**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ**
**CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS**

**LUIZ GUSTAVO BOURSCHEIT**

# MINERAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS ANDROID: GERAÇÃO DE OPÇÕES GASTRONÔMICAS UTILIZANDO REGRAS DE ASSOCIAÇÃO

**TRABALHO DE CONCLUSÃO DE CURSO**

**PATO BRANCO**
**2017**

**LUIZ GUSTAVO BOURSCHEIT**

# MINERAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS ANDROID: GERAÇÃO DE OPÇÕES GASTRONÔMICAS UTILIZANDO REGRAS DE ASSOCIAÇÃO

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Dalcimar Casanova

**PATO BRANCO**

**2017**

**Ministério da Educação**
**Universidade Tecnológica Federal do Paraná**
**Câmpus Pato Branco**
**Departamento Acadêmico de Informática**
**Curso de Tecnologia em Análise e Desenvolvimento de Sistemas**

# TERMO DE APROVAÇÃO

## TRABALHO DE CONCLUSÃO DE CURSO

## MINERAÇÃO DE DADOS EM DISPOSITIVOS ANDROID: GERAÇÃO DE OPÇÕES GASTRONÔMICAS UTILIZANDO REGRAS DE ASSOCIAÇÃO

**POR**

**LUIZ GUSTAVO BOURSCHEIT**

Este trabalho de conclusão de curso foi apresentado no dia 03 de julho de 2017, como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, pela Universidade Tecnológica Federal do Paraná. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

**Banca examinadora:**

Prof. Dr. Dalcimar Casanova
Orientador

Prof. Dr. Pablo Gauterio Cavalcanti

Prof. MSc. Vinicius Pegorini

Prof. Dr. Edilson Pontarolo
Coordenador do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Profª Drª Beatriz Terezinha Borsoi
Responsável pela Atividade de Trabalho de Conclusão de Curso

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

## RESUMO

BOURSCHEIT, Luiz Gustavo. Mineração de dados em dispositivos móveis Android: geração de opções gastronômicas utilizando regras de associação. 2017. 65f. Monografia (Trabalho de Conclusão de Curso) - Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2017.

A mineração de dados é um conceito existente desde a década de 80 e que surgiu com o intuito de identificar padrões úteis e compreensíveis em uma grande massa de dados. Um dos principais interesses da mineração é a indução de regras de associação, e um dos algoritmos mais conhecidos para geração destas regras é o *A priori.* No entanto, a aplicabilidade das regras de associação está, em sua grande maioria, no escopo comercial, sendo que a exploração das regras de associação para uso pessoal é pouco executada. Neste trabalho é apresentado o uso do algoritmo de mineração *A priori*, adaptado a linguagem Java, para uso pessoal, isto é, aplicado a uma situação corriqueira das pessoas, sem envolvimento com a divulgação ou promoção de marcas, produtos e afins, sendo que o objetivo principal é gerar opções de locais gastronômicos para visita aos seus usuários.



**Palavras-chave**: Mineração de Dados. Regras de Associação. Algoritmo *A priori*. Conjunto de itens frequentes. Aplicações Android.

## ABSTRACT

BOURSCHEIT, Luiz Gustavo. Data mining on Android mobile devices:generation of gastronomic options using association rules. 2017. 65 f. Monografia (Trabalho de Conclusão de Curso) - Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2017.

Data mining is an existing concept from the 80s. This area of research has come up in order to identify useful and understandable patterns in large data. One of the main interests of the data mining is the induction of association rules and, one of the most known algorithms is the A priori. However, the main applicability of association rules is in the commercial scope, whereas the exploration of association rules on the personal scope is underachieved. This work presents the use of data mining algorithm A priori, adapted to the Java language, for personal application. The main goal is the generation of gastronomic options for visits and tasting.



**Keywords**: Data mining. Association Rules. *A priori* algorithm. Set of frequent itens. Android applications.

## LISTA DE FIGURAS

## LISTA DE QUADROS

## LISTAGENS DE CÓDIGOS

## LISTA DE SIGLAS/ABREVIATURAS/ACRÔNIMOS

| | |
|---|---|
| KDD | *Knowledge discovery in database* |
| SQL | *Structured Query Language* |
| IDE | *Integrated Development Environment* |
| UML | *Unified Modeling Language* |
| XML | *Extensible Markup Language* |
| API | *Application Programming Interface* |
| SGBD | Sistema Gerenciador de Banco de Dados |
| SDK | *Software Development Kit* |
| SO | Sistema Operacional |
| JVM | *Java Virtual Machine* |
| DEX | *Dalvik Executable* |
| APK | *Android Package File* |
| DAO | *Data Access Object* |
| JAR | Java *Archive* |
| IP | *Internet Protocol* |
| REST | *Representational State Transfer* |
| MVC | *Model-View-Controller* |

**SUMÁRIO**

# 1 INTRODUÇÃO

Desde que a informática se tornou uma ferramenta essencial no cotidiano da população, imensos volumes de dados têm sido regularmente coletados e armazenados, sendo que o simples fato de ter esses dados guardados digitalmente cria uma sensação de conforto, segurança e praticidade em cada um. No entanto, com esse grande volume de dados armazenados e crescendo diariamente, o processamento deles, a fim de gerar informações, se tornou crucial, tendo em vista que, para Galvão e Marin (2009), a informação e o conhecimento são prerrogativas legais, estratégicas e imprescindíveis à busca de maior autonomia nas ações.

Nesse sentido, Galvão e Marin (2009) ainda destacam que a informática e as tecnologias voltadas para a coleta, armazenamento e disponibilização de dados vêm evoluindo e disponibilizando técnicas, métodos e ferramentas automatizadas, capazes de auxiliar na extração de informações úteis contidas nesse grande volume de dados complexos. E como ferramenta dessa necessidade exposta, surge na década de 80 o conceito de mineração de dados (do inglês *data mining*), que para Fayyad et al (1996), nada mais é do que o processo não trivial de identificar, em dados, padrões válidos, novos, potencialmente úteis e ultimamente compreensíveis.

Pode-se destacar ainda que *data mining* é o processo de análise de conjuntos de dados que tem por objetivo a descoberta de padrões interessantes e que possam representar informações úteis. Com a utilização desse conceito, é possível analisar dados comportamentais, obtendo conhecimento que estava "escondido" na base de dados, gerando dados complementares que podem influenciar no desenvolvimento de estratégias e ações (SFERRA E CORRÊA, 2003).

No entanto, a aplicabilidade dessa tecnologia hoje é mais difundida no âmbito profissional, visando obtenção de lucros para empresas, assim como a geração de estratégias de marketing para promover marcas e produtos, sendo que o seu uso para geração de conhecimento para uso pessoal é pouco executado. É bem verdade que a necessidade empresarial em processamento de dados é maior que a pessoal, porém existem situações práticas onde a extração de dados se torna extremamente útil na vida de cada ser.

Com base em tal premissa, foi possível verificar que são diversos os problemas que apresentam associações fortes e que podem ser analisados utilizando regras de associação. Assim como há padrões frequentes nas compras das pessoas, há também padrões frequentes

nos locais de visitação destas, isto é, após visitar determinado local, a pessoa acaba se deslocando até um segundo lugar, e assim por diante.

É alicerçado nesse aspecto que o projeto irá se desenvolver, se propondo a implantar regras de associação para analisar e sugerir pontos gastronômicos para visitação com base nas avaliações de seus usuários, sendo que o método de associação a ser utilizado será o algoritmo *A priori*, implementado na linguagem Java. Maiores informações sobre este algoritmo podem ser encontradas no capítulo 2.2.

Para facilitar a utilização dos usuários, a aplicação será desenvolvida sob a plataforma móvel Android.

O mercado de celulares está crescendo cada vez mais. Estudos mostram que hoje em dia mais de três bilhões de pessoas possuem um aparelho celular, sendo que isso corresponde a mais ou menos metade da população mundial (LECHETA, 2013).

De acordo com o NetMarketShare (https://www.netmarketshare.com/), portal web que aponta estatísticas das tecnologias, o sistema operacional Android é atualmente o mais utilizado nos celulares. Ele consiste em uma plataforma de desenvolvimento para aplicativos móveis, baseada no sistema operacional Linux, com diversas aplicações instaladas e, ainda, um poderoso ambiente de desenvolvimento. Trata-se do primeiro projeto de código aberto de uma plataforma móvel e envolve um pacote com programas para celulares, *middleware*, aplicativos e interface do usuário (LECHETA, 2013).

## 1.2 OBJETIVOS

### 1.2.1 Objetivo Geral

Desenvolver um aplicativo móvel minerador de dados baseado na verificação de padrões frequentes, cuja finalidade será sugerir opções de locais gastronômicos para visita, com base nas avaliações dos usuários.

### 1.2.2 Objetivos Específicos

- Contribuir com os usuários da aplicação, oferecendo praticidade e facilitando sua busca por locais gastronômicos, com base em seu perfil de preferências.
- Inferir regras de associação utilizando o algoritmo *A priori*.
- Alimentar a base de conhecimento com as avaliações obtidas de cada usuário.

## 1.3 JUSTIFICATIVA

A mineração de dados é uma importante ferramenta de obtenção de conhecimento e suporte a decisão, portanto convém aplicá-las em situações cotidianas. Nesse contexto, pode-se executá-la no auxílio a pessoas em trânsito, ou seja, se deslocando a procura de opções de visita, sugerindo locais tendo como base as ações presentes e passadas e correlacionando-as com ações de terceiros com perfis similares.

Relacionado a isso, alguns dados estatísticos estimulam a presente pesquisa, como por exemplo, o fato de que inúmeros estudantes – universitários principalmente – deixam seus lares para estudar fora de suas cidades. De acordo com Borges (2011), em matéria publicada no portal IG, mais de 11 mil candidatos do SISU (Sistema de Seleção Unificada) deixaram suas casas por conta dos estudos. Outro índice, apontado pelo G1, indica que, em 2014, na Universidade de Brasília, 36% dos aprovados não eram do Distrito Federal.

Além disso, o fator da inovação foi levado em consideração, visto que, nos referenciais teóricos, foram encontrados poucos softwares com características semelhantes às que serão propostas. Como exemplo desse tipo de aplicação, pode ser citado o *TripAdvisor* (https://www.tripadvisor.com.br/), uma ferramenta web que permite ao usuário verificar as melhores opções de visitação, em vários segmentos, de acordo com as avaliações dos demais usuários.

Por fim, devido a grande quantidade de usuários e aos inúmeros recursos oferecidos pelo ambiente no que diz respeito ao desenvolvimento, foi escolhida a plataforma Android para o desenvolvimento do aplicativo.

