## LINEAR REGRESSION

Linear regression needs clean, well-preprocessed data, a suitable loss (cost) function such as Mean Squared Error, is commonly trained with Gradient Descent, and evaluated using metrics like MSE, RMSE, MAE, and $R^2$, each with standard formulas.

### Preprocessing for linear regression

Typical preprocessing steps for linear regression include:

Handle missing values (drop rows/columns or impute with mean/median, etc.).

- Treat outliers (remove, cap, or transform if they distort the fit).
- Encode categorical features (one-hot/dummy encoding).
- Scale/standardize features if magnitudes differ greatly (e.g., z-score standardization).
- Remove perfectly collinear or near-constant features to avoid multicollinearity issues.

### Loss function (cost) for linear regression

For a dataset $\{(x_i, y_i)\}_{i=1}^{n}$ and a linear model $\hat{y}_i = w^\top x_i + b$, the standard **loss (cost) function** is Mean Squared Error (MSE):

$$\mathbf{MSE}(\boldsymbol{w}, \boldsymbol{b}) = \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{y_i} - \hat{\boldsymbol{y}}_{\boldsymbol{i}})^{\mathbf{2}}$$

Sometimes the cost uses $\frac{1}{2n}$ for convenience in derivatives:

$$\boldsymbol{J}(\boldsymbol{w}, \boldsymbol{b}) = \frac{\mathbf{1}}{\mathbf{2}\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{y_i} - \hat{\boldsymbol{y}}_{\boldsymbol{i}})^{\mathbf{2}}$$

Other common regression losses:

- Mean Absolute Error (MAE):

$$\boldsymbol{MAE} = \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} |\boldsymbol{y_i} - \hat{\boldsymbol{y}}_{\boldsymbol{i}}|$$

- Huber loss (quadratic near zero, linear for large errors).

## Gradient Descent for linear regression

Goal: minimize the cost $J(w, b)$ by iteratively updating parameters in the negative gradient direction.[11][6]

For MSE cost $J(w, b) = \frac{1}{2n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$:

- Prediction: $\hat{y}_i = w^\top x_i + b$.
- Gradients:

$$\frac{\partial J}{\partial w} = -\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)x_i$$

$$\frac{\partial J}{\partial b} = -\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)$$

- Gradient Descent updates with learning rate $\alpha$:

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

$$b := b - \alpha\frac{\partial J}{\partial b}$$

Variants include batch GD, stochastic GD (update per sample), and mini-batch GD.[6]

## Evaluation metrics with formulas

For true values $y_i$ and predictions $\hat{y}_i$, $i = 1, \ldots, n$:

- **Mean Squared Error (MSE):**

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

- **Root Mean Squared Error (RMSE):**

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

- **Mean Absolute Error (MAE):**

$$\mathbf{MAE} = \frac{\mathbf{1}}{\boldsymbol{n}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} |\boldsymbol{y_i} - \hat{\boldsymbol{y}}_{\boldsymbol{i}}|$$

- **Coefficient of Determination $R^2$:**

Let $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. Then:

$$\mathbf{SSE} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{y_i} - \hat{\boldsymbol{y}}_{\boldsymbol{i}})^{\mathbf{2}}$$

$$\mathbf{SST} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{y_i} - \bar{\boldsymbol{y}})^{\mathbf{2}}$$

$$\boldsymbol{R^2} = \mathbf{1} - \frac{\mathbf{SSE}}{\mathbf{SST}}$$

**Logistic Regression**

Logistic Regression, Preprocessing Is Similar To Linear Regression, The Loss Is Usually Binary Cross-Entropy (Log Loss), Optimization Commonly Uses Gradient Descent, And Evaluation Relies On Classification Metrics Such As Accuracy, Precision, Recall, F1, ROC-AUC, With Standard Formulas.

Preprocessing for logistic regression

Core preprocessing steps for logistic regression:

- Handle missing values (drop or impute with mean/median/mode as suitable).
- Detect and treat outliers that can distort the decision boundary.
- Encode categorical variables (one-hot/dummy encoding for non-ordinal categories).
- Optionally scale/standardize numeric features, especially when using regularization or gradient-based solvers.
- Remove highly collinear features and ensure the target is binary (0/1) for standard binary logistic regression.

