Technical Report Template & Guidance

for Submission to the

Chair of Financial Economics & Risk Management

Student Name (ID: 123456)

July 31, 2025

LaTeX Template Recommendation & KIT Presentation Guidelines

For consistency with KIT formatting standards, we recommend using the official KSP LaTeX class file (`ksp.cls`) and the SDQ PowerPoint template for final presentations.

More details can be found at:
`https://www.bibliothek.kit.edu/ksp-toolbox-dokumentvorlagen.php`
`https://sdq.kastel.kit.edu/wiki/Dokumentvorlagen`

Contents

Karlsruher Institut für Technologie

Report Title Placeholder

Machine Learning in Finance
BSc / MSc / PhD Programme

Author: Student Name
Matriculation No.: 123456
Submission: 2025-07-31
Supervisor: *Name*
Course / Module: *Name / Code*

Git Repository: `https://github.com/username/project`

Abstract

A concise summary of the problem, technical approach, key quantitative results, and main conclusions. Avoid undefined acronyms; write for a technically literate but non-expert reader.

1 Introduction

1. **Problem Statement & Motivation**
2. **Objectives**
3. **Background & Related Work**
4. **Contributions**

2 Mathematical Formulation (replace with your own)

Present the governing equations of your method. Each equation must be numbered, and every symbol must be introduced immediately after the equation. Use a short `description` list as shown in the loss equation.

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N}\Big[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)\Big] \tag{1}$$

$\mathcal{L}$ Binary cross-entropy loss.

θ Trainable model parameters.

N Number of training samples.

y_i True label of sample i.

$\hat{y}_i$ Predicted probability for sample i.

3 Use-Case Scenario (replace with your own)

Briefly describe the application context, business motivation and the key performance indicators (KPIs).

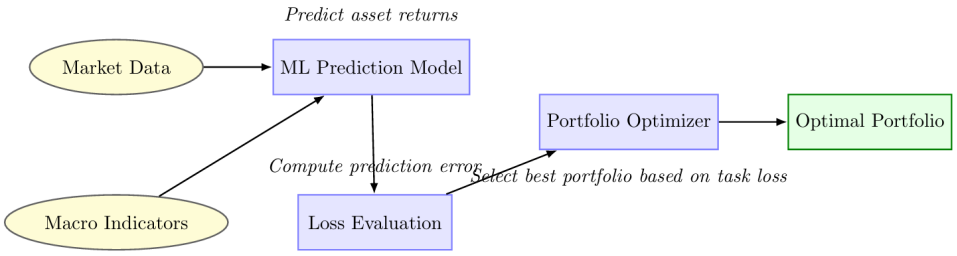

Figure 1: Illustrative use-case diagram.

4 System Architecture (replace with your own)

Provide a top-level architecture highlighting data sources, processing layers, and deployment targets.

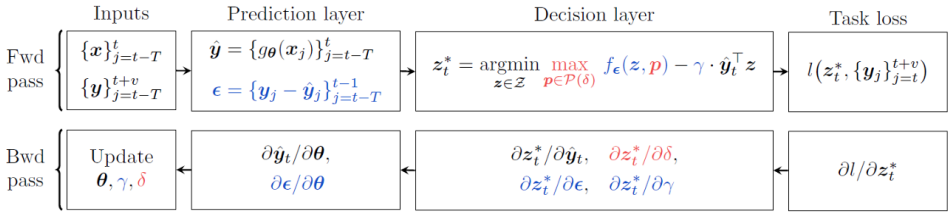

Figure 2: High-level system architecture.

5 Unified Layered Model (ULM) (replace with your own)

If applicable, map the system into *Data Layer*, *Model Layer*, and *Service Layer*.

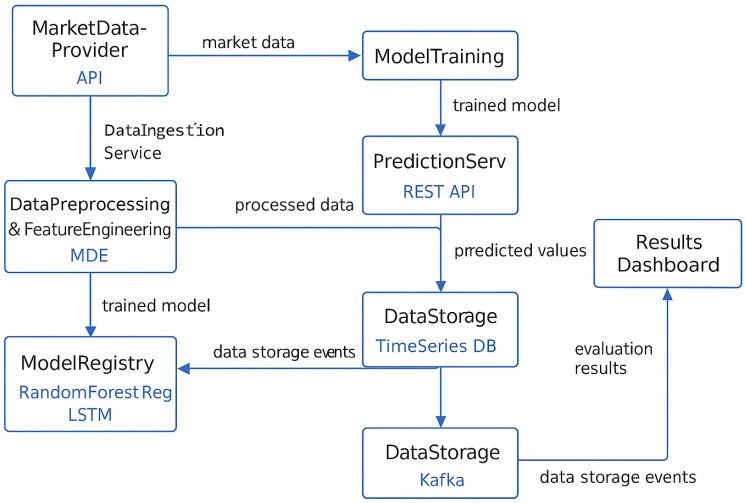

Figure 3: Unified Layered Model.

