ChatGPT

Week 2: Model Building and Evaluation Metrics

1. Model Building

In this section, we cover fundamental machine learning models, their theoretical underpinnings, mathematical formulas, and practical implementation in Python (using scikit-learn). We focus on both **regression** (predicting continuous numeric targets) and **classification** (predicting categorical labels) algorithms, including linear and logistic regression, decision trees, random forests, support vector machines, and a brief introduction to boosting (with **XGBoost** as an example). Each subsection provides a clear explanation, key formulas, and a code snippet or exercise to illustrate usage.

Linear Regression

Concept: Linear regression is a simple yet powerful supervised learning algorithm for regression tasks. It assumes a linear relationship between the input features (independent variables) and the target output (dependent variable). The model predicts targets by fitting a straight line (or hyperplane in higher dimensions) that best represents the relationship in the training data [1]. In mathematical terms, the prediction $\hat{y}$ is expressed as a weighted sum of the features plus an intercept term:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_p x_p,$$

where $x_1, \dots, x_p$ are feature values, $w_1, \dots, w_p$ are the learned coefficients (weights) for each feature, and w_0 is the intercept (bias) [1]. Linear regression finds the *optimal* weights $w = (w_1,\dots,w_p)$ by minimizing the **Residual Sum of Squares (RSS)** between the observed targets and the predictions [2] [3]. This is the **Ordinary Least Squares (OLS)** criterion, often written as minimizing the mean squared error of predictions:

$$\min_{w_0, w_1, \dots, w_p} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2,$$

where y_i is the true value and $\hat{y_i}$ is the prediction for sample i. The closed-form solution (normal equation) for OLS is $w = (X^T X)^{-1} X^T y$ under assumptions of $X^T X$ being invertible, but in practice scikit-learn's `LinearRegression` uses an optimized solver. LinearRegression in scikit-learn "fits a linear model with coefficients $w = (w_1,\dots,w_p)$ to minimize the residual sum of squares between the observed targets and the targets predicted by the linear approximation" [2].

Assumptions: Linear regression works best when the true relationship is linear and the residual errors are uncorrelated and normally distributed with constant variance (homoscedasticity). Violations (e.g. strong non-linearity or correlated residuals) can reduce predictive power.

Training: The model can be fit by analytic solution or iterative methods (like gradient descent). Scikit-learn's implementation finds the best-fit line automatically. No hyperparameters need tuning in basic OLS linear regression, though variants like **Ridge** and **Lasso** introduce regularization.

Python Example: Below is a simple example of fitting a linear regression model on a small synthetic dataset. We use scikit-learn's `LinearRegression` estimator. After fitting, we examine the learned coefficients and intercept. We also demonstrate making a prediction and evaluating the fit with an $R^2$ score.

```
import numpy as np
from sklearn.linear_model import LinearRegression
# Example data (simple linear relation y ≈ 2*x)
X = np.array([[1], [2], [3], [4], [5]], dtype=float)  # feature matrix
(n_samples x n_features)
y = np.array([2.3, 4.1, 6.0, 8.2, 10.1])             # target vector

model = LinearRegression()
model.fit(X, y)  # train the model on the data

print("Coefficient (w1):", model.coef_[0])
print("Intercept (w0):", model.intercept_)
print("R^2 score:", model.score(X, y))
```

When you run this code, you should find the coefficient is around 2 (reflecting the linear relationship $y \approx 2x$) and a small intercept (close to 0). The $R^2$ score will be very close to 1.0 on this toy data (indicating an almost perfect fit). In practice, you'd train on a training set and evaluate on a separate test set.

Exercise: As a practice, generate a larger synthetic dataset using `sklearn.datasets.make_regression` and fit a LinearRegression model. Check the learned coefficients against the true coefficients used to generate the data. This will help solidify understanding of how linear regression captures linear patterns.

Logistic Regression

Concept: Despite its name, *logistic regression* is used for **classification**, not regression. It is a linear model for binary (and multiclass) classification that predicts the probability of a sample belonging to a class using the logistic (sigmoid) function. Logistic regression models the **log-odds** of the positive class as a linear combination of features. The model output is a probability $\hat{p}(X)$ that the given input $X$ belongs to the positive class (usually denoted class 1). This probability is given by the sigmoid of a linear function [4]:

$$\hat{p}(X) = \text{sigmoid}(w^T X + w_0) = \frac{1}{1 + \exp(-(w^T X + w_0))},$$

where $w$ are the feature weights and $w_0$ is the intercept (bias). The predicted class label is then determined by applying a threshold to $\hat{p}$, typically 0.5: if $\hat{p} \ge 0.5$, predict class 1, otherwise class 0.

