Logistic Regression

Algorithm

1. Initialize weights (w) and bias (b) to zero.
2. For each iteration:
   - Compute $z = wx + b$
   - Apply Sigmoid $\hat{y} = \frac{1}{1+e^{-z}}$
   - Calculate log-loss.
   - Update w, b via gradient descent
3. Stop when loss converges or max iterations reached.

Formulas

1. log-loss: $L = \left(\frac{-1}{n}\right) \sum [y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i)]$
2. Gradient: $w \leftarrow w - \eta \left(\frac{1}{n}\right) \sum (\hat{y}_i - y_i) x_i$
3. Regularization: $L + \lambda \sum w_i^2 (L_2)$

When Stop

1. loss change < tol
2. Max iter reached
3. Early Stopping if validation loss worsens

How it predicts

1. Binary:

   $\hat{y} \geq 0.5 \rightarrow$ class 1
   else 0.

2. Multi class: [Softmax]

$$P(C_k) = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

↳ Class k with max P(Ck)

Deal with Variance error

1. L2 regularization shrinks weights.

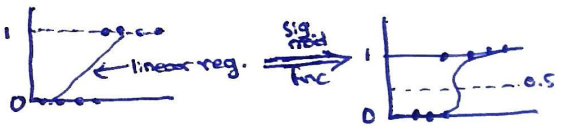
Deal with Bias error

1. optimize η (learning rate), increase iterations.

Objective: min. cost func

Code

```
from Sklearn.linear_model import
LogisticRegression
model = LogisticRegression (C=1,
        max_iter=100)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```



maximum likelihood = - Log loss

Maximum likelihood (Gradient descent)

Bonus

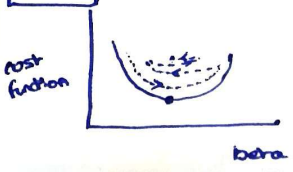


Fig: Gradient descent

Costfunction
↳ MSE ← Lasso Ridge elastic
↳ log loss ← Lasso Ridge elastic

KNN

Algorithm

1. Store training data (X, y)
2. For each test point:
   - compute distances to training points.
   - select K nearest neighbors
   - predict via majority vote [Classification]

Formulas

1. Minkowski distance

$$L_p = \left(\sum_{i=1}^{n} |A_i - B_i|^p\right)^{\frac{1}{p}}$$

When stop

1. Non iterative; stops after computing k neighbors and predicting.

How it predicts

1. compute distances to training points
2. select k nearest neighbors [k → add no.]
3. Classification: Majority vote among neighbors.

Deal with variance error and Bias error:

1. By optimal k value.

* Overfit $\xrightarrow[\uparrow k]{\text{by}}$ Good fit
* $\uparrow k$ by → underfit
* more islands $\xrightarrow[\uparrow k]{\text{by}}$ less islands
* specific $\xrightarrow[\uparrow k]{\text{by}}$ generalizⁿ.

Objective: NO explicit

Code

```
from sklearn.neighbors import
KNeighborsClassifier
model = KNeighborsClassifier(n_neigh
        bors = 5)
model.fit(Xtrain, ytrain)
ypred = model.predict(Xtest)
```

Naive Bayes (NB)

Algorithm

1. Compute class prior probability

$$P(C_k) = \frac{\text{count}(C_k)}{n}$$

$C_k \rightarrow$ class ; $n \rightarrow$ total no. samples

2. Compute conditional probabilities

$P(x_i/C_k)$ ← GNB, BNB, MNB

3. Apply Bayes theorem

$$P(C_k/x) = \frac{P(x/C_k)P(C_k)}{P(x)}$$

4. Predict the class

$$\hat{y} = \underset{C_k}{\text{argmax}}\ P(C_k/x)$$

Formula

1. GNB : $P(x_i/C_k) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}$

2. MNB : $P(x_i/C_k) = \frac{\text{count}(x_i, C_k) + \alpha}{\sum(\text{count}(x_j, C_k) + \alpha}$

$\alpha \rightarrow$ Smoothing factor

3. BNB : $P(x_i/C_k) = p^{x_i}(1-p)^{(1-x_i)}$

When to Stop

1. Non-iterative ; Stops after computing probabilities.

How it predicts

$$\hat{y} = \underset{C_k}{\text{argmax}}\ P(C_k/x)$$

Deals with Variance Error

1. probabilistic averaging reduces variances.
2. Smoothing prevents overfitting.

Deals with Bias Error

1. High bias from independence assumption.
2. Feature engineering.
3. Proper distribution (GNB, MNB, BNB) choice.

Objective: min -ve log likelihood
instead of product on multiply, work on addition.

$P(x)$ is constant & apply log in Bayes theorem

$$\log P(C_k/x) = \log(P(C_k)) + \sum \log P(x_i/C_k)$$

Code