## 1.4 ESTRUTURA DO TRABALHO

O presente texto está organizado em capítulos, sendo este o primeiro, expondo as considerações iniciais, os objetivos e a justificativa da pesquisa.

O capítulo 2 apresenta o referencial teórico, destacando principalmente os conceitos de mineração de dados e da computação móvel para tecnologia Android. As ferramentas e tecnologias utilizadas no desenvolvimento do projeto serão destacadas no capítulo 3.

No capítulo 4, serão exibidos os resultados obtidos com a realização deste trabalho. No capítulo subsequente, são descritas as conclusões e, como último item, são destacadas as referências bibliográficas utilizadas.

# 2 REFERENCIAL TEÓRICO

## 2.1 MINERAÇÃO DE DADOS E O KDD

A constante evolução da computação, relacionada especialmente às tecnologias de coleta, armazenamento e transmissão de dados, ao aumento da velocidade de comunicação e à redução dos custos associados a estas tecnologias, tem proporcionado às organizações a capacidade de armazenar e processar suas operações de maneira detalhada. No entanto, isto pode gerar, em pouco tempo, grandes quantidades de dados, podendo exceder, em muito, a capacidade humana de realizar análises e identificar informações úteis (MELANDA, 2015).

Por sua vez, o conceito de Mineração de Dados está se tornando cada vez mais popular como uma ferramenta de descoberta de informações, que podem revelar estruturas de conhecimento a fim de guiar decisões. Dentre as mais variadas definições sobre o tema, pode-se destacar a de Berry e Linoff (1997), que caracterizam a mineração como a exploração e análise de dados, por meios automáticos ou semiautomáticos, em grandes quantidades, com o objetivo de descobrir regras ou padrões interessantes.

Paralelo a esse pensamento, Weis et. al (1999) definem a mineração de dados como a busca de informações valiosas em grandes bancos de dados. É também um esforço de cooperação entre seres humanos e computadores, com os primeiros projetando bancos de dados, descrevendo os problemas e definindo seus objetivos e os computadores verificando dados e procurando padrões que casem com as metas estabelecidas.

Outro ponto importante a ser relatado é a correlação entre a mineração de dados e o KDD; Enquanto o KDD compreende todo o processo de extração de conhecimento, a mineração caracteriza-se por ser uma de suas etapas (CARVALHO, 2005). Alguns autores consideram estes termos referentes a processos distintos (FAYYAD ET AL, 1996).

Neste projeto, os conceitos de KDD e *data mining* serão utilizados indistintamente como referência ao processo de extração de conhecimento a partir de dados. Esta abordagem é adotada em (Rezende et. al, 2003) e justifica-se pelo agrupamento de algumas etapas e pela dimensão das bases de dados usualmente utilizadas.

Vale destacar a definição de Fayyad et. al (1996) a respeito do KDD, sendo que este destaca-o como um processo, de várias etapas, não trivial, interativo e iterativo, para identificação de padrões compreensíveis, válidos, novos e potencialmente úteis a partir de grandes conjuntos de dados. Para melhor compreensão do conteúdo dessa definição, Melanda

(2005) sugere considerar individualmente cada componente da citação e os define da seguinte forma:

- Dados: Conjunto de fatos ou casos em um repositório. Por exemplo, dados correspondem aos valores dos campos de um registro de vendas em uma base de dados qualquer;
- Padrões: Denota alguma abstração de um subconjunto dos dados em alguma linguagem descritiva de conceitos;
- Processo: A extração de conhecimento de bases de dados;
- Válidos: Os padrões descobertos devem satisfazer funções que garantam que os exemplos cobertos e os casos relacionados ao padrão encontrado sejam aceitáveis;
- Novos: Padrões identificados devem fornecer novas informações sobre os dados. O grau da novidade é um indicador da originalidade desses padrões;
- Úteis: Os padrões descobertos devem ser incorporados e utilizados;
- Compreensíveis: Um dos objetivos da realização do processo de mineração de dados é identificar padrões que possam ser compreendidos pelos usuários permitindo uma análise mais aprofundada de base de dados;
- Conhecimento: O conhecimento é dependente do domínio da aplicação e está relacionado com a capacidade de inovação de um conceito, sua utilidade e inovação para o usuário.

A Figura 1 ilustra as etapas do processo do KDD.


**Figura 1: Etapas do KDD**
**Fonte: Rezende et al. (2003).**

Diversas são as aplicabilidades destes conceitos a fim de gerar resultados, mas é no âmbito comercial que ele é mais difundido. Prova disso está no surgimento da mineração de dados, visto que esta foi proposta pela primeira vez em 1993, por Agrawal et. al. para uma análise de cestas de supermercados. Este processo era uma verificação do comportamento de compra dos clientes para descobrir associações entre diferentes itens que os clientes carregam em suas cestas de compras (AGRAWAL ET AL., 2010).

Tal aspecto justifica-se devido à necessidade de grandes empresas desejarem transformarem sua massa de dados em informações úteis – outro aspecto relevante do KDD. Empresas de diversos segmentos geram diariamente uma quantidade de dados considerável sobre seus serviços e clientes. Logo, esses dados são passíveis de análise – através da mineração – e obtenção de resultados.

Complementando a questão da aplicabilidade, é prudente destacar um exemplo prático, e este pode ser visto na rede americana *Wal-Mart*, que descobriu que pessoas que vão as suas lojas às quintas-feiras para comprar fraldas *Huggies*, tendem a adquirir dezenove itens adicionais. Assim, toda quinta-feira a Wal-Mart altera a disposição dos produtos de suas lojas para assegurar que os compradores de *Huggies* encontrem os tais dezenove produtos (MENCONI, 1998).

O mecanismo aplicado no exemplo supracitado é definido como “regra de associação”. Este nada mais é do que a busca por padrões de combinações fortes, e estas, para Vasconcelos e Carvalho (2004), têm como premissa básica encontrar elementos que implicam

na presença de outros elementos em uma mesma transação, ou seja, encontrar relacionamentos ou padrões frequentes entre conjuntos de dados. Ou ainda, uma regra de associação é uma expressão da forma A → B, onde A e B são *itemsets.* Imaginando que A e B são conjuntos de produtos, a ideia por trás destra regra é: pessoas que consomem o produto A têm a tendência de também consumir o produto B.

No decorrer das definições dos termos que envolvem as regras de associação, outros conceitos são aplicados e merecem ser destacados individualmente, visto que são termos chave no processo de KDD e da mineração. Transação, por exemplo, para Goldschmidt e Passos (2005) é um nome atribuído ao elemento de ligação que existe em cada ocorrência de itens no banco de dados.

Ainda para Goldschmidt e Passos (2005), uma associação é considerada frequente se o número de vezes em que a união de *itemsets* ocorrer em relação ao número total de transações do banco de dados for superior a uma frequência mínima, chamada de suporte mínimo, sendo que esta é estabelecida em cada aplicação. Tal definição é utilizada para identificar quais associações surgem em uma quantidade expressiva a ponto de ser destacada das demais.

Por sua vez, vale mencionar a forma com que uma associação é considerada válida. Goldschmidt e Passos (2005) descrevem que isso ocorre quando o número de vezes em que a união de dois *itemsets*, denotados por X e Y, ocorrer em relação ao número de vezes que o primeiro *itemset* ( X ) ocorrer for superior a um valor denominado de confiança mínima, sendo que este também é especificado em cada aplicação. Busca-se através da confiança expressar a qualidade de uma regra, indicando o quanto uma ocorrência pode assegurar uma segunda ocorrência.

Ainda a respeito dos conceitos de suporte e confiança, Melanda (2015) define tais aspectos como:

- Suporte: Quantifica a incidência de um *itemset* em um conjunto de dados, ou seja, indica a frequência com que um *itemset* ocorre em relação ao total de transações de uma base de dados.
- Confiança: Indica a frequência com que determinados *itemsets* ocorrem juntos em relação ao número total de transações em que o primeiro *itemset* especificado ocorre.

Esses valores de suporte e confiança mínimos, para Goldschmidt e Passos (2005), devem ser especificados por um especialista em KDD juntamente com o especialista no

contexto da aplicação. Tal premissa justifica-se pelo conhecimento que ambos detêm, a fim de gerar regras consistentes e usuais.

## 2.2 ALGORITMO *A PRIORI*

Existem diversos algoritmos cuja tarefa é a descoberta de associações. Historicamente, o primeiro algoritmo proposto para se aplicar o conceito foi o AIS (Agrawal et al 1993), sendo este um algoritmo de vários ciclos em que um *itemset* é gerado enquanto se verifica uma base de dados. A cada transação, o *itemset* frequente é incrementado com os itens gerados.

No entanto, o grande problema do algoritmo AIS é que são gerados muitos itens que mais tarde acabam sendo pouco frequentes. Outra desvantagem do algoritmo é que a estrutura de dados necessária para manter os *itemsets* frequentes e os que são usados como base para a mineração não foi especificada. (AGRAWAL ET AL., 2010)

O desejo de se utilizar a linguagem SQL para geração de *itemsets* frequentes resultou na introdução de um novo algoritmo de mineração, conhecido por SETM. Este algoritmo representa cada membro dos *itemsets* base/frequentes na forma <ID, itemset> onde o ID é um identificador único da transação. O problema do algoritmo SETM é que ele também realiza múltiplas verificações no banco de dados, assim como o algoritmo AIS. (AGRAWAL ET AL., 2010)

Agrawal e Srikant (1994) observaram que o principal problema que surgiu com o algoritmo SETM é devido ao número de *itemsets* base para análise. Isto porque, para cada conjunto de itens tem-se um identificador associado, e isso requer um espaço maior para armazenar o grande número de identificadores. Além disso, Sarawagi et. al. (1998) mencionam que o algoritmo é ineficiente.

Os algoritmos AIS e SETM foram seguidos pelo algoritmo *A priori,* que mostrou ter desempenho superior ao de ambos. O algoritmo *A priori* é um algoritmo clássico de mineração de regras de associação (Agrawal, 1993). Para Goldschmidt e Passos (2005, p. 105),

> "[...] diversos algoritmos foram inspirados no funcionamento do *A priori* e se baseiam no princípio da antimonotonicidade do suporte, que diz que "um itemset somente pode ser frequente se todos os seus subconjuntos forem frequentes".

> Desta forma, a combinação de *itemsets* para gerar um novo *itemset* somente ocorre quando estes são frequentes".

O algoritmo *A priori*, de acordo com Assunção (2012, p. 11) em sua tese de mestrado,

> "[...] Foi o primeiro algoritmo de mineração de regras de associação que explorou o uso do suporte para controlar o crescimento da quantidade de *itemsets* candidatos".

