3
Implementação do PLSA

A implementação do *PLSA* não é uma tarefa simples. Como o algoritmo possui complexidade tempo e espaço quadrático a sua implementação não pode desperdiçar recursos computacionais, pois isto levaria à inutilidade do programa. Devido a esta restrição foi utilizada a linguagem de programação C (ISO/IEC 9899, 1990) para o desenvolvimento, sendo escolhida por proporcionar maior velocidade de execução e permitir o controle total da memória utilizada. Além do requisito não funcional de tempo e espaço, para uma boa engenharia do algoritmo é necessário torná-lo flexível e reutilizável, de forma a podermos utilizar o mesmo código para diversos segmentos que podem ser atendidos pelo *PLSA*.

Alguns dos pontos flexíveis do algoritmo são: o formato de entrada de dados e a inicialização das probabilidades a priori. Como vimos anteriormente, a inicialização influi diretamente no ponto de convergência do *PLSA*, além disto, para que a reutilização do algoritmo possa ser mais abrangente este disponibiliza alguns formatos de entrada de dados. Um destes formatos fornece a integração com a linguagem de programação *Python* (Python, 2008) que é muito utilizada na área de aprendizado de máquina.

Devido a esta flexibilidade e possibilidade de integração com outras linguagens de programação algoritmo foi incorporado ao framework de recomendação de anúncios na web *LearnAds*. Neste o *PLSA* se integra através de um módulo, e também permite a sua utilização por comitês de modelos através de algoritmos de *boosting* (Freund & Schapire, 1996).

Mostraremos a seguir a forma com que o *PLSA* foi implementado atendendo os requisitos acima descritos juntamente com o módulo desenvolvido para sua integração com o *LearnAds*.

3.1. Estratégias de Implementação

Para programarmos o *PLSA* é necessário calcular iterativamente as fórmulas 2.16, 2.19, 2.20 e 2.21. Uma forma intuitiva de fazê-lo é armazenar as tabelas geradas por estas em memória e calculá-las através dos dois passos do *EM*, o *E* que calcula a verossimilhança dos dados aos grupos (fórmula 2.16), e o passo *M* que calcula as demais fórmulas. As tabelas geradas são das dimensões *I*x*Z*, *U*x*Z*, *Y*x*Z* e *Y*x*Z* respectivamente, onde *I* representa o número de exemplos, *Z* o número de grupos latentes, *U* o número de usuários e *Y* o número de itens. Porém, esta abordagem leva a um problema de espaço em memória, pois o número de exemplos (*I*) necessário para o treinamento do algoritmo é normalmente grande e devido ao armazenamento das verossimilhanças, que gera a tabela de dimensão *I*x*Z,* o aumento do número de grupos latentes se torna inviável.

Analisando as fórmulas do *PLSA* observa-se que não é necessário armazenar as verossimilhanças (fórmula 2.16) inteiramente em memória, podendo guardar somente as verossimilhanças do exemplo atual, o que gera um vetor de tamanho *Z*. Isto é possível porque o calculo de $p(z|u)$ (fórmula 2.19), das médias μ_{zy} (fórmula 2.20) e das variâncias σ_{zy} (fórmula 2.21) se dão através da soma da contribuição de cada exemplo para cada grupo, sendo somente necessária a divisão pelas respectivas constantes de normalização. Desta forma o cálculo da contribuição do exemplo *i* para o grupo *z* é somado diretamente nas respectivas tabelas na posição do usuário *u* ou item *y*. Porém, o armazenamento direto da soma das contribuições torna necessário o guardar o resultado das tabelas de $p(z|u)$, μ_{zy} e σ_{zy} da interação anterior. Isto se dá porque o cálculo da verossimilhança dos exemplos necessita desta informação, a qual não se pode retirar das tabelas que estão guardando as contribuições da iteração atual.

Esta estratégia de implementação, onde o calculo é efetuado sem armazenar a tabela das verossimilhanças, apesar de duplicar as demais tabelas, necessita de menos espaço em memória que a outra estratégia quando o conjunto de dados é grande. Este fato ocorre porque em um conjunto de dados grande cada usuário possui diversos exemplos e cada item também, fazendo com que duplicar o espaço necessário para o armazenamento das tabelas (veja fórmula 3.1) seja menor que o espaço necessário para armazenar todas as verossimilhanças (ver fórmula 3.2).

$$2UxZ + 4YxZ + Z \tag{3.1}$$

$$IxZ + UxZ + 2YxZ \tag{3.2}$$

Com isto vimos uma boa estratégia para armazenar as tabelas que serão calculadas, porém ainda temos que armazenar os dados utilizados para treinar o algoritmo. Uma forma simples de fazê-lo é através da utilização de um vetor de triplas contendo usuário, item e valor (u, y, v) uma vez que os dados normalmente são esparsos. Porém, ao utilizarmos este formato repetimos a informação do usuário e do item para todos os exemplos, quando uma destas dimensões pode ser omitida caso os dados estejam ordenados. Para tal utilizamos um formato bastante difundido para o armazenamento de matrizes esparsas, o formato comprimido por linha (Compressed Row Sotrage, 2008). Neste formato uma matriz com n colunas, m linhas e k entradas não vazias é armazenada em três vetores *rowptr*, *rowind* e *rowval* de tamanhos $n + 1$, k e k respectivamente. O vetor *rowind* armazena o índice da coluna de cada entrada preenchida na matriz ordenadamente, começando da primeira linha seguindo para a segunda até atingir a última. O vetor *rowval* armazena cada valor não nulo da matriz na mesma ordem que o vetor *rowind*. Já o vetor *rowptr* armazena em qual posição de *rowind* e *rowval* começa cada linha, identificando assim o primeiro elemento da linha i por *rowptr*[i] e o último por *rowptr*[i +1] -1. Um exemplo do formato comprimido por linha é mostrado na figura 3.1.

