

Python Introduction - Session 3 Exercises

ISEG Executive Education - PG AAIML

Diogo Silva

Table of contents

Session 3: U8 – U13	2
Exercise 29: Feature Transform Library	2
Exercise 30: Feature Selection Pipeline	2
Exercise 31: Data Augmentation Generator	3
Exercise 32: Cross-Validation Split Generator	3
Exercise 33: Model Evaluation Module	4
Exercise 34: Investment analyst – files	4
Exercise 35: Investment analyst – modules	4
Exercise 36: Boston Housing Dataset - Data Normalization	4
Exercise 37: Experiment Logger	5
Exercise 38: Dataset File Processor	5
Exercise 39: Training Configuration Manager	6
Exercise 40: Ensemble Results Combiner	6
Exercise 41: Model Checkpoint Manager	9
Background	9
Provided Code	9
Your Tasks	10
Expected Output	11
Exercise 42: Investment analyst – catching errors	11
Exercise 43: Investment analyst – debugging with colleagues	11

Tips:

- Start by understanding what the exercise asks
- Break the problem into smaller steps
- Test your code with the provided examples
- Try variations of the solutions
- Don't hesitate to combine techniques you've learned
- For ML exercises: reflect on why these operations matter in real world projects

Each exercise lists the course units it practises. Exercises named *Investment analyst* build on each other around the same problem, from session 1 to session 3.

Session 3: U8 – U13

In class: Exercise 41 and Exercise 42. Exercise 36 is walked through as a demo, including exception handling.

Unit	Recommended
U8 map and filter	29, 30
U9 Modules and packages	33, 35
U10 Files	37, 41
U11 Exceptions	42, 39
U12 Classes	41 (uses a class), 39 (extra challenge)
U13 Debugging	43

Tips for this session:

- Always use context managers (`with` statement) when working with files
- Use `pathlib.Path` instead of string manipulation for file paths
- Keep functions focused on a single task (single responsibility principle)
- Use meaningful function and parameter names
- Test edge cases (empty lists, missing files, invalid inputs)
- Remember that `map()` returns an iterator - convert to list if needed
- When creating modules, think about code organization and reusability

Exercise 29: Feature Transform Library

Units: U7, U8, U9

Create a module `transformers.py` with transformation functions and use `map` to apply them:

1. Function `log_transform(x)` - applies logarithmic transformation (handle $x \leq 0$)
2. Function `square_transform(x)` - squares the value
3. Function `clip_outliers(x, lower, upper)` - clips values outside the range
4. Function `apply_transform(values, transform_func)` - uses `map()` to apply any transformation to a list. Note that what is being passed in the second argument is the actual function.

Test with: `[1, 5, 10, 50, 100, 500, 1000]`

Exercise 30: Feature Selection Pipeline

Units: U8, U10

Create functions to select the most informative features:

1. Function `calculate_variance(values)` - calculates variance of a feature
2. Function `select_high_variance_features(data_dict, threshold)` - `data_dict` has feature names as keys and lists of values as values. Use `filter()` to keep only features with variance above threshold

3. Function `save_selected_features(selected_features, filepath)` - saves the list of selected feature names to a file
4. Function `load_feature_config(filepath)` - loads the feature names from the file
5. Function `apply_feature_selection(data_dict, feature_names)` - returns a new dict with only the selected features

Test with `BostonHousing.csv` dataset ([link](#)).

Reflection: What threshold should you pick? Should some normalization be applied first?

Exercise 31: Data Augmentation Generator

Units: U8, U10

Create functions that generate augmented versions of data:

1. Function `add_noise(value, noise_level=0.1)` - adds Gaussian noise to a value (simulate by adding/subtracting `noise_level * value`).
2. Function `scale_value(value, scale_factor)` - scales a value by a factor
3. Function `augment_sample(sample, augmentation_funcs)` - takes a list of values and a list of augmentation functions, applies all functions
4. Use `map()` to create augmented versions of an entire dataset
5. Function `save_augmented_data(original, augmented, filepath)` - saves both original and augmented data to a file with clear labels

Test with: `[1.0, 2.5, 3.7, 4.2, 5.8]` and create 2 augmented versions.

