Machine Learning Algorithms

- Supervised (has "right answers")
- Unsupervised

→ Tools are important
→ learning how to apply is even more important.

(Given Output labels)

Supervised (Majorly)

Regression

Predicting a number from infinitely many possible outputs.

Classification

Predict finite set of possible output - categories.

Unsupervised

Clustering

Ex: Clustering customers into homogeneous market segments.

Terminologies: (Regression)

i) Training Set: Data used to train the model.

Notation:

- features / "input" variables (x)
- target / "output" variable (y)
- No. of training examples (m)
- single training example $(x^{(i)}, y^{(i)})$

training set

↓

learning Algorithm

↓

$x \Rightarrow \boxed{f} \Rightarrow \hat{y}$

model

$f_{w,b}(x) = wx + b$

or, $f(x) = wx + b$

w, b : Parameters

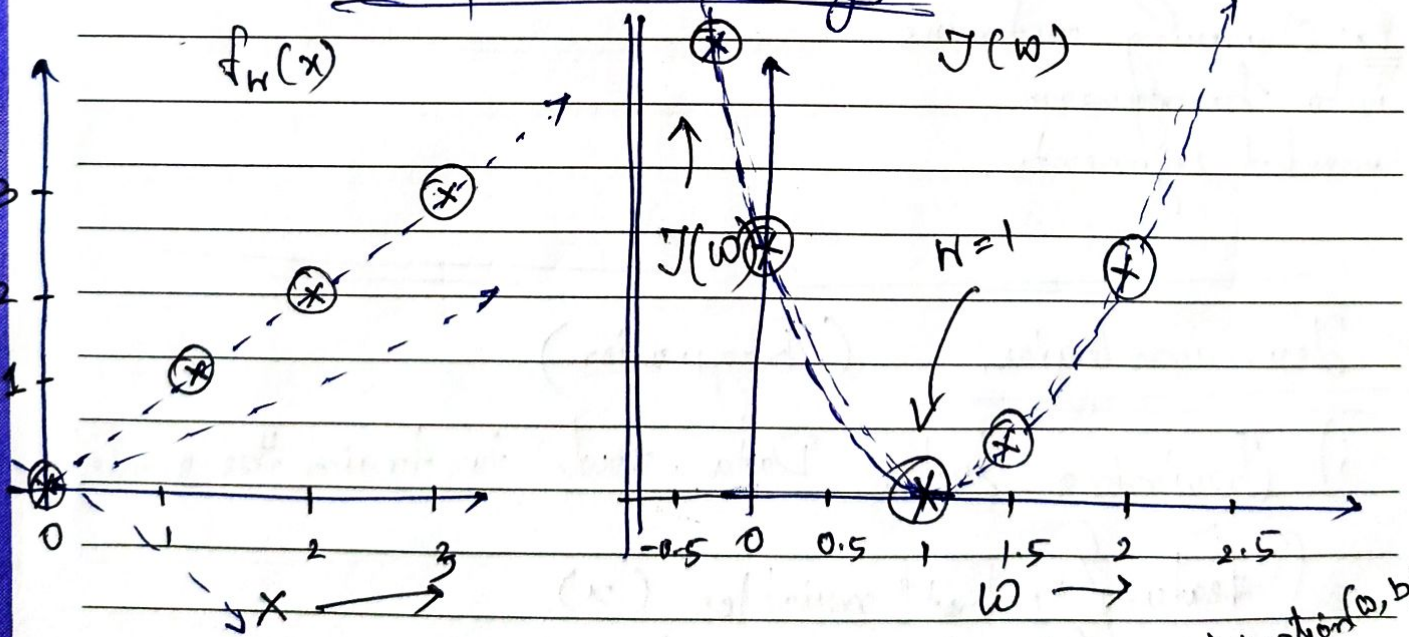
Cost fuction :

$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$$

here, m is the No. of training examples.

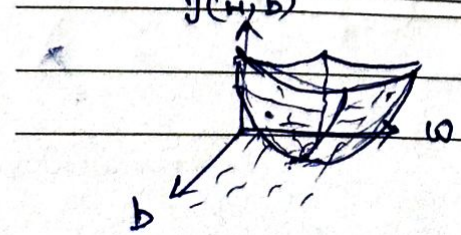
goal : minimize $J(w,b)$

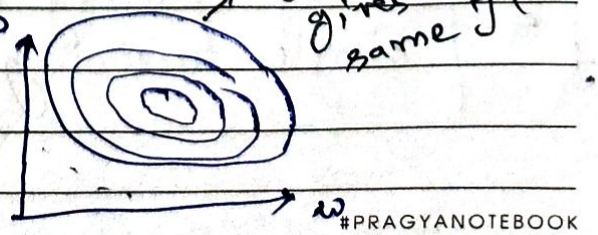
Graphical Analogy (say $b = 0$)



Contour Plots

$J(w,b)$





Gradient Descent → function used to minimize any function

repeat until convergence

Algorithm: (Simultaneous Update)

{
$tmp\_w = w - \alpha \frac{\partial}{\partial w} J(w,b)$ ← assignment

$tmp\_b = b - \alpha \frac{\partial}{\partial b} \{J(w,b)\}$

$w = tmp\_w$

$b = tmp\_b$
}

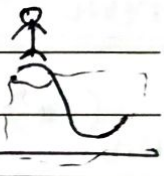
α = learning rate (>0)

Intution of Gradient Descent

α = length of each step

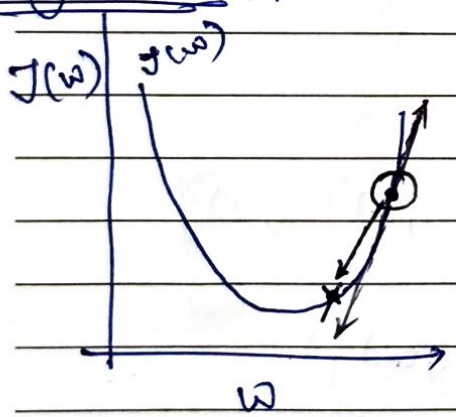
derivative term = Direction

Rember the Valley example??



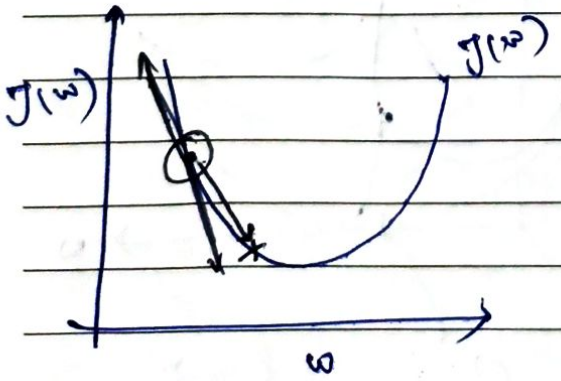
Say $b=0$ (for simplification)

(*) $\alpha > 0$ always



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

$\frac{\partial}{\partial w} J(w) > 0$



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

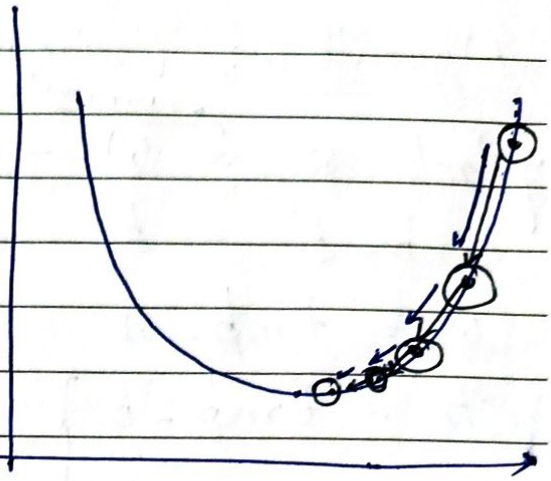
$\frac{\partial}{\partial w} J(w) < 0$

(*) Once reached a local minima $J(w) = 0$, so, w will not change.

Choosing efficient learning rate (α)

1. Near a local minimum:
   - Derivative becomes smaller.
   - Update step becomes smaller.

∴ Can reach minimum without decreasing learning rate.



Linear Regression Model

$$f_{w,b}(x) = wx + b$$

Cost Function

$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right)^2$$

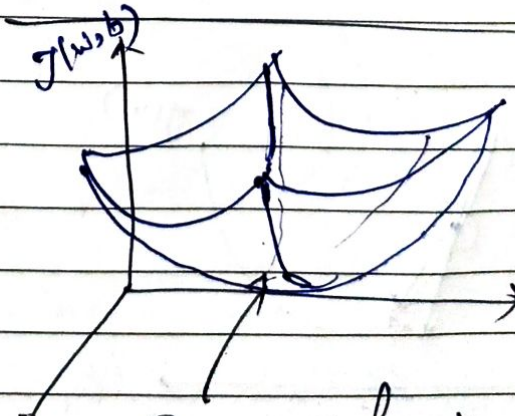
Gradient Descent Algorithm:

repeat until convergence { (update w, b simultaneously)

$$w = w - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right) x^{(i)} \right]$$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)} - y^{(i)}) \right)$$

}



Convex function like this will always converge to a global minima

Multiple Features:

$x_j \Rightarrow j^{th}$ feature

$\vec{x}^{(i)} \Rightarrow$ features of i^{th} training example

$x_j^{(i)} \Rightarrow$ value of feature j in i^{th} training example.