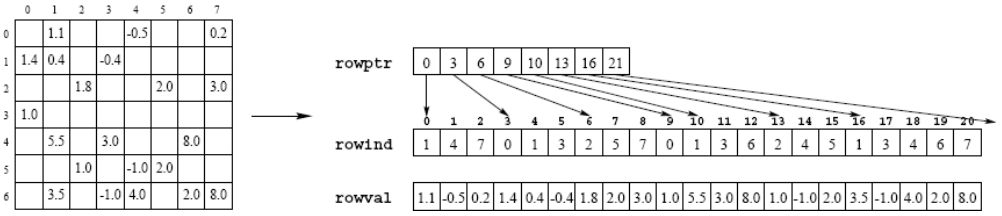

Figura 3.1 – Exemplo de dados no formato comprimido por linha (Karypis, 2002)

Assim utilizamos um formato de dados comprimido exigindo menos espaço em memória para o treinamento do *PLSA,* porém ainda precisamos definir se a matriz esparsa terá os usuários ou itens nas linhas da matriz. Analisando a

natureza das fórmulas do *PLSA* notamos que os cálculos da média e da variância para serem eficientes computacionalmente necessitam de alguns vetores intermediários. Estes servem para armazenar as somas dos elementos parciais de cálculo sendo agrupados por item e por grupo. A utilização do formato comprimido por linha exige que façamos a iteração pelas linhas, ou seja, visitando todos os exemplos de um determinado usuário ou item seqüencialmente. Desta forma se optarmos pelos itens serem alocados nas linhas todos os exemplos de um determinado item serão lidos em seqüência, podendo armazenar as somas parciais da média e variância somente para o item em questão, o que não aconteceria no caso de os usuários representarem as linhas. Por este motivo optamos por colocar os itens nas linhas e os usuários nas colunas.

Até aqui analisamos as estratégias de implementação do *PLSA* e decidimos pela que presta as maiores vantagens. Porém ainda é preciso atender os requisitos de flexibilidade e reusabilidade do programa. Para mostrar como estes requisitos foram atendidos serão veremos a seguir o projeto do algoritmo em detalhes.

3.2. Projeto

Por motivos de desempenho a linguagem de programação C foi escolhida para a programação do *PLSA*, o que influenciou fortemente a arquitetura do mesmo. Devido à linguagem C não ser orientada a objeto o projeto do *PLSA* se baseou na técnica estruturada de programação e no fluxo de informação que esta fornece. Com isto conseguimos uma visão de alto nível do programa através de dois processos de transformação de informação, o treino do algoritmo que gera um modelo aprendido através dos dados e a predição de novos elementos a partir do modelo gerado. Tais transformações são representadas pelos diagramas de fluxo de dados das figuras 3.2 e 3.3, mostrando os dados necessários para cada transformação.

Estas transformações de informação são representadas por funções na linguagem C. Dentre as funções que foram programadas no *PLSA* as duas principais são as que implementam o treino e a predição denominadas *train_plsa_csr_data* e *predict_value* respectivamente. Porém estas funções isoladamente não proporcionam flexibilidade ao programa desenvolvido uma vez que precisamos efetuar transformações nos dados antes de passá-los para estas funções. A função *train_plsa_csr_data* precisa que os dados estejam no formato comprimido por linha e também que os índices das colunas não sejam superiores ao número de colunas totais, onde a última restrição também é necessária para o método *predict_value*.

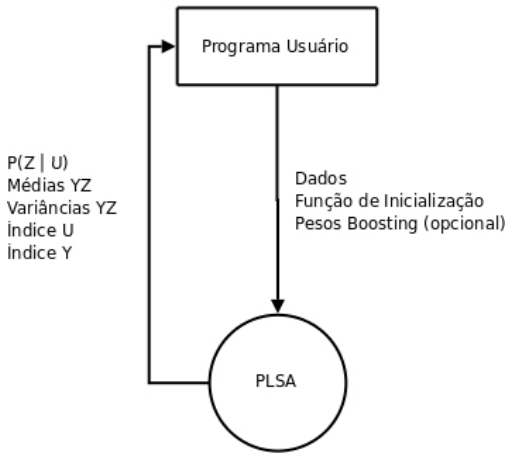

Figura 3.2 – Diagrama de fluxo de dados do treino do PLSA

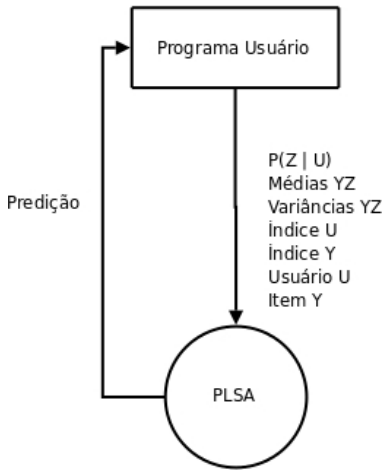

Figura 3.3 – Diagrama de fluxo de dados da recomendação do PLSA

É bastante incomum encontrarmos dados que já atendam estas restrições, sendo necessário o processamento para sua utilização. Com o intuito de resolver este inconveniente foram adicionados dois métodos à implementação do *PLSA* o *índex_data* e o *to_csr_fromat.* O primeiro recebe os dados no formato de vetor de triplas (*u*, *y*, *v*) e gera dois índices, um sobre os usuários e outro sobre os itens, de forma que valor máximo dos índices não ultrapasses o número de usuários e itens respectivamente. Já o método *to_csr_fromat* transforma o formato de dados de vetor de triplas para o formato comprimido por linha indexando os usuários e itens caso os índices sejam passados.

Com a inclusão dos métodos *índex_data* e *to_csr_fromat* temos uma nova opção de entrada de dados para o algoritmo, o vetor de triplas, onde as triplas são representadas pela estrutura *t_triple* (quadro 3.1). Porém para que a utilização destes dados seja feita de maneira simples precisamos encapsular o processo de transformação criando um novo método. Para atingir este objetivo foi criado o método *train_plsa_full_data*.

