$RMSE = \sqrt{\sum_{i=1}^n \frac{(\hat{y}_i - y_i)^2}{n}} \text{MSE}$$

Choosing the right metric

- A nice blog post that you can read more about metrics for evaluation regression models.
 - <https://medium.com/usf-msds/choosing-the-right-metric-for-machine-learning-models-part-1-a99d7d7414e4>

Case 1: Actual Values = [2,4,6,8], Predicted Values = [4,6,8,10] $\quad // \quad m_1$

Case 2: Actual Values = [2,4,6,8], Predicted Values = [4,6,8,12] $\quad m_2$

MAE for case 1 = 2.0, RMSE for case 1 = 2.0

MAE for case 2 = 2.5, RMSE for case 2 = 2.65

Calculus Review: Derivatives

Recall that the derivative

$$\frac{df(\theta_0)}{d\theta}$$

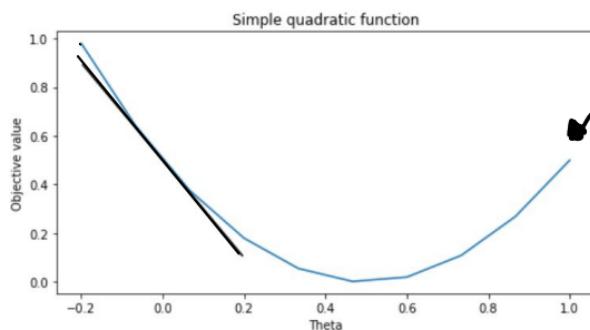
of a univariate function $f : \mathbb{R} \rightarrow \mathbb{R}$ is the instantaneous rate of change of the function $f(\theta)$ with respect to its parameter θ at the point θ_0 .

Calculus Review: Derivatives

```
In [5]: def quadratic_derivative(theta):
    return (2*theta-1)*2
    df0 = quadratic_derivative(np.array([[0]])) # derivative at zero
    f0 = quadratic_function(np.array([[0]]))
    line_length = 0.2

    plt.plot(thetas, f_vals)
    plt.annotate('', xytext=(0-line_length, f0-line_length*df0), xy=(0+line_length,
    f0+line_length*df0),
                  arrowprops={'arrowstyle': '-.', 'lw': 1.5}, va='center', ha='cente
    r')
    plt.xlabel('Theta')
    plt.ylabel('Objective value')
    plt.title('Simple quadratic function')
```

Out[5]: Text(0.5, 1.0, 'Simple quadratic function')



$$f(\theta) = \theta^2$$
$$f'(\theta) = 2\theta$$
$$f'(0) = 2(2*0-1)$$

Calculus Review: Partial Derivatives

The partial derivative

$$\frac{\partial f(\theta_0)}{\partial \theta_j}$$

of a multivariate function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is the derivative of f with respect to θ_j while all other inputs θ_k for $k \neq j$ are fixed.



Calculus Review: The Gradient

The gradient $\nabla_{\theta} f$ further extends the derivative to multivariate functions $f : \mathbb{R}^d \rightarrow \mathbb{R}$, and is defined at a point θ_0 as

$$\nabla_{\theta} f(\theta_0) = \begin{bmatrix} \frac{\partial f(\theta_0)}{\partial \theta_1} \\ \frac{\partial f(\theta_0)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta_0)}{\partial \theta_d} \end{bmatrix}.$$

The j -th entry of the vector $\nabla_{\theta} f(\theta_0)$ is the partial derivative $\frac{\partial f(\theta_0)}{\partial \theta_j}$ of f with respect to the j -th component of θ .