⇒ New Model Notation:

$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$$

(Dot Product)

Hence $\vec{w} = [w_1, w_2 \dots w_n]$

$\vec{x} = [x_1, x_2 \dots x_n]$

Vectorization:

```
f = np.dot(w, x) + b
```

→ Shorter code

→ faster computation due to parallel processing

Behind the Scene:

Gradient Descent

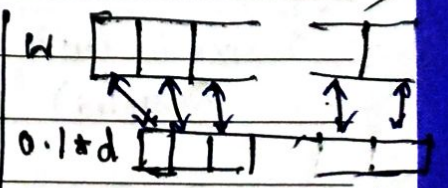
$\vec{w} = (w_1, w_2 \dots w_{16})$

$\vec{d} = (d_1, d_2 \dots d_{16})$

```
for j in range(16):
    w[j] = w[j] - (0.1 * d[j])
```

iterates 16 steps each at a time

```
w = np.array([0.5, 1.3, ..., 3.4])
d = np.array([0.3, 0.2, ..., 0.4])
w = w - (0.1 * d)
```



⇒ parallel multiplication + addition in a single step.

Gradient Descent

One feature

repeat {

$$w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right) x^{(i)}$$

$$\rightarrow \frac{\partial}{\partial w} J(w,b)$$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right)$$

n feature ($n > 1$)

repeat {

$$w_1 = w_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_1^{(i)}$$

$$\rightarrow \frac{\partial}{\partial w_1} J(\vec{w},b)$$

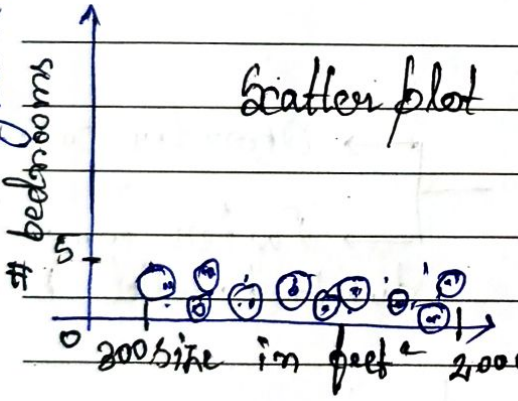
$$w_n = w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_n^{(i)}$$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right)$$

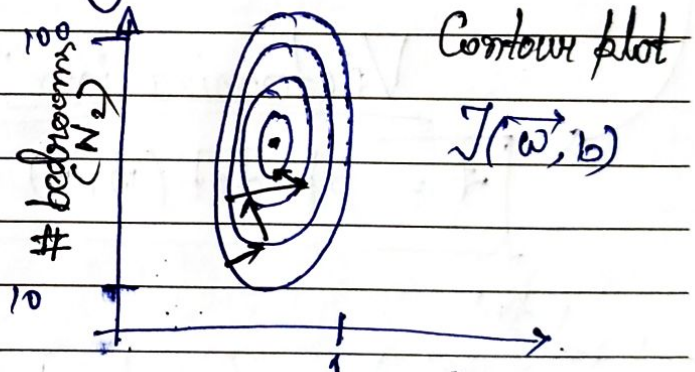
different ranges of values
slows down gradient descent

Feature Scaling

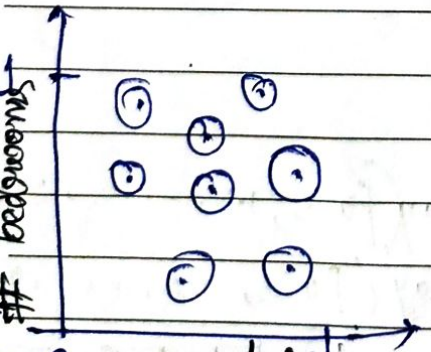
Scatter plot

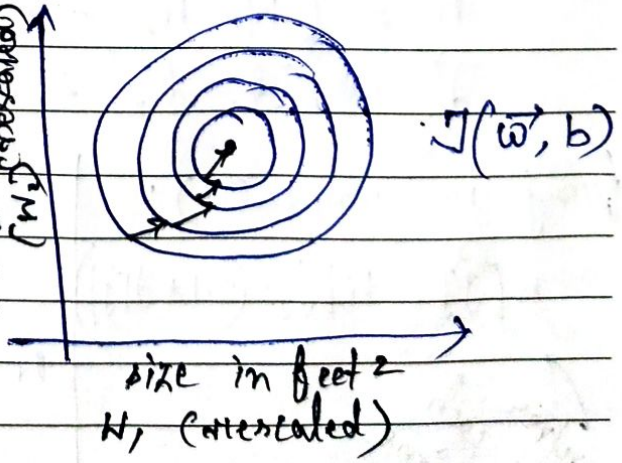


Contour plot $J(\vec{w},b)$

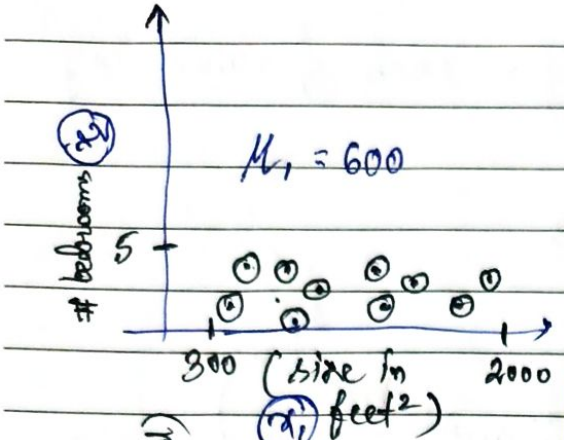


scaling makes it run faster
significantly





Mean Normalization



$300 \le x_1 \le 2000$

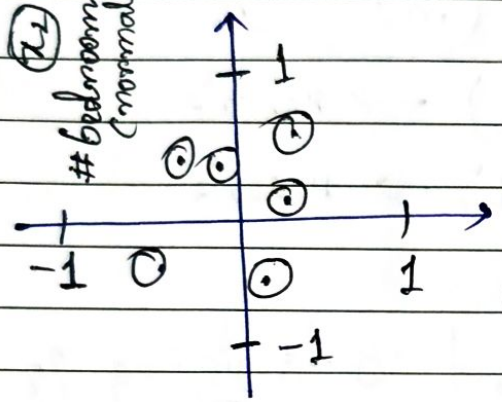
$x_1 = \frac{x_1 - \mu_1}{2000 - 300}$

$-0.18 \le x_1 \le 0.82$

$0 \le x_2 \le 5$

$x_2 = \frac{x_2 - \mu_2}{5 - 0}$

$-0.46 \le x_2 \le 0.54$



$$x = \frac{x - \text{mean}}{\max - \min}$$

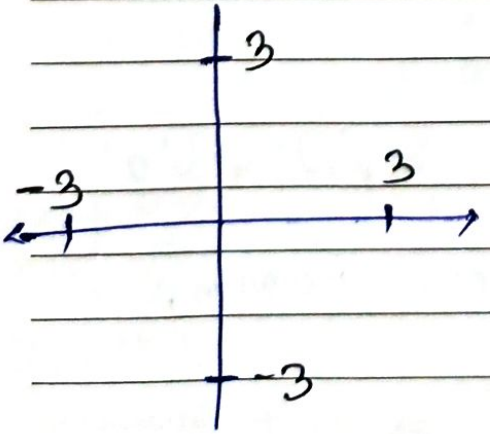
Z-score Normalization

$300 \le x_1 \le 2000$

$-0.67 \le \frac{x_1 - \mu_1}{\sigma_1} \le 3.1$

$0 \le x_2 \le 5$

$-1.6 \le \frac{x_2 - \mu_2}{\sigma_2} \le 1.9$



Where to scale & where not needed

Aim for about acceptable ranges:
- $-1 \le x_j \le 1$ for each feature x_j
- $-3 \le x_j \le 3$
- $-0.3 \le x_j \le 0.3$

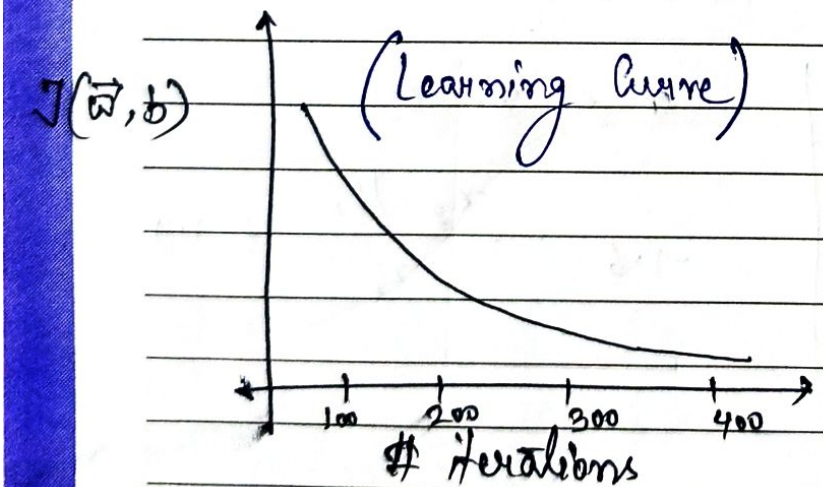
$0 \le x_1 \le 3$
$-2 \le x_2 \le 0.5$
} not need but still (not harm in rescaling)

$-100 \le x_3 \le 100$ (too large) → rescale

$-0.001 \le x_4 \le 0.001$ (too small) → rescale

Is your gradient descent converging !?