Para se utilizar o algoritmo, devem-se seguir alguns passos que encaminharão ao resultado. Inicialmente, o usuário deve especificar os valores de suporte e confiança mínimos. Com isso, o algoritmo irá encontrar todos os conjuntos de itens frequentes, desde que satisfaçam a condição de suporte mínimo. A partir desse conjunto de itens, o algoritmo gera as regras de associação que satisfazem à condição de confiança mínima. (GOLDSCMIDT E PASSOS, 2005).

A respeito da execução do algoritmo, Goldschmidt e Passos (2005) citam que como a primeira tarefa – encontrar os conjuntos de itens frequentes – demanda maior custo computacional e, com todos os *itemsets* frequentes gerados, a segunda tarefa – geração de regras de associação – se torna mais imediata, e grandes esforços de otimização tem sido concentrados na primeira tarefa.

A parte da geração de *itemsets* frequentes do algoritmo segue duas características importantes. A primeira diz que o *A priori* trata-se de um algoritmo que aumenta gradativamente o tamanho de seus *itemsets* frequentes, começando do menor tamanho possível até o máximo que se pode chegar. Já a segunda discorre que ele emprega uma estratégia chamada gerar-e-testar, isto é, como a busca por *itemsets* frequentes é iterativa, a cada ciclo são gerados novos *itemsets* frequentes, assim como são eliminados os itemsets que não satisfazem os valores mínimos – ou de suporte, ou de confiança.

Para exemplificar o funcionamento do algoritmo, a seguir são descritos alguns passos e exemplos definidos por Goldschmidt e Passos (2005). Inicialmente, deve-se ter uma base a ser explorada, como no Quadro 1:

| Transação | Leite | Café | Cerveja | Pão | Manteiga | Arroz | Feijão |
|---|---|---|---|---|---|---|---|
| **1** | Não | Sim | Não | Sim | Sim | Não | Não |
| **2** | Sim | Não | Sim | Sim | Sim | Não | Não |
| **3** | Não | Sim | Não | Sim | Sim | Não | Não |
| **4** | Sim | Sim | Não | Sim | Sim | Não | Não |
| **5** | Não | Não | Sim | Não | Não | Não | Não |
| **6** | Não | Não | Não | Não | Sim | Não | Não |
| **7** | Não | Não | Não | Sim | Não | Não | Não |
| **8** | Não | Não | Não | Não | Não | Não | Sim |
| **9** | Não | Não | Não | Não | Não | Sim | Sim |
| **10** | Não | Não | Não | Não | Não | Sim | Não |

**Quadro 1: Amostra de base de conhecimento**

Para geração das regras, são definidos os valores 0.3 e 0.8 como suporte e confiança mínimos, respectivamente. A primeira etapa – encontrar os *itemsets* frequentes – é iterativa, ou seja, se repete diversas vezes. Além disso, o tamanho dos *itemsets* analisados sofre aumentos a cada ciclo. Na Figura 2 é destacada a fórmula aplicada aos *itemsets* durante a primeira etapa do algoritmo. Esta fórmula é denominada de cálculo do suporte.

$$Suporte = \frac{Freqüência\ de\ X\ e\ Y}{Total\ de\ T}$$

**Figura 2: Fórmula do suporte**

No exemplo da base acima, na primeira iteração serão identificados os *itemsets* frequentes em uma análise dos conjuntos de um elemento apenas, sendo que seu suporte deve ser maior que o mínimo especificado. Para tanto, a fórmula da Figura 2 é aplicada, verificando o número de ocorrências de cada *itemset*, ou seja, quando os registros são destacados com valor "Sim" e dividindo pelo número total de transações. Adotando o *itemset* {Leite}, como exemplo, identificamos que, de 10 transações, apenas 2 possuem seus conteúdos destacados como "Sim". Ou seja, aplicando a fórmula temos que o suporte deste *itemset* é 0.2. No Quadro 2 são apresentados os resultados de todos os *itemsets*.

| ***Itemsets*** | **Suporte** |
|---|---|
| Leite | 0.2 |
| Café | 0.3 |
| Cerveja | 0.2 |

| | |
|---|---|
| Pão | 0.5 |
| Manteiga | 0.5 |
| Arroz | 0.2 |
| Feijão | 0.2 |

**Quadro 2: Resultado da primeira iteração do *A priori***

Para segunda iteração, apenas os *itemsets* com valor de suporte maior que o mínimo proposto são utilizados. No exemplo, todos os *itemsets* com valor de suporte menor que 0.3 são descartados, restando apenas os elementos listados no Quadro 3.

| ***Itemsets*** | **Suporte** |
|---|---|
| Café | 0.3 |
| Pão | 0.5 |
| Manteiga | 0.5 |

**Quadro 3: *Itemsets* com valor de suporte maior que o mínimo (0.3)**

Esses *itemsets*, por sua vez, devem ser agrupados a fim de gerar conjuntos de dois elementos para que, na sequência, possa se aplicar a fórmula do suporte novamente. O Quadro 4 exibe o resultado.

| ***Itemsets*** | **Suporte** |
|---|---|
| Café, Pão | 0.3 |
| Café, Manteiga | 0.3 |
| Pão, Manteiga | 0.4 |

**Quadro 4: Resultado da segunda iteração do *A priori***

Na terceira iteração, são combinados os *itemsets* frequentes gerados na iteração anterior para gerar conjuntos de três itens. Como todos os conjuntos possuem valor de suporte maior que o mínimo, todos são utilizados no próximo ciclo. Após definir o novo conjunto, de três elementos, aplica-se novamente o cálculo. O resultado é apresentado no Quadro 5.

| ***Itemsets*** | **Suporte** |
|---|---|
| Café, Pão, Manteiga | 0.3 |

**Quadro 5: Resultado da terceira iteração do *A priori***

Como um novo *itemset* não pode ser gerado devido ao tamanho atual do mesmo, a primeira etapa é encerrada e, por consequência, o conjunto de itens frequentes é determinado: Café, Pão, Manteiga. A próxima etapa que se espera do algoritmo é a geração das regras de associação.

Antes de iniciar a segunda etapa do algoritmo, dividimos o *itemset* frequente em novos conjuntos. Realizando essa divisão, temos que os *itemsets* sujeitos à geração das regras de associação são: Café e Pão; Café e Manteiga; Pão e Manteiga; e Café, Pão e Manteiga.

A finalidade da geração das regras de associação é a de identificar as regras válidas, ou seja, regras cujo valor de confiança é superior à mínima definida. Para tanto, são geradas todas as regras possíveis a partir de cada *itemset*. O cálculo da confiança é feito obtendo o número de transações onde ambos os elementos do *itemset* ocorrem e dividindo esse valor pelo número de transações onde apenas o primeiro elemento é presente, conforme a fórmula da Figura 3.

$$Confiança = \frac{Freqüência\ de\ X\ e\ Y}{Freqüência\ de\ X}$$

**Figura 3: Fórmula da confiança**

Na sequência são apresentados os resultados obtidos na geração das regras:

*Itemset*: {Café, Pão}.

| **Regra** | **Confiança** |
|---|---|
| SE café ENTÃO pão | 1.0 |
| SE pão ENTÃO café | 0.6 |

**Quadro 6: Possíveis regras para o *itemset* {Café, Pão}**

*Itemset*: {Café, Manteiga}.

| **Regra** | **Confiança** |
|---|---|
| SE café ENTÃO manteiga | 1.0 |
| SE manteiga ENTÃO café | 0.6 |

**Quadro 7: Possíveis regras para o *itemset* {Café, Manteiga}**

*Itemset*: {Pão, Manteiga}.

| **Regra** | **Confiança** |
|---|---|
| SE pão ENTÃO manteiga | 0.8 |
| SE manteiga ENTÃO pão | 0.8 |

**Quadro 8: Possíveis regras para o *itemset* {Pão, Manteiga}**

*Itemset*: {Café, Pão, Manteiga}.

| Regra | Confiança |
|---|---|
| SE café, pão ENTÃO manteiga | 1.0 |
| SE café, manteiga ENTÃO pão | 1.0 |
| SE manteiga, pão ENTÃO café | 0.75 |
| SE café ENTÃO pão, manteiga | 1.0 |
| SE pão ENTÃO café, manteiga | 0.6 |
| SE manteiga ENTÃO café, pão | 0.6 |

**Quadro 9: Possíveis regras para o *itemset* {Café, Pão, Manteiga}**

Após filtrar as regras, deixando apenas as que o valor da confiança é maior que o mínimo proposto (0.8), são produzidas as seguintes regras de associação:

| Regras |
|---|
| SE café ENTÃO pão |
| SE café ENTÃO manteiga |
| SE manteiga ENTÃO pão |
| SE pão ENTÃO manteiga |
| SE café, pão ENTÃO manteiga |
| SE café, manteiga ENTÃO pão |
| SE café ENTÃO pão, manteiga |

**Quadro 10: Resultado final da execução do algoritmo**

## 2.4 ANDROID

Tanto em computadores como em aparelhos móveis, os sistemas operacionais têm por objetivo tornar a utilização destes equipamentos mais eficazes e práticas, isto porque um SO oferece diferentes recursos para manipulação de tarefas nativas dos dispositivos (OLIVEIRA ET AL., 2001). Restringindo o conceito apenas aos dispositivos móveis, alguns SO podem ser destacados, haja vista a sua ampla aceitação por parte dos usuários e os recursos oferecidos. São eles: Android, IOS, Symbian e Windows Mobile.

O Android, por sua vez, é um conjunto de *softwares* para dispositivos móveis, que inclui um sistema operacional, *middleware* e vários aplicativos importantes (COMACHIO, 2011). Além disso, Lecheta (2013) cita que, o fato de o Android ser de código aberto, colabora muito para seu aperfeiçoamento, uma vez que desenvolvedores podem contribuir para seu código-fonte.

A Figura 4 demonstra as camadas de componentes que formam o Android.