Logistic model and prediction

For input $x_i \in \mathbb{R}^d$, parameters $w, b$, the logistic regression model predicts a probability of class 1:

$$\boldsymbol{z_i = w^\top x_i + b}$$
$$\boldsymbol{\hat{p}_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}}$$

To convert probability to a class label (binary case):

$$\boldsymbol{\hat{y}_i = \begin{cases} 1 & \text{if } \hat{p}_i \geq \tau \\ 0 & \text{if } \hat{p}_i < \tau \end{cases}}$$

where $\tau$ is a threshold, often 0.5.

Loss (cost) function for logistic regression

For binary logistic regression with true labels $y_i \in \{0,1\}$ and predicted probabilities $\hat{p}_i$, the usual loss is binary cross-entropy (log loss):

Single sample loss:

$$\boldsymbol{\ell(y_i, \hat{p}_i) = -[y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]}$$

For a dataset of size $n$, cost function:

$$\boldsymbol{J(w, b) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \hat{p}_i) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]}$$

This cost is convex in $w, b$ for logistic regression.

**Gradient Descent for logistic regression**

Using the cost $J(w, b)$ above, with $\hat{p}_i = \sigma(w^\top x_i + b)$, the partial derivatives are:

$$\boldsymbol{\frac{\partial J}{\partial w} = \frac{1}{n} \sum_{i=1}^{n} (\hat{p}_i - y_i) x_i}$$

$$\boldsymbol{\frac{\partial J}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} (\hat{p}_i - y_i)}$$

Gradient Descent updates (learning rate $\alpha$):

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

$$\boldsymbol{b := b - \alpha \frac{\partial J}{\partial b}}$$

Again, there are batch, stochastic, and mini-batch variants.

Evaluation metrics and formulas

Assume binary classes 0 (negative) and 1 (positive), with:

- True Positives (TP)
- True Negatives (TN)
- False Positives (FP)

- False Negatives (FN)

Key metrics:

- Accuracy

$$\textbf{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Precision (Positive Predictive Value)

$$\textbf{Precision} = \frac{TP}{TP + FP}$$

- Recall (Sensitivity, True Positive Rate)

$$\textbf{Recall} = \frac{TP}{TP + FN}$$

- F1-score (harmonic mean of precision and recall)

$$F1 = 2 \cdot \frac{\textbf{Precision} \cdot \textbf{Recall}}{\textbf{Precision} + \textbf{Recall}}$$

- Specificity (True Negative Rate)

$$\textbf{Specificity} = \frac{TN}{TN + FP}$$

From these, the ROC curve plots True Positive Rate vs False Positive Rate, where:

$$\textbf{FPR} = \frac{FP}{FP + TN}$$

ROC-AUC is the area under this curve; PR-AUC is area under the Precision-Recall curve.

## Decision trees

Decision trees need basic preprocessing (clean labels, handle missing values, encode categories if required) but usually do not need feature scaling; they choose splits using impurity/loss measures such as Gini or entropy and are evaluated with the same classification metrics used for logistic regression (accuracy, precision, recall, F1, ROC-AUC, etc.).

## Preprocessing for decision trees

Compared to linear/logistic models, decision trees are less sensitive to monotonic transformations and scaling of features.

- Handle missing values (impute or drop; some libraries can handle a limited amount directly).
- Encode categorical features (label or one-hot encoding depending on implementation; many tree algorithms can work directly with categories).
- Detect extreme outliers if they are data errors; trees are relatively robust but bad data can still hurt.
- No strict need to normalize/standardize features, but cleaning data and removing obvious data-quality issues remains important.

## Tree prediction

For classification trees, each leaf holds a class distribution; prediction is commonly the majority class.

- For an input $x$, the tree routes down internal nodes via tests like "feature $j \leq t$?" until reaching a leaf.
- Let $p_k$ be the proportion of class $k$ among training samples in the leaf; predicted class is $\arg\max_k p_k$.

For regression trees, the prediction is typically the mean of the target values in the leaf.

**Split "loss" functions (impurity)**

At each node, the tree chooses the split that maximizes impurity reduction (information gain).

Assume a node has class proportions $p_1, \dots, p_K$. Common impurities:

- **Gini impurity**

  Gini Impurity checks how often a randomly selected sample would be mislabeled if assigned by class probability. It is computationally simple and used in tree-based classifiers.

$$G = 1 - \sum_{k=1}^{K} p_k^2$$

Where $p_k$ is the probability of class $k$.