Added challenge: augment the `BostonHousing.csv` dataset ([link](#)) and use column-wise variance for the random noise augmentation;

Exercise 32: Cross-Validation Split Generator

Units: U8, U9, U10

Create a module for generating cross-validation splits using k-fold:

1. Function `create_folds(data_indices, k=5)` - splits indices into k approximately equal folds and returns a list of folds
2. Function `get_train_test_split(folds, test_fold_index)` - given the folds and which fold to use as test, returns train and test indices
3. Function `save_cv_splits(folds, directory)` - saves each fold to a separate file (`fold_0.txt`, `fold_1.txt`, etc.) using `pathlib`
4. Function `load_cv_splits(directory)` - loads all fold files from the directory and returns them as a list
5. Write a function that uses `map()` to apply a validation function to all k folds

Test with 50 sample indices and k=5 folds.

Reflection: imagine you have an imbalanced dataset (class wise) - would you change anything when creating the folds?

Exercise 33: Model Evaluation Module

Units: U7, U9

Create a Python module called `ml_metrics.py` with functions to calculate classification metrics:

1. Function `accuracy(y_true, y_pred)` - returns accuracy score
2. Function `precision(y_true, y_pred)` - returns precision for class 1
3. Function `recall(y_true, y_pred)` - returns recall for class 1
4. Function `f1_score(y_true, y_pred)` - returns F1 score (harmonic mean of precision and recall)
5. Function `evaluate_model(y_true, y_pred)` - returns a dictionary with all metrics

In a separate notebook or script, import your module and test it with: - `y_true = [1, 0, 1, 1, 0, 1, 0, 0, 1, 0]` - `y_pred = [1, 0, 1, 0, 0, 1, 1, 0, 1, 0]`

Exercise 34: Investment analyst – files

Units: U10

- Redo the previous exercise to load the parties and their details from a file.

Exercise 35: Investment analyst – modules

Units: U9, U10

- Redo the previous exercise and make a module with the functions for the individual simulations and the Monte Carlo simulation in a module.
- On another module, add the functions to load the data and returned it in a format that the simulation functions can use.
- Modify the original program to implement the full Monte Carlo simulation using the functions from the module.

Exercise 36: Boston Housing Dataset - Data Normalization

Units: U7, U10

In this exercise, you will work with a real dataset - the well known Boston House Prices dataset. You can download it from here or find it easily online (e.g. Kaggle). The exercise will require you to program the data loading functionality, as well as some basic normalization routines. Don't use any external package, like Pandas, as that has all these features built-in.

- **Data loading**
 - Create a function that will load the dataset, in Comma Separated Values (CSV) format, and returns a `dict` with pairs of {column, list of values}.
 - The name of each column is specified in the first line of the CSV file.
 - Convert each value to float.
- **Scaling**
 - Write a function `mean(values)` that computes the mean of a list of values.
 - Write a function `std(values)` that computes the standard deviation of a list of numbers.
 - Output the mean and std of every column.

- Write a function `min_max_scale(values)` that normalizes a list of numbers to the 0-1 range
- Write a function `standardize(values)` that standardizes data (subtract mean, divide by standard deviation)
- Write a function `apply_scaling(data, method)` that takes a list and a method name (“minmax” or “standard”) and applies the appropriate scaling
- Use `apply_scaling` with both methods to an arbitrary column and compare results.
- **Data card** (capitalized column name, followed by description)
 - CRIM per capita crime rate by town
 - ZN proportion of residential land zoned for lots over 25,000 sq.ft.
 - INDUS proportion of non-retail business acres per town
 - CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
 - NOX nitric oxides concentration (parts per 10 million)
 - RM average number of rooms per dwelling
 - AGE proportion of owner-occupied units built prior to 1940
 - DIS weighted distances to five Boston employment centres
 - RAD index of accessibility to radial highways
 - TAX full-value property-tax rate per \$10,000
 - PTRATIO pupil-teacher ratio by town
 - $B 1000(B_k - 0.63)^2$ where B_k is the proportion of blacks by town
 - LSTAT % lower status of the population
 - MEDV Median value of owner-occupied homes in \$1000's

Exercise 37: Experiment Logger

Units: U7, U10

Create a system that logs ML experiments to a text file:

1. Write a function `log_experiment(model_name, hyperparameters, metrics, filename="experiments.txt")` that appends experiment details to a file
2. The log entry should include: timestamp (you can use a simple counter), model name, hyperparameters (as dict), and metrics (as dict)
3. Write a function `read_experiments(filename="experiments.txt")` that reads and prints all logged experiments
4. Test by logging 3 different experiments with different models and hyperparameters
5. Use context managers for file operations

Exercise 38: Dataset File Processor

Units: U10

You have multiple CSV files with training data from different sources. Create functions to process them:

1. Write a function `load_dataset(filepath)` that reads a CSV file and returns the data as a list of dictionaries (each row is a dict).