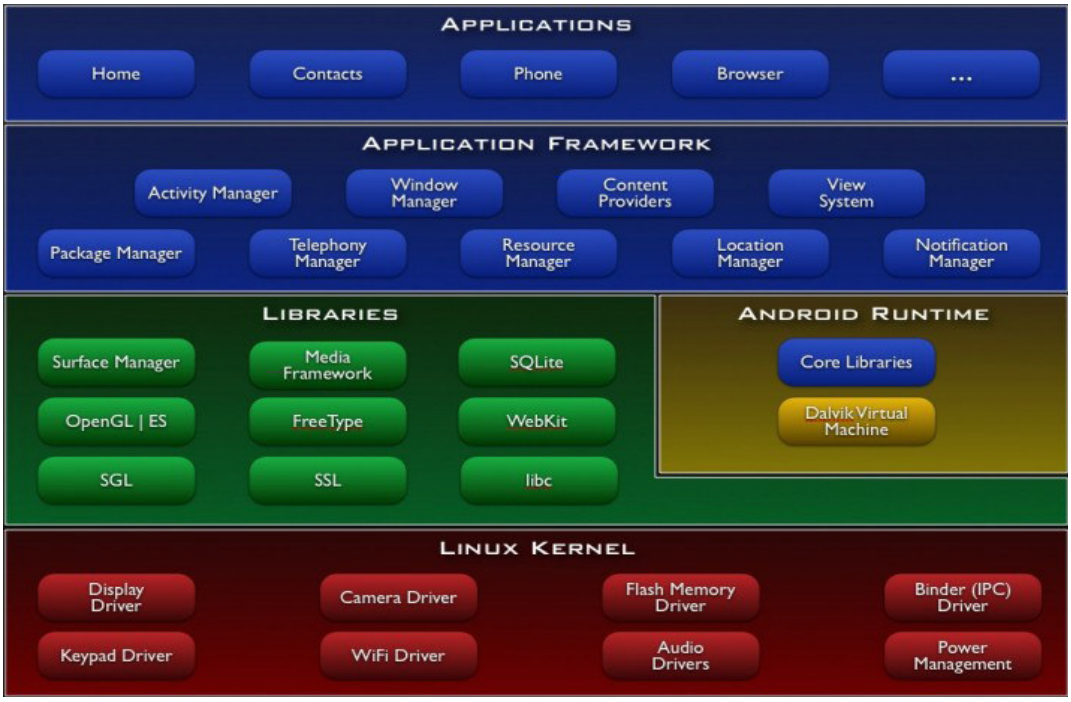


**Figura 4: Camadas da arquitetura Android**
**Fonte: TOSIN, Carlos (2011).**

As camadas destacadas na Figura 4 são:

- *Linux Kernel*: O Android é baseado no *kernel* do Linux, da versão 2.6, sendo que este foi otimizado, deixando de lado algumas características inapropriadas ao ambiente *mobile*. Essa escolha se deve ao fato de que o *kernel* do Linux possui uma considerável quantidade de drivers para dispositivos, que são maduros e confiáveis, além de um bom gerenciamento de memória e de processos (GIROLLETE, 2012).
- *Libraries* – Bibliotecas: Nesta camada encontram-se as principais bibliotecas utilizadas pelo Android, dentre elas a SQLite, que permite manipular a base de dados (PRADO, 2011).
- *Application Framewors:* Desenvolvida basicamente em Java, esta camada é responsável pela interface com as aplicações Android. Ela provê um conjunto de bibliotecas para acessar os diversos recursos do dispositivo como interface gráfica, GPS, banco de dados persistente, etc. Fornece blocos de alto nível de construção utilizados para a criação de aplicações. O framework vem pré-instalado com o Android (MOBILEIN, 2010).

- *Applications* – Aplicações: Nesta camada ficam as aplicações – desenvolvidas em Java – para o Android.

Discorrendo sobre a segurança do Android, Lecheta (2013) cita que, por ser baseada na segurança do Linux, cada aplicação é executada em um único processo sendo que cada um possui uma *thread* dedicada. Além disso, declara ainda que para cada aplicação instalada no celular, é criado um usuário no SO a fim de prover acesso a sua estrutura de diretórios.

A respeito do desenvolvimento de aplicações Android, vale destacar que estas utilizam a linguagem Java. No entanto, não existe uma máquina virtual Java (JVM) para a construção das aplicações, o que se tem é uma máquina virtual chamada Dalvik que é otimizada para a execução em dispositivos móveis (LECHETA, 2013). Traçando um paralelo com sua arquitetura, a Dalvik encontra-se na camada *Android Runtime*, conforme destacado na Figura 4.

Complementando a ideia, Lecheta (2013) ainda cita que, ao desenvolver aplicações para o Android, utiliza-se a linguagem Java e todos os seus recursos normalmente, mas depois que o *bytecode* é compilado, ele é convertido para o formato *.dex,* que simboliza a aplicação do Android compilada. Após essa etapa, esse arquivo é compactado juntamente com outros recursos que por ventura foram utilizados na aplicação, como imagens, gerando um único arquivo com a extensão *.apk,* sendo que esse simboliza a aplicação pronta.

Desde o seu lançamento, o Android passou por várias atualizações de versões. O Quadro 11 descreve brevemente a evolução das versões do Android.

| Versão | Descrição |
|---|---|
| 1.0 | Primeira versão lançada (SOARES, 2011). Como uma das características, apresentou suporte a câmera – com funcionalidades reduzidas. |
| 1.1 | Considerada a primeira versão de verdade. Como funcionalidade acrescentada, vale destacar a possibilidade de enviar e salvar anexos em mensagens (SOARES, 2011). |
| 1.5 – Cupcake | Trouxe recursos de transferência de vídeo diretamente para o *youtube*. Além disso, envio de fotos para o *Picasa* e presença de *widgets* (SOARES, 2011). |
| 1.6 – Donut | Trouxe uma interface nova e o suporte a câmeras para fotografia e vídeo (SOARES, 2011). |
| 2.0/2.1 – Eclair | Nova interface e um aplicativo de contatos, suporte para câmeras com flash, Zoom digital, Bluetooth 2.1, entre outros (SOARES, 2011). |
| 2.2 – Froyo | Motor de JavaScript ao navegador, suporte a GIF no navegador, otimizações de memória, desempenho e velocidade, suporte à instalação de aplicativos na memória externa, como cartões de memória, etc (SOARES, 2011). |
| 2.3 – GingerBread | Economia de Energia, suporte nativo a protocolos de telefonia via *internet* SIP e VoIP, suporte para múltiplas câmeras no dispositivo, entre outros (SOARES, 2011). |
| 3.0/3.1/3.2 – Honeycomb | Suporte para processadores multi-core, capacidade de criptografar todos os dados do usuário, interface otimizada para o uso em tablets, conectividade para acessórios USB, etc (SOARES, 2011). |
| 4.0 – Ice Cream | Pastas *drag-and-drop*, captura de tela integrada através dos botões de energia e |

| | |
|---|---|
| Sandwich | de volume, aceleração de hardware da interface do usuário, melhorias para gráficos, banco de dados, correções ortográficas e funcionalidade Bluetooth, editor de fotos nativo, Android Beam, entre outros (SOARES, 2011). |
| 4.1/4.2/4.3 – Jelly Bean | Áudio multicanal, Áudio USB conversores digital-analógico, melhoria da aplicação da câmera, buffer triplo para gráficos, recursos de auto-completar na discagem de telefones, etc (SOARES, 2011). |
| 4.4 – Kitkat | Suporte ao GPS, melhorias de áudio, como monitoramento e potencializador de volume máximo, mais opções de acessibilidade, como estatísticas de bateria, capacidade de impressão sem fio, entre outros (SOARES, 2011). |
| 5.0 - Lollipop | Suporte para CPUs de 64 bits, gráficos vetoriais, que melhoram a definição de imagens, entrada e saída de áudio através de dispositivos USB, suporte para múltiplos cartões SIM, etc (SOARES, 2011). |

**Quadro 11: Versões Android**

# 3 MATERIAIS E MÉTODOS

A ênfase deste capítulo está em apresentar as ferramentas utilizadas para o desenvolvimento do trabalho. Está subdividido em duas seções, sendo uma para destacar os materiais e outra para o método.

## 3.1 MATERIAIS

O Quadro 12 expõe as ferramentas e tecnologias para modelar e implementar o sistema.

| **Ferramenta/Tecnologia** | **Versão** | **Referência** |
|---|---|---|
| Astah Community | 6.9.0 | http://astah.net/editions/community |
| Visual Paradigm | 11.1 | https://www.visual-paradigm.com/ |
| Java | JDK 1.8 | http://www.oracle.com |
| Eclipse | Luna | https://eclipse.org/ |
| Apache Tomcat | 8.0 | http://tomcat.apache.org/ |
| Spring Boot | 1.3.6 | https://spring.io/ |
| Hibernate | 5.0.12.Final | http://www.hibernate.org/ |
| PostgreSQL | 9.4 | https://www.postgresql.org/ |
| Android Studio | 2.1.2 | https://developer.android.com/studio/index.html |
| SQLite | x | https://www.sqlite.org/ |
| ORMLite | 5.0 | http://ormlite.com/ |
| Firebase | x | https://firebase.google.com/?hl=pt-br |
| Git | 2.11.1 | https://git-scm.com/ |
| Github | x | https://github.com/ |

**Quadro 12: Ferramentas e tecnologias utilizadas**

O Astah Community e o Visual Paradigm são ferramentas voltadas para UML, que disponibilizam a criação de diversos diagramas da UML 2.0, tais como diagrama de casos de uso, diagrama de atividades, diagrama de entidades e relacionamentos, entre outros.

Java nada mais é do que uma linguagem de programação desenvolvida na década de 90 por uma equipe de programadores liderada por James Golling, na empresa Sun Microsystems, cuja principal característica é a orientação a objetos. Atualmente, a linguagem é mantida pela Oracle Corporation.

O Eclipse trata-se de uma IDE para desenvolvimento Java, porém com suporte a várias outras linguagens de programação, como PHP, Python e C, por exemplo. Tem como característica marcante o amplo suporte ao desenvolvedor com vários *plug-ins* que atendem a diferentes necessidades.

O Apache Tomcat é um servidor web Java *open source*, que implementa tecnologias como Java *Servlet*, JavaServer *Pages*, Java *Expression Language* e Java WebSocket.

O Spring Boot é uma especificação do *framework* Spring. Herda as principais características do Spring (inversão de controle e injeção de dependências), porém possui como destaque o fato de ser mais facilmente configurado e preparado para o desenvolvimento.

O PostgreSQL e o SQLite são bancos de dados relacionais. O primeiro possui recursos como integridade transacional, *triggers*, *views*, linguagem procedural, entre outros. Já o segundo possui como característica principal o fato de ser suportado nativamente pelos dispositivos Android.

O Hibernate e o ORMLite são *frameworks* utilizados para fazer o mapeamento objeto-relacional. Eles facilitam o mapeamento dos atributos entre uma base de dados relacional e o modelo objeto de uma aplicação, ou seja, as tabelas do banco de dados são representadas através de classes na aplicação. Esse mapeamento é feito por meio de anotações Java.

O Android Studio, por sua vez, é a IDE oficial para desenvolvimento de aplicações Android.