- **Entropy**

  Entropy measures uncertainty in a node's class distribution and originates from information theory. Higher entropy indicates greater disorder among class labels.

$$H = -\sum_{k=1}^{K} p_k \log_2 p_k$$

Where $p_k$ represents the proportion of class $k$ in the node

Given a parent node $P$ split into children $C_1, \dots, C_m$ with $N_P$ total samples and $N_j$ samples in child $C_j$, the information gain (using entropy) is:

$$\text{Gain} = H(P) - \sum_{j=1}^{m} \frac{N_j}{N_P} H(C_j)$$

Analogously, Gini-based trees minimize weighted child Gini. For regression trees, a common "loss" at a node is variance or MSE:

$$\text{Var}(P) = \frac{1}{N_P} \sum_{i \in P} (y_i - \bar{y}_P)^2$$

and splits are chosen to minimize weighted child variance.

**Training / "gradient" intuition**

Standard decision trees do not use Gradient Descent; training is a greedy, top-down search for the best split at each node.

- Start at root with all data.
- For each candidate feature/threshold, compute impurity before/after split and pick the best.
- Recurse on each child until stopping criteria (max depth, min samples, no gain, etc.) are met.

**Evaluation metrics and formulas**

For classification trees, metrics are the same as for logistic regression.

Let TP, TN, FP, FN be counts from the confusion matrix:

- Accuracy: $\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$
- Precision: $\text{Precision} = \frac{TP}{TP+FP}$
- Recall (TPR): $\text{Recall} = \frac{TP}{TP+FN}$
- F1-score: $F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

ROC-AUC, PR-AUC and other metrics are defined exactly as for logistic regression, using the tree's class probabilities (or scores) across thresholds.

For regression trees, use MSE, RMSE, MAE, and $R^2$ with the same formulas as in linear regression.

## Random Forests

Random Forests use similar basic preprocessing to decision trees, train many trees on bootstrapped and feature-sampled data, aggregate predictions by averaging (regression) or majority vote (classification), and are typically evaluated with regression or classification metrics depending on the task.

## Preprocessing for Random Forests

For tabular data, Random Forests are quite robust and do not need feature scaling.

- Handle missing values: use imputers such as KNN Imputer or simple mean/median; some implementations can handle limited missingness.
- Encode categorical features (label or one-hot encoding) before fitting in libraries like scikit-learn.
- Clean obvious data errors and consider handling severe class imbalance (class weights, resampling) for classification tasks.

## Model, training, and "loss"

Random Forests are ensembles of decision trees, trained with randomness in both data and features.

- Bootstrap sampling: each tree is trained on a bootstrap sample (sampling rows with replacement) from the original dataset.
- Feature sampling: at each split, a random subset of features is considered, promoting diversity among trees.
- Tree-level split criteria:
    - Classification: use Gini impurity or entropy, as in standard decision trees.
    - Regression: minimize MSE / variance within nodes.

For regression, tree $t$ outputs prediction $\hat{y}^{(t)}(x)$; Random Forest prediction is:

$$\hat{y}(x) = \frac{1}{T}\sum_{t=1}^{T} \hat{y}^{(t)}\,(x)$$

For classification with $K$ classes, if tree $t$ predicts class $\hat{c}^{(t)}(x)$, the forest prediction is the majority vote:

$$\hat{c}(x) = \arg \max_{k \in \{1,\dots,K\}} \sum_{t=1}^{T} \mathbf{1}\{\hat{c}^{(t)}(x) = k\}$$

No Gradient Descent is used; training is greedy at each tree, like decision trees.

**Out-of-bag (OOB) estimate**

Because each tree sees only a bootstrap sample, some rows are left out (OOB samples) for that tree.

- For each data point, aggregate predictions from trees where it was OOB.
- Compute an OOB performance metric (e.g., $R^2$ for regression, accuracy for classification).

This gives an internal validation estimate without a separate validation set.

**Evaluation metrics (regression forests)**

For Random Forest regression, use the same metrics as linear regression.

Let true values $y_i$ and predictions $\hat{y}_i$, $i = 1, \dots, n$:

- MSE: $\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
- RMSE: $\text{RMSE} = \sqrt{\text{MSE}}$
- MAE: $\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} | y_i - \hat{y}_i |$
- $R^2$: with $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$,

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

**KNN model**

KNN "trains" by storing the labeled training data.

