# MultiValue Bookstore

# Python Sample 1

Recommendation Engine

## Introduction

The Bookstore development team received the following user story into their backlog:

> **As a** Sales Director
> **I want to** improve our recommendations
> **So that** we can realize more cross-selling opportunities.

After speaking with the director, the team decided to adopt a collaborative filtering approach similar to that used by other leading resellers (people who bought ‘x’ also bought ‘y’).

This would need to have the following features:

- A recommendation would be based on a publication and optionally a client.
- The recommendations would look at who had bought that publication, and what other publications those same customers had purchased.
- Repeat orders would not be counted.
- The recommendations would initially be ranked on popularity based on numbers. It is assumed (naively) that all publications have been available for the same length of time.
- Weightings would then be applied as follows:
    - For the same author a 3x weighting would be applied.
    - For the same genre a 2x weighting would be applied.
- Finally, if a client was specified for the recommendation, any books they had already purchased would be filtered from the recommendation.
- Only the top 10 recommendations would be returned along with their rankings, in the order of the most popular to the least.

# History

## 1. Proof of Concept

**books.bp u2_recommender_v1**

In order to understand the requirements, a first attempt was made to build this using regular MultiValue techniques and the business language, by means of a Proof of Concept.

This created pairing records by examining the orders and writing combinations of titles purchased together to a U2_PAIRS file. A new U2_CLIENT_BOOKS file also kept track of which clients had purchased which titles, to speed up the lookups and to filter out purchases by the client to whom the recommendations were being shown. This could have been fulfilled using the existing U2_CLIENT_ORDERS but that would have slowed the retrieval of recommendations.

This proved workable in providing meaningful recommendations, but was slow in building and the team decided it could be improved upon.

See the test script: RECOMMENDER_1

## 2. First iteration

**books.bp u2_recommender_v2**

After meetings with the team to review the Proof of Concept, it was decided that the design of the U2_PAIRS and U2_CLIENT_BOOKS were potential bottlenecks to the process. Whilst the recommendations were correct, it would be better to cache these details in memory following the initial build to improve performance.

One team member mentioned that the U2 Dynamic Objects (UDOs) supported by the business language provided a good model for this, as these flexible JSON-style structures are ably suited to holding the key/value pairs that can match books with clients and vice versa. The proof of concept was rewritten to store details using UDOs in place of new data files.

This proved to be much more efficient, but there were concerns about the maintainability and extensibility of this approach.

See the test script: RECOMMENDER_2

## 3. Second iteration

**books.pysrc u2_recommender_v3**
**books.bp u2_recommender_v3**

At this point, the team felt that they understood the problem domain and that the approach taken by the recommendation engine was correct. However, it would be better to develop this in a language that made the structure more apparent and that would be easier to extend in future.

Matthew, the new Python intern, suggested rewriting this as a Python routine. Python has the same data structures as UDOs but would be cleaner and simpler to maintain. It also had access to the same orders, books and client files as the business language.

The python routine would cache the recommendations along the same lines as the previous version. For this it would need to persist, and so Matthew suggested writing it as a service that could run in the background and that would be fed new sales orders as they were placed.

The team liked this approach, and another team member pointed out that the routine could run using the RUNPY command, as a UniVerse phantom process. All that was needed was a means for the two to communicate.

The team decided that since both UniVerse and Python have socket support and they could use that to call directly into the Python routine passing instructions and results in a JSON format. This could be called stand-alone from the website, whilst for backward compatibility an interfacing subroutine would keep the same API as the previous versions.

See the test script: RECOMMENDER_3.