O Firebase trata-se de um conjunto de serviços na nuvem oferecidos pela Google, com o objetivo de prover novas funcionalidades para aplicações móveis e *web*. Neste projeto, o serviço utilizado foi o de "notificações *push*", que nada mais é do que um mecanismo que faz com que os dispositivos móveis sejam "receptores" de mensagens enviadas através de requisições POST, do protocolo *HTTP*.

Por fim, o Git nada mais é do que um sistema de controle de versão de arquivos. Através dele é possível desenvolver projetos onde diversas pessoas podem contribuir

simultaneamente. Possui uma ferramenta *online* para armazenamento de repositórios denominada Github, sendo que esta também será usada neste projeto.

## 3.2 MÉTODO

Como método de desenvolvimento do projeto, foi utilizado o modelo iterativo e incremental. Tal modelo sugere uma abordagem para o desenvolvimento do projeto que consiste em iterações que incrementam o software a cada novo ciclo. Desta forma, o processo básico de análise de requisitos, seguido pelo desenvolvimento e pelos testes foi repetido diversas vezes. Assim, o método seguiu as seguintes etapas:

1) Análise de requisitos – O ciclo se iniciou com a etapa de levantamento de requisitos, onde foram delimitadas as necessidades da aplicação, suas restrições e as tecnologias mais adequadas para utilização no desenvolvimento. Nesta etapa também foi realizado um estudo de caso das tecnologias, aplicando o algoritmo *A priori* em uma base fictícia para garantir a efetividade do mesmo.

2) Modelagem do projeto – Finalizado o levantamento de requisitos, foi realizada a modelagem da arquitetura do projeto, através dos diagramas de casos de uso, de atividades e de entidades e relacionamentos.

3) Desenvolvimento – O desenvolvimento iniciou-se com a definição das classes de negócio mais básicas, implementando suas estruturas e regras de negócio. Em um primeiro momento, o foco foi codificar o *web service* para torná-lo apto a integrações com o aplicativo. Na sequência, foram organizadas as classes do aplicativo, desde as representações das telas até as regras de negócio. Alguns padrões de projeto foram adotados, como *Singleton*, *Delegate* e MVC, visando obter uma arquitetura mais organizada e compreensível. Para manter os históricos de alterações dos códigos-fontes, considerou-se de grande importância a utilização de um sistema de versionamento.

4) Testes – Os testes foram implantados a cada *release* do aplicativo e do *web service*, sendo delimitados pelo autor do projeto. Nessa etapa, buscou-se corrigir os *bugs* retornados pelos softwares, melhorar a definição das regras de negócio e facilitar a usabilidade do aplicativo por parte do usuário. Além disso, foram realizadas algumas experiências para forçar a exibição de erros e, com isso, analisar o comportamento do aplicativo perante essas inconsistências.

# 4 RESULTADOS

Este capítulo apresenta os resultados da realização deste trabalho. Os resultados estão centrados na modelagem do sistema e na implementação de funcionalidades básicas que tem o objetivo de apresentar o uso das tecnologias de implementação.

## 4.1 ESCOPO DO SISTEMA

A aplicação possibilitará ao usuário requisitar opções de locais para visitação baseados nos seus históricos de avaliações. Para isso, o projeto foi dividido em dois *softwares*: um aplicativo Android e um *web service* REST Java. O uso da aplicação ficará concentrado no aplicativo móvel, onde os usuários realizarão as avaliações dos estabelecimentos desejados e posteriormente solicitarão opções de visita, sendo que as requisições ao *web service* ocorrerão quando for necessário sincronizar as avaliações de todos os dispositivos ou quando um usuário solicitar indicações de locais para visita. Em ambos os casos, as requisições ao *web service* serão transparentes ao usuário.

A priori, a lista de estabelecimentos a ser exibida ficará restrita a pontos gastronômicos, tais como bares, lanchonetes e restaurantes, sendo que estes registros são inseridos manualmente no banco de dados do *web service* e sincronizados aos dispositivos regularmente.

As principais funcionalidades do aplicativo são:

a) Avaliação dos estabelecimentos;
b) Sincronização das avaliações realizadas nos dispositivos móveis ao banco de dados do *web service*;
c) Solicitação de locais para visita, execução da mineração de dados e exibição dos resultados.

**Avaliação dos estabelecimentos –** É a primeira ação que o aplicativo espera dos seus usuários. O usuário deverá realizar a avaliação utilizando-se de dois botões: “Gostei” e “Não Gostei”. Em um primeiro momento, estas avaliações serão salvas apenas no dispositivo, sendo que a sua persistência no *web service* ocorrerá apenas durante a sincronização dos dados. Caso uma avaliação seja realizada em um estabelecimento que já foi avaliado anteriormente pelo usuário, esta será atualizada. Novas avaliações serão incluídas apenas para estabelecimentos que ainda não foram avaliados pelo usuário.

**Sincronização das avaliações realizadas nos dispositivos móveis ao banco de dados do *web service* –** Será estabelecido um horário para que as avaliações salvas nos dispositivos sejam exportados para a base de dados do *web service*, que posteriormente será minerada. A estrutura do banco de dados do aplicativo será semelhante à estrutura do banco do *web service.* A diferença ficará apenas na tabela de usuários, uma vez que esta tabela não existe no banco de dados do dispositivo (o usuário será obtido através da conta vinculada ao aparelho), mas no banco de dados do *web service* sim. Isto é justificado pela premissa de que um dispositivo simboliza um usuário, portanto não seria performático ter uma tabela para salvar apenas um registro.

**Solicitação de locais para visita, execução da mineração de dados e exibição dos resultados –** Trata-se da ação principal do software. O usuário deverá executar esta ação através de um botão e é a partir disto que a mineração iniciará. Esta ação irá disparar um evento no *web service*, ou seja, será necessário que o usuário esteja conectado a uma rede de *internet* neste momento. O algoritmo executado na mineração é o *A priori,* escrito na linguagem Java. Os valores de suporte e confiança mínimos foram definidos em 0.3 e 0.8, respectivamente. A escolha por esses valores se deu devido ao tamanho da base de dados explorada. Os dados que serão minerados estarão persistidos em um banco de dados relacional. Ao finalizar a mineração, uma lista com os locais sugeridos será exibida no dispositivo.

## 4.2 MODELAGEM DO SISTEMA

O usuário da aplicação terá, como pré-requisito, que possuir uma conta vinculada ao Google. Tal afirmação justifica-se pelo fato de que para ser usuário de aparelhos Android existe essa mesma necessidade. Além disso, essa conta será utilizada como o identificador do usuário na aplicação, além de ser o mecanismo de *login*. Não existirão delimitadores de permissões de acesso, uma vez que a interatividade do usuário com a aplicação será com um objetivo extremamente específico.

A partir da inicialização do aplicativo, o usuário terá a disposição as seguintes ações: visualizar os estabelecimentos disponibilizados pelo aplicativo, avaliar os estabelecimentos, ver as informações detalhadas sobre determinado local e solicitar indicações de locais para visita.

A Figura 5 exibe o diagrama de casos de uso, sendo o ator um usuário da aplicação:

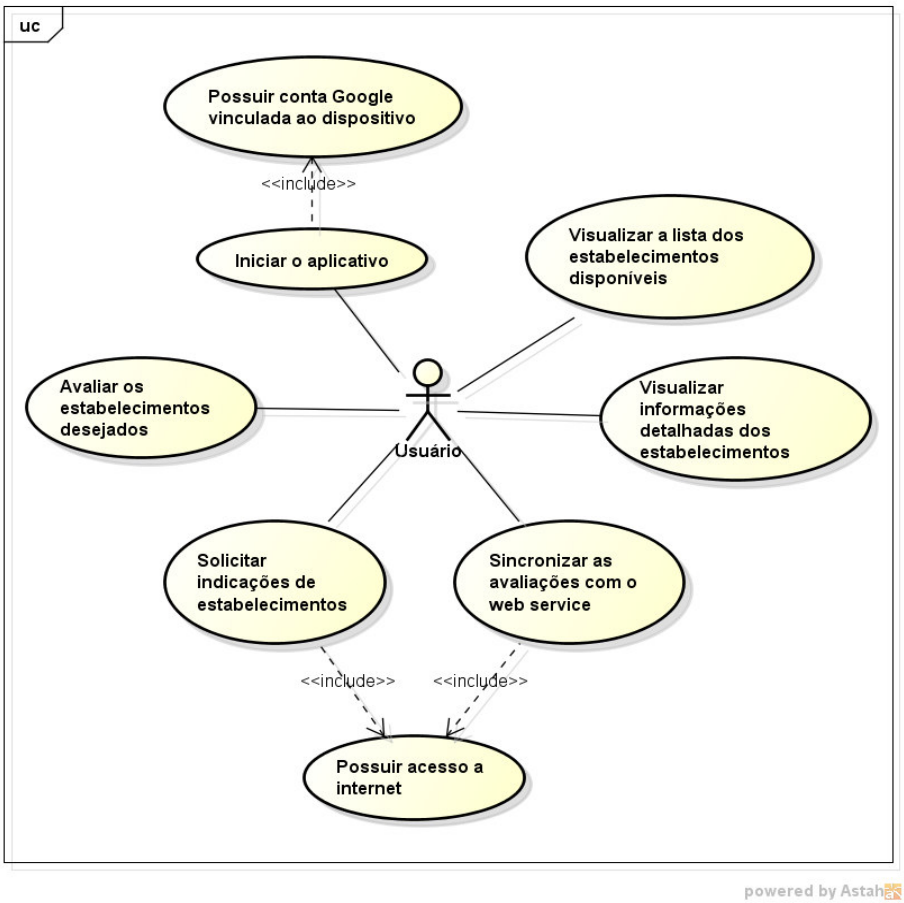


**Figura 5: Diagrama de casos de uso**

O Quadro 13 explicita os requisitos funcionais da aplicação, associados aos casos de uso supracitados. Neste Quadro, o termo RF significa requisito funcional.