Given a query point $x$:

1. Compute distance from $x$ to every training point $x_i$, often Euclidean:

$$d(x, x_i) = \sqrt{\sum_{j=1}^{d} (x_j - x_{ij})^2}$$

2. Select the $k$ nearest neighbors $\mathcal{N}_k(x)$ with smallest distances.

- Classification: majority vote among neighbors' labels $y_i$:

$$\hat{y} = \arg\max_c \sum_{i \in \mathcal{N}_k(x)} \mathbf{1}\{y_i = c\}$$

Optionally weight votes by inverse distance.

- Regression: average neighbors' targets $y_i$:

$$\hat{y} = \frac{1}{k} \sum_{i \in \mathcal{N}_k(x)} y_i$$

or a distance-weighted average.

**“Loss” / optimization perspective**

KNN does not learn parameters with Gradient Descent; there is no global cost function being minimized during training.

- The main “hyperparameters” are $k$, distance metric (Euclidean, Manhattan, etc.), and weighting scheme.
- $k$ is usually chosen via validation or cross-validation to balance overfitting (small $k$) vs underfitting (large $k$).

**Evaluation metrics for KNN classification**

Use the same metrics as logistic regression / decision tree / Random Forest classification.

With TP, TN, FP, FN:

- Accuracy: Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$
- Precision: Precision $= \frac{TP}{TP+FP}$
- Recall: Recall $= \frac{TP}{TP+FN}$
- F1-score: $F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision}+\text{Recall}}$

Using predicted class probabilities (fraction of neighbors in each class) across thresholds, compute ROC-AUC and PR-AUC as usual.

**Evaluation metrics for KNN regression**

For regression KNN, use the same regression metrics as linear regression / Random Forest regression.

With true targets $y_i$, predictions $\hat{y}_i$:

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

- $R^2$: $R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$, with $\bar{y}$ the mean of $y_i$.

**Evaluation metrics (classification forests)**

For Random Forest classification, use the same metrics as logistic regression and decision trees.

With TP, TN, FP, FN:

- Accuracy: Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$
- Precision: Precision $= \frac{TP}{TP+FP}$
- Recall: Recall $= \frac{TP}{TP+FN}$
- F1-score: $F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

Using predicted class probabilities across thresholds, ROC-AUC and PR-AUC are defined as for any classifier.

## Gradient Boosting

**Gradient Boosting** builds an ensemble of weak learners (usually shallow trees) sequentially, each new model fitting the negative gradient of a chosen loss (e.g., MSE for regression, log-loss for classification) and is evaluated with the usual regression or classification metrics.

### Preprocessing for Gradient Boosting

Gradient Boosted Trees (GBMs like XGBoost, LightGBM) have similar preprocessing needs to decision trees and Random Forests.

- Handle missing values; many implementations can natively route missing values but basic cleaning is still recommended.
- Encode categorical variables when required (some libraries need one-hot; others like CatBoost handle categories directly).
- Feature scaling is usually not required for tree-based GBMs, though it can help if using non-tree base learners.
- Remove obvious data errors and consider strategies for imbalance (class weights, sampling) for classification.

### Model and prediction

Gradient Boosting builds a strong model $F_M(x)$ as a sum of $M$ weak learners $h_m(x)$, often small decision trees.

- Start with an initial model, e.g.

$$F_0(x) = \arg\min_{\gamma} \sum_i L(y_i, \gamma)$$

- At each iteration $m = 1, \ldots, M$:
    - Compute pseudo-residuals (negative gradient of the loss):

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$

- Fit a weak learner $h_m(x)$ to $r_{im}$.
- Find optimal step size $\rho_m$ (line search).
- Update model:

$$F_m(x) = F_{m-1}(x) + \nu\rho_m h_m(x)$$

text

where \(\nu\) is the learning rate (shrinkage).

- Regression prediction: $\hat{y}(x) = F_M(x)$.
- Classification prediction: use $F_M(x)$ as a score or log-odds, convert to probability via a link function (e.g., logistic), then apply a threshold.

**Loss functions ("cost")**

The key idea is: each new tree fits the negative gradient of the chosen differentiable loss function.

