ICT MyMedia Project 2008-215006

Engine Quickstart

17 June 2009

Public Document

1 Contents

Project acronym: MyMedia

Project full title: *Dynamic Personalization of Multimedia*

Work Package:	**2**

Document title:	**Engine Quickstart**
Version:	**N/A**
Official delivery date:	**N/A**
Actual publication date:	N/A
Type of document:	Open Source Software and Documentation
Nature:	Public

Authors:	Artus Krohn-Grimberghe, UHI
	Rich Hanbidge, EMIC

Approved by:	N/A

2 Target Audience

The intended target audience of this document are consumers of the MyMedia project software framework. Espcecially, this document focuses on developers intending to convert their own algorithms to the MyMedia platform as an Engine and field trial partners providing data to those algorithms. Developers consuming already existing Engines in their applications are referred to the companion document “MyMediaRecommenderQuickstart.docx”.

See the MyMedia softare license, included at the root of the distribution, for terms of use.

3 Introduction

For any software developer involved in writing recommender algorithms or consuming recommender algorithms already embedded in the MyMedia framework it is important to distinguish between two tasks: the task of using already present MyMedia recommender algorithms (“Engines”)—which is described in the companion document “MyMediaRecommenderQuickstart.docx”—and the task of adding new custom algorithms as Engines to the MyMedia framework. The latter task is described in the remainder of this document.

The main task for any recommender algorithm developer with respect to MyMedia is plugging in his algorithm into the MyMedia framework itself.

The MyMedia framework is split into two parts with respect to recommender algorithms. One part, the framework core, is represented by the interface IEngine which implicitly or explicitly is called by applications residing in the MyMedia framework.[1] The other part, called framework, has to be implemented by the algorithm developer / integrator. This second part is responsible for glueing together the algorithm, a data source and the core framework. The necessary work in this part of MyMedia is described below.

[1] Application developers shall look into the companion document mentioned above.

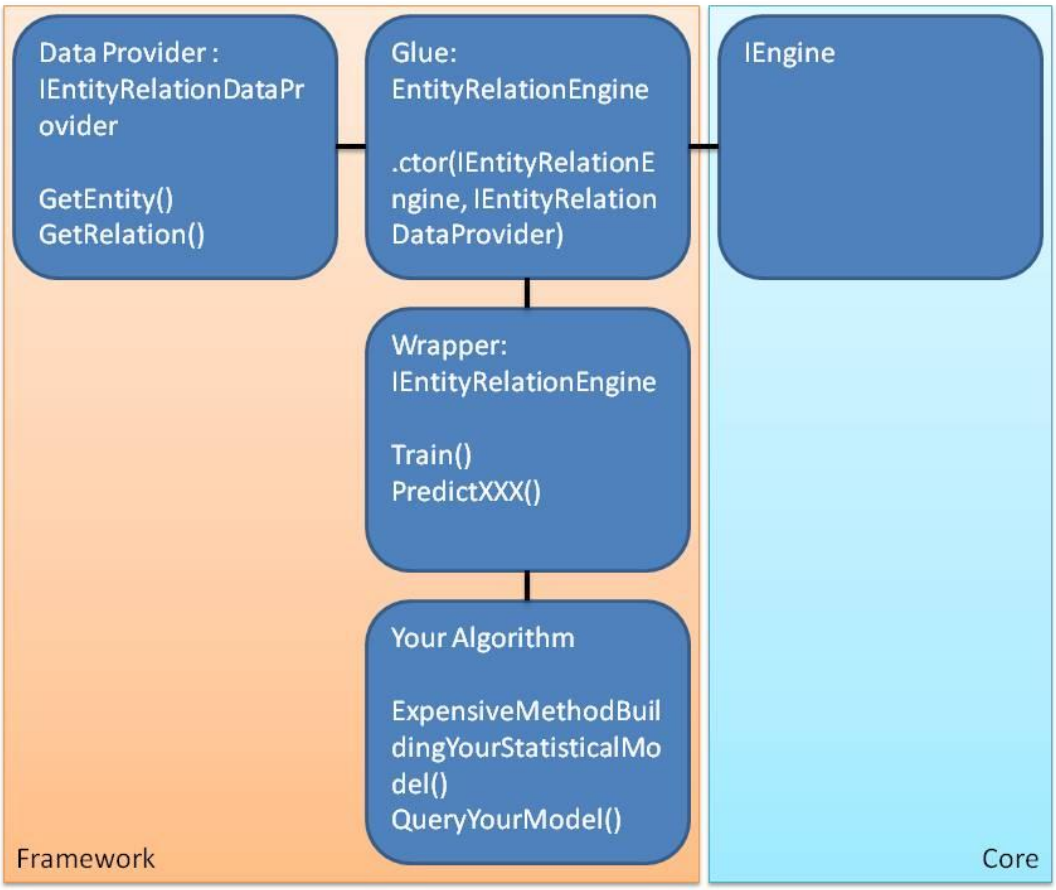

Figure 1: Recommender algorithm framework

Figure 1 summarizes the relationship between the components necessary to add new recommender algorithms to the MyMedia framework. On the right hand side are the components used by the comsumers of a MyMedia recommender algorithm. On the left hand side are the aspects the algorithm developer has implement before his code can be called via the IEngine interface: the data provider (usually written by the owner of the database) returning a MyMediaDataReader via either `GetEntity()` or `GetRelation()`; the algorithm itself; and the two necessary wrapper / glue classes around the algorithm.

4 Overview

For any algorithm to be implemented or ported against the MyMedia framework the most important starting point is the `MyMediaProject.RecommenderSystem.Framework.EntityRelationAlgorithm.EntityRelationEngine`, the common base class for all Engines that will be made available to its consumers / applications via the `MyMediaProject.RecommenderSystem.Core.Engine`.

From a conceptual point of view only the following steps need be followed:

1. Let the owner of the data you are learning your recommender algorithm model either extend the MyMedia data provider (`MyMediaEntityRelationDataProvider`) or get you a custom `IEntityRelationDataProvider`.
    a. Given the data you need for training your algorithm, clarify the necessary RelationType and EntityType (see below) your algorithm shall respond to
2. Implement your custom recommender algorithm in a way that it
    a. Gets all its training data at one point in time
    b. May spend a long time training on that data with the result of a compact and generalizable representation of the patterns learned
    c. Return this representation ("data mining model" or "statistical model") or an efficient way of querying it
3. Wrap your algorithm in a class that derives from the ***interface*** `IEntityRelationEngine`
    a. Make sure you implement all the necessary interfaces
    b. Call the expensive method building your data mining model during `Train()`
    c. Assign your algorithm a unique ID and name.
4. Wrap your class that derives from the interface `IEntityRelationEngine` into a class derived from the ***base class*** `EntityRelationEngine`. This class is the glue beween your algorithm encapsulated in the `IEntityRelationEngine` and the consumers of your algorithm calling the methods of `IEngine`.

Every single of the above mentioned four steps is described in one of the remaining sections of this document.

Please notice that your algorithm shall query the data provider only once – before building its data mining model. Further calls – especially at run- and querytime after this one-time training step – are highly discouraged.

5 About Entities and Relations

In order to model some of the semantics of the data presented to the algorithm, the concepts of an "Entity" and a "Relation" have been derived. In the rest of the document an Entity always refers to an object in the MyMedia framework application that can "exist on its own" like a user object or an item object. A Relation, however, always connects two or more Entities like a rating that, besides its rating value, has a connected user that gave the rating and a connected item that the rating of that user is about. Though arbitrary to some extend, this definition is implemented by the MyMedia interfaces and necessary to understand for successful development of algorithms in the framework.

6 How To Implement a Data Provider

Basically, the `IEntityRelationDataProvider` is a wrapper around an arbitrary data source, though in a MyMedia scenario the most common data source will be a database. The main tasks of the

`IEntityRelationDataProvider` are providing an abstraction over the base data source and returning an ADO.NET-like data reader back to the algorithm that can be traversed "row by row". The data provider developer shall turn his attention to fast delivery of large amounts of data as this interface is not intended to be queried of each recommendation given by the algorithm but rather only once during the inital training of the algorithm.

Usually, the field trial partner is responsible to adapt the data provider to his or her infrastructure. However, the Engine designer (algorithm developer) has to take care which data to request. As such, the Engine designer should provide an initial version of the respective Data Provider implemented against the core framework or a sample database he or she provides.

Depending on whether the data requested belongs to an Entity or a Relation the repective GetXXX() method has to be implemented.

6.1 Methods

- `IEntityRelationDataReader GetEntity(EntityType entityType, int[] attributes);`
  If data requested is from an Entity type such as a user or an item, this method will be invoked. Given the proper entityType is specified this method shall return the attributes (specified by the positions in the respective Entity given in the attributes array) for all instances of the requested entityType.
- `IEntityRelationDataReader GetRelation(RelationType relationType, int[] attributes);`
  If data requested is from a Relation type such as a rating (belonging to a user and an object at the same time), this method will be invoked. Given the proper relationType is specified this method shall return the attributes (specified by the positions in the respective Relation given in the attributes array) for all instances of the requested relationType.

6.2 Important

Data shall be consumed from any data provider in a "bulk" way, meaning all at a time and only once. Accordingly, the implementation of the respective GetXXX methods itself shall be able to deliver all of the requested data in a performing way – in one "chunk" only. The preferred data reader returned is the wrapper around the ADO.NET SqlDataReader though any other data reader such as a text file reader is also possible and several others have already successfully been implemented.

7 How To Implement Your Algorithm

As already emphasised, the developer of any algorithm shall focus on efficient query behaviour: though initial training of the algorithm (the statiscial learning / data mining) using the bulk data given by the data provider may take a long time, it is important that after this training the applications requesting recommendations are serviced fast—as the MyMedia UI and the whole MyMedia framework is issuing these requests continuously and dynamically as the users interact with the platform.

Thus it is smart to split up the algorithm in a possibly time- and resource-expensive training method that has to be completed before queries against the algorithm can be submitted and a lightweight way of retrieving information from the algorithm after training has completed. It is a good move to devise and algorithm that can later on be updated when new data is made available to it after the initial training.[2]

8 How To Write a Wrapper (IEntityRelationEngine)

The main purpose of the wrapper class derived from `IEntityRelationEngine` the algorithm developer / integrator has to create is a.) mapping the "expensive" method used to create the statistical model learned from the data to the `Train()` method of MyMedia and allowing predictions against this model via the `PredictRelation()` and `PredictEntity()` methods and b.) making the data provider accessible to the algorithm class via the `EntityRelationDataProvider` property.

8.1 Methods

- `EngineId Id` and
- `string Description`
  Both properties are necessary to use retrieve the respective algorithm in the MyMedia core framework. Both properties must return values that are unique among all Engines.
- `IEntityRelationDataProvider EntityRelationDataProvider`
  This property provides access to the data provider and thus the bulk data for training the algorithm / building the statistical model ("data mining model").
- `void Train()`
  This method is the major workhorse of any respective Eninge / algorithm. All time consuming steps should take place in this method and furthermore this method should be used to create an internal representation of the algorithm's results that is efficiently queryable when this Engine has finished training (i.e. it should create its internal Data Mining model). Access to bulk data shall only occur here.
- `bool IsTrainingCompleted()`
  After training is completed that event should fire in order to enable asynchronous programming.
- `double PredictRelation(RelationType relationType, int[] keys)`
  The Predict* methods are used to query to statistical model created during Train(). These methods must never retrieve any data from the core framework.
  The only data available in this method call is the learnt model created beforehands during Train() and the "keys" array necessary for indexing into the model and retrieving the proper recommendation score. ("keys" contains the key values necessary to uniquely identify one instance of the given RelationType.)
- `int[] PredictEntity(EntityType entityType, int[] keys, int count)`
  The Predict* methods are used to query to statistical model created during Train(). These methods must never retrieve any data from the core framework.

[2] For more information about incremental updates to your algorithm see section 10.

The only data available in this method call is the learnt model created beforehands during Train() and the “keys” array necessary for indexing into the model and retrieving the proper recommendation list. (“keys” contains the key values necessary to uniquely identify one instance of the given EntityType. The “count” specifies the maximum number of elements to return.)

8.2 Important

Never access any data during the PredictXXX methods except what is available as a parameter in the function call (the “keys” are necessary to address the proper element and the “count” specifies how many elements shall be returned at maximum) and the statistical model leanrt before during training of the algorithm. It is important that the PredictXXX methods return as soon as possible and always in the order of ns or few ms per call.

9 How To Write the Glue Code (EntityRelationEngine)

The `EntityRelationEngine` provides two options for incorporating existing and new Recommender `Engine` implementations.

The first is to use the existing `EntityRelationEngine` and `MyMediaEntityRelationDataProvider` classes, as in the folloing code snippet. A complete example is available in the MovieLens sample available with the source distribution. See the Recommender.cs file for complete implementation.

```
EntityRelationEngine matrixFactorizationEngine = new
EntityRelationEngine(new MatrixFactorization());
```

This will create a Recommender `Engine` class, which can be registered with the Recommender in the usual way. The `EntityRelationEngine` class will create and register a `MyMediaEntityRelationDataProvider` class, which provides the default `EntityType` and `RelationType` implementations from the core software framework. This includes updates for `User`, `CatalogItem`, and `UserAction`.

The second option is specifiy a custom `IEntityRelationDataProvider`, using the second constructor overload for `EntityRelationEngine`. A code snippet follows.

```
EntityRelationEngine entityRelationEngine = new EntityRelationEngine(new
MatrixFactorization(), new CustomeEntityRelationDataProvider());
```

This will create a Recommender `Engine` class, which can be registered with the Recommender in the usual way. The `EntityRelationEngine` class will create and register the `CustomeEntityRelationDataProvider` class, which provides the `IEntityRelationDataProvider` impelementation.

The `MyMediaEntityRelationDataProvider` class can be inherited to provide extened or overloaded support for `IEntityRelationDataProvider` implementations, while leveraging the

already present connective software to the core framework. See the software documentation help files for additional information.

10Incremental Updates

As already stated training of the algorithms shall run once for multiple recommendations requested by the applications. The main aim of this setting is preventing a.) unnecessary data transfer between the algorithm and the data source and, most importantly, b.) long delays between an application request for a recommendation and the recommendation being returned. These aims can be achieved by allowing for a possibly time consuming training of the algorithm when after this training the algorithms internal data structure ("data mining model") can be queried sufficiently fast to fulfil the timing requirements the applications demand.

Obviously, such an approach of training on the available data at one point in time only and delivering recommendations based on that data for a possibly long timespan (until the next iteration of the training of algorithm is completed using newer data) will lead to outdated recommendations unless a means of updating the model the recommendations come from is implemented.

While, algorithmically, the way of updating the model after training has been completed has to be derived by the algorithm developer, the infrastructure necessary to supply the algorithm with the data that was made available to the data source after the initial training of the algorithm is already in place: implemented in all interfaces the algorithm developer has to fill with life are the necessary UPDATE methods encapsulating the functionality of continuously supplying the algorithm with the change data – if it has opted in to receive those changes.

A more detailed description of the incremental update mechanism can be found in a later version of this document.

11Example

Please see the MovieLens sample project, in the source code distribution, for how to consume already existing engine algorithms.

See also the reference implementations of various algorithms, for source code examples of how to implement the various interfaces. These are included in the source distribution as well.