| | Caso de Uso | Requisito | Descrição |
|---|---|---|---|
| RF01 | Iniciar o aplicativo | Identificar o usuário do aplicativo. | O mecanismo tradicional de *login* – com usuário e senha – não será aplicado. A forma de identificar os usuários será através da conta Google vinculada ao dispositivo. |
| RF02 | Avaliar os estabelecimentos desejados. | Salvar as avaliações dos estabelecimentos no banco de dados do dispositivo. | Todas as avaliações serão persistidas, em um primeiro momento, localmente. Apenas em segundo momento serão exportadas para o banco de dados do *web service*. |
| RF03 | Sincronizar as avaliações com o *web service*. | Salvar as avaliações dos estabelecimentos no banco de dados do *web service*. | Persistir no banco de dados do *web service* as avaliações de todos os usuários, para alimentar a base a ser minerada. |
| RF04 | Solicitar indicações de estabelecimentos. | Solicitar indicações de visita. | Requisitar ao *web service* a geração de indicações de locais para visita. O algoritmo *A priori*, implementado em Java, será executado neste momento. |
| RF05 | Ver informações detalhadas dos | Entrar em contato com o estabelecimento desejado. | Possibilitar que o usuário entre em contato através de chamadas |

| | | | |
|---|---|---|---|
| | estabelecimentos | | telefônicas com o estabelecimento desejado |
| RF06 | Visualizar a lista dos estabelecimentos disponíveis. | Pesquisar no aplicativo os estabelecimentos desejados. | Facilitar a navegação e busca por determinados estabelecimentos. |
| RF07 | Sincronizar as avaliações com o servidor. | Verificar no aplicativo se a conexão com o *web service* está funcionando. | Como algumas ações do aplicativo dependem de uma conexão com a *internet*, deve-se permitir que o usuário verifique se o aplicativo está conectado em alguma rede. |

**Quadro 13: Requisitos funcionais da aplicação**

No Quadro 14, são listados os requisitos não funcionais da aplicação. O termo RNF é usado com o significado de requisito não funcional.

| | Requisito | Descrição |
|---|---|---|
| RNF01 | Exibição dos principais dados dos estabelecimentos na listagem inicial. | Para facilitar a identificação do estabelecimento através de suas principais informações, na tela de listagem deverá ser exibido o nome e o endereço do local. |
| RNF02 | Conexão com a *internet* | Ocorrerá regularmente uma sincronização de registros entre o aplicativo e o *web service*. Por conta disso, é necessário que nestes momentos o dispositivo esteja conectado a *internet*. Além disso, a conexão com a *internet* é um requisito para a geração de indicações de visita, uma vez que a mineração é realizada no *web service*. |
| RNF03 | Mineração dos Dados | O processo de mineração não ocorrerá no aparelho físico, mas sim em um *web service*, visto que o seu banco de dados possuirá as avaliações de todos os usuários do aplicativo. |
| RNF04 | Permissões para recursos do dispositivo. | As solicitações de permissão para determinados recursos do aplicativo serão exibidas apenas quando ações requeridas pelo usuário necessitarem de tal ação. Desta forma, não serão concedidas permissões desnecessárias. |

**Quadro 14: Requisitos não-funcionais da aplicação**

Por sua vez, a Figura 6 exibe o diagrama de atividades do projeto.

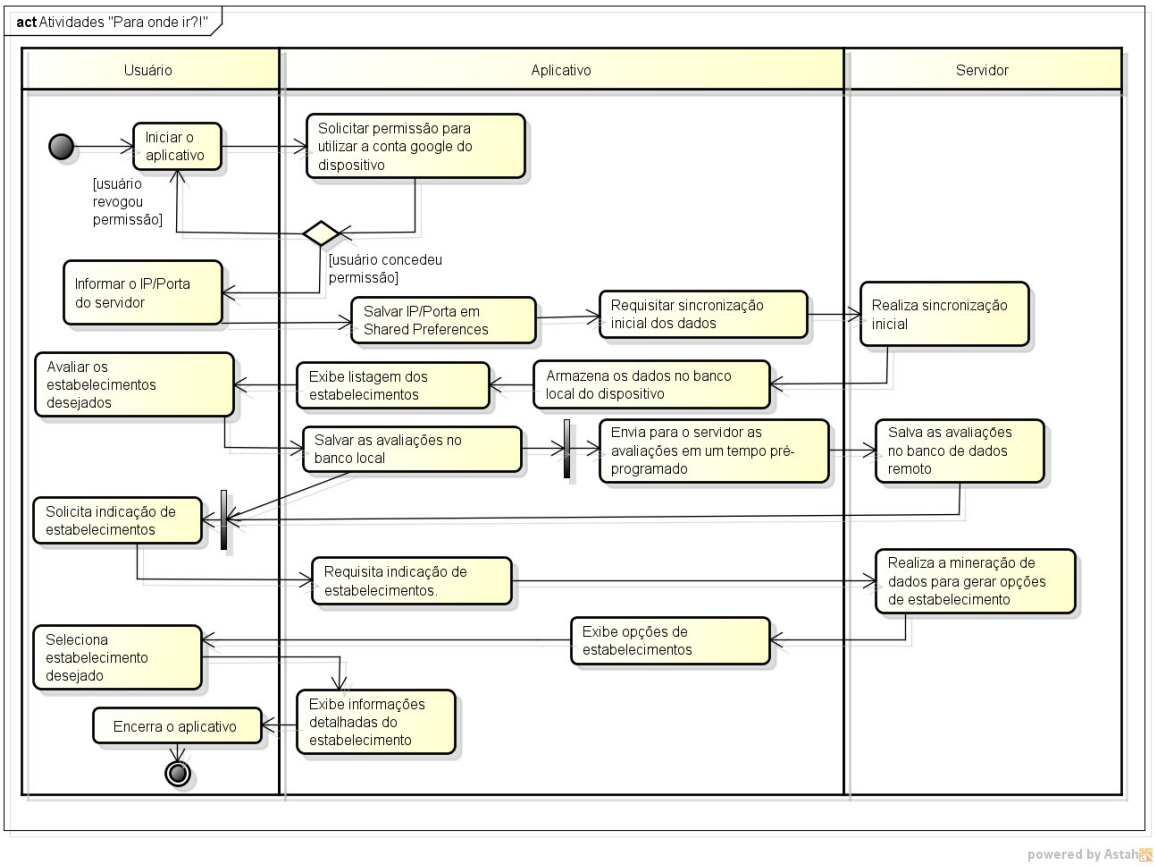


**Figura 6: Diagrama de atividades**

A Figura 7 apresenta o diagrama de entidades e relacionamentos do banco de dados do *web service*.

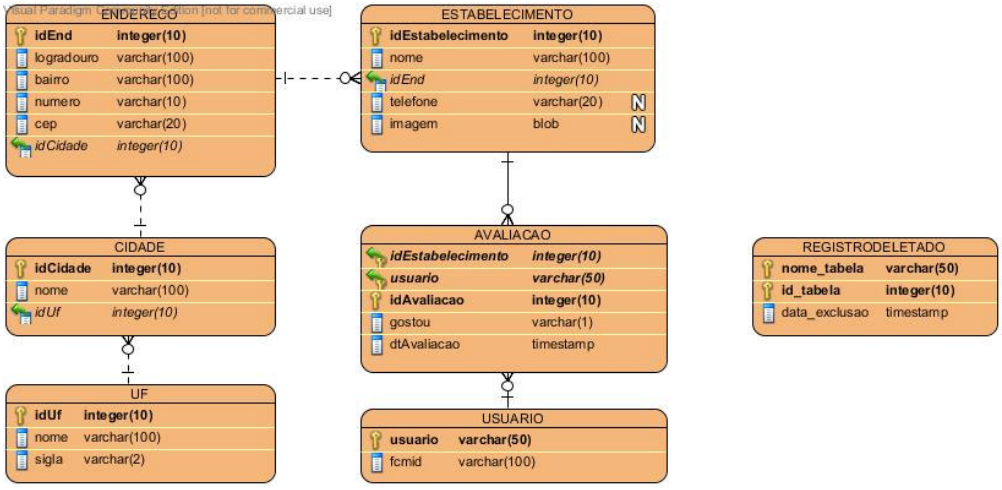


**Figura 7: Diagrama de entidades e relacionamentos do banco de dados do web service**

No diagrama é possível verificar a existência de uma tabela sem relação direta com nenhuma outra. A finalidade desta tabela será manter a consistência dos dados do *web service*

e do aplicativo, ou seja, garantir que os registros sejam os mesmos nos dois ambientes. Ela será valorizada através de *triggers*, disparadas no evento "*after delete"* de todas as outras tabelas do modelo.

O Quadro 15 apresenta a listagem dos campos da tabela de estados.

| **Campo** | **Tipo** | **Nulo** | **Chave primária** | **Chave estrangeira** | **Observações** |
|---|---|---|---|---|---|
| idUf | Numérico | Não | Sim | Não | |
| nome | Texto | Não | Não | Não | |
| sigla | Texto | Não | Não | Não | |

**Quadro 15: Tabela de estados**

O Quadro 16 exibe os campos presentes na tabela de cidades. Um estado pode conter várias cidades.

| **Campo** | **Tipo** | **Nulo** | **Chave primária** | **Chave estrangeira** | **Observações** |
|---|---|---|---|---|---|
| idCidade | Numérico | Não | Sim | Não | |
| nome | Texto | Não | Não | Não | |
| idUf | Numérico | Não | Não | Sim | Chave estrangeira da tabela de estados. |

**Quadro 16: Tabela de cidades**

O Quadro 17, por sua vez, lista os campos da tabela de endereços. Uma cidade pode conter vários endereços.

| **Campo** | **Tipo** | **Nulo** | **Chave primária** | **Chave estrangeira** | **Observações** |
|---|---|---|---|---|---|
| idEnd | Numérico | Não | Sim | Não | |
| logradouro | Texto | Não | Não | Não | |
| bairro | Texto | Não | Não | Não | |
| numero | Texto | Não | Não | Não | |
| cep | Texto | Não | Não | Não | |
| idCidade | Numérico | Não | Não | Sim | Chave estrangeira da tabela de cidades. |

**Quadro 17: Tabela de endereços**

O Quadro 18 apresenta a listagem dos campos da tabela de estabelecimentos. Um endereço pode conter vários estabelecimentos.

| **Campo** | **Tipo** | **Nulo** | **Chave primária** | **Chave estrangeira** | **Observações** |
|---|---|---|---|---|---|
| idEstabelecimento | Numérico | Não | Sim | Não | |
| nome | Texto | Não | Não | Não | |
| idEnd | Numérico | Não | Não | Sim | Chave estrangeira da tabela |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | de endereços. |
| telefone | Texto | Sim | Não | Não | |
| imagem | Imagem | Sim | Não | Não | |

**Quadro 18: Tabela de estabelecimentos**

O Quadro 19 apresenta a listagem dos campos da tabela de usuários.

| **Campo** | **Tipo** | **Nulo** | **Chave primária** | **Chave estrangeira** | **Observações** |
|---|---|---|---|---|---|
| usuario | Texto | Não | Sim | Não | |
| fcmId | Texto | Não | Não | Não | Identificador único de cada dispositivo gerado pelo serviço Firebase. |

**Quadro 19: Tabela de usuários**

O Quadro 20 exibe os campos da tabela de avaliações. Ela representa uma relação "muitos para muitos" entre as tabelas de estabelecimentos e usuários.

| **Campo** | **Tipo** | **Nulo** | **Chave primária** | **Chave estrangeira** | **Observações** |
|---|---|---|---|---|---|
| idAvaliacao | Numérico | Não | Sim | Não | |
| idEstabelecimento | Numérico | Não | Não | Sim | Chave estrangeira da tabela de estabelecimentos. |
| usuario | Texto | Não | Não | Sim | Chave estrangeira da tabela de usuários. |
| gostou | Texto | Não | Não | Não | |
| dtAvaliacao | Data/Hora | Não | Não | Não | |

**Quadro 20: Tabela de avaliações**

Por fim, o Quadro 21 apresenta a listagem dos campos da tabela de registros deletados.

| **Campo** | **Tipo** | **Nulo** | **Chave primária** | **Chave estrangeira** | **Observações** |
|---|---|---|---|---|---|
| nome_tabela | Texto | Não | Sim | Não | Armazena o nome físico da tabela que teve algum registro removido. |
| id_tabela | Numérico | Não | Sim | Não | Armazena o valor da chave primária do registro removido. |
| data_exclusao | Data/Hora | Não | Não | Não | |

**Quadro 21: Tabela de registros deletados**

Apesar de o projeto ter sido dividido em dois sistemas (aplicativo móvel e *web service*), a arquitetura dos bancos de dados se assemelha em quase todos os aspectos, até para facilitar a integração entre os dois ambientes. A diferença fica por conta da não existência das tabelas de usuários e de registros deletados no aplicativo.