Common choices:

- Regression:
    - Squared error (MSE):

$$L(y, F(x)) = \frac{1}{2}(y - F(x))^2$$

- Absolute error (MAE) or Huber loss for robustness to outliers.
- Binary classification:
    - Logistic (cross-entropy) loss:

$$L(y, F(x)) = \log\left(1 + e^{-yF(x)}\right), y \in \{-1, +1\}$$

Multiclass classification:

- Multinomial log-loss (softmax cross-entropy).

The algorithm is effectively doing Gradient Descent in function space on these losses.

**Evaluation metrics (regression GBM)**

For gradient boosting regression, use the standard regression metrics as before.

Given true targets $y_i$ and predictions $\hat{y}_i$:

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

Some libraries also report “training deviance”, which is essentially the average loss.

**Evaluation metrics (classification GBM)**

For gradient boosting classification, use the same metrics as logistic regression / Random Forest classification.

With TP, TN, FP, FN:

- Accuracy: $\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$
- Precision: $\text{Precision} = \frac{TP}{TP+FP}$
- Recall: $\text{Recall} = \frac{TP}{TP+FN}$
- F1-score: $F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

From predicted probabilities, compute ROC-AUC and PR-AUC over thresholds. Many GBM frameworks also report log-loss directly, since that is the optimized objective in classification.

**Naive Bayes**

Naive Bayes is a probabilistic classifier that assumes features are conditionally independent given the class; it needs relatively light preprocessing, uses Bayes' theorem to compute posterior probabilities, and is evaluated with the same classification metrics as other classifiers (accuracy, precision, recall, F1, etc.).

**Preprocessing for Naive Bayes**

Preprocessing depends on data type (numeric vs text) and variant (Gaussian, Multinomial, Bernoulli).

- Handle missing values (impute or drop), since probability estimates require complete features.
- For numeric features (Gaussian NB), optional scaling/standardization can help but is not strictly required.
- For categorical/text features (Multinomial/Bernoulli NB):
    - Clean text (lowercasing, removing punctuation, tokenization).
    - Convert to counts or binary indicators (e.g., bag-of-words, TF-IDF).
- Remove or merge highly correlated / duplicate features when possible, since these violate independence and can distort probabilities.

**Naive Bayes model and formula**

Naive Bayes applies Bayes' theorem with the naive independence assumption.

Bayes' theorem for class $y$ and features $x = (x_1, \dots, x_n)$:

$$P(y \mid x) = \frac{P(y)\, P(x \mid y)}{P(x)}$$

With conditional independence:

$$P(x \mid y) = \prod_{i=1}^{n} P(x_i \mid y)$$

So, up to a normalization constant:

$$P(y \mid x) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

Classifier prediction is:

$$\hat{y} = \arg \max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

In practice, log-probabilities are used to avoid underflow:

$$\hat{y} = \arg \max_{y} \left[ \log P(y) + \sum_{i=1}^{n} \log P(x_i \mid y) \right]$$

Different variants model $P(x_i \mid y)$ differently (Gaussian for continuous, Multinomial for counts, Bernoulli for binary).

**"Loss" / optimization view**

Naive Bayes typically does not use Gradient Descent; parameters are estimated by maximum likelihood via simple counting/averaging.

- Class prior:

$$P(y = c) = \frac{\text{number of samples with class } c}{\text{total samples}}$$

- Conditional probabilities:
    - Categorical/count features: smoothed relative frequencies (e.g., Laplace/add-one smoothing).
    - Gaussian: estimate mean $\mu_{c,i}$ and variance $\sigma_{c,i}^2$ of feature $i$ within class $c$, then use a normal density.

The implicit loss minimized by the maximum likelihood estimate under the model corresponds to the negative log-likelihood (cross-entropy) of the training data, but this is solved in closed form rather than with iterative gradient updates.

**Evaluation metrics and formulas**

Naive Bayes is almost always used for classification, so the standard classification metrics apply.

With:

- TP: true positives
- TN: true negatives
- FP: false positives
- FN: false negatives

Key metrics:

- Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Precision

$$\text{Precision} = \frac{TP}{TP + FP}$$

- Recall (Sensitivity, True Positive Rate)

$$\text{Recall} = \frac{TP}{TP + FN}$$

- F1-score (harmonic mean of precision and recall)

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

For probabilistic outputs from Naive Bayes, ROC-AUC and PR-AUC are computed over varying decision thresholds exactly as for logistic regression or other classifiers