```
from sklearn.naive_bayes
import GaussianNB / MultinomialNB / BernoulliNB

model = GaussianNB / MultinomialNB / BernoulliNB ()

model.fit(X_train, ytrain)

y_pred = model.predict (xtest)
```

Decision Tree

Algorithm

1. Start at root node
2. For each node:
   - Select feature and threshold maximizing information gain.
   - Split data into child nodes.
3. Stop at max depth or min samples or node is pure
4. Assign leaf class (majority)

Formula

1. Gini impurity (CART) $= 1 - \sum_{i=1}^{c} P_i^2$ [G]
2. Entropy $(C(4,5)) = -\sum_{i=1}^{c} P_i \log_2 P_i$
3. Info. Gain = $G_{parent} - G_{child}$ (weighted average)
   ↳ weight = no. of samples.

When to stop

1. Max depth reached.
2. Min samples split/ leaf unmet.
3. Min impurity decrease unmet.

How it predicts

1. Traverse tree from root to leaf using feature thresholds.
2. predict majority class in leaf.

Deals with variance Error and Bias Error

1. Based on prunning
   - pre prunning — e.g. min samples split
   - post prunning — e.g. min impurity decrease.

*Prunning → used for Generalizn (reduce variance error)

2. Deep tree → reduces bias error.

Objective: pure child node

Code

```
from sklearn.tree import
DecisionTreeClassifier
model = DecisionTreeClassifier
        (max_depth = 5)
model.fit (Xtrain, ytrain)
ypred = model.predict (Xtest)
```

In Splitting

(CART) → only two child nodes.

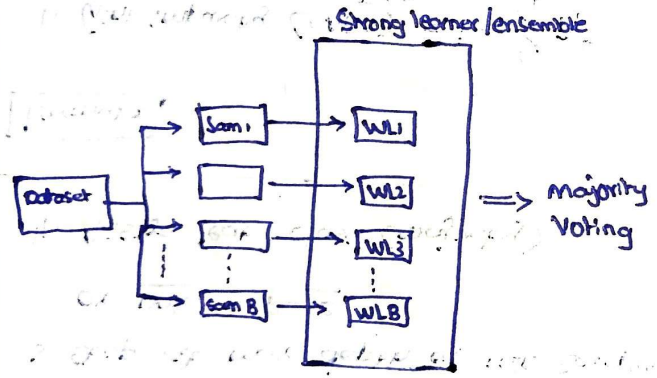
Numeric feature
↳ Sort & ² mid points of Consecutive values
⇓
to get threshold & choosen the max info. Gain threshold as info. Gain for this feature & used in future comparison with other features

Categorical feature
↳ tries different combination of categorical labels in two subgroup.

BAGGING

Algorithm

1. Generate B bootstrap samples (random sampling with replacement)
2. Train weak learner (e.g. Decision tree) on each sample.
3. Predict by majority vote across weak learners.



When to Stop

1. After n estimators/weaklearners trained.
2. Base learner criteria (e.g. tree depth)

How it predicts

1. Each tree (weaklearner/) predicts independently.
2. Majority vote across weak learners/trees

Deals with variance error

1. Designed to reduce variance errors by bootstrap samples

Deals with Bias error

1. Limited by base/weak learner capacity. (e.g. use deep trees)

Objective: Each weaklearner Objective

Code