LogisticRegression in scikit-learn is implemented as a linear classifier using the logistic sigmoid function [5]. It is also known as **logit regression** or **maximum entropy classifier** [5]. Under the hood, logistic regression is usually trained by maximizing the **likelihood** of the data (or equivalently minimizing the **logistic loss** / **cross-entropy loss**). For binary classification, the loss for a single sample is:

$$\mathcal{L}_{\text{logistic}} = -\Big[y \log(\hat{p}(X)) + (1-y)\log(1-\hat{p}(X))\Big],$$

which is derived from the Bernoulli negative log-likelihood [6]. The training process adjusts $w$ to minimize the average of this loss (plus regularization, if applied). Logistic regression can be regularized (e.g. L2 regularization is common by default in scikit-learn) to prevent overfitting.

Multiclass Extension: For multiclass classification, logistic regression can be extended via a *softmax* (multinomial logistic regression) or using one-vs-rest strategies. Scikit-learn's `LogisticRegression` supports both ("multinomial" option for softmax or default one-vs-rest). In multinomial logistic regression with $K$ classes, the model learns $K$ weight vectors (or equivalently one weight matrix) and uses a softmax function to output $K$ class probabilities.

Python Example: Below, we illustrate logistic regression on a simple binary classification problem (e.g., predicting whether a point is above or below a line). We'll create a small dataset, fit a logistic regression, and then output the predicted probabilities and classes.

```
from sklearn.linear_model import LogisticRegression

# Toy dataset: points (x) and a binary label (1 if y > 5, else 0)
X = np.array([[1], [2], [3], [4], [5], [6], [7]], dtype=float)
y = np.array([0,   0,   0,   0,   1,   1,   1])  # 0 for y<=5, 1 for y>5 for
illustration

clf = LogisticRegression(solver='lbfgs', penalty='none')  # no regularization
for clarity
clf.fit(X, y)

# Predict probabilities for a new set of points
X_new = np.array([[4.5], [5.5], [6.5]], dtype=float)
probs = clf.predict_proba(X_new)
preds = clf.predict(X_new)
print("Predicted probabilities (class0, class1):\n", probs)
print("Predicted classes:", preds)
```

Here `predict_proba` will output two columns: probability of class 0 and class 1 for each input. You should see the model output a higher probability for class 1 as the input value increases. For instance, at $X=4.5$ (below the threshold 5) the model likely predicts class 0 (low probability of class 1), whereas at $X=6.5$ it predicts class 1 with high probability. Logistic regression finds the best-fitting logistic curve to separate the classes.

Exercise: Try using `LogisticRegression` on a real dataset, such as the **Iris** dataset (making it binary by classifying one species vs. the rest), or the **Breast Cancer** dataset (`sklearn.datasets.load_breast_cancer`). Evaluate the accuracy and also inspect the coefficients (`clf.coef_`) to see which features positively or negatively contribute to the prediction.

Decision Trees

Concept: A decision tree is a versatile model capable of performing both classification and regression. It learns a set of hierarchical if-else rules to partition the data into homogeneous subsets (in terms of the target variable). The model is structured as a binary tree: internal **nodes** represent tests on features, **branches** represent outcomes of these tests, and **leaf nodes** represent final predictions (a class label for classification, or a numeric value for regression).

For **classification**, a common algorithm is CART (Classification and Regression Tree), which constructs binary splits. At each node, the algorithm chooses a feature and a split point that best separates the classes in the training data. The "purity" of a node is measured by criteria like **Gini impurity** or **entropy** (information gain). For example, Gini impurity for a node with class proportions $p_1, p_2,\dots,p_K$ is $G = \sum_{k=1}^K p_k(1-p_k)$, which is minimized (0) when all samples in the node belong to one class. The tree algorithm greedily chooses splits that maximize the reduction in impurity of the child nodes (or equivalently, maximize information gain) [7] . This process continues recursively until stopping criteria are met (e.g., maximum depth reached or no improvement possible).

For **regression**, the tree algorithm typically uses variance or mean squared error (MSE) as the criterion. At each split, it chooses a feature and threshold that minimize the MSE of the target in the two resulting subsets. Leaves contain a numeric prediction (often the mean of target values in that leaf).

Decision trees **partition the feature space** into rectangular regions (for numerical features), making piecewise constant predictions in each region. They are easy to interpret (you can visualize the tree structure) and handle heterogeneous data well (no need for feature scaling).