6 Algorithm Specification & Pseudocode

6.1 Plain-Language Overview

Summarize the core idea of the algorithm and outline why it is suitable for the stated problem.

6.2 Pseudocode (replace with your own)

Provide your algorithm in readable pseudocode form using the ‘algorithm2e‘ environment. Every variable must be defined in-line or in a short caption, and the algorithm should be referenced in the text as an algorithm.

Algorithm 1: Mini-batch Gradient Descent (example)

Input: Initial parameters $\theta^{(0)}$, learning rate η, batch size B, dataset $\mathcal{D}$
Output: Optimised parameters $\theta^{(*)}$
for $t \leftarrow 0$ **to** $T - 1$ **do**
  Sample mini-batch $\mathcal{B} \subset \mathcal{D}$ of size B;
  Compute gradient $g \leftarrow \nabla_\theta \mathcal{L}(\theta^{(t)}; \mathcal{B})$;
  Update parameters $\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta g$;
return $\theta^{(T)}$

6.3 Reference Implementation

Link to the corresponding function in your codebase (e.g., `src/optim/gradient.py`).

6.4 Complexity Analysis

Time complexity $\mathcal{O}(T)$; memory complexity $\mathcal{O}(S)$.

6.5 Executable Example

If helpful, you may additionally include a short code snippet illustrating the API usage. For instance:

Listing 1: Training loop snippet

```
for episode in range(num_episodes):
    state = env.reset()
    while True:
        action = policy(state)
        next_state, reward, done, _ = env.step(action)
        update_policy(state, action, reward, next_state)
        state = next_state
        if done:
            break
```

7 Experimental Setup

This section outlines the datasets used, preprocessing steps, training environment, and hyper-parameter search strategy. All elements are tracked and versioned to ensure full reproducibility.

Key configuration and dependency files include:

- `environment.yml` – Specifies the exact Python version and required libraries (including non-Python dependencies) using a Conda environment.
- `requirements.txt` – Lists all Python packages needed to recreate the environment with `pip`.
- `README.md` – Provides project documentation, setup instructions, and usage examples.

7.1 Hyper-Parameter Search Protocol (replace with your own)

Search protocol. To explore model capacity while guarding against over-fitting, we perform a ***<search-type>*** hyper-parameter optimisation using ***<library/tool>*** under a fixed random seed (*<seed-value>*).[1] All trial configurations, objective scores, and diagnostic metrics are logged to ***<experiment-tracking platform>*** to guarantee full reproducibility and auditability. Each optimisation run is capped at *<N>* trials, with *<early-stopping rule>* applied to terminate unproductive searches.

Table 1: Hyper-parameter search space and initial values (*replace italics with your own settings*).

Hyper-parameter	Symbol	Search space	Strategy	Init. value
Learning rate	η	$[10^{-5}, 10^{-1}]$ *(log-uniform)*	*<strategy>*	η_0
Batch size	B	*{32, 64, 128, 256}*	*<strategy>*	B_0
Weight decay	λ	$[0, 10^{-2}]$ *(linear)*	*<strategy>*	λ_0
Drop-out rate	p	[0.0, 0.5] *(uniform)*	*<strategy>*	p_0
Hidden units	h	*{64, 128, 256, 512}*	*<strategy>*	h_0

Random seeds for each trial are drawn from the project-wide set {42, 1337, 2025} to maintain consistent initialisation while still sampling multiple configurations.

Early-stopping criterion. Optimisation terminates when *<k>* consecutive validation checkpoints fail to improve the primary objective by at least *<tolerance>*.

Data Cleaning and Feature Preprocessing. We apply the following deterministic preprocessing steps after data ingestion. These are implemented as a tracked `pandas` pipeline to ensure full reproducibility:

- **Missing-target removal** – Discard records without a prediction target ($n = 184$).
- **Winsorization** – Clip numeric features to the 0.5^{th} and 99.5^{th} percentiles to mitigate outliers.
- **Rare-category grouping** – Collapse categorical levels with <0.1% frequency into a common "other" label.
- **Standardization** – Scale numerical features to zero mean and unit variance, using statistics computed on the training set only.