Contudo a facilidade gerada pelo método *train_plsa_full_data* tem suas desvantagens, a utilização dos dados no formato de vetor de triplas ocupa mais memória que o formato comprimido por linha, e, como é necessário armazenar os dois formatos em memória para efetuar a conversão este recurso pode se tornar escasso. Com isto não é recomendada a utilização deste método para conjunto de

dados grandes, sendo mais indicado neste caso a utilização do *train_plsa_csr_data.*

```
typedef struct _t_triple {
    int user;
    int item;
    float value;
    short test;
    float weight;
} t_triple;
```

Quadro 3.1 – Estrutura de triplas

Como vimos o método *train_plsa_csr_data* exige menos memória do que o seu similar, porém ainda esbarramos nos dados que normalmente não estão organizados no formato comprimido por linha, sendo necessário processá-los antes de serem passados para o *PLSA*. A forma ideal para contornar este problema é converter os dados para o formato comprimido conforme são lidos, ocupando assim o mínimo de memória possível. Para disponibilizar esta funcionalidade foi introduzido um novo método o *iterator_to_csr_format*, que recebe como parâmetro um ponteiro para uma função de interação sobre os dados e retorna os dados já convertidos e indexados no formato comprimido por linha.

Para efetuar esta operação o método *iterator_to_csr_format* chama a função de iteração sempre que é necessário ler um dado, devendo este fornecê-lo ordenado por item. Este método permite que a função de interação seja passada como parâmetro tornando a leitura de dados flexível. Pode-se, por exemplo, substituir a leitura dos dados de um arquivo de texto para um banco de dados trocando apenas uma função. Isto é possível devido a dois de seus argumentos serem ponteiros para *void* permitindo que seja passado um ponteiro de qualquer tipo em seu lugar, seja um ponteiro para arquivo, conexão de banco ou simplesmente um vetor. Para facilitar a utilização do método encapsulamos a sua passagem de parâmetro e treinamento do PLSA no método *train_plsa_iterator_data* onde é passado para esta a função de interação sobre os dados e é retornado modelo treinado.

Com isto atingimos um nível razoável de reusabilidade e flexibilidade quanto à entrada de dados, porém ainda é necessário tratar a inicialização das verossimilhanças no *PLSA*. Este ponto é abordado através de um ponteiro para a

função de inicialização, sendo este passado diretamente para o método de treino. A utilização de um ponteiro para função é necessária porque as verossimilhanças não são armazenadas completamente em memória, precisando ser recuperadas uma a uma conforme os exemplos são processados. Desta forma a função de inicialização recebe como parâmetro o exemplo a ser processado, o grupo em questão, além de alguns argumentos do tipo ponteiro para *void* que permitem certa flexibilidade ao método e recupera a verossimilhança.

Através da arquitetura descrita é possível definir uma forma de inicialização do *PLSA* pela programação de uma função de inicialização e passagem de seu ponteiro para o método de treinamento. Porém existem formas de inicialização que são bastante comuns, como por exemplo, a inicialização aleatória das probabilidades. Devido a isto foram adicionados ao programa três métodos que fornecem inicializações de verossimilhanças típicas, são eles: *random_init*, *file_init* e *cluster_init*.

O método *random_init* disponibiliza a inicialização aleatória das probabilidades, fornecendo simplesmente um número aleatório toda vez que o método é chamado. Já o *file_init* fornece uma inicialização através de um arquivo texto. Para isto são passados como argumento o nome do arquivo que será utilizado e um ponteiro para este (*FILE* *) onde será armazenada a referência do arquivo aberto. Seu formato de entrada é constituído de linhas de dados, onde cada linha possui apenas uma probabilidade e estas são ordenadas primeiramente por exemplo (*i*) e depois por grupo (*z*). Finalmente o método *cluster_init* fornece a inicialização das verossimilhanças a partir da resposta de um algoritmo de clustering. Tal resposta deve ser passada como parâmetro para esta função através de um vetor que indica em qual cluster cada exemplo foi classificado, sendo passado também o número de clusters totais gerados *nclusters*. A partir destas informações a função *cluster_inti* retorna para cada exemplo *i* e grupo *z* uma probabilidade onde caso o exemplo *i* tenha sido classificado no cluster *z* esta é *nclusters* vezes maior do que se ele pertencer a outro cluster. As verossimilhanças são fornecidas desta maneira com base em resultados experimentais que serão apresentados no capítulo 4.

As formas de inicialização apresentadas juntamente com o formato de entrada de dados possibilitam a flexibilidade e reusabilidade do código do *PLSA,* facilitando assim a integração com outros programas. Porém, gostaríamos de

disponibilizar a integração do *PLSA* para comitês de modelos, em especial utilizando a técnica de *boosting* (Freund & Schapire, 1996). Tal técnica visa construir um modelo de aprendizado para um problema utilizando um conjunto de modelos especializados em pedaços do mesmo. Com este objetivo os algoritmos de *boosting* passam pesos para os exemplos, gerando assim modelos especializados nas áreas onde os pesos são maiores.