Overfitting and Pruning: A key challenge is that an unconstrained decision tree can grow very deep and complex, fitting the training data perfectly (zero error) but failing to generalize (overfitting). To mitigate this, we use hyperparameters to limit complexity: e.g., `max_depth` (maximum depth of the tree), `min_samples_split` (minimum samples required to split a node), `min_samples_leaf` (minimum samples per leaf), etc. **Pruning** techniques can also be applied (scikit-learn implements *cost-complexity pruning* via the `ccp_alpha` parameter) to cut back the tree after fully growing it, by removing branches that have little predictive power.

Python Example (Classification): We demonstrate a decision tree on the Iris dataset for classification. We limit the depth for simplicity and visualization.

```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree

# Load data and train a small decision tree
iris = load_iris()
```

```
X, y = iris.data, iris.target
clf_tree = DecisionTreeClassifier(max_depth=3, random_state=42)
clf_tree.fit(X, y)

# Visualize the tree structure (textual output)
from sklearn.tree import export_text
print(export_text(clf_tree, feature_names=iris.feature_names))
```

This will print a text representation of the tree. Each split is shown as a condition (e.g., "petal width (cm) <= 0.8"), and leaves show the predicted class. The tree stops at depth 3 due to `max_depth=3`. In the output, you'll see how the tree asks a sequence of questions about petal width, petal length, etc., to classify the Iris species.

Python Example (Regression): For regression, let's create a simple dataset and fit a `DecisionTreeRegressor`:

```
from sklearn.tree import DecisionTreeRegressor
# Synthetic dataset: y = x^2 (quadratic), with some noise
X = np.linspace(-3, 3, 50).reshape(-1, 1)
y = X[:, 0]**2 + np.random.normal(scale=1.0, size=X.shape[0])
tree_reg = DecisionTreeRegressor(max_depth=3, random_state=42)
tree_reg.fit(X, y)

# Predict on new data
X_test = [[-2], [0], [2]]
print("Predictions:", tree_reg.predict(X_test))
```

With `max_depth=3`, the tree will make piecewise constant predictions. If you increase the max_depth (try removing the limit), the tree can perfectly interpolate the training points (leading to overfitting). In practice, you'd tune the tree's depth or other parameters to balance bias and variance.

Key point: Decision trees are **non-parametric** models that can capture complex interactions. They are easy to interpret but can be unstable (small data perturbations might drastically change the structure). They form the basis for more powerful ensemble methods like random forests and boosting.

Random Forest

Concept: A Random Forest is an **ensemble** of decision trees, built using the technique of *bagging* (Bootstrap AGGregatING) and random feature selection. The idea is to build many decision trees on random variations of the training data and average their predictions to improve generalization. Each tree in a random forest is trained on a **bootstrap sample** (a random sample of the training set with replacement) [8]. Additionally, at each split in the tree, a random subset of features is considered (rather than all features) when finding the best split [8]. This randomization decorrelates the trees: because each tree sees a different subset of data and features, their errors are less correlated.

After training, the forest makes a prediction by aggregating the predictions of all its trees. For regression, this is usually the **average** of the tree outputs. For classification, it can be the **majority vote** or equivalently the class with the highest average probability across trees. The combined model typically achieves higher accuracy and stability than a single tree, as the averaging reduces variance [9]. In essence, random forests **reduce overfitting** compared to individual deep trees, while maintaining good performance on training data.

Why it works: Individual decision trees have high variance (they can overfit). By averaging many such high-variance models, the overall variance is reduced. The two sources of randomness (bootstrap sampling and random feature splits) ensure the trees are diverse. As a result, errors tend to cancel out when averaging [9]. Empirically, random forests often achieve excellent performance out-of-the-box and are less sensitive to hyperparameters than many other models, making them a popular choice.

Hyperparameters: The key hyperparameters include `n_estimators` (number of trees, often 100+), `max_depth` (or letting them grow fully and relying on many trees), `max_features` (number of features to consider at each split, e.g. sqrt of total features for classification by default), and others like `min_samples_leaf`. Random forests can also give an estimate of feature importance by measuring how much each feature split improves the purity across the forest (accessible via `feature_importances_` attribute).

Python Example: Using scikit-learn's `RandomForestClassifier` on the Iris dataset:

```python
from sklearn.ensemble import RandomForestClassifier
clf_rf = RandomForestClassifier(n_estimators=100, random_state=42)
clf_rf.fit(X, y)  # X, y are Iris data as before
# Predict class probabilities for a new sample
print("Class probabilities:", clf_rf.predict_proba([[5.0, 3.5, 1.5, 0.3]]))
print("Predicted class:", clf_rf.predict([[5.0, 3.5, 1.5, 0.3]]))
```

This fits 100 decision trees on bootstrap samples of the Iris dataset. The output probabilities are the average probabilities from all trees. You should see that the model is quite confident if the sample resembles a specific Iris species. You can also inspect `clf_rf.feature_importances_` to see which features were most useful (for Iris, petal length and width typically are most important).