Objective: $\min_{\vec{w}, b} J(\vec{w}, b)$

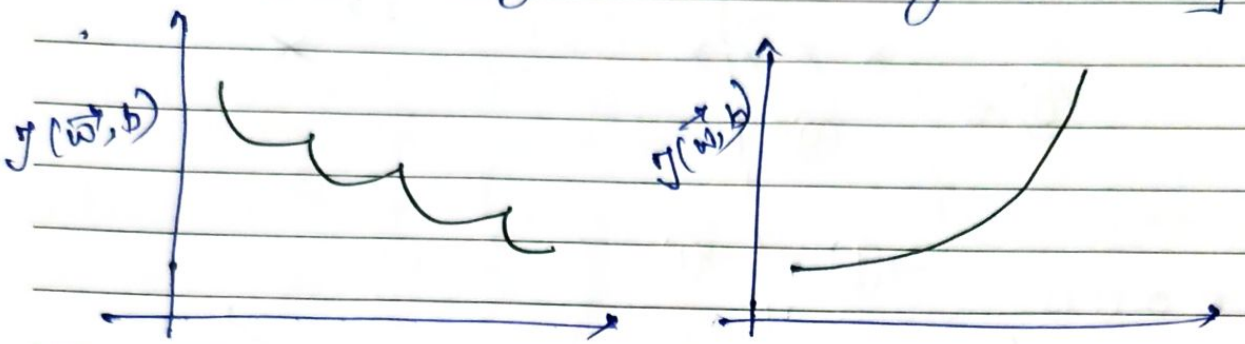


* If $J(\vec{w}, b)$ ever increases
→ i) α (learning rate) is wrongly chosen. (too large)
ii) Bug in the code.

After: Automatic convergence test. Let, ϵ = 'a' if $J(\vec{w}, b)$ decreases by $\le \epsilon$ in one iteration, declare convergence.

Choosing learning rate:

Problematic learning rate: [i) α is too large
ii) Bug in the code]



(*) Choose smaller α and that may take more time to converge but has to decrease Cost function ($J(\vec{w}, b)$) to decrease in each iterations.

Feature Engineering (making custom features)

$$f_{\vec{w}, b}(\vec{x}) = \underbrace{w_1 x_1}_{\text{length}} + \underbrace{w_2 x_2}_{\text{width}} + b$$

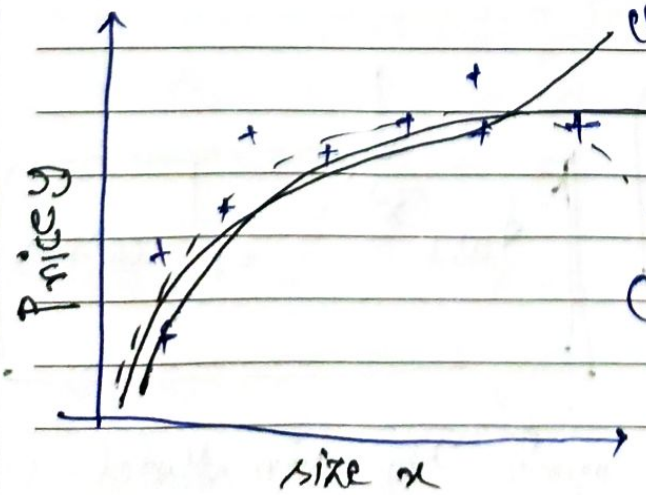
$x_3 = x_1 x_2$ (new feature)
↳ (area)

Polynomial regression

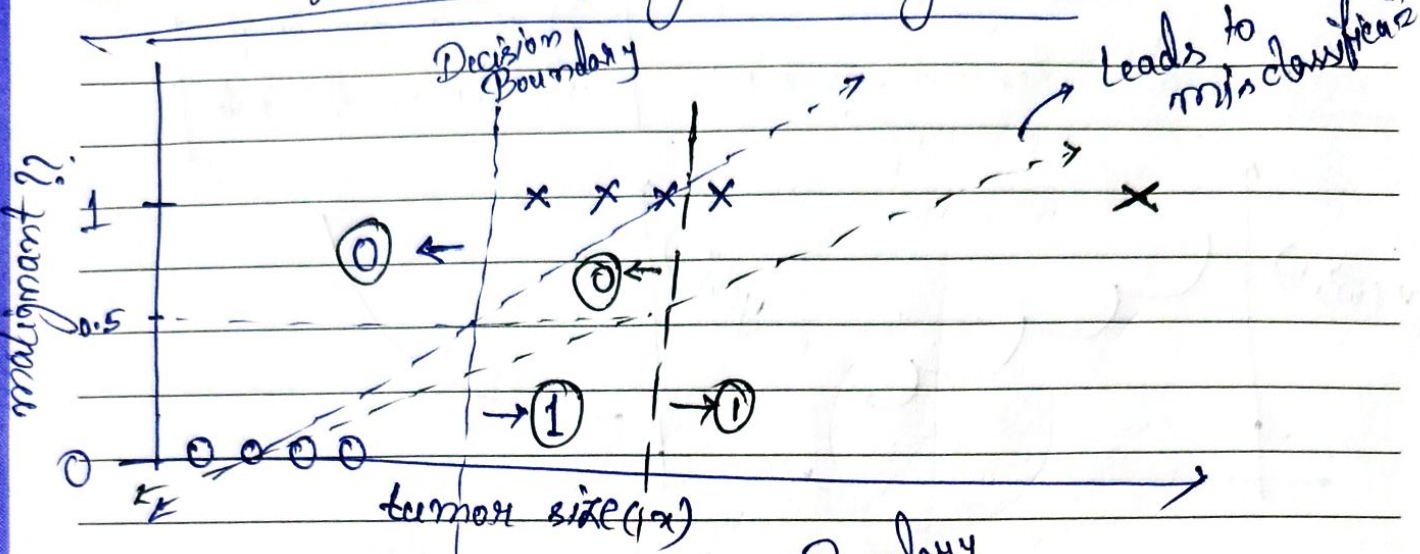
$f_{\vec{w}, b}(\vec{x}) = (w_1 x + w_2 \sqrt{x} + b)$ (cubic)

$(w_1 x + w_2 \sqrt{x} + b)$

(quadratic function)



Classification (Why not Regression)

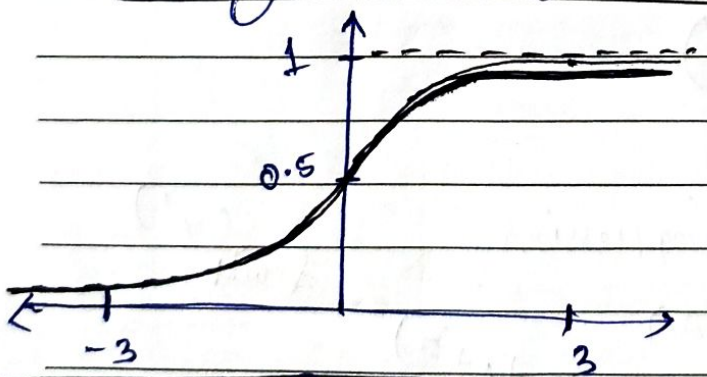


Decision Boundary (shifted)

if $f_{\vec{w},b}(\vec{x})$ $f_{w,b}(x) < 0.5$, $\hat{y} = 0$

if $f_{w,b}(x) \geq 0.5$, $\hat{y} = 1$

Logistic Regression (~~not a~~ used for classification)



$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$$

$$g(f_{\vec{w},b}(\vec{x})) = \frac{1}{1+e^{-f_{\vec{w},b}(\vec{x})}}$$

Sigmoid Function

$$g(z) = \frac{1}{1+e^{-z}}, \quad 0 < g(z) < 1$$

$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

Logistic Regression Model

Interpretation of Logit Output

$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

"Probability" that class is 1

alternatively, $f_{\vec{w},b}(\vec{x}) = P(y=1 \mid \vec{x}; \vec{w}, b)$

Probability that y is 1, given input $\vec{x}$; parameters $\vec{w}, b$

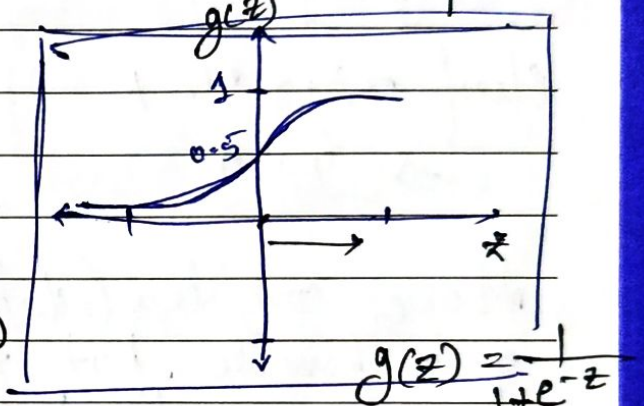
So, $f_{\vec{w},b}(\vec{x}) \geq 0.5$?