Para disponibilizar a integração do *PLSA* com o *boosting* foram alteradas as fórmulas de cálculo das médias e variâncias, adotando uma heurística que pondera seu cálculo utilizando os pesos dos exemplos. Tal heurística é mostrada na fórmula 3.1 e 3.2, onde w_i representa o peso do exemplo *i*. Nesta ao passarmos todos os pesos iguais a 1 temos o resultado idêntico a das fórmulas originais, caso contrário, quanto maior for o peso do exemplo maior será a sua influência no valor da média e variância.

$$\mu_{ZY} = \frac{\sum_{yi=y} v_i w_i p(z \mid u_i, y_i, v_i, \theta')}{\sum_{yi=y} w_i p(z \mid u_i, y_i, v_i, \theta')} \tag{3.1}$$

$$\sigma_{ZY} = \frac{\sum_{yi=y} p(z \mid u_i, y_i, v_i, \theta')(v_i - \mu_{ZY}) w_i}{\sum_{yi=y} p(z \mid u_i, y_i, v_i, \theta') w_i} \tag{3.2}$$

No programa do *PLSA* os pesos dos exemplos podem ser passados diretamente para o método *train_plsa_csr_data* através de um vetor, ou através da estrutura *t_triple* para os métodos *train_plsa_full_data* ou *train_plsa_iterator_data.* Apesar do nome a estrutura *t_triple* (quadro 3.1) possui mais de três atributos, nesta os atributos *user*, *item* e *value* são intuitivos, o atributo *test* permite que um exemplo não seja levado em consideração no treino e *weight* representa o peso de um exemplo no algoritmo de *boosting*. Em todos os métodos é facultativa a utilização dos pesos do *boosting*, tal opção é dada pelo parâmetro *boosting* onde 0 (zero) indica que não serão utilizados os pesos e 1 indica a sua utilização.

Com isto concluímos a gama de opções de parâmetros dadas pelo programa do *PLSA*, tais opções podem ser visualizadas em forma de diagrama na figura 3.4. Neste diagrama são exibidas as opções de escolha da função de inicialização e do

formato de entrada de dados através de setas tracejadas, indicando que é escolhida somente uma opção por vez, já os parâmetros de entrada e saída do algoritmo são representados através de setas sólidas (não tracejadas), indicando que são passados e retornados sempre pelo algoritmo.

Mostraremos a seguir como esta implementação pode ser utilizada em um sistema de recomendação, para isto será apresentado o framework de recomendação de anúncios na web *LearnAds,* e sua instanciação para comportar o *PLSA*.

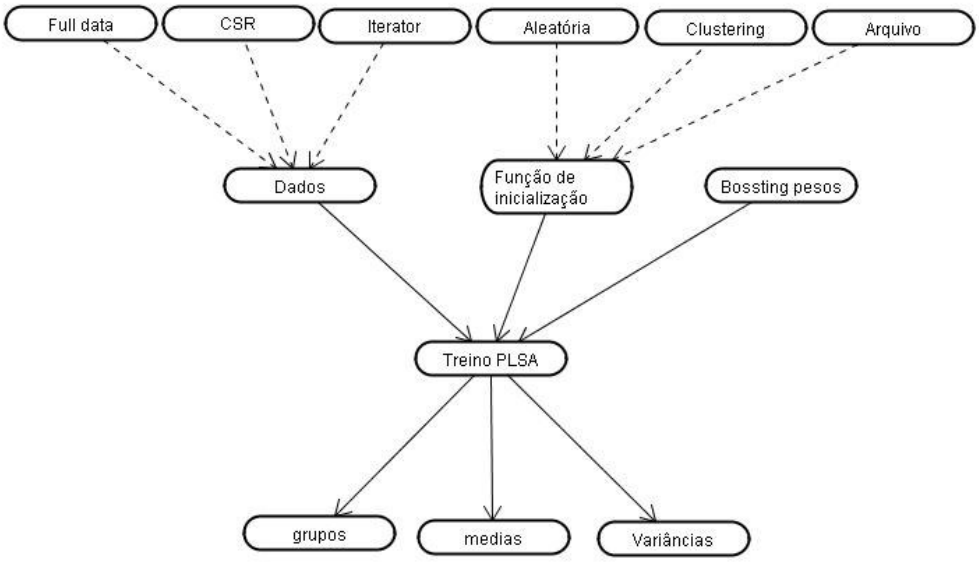

Figura 3.4 – Opções de parâmetros fornecidos pelo *PLSA*

3.3. Integração com o LearnAds

O framework *LearnAds* foi desenvolvido com o intuito de suportar diversos métodos de aprendizado de máquina, podendo ser adaptado através da instanciação de seus pontos flexíveis. Para obter uma maior flexibilidade e agilidade de desenvolvimento a linguagem escolhida para sua implementação foi o *Python* (Python, 2008), que é uma linguagem interpretada, orientada a objeto com um grande suporte a computação de vetores. Devido ao framework *LearnAds* ter sido desenvolvido nesta linguagem foi construída uma interface de comunicação entre o *PLSA*, que foi implementado em C, e a linguagem *Python*. Os métodos desta interface são listados a seguir:

- *setParameters*: Registra parâmetros necessários para a alocação e memória pelo algoritmo. Alguns parâmetros deste método são: Número de exemplos, número de grupos latentes, número de usuários, entre outros.
- *addData:* Método utilizado para passar os dados para a implementação do *PLSA*. Adiciona um exemplo ao vetor de exemplos.
- *train:* Método que inicia o treinamento do *PLSA*. É passado como parâmetro o número de iterações que serão executadas no mesmo.
- *predict:* Método que gera uma predição para um usuário e um item.
- *setFileName:* Método que armazena o nome do arquivo de entrada das probabilidades iniciais do *PLSA*, caso este método de inicialização seja escolhido.
- *storeModel:* Grava em um arquivo texto o modelo aprendido pelo *PLSA*.
- *loadModel:* Carrega para memória um modelo previamente armazenado em um arquivo texto.

A arquitetura do framework baseia-se no padrão MVC (*Model-View-Control*) com o objetivo de diferenciar e desacoplar classes de camadas distintas. Dada esta diferenciação apresentaremos aqui os modelos de classes de duas destas

camadas, a de modelo e a de controle, já a camada de vista fica por conta do programa cliente que utiliza o framework. Começaremos então pelo modelo.

A camada de modelo do framework, mostrado na figura 3.4, é constituída por entidades de negócio que representam objetos do mundo real, e, classes auxiliares que tem como objetivo organizar e tornar flexível o modelo. Faremos aqui uma breve descrição de cada classe começando pelas entidades de negócio.