A primeira tabela citada não foi criada pelo fato de não ser performático ter a sua presença, uma vez que cada dispositivo simboliza um usuário, ou seja, ter uma tabela para armazenar um único registro não seria prudente. Já a segunda é um mecanismo necessário apenas no *web service*, pois através de seus atributos é possível realizar os tratamentos necessários, durante a sincronização, para garantir que os registros existentes no *web service* sejam os mesmos do aplicativo. Para elucidar a finalidade da tabela de registros deletados, a seguir é apresentado um pequeno exemplo.

Caso um registro da tabela de estados seja excluído do banco de dados do *web service*, um *insert* é realizado na tabela de registros deletados, contendo o valor do registro que foi removido. O Quadro 22 mostra como é feito este armazenamento.

| nome_tabela | id_tabela | data_exclusão |
|---|---|---|
| UF | 1 | 01/01/2017 |

**Quadro 22: Exemplo de conteúdo da tabela de registros deletados**

Durante a sincronização, com base no conteúdo desta tabela, o aplicativo móvel irá elaborar instruções SQL para que este registro também seja removido do seu banco de dados, impedindo que sejam realizadas avaliações ou simples visualizações de locais que o *software* não possui mais.

## 4.3 APRESENTAÇÃO DO SISTEMA

Inicialmente destaca-se que a utilização do sistema por parte do usuário será concentrada no aplicativo móvel, e este possui um layout composto basicamente por quatro telas. A primeira delas é responsável por exibir apenas um *splash* e foi implementada para deixar o aplicativo com uma inicialização mais amigável. Nela é apresentada uma imagem com a logo do *software*, em uma animação *fade in*. A Figura 8 ilustra essa tela.


**Figura 8: Tela de splash**

Quando a inicialização com o *splash* encerrar, a segunda tela do aplicativo é apresentada. Nela o usuário deve definir o IP e a porta onde o *web service* foi disponibilizado. Esta ação é de extrema importância, visto que algumas funcionalidades do aplicativo dependem da integração com o *web service*, como por exemplo, a geração de indicações de locais para visita. A Figura 9 apresenta a tela para inserção do IP e a porta do *web service*. Importante destacar que o conteúdo digitado nesta tela não é persistido no banco de dados, mas sim nas *shared preferences* do dispositivo.

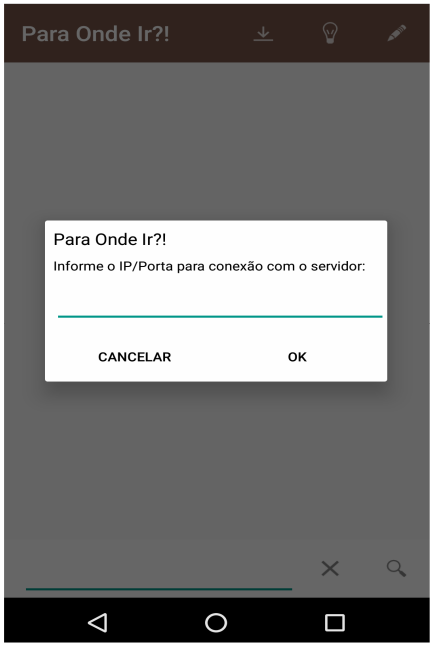


**Figura 9: Tela para inserir IP e porta do *web service***

Após definir o IP e a porta do *web service*, o aplicativo realiza a sincronização inicial dos registros, fazendo com que todos os dados salvos no banco de dados do *web service* sejam exportados para o aplicativo. Esse processo foi dividido em etapas, sendo que cada etapa realiza a sincronização de uma tabela do banco de dados. A Figura 10 apresenta o estado do aplicativo durante o processo de sincronização.

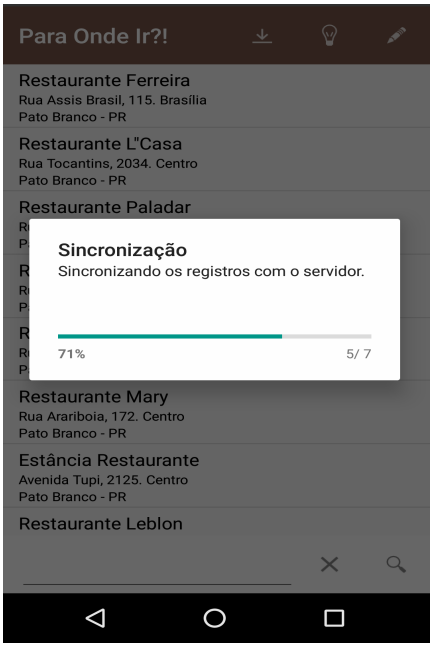


**Figura 10: Tela de sincronização**

É necessário citar que o fluxo de sincronização não é apenas do *web service* para o aplicativo, ou seja, o aplicativo também envia informações para o *web service*, como por exemplo, as avaliações realizadas pelos usuários.

Além das avaliações, uma segunda informação exportada é o *token* de registro, único por dispositivo, gerado pelo serviço Firebase. Este será associado ao usuário do dispositivo e ambos os valores sincronizados para a tabela de usuários do *web service*. A justificativa para que os dispositivos tenham identificadores únicos é devido à existência de um processo de sincronização automática de dados, tirando essa responsabilidade do usuário. Ao final da sincronização automática, o *software* se utiliza destes *tokens* para notificar os aparelhos que a sincronização foi concluída, conforme Figura 11.

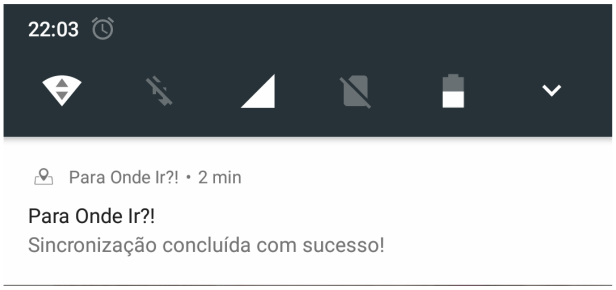


**Figura 11: Notificação da sincronização automática concluída**

Quando a sincronização inicial for encerrada, a terceira tela do aplicativo é exibida. Sua responsabilidade é apresentar os estabelecimentos disponíveis para avaliações e indicações, e foi dividida em três seções. A primeira seção é apresentada na parte superior, através de uma *toolbar*, onde o usuário pode sincronizar suas avaliações com o banco de dados do *web service* (apesar de o aplicativo oferecer um processo automático, existe a possibilidade de sincronizar manualmente), bem como solicitar indicações de locais para visita e redefinir o IP e a porta do *web service*. Todas estas ações serão realizadas através de botões, sendo que a ordem descrita acima é a mesma ordem dos botões.

A segunda seção apresenta uma *listview* que exibe os estabelecimentos disponíveis, sendo que essa será atualizada sempre que novas solicitações de indicações forem realizadas. Por fim, a terceira seção contém uma *toolbar* para pesquisa de estabelecimentos. A Figura 12 apresenta a tela descrita.

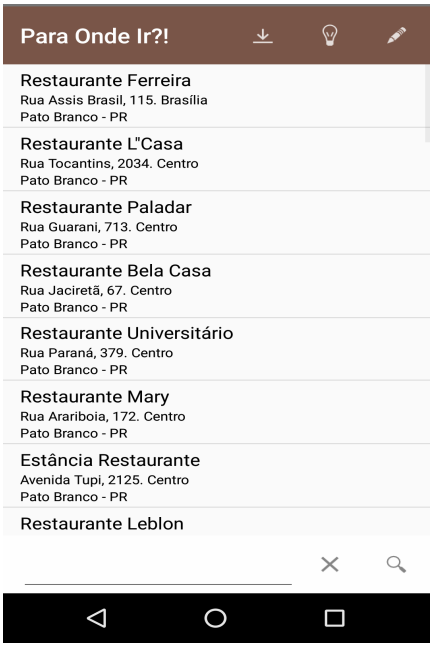


**Figura 12: Tela de listagem dos estabelecimentos**

Na *toolbar* superior, ao selecionar o primeiro ícone, da esquerda para a direita, o processo de sincronização é realizado. Tal funcionalidade já foi descrita neste capítulo anteriormente. O segundo botão, por sua vez, requisita ao *web service* a geração de indicações para visita, momento onde o algoritmo *A priori*, implementado em Java, é executado. Como esta ação é realizada em outro ambiente, o aplicativo deve estar conectado à *internet* neste momento. A Figura 13 demonstra o estado do aplicativo quando esta ação é requisitada.

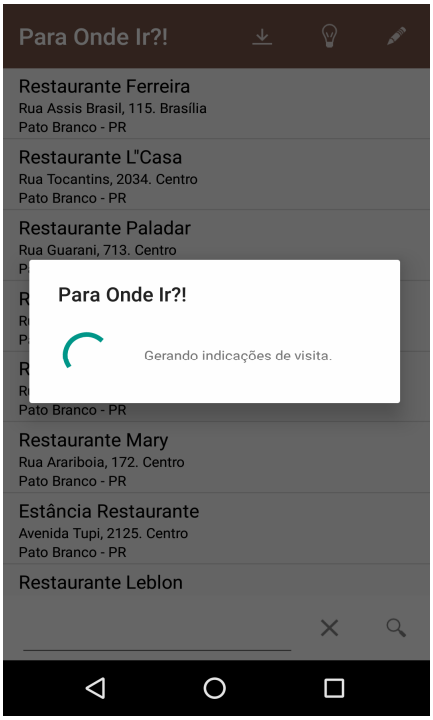


**Figura 13: Tela do aplicativo durante a geração de indicações**

Ao finalizar o processo de geração de indicações de visita, a *listview* da tela é atualizada com os estabelecimentos retornados pelo algoritmo, conforme a Figura 14.