```
from sklearn.ensemble import
BaggingClassifier
model = BaggingClassifier (n_estim
        ators = 10)
model.fit (xtrain, ytrain)
ypred = model.predict (xtest)
```

AdaBoost

Algorithm

1. Initialize sample weights $w_i = \frac{1}{n}$ for n samples.
2. For $t = 1$ to T:

- Train weak learner (e.g. shallow tree) on training data / newly created dataset.
- Amount of say of the weak learner

$$= \frac{1}{2} \log\left(\frac{1 - \text{Total error}}{\text{Total error}}\right)$$

Total error → Sum of weights of misclassified (FN, FP) i.e. weighted error

- update weight for each data point,

$$\text{new sample weight} = \text{Sample weight} \times e^{(k \times \text{Amt. of say})}$$

$k = 1$ → incorrectly classified
$k = -1$ → correctly classified

- Normalized weight
- Cumulative sum for normalized weight
- Create buckets using cum. sum for normalized weight
- Create new dataset using this bucket & $[0, 1]$

When to stop

1. After n_estimators iterations.

How to predict

1. For each class, Sum the Amt. of say of weak learners predicted so.
2. compare & get the class as max. Sum. of Amt of say.

Amt of say of $\frac{1}{\text{error}}$?

Deals with variance error

1. Shallow learners

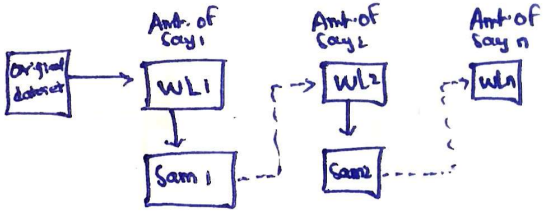
Deals with Bias error

1. Emphasis misclassified samples to reduce bias.

Objective: min. weighted error

Code

```
from sklearn.ensemble import
AdaBoostClassifier
model = AdaBoostClassifier(n_estimat
                ors = 50)
model.fit(xtrain, ytrain)
ypred = model.predict(xtest)
```



Gradient Boosting

Algorithm

1. Compute initial prediction

$$F_0 = \ln\left(\frac{\hat{y}}{1-\hat{y}}\right)$$

- $\hat{y}$ → mean of labels

2. For each Iteration (weak tree Addition)

- estimate probabilities:

$$p_i = \frac{1}{1+e^{-F_{t-1}(x_i)}}$$

- Compute residuals (errors):

$$r_i = y_i - p_i$$

- Train a Decision tree on residuals.

3. In weak tree:

- Compute leaf values:

$$h_t(x_i) = \frac{\sum r_i}{\text{no. of obs. in leaf}}$$

- update Log odds:

$$F_t(x_i) = F_{t-1}(x_i) + \eta \cdot h_t(x_i)$$

4. Final prediction

$F_T(x) = F_0 + \sum \eta\, h_t(x)$ & apply Sigmoid.

When to Stop

1. After n estimators iteration
2. Early Stopping

How it predicts

1. Sum trees, apply Sigmoid.

In weak tree

- For each node:

* pick the best feature and best split (min. residual error) i.e. max. Gain

Error = $\sum (r_i - r)^2$

Gain = $\text{Error}_{parent} - (\text{Error}_{left} + \text{Error}_{right})$

Deals with Variance error

1. Low η
2. Shallow trees

Deals with bias error

1. Sequential residual fitting reduces bias.

Code

```
from sklearn.ensemble import
GradientBoostingClassifier
model = GradientBoostingClassifier
    (n_estimators = 100, learning_rate
    = 0.1)
model.fit (Xtrain, ytrain)
ypred = model.predict (xtest)
```

XGBoost

Algorithm

1. Initialize predictions (often zero)
2. For each iteration (weak tree addition):
   - Compute residuals (errors):

$$r_i = y_i - \hat{y}_i$$

   - Compute Gradient and Hessian (for optimization):

$$g_i = \frac{\partial L}{\partial \hat{y}_i} \qquad h_i = \frac{\partial^2 L}{\partial \hat{y}_i^2}$$

   - Train a decision tree on residuals
3. In weak tree:
   - Compute leaf values:

$$w_j = -\frac{\sum g_i}{\sum h_i + \lambda}$$

   - Update predictions:

$$\hat{y}_i^t = \hat{y}_i^{t-1} + \eta \cdot w_j$$

4. Final Prediction

$$\hat{y}_i = 0 + \sum \eta w_j \quad \& \text{ apply sigmoid.}$$

When to Stop

1. After n_estimators iterations.
2. Early Stopping if validation metric stops improving

How it predicts

1. Sum trees
2. apply sigmoid

Deals with variance error

- Regularization (λ, γ) Controls complexity.

Deals with bias error

- Gradient-based fitting reduces bias.
- Deeper trees, more iterations improve fit.

Code