For regression, you would use `RandomForestRegressor` similarly. Random forests handle non-linear relationships and feature interactions automatically, and they typically don't overfit as drastically as single trees (though with enough trees they can still eventually overfit noisy data).

Exercise: Try using `RandomForestRegressor` on a dataset like the California housing data (`sklearn.datasets.fetch_california_housing`) to predict house prices. Experiment with changing `n_estimators` (e.g. 10 vs 100 vs 1000 trees) and `max_features` to see their effect on performance (use cross-validation to measure validation error). Observe how even with fewer trees, the ensemble can outperform a single decision tree.

Support Vector Machines (SVM)

Concept: Support Vector Machines are powerful models for both classification (SVC) and regression (SVR). The core idea is to find the **best decision boundary** (hyperplane) that separates classes in a high-dimensional feature space with the **maximum margin**. The margin is the distance between the hyperplane and the nearest points from each class (these nearest points are the **support vectors**). By maximizing this margin, SVMs aim to improve generalization (a wider margin means the classifier is more robust to noise) [7].

For **classification (SVC)**: In the simplest linear case, SVM finds a hyperplane defined by (w, b) (weights and bias) that separates class +1 and -1, while maximizing the margin and allowing some slack for misclassified or noisy points. The optimization problem can be written as [10]:

$$\min_{w,b,\{\xi_i\}} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i,$$

subject to the constraints $y_i (w^T x_i + b) \ge 1 - \xi_i$ for each sample i, with $\xi_i \ge 0$ [10]. Here $y_i \in {-1, +1}$ are class labels and ξ_i are slack variables that allow misclassifications. C is a regularization parameter that controls the trade-off between maximizing the margin and minimizing classification errors (penalizing slack) [11]. A large C means less tolerance for misclassification (harder margin, potentially smaller margin), while a small C allows a softer margin (more regularization).

The above is the *primal* form; SVMs are often solved in their **dual form**, which introduces Lagrange multipliers α_i. Importantly, in the dual, training examples only appear inside inner products $x_i \cdot x_j$. This leads to the famous **kernel trick**: by replacing the dot product with a kernel function $K(x_i, x_j)$, SVMs can efficiently compute in high-dimensional (even infinite-dimensional) feature spaces without explicitly mapping to them [12]. Common kernels include **RBF (Gaussian)**, polynomial, and linear. An RBF kernel, for example, allows the model to create a non-linear decision boundary by implicitly mapping data to a higher dimension.

Support Vectors: Only a subset of training samples (the support vectors) end up with non-zero α_i in the dual solution. These are the samples exactly on or inside the margin boundaries—they carry all the information needed for the final classifier. The decision function can be expressed in terms of support vectors: for a new point x,

$$f(x) = \sum_{i \in SV} \alpha_i y_i K(x_i, x) + b,$$

and the predicted class is $\text{sign}(f(x))$ [13].

SVM for Regression (SVR): The Support Vector Regressor works on a similar principle but tries to fit a function within an **epsilon-insensitive tube** around the data. The idea is to ignore errors (differences between predicted and actual values) as long as they are within ϵ, and only penalize errors beyond ϵ. The optimization for SVR introduces slack variables for points outside the epsilon tube, similarly trading off margin (flatness of function) and errors via a parameter C [14]. The result is a regression line (or surface) that is as flat as possible while keeping most training points within the tube of radius $ \epsilon$. Only points outside the tube become support vectors that influence the regression function.

When to use SVM: SVMs are effective in high-dimensional spaces and in cases where number of features > number of samples [15]. They are memory-efficient since only support vectors are used in decision function [16]. With an appropriate kernel, they are quite flexible. However, SVMs do not provide probability estimates naturally (you can enable probability calibration in scikit-learn at some cost), and they can be slower to train on large datasets (training time can be quadratic in number of samples in worst case). They also require careful tuning of kernel parameters (like the RBF kernel width $\gamma$ and the regularization $C$).

Illustration of an SVM classifier in a two-class scenario. The figure shows the decision boundary (red line) maximizing the margin between two classes (blue vs. green points). The support vectors (circled points on the margin boundaries) are the only points influencing the position of the decision boundary. A larger margin (distance between the red line and dashed lines on either side) generally implies better generalization [7]. Points within the margin or on the wrong side can incur slack penalties controlled by $C$.