Yes: $\hat{y} = 1$

$f_{\vec{w},b}(\vec{x}) \geq 0.5$ $\quad f_{\vec{w},b}(\vec{x}) = g(z)$

$\Rightarrow z \geq 0$

$\Rightarrow \vec{w}\cdot\vec{x} + b \geq 0$



Hence, algorithm is,

if $\vec{w}\cdot\vec{x} + b \geq 0.5$: $\hat{y} = 1$

else if $\vec{w}\cdot\vec{x} + b < 0.5$: $\hat{y} = 0$

Decision Boundary

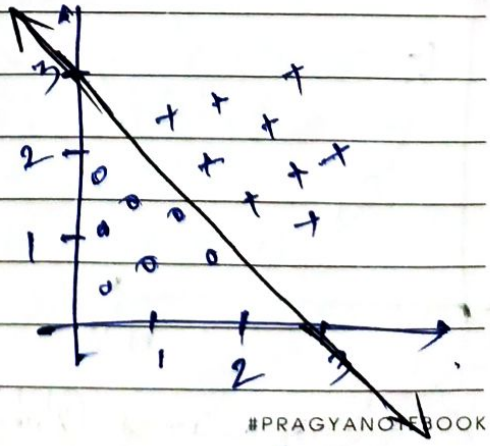
Consider, $w_1 = 1$; $w_2 = 1$, $b = -3$

$f_{\vec{w},b}(\vec{x}) = g(z) = g(w_1x_1 + w_2x_2 + b)$

Decision Boundary: $z = \vec{w}\cdot\vec{x} + b = 0$

$z = x_1 + x_2 - 3 = 0$

$\Rightarrow x_1 + x_2 = 3$

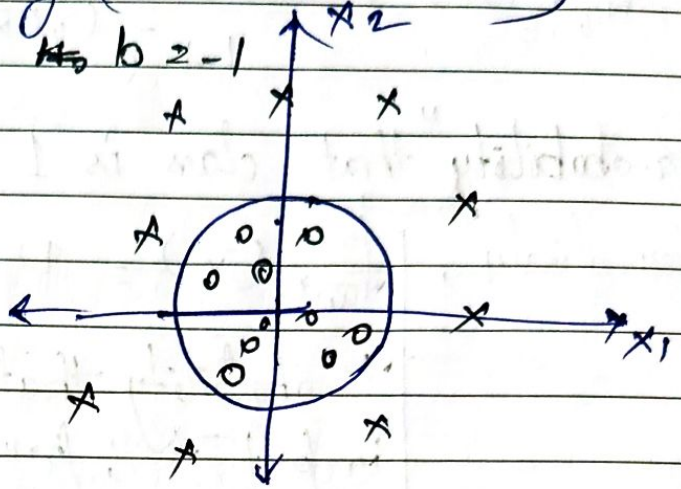


i.e., $g(z) = 0$

Non-linear Decision Boundaries:

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(w_1 x_1^2 + w_2 x_2^2 + b)$$

Consider, $w_1 = 1$, $w_2 = 1$, $b = -1$



Decision Boundary,

$$x_1^2 + x_2^2 = 1$$

if $x_1^2 + x_2^2 \geq 1$
$\Rightarrow \hat{y} = 1$

elseif $x_1^2 + x_2^2 < 1$
$\Rightarrow \hat{y} = 0$

More complex (ellips,) decision boundaries are possible. but for linear model it's decision boundary will always be linear.

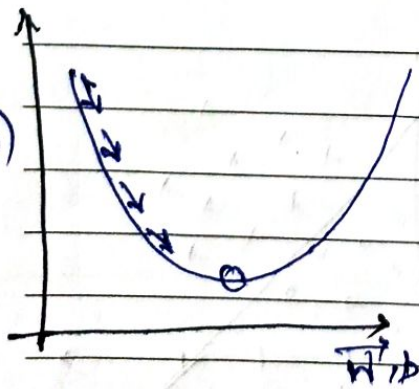
Cost Function (Why Squared Error cost function is not the best for Logit Model)

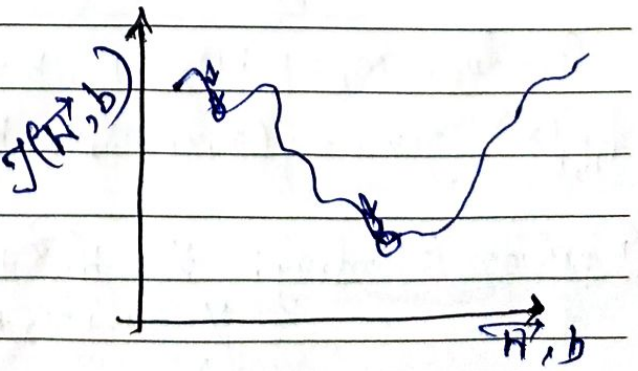
Linear Regression:

$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$$

Logistic Regression

$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$$





Convex → Global min = local min

non-convex

Cost function for Linear Regression:

$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y \right)^2$$

[where, $f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$]

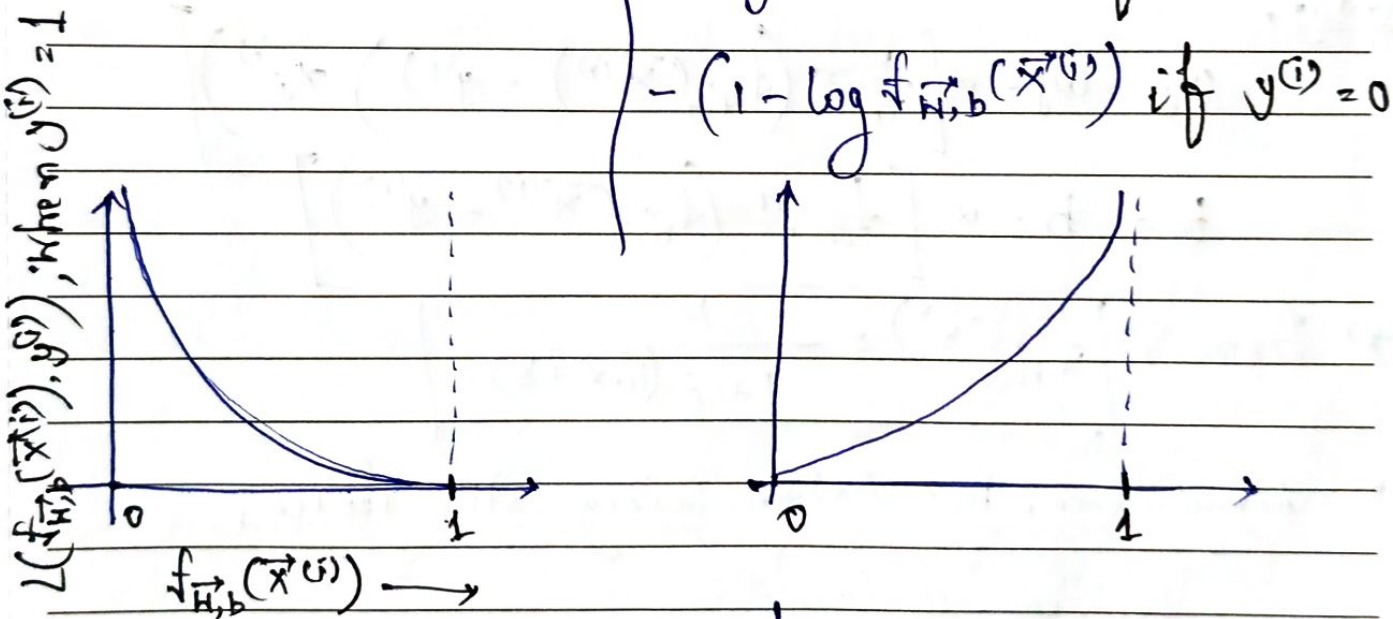
$$= \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \left(f(\vec{x}^{(i)}) - y^{(i)} \right)^2$$

$$\boxed{J(\vec{w}, b) = \frac{1}{m} \sum L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y \right)}$$

→ Called loss function

Now let's construct loss-function for logit-mode:

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)} \right) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)}) \right) & \text{if } y^{(i)} = 1 \\ -\left(1 - \log f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$



Note: $f_{\vec{w},b}(\vec{x}^{(i)}) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$

Simplified ~~Cost~~ Loss function:

$$\boxed{L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)} \right) = y^{(i)} \left(-\log\left(f_{\vec{w},b}(\vec{x}^{(i)}) \right) \right) + \left(1 - y^{(i)}\right) \left(-\left(1 - \log f_{\vec{w},b}(\vec{x}^{(i)})\right) \right)}$$

$$\text{Cost} = J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)} \right)$$

But why particularly this function??

⇒ Maximum Likelihood Estimation.

Gradient Descent

$$J(\vec{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)\right]$$

Repeat {

$$w_j = w_j - \alpha\left[\frac{\partial}{\partial w_j}J(\vec{w},b)\right] \rightarrow \left[\frac{1}{m}\sum\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right]$$

$$b = b - \alpha\left[\frac{\partial}{\partial b}J(\vec{w},b)\right] \rightarrow \left[\frac{1}{m}\sum\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)\right]$$

finally,

$$w_j = w_j - \alpha\left[\frac{1}{m}\sum\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right]$$

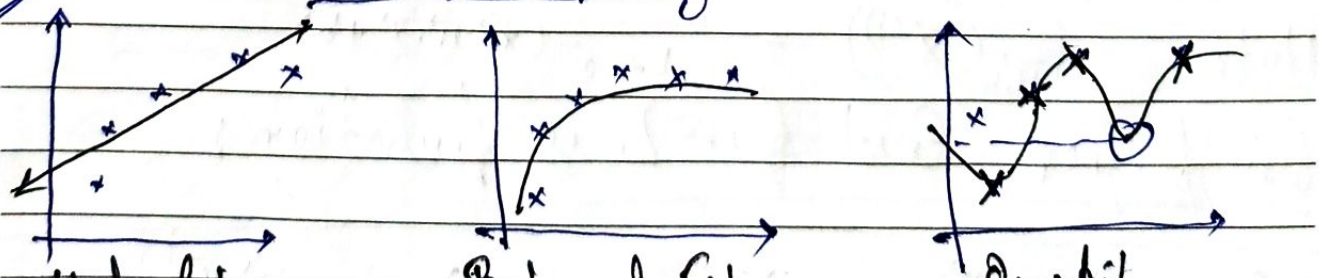
$$b = b - \alpha\left[\frac{1}{m}\sum\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)\right]$$

(*) Where, $f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$

N.B. Convergence, Vectorization, Scaling still applies.