```
from xgboost import XGBClassifier
model = XGBClassifier(n_estimators
        =100, learning_rate=0.1)
model.fit(xtrain, ytrain).
ypred = model.predict(xtest)
```

SMOTE

Synthetic minority Over-sampling Technique

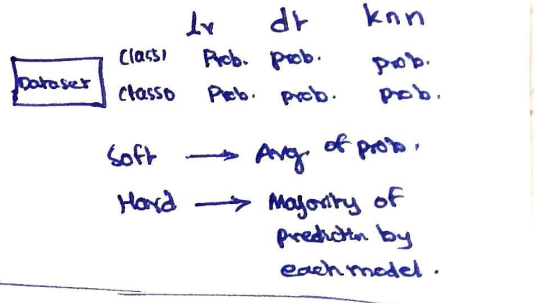
Code

```
from imblearn.over_sampling import SMOTE
Smote = SMOTE (sampling_strategy = 1,
               k_neighbors=5,
               random_state=42)
X_resampled, y_resampled = Smote.fit_
                           resample(X,y)
```

Voting Classifier

Code

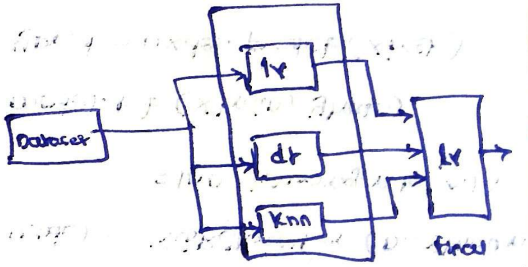
```
from sklearn.ensemble import VotingClassifier
Voting_clf = VotingClassifier ( estimators =
   [ ('lr', lr) , ('dt',dt) , ('knn',knn)],
    voting = 'soft' )

Voting_clf.fit (Xtrain, ytrain)
ypred = Voting_clf.predict(Xtest)
```



Stacking Classifier

Code