2. Write a function `filter_by_threshold(data, column, threshold)` that filters rows where a column value exceeds a threshold
3. Write a function `save_processed_data(data, output_path)` that saves the processed data to a new CSV file
4. Use `pathlib.Path` for file operations
5. Create a function `process_pipeline([input_file1, input_file2], output_file, column, threshold)` that combines all steps, but also accepts a list of input CSV files instead of just one. The data from those files must be combined before being fed to `filter_by_threshold` and `save_processed_data`.

Split the `BostonHousing.csv` (link) dataset into 3 different files manually, e.g. first 100 rows for file 1, rows 100-200 for file 2, and the remaining rows for file 3.

Exercise 39: Training Configuration Manager

Units: U9, U10, U11

Create a system to manage training configurations:

1. Function `create_config(learning_rate, batch_size, epochs, optimizer)` - returns a configuration dictionary
2. Function `save_config(config, filepath)` - saves the configuration to a text file in a readable format (key: value pairs)
3. Function `load_config(filepath)` - loads configuration from file and returns it as a dictionary
4. Function `validate_config(config)` - checks that all required keys exist and values are within reasonable ranges
5. Create a package structure with separate modules for config management and validation

Test by creating, saving, loading, and validating different configurations.

Tip: use the `json` module.

`validate_config` should raise a `ValueError` with an informative message when the configuration is invalid (U11).

Extra challenge (U12): implement the configuration as a `TrainingConfig` class with `save`, `load` and `validate` methods.

Exercise 40: Ensemble Results Combiner

Units: U10

Create functions that combine predictions from multiple models (stored in separate files) using different strategies: majority voting for classification, averaging for regression.

Added challenge: for classification, use the estimated confidence alongside the actual prediction for refining the final ensemble result.

Data:

```

# Mock Classification Predictions (3 models predicting classes 0, 1, 2)
modell_classification = """sample_id,prediction,confidence
0,1,0.85
1,0,0.92
2,2,0.78
3,1,0.88
4,0,0.95
5,2,0.71
6,1,0.83
7,0,0.89
8,2,0.76
9,1,0.91"""

model2_classification = """sample_id,prediction,confidence
0,1,0.82
1,0,0.88
2,1,0.65
3,1,0.90
4,0,0.93
5,2,0.68
6,1,0.86
7,1,0.72
8,2,0.81
9,1,0.87"""

model3_classification = """sample_id,prediction,confidence
0,1,0.79
1,0,0.91
2,2,0.73
3,0,0.67
4,0,0.96
5,1,0.62
6,1,0.84
7,0,0.85
8,2,0.79
9,1,0.89"""

# Model 1 Regression Predictions
modell_regression = """sample_id,prediction
0,42.5
1,38.2
2,55.7

```

```

3,61.3
4,29.8
5,48.9
6,52.1
7,45.6
8,33.4
9,58.2"""

# Model 2 Regression Predictions
model2_regression = """sample_id,prediction
0,43.1
1,37.8
2,54.9
3,62.5
4,30.2
5,49.5
6,51.8
7,46.2
8,34.1
9,57.8"""

# Model 3 Regression Predictions
model3_regression = """sample_id,prediction
0,42.8
1,38.5
2,56.2
3,60.9
4,29.5
5,48.2
6,52.5
7,45.1
8,33.8
9,58.9"""

ground_truth_classification = """sample_id,true_label
0,1
1,0
2,2
3,1
4,0
5,2
6,1

```

```

7,0
8,2
9,1"""

ground_truth_regression = """sample_id,true_value
0,42.7
1,38.0
2,55.5
3,61.5
4,30.0
5,49.0
6,52.0
7,45.5
8,33.5
9,58.5"""

```

Exercise 41: Model Checkpoint Manager

Units: U10, U12

#In machine learning, it's crucial to save model states during training so you can:

- Resume training if it gets interrupted
- Compare performance across different epochs
- Select the best model based on validation metrics
- Detect and analyze overfitting

In this exercise, you'll implement a checkpoint management system that saves model states during training and helps identify overfitting patterns.

Background

You are provided with a `MockModel` class in `mockmodel.py` that simulates realistic neural network training. The model exhibits a common problem in deep learning: **overfitting**. Overfitting generally happens when the training accuracy continues to improve, but validation accuracy starts to decrease, indicating the model is memorizing the training data rather than learning generalizable patterns.

Your task is to implement a checkpoint system to track performance metrics (loss and accuracy), and store checkpoints of model parameters, during the training procedure.

Provided Code

- The `mockmodel` module with the `MockModel` class is provided. This class has the following relevant methods:
 - `get_model_state()` - Returns the current model weights/parameters
 - `train_eval_model_iteration()` - Simulates one epoch and returns `(train_acc, train_loss, val_acc, val_loss)`

- **Basic training loop** that runs for 80 epochs
- Download the module with the mock model from this link: [mockmodel.py](#)
- Download the starter code with the basic training loop from this link: [startercode.py](#)

Your Tasks

1. Implement Checkpoint Saving Function

Write a function that saves model checkpoints to disk:

- **Input:** Current epoch number, model state (from `model.get_model_state()`), and metrics dictionary
- **Format:** Should you save in text or binary format? Consider what you're storing (numpy arrays)
- **Filename:** Design a unique naming scheme for each checkpoint (hint: include the epoch number)
- **Directory:** Save all checkpoints to a directory called `"model_checkpoints"`
- **Path handling:** Use `pathlib.Path` to create the directory if it doesn't exist
 - Look up the `makedirs()` method and understand what `parents` and `exist_ok` arguments do
 - Reference: <https://docs.python.org/3/library/pathlib.html>

2. Save Checkpoints During Training

Modify the training loop to save checkpoints **every 5 epochs**.

3. Post-Training Analysis

After training completes, implement the following analyses:

3.1 Display All Checkpoints

Print the training and validation accuracies of all saved checkpoints, ordered by epoch number.

3.2 Load Best Checkpoint

Load the checkpoint with the best validation accuracy and print:

- Which epoch it was from
- Its validation and training accuracies

3.3 Detect Overfitting Point

Write code to automatically detect around which epoch the validation accuracy started decreasing (overfitting began).

Hints for detecting overfitting:

- Compare consecutive validation accuracies
- Look for when validation accuracy peaks and then declines
- Consider using a window of epochs to avoid noise

4. Early stopping (Optional challenge)

Implement a **patience mechanism** inside the training loop:

- Track if validation accuracy hasn't improved for 5 consecutive epochs
- Stop training early if this condition is met
- Print a message indicating early stopping was triggered and at which epoch

5. Spaced checkpoints (Optional challenge)

Instead of saving a checkpoint at every epoch, it's common to save at every N epochs (e.g. N=10) and also keep a checkpoint of the model parameters with the best validation accuracy so far. Change the training loop to reflect this idea.

Expected Output

Your implementation should:

1. Create a `model_checkpoints/` directory
2. Save checkpoint files
3. Display all checkpoint metrics after training
4. Identify the best checkpoint
5. Detect when overfitting started
6. (Optional) Stop early if patience limit is reached
7. (Optional) Implement spaced checkpoints

Exercise 42: Investment analyst – catching errors

Units: U11

- Following the Monte Carlo simulation exercises, add error handling to the functions.
- Make sure to handle errors such as incorrect file paths, incorrect file formats, and invalid user inputs, e.g. negative investment values, negative standard deviations, non numeric values, etc.
- Use try-except blocks to catch errors and provide informative messages to the user.

Exercise 43: Investment analyst – debugging with colleagues

Units: U13

- Use your implementation from the previous exercises and introduce bugs in the code.
- Share the code with a colleague and ask them to debug it (don't tell them what is the bug).
- Analogously, find and fix bugs in your colleague's code.