O framework tem como objetivo recomendar anúncios baseados em palavras chaves, estas são representadas pelo objeto *Query*, que no contexto de um mecanismo de busca representa a consulta do usuário. Esta consulta pode ser efetuada várias vezes pelo mesmo usuário ou por outros usuários, tais ocorrências de uma determinada *Query* são representadas pelo objeto *Search*.

Uma busca retorna vários documentos (objeto *Document*) e anúncios (objeto *Advertisement*), que por sua vez podem ser retornados em diversas buscas, configurando um relacionamento *m* para *n*. Tais relacionamentos se dão através dos objetos *AdPrintAtributes* e *DocumentClick*, respectivamente. Estes objetos armazenam informações relevantes sobre os cliques dos documentos em uma busca e sobre a exibição de um anúncio, tanto em uma busca (relacionamento com o objeto *Search*), quanto em anúncio contextual, ou seja, apresentado dentro de uma página de conteúdo na internet (representado pelo relacionamento com o objeto *Document*).

Para que um anúncio seja exibido é preciso que ele esteja vinculado a palavras chaves, devendo ter também um valor associado ao clique do anúncio quando ele é disparado através da palavra. Este relacionamento é representado pelo objeto *Bid*.

Com isto concluímos as entidades de negócio do framework, faltando explicar as suas classes auxiliares, representadas pelas classes *Feature*, *DocumentFactory*, *AdvertisementFactory* e *ModelFacade*.

A classe *Feature* é um ponto flexível do modelo (os pontos flexíveis aparecem no modelo em amarelo), representando qualquer propriedade que um documento ou anúncio possa ter e seja relevante para recomendação dos anúncios. Este ponto flexível é representado por uma classe genérica, constituída do id da propriedade, nome representativo desta, tipo de dado armazenado e valor da propriedade.

Quando é criada uma instância do framework são criados modelos de anúncios e documentos com conjuntos de propriedades fixas e específicas da instância criada. Para que isto seja possível é utilizado o padrão de projeto *AbstractFactory* representado pelas classes *DocumentFactory* e *AdvertisementFactory*.

Estas classes encapsulam a criação de modelos dos objetos *Document* e *Advertisement* respectivamente, permitindo a troca de um modelo de *features* apenas pela troca das fábricas concretas.

Finalmente a classe *ModelFacade* representa o padrão *Facade*. Este é utilizado para desacoplar a camada de modelos da camada de controles através do direcionamento de todas a comunicação entre estes pacotes para a classe *ModelFacade*. Assim, com a comunicação centralizada, em caso de alterações no modelo reduz-se o número de classes afetadas diretamente pela mudança, reduzindo o número de alterações de códigos.

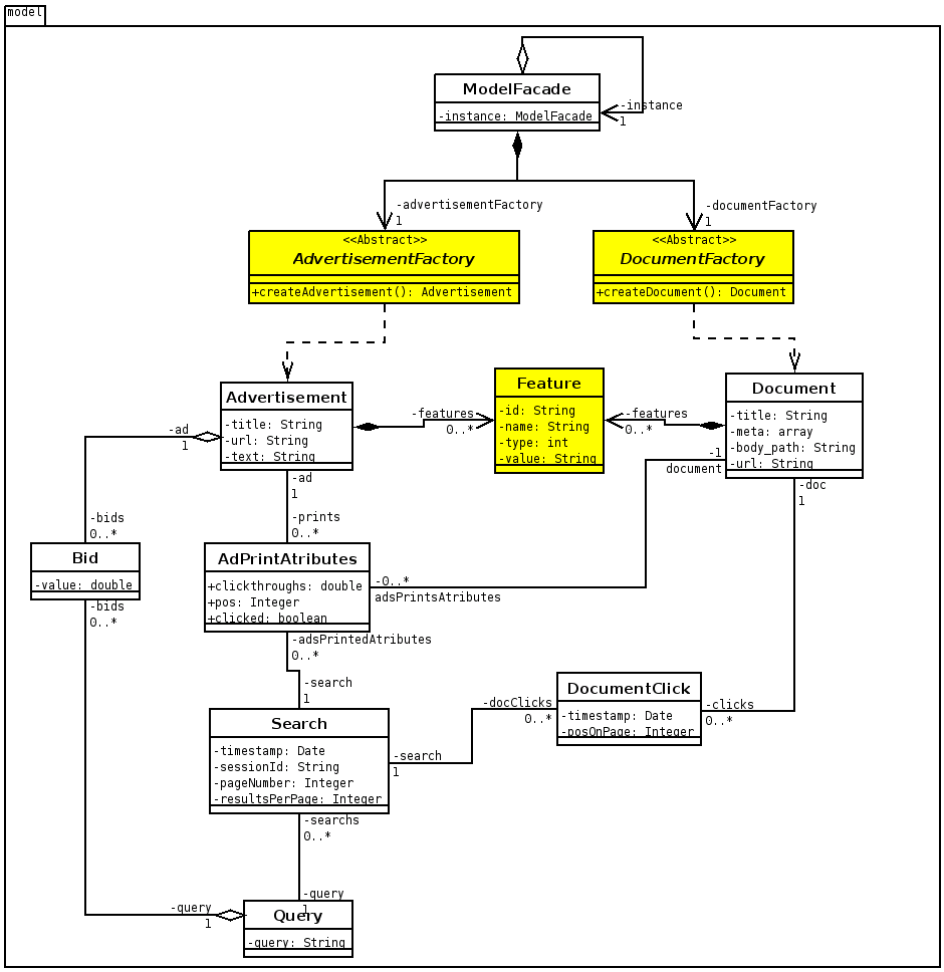

Figura 3.4: Diagrama de classes da camada de modelo

A camada de controle é constituída de classes capazes de tratar um comando do usuário e executar uma recomendação de um conjunto de anúncios. Para isto temos classes que organizam as requisições possíveis ao sistema, utilizando o padrão *Command,* e classes responsáveis pela recomendação dos anúncios em si.