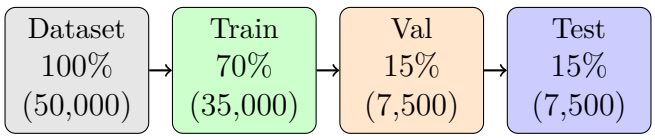

Figure 4: Dataset split into training, validation, and test sets.

[1] Replace with the actual seed or strategy you used— e.g. `seed=42`.

8 Results & Visualisation

Present quantitative results (tables) and qualitative results (plots).

8.1 Formatting Conventions

- Caption figures as **Figure X.Y** and tables as **Table X.Y**, where X is the current section number and Y is the item's sequence within that section.
- Always reference visuals with `\autoref{}` — e.g., "As shown in Figure 2, ...".

For additional guidelines, see `https://www.bibliothek.kit.edu/downloads/KSP/KSP-Latex-Tipps.pdf`.

9 Evaluation Metrics

Authors must include the evaluation metric(s) that best fit their specific project. The table and formulas below are provided solely as illustrative examples. We bring the following part as an example.

9.1 Regression Metrics

Table 2: Common regression metrics.

Metric	Formula	Range	Interpretation
MSE	mse	$[0, \infty)$	Lower is better
RMSE	rmse	$[0, \infty)$	Lower is better
MAE	made	$[0, \infty)$	Lower is better
R^2	r2	$(-\infty, 1]$	Closer to 1 better
Adj. R^2	adjr2	$(-\infty, 1]$	Penalises extra variables

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

$$\text{RMSE} = \sqrt{\text{MSE}} \tag{3}$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4}$$

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2} \tag{5}$$

$$R^2_{\text{adj}} = 1 - (1 - R^2)\frac{n-1}{n-p-1} \tag{6}$$

10 Discussion

Interpret the results, discuss potential biases and threats to validity, and outline avenues for future work.

11 Reproducibility Checklist

- Code and data **publicly available** (preferably under an open-source licence).
- Exact library versions and random seeds **reported**.
- Script to reproduce main results within ~10 %.
- Docker/Conda environment builds and **passes all tests**.
- Exact environment snapshot provided in `environment.yml` and `requirements.txt`; see `env/` folder for full `pip freeze`. Files are duplicated verbatim in Code Appendix A so the PDF is self-contained.

A raw text version of the dependency list is available at `alg_code/env/requirements.txt`.[2]

12 Conclusion

Summarize key findings, contributions, and impacts. Keep it short (1 page).

References – Formatting Note

Citations follow the author–year style using the `biblatex` package with the 'authoryear' option:

```
\usepackage[style=authoryear]{biblatex}
```

For direct quotes, please include page numbers and avoid using `\cite{}` without them.

For further guidance, refer to the official documentation at: `https://www.overleaf.com/learn/latex/Biblatex_bibliography_styles`

References

A Code Appendix

This section shows exactly **what** is in the submission repository, **how** it implements Algorithm1 step-for-step, and **how** you can rerun it on a standard workstation (CPU or singleGPU).

Repository Contents

The repository contains the following main components:

- `alg_main.py`: Canonical, line-for-line implementation of Algorithm 1.
- `alg_utils.py`: Reusable utilities for sampling, logging, and I/O; each function is cross-referenced to its pseudocode line (e.g., Alg. 1, L9).

[2] Replace the URL with the actual repository link before submission.

- `configs/`: Version-controlled YAML files that specify every hyperparameter and random seed used in the Section 4.2 experiments.
- `notebooks/`: Jupyter notebooks for exploratory data analysis and figure generation (read-only with respect to the main results).
- `tests/`: Lightweight fixtures and `pytest` scripts that comprise the unit-test suite (see Appendix B).
- `env/`: Frozen dependency snapshots: `environment.yml`, `requirements.txt`, and the full `pip freeze` log; included here for a self-contained PDF record.

The following guarantees apply to all released code:

- **1:1 implementation guarantee:** All code uses the exact same symbols, variable names, and step order as Algorithm 1—no deviations.
- **No hidden tricks:** There are no extra heuristics, undocumented hyperparameters, or post-processing steps beyond what's described in the paper.
- **Version tag:** The commit that produced Table 2 is tagged `v1.0-submission`.