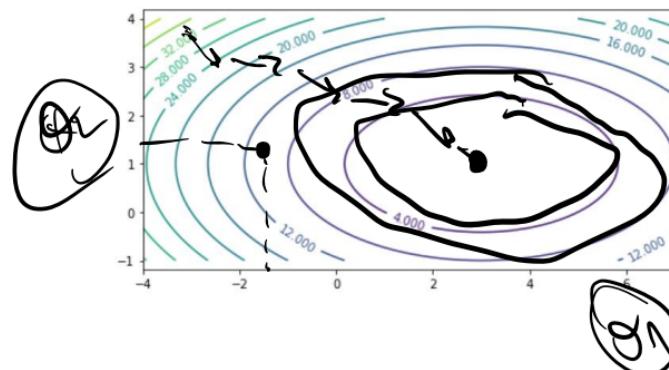
Calculus Review: The Gradient

Let's visualize this function.

```
In [8]: theta0_grid = np.linspace(-4,7,101)
theta1_grid = np.linspace(-1,4,101)
theta_grid = theta0_grid[np.newaxis,:], theta1_grid[:,np.newaxis]
J_grid = quadratic_function2d(theta0_grid[np.newaxis,:], theta1_grid[:,np.newaxis])

X, Y = np.meshgrid(theta0_grid, theta1_grid)
contours = plt.contour(X, Y, J_grid, 10)
plt.clabel(contours)
plt.axis('equal')
```

Out[8]: (-4.0, 7.0, -1.0, 4.0)

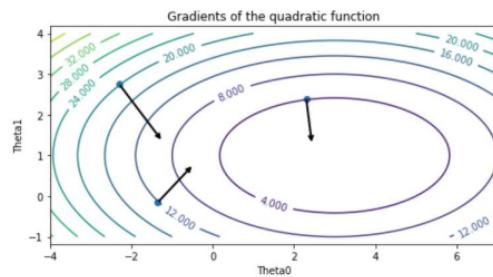


Calculus Review: The Gradient

```
In [10]: theta0_pts, thetal_pts = np.array([2.3, -1.35, -2.3]), np.array([2.4, -0.15, 2.75])
dfs = quadratic_derivative2d(theta0_pts, thetal_pts)
line_length = 0.2

contours = plt.contour(X, Y, J_grid, 10)
for theta0_pt, thetal_pt, df0 in zip(theta0_pts, thetal_pts, dfs):
    plt.annotate('', xytext=(theta0_pt, thetal_pt),
                 xy=(theta0_pt-line_length*df0[0], thetal_pt-line_length*df0[1]),
                 arrowprops={'arrowstyle': '->', 'lw': 2}, va='center', ha='center')
plt.scatter(theta0_pts, thetal_pts)
plt.clabel(contours)
plt.xlabel('Theta0')
plt.ylabel('Theta1')
plt.title('Gradients of the quadratic function')
plt.axis('equal')
```

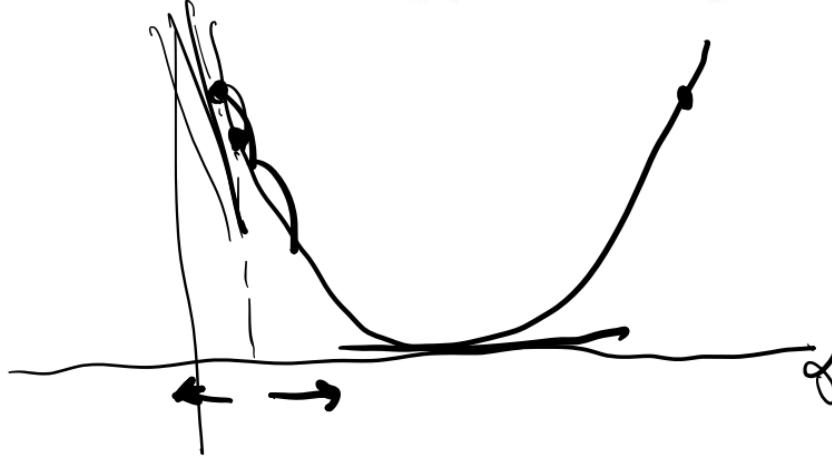
Out[10]: (-4.0, 7.0, -1.0, 4.0)



Gradient Descent

Gradient descent is a very common optimization algorithm used in machine learning.

The intuition behind gradient descent is to repeatedly obtain the gradient to determine the direction in which the function decreases most steeply and take a step in that direction.



Slide Credit: Cornell CS 5785

Gradient Descent: Notation

More formally, if we want to optimize $J(\theta)$, we start with an initial guess θ_0 for the parameters and repeat the following update until θ is no longer changing:

$$\theta_i := \theta_{i-1} - \alpha \cdot \nabla_{\theta} J(\theta_{i-1}).$$

As code, this method may look as follows:

```
theta, theta_prev = random_initialization()
while norm(theta - theta_prev) > convergence_threshold:
    theta_prev = theta
    theta = theta_prev - step_size * gradient(theta_prev)
```

In the above algorithm, we stop when $\|\theta_i - \theta_{i-1}\|$ is small.

Mean Squared Error: Partial Derivatives

Let's work out what a partial derivative is for the MSE error loss for a linear model.

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_j} &= \underbrace{\frac{\partial}{\partial \theta_j} \frac{1}{2} (f_{\theta}(x) - y)^2}_{(f_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} (f_{\theta}(x) - y)} \\ &= \underbrace{(f_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{k=0}^d \theta_k \cdot x_k - y \right)}_{(f_{\theta}(x) - y) \cdot x_j} \\ &= (f_{\theta}(x) - y) \cdot x_j\end{aligned}$$

$\frac{\partial J(\theta)}$
 $\frac{\partial J(\theta)}$
MSE

Mean Squared Error: Partial Derivatives

Let's implement the gradient.

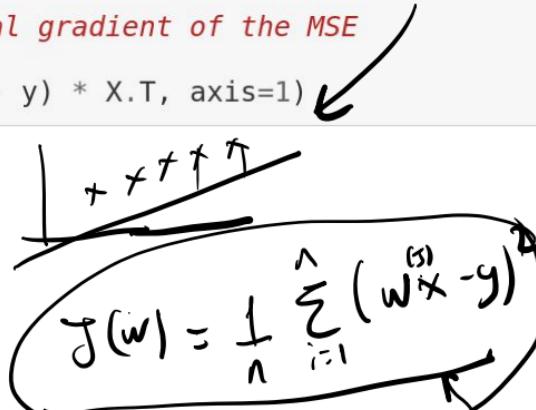
```
In [28]: def mse_gradient(theta, X, y):
    """The gradient of the cost function.

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional design matrix
    y (np.array): n-dimensional vector of targets

    Returns:
    grad (np.array): d-dimensional gradient of the MSE
    """
    return np.mean((f(X, theta) - y) * X.T, axis=1)
```

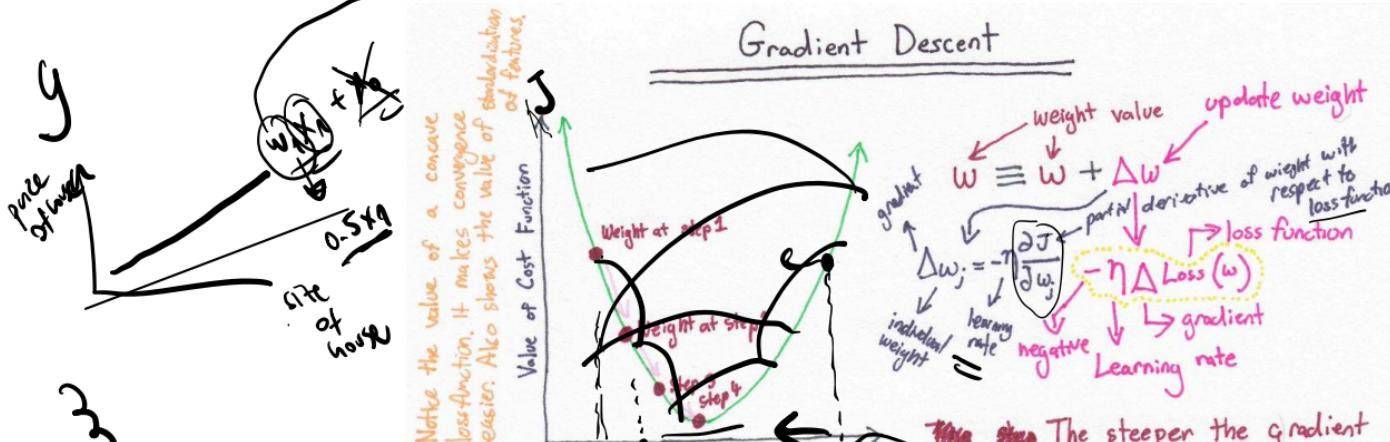
continuous
convex
differentiable

$$\begin{aligned} & w^T x \\ & = \theta^T x \\ & = \text{MSE} \end{aligned}$$





Gradient Descent $\frac{dJ}{dw_j}$



IF LEARNING RATE IS TOO SMALL: convergence takes a long time.

IF LEARNING RATE IS TOO LARGE: we will overshoot the bottom of the loss function.

$w_{\text{new}} > w_{\text{old}}$

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \left(\frac{dJ}{dw_j} \right) < 0$$

We can now visualize gradient descent.

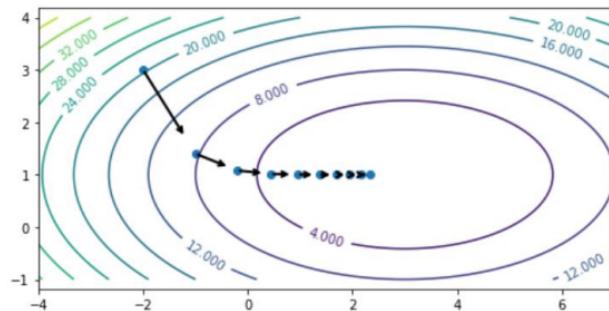
```
In [25]: opt_pts = np.array(opt_pts)
opt_grads = np.array(opt_grads)

contours = plt.contour(X, Y, J_grid, 10)
plt.clabel(contours)
plt.scatter(opt_pts[:,0], opt_pts[:,1])

for opt_pt, opt_grad in zip(opt_pts, opt_grads):
    plt.annotate('', xytext=(opt_pt[0], opt_pt[1]),
                 xy=(opt_pt[0]-0.8*step_size*opt_grad[0], opt_pt[1]-0.8*step_size*opt_grad[1]),
                 arrowprops={'arrowstyle': '->', 'lw': 2}, va='center', ha='center')

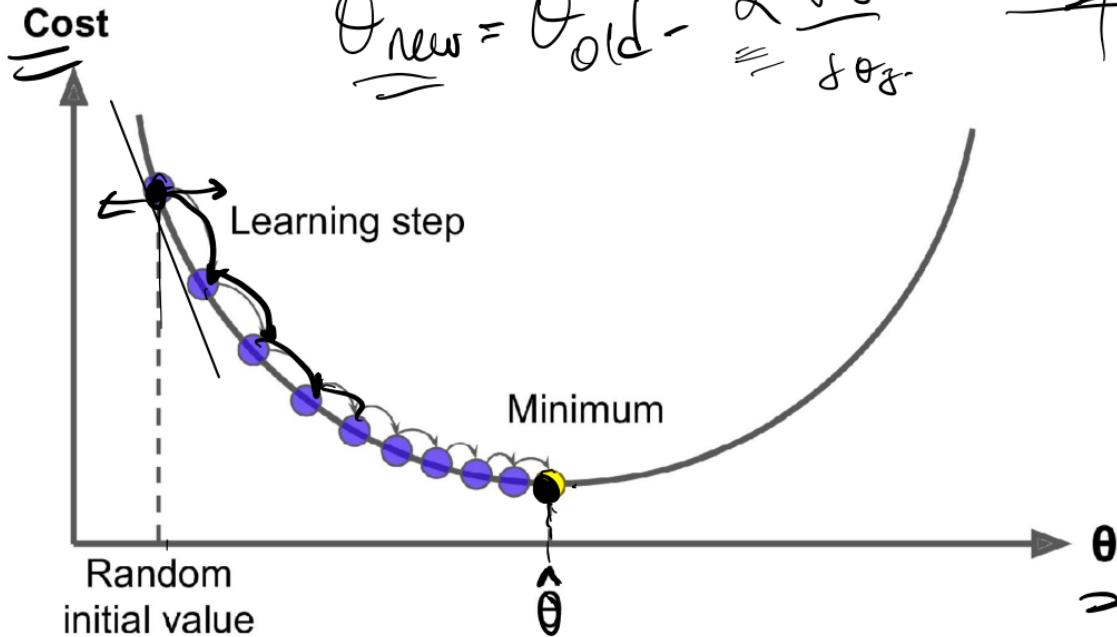
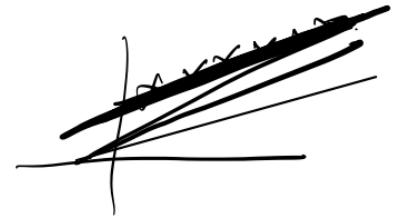
plt.axis('equal')
```

Out[25]: (-4.0, 7.0, -1.0, 4.0)

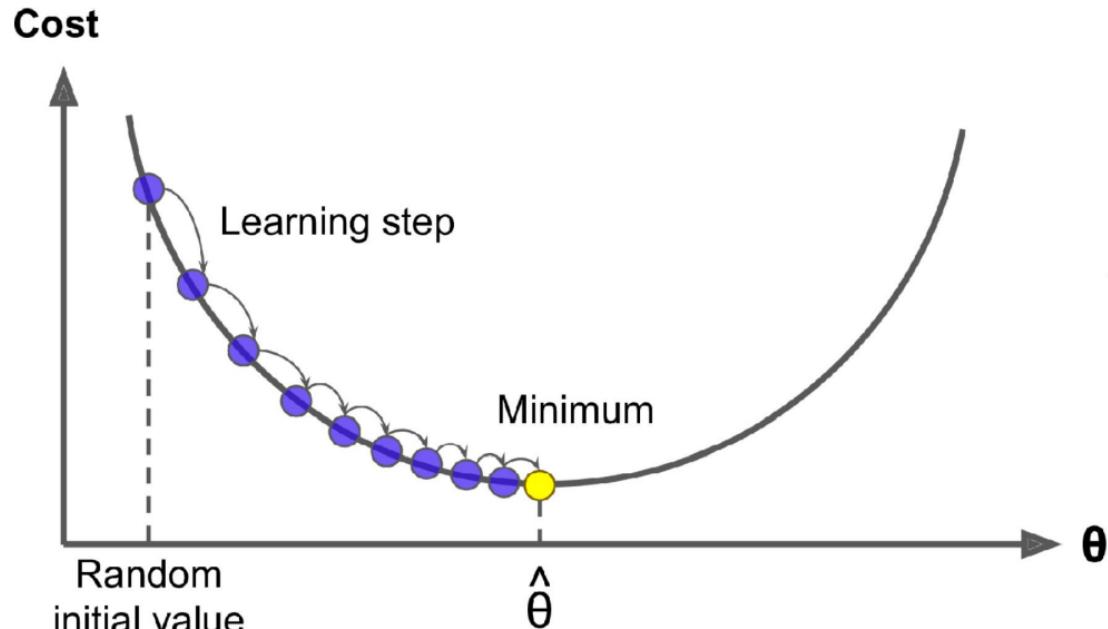


Gradient Descent

$$\theta_{\text{new}} = \theta_{\text{old}} - \frac{\alpha}{\|f\|} \nabla f$$



Gradient Descent



Equation 4-7. Gradient Descent step

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$$

Gradient Descent

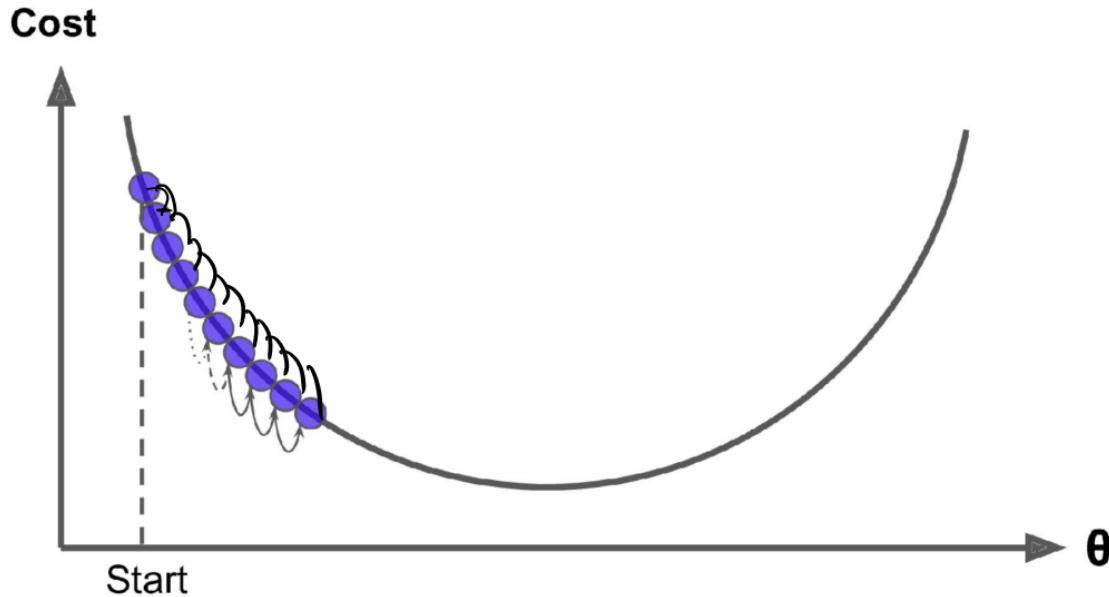


Figure 4-4. The learning rate is too small

Gradient Descent

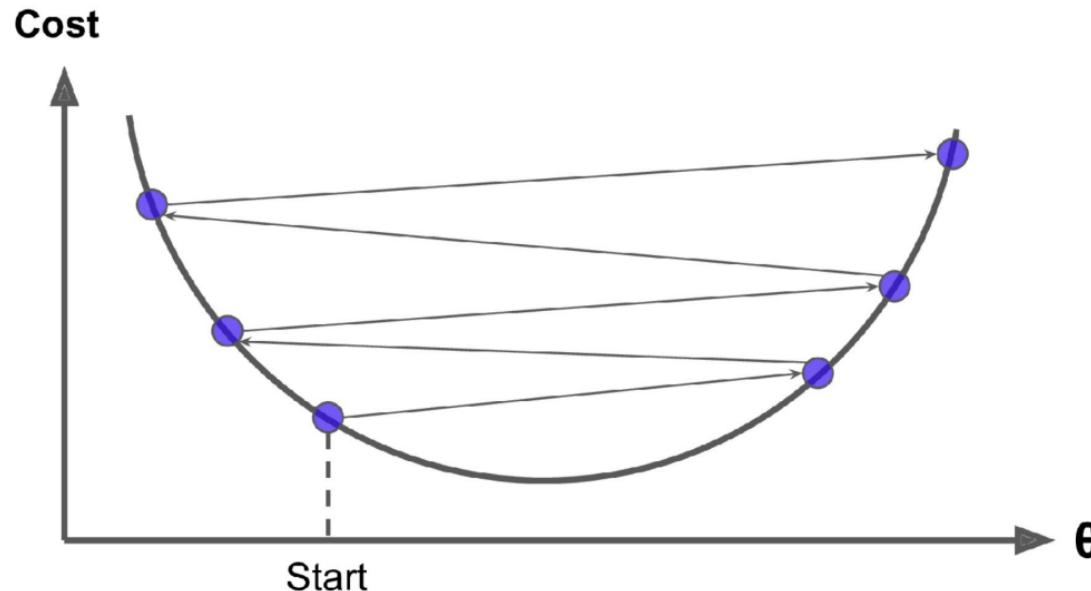


Figure 4-5. The learning rate is too large

Gradient Descent

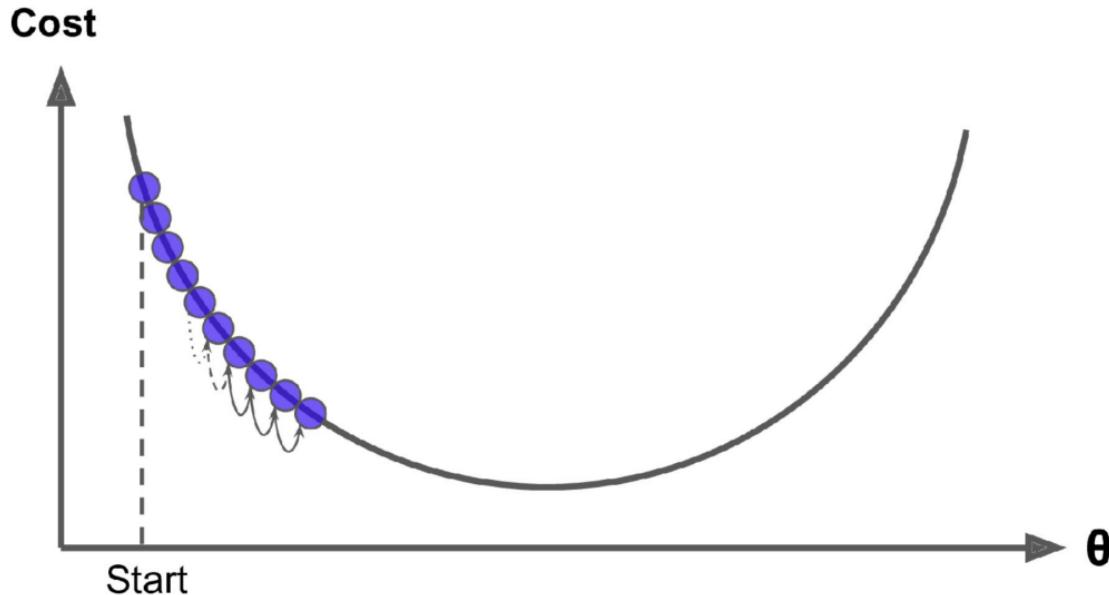
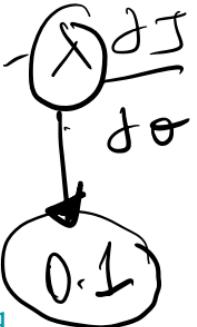
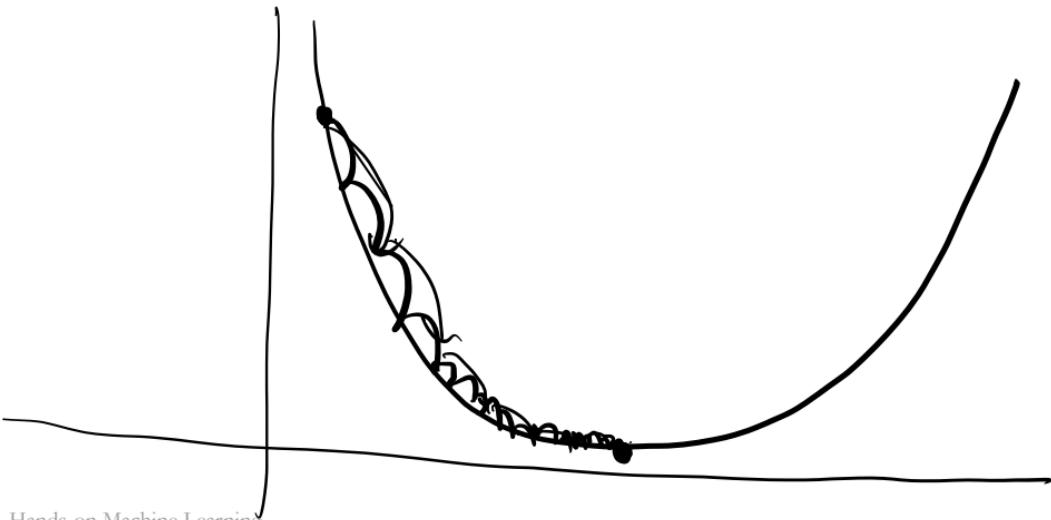


Figure 4-4. The learning rate is too small

Gradient Descent

$$w = w - \alpha \frac{\partial J}{\partial w}$$


- Check this blog post for interactive explanation of Linear Regression
 - https://machinelearningcompass.com/machine_learning_math/gradient_descent_for_linear_regression/



Adaptive learning Rates: \rightarrow Deep learning
 \rightarrow Adagrad.

RMS prop.

Adam.

Adam Grad.

- Each parameter
- on the historical gradients for that parameter

for each parameter θ_i and its gradient g_i :

1. Accumulate squared gradient.

$$s_{\theta_i} = s_{\theta_i} + g_i^2$$

2. Update the parameter.

$$\theta_i = \theta_i - \frac{\alpha}{s_{\theta_i} + \epsilon} g_i$$

$1e^{-8}$