As classes *ControlServlet* e *Command* são utilizadas para fazer o tratamento das requisições enquanto as classes restantes são utilizadas para a recomendação. Durante está são necessárias operações para distinguir quais anúncios podem ser recomendados dado uma situação (*match*) e qual será a ordem determinada para a exibição dos anúncios (*rank*). Estas responsabilidades são endereçadas respectivamente pelas classes *AdMatcher* e *AdRanker*.

No framework as operações de *match* e *rank* são pontos flexíveis. Tais pontos são representados na arquitetura através do padrão de projeto *Strategy* que permite a troca suave de estratégia em uma determinada operação. Tal padrão é representado pelas classes *AdMatcher* e *AdMatcherStrategy* na operação de *match* e *AdRanker* e *AdRankerStrategy* na operação de *rank*.

Finalmente as classes *Predictor* e *CrossValidator* são responsáveis pela utilização e validação do aprendizado de máquina na recomendação. A classe *Predictor* possui métodos para efetuar o aprendizado de uma abordagem e também efetuar a predição dado o modelo aprendido. Já a classe *CrossValidator* é utilizada para efetuar a validação cruzada da abordagem utilizada por um *Predictor*.

Para instanciarmos o framework com o *PLSA* é necessário instanciar algumas classes abstratas do modelo, são elas: *Command*, *AdMatcherStrategy*, *AdRankerStrategy*, *Predictor*, *AdvertisementFactory* e *Advertisement*. Além disto, será necessária a associação dos comandos na classe ControlServlet, associação do *AdvertisementFactory* concreto na classe *AdvertisementFactory* e do *DocumentFactory* analogamente.

O PLSA é uma modelagem onde aprendemos sobre variáveis não observáveis. Dentro do contexto de recomendação de anúncios baseado em palavras chaves, uma das modelagens probabilísticas possíveis é considerar que existem grupos de *queries* onde a posição mais adequada de cada anúncio depende diretamente do grupo da *query*. Os grupos fazem o papel de variável não

observável, e, temos também, que a distribuição das posições de cada anúncio varia com as probabilidades da consulta estar em cada grupo.

Para que seja possível recomendar efetivamente anúncios são necessárias três atividades no framework: a avaliação de diversos modelos variando os parâmetros do *PLSA* para que sejam identificados os melhores parâmetros para o problema, após a identificação destes parâmetros o treinamento do modelo com os melhores resultados, e, finalmente a recomendação dos anúncios através do modelo gerado. Estes três passos são modelados através dos casos de uso mostrados a seguir:

- UC01 – Avaliação de estratégias de recomendação de anúncios;
- UC02 – Geração de um modelo baseado em uma estratégia definida;
- UC03 – Recomendação de anúncios baseada num modelo existente.

Os casos de uso UC01, UC02 e UC03 são descritos respectivamente no Quadro 3.1, Quadro 3.2 e Quadro 3.3, e têm sua implementação no framework mostrado através dos diagramas de seqüência exibidos nas figuras 3.6, 3.7 e 3.8. Em tais diagramas os métodos da interface do *PLSA* com o framework são apresentados através do objeto *libopenplsa*, que nada mais é que uma biblioteca de vinculo dinâmico (*DLL* ou *SO*) compilada com a interface *Python-C*.

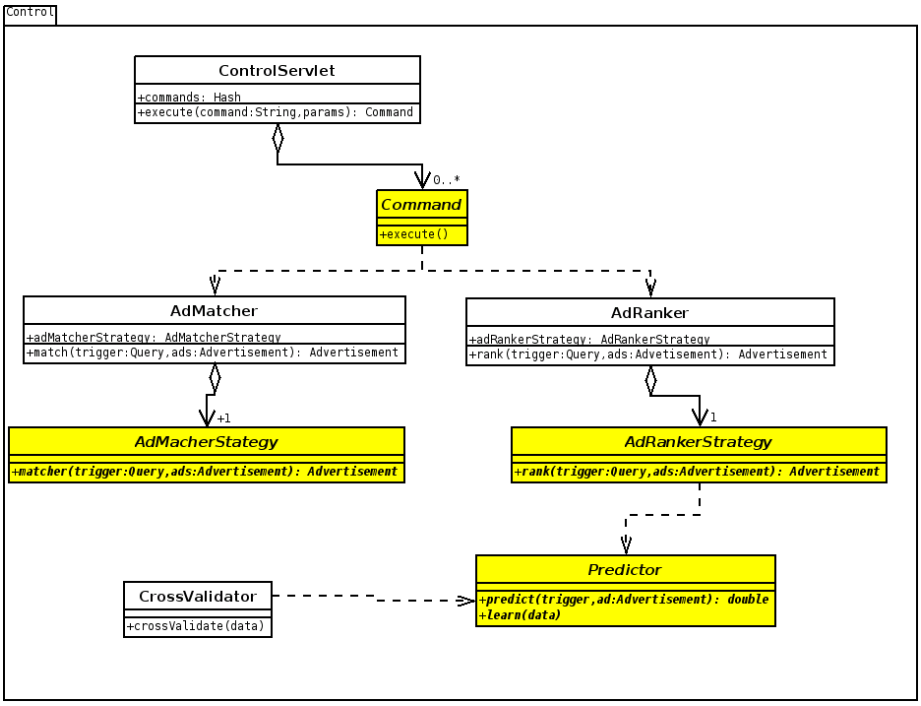

Figura 3.5: Diagrama de classes da camada de controle

Quadro 3.1: UC01 – Avaliação de estratégias de recomendação de anúncios.

Ator Primário: Programador Pré-Condições: banco de dados preenchido com dados de *query logs* Fluxo Normal: 1) Programador seleciona parâmetros da validação cruzada, a métrica de avaliação, algoritmo e/ou parâmetros que representam a estratégia a ser testada. 2) *Framework* retorna o resultado da avaliação.