```
from sklearn.ensemble import StackingClassifier
Stacking_clf = Stacking Classifier (estimators=
   [ ('lr', lr), ('dt', dt), ('knn', knn)],
  final_estimator = LogisticRegression())
Stacking_clf.fit (Xtrain, ytrain)
ypred = Stacking_clf.predict (Xtest)
```



Heterogeneous ensembling

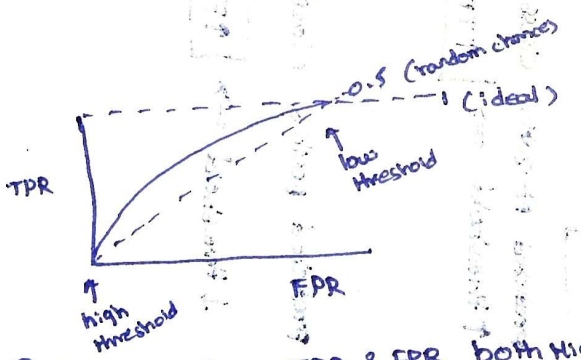
Algorithm	Regression exists?	key diff.	Code
Logistic Regression	NO	—	—
KNN	Yes	majority vote vs. mean of targets.	from Sklearn.neighbors import KNeighborsRegressor
NB	NO	—	—
Decision Tree	Yes	Gini/entropy vs. MSE/variance reduction	from Sklearn.tree import DecisionTreeRegressor
Bagging	Yes	majority vote vs. averaging predictions	from Sklearn.ensemble import BaggingRegressor
AdaBoost	Yes	Misclassification error & resampling vs. MSE & weighted errors.	from Sklearn.ensemble import AdaBoostRegressor
Gradient Boosting	Yes	log-loss & sigmoid vs. MSE & raw output	from sklearn.ensemble import GradientBoostingRegressor
XGBoost	Yes	log-loss with sigmoid vs. MSE with simplified hessians	from xgboost import XGBRegressor
Stacking	yes	Probability Stacking vs. Value Stacking; log-loss vs. MSE	from sklearn.ensemble import StackingRegressor
Voting	yes	Voting vs. Averaging predictions.	from sklearn.ensemble import VotingRegressor

		Purpose	Limitation	Interrelation
Confusion matrix	Confusion_matrix (ytrue, ypred) **Note:** P → is area of interest (typically always 1) ★ imp	Displays classification performance (TP, TN, FP, FN) e.g. Speech recognition misclassifying words.	Doesn't provide a single score.	Basis for calculating TPR, TNR, FPR, FNR, precision, accuracy, etc.
Accuracy	accuracy_score (ytrue, ypred) (TP + TN) / (TP + TN + FP + FN)	Measures overall correctness of predictions. e.g. Spam detection classifying emails correctly.	Misleading in imbalanced datasets.	1. High accuracy may mask poor performance on minority class 2. complements F1, Kappa
True positive Rate (TPR) / Recall / Recall (P) / Sensitivity	recall_score (ytrue, ypred) $TPR = \frac{TP}{(TP + FN)}$	Measures proportion of actual positives correctly predicted. e.g. Detecting disease in medical tests.	High recall may lower precision.	1. Inversely related to FNR. 2. trades off with a. precision, b. TNR.
True Negative Rate (TNR) / Recall (N) / Specificity	$TNR = \frac{TN}{(TN + FP)}$	Measures proportion of actual negatives correctly predicted. e.g. Avoiding false fraud flags in txn.	High specificity may increase false negatives.	1. Inversely related to FPR. 2. trades off with recall.

Confusion matrix layout (rows: Actual, columns: predicted):

Actual \ predicted	N	P
N	TN	FP
P	FN	TP

False positive rate (FPR)	$FPR = \frac{FP}{(FP+TN)}$	Indicates negative instances wrongly classified as positives. e.g. Spam filters marking good emails as spam	Low FPR may increase false negatives.	1 Inversely related to TNR. 2 Impacts precision.
False negative rate (FNR)	$FNR = \frac{FN}{(FN+TP)}$	Indicates positive instances wrongly classified as negatives. e.g. Cancer screening missing real cases.	Lowering FNR may raise false positives.	1 Inversely related to TPR. 2 Impacts recall.
precision /precision(P) / PPV	$precision(P) = \frac{TP}{(FP+TP)}$ precision_score (ytrue, ypred)	Measures accuracy of positive predictions. e.g. facial recognition avoiding misidentifications.	High precision may lower recall.	1 Inversely related to FPR 2 trades off with recall 3 impacts F1
Negative predictive value (NPV) /precision(N)	$NPV = \frac{TN}{(TN+FN)}$	Measures accuracy of negative predictions. e.g. ensuring healthy patients are correctly cleared in medical tests.	High NPV may miss positive cases in imbalanced datasets.	1 complements TNR. 2 Inversely related to FNR. 3 trades off with recall.

F1 score	F1_score (ytrue, ypred)	Balances 1precision & 2recall e.g. Fraud detection balancing falase alarms.	Not ideal if precision or recall is prioritized.	1 Combines precision & recall 2 useful when accuracy is misleading.