Problem of Overfitting

Regression

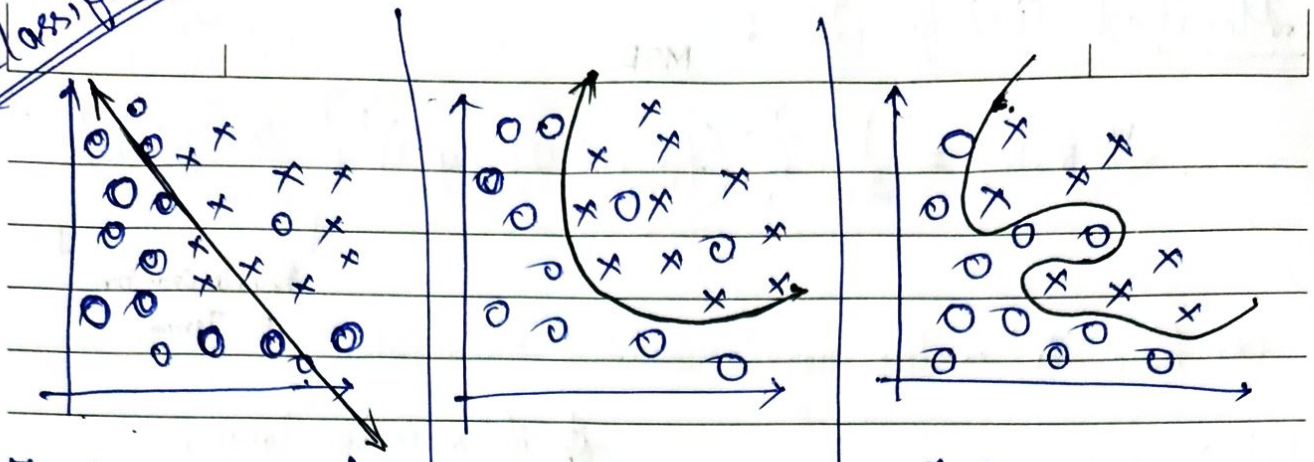


Underfit (High Bias) — $(w_1x + b)$ [= too few feature]

Balanced fit (Generalization) — $(w_1x + w_2x^2 + b)$ [= Just right]

Overfit (High Variance) [= too many feature] — $(w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b)$

Classification



$\Rightarrow z = w_1x_1 + w_2x_2 + b$

$\Rightarrow f_{\vec{w},b}(\vec{x}) = g(z)$

where,

$g(x) = \frac{1}{1+e^{-x}}$

(High Bias)

$z = w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2 + b$

(Balancefit, generalized)

$z = w_1x_1 + w_2x_2 + w_3x_1^2x_2 + w_4x_1^2x_2^2 + w_5x_1^2x_2^3 + w_6x_1^3x_2 + \ldots + b$

(

Solution

~~Regularization~~ | Solutions → (Overfitting)

i) Data Augmentation:

⇒ Use more data to fit the model.

ii) More features + Insufficient data = Overfitting

⇒ Use fewer features (Feature Selection)

iii) Regularization (regulating the impact of features)

N.B: a) Simpler models usually don't overfit.

b) Penalizing all the features usally results in a more effective model less prone to overfitting.

c) Penalizing specific features using its cost function can also be done.

Modified Cost-function:

MSE

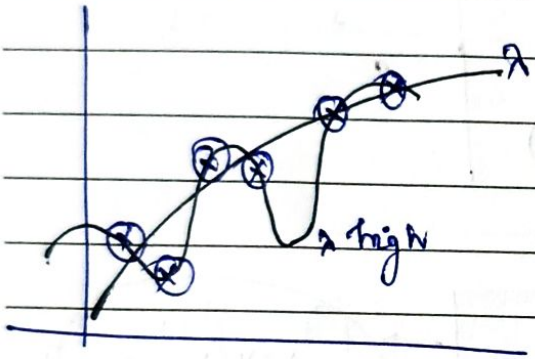
$$J(\vec{w},b) = \frac{1}{2m}\sum_{i}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \underbrace{\frac{\lambda}{2m}\sum w_j^2}_{\text{Regularization Term}}$$

(*) here 'λ' is the regularization parameter.

Note:

$\lambda \uparrow \Rightarrow \downarrow w_i$'s

(if λ is too large, model shrink to a linear one, if $\lambda \approx 0$, will result in a wiggly one having no regularization effect)



Modified Gradient Descent:

Regularized Linear Regression

derivative term:

$$\frac{\partial}{\partial w_j}\left(J(\vec{w},b)\right) = \frac{1}{m}\sum_{i}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}w_j$$

simultaneous Update:

$$w_j = w_j - \alpha\left[\frac{1}{m}\sum_{i}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right] - \alpha\left(\frac{\lambda}{m}w_j\right)$$

$$\Rightarrow \boxed{w_j = w_j\left(1 - \alpha\left(\frac{\lambda}{m}\right)\right) - \alpha\left[\frac{1}{m}\sum_{i}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right]}$$

similarly,

Note, here,

$$\boxed{f_{\vec{w},b}(\vec{x}) = \vec{w}\cdot\vec{x} + b}$$

similarly, Regularized Logistic Regression follows.....

with distinction in the functional def^{n}.

Stat-ML Playlist (codebasics)

→ Univariat Linear Regression:

(from sklearn import linear_model)

Library

```
>> lin_reg = linear_model.LinearRegression()
>> lin_reg.fit([[' ']], )
>> lin_reg.predict([[' ']])
>> lin_reg.coef_          # fitted parameter value
>> lin_reg.intercept_     #  "
```

i) Scatter plot

```
import matplotlib.pyplot as plt
>> plt.xlabel(" ")
>> plt.scatter( )
```

→ Multiple Linear Regression

```
>> df["..."]   # Creates pandas.series object.
>> df[["..."]] # creates pandas.dataframe object.
>> df.fillna(val) # fill up NA values with val given.
>> W2n.word_to_num( ) # converts words to number
>> map( fn, na_action = "ignore") # ignores na values.
```

(**) Instantiating LinearRegression object

```
>> [lin_reg = linear_model.LinearRegression( )]
```

$\hat{y}^{(i)} = wx^{(i)} + b$

↗ Algorithmic Parameters
⇒ Iteration
⇒ Learning rate

→ Gradient Descent :

```
w=b=1 , m = len(x) , ~~cost~~
def grades(x, y, iter=100, l=0.001):
    for _ in range(iter):
        yp = (w*x) + b
        W = (1/m) * np.sum((yp - y)*x)
        B = (1/m) * np.sum(yp - y)
        w = w - (l * W)
        b = b - (l * B)
```

Numpy

» np.array( )
» np.sum( )
» Broadcasting
ndarray * scalar
= ndarray

→ Saving and loading a model: (pickle / joblib / cloudpickle)

```
import pickle
with open("file_name", "wb") as f:
    pickle.dump(~~reg~~, model_name, f)
with open("file_name", 'rb') as f:
    ~~pick~~ m1 = pickle.load(f)
```

```
import joblib
joblib.dump(model, "file_n
~~joblib.load("file_na")~~
m2 = joblib.load('file_na
```

→ Dummy variable:

manual

```
» pd.get_dummies(df["col_name"], dtyp='int')
```
↳ ind. training set

```
» model.score(X, y)
```
↳ ~~dependent~~ labels

fitted model

```
» from pandas.plotting import scatter_matrix
scatter_matrix(df[["  "]], figsize=( ))
```

dummy variable

Colors	Red	Blue	Green
Red	1	0	0
Red	1	0	0
Blue	0	1	0
Green	0	0	1

To avoid multi collinearity we need to drop one of the dummy variable columns

Label encoder & OneHotEncoder

```
>> from sklearn.preprocessing import LabelEncoder
>> leben = LabelEncoder()  # Instantiation
>> leben.fit_transform(df['1'])
```

df.column ⇒ Dataframe
df.column.values ⇒ np.ndarray

```
# ColumnTransformer
from sklearn.compose
        import ColumnTransformer
ct = ColumnTransformer([("name", OneHotEncoder(0), [col_num])],
                       remainder = 'passthrough')
X = ct.fit_transform(X1)
```

→ train_test_split :

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test
        = train_test_split(X, y, test_size = 0.2,
                           random_state = value)
```

→ Logistic Regression [Prediction of Categories]

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

```
>> from sklearn.linear_models import LogisticRegression
>> model = LogisticRegression()
>> model.score(X_test, y_test)
```

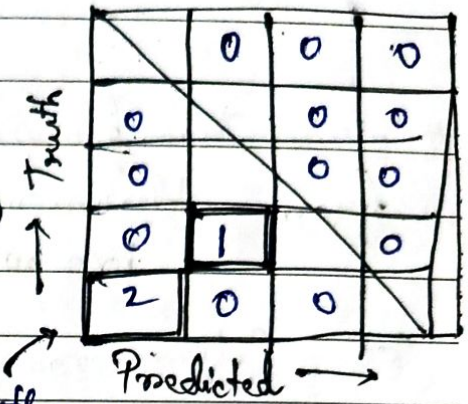
Confusion matrix :