Reproducibility instructions

Tested on Python3.11 with CUDA12.3:

```
git clone https://github.com/<your-org>/<repo>.git
cd <repo>
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
python alg_main.py --config configs/baseline.yaml
```

You should reproduce the §4.2 metrics within ±0.1%.

B Unit Tests

Unit tests act as a *safety net*: they run in seconds, catch bugs early, and ensure the code continues to match Algorithm 1 after any change.
To run the tests, use the following command:

- `pytest -q` (required for Python code)

Each test performs the following:

1. Loads a **small, representative dataset** (approximately 1MB) designed to exercise every code branch in under 3 seconds.
2. Runs the full module end-to-end.
3. Asserts that key outputs—metrics, tensor shapes, and data types—match pre-computed reference values.

Category	Purpose (for newcomers)
Fixture dataset	A minimal dataset to ensure all code paths run quickly (<3s).
Smoke test	Verifies each CLI command executes without errors and writes output.
Deterministic	Checks that final loss/metric matches the saved result within 10^{-6}.
Shape/type	Confirms tensors have the expected dimensions (e.g. $B \times C \times H \times W$) and `dtype`s.
Gradient	Performs a finite-difference check on $\nabla_\theta \mathcal{L}$ in `alg_utils.backward_pass`.

Table 3: Overview of the unit-test suite.

Test categories

CI pass criteria

- **All tests green.** Any non-zero exit code from `pytest` (or `ctest`) fails the build.
- **Coverage 90%.** We track line coverage with `pytest -cov`; if it drops below 90%, the CI blocks merging until additional tests are added.

With this setup, even first-year undergraduates can make changes, rerun `pytest`, and immediately see whether any functionality broke—no deep ML expertise needed.

Unit tests act as a *safety net*: they run in seconds, catch bugs early, and ensure the code continues to match the equations above after any change.

This appendix shows how to run a simple ordinary least squares (OLS) regression on a mini financial dataset and how to wrap the code with a fast, pedagogical unit-test suite. Copy these files into an Overleaf project (or clone the repository) and hit `pytest -q`—all tests finish in under three seconds on a laptop.

Worked Example – CAPM β via OLS

Listing 2: Minimal module `regression.py`.

```
"""CAPM β estimation on a tiny Yahoo Finance fixture."""
import yfinance as yf
import pandas as pd
import statsmodels.api as sm

def _returns(ticker: str, start: str, end: str) -> pd.Series:
    """Download daily log-returns for <ticker>."""
    px = yf.download(ticker, start=start, end=end, progress=False
       )["Adj Close"]
    return px.pct_change().dropna()  # approx 60 rows -> <1 MB

def fit_capm(asset="AAPL", market="SPY",
             start="2024-01-01", end="2024-03-31"):
    r_asset = alpha + beta * r_market + epsilon //OLS
    r_a = _returns(asset, start, end)
    r_m = _returns(market, start, end).reindex(r_a.index)
    X    = sm.add_constant(r_m.values)
    model = sm.OLS(r_a.values, X).fit()
    return model
```

Listing 3: PyTest file `tests/test_regression.py`.

```
"""Unit tests for CAPM example."""
from regression import fit_capm
import numpy as np

def test_beta_positive():
    res = fit_capm()
    beta = res.params[1]
    # Large-cap tech stocks usually have β > 0
    assert beta > 0

def test_rsquared_bounds():
    res = fit_capm()
    assert 0.0 <= res.rsquared <= 1.0 + 1e-12
```

How to run.

```
# one-time setup
pip install yfinance statsmodels pandas pytest

# execute the full suite (including the example above)
pytest -q
```

How to run the example

The example meets every criterion in Table B: a **fixture dataset** under 1 MB, a **smoke-test-level run time** (< 3 s), and **deterministic tests** on both the CAPM β and R^2.

Students can modify the tickers or date range, rerun `pytest`, and instantly see whether their changes still pass—mirroring the workflow they'll use for your main algorithm.

C Authorship Contribution Matrix

The following matrix outlines the specific contributions of each author to various aspects of the project. A checkmark (✓) indicates primary responsibility, while percentages represent shared efforts.

Author	**Conceptualisation**	**Code**	**Experiments**	**Writing**	**Review**
Main Author	✓	✓	50%	✓	✓
Collaborator A	✓		50%		✓
Collaborator B		✓		✓	

Table 4: Individual contributions to the project. Cells marked with ✓ denote primary responsibility; percentages indicate shared work.