Python Example: We'll demonstrate using `sklearn.svm.SVC` for classification on a toy dataset and `sklearn.svm.SVR` for regression.

```
from sklearn import svm

# Classification with SVC (linear kernel for simplicity)
X = [[0,0], [0,1], [1,0], [1,1]]  # XOR problem data
y = [0, 1, 1, 0]                  # XOR labels (not linearly separable)
clf_linear = svm.SVC(kernel='linear', C=1.0)
clf_linear.fit(X, y)
print("Predictions (linear kernel):", clf_linear.predict(X))

# Now using a non-linear kernel (RBF) to handle XOR
clf_rbf = svm.SVC(kernel='rbf', C=1.0, gamma=1.0)
clf_rbf.fit(X, y)
print("Predictions (RBF kernel):", clf_rbf.predict(X))
```

In this example, a linear kernel SVM cannot separate XOR perfectly (it will have errors), whereas an RBF kernel can separate the XOR data by mapping to a higher dimension. You'll see the linear SVM might predict something like [0, 0, 0, 0] or similar (failing on XOR), while the RBF SVM will predict [0,1,1,0] correctly after training (since XOR is separable in an RBF feature space).

For SVR, an example:

```
import numpy as np
from sklearn.svm import SVR
# Dataset: y = cos(x) with noise
X = np.linspace(0, 2*np.pi, 100).reshape(-1,1)
```

```python
y = np.cos(X).ravel() + np.random.normal(0, 0.1, 100)
svr = SVR(kernel='rbf', C=1.0, epsilon=0.1)
svr.fit(X, y)
predictions = svr.predict(X)
print("Mean absolute error:", np.mean(np.abs(predictions - y)))
```

The SVR with an RBF kernel will try to fit the noisy cosine curve within an epsilon tube of 0.1. If you lower C (more regularization), the fit will be smoother (underfitting more, wider tube), whereas a very high C would try to pass through all points (risking overfitting). You can experiment with ϵ as well to see how the tolerance for error affects the number of support vectors (fewer support vectors if epsilon is large, since more points lie in the tube without penalty).

Summary: SVMs are robust and effective, especially for smaller to medium-sized datasets and for complex but well-behaved decision boundaries. They require selecting a suitable kernel and tuning parameters like C and γ (for RBF). When properly tuned, they can achieve high performance and are theoretically grounded in margin maximization. For very large datasets, other algorithms (or using the linear SVM variant or stochastic methods) might be more feasible due to computational constraints.

Introduction to Boosting Algorithms (e.g., XGBoost)

Concept: *Boosting* is an ensemble technique that turns a collection of weak learners into a strong learner. Unlike bagging (used in random forests) where models are trained independently in parallel, boosting trains models sequentially, each trying to correct the errors of the previous one [17]. The general idea is:

1. Start with an initial base model (could be as simple as a constant prediction).
2. Iteratively add new models that focus on the **residual errors** or "hard" examples from the previous models.
3. Each new model's contribution is scaled (often by a learning rate) and added to the ensemble.

Early boosting algorithms like **AdaBoost** (Adaptive Boosting) use decision stumps (depth-1 trees) as weak learners. AdaBoost adjusts sample weights at each iteration: misclassified instances get higher weight so that the next learner focuses on them [17]. After many rounds, it combines the weak rules into a single strong prediction rule [18].

A more modern approach is **Gradient Boosting**, which views the boosting procedure as an optimization problem. It trains new models to be maximally correlated with the **negative gradient** of the loss (hence "gradient boosting"). **XGBoost (Extreme Gradient Boosting)** is a popular, optimized implementation of gradient boosting trees. It includes enhancements like regularization, efficient handling of missing data, and parallel computation, which often make it faster and more accurate than a plain implementation of gradient boosting. XGBoost has been very successful in machine learning competitions due to its speed and performance.

XGBoost specifics: In gradient boosting for regression, for example, you start with an initial prediction (like the mean of y). Then at each stage, you compute the residuals $r_i = y_i - \hat{y}_i$ (which are the gradients of squared error loss), and you fit a small regression tree to these residuals. The new tree is added to the model with a scaling factor (learning rate η). In XGBoost, an objective function is defined that includes a regularization term for tree complexity, and the algorithm uses second-order gradient

(Newton) information to optimize more efficiently. The regularization helps to prevent the boosted model from overfitting by penalizing overly complex trees.

Use as assignment: Boosting algorithms are a bit more advanced, so students are encouraged to independently research them. As an assignment, one might explore how XGBoost works and perhaps use it on a dataset to compare performance with, say, a random forest. Key points to learn include how the learning rate and number of estimators trade off (more trees with a lower learning rate often perform best), and how regularization parameters (like `max_depth` of trees, `subsample`, `colsample_bytree`, and XGBoost-specific `gamma`, `lambda`) help tune model complexity.

Practical usage of XGBoost: XGBoost is available via the `xgboost` Python package and also through scikit-learn API wrappers. A simple example of using XGBoost (for classification):

```python
# Make sure to install xgboost package first: pip install xgboost
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Example: binary classification on scikit-learn’s breast cancer dataset
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,
random_state=42)

model = xgb.XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=3,
use_label_encoder=False, eval_metric='logloss')
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("XGBoost Accuracy:", accuracy_score(y_test, y_pred))
```

This snippet trains an XGBoost classifier with 100 boosting rounds, a learning rate of 0.1, and max tree depth of 3 (small trees). In practice, you would tune these parameters and possibly use cross-validation or XGBoost’s built-in `cv` function to find the optimal number of rounds.

Further resources: To truly understand boosting, it’s recommended to study sources like the original AdaBoost paper by Freund & Schapire, or Friedman’s gradient boosting paper [19] . There are also excellent video tutorials – for example, the YouTube channel StatQuest has approachable videos on AdaBoost and Gradient Boost (and XGBoost) which many students find helpful. These can provide the intuition behind how each subsequent model corrects errors of the ensemble.

In summary, boosting is a powerful technique that often yields state-of-the-art results on structured data. **XGBoost**, in particular, is a go-to tool for many Kaggle competition winners. As an assignment, implementing a simple boosting algorithm or using XGBoost on a dataset will deepen your understanding of how ensemble models can significantly outperform individual models by addressing their weaknesses in sequence.

2. Evaluation Metrics

Choosing the right **evaluation metrics** is crucial for assessing model performance. Metrics translate model predictions into quantitative measures of quality. We will discuss common metrics for regression and classification tasks. For regression: **MAE, MSE, RMSE, R^2**; for classification: **Accuracy, Precision, Recall, F1-score**. We will define each, provide formulas, and context on when to use them. (Note: In classification, these metrics are based on the confusion matrix concepts of True Positives, False Positives, True Negatives, False Negatives. We assume binary classification for simplicity, with "positive" meaning the class of interest.)

Regression Metrics

For regression problems (predicting continuous values), error-based metrics are used to measure the discrepancy between predicted values $\hat{y}_i$ and true values y_i over n samples:

- **Mean Absolute Error (MAE):** The average of the absolute errors. Formula:

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \,.$$

  MAE is a straightforward measure of model error in the same units as the target. It's more robust to outliers than MSE since it doesn't square the errors (outliers contribute linearly) [20]. A MAE of 0 means perfect predictions. MAE is useful when all errors are equally important, and you want an interpretable metric (e.g., "on average, our prediction is off by \$500").

- **Mean Squared Error (MSE):** The average of the squared errors:

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

  It penalizes larger errors more severely (squaring emphasizes outliers). It's the loss function often used for training linear regression. A lower MSE indicates better performance; 0 is perfect. However, because of squaring, the units of MSE are the square of the target's units (e.g., dollars^2). Sometimes an unbiased estimator of variance is referred to as MSE in statistical contexts, but here we simply mean the empirical mean of squared residuals [21].

- **Root Mean Squared Error (RMSE):** Simply the square root of MSE:

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

  RMSE brings the error metric back to the original units of y, making it more interpretable (it can be directly compared to y values) [22]. RMSE is often preferred for interpretation over MSE. It's also more sensitive to large errors (due to the squaring before the root). In practice, RMSE is one of the most commonly reported regression metrics (for example, in competitions).

Insight: Both MSE and RMSE disproportionately penalize large errors. If your application cannot tolerate large errors, these metrics will capture that. If outliers are not as important, MAE might be more appropriate.

- **R² Score (Coefficient of Determination):** This is not an error but a relative measure of how well the model explains the variance in the target. The formula is:

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

  where the numerator is the model's RSS (residual sum of squares) and the denominator is the total sum of squares (variance of the data around the mean) [23] . R^2 represents the proportion of variance in y that is captured by the model. An R^2 of 1 indicates a perfect fit (all predictions match actual values, zero residual error). An R^2 of 0 indicates the model is no better than always predicting the mean $\bar{y}$. An R^2 can even be negative if the model is worse than the mean baseline (which can happen if it's poorly trained or overfits bizarrely on test data) [23] .

Example: If $R^2 = 0.85$, we say "the model explains 85% of the variance in the target." Keep in mind, a high R^2 doesn't guarantee the model is good in an absolute sense; always check residual plots and other metrics. For nonlinear relationships, R^2 is still applicable but one should be cautious in interpretation if assumptions of linear correlation are violated.

When to use which: MSE/RMSE are useful when large errors are especially undesirable (e.g., in some financial forecasts a single big miss can be catastrophic). MAE is more interpretable and robust when errors have a roughly linear cost. R^2 is a unit-less measure to quickly communicate goodness-of-fit, but it should be complemented with absolute error measures. It's common to report both RMSE (or MAE) and R^2 together.

Python snippets: Given true values `y_true` and predictions `y_pred`, you can compute these metrics with scikit-learn functions:

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
mae = mean_absolute_error(y_true, y_pred)
mse = mean_squared_error(y_true, y_pred)
rmse = mean_squared_error(y_true, y_pred, squared=False)  # directly get RMSE
r2 = r2_score(y_true, y_pred)
```

These will give you the numeric values for each metric.

Classification Metrics

For classification (especially binary classification), metrics are computed from the confusion matrix, which compares predicted labels vs true labels. Let's define terms for a binary positive/negative class scenario:

- **True Positive (TP):** Model predicted Positive, and the actual was Positive.
- **True Negative (TN):** Model predicted Negative, and actual was Negative.

- **False Positive (FP):** Model predicted Positive, but actual was Negative (Type I error, also called a “false alarm”).
- **False Negative (FN):** Model predicted Negative, but actual was Positive (Type II error, a “miss”).

Based on these, we define:

- **Accuracy:** The proportion of all predictions that were correct. Formula:

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

  It is perhaps the simplest metric [24]. An accuracy of 1.0 (or 100%) means all predictions were correct. Accuracy is a good metric when classes are fairly balanced and errors cost roughly the same regardless of class. However, **accuracy can be misleading** in imbalanced datasets. For example, if 99% of instances are negative, a model that always predicts “Negative” achieves 99% accuracy but is useless. Thus, we often look at precision/recall in tandem for imbalanced scenarios.

- **Precision:** Of the instances *predicted* as Positive, what fraction were truly Positive? It focuses on the quality of positive predictions. Formula:

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

  This is also known as **Positive Predictive Value** [25] [26]. Precision answers: “When the model predicts positive, how often is it correct?” A high precision means that **few false alarms** are made (FP is low relative to TP). Precision is crucial in scenarios where false positives are costly – for instance, a spam filter where predicting non-spam as spam (false positive) might drop important emails; you’d want high precision in flagging spam (so that when something is flagged, it’s very likely truly spam).

- **Recall:** Of the instances that are actually Positive, what fraction did the model correctly identify as Positive? Formula:

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

  which is also the **True Positive Rate (TPR)** or **Sensitivity** [27] [28]. Recall answers: “Out of all actual positives, how many did we catch?” A high recall means **few positive instances are missed** (FN is low). Recall is crucial when false negatives are costly – e.g., in disease screening, missing a sick patient (false negative) is far more serious than a false alarm, so we want high recall (detect as many sick patients as possible).

- **F1-Score:** The harmonic mean of precision and recall:

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

  It combines precision and recall into one number [29]. The F1-score is high only when both precision and recall are high; if one is low, F1 will be closer to the lower one. It’s a convenient metric for overall performance, especially when classes are imbalanced, since it doesn't get inflated by the majority class like accuracy can. An F1 of 1 means perfect precision and recall (no false positives or false negatives) [30] . An F1 of 0 means either precision or recall is zero (the model fails completely on one of those measures).

Precision/Recall Trade-off: There is often a trade-off between precision and recall. By adjusting the decision threshold of a classifier (especially for models like logistic regression or SVM that output a score or probability), one can increase precision at the cost of recall or vice versa [31] . For instance, to get higher precision, one might use a higher threshold for positive prediction (the model then only flags cases it's very sure about, reducing false positives but likely increasing false negatives, thus lower recall). Conversely, a lower threshold yields more positives (catching more true positives = higher recall, but also more false positives = lower precision) [25] [32] . This trade-off can be visualized with a **Precision-Recall curve**, and summarized by metrics like **Average Precision (AP)** or area under the PR curve.

When to use: - Accuracy is fine for balanced datasets with equal error costs. - Precision and recall are critical for imbalanced datasets or where false positive vs false negative have different implications. Often quoted together: e.g., "Precision = 90%, Recall = 70%". - F1-score is useful as a single measure of a classifier's accuracy that balances precision and recall, especially in comparative studies or model selection.

Python Example: Suppose we have ground truth labels and model predictions for a binary classification:

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix

y_true = [1, 0, 1, 1, 0, 0, 1, 0]       # 1 = Positive, 0 = Negative (example)
y_pred = [1, 0, 1, 0, 0, 0, 1, 1]       # model predictions

acc = accuracy_score(y_true, y_pred)
prec = precision_score(y_true, y_pred)
rec = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)
cm = confusion_matrix(y_true, y_pred)

print("Confusion Matrix:\n", cm)
print(f"Accuracy: {acc:.2f}, Precision: {prec:.2f}, Recall: {rec:.2f}, F1: {f1:.
2f}")
```

This will output the confusion matrix (which might look like `[[TN, FP], [FN, TP]]`) and the computed metrics. For example, suppose the confusion matrix is:

```
[[3 1]
 [1 3]]
```

This means: TN=3, FP=1, FN=1, TP=3. Then Accuracy = (3+3)/8 = 0.75 (75%). Precision = 3/(3+1) = 0.75. Recall = 3/(3+1) = 0.75. F1 = 0.75 as well (in this symmetric case).

If we had an imbalanced scenario or different predictions, we could see precision and recall diverge. For instance, if the model predicted positive only when extremely sure, we might get a high precision but low recall. If it predicts positive for nearly everything, we'd get high recall but low precision. The F1 would capture the balance.

Multi-class classification: The above precision/recall are defined for binary. For multi-class, these can be generalized (one can compute precision/recall per class in a one-vs-rest manner, then average – either "macro" average treating all classes equally, or "weighted" average weighting by class frequency). Scikit-learn's `precision_score` and others have an `average` parameter to handle this.

Summary: Use these metrics according to the problem needs. For example: - In information retrieval (search engines), **precision** is important (the results you show should be relevant) but also **recall** if you don't want to miss relevant documents [33]. - In medical tests, **recall (sensitivity)** is crucial (catch all the disease cases), but one also looks at **precision** (positive predictive value) to gauge how many flagged positives are actually sick, and possibly **specificity** (which is TN/(TN+FP), the counterpart to recall for negatives). - **Accuracy** is fine in cases like recognizing handwritten digits where each class is equally likely and we care about overall correctness. - **F1** is a handy single metric for model comparison, especially under class imbalance.

By understanding and computing these metrics, we can better interpret how our models perform and where they might be failing. Always choose the metric that aligns with the real-world objectives of your application (for instance, if false negatives are far worse than false positives, focus on maximizing recall even if precision suffers). Often, multiple metrics are used together to get a full picture of performance.

References: The formulas and definitions above are drawn from standard definitions [24] [26] [29] and the scikit-learn documentation. For instance, scikit-learn defines precision as $\frac{TP}{TP+FP}$ and recall as $\frac{TP}{TP+FN}$ [25] [27]. The F1 formula is given by scikit-learn as well [30]. These metrics, along with confusion matrix terminology, are widely used in machine learning evaluations.

---

[1] [3] [4] [5] [6] 1.1. Linear Models — scikit-learn 1.7.2 documentation
https://scikit-learn.org/stable/modules/linear_model.html

[2] LinearRegression — scikit-learn 1.7.2 documentation
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

[7] [10] [11] [12] [13] [14] [15] [16] 1.4. Support Vector Machines — scikit-learn 1.7.2 documentation
https://scikit-learn.org/stable/modules/svm.html

[8] [9] [19] 1.11. Ensembles: Gradient boosting, random forests, bagging, voting, stacking — scikit-learn 1.7.2 documentation
https://scikit-learn.org/stable/modules/ensemble.html

[17] [18] Implementing the AdaBoost Algorithm From Scratch - KDnuggets
https://www.kdnuggets.com/2020/12/implementing-adaboost-algorithm-from-scratch.html

[20] Mean absolute error - Wikipedia
https://en.wikipedia.org/wiki/Mean_absolute_error

[21] R2 Score & Mean Square Error (MSE) Explained – BMC Software | Blogs
https://www.bmc.com/blogs/mean-squared-error-r2-and-variance-in-regression-analysis/

[22] Mean squared error - Wikipedia
https://en.wikipedia.org/wiki/Mean_squared_error

[23] R-Squared: Coefficient of Determination in Machine Learning
https://arize.com/blog-course/r-squared-understanding-the-coefficient-of-determination/

[24] [26] [28] [29] [31] Classification: Accuracy, recall, precision, and related metrics | Machine Learning | Google for Developers
https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall

[25] [27] [32] [33] Precision-Recall — scikit-learn 1.7.2 documentation
https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html

[30] f1_score — scikit-learn 1.7.2 documentation
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html