```
* from sklearn.metrics import confusion_matrix
* cm = confusion_matrix(y_test, model.predict(X_test))
```

Visualization:

```
import matplotlib.pyplot as plt
import seaborn as sns

% matplotlib inline
plt.figure(figsize = (7,5))
sns.heatmap(cm, annot = True)
plt.xlabel("Predicted")
plt.ylabel("Truth")
```



Truth was something and predicted something else.

→ Decision Tree:

>> from sklearn.tree import DecisionTreeClassifier

(*) Only label encoding suffices, ~~One~~ Dummy variable not needed.

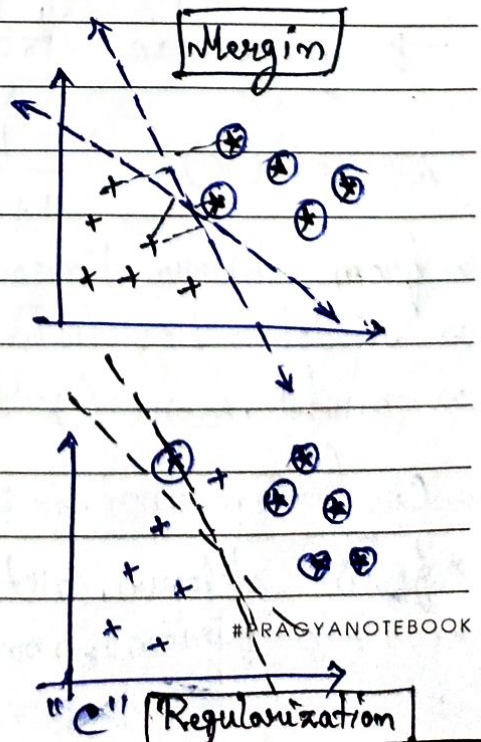
>> from sklearn.tree import DecisionTreeClassifier

→ Support Vector Machine:

>> from sklearn.svm import SVC

Instantiation through Constructor

sp = SVC(C= , kernal=' ')



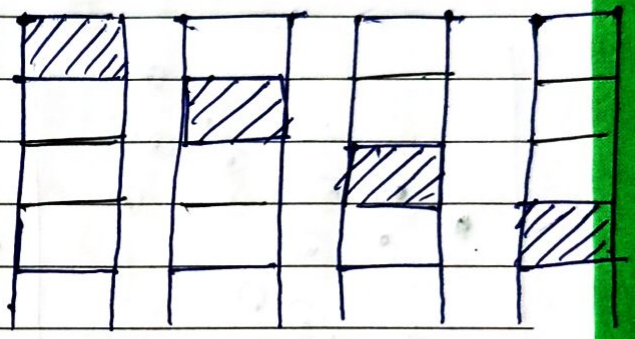
→ Random Forest (i) Generating random decision trees ii) Take majority vote

>> from sklearn.ensemble import RandomForestClassifier

(*) Based on Bagging + feture sampling

What ML model to use?

→ K-fold Cross Validation:



Parameter	Vs	Hyperparameter
i) Weights in neural network		i) Learning rate, no. of iterations in neural network.
ii) Learned from data		ii) To be set heuristically by the practitioner.
iii) intercept, coefficients of Linear Regression model.		iii) fit_intercept : bool = True, is a hyperparameter in LinearRegression Model.

Extraction of rows using index:

df.iloc[index]

Manual ~~cross-va~~ K-fold:

from sklearn.model_selection import StratifiedKFold // KFold

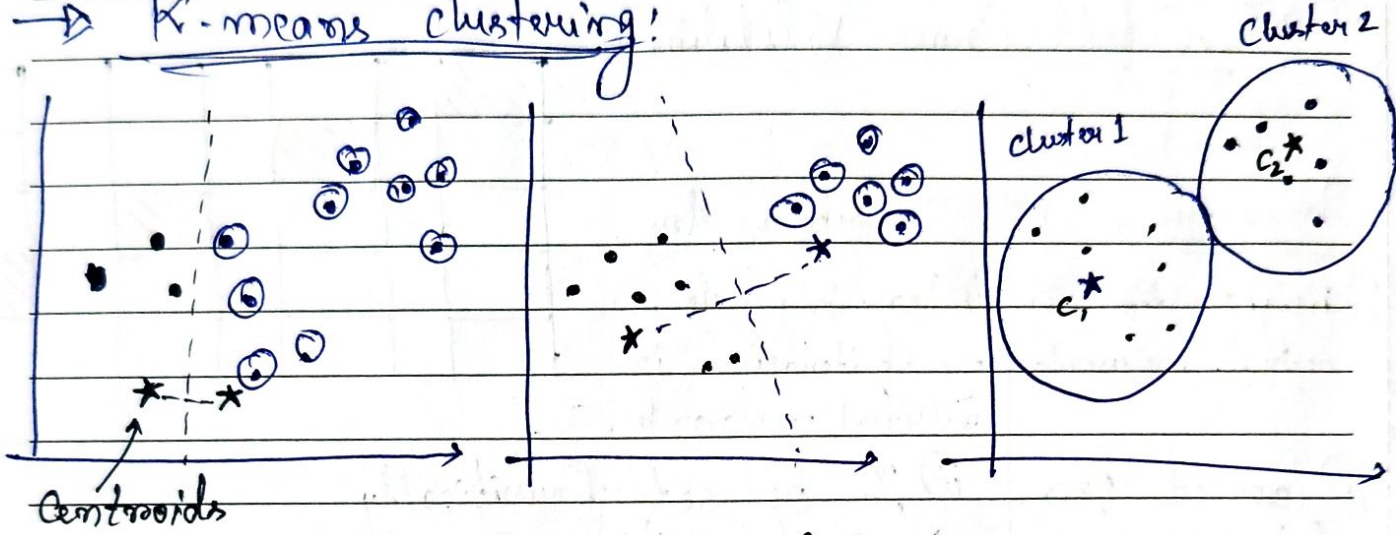
sk = StratifiedKFold(n_splits =)

for i, (train_index, test_index) in enumerate(sk(X, y)):

cross-validation direct implementation

~~from~~ from sklearn.model_selection import cross_validation_score

→ K-means clustering: (# of clusters)



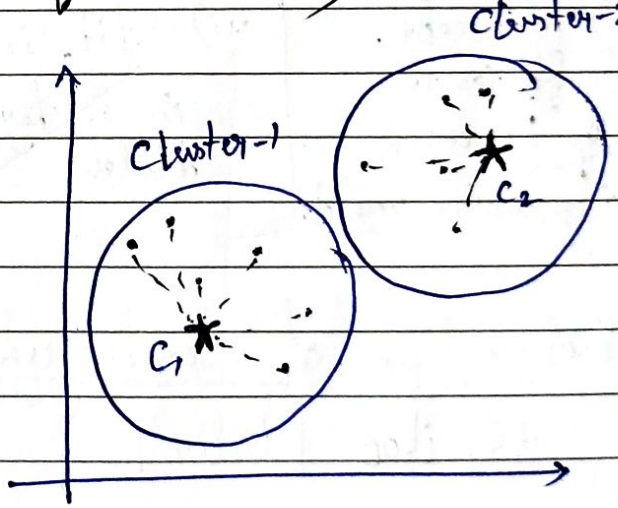
Determination of K (no. of clusters)

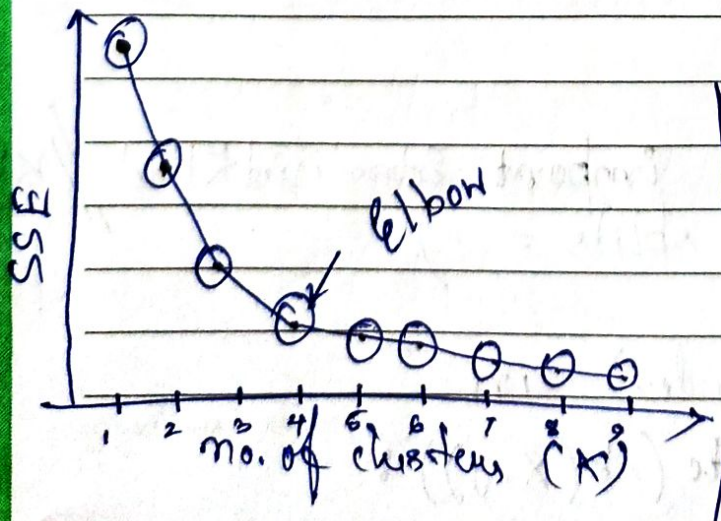
→ Elbow Method:

✓ Sum of squared errors:

$$SSE_1 = \sum_{i=1}^{n} dist(x_i - c_i)^2$$

$$SSE = \sum_{i=1}^{k} SSE_i$$





* Feature scaling matters a lot

from sklearn.preprocessing import MinMaxScaler

sc = MinMaxScaler()

~~sc.fit(df[["a","b"]])~~

~~sc.transform[["a","b"]] = sc.fit_transform~~

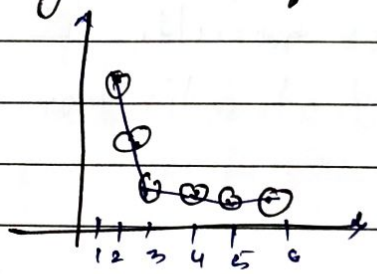
KMeans Implementation:

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters = )
df["cluster"] = km.fit_predict(df)
```

Elbow Method: (~~Scree plot~~)

```
K_range = range(1,10)
sse = []
for i in K_range:
    km = KMeans(n_clusters = i)
    km.fit(df)
    sse.append(km.inertia_)  # gives Sum of Squared Errors
```

```
import matplotlib.pyplot as plt
plt.scatter(K_range, sse)
plt.plot(K_range, sse)
```



→ Naïve Bayes:

(Continuous classes) (Discrete classes)

```
from sklearn.naive_bayes import GaussianNB / MultinomialNB /
                                BernoulliNB → (Binary class)
```

Pipeline → transforms all columns

```
from sklearn.pipeline
    import Pipeline
pipe = Pipeline(
  steps = [("s1", Estimator()),
  ....("sn", Estimator()),
  ("p/t", Predictor_Trans())]
    ↳ should have the transform() method.
```

Column - Transformer

```
from sklearn.compose
    import ColumnTransformer
final = (transformer =
  [("n1", Transformer()/Pipeline,
      attribute (column names)),
   ... ], remainder = "ignore"/"passthrough")
```

(*) transforms only specified columns.

Estimator	Transformer	Predictor
(fit() method)	(Estimator + transform() method)	(transformer + predict())

→ GridSearchCV & RandomSearchCV

```
from sklearn.model_selection import
                GridSearchCV

from sklearn.model_selection import
                RandomSearchCV
```

```
# Supress Warning
import warnings
warnings.filterwarnings("ignore")
```

Consume more time

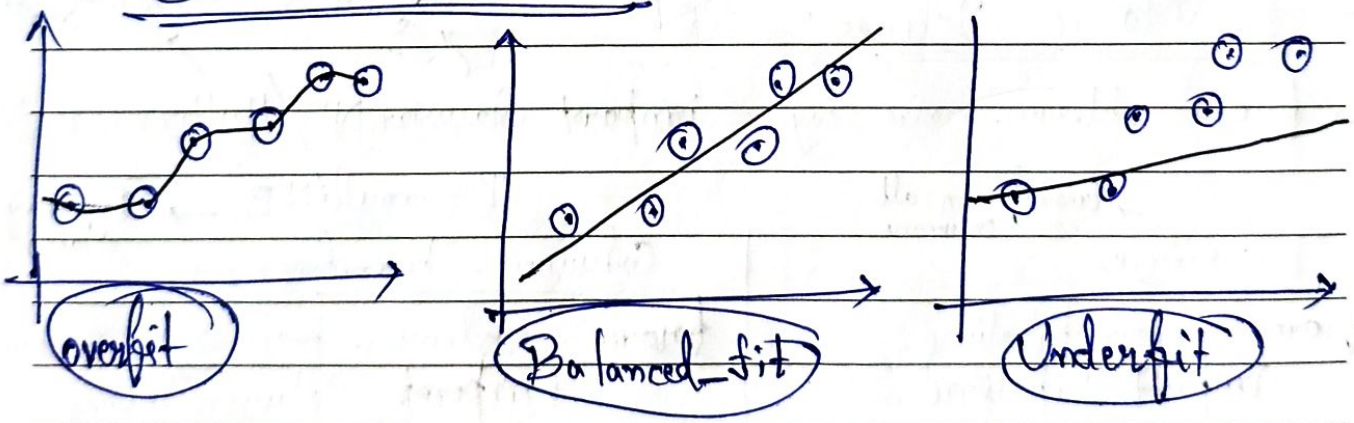
```
gcv = GridSearchCV( estimator = SVC(),
      param_grid = {'kernel': ('linear', 'rbf'),
      'C': [1, 10]}, cv = 7 }
```

```
gcv.fit(X,y)
gcv.cv_results
gcv.best_estimator
```

less time consuming / optimized

```
rgcv = RandomSearchCV
       (..(+), n_iter = 25)
```

→ Bias & variance:

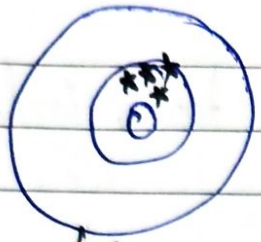


→ Bias: Train error.

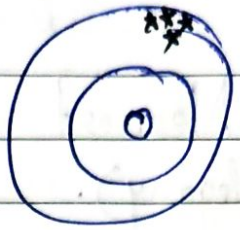
Variance: Test error due to difference in selection of "test-set".

→ Bottle eye diagram:

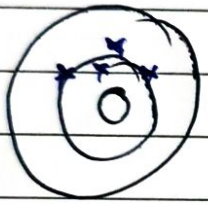
Bull's eye diagram:



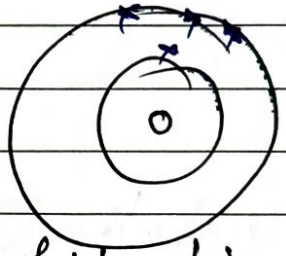
low bias and low variance



high bias and low variance



low bias and high variance



high bias and high variance

(*) Near to the inner-circle (low bias)

(*) ~~if~~ cluster more compact (lesser variance)

Way to balanced fit model:

i) K-fold Cross Validation.

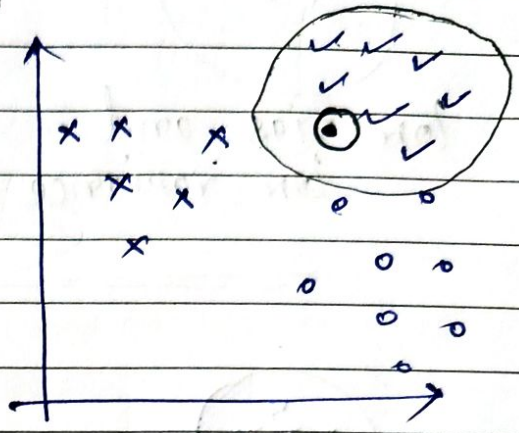
ii) Regularization.

iii) Dimentionality Reduction.

iv) Bagging and boosting.

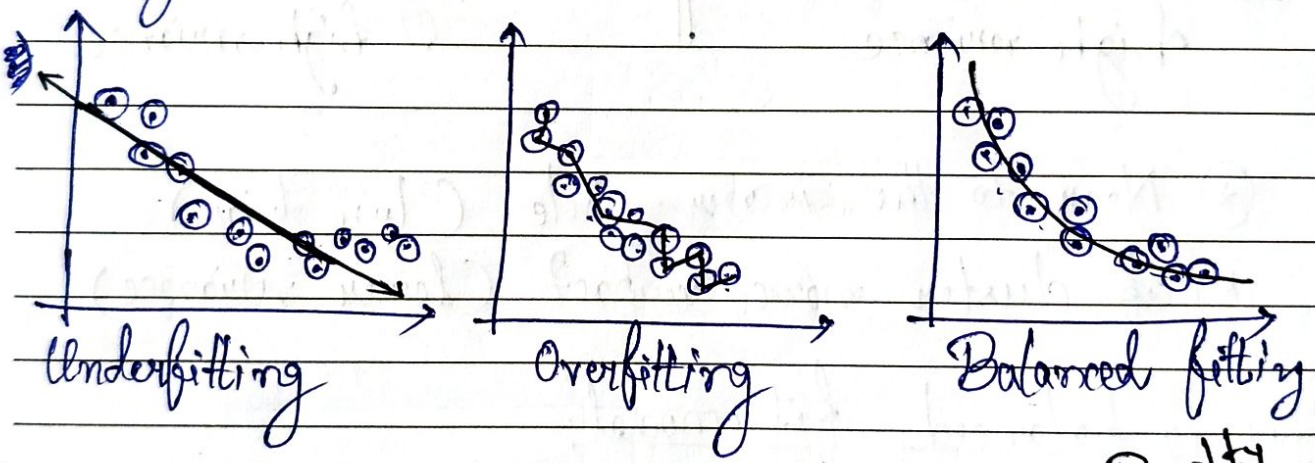
→ K-Nearest-Neighbour (KNN) → supervised classification technique

Philosophy: Nearest K-vectors of a vector, indicates its class (category).



```
from sklearn.neighbors import
    KNeighborsClassifier
```

→ Regularization



i) Ridge Regression (l_2) :

→ Cost function: $J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} \theta_i^2$ ← Penalty

↳ need to be minimized ⇒ smaller values of θ_i's are allowed.

```
>> from sklearn.linear_model import Ridge
```

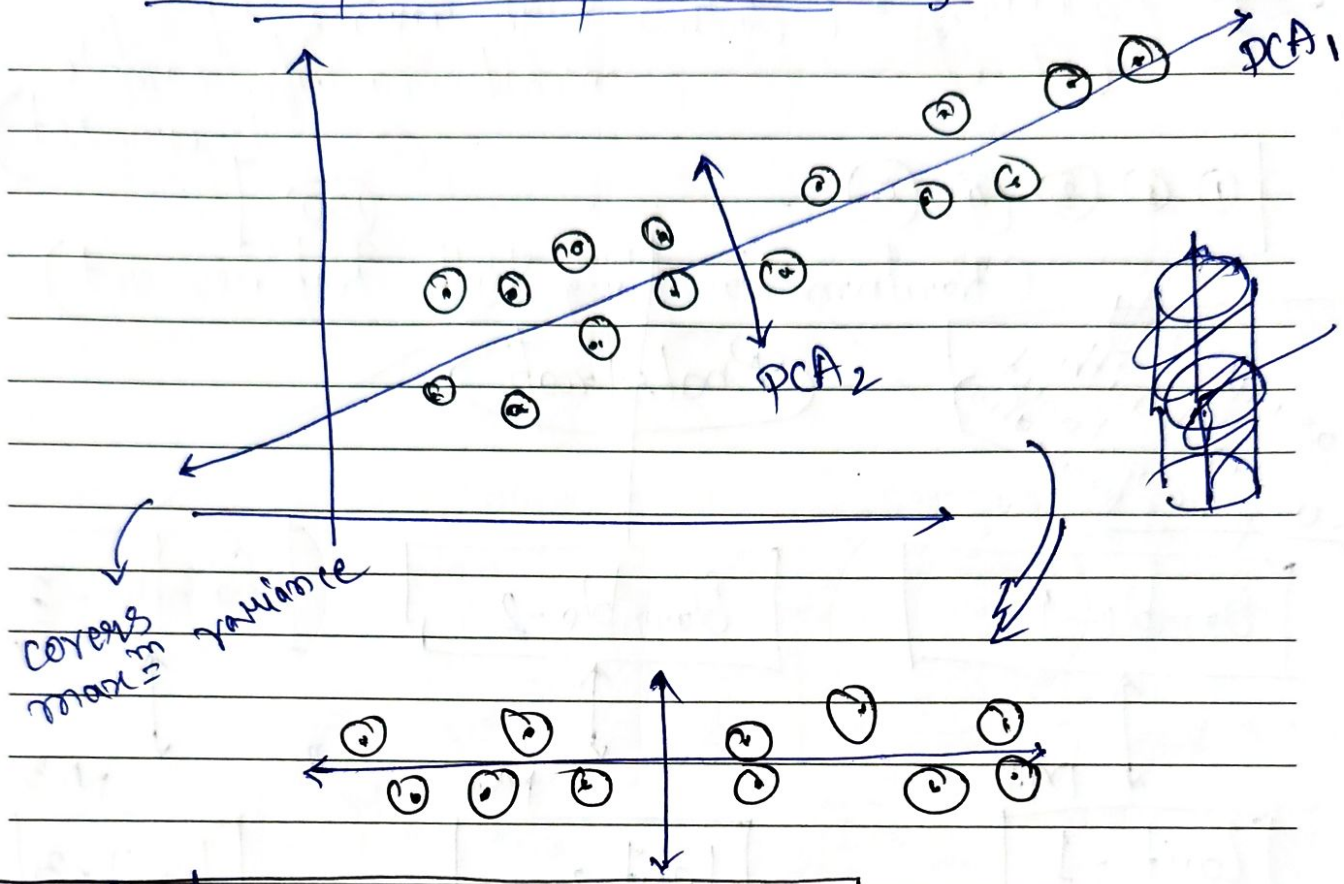
ii) Lasso Regression (l_1) (Least Absolute Shrinkage and Selection) :

→ Cost function: $J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$ ← Penalty

```
>> from sklearn.linear_model import Lasso
```

(All about variation max^{m})

→ Principal Component Analysis :

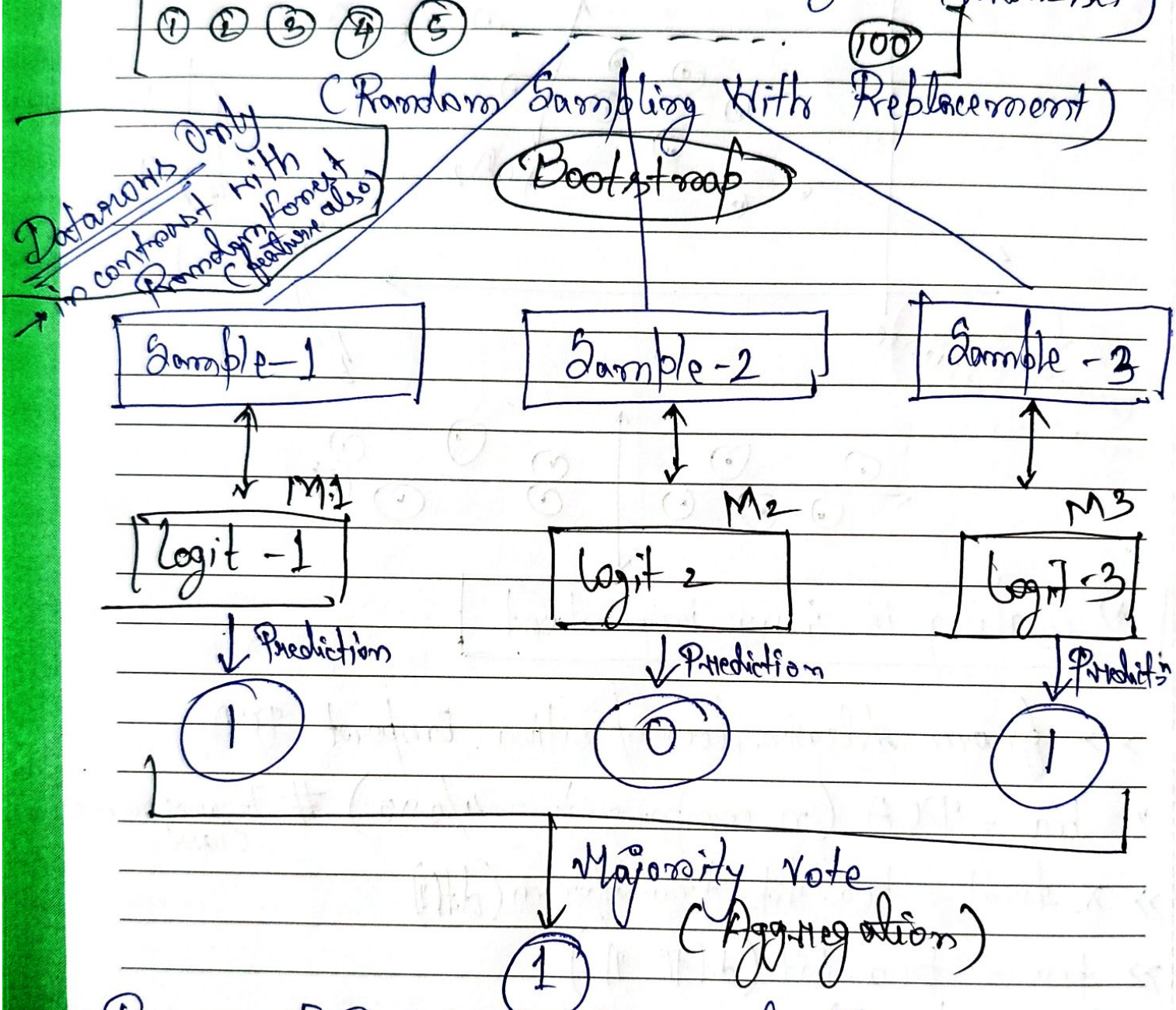


(*) Scaling is very important

```
>>> from sklearn.decomposition import PCA
>> pca = PCA(n_components = 4/0.95) # Transformer Class
>> X_final = pca.fit_transform(df[[]])
>> pca = pca.fit(df[[ ]])
>> pca.components_
>> pca.explained_variance_ratio_
```

→ Majority Vote

Bagging (Tackling high variance by considering diff. sample train_set)



∴ Bagging [Bootstrap - Aggregation]

Difference

Bagging ⇒ Underlying model can be anything (SVM, Knn etc.)

Bagged Tree ⇒ each model is a tree

>> from sklearn.ensemble import BaggingClassifier

/Pragyan

Gradient Boosting
(Sequential Stagewise Addition)

(*) $M_1 + M_2 + \cdots + M_n \equiv \Sigma_i HB$

M_1 → Weak-learner (High Bias)

↓ to get

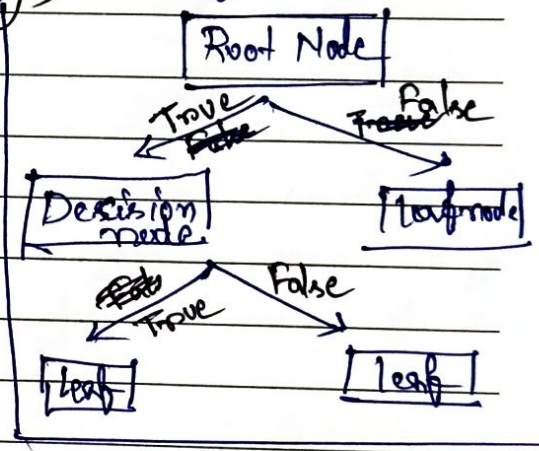
Generalized Model
(Balanced fit)

→ Basically training models (typically Decision Trees) on data having residuals as target. (till last stage)

→ **Prediction:**

$$m_1 + \alpha(m_2) + \alpha(m_3) + \cdots$$

(learning rate)



→ **Pseudo Residual:**

Actual − Predicted

(*) **Base Model**

a) Regression → Simple mean

b) Classification → log(odds) → to → Probability

sigmoid(log(odds))

(*) Ada Boost	Gradient Boost
◻ Decision stump (2 leafs)	◻ 8-32 leaf-nodes
◻ "Weighted" Addition of models	◻ Same learning-rate throughout

(*) This leads to the extreme application "XGBoost".