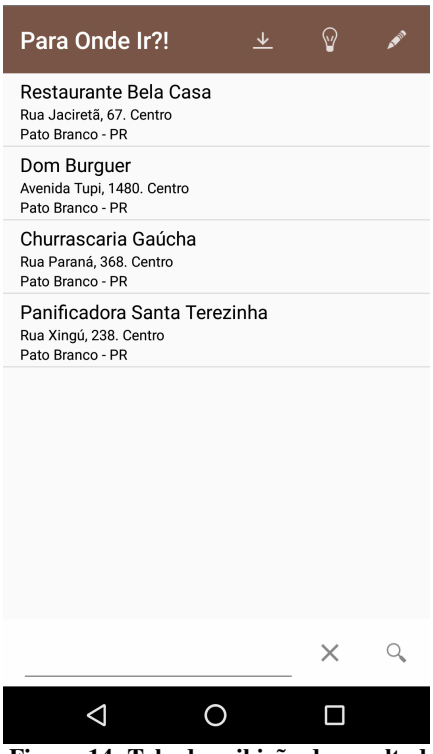


**Figura 14: Tela de exibição do resultado do algoritmo *A priori***

A quantidade de registros exibidos dependerá de alguns fatores, tais como o número de avaliações dos usuários e a forma como os estabelecimentos foram avaliados, positiva ou negativamente.

Outro ponto a ser descrito é o fato de que, durante a execução do algoritmo para geração de indicações, é realizado um "refinamento" no resultado. Esse processo tem como objetivo descartar os estabelecimentos que, por ventura, foram gerados como possíveis indicações, porém o usuário já os avaliou negativamente.

Uma outra funcionalidade presente na tela de listagem de estabelecimentos é a pesquisa por determinados locais. Ela é feita através da *toolbar* presente na seção inferior da tela, sendo que o usuário deve inserir o nome ou o endereço do local desejado. Após inserir, deve-se clicar no ícone representado por uma lupa e a *listview* será atualizada com o resultado da pesquisa. Caso o usuário queira cancelar a pesquisa, basta clicar no ícone representado por um "x". A Figura 15 ilustra esse processo.

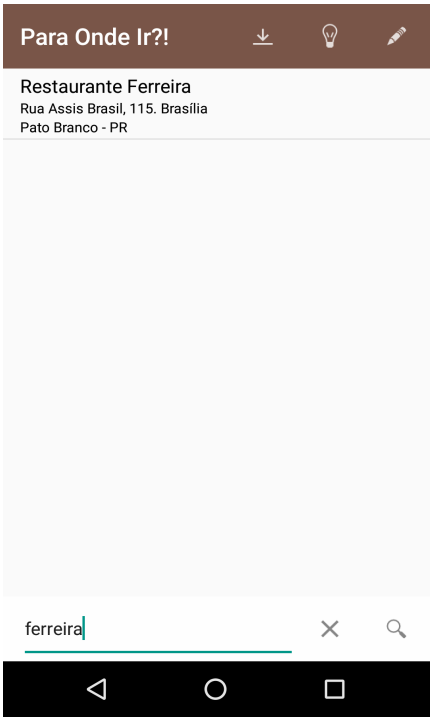


**Figura 15: Tela de pesquisa de estabelecimentos**

Para acessar a quarta e última tela do aplicativo, o usuário deve selecionar algum registro da listagem. Ao realizar esta ação, serão exibidos os dados detalhados do estabelecimento desejado, como nome, endereço, telefone e uma imagem para facilitar a identificação. Além disso, são apresentados dois botões para que o usuário realize a sua

avaliação: “Gostei” e “Não Gostei”. Ao realizar a avaliação, uma mensagem de confirmação é exibida ao usuário. Como um adicional, foi disponibilizado, na *toolbar* superior da tela, um botão para realizar ligações telefônicas para o estabelecimento. A Figura 16 demonstra o *layout* desta tela.

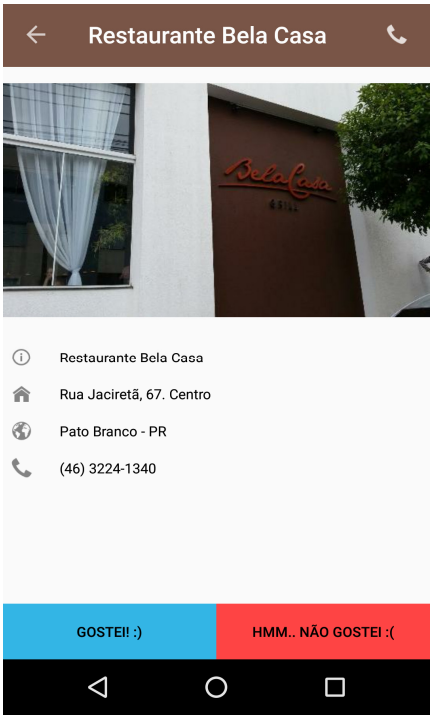


**Figura 16: Tela de detalhes do estabelecimento**

Como forma de auxílio ao usuário na execução dos processos que dependem da integração com o *web service*, foi adicionado no menu de contexto da *listview*, da tela principal, uma opção para que o usuário verifique se a conexão com a *internet* está funcionando. Para utilizar esta funcionalidade, basta selecionar o menu “Testar Conexão” e uma requisição de teste será disparada ao *web service*. O status da conexão será exibido como resposta. A Figura 17 demonstra esse processo.

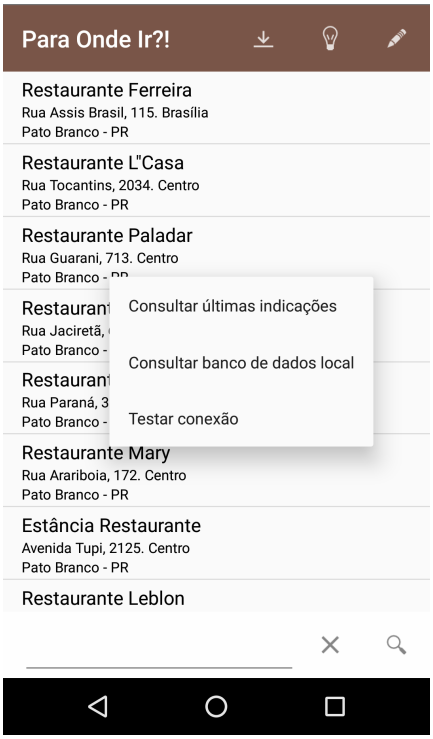


**Figura 17: Menu de contexto da listview**

Neste mesmo menu de contexto, existem ainda outras duas funcionalidades, intituladas de “Consultar últimas indicações” e “Consultar banco de dados local”. A primeira tem como objetivo listar para o usuário o resultado da última indicação gerada para ele. No entanto, estas informações não são persistidas em banco, ou seja, para que sejam exibidas é necessário que o aplicativo tenha se mantido em aberto após a execução da última geração de indicações. A segunda tem a finalidade de recarregar todos os estabelecimentos, caso o usuário tenha utilizado o menu citado anteriormente.

## 4.4 IMPLEMENTAÇÃO DO SISTEMA

Devido a quantidade de linhas de código produzidas, serão exibidos apenas alguns trechos, que evidenciam as partes principais do sistema elaborado. No desenvolvimento do aplicativo *mobile*, foram codificadas cerca de 50 classes, totalizando por volta de 2700 linhas de código. Já no desenvolvimento do *web service*, cerca de 1500 linhas de código foram divididas em 28 classes.

Para realizar o processo de persistência no banco de dados do dispositivo, foi utilizada a biblioteca “ORMLite”, sendo que a sua principal característica é o mapeamento objeto-

relacional das classes. Para poder utilizá-la, além de importar alguns JARs ao projeto, é necessário definir as classes de modelo, isto é, as representações das tabelas do banco, além das classes DAO, responsáveis pela interação com o banco em si. A Listagem 1 demonstra a configuração da classe de modelo “Avaliacao”.

```
1 @DatabaseTable(tableName = IConstantesDatabase.TABELA_AVALIACAO)
2 public class Avaliacao implements Serializable {
3
4   @DatabaseField(columnName = IConstantesDatabase.AVALIACAO_ID,
5       id = true, canBeNull = false, dataType = DataType.INTEGER)
6   private int idAvaliacao;
7
8   @DatabaseField(canBeNull = false,
9       columnName = IConstantesDatabase.AVALIACAO_ESTABELECIMENTO,
10      foreign = true, foreignAutoRefresh = true,
11      foreignColumnName = IConstantesDatabase.ESTABELECIMENTO_ID)
12  private Estabelecimento estabelecimento;
13
14  @DatabaseField(canBeNull = false,
15      columnName = IConstantesDatabase.AVALIACAO_LIKE,
16      dataType = DataType.ENUM_STRING)
17  private YesNo gostou;
18
19  @DatabaseField(canBeNull = false,
20      columnName = IConstantesDatabase.AVALIACAO_USER,
21      dataType = DataType.STRING)
22  private String usuario;
23
24  @DatabaseField(columnName = IConstantesDatabase.AVALIACAO_DTAVALIACAO,
25      dataType = DataType.STRING)
26  private String data;
```

**Listagem 1: Mapeamento da classe Avaliacao**

Pode-se perceber que a configuração da classe é feita utilizando *annotations*. Na linha 1 indicamos a tabela associada a classe através da *annotation* “DatabaseTable” e do atributo “tableName”. Neste caso, a classe “Avaliacao” representa a tabela do banco de dados de mesmo nome.

Na linha 2 foi definido o nome da classe, assim como a implementação da interface “Serializable”. Isto se faz necessário para que essa classe possa ser exportada ao *web service* nas sincronizações. Nas linhas seguintes foram mapeados os atributos da tabela através da *annotation* “DatabaseField”. Na linha 4, por exemplo, verifica-se o mapeamento do atributo “idAvaliacao” com algumas características, tais como o nome da coluna no banco de dados (atributo “columnName”), o fato de ser chave primária (atributo “id”), a indicação de que o campo deve possuir algum valor (atributo “canBeNull”) e o tipo de dado (atributo “dataType”). Ainda em tempo, destaca-se que os nomes da tabela e das colunas do banco foram centralizados na interface “IConstