Guidelines for the Exam Project

Programming and Machine Learning for Biological Data (PMLBD)

Sapienza University of Rome — MSc in Genetics and Molecular Biology

Who is this document for? — These guidelines explain how to complete the PMLBD exam project. The goal is not to provide a recipe to follow mechanically, but to help you **understand what you are doing and why**. The evaluator wants to see not only that your code works, but that you understand the rationale behind each methodological choice, the risks of common mistakes, and the implications of your results in the context of the biological or clinical problem you have chosen.

The guidelines follow the exact order of the workflow you will need to implement. Between sections you will find coloured boxes: ⚠️ **Warning** flags common errors and data leakage risks; ✅ **Recommended practice** indicates the correct solution; ℹ️ **Methodological note** elaborates on the reasoning behind a choice.

Document structure

1. Dataset selection and description
2. Exploratory Data Analysis (EDA)
3. Preprocessing before the split
4. Dataset split
5. Preprocessing after the split — the Pipeline
6. Model selection and comparison
7. Hyperparameter tuning
8. Final evaluation and overfitting/underfitting check
9. Feature importance analysis
10. Conclusions

Appendix A — Data Leakage: visual summary

Appendix B — Final checklist before submission

1. Dataset selection and description

1.1 Selection criteria

The dataset can be chosen from Kaggle and must be suitable for a **supervised binary or multiclass classification** problem. Before choosing it, verify that:

- **The target is categorical** (binary or multiclass) and has a clear, interpretable meaning in the context of the problem.
- **The number of observations is adequate**: with fewer than 200 rows the statistical results will be very unstable. Ideally, use at least 500–1000 observations.

- **The features make sense**: there must be variables with a plausible relationship with the target.
- **The ratio between the number of data points and the number of features should be adequate**: as a general rule, at least 10–20 observations per feature are needed to reduce the risk of overfitting and obtain statistically stable estimates.
- **There are no features that constitute structural data leakage**: variables that directly encode the target or are only available after the event to be predicted (e.g. 'final_diagnosis' to predict a diagnosis, 'final_score' to predict success).

1.2 Dataset description in the notebook

Dedicate the first section of your notebook to presenting the dataset. This section must contain all of the following elements — none are optional:

1. **Problem context:** What is the real problem? Why is it relevant? In a clinical or biological context, what does it mean to correctly predict the target? What are the consequences of a false negative or a false positive?
2. **Source:** Link to the dataset on Kaggle. Number of observations and features.
3. **Target description:** What is the target variable? How many classes does it have? What does each class represent? How is it encoded in the dataset?
4. **Feature table:** For each variable write: name, type (continuous numerical / nominal categorical / ordinal categorical / binary), range or possible values, and meaning in the context of the problem.

> ℹ️ **Methodological note**
>
> The distinction between variable types is not a formality: it directly determines which preprocessing to apply.
>
> **Continuous numerical variable** (e.g. Age, budget in USD) → standardisation with StandardScaler.
>
> **Nominal categorical variable** (3+ categories with no order, e.g. country, film genre) → One-Hot Encoding inside the Pipeline.
>
> **Ordinal categorical variable** (with order, e.g. low/medium/high) → deterministic numerical mapping that respects the order.
>
> **Binary variable** (already 0/1, or mapped to 0/1) → passthrough, no transformation.
>
> Applying the same preprocessing to all variable types is a **conceptual error**.

1.3 Attention to target creation

When the target is not already in the dataset and must be created by binarising a continuous variable (e.g. score ≥ 7 → successful film), you must:

- **Show the distribution of the source variable** before choosing the threshold (histogram).
- **Justify the chosen threshold** with quantitative arguments (e.g. median, percentile) or bibliographic references.

- **Immediately remove the source variable** from the features after creating the target.
- **Verify the class balance** of the resulting target.

> ⚠️ **Warning — Data Leakage**
>
> If you create the target by binarising a continuous variable (e.g. score_high from score) and do NOT remove the source variable from the features, the model will use that value to trivially reconstruct the threshold.
>
> The metrics will look excellent (>95%) but will be completely meaningless as predictive evidence: you are training the model to remember a rule you wrote, not to find patterns in the data.

2. Exploratory Data Analysis (EDA)

EDA is not decorative: it serves to understand the structure of the data and to make informed preprocessing decisions. **Every output must be commented**: it is not enough to display a graph or a table, you must explain what you observe and how it influences subsequent choices.

2.1 Initial inspection

- Display the first rows with `df.head()` and verify that the data makes sense.
- Check `df.shape`, `df.dtypes` and `df.nunique()` for each column.
- **Identify variables that look numerical but are actually categorical** (e.g. Gender=1/2, Group=1/2). This is one of the most frequent errors: treating these variables as numbers introduces non-existent artificial orderings.

2.2 Missing values

Use `df.isnull().sum()` per column. For each column with NaN, decide and justify:

Situation	Recommended strategy
< 5% NaN	Impute with mean/median (numerical) or mode (categorical) — INSIDE the Pipeline, after the split.
5–30% NaN	Evaluate case by case. If the variable is relevant, impute. Otherwise consider removing it. Justify the choice.
> 30% NaN	It is generally better to remove the column. Too many NaN values make the variable unreliable.
NaN concentrated in a few rows	You may remove those rows before the split (learns nothing from the data), but verify they are not systematically different from the others.

> ⚠️ **Warning — Data Leakage**
>
> Imputation (SimpleImputer) must occur AFTER the split and INSIDE the Pipeline, so that the statistics (mean, median, mode) are computed only on the training set. If you impute before, the test set statistics 'contaminate' the training: this is data leakage.

2.3 Duplicates

Check with `df.duplicated().sum()`. Unremoved duplicates can cause the same row to end up in both the training and test sets, making metrics artificially optimistic. Remove them with `df.drop_duplicates()` before the split and comment on how many there were.

2.4 Target distribution and class balance

Always verify the target distribution with **value_counts()** and a graph before proceeding. Report the exact numerical proportions. This step is fundamental:

- **Balanced classes**: accuracy is a reliable metric. Resampling is not necessary.
- **Imbalanced classes**: accuracy can be misleading. Use F1-score, precision and recall as the main metrics. Consider resampling techniques only AFTER verifying performance without resampling.

> ℹ️ **Methodological note**
>
> When and how to apply resampling:
>
> **Undersampling** (RandomUnderSampler): reduces the majority class. Preferable when the dataset is large.
>
> **Synthetic oversampling** (SMOTE): generates synthetic examples of the minority class. Useful when the dataset is small, but requires caution: it can cause overfitting if used incorrectly.

2.5 Distributions and correlations

- Visualise the distributions of the main numerical variables (histograms, boxplots). Comment on any evident outliers.
- Compute the Pearson correlation matrix **only on continuous numerical variables** (not on categorical variables encoded as numbers) and visualise it as a heatmap.
- Comment on the strongest correlations: between features (multicollinearity) and between features and the target.
- Do not use correlation with the target as the sole criterion for removing features before modelling: some features with low linear correlation may be relevant for non-linear models.

3. Preprocessing before the split

Fundamental rule: before the split you may only perform operations that **do not learn information from the data**.

3.1 What is allowed before the split

✅ **Recommended practice**
Correct operations before the split (they do not learn from the data):

- Removal of useless columns (identifiers such as ID, patient name, etc.)
- Removal of duplicate rows
- Removal of columns with too many missing values (> 30–50%)
- Deterministic mapping of binary categorical variables (e.g. Gender: 1→1 Male, 2→0 Female)
- Creation of the target and removal of the source variable (if the target is derived from a continuous variable)
- Re-encoding the target as 0/1 (if the original values are 1/2 or similar)

⚠️ **Warning — Data Leakage**
Operations NOT allowed before the split:

- Missing value imputation (SimpleImputer): learns mean/median/mode from the dataset → data leakage
- StandardScaler or MinMaxScaler: learn mean and standard deviation from the dataset → data leakage
- OneHotEncoder or pd.get_dummies on variables with 3+ categories: learn which categories exist in the dataset → data leakage
- Removal of outliers based on statistics computed on the entire dataset (e.g. IQR, z-score)
- Feature selection based on correlations or statistical tests computed on the entire dataset

3.2 Deterministic mapping of binary variables

Binary variables with non-0/1 encoding should be remapped before the split. Example:

```
df['Gender'] = df['Gender'].map({1: 1, 2: 0})  # Male→1, Female→0
```

For each mapping, explain in the markdown: which value corresponds to which category and why you chose which to assign to 0 and which to 1.

3.3 Variable classification for preprocessing

In this section of the notebook, explicitly list the variables divided into three groups, as they will determine the structure of the Pipeline:

- **Continuous numerical variables**: will receive imputation + StandardScaler.
- **Nominal categorical variables with 3+ categories**: will receive imputation + OneHotEncoder.
- **Binary variables (0/1)**: will receive only imputation if necessary, then passthrough.

4. Dataset split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=SEED, stratify=y)
```

Parameter	Why it matters
test_size=0.2	20% for the test set (80% training). With small datasets you can use 0.25–0.30.
random_state=SEED	Guarantees reproducibility. Define SEED=42 at the start of the notebook and use it everywhere.
stratify=y	Maintains the same target proportions in both sets. ALWAYS required for classification.

After the split, verify: (1) the dimensions of X_train and X_test; (2) the target distribution in both sets with `value_counts(normalize=True)`.

> ⚠️ **Warning — Data Leakage**
>
> From the moment you perform the split, the test set is untouchable until the final evaluation.
>
> Do not use it to choose hyperparameters, do not scale on it, do not 'peek' at its metrics during development.
>
> The test set is used ONLY ONCE, at the end, for the evaluation of the definitive model.

5. Preprocessing after the split — the Pipeline

All preprocessing that 'learns' from data (scaling, encoding, imputation) must occur inside a **sklearn Pipeline**. This automatically guarantees that the fit occurs only on the training set and that the same fitted transformer is applied to the test set, with no possibility of errors or leakage.

5.1 Pipeline structure with ColumnTransformer

The correct structure is a `ColumnTransformer` that applies different transformers to different groups of columns:

```
numeric_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])
categorical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="most_frequent")),
          ("encoder",  OneHotEncoder(drop="first",  handle_unknown="ignore",
sparse_output=False))
])
binary_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="most_frequent"))
])
preprocessor = ColumnTransformer(transformers=[
    ("num", numeric_transformer, numeric_features),
    ("cat", categorical_transformer, categorical_features),
    ("bin", binary_transformer, binary_features)
])
```

5.2 Why not standardise binary variables

Binary variables (0/1) should not be standardised. Standardisation solves the problem of features with very different ranges and variances. Binary variables already all have the same maximum possible variance for a 0/1 variable. Standardising them brings no benefit and may introduce confusion.

5.3 OneHotEncoder vs pd.get_dummies

Use sklearn's `OneHotEncoder` (inside the Pipeline) rather than `pd.get_dummies`. The fundamental difference is:

- **pd.get_dummies** applied before the split → data leakage (sees the test set categories). Even applied after the split on X_train, it does not guarantee that X_test will have the same columns.
- **OneHotEncoder inside the Pipeline**: fitted only on X_train, then applied to X_test with the same categories. If a category appears in the test set that was never seen in training, `handle_unknown='ignore'` handles it without crashing.

5.4 Adding the final model to the Pipeline

The classifier is added as the last step of the Pipeline:

```
full_pipeline = Pipeline(steps=[('preprocessor', preprocessor), ('model', RandomForestClassifier(random_state=SEED))])
```

By calling `full_pipeline.fit(X_train, y_train)`, the preprocessor is fitted on X_train and the model is trained on its outputs. By calling `full_pipeline.predict(X_test)`, the preprocessor transforms X_test using the parameters already fitted on X_train.

6. Model selection and comparison

Before optimising hyperparameters, compare at least 3–4 different models using **cross-validation on the training set**. This allows you to understand which family of models works best on your data, without touching the test set.

6.1 Recommended models to compare

Model	When it works well
K-Nearest Neighbors (KNN)	Small/medium datasets, features on similar scales. Suffers in high dimensionality (curse of dimensionality).
Logistic Regression	Linear relationships between features and target. Fast, interpretable, good baseline.
Decision Tree	Non-linear relationships, mixed data. Risk of overfitting if not regularised (max_depth).
Random Forest	Robust, handles mixed features and outliers well. Provides feature_importances_. Excellent in many contexts.
Gradient Boosting / XGBoost	Often the most accurate on tabular data. Slower to optimise. Good alternative to Random Forest.

6.2 How to compare models correctly

Use `StratifiedKFold` with `cross_val_score` on X_train, for each model (inside its Pipeline):

```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED)
scores = cross_val_score(pipeline, X_train, y_train, cv=cv, scoring='f1')
```

For each model, report the **mean and standard deviation** of the CV scores. The std matters: a model with mean F1 0.78 ± 0.02 is more reliable than one with 0.80 ± 0.12.

6.3 Which metric to use for model comparison

Situation	Main metric
Balanced classes	Accuracy or F1-score (macro)

Imbalanced classes	F1-score (weighted or on the minority class), ROC-AUC
Context where false negatives are costly (e.g. medical diagnosis)	Recall (sensitivity) of the positive class
Context where false positives are costly (e.g. aggressive spam filter)	Precision of the positive class

> ⚠️ **Warning — Data Leakage**
>
> Do not optimise hyperparameters for each model at this stage: compare models with default hyperparameters (or a reasonable, equal value for all).
>
> If you optimise some models and not others before the comparison, the comparison is not fair.
>
> Hyperparameter optimisation must be done ONLY on the winning model, in the next phase.

7. Hyperparameter tuning

After choosing the best model, optimise its hyperparameters using **GridSearchCV or RandomizedSearchCV** with cross-validation on the training set.

7.1 GridSearchCV inside the Pipeline

```
grid_search = GridSearchCV(estimator=full_pipeline, param_grid=param_grid, cv=cv, scoring='f1', n_jobs=-1, refit=True)
grid_search.fit(X_train, y_train)
```

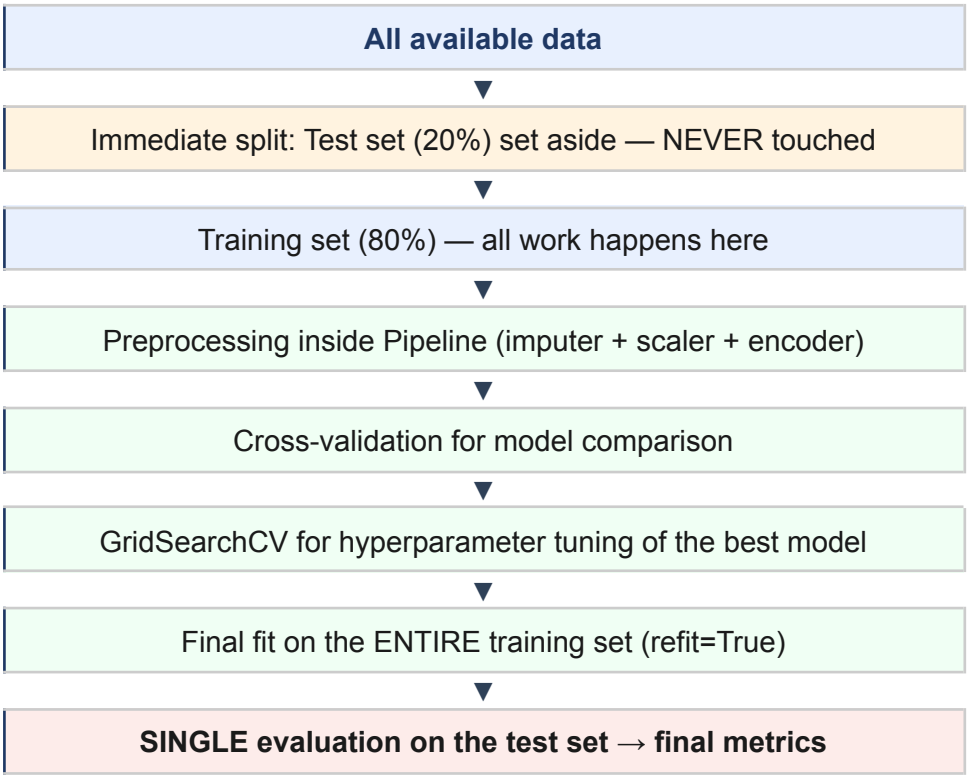
Parameter	Meaning
param_grid	Dictionary with the hyperparameters to explore. For a Pipeline, the parameter name is 'step__parameter' (e.g. 'model__n_estimators').
cv=cv	The same StratifiedKFold used for model comparison.
scoring='f1'	The metric to optimise. Must be the same used for model comparison.
refit=True	After finding the best hyperparameters, automatically re-trains the model on the ENTIRE training set. This produces the final model ready for evaluation.
n_jobs=-1	Uses all available cores to parallelise the search. Greatly speeds up the process.

7.2 Which hyperparameters to optimise

Model	Main hyperparameters to explore
KNN	n_neighbors (wide range, e.g. 1–40), weights ('uniform'/'distance'), metric ('euclidean'/'manhattan')
Logistic Regression	C (regularisation strength: 0.01, 0.1, 1, 10, 100), solver ('liblinear'/'lbfgs')
Decision Tree	max_depth (None, 3, 5, 10), min_samples_split (2, 5, 10), min_samples_leaf (1, 2, 4)
Random Forest	n_estimators (50, 100, 200), max_depth (None, 5, 10), min_samples_split, min_samples_leaf

7.3 The correct workflow — diagram

The workflow that MUST be followed:



> ⚠️ **Warning — Data Leakage**
>
> *Common errors in tuning:*
>
> - Optimising hyperparameters by evaluating performance on the test set instead of using CV on the training set: this is data leakage - the model indirectly sees the test set and the final metrics will not be reliable
> - After GridSearchCV, running a further CV on `best_estimator_`: this makes no sense. GridSearchCV with `refit=True` has already fitted the optimal model on the

entire training set. The final evaluation is done ONCE on the test set, not with another CV
- Using a different metric for model selection and for tuning (e.g. accuracy for comparison, f1 for GridSearchCV): the results are not directly comparable — use the same metric throughout
- Not including the preprocessor in the GridSearchCV Pipeline: the scaler and encoder are fitted on the entire training set before the internal CV, causing data leakage on the validation folds

8. Final evaluation and overfitting/underfitting check

This is the final phase of the workflow. The optimal model (`grid_search.best_estimator_`) is evaluated on the test set for the first and only time.

8.1 Metrics to report

Compute and comment on all of the following metrics, both on the training set and the test set:

- **Accuracy**: percentage of correct predictions. Reliable only with balanced classes.
- **Precision**: among the samples predicted as positive, how many actually are?
- **Recall (Sensitivity)**: among the truly positive samples, how many are correctly identified?
- **F1-score**: harmonic mean of precision and recall. Main metric for imbalanced classes.
- **Confusion matrix**: shows TP, TN, FP, FN. Comment on false positives and false negatives in the context of the real problem.
- **Classification report**: reports precision, recall and F1 per class. Do not just print it: read the numbers and comment on them.

8.2 Overfitting and underfitting check

This step is mandatory. Compare the metrics on the training set with those on the test set:

Situation	Interpretation and action
Training score ≫ Test score (e.g. train F1=0.98, test F1=0.72)	Overfitting: the model has memorised the training set. Regularise the complexity (max_depth, min_samples_leaf, smaller C).
Training score ≈ Test score (good value)	Ideal situation: the model generalises well.
Training score ≈ Test score (low value)	Underfitting: the model is too simple or the features are not informative enough.

Test score > Training score	Anomaly signal. Probable causes: data leakage, test set too small or 'lucky'. Review the workflow from the start.

> ⚠️ **Warning — Data Leakage**
>
> A model that performs better on the test set than on the training set is almost always a signal that something is wrong in the workflow.
>
> The most probable causes are: (1) data leakage — test set information has leaked into training during preprocessing or scaling; (2) test set too small — with few samples, metrics are very unstable; (3) non-stratified split — the test set contains a favourable class proportion.
>
> If you observe this situation, review the workflow from the start, in particular: do encoding and scaling occur inside the Pipeline? Is resampling applied only to the training set?

8.3 ROC-AUC curve (optional but recommended)

The ROC curve and AUC (Area Under the Curve) provide an evaluation of the model's discriminative ability, independent of the classification threshold. An AUC of 0.5 is equivalent to a random classifier; 1.0 is a perfect classifier. They are particularly useful for comparing multiple models in a single graph.

9. Feature importance analysis

Feature importance analysis is a **mandatory** step. It serves to understand which variables the model found most useful for prediction, and to draw interpretable conclusions in the biological or clinical context of the problem.

9.1 How to obtain it depending on the model

Model	How to obtain feature importance
Random Forest	rf.feature_importances_ — importance based on mean impurity decrease (Gini). Directly accessible.
Gradient Boosting / XGBoost	Same feature_importances_ attribute. Similar to Random Forest.
Logistic Regression	Coefficients (coef_) after standardisation. A high coefficient (in absolute value) indicates high importance.
Decision Tree	feature_importances_. Same approach as Random Forest.
KNN	Not natively available. Use model-agnostic methods such as Permutation Importance (sklearn.inspection.permutation_importance).

9.2 How to visualise and comment on it

Visualise feature importances as a horizontal bar chart, ordering features from most to least important. Show the top 10–20 features.

In the markdown, **comment on the result in the context of the problem**: which variables were most important? Does this make biological/clinical sense? Are there features you expected to be important that turned out not to be? Are there surprisingly relevant variables?

> ℹ️ **Methodological note**
>
> *Notes on interpreting Random Forest feature importance:*
>
> - Gini importance may overestimate variables with high cardinality or many unique values
> - Permutation Importance (`permutation_importance` from sklearn) is a more robust alternative: it measures how much the model deteriorates when the values of a feature are randomly shuffled. Unlike Gini importance, it is evaluated directly on a validation set and reflects actual predictive contribution rather than training-time impurity reduction
> - If two features are highly correlated, the importance tends to be distributed between them, artificially reducing the measured value of each — even if both are genuinely informative

9.3 Conclusions on feature importance

After the analysis, write a paragraph that answers these questions:

- Which are the 3–5 most important features? What do they have in common?
- Do the results confirm prior knowledge about the problem domain, or do they call it into question?
- Are there features you could collect in the future to improve the model?

10. Conclusions

The conclusions must be **exhaustive and structured**. It is not sufficient to say 'the model works well'. The reader must understand what you did, why, and what the results mean.

10.1 Recommended structure

5. **Workflow summary:** Briefly summarise each phase: dataset chosen, preprocessing applied, models compared, model selected, hyperparameters optimised.
6. **Problems encountered:** Did you find missing values? How did you handle them? Were the classes imbalanced? Did you apply resampling? Did you encounter overfitting? How did you address it?

7. **Final model performance:** Report the main metrics (accuracy, F1, precision, recall) on the test set. Comment on the confusion matrix in the context of the problem: do false positives or false negatives have different implications?
8. **Biological/clinical interpretation:** What do the feature importances tell us? Which variables matter most? What does this mean in the real-world context of the problem?
9. **Model limitations:** Is the dataset small? Were some relevant features unavailable? Might the results not generalise to other populations? Be honest about the limitations.
10. **Possible future developments:** What would you improve with more data, more time, or more resources? What additional features would you collect?

Appendix A — Data Leakage: summary

Data leakage is the most insidious methodological error in machine learning: the metrics look great, but the model will never work on new data. It occurs when information from the test set (or from the future) 'leaks' into the training.

A.1 Most common forms of data leakage

Form of leakage	How to recognise it	How to avoid it
Preprocessing before the split	Scaling, encoding or imputation computed on the entire dataset before train_test_split.	Move everything inside the Pipeline. Do the split before any operation that learns from data.
Target leakage	A feature is derived from the target or only available after the event to be predicted.	Remove features that 'see the future'. Analyse each variable and ask: is this information available at the time of prediction?
Target source variable in the dataset	You created the target from score≥7, but score is still in X.	Immediately remove the source variable after creating the target.
Scaler outside CV	You fit StandardScaler on X_train, then run CV: the validation folds have already seen the scaler.	Put the scaler INSIDE the Pipeline passed to cross_val_score or GridSearchCV.
Resampling before split or outside Pipeline	You apply SMOTE on the entire dataset or on X_train before CV.	Apply resampling only to the training set. If using SMOTE in CV, put it in the Pipeline (use imblearn.pipeline).

Appendix B — Final checklist before submission

Before submitting the project, verify each point:

Dataset and description

- □ The dataset is described with context, source, number of observations and features.
- □ Each feature is classified as continuous numerical / nominal categorical / ordinal categorical / binary.
- □ The target is described (classes, meaning, encoding).
- □ If the target was created from a continuous variable, the source variable has been removed.

EDA

- □ Missing values have been analysed column by column.
- □ Duplicates have been checked and removed if present.
- □ The target distribution has been visualised and commented on quantitatively.
- □ The distributions of the main numerical variables have been visualised.
- □ The correlation matrix has been computed only on continuous numerical variables.

Preprocessing and split

- □ The split occurred BEFORE scaling, encoding and imputation.
- □ Binary variables have been mapped to 0/1 before the split with deterministic mapping.
- □ Categorical variables with 3+ categories are encoded with OneHotEncoder INSIDE the Pipeline.
- □ StandardScaler is applied only to continuous numerical variables.
- □ Binary variables are not standardised.
- □ train_test_split uses stratify=y.

Models and tuning

- □ At least 3 different models have been compared with CV on the training set.
- □ The comparison uses the same metric for all models.
- □ Hyperparameter optimisation (GridSearchCV) uses CV on the training set.
- □ GridSearchCV is configured with refit=True.
- □ No evaluation on the test set occurred during comparison or tuning.

Final evaluation

- □ The test set has been evaluated ONLY ONCE with the final model.
- □ Training and test metrics are compared to verify overfitting/underfitting.
- □ The confusion matrix is commented in the context of the problem.
- □ The classification report is read and commented on, not just printed.

Feature importance and conclusions

- □ Feature importance is computed and visualised.
- □ Feature importance results are interpreted in the biological/clinical context.

- □ Conclusions include: workflow summary, problems encountered, final performance, interpretation, limitations, future developments.

Markdown and comments

- □ Every code cell is preceded or followed by a markdown cell explaining WHAT is being done and WHY.
- □ Every methodological choice is justified (e.g. why OHE instead of LabelEncoder? Why random forest instead of KNN? Why the chosen threshold for the target?).
- □ Every output (graph, table, metrics) is commented: it is not enough to display it, you must say what you observe.

Good luck!