Quadro 3.2: UC02 – Geração de um modelo baseado em uma estratégia definida.

Ator Primário: Programador Pré-Condições: banco de dados preenchido com dados de *query logs* Fluxo Normal: 1) Programador seleciona algoritmo e/ou parâmetros que representam a estratégia desejada para a geração do modelo preditivo. 2) *Framework* armazena o modelo gerado em um arquivo.

Quadro 3.3: UC03 – Recomendação de anúncios baseada num modelo existente.

Ator Primário: Programa Cliente Pré-Condições: modelo preditivo de recomendação previamente gerado. Fluxo Normal: 1) Usuário seleciona o modelo pretendido e entra com a consulta/palavra-chave. 2) *Framework* retorna uma lista ordenada de anúncios relacionados com a consulta/palavra-chave.

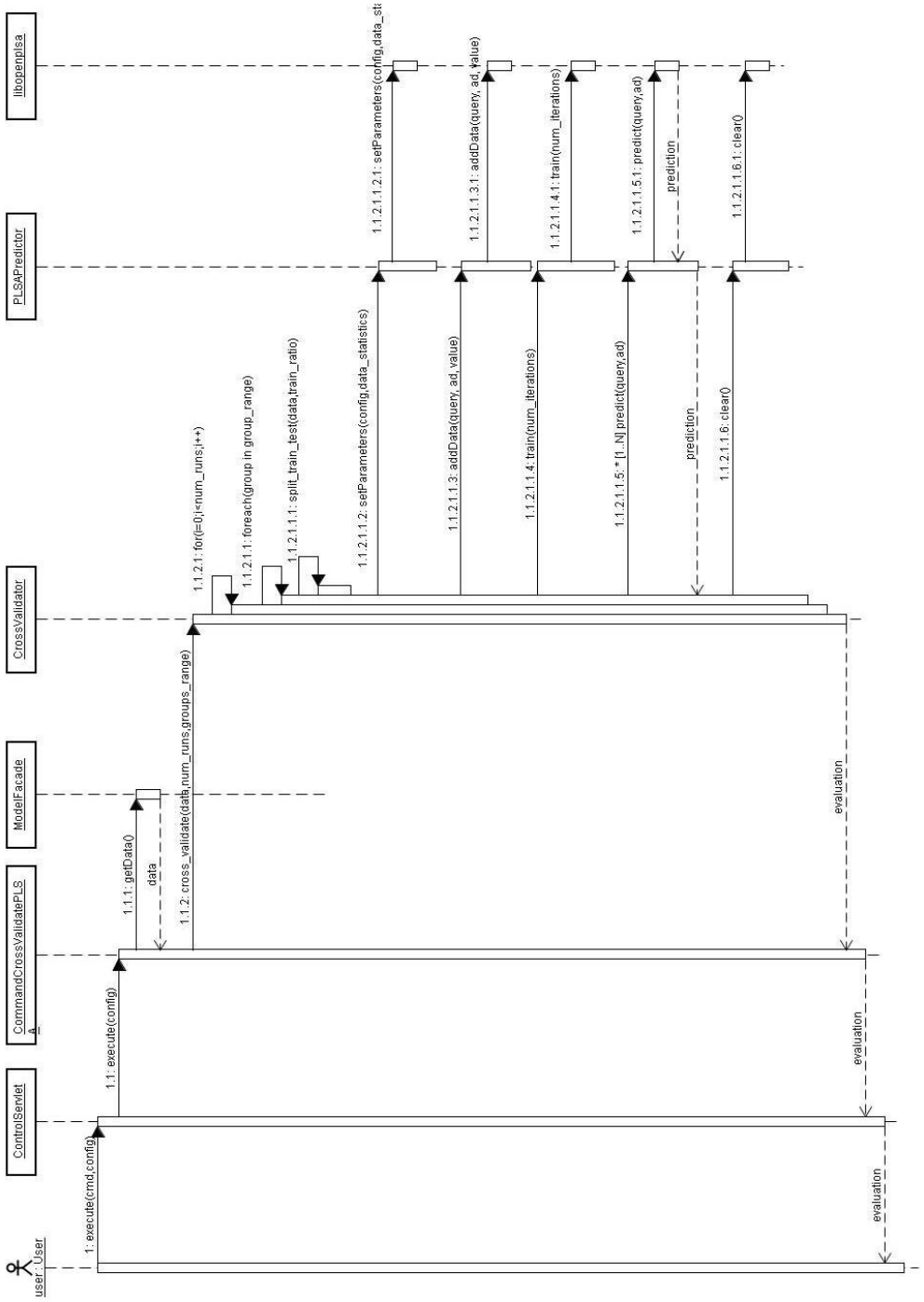

Figura 3.6: Diagrama de Seqüência referente ao caso de uso UC01 – Avaliação de estratégias de recomendação de anúncios

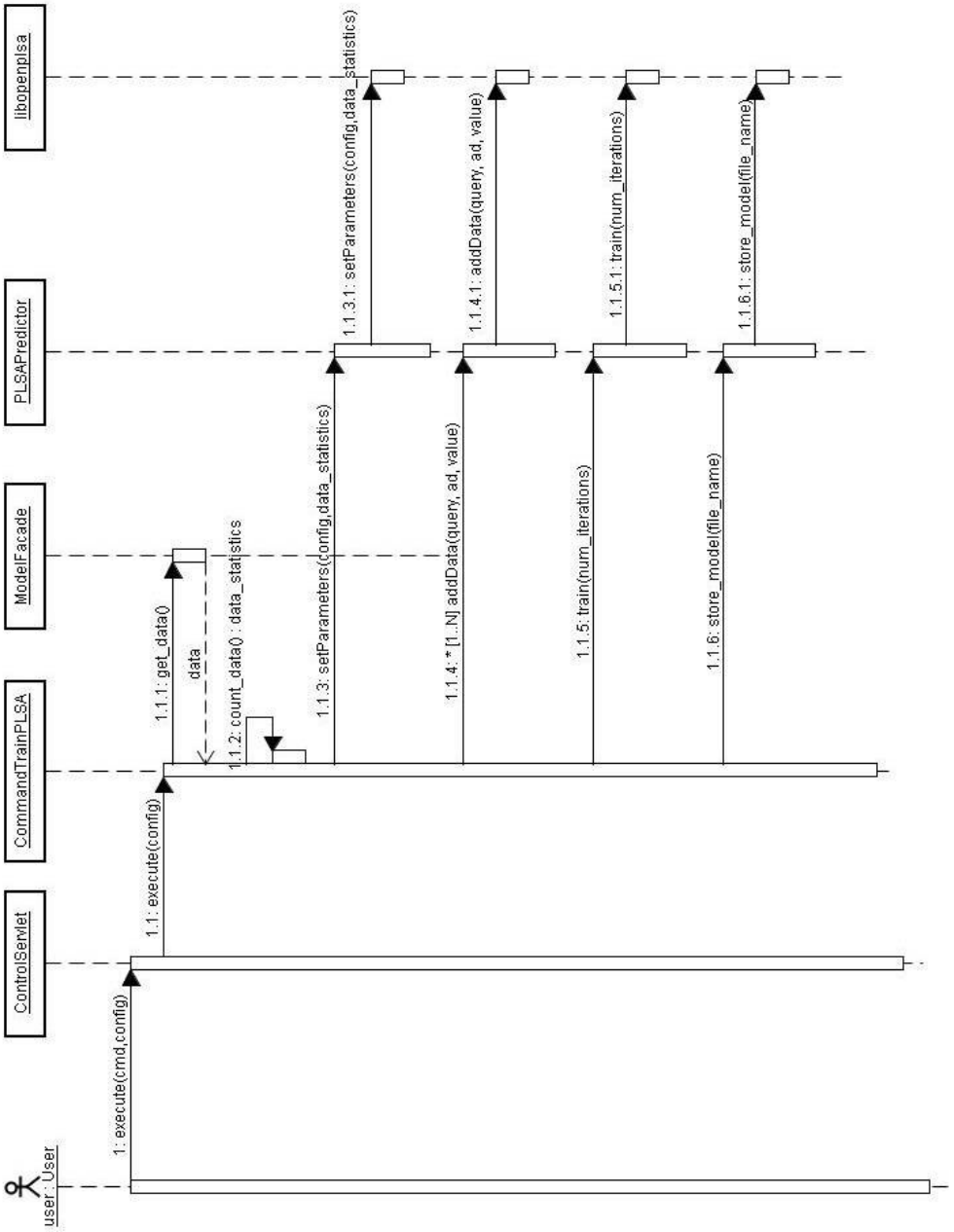

Figura 3.7: Diagrama de Seqüência referente ao caso de uso UC02 – Geração de um modelo baseado em uma estratégia definida

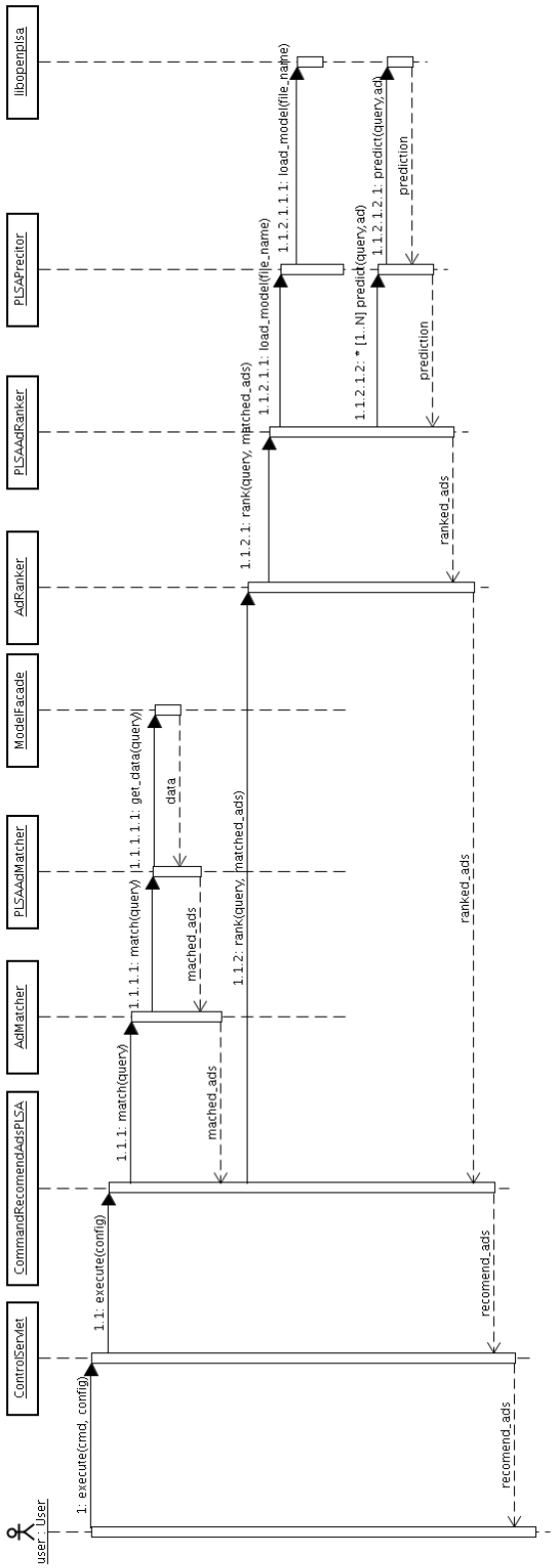

Figura 3.8: Diagrama de Seqüência referente ao caso de uso UC03 – Recomendação de anúncios baseada num modelo existente