Cohen's Kappa	Cohen_Kappa_score (ytrue, ypred) $\geq 0.6 \rightarrow$ is better	Measures agreement b/w 1predicted & 2actual, beyond chance. e.g. Doctors agreeing on diagnoses.	Misleading with highly imbalanced data.	Complements accuracy by accounting for chance agreement.
ROC AUC (only in binary classification) ↳ TPR vs FPR for various thresholds.	roc_auc_score (ytrue, ypred)  @ low threshold => TPR & FPR both High (↑ +ve class prediction) @ High threshold => TPR & FPR both low (↑ -ve class prediction)	Assesses ability (of model) to distinguish classes. e.g. Predicting risk levels in insurance. Additional Class probabilities distribution by the model ⟷ effect ⟷ ROC curve	Doesn't consider threshold impact.	1 Relates to TPR, FPR across thresholds 2 less effective in imbalanced datasets.

Trade-offs in confusion matrix and metrics

Actual \ Predicted	N	P
N	TN	FP
P	FN	TP

Trade offs
- Within same class (i.e positive / Negative)
- b/w classes (i.e positive vs. Negative)

⇓ gives

tradeoff b/w Sensitivity / TPR & Specificity / TNR

Within same class

1. Actual Negative class :
   - ↑ threshold ⇒ ↑ TN ⇒ ↓ FP (trade off) ⇒ ↑ TNR, ↓ FPR
2. Actual positive class :
   - ↓ threshold ⇒ ↑ TP ⇒ ↓ FN (trade off) ⇒ ↑ TPR, ↓ FNR

B/w classes

1. predicted Negative class :
   - ↑ threshold ⇒ ↑ TN & ↑ FN ⇒ ↓ FP & ↓ TP (causing tradeoff) ⇒ ↑ TNR, ↓ TPR (main tradeoff)
2. predicted positive class :
   - ↓ threshold ⇒ ↑ FP & ↑ TP ⇒ ↓ TN & ↓ FN (causing tradeoff) ⇒ ↓ TNR, ↑ TPR (main tradeoff)

Sum up

Metric	Decreasing Threshold [↑ positive predict class]	Increasing Threshold [↑ negative prediction class]
TP	↑	↓
FN	↓	↑
FP	↑	↓
TN	↓	↑
TPR/Recall	↑	↓
TNR	↓	↑
FPR	↑	↓
FNR	↓	↑
Precision ($\frac{TP}{TP+FP}$)	↓ (often) as denominator often ↑ than numerator	↑ (often) as denominator often ↓ than numerator
ROC AUC	–	–
F1	Varies	Varies
Kappa	Varies	Varies
Accuracy	Varies	Varies

* V.V.V. Imp

Factors for metric choose

1. Cost of error (FN, FP)
2. Balance / not
3. Small dataset?
4. Stakeholder Focus
5. Threshold tuning

As ROC AUC is threshold-independent.

dataset balanced / +ve heavy / -ve heavy [other factor to consider]

How to Summarize the best model based on metrics?

Factor	Description	Metrics to prioritize	Example use case
High FN cost	Missing positives is costly.	Recall, F1	Medical diagnosis (e.g. cancer detection)
High FP cost	~~Missing~~ Incorrect positives are costly.	Precision, F1	Spam filtering
Class imbalance	one class dominates	F1, precision, recall, Kappa	Fraud detection

Factor	Description	metrics prioritized	example usecase
Balanced classes	equal class distribution	Accuracy, F1, Kappa	Sentiment analysis
Small Dataset	Limited Samples	F1, Kappa (caution)	Rare disease studies
Stakeholder Focus	Specific business needs	Align with goal (e.g. precision for customer trust)	Marketing Campaign
Threshold Tuning	Need optimal operating point for optimal tradeoff b/w precision & recall	F1, ROC AUC	Risk prediction

Based on Metric as Criteria ⟹ finally use like "weighted scoring"

Metric	When to prioritize	possible range	optimal value* (may change based on objective)
Accuracy	1 Balanced dataset 2 equal error costs (FN, FP)	[0,1]	[0.85 - 0.95] balanced data
Recall (TPR)	1 High FN cost 2 need to capture positives	[0,1]	[0.80 - 0.95]
precision	1 High FP cost 2 need accurate positives	[0,1]	[0.75 - 0.90]
F1 score	1 Imbalanced data 2 balance precision-Recall	[0,1]	[0.70 - 0.85]
Cohen's Kappa	1 Imbalanced or multi class 2 need agreement	[-1,1]	[0.6 - 0.8]
ROC AUC	1 Overall model ranking 2 Class seperation	[0.5,1]	[0.8 - 0.95]

future work:

1. ROC AUC → purpose: Measure global Seperation b/w classes.
   → use when: both classes equal important & balance dataset.
2. P-R curve AUC
   → purpose: Measure precision - recall tradeoff.
   → use when: +ve classes is concern esp. in imbalanced dataset

Sklearn.metrics

```
P, R = Precision_recall_curve (ytrue, ypred)

Pr_auc = auc (R, P)
              ↓ x axis   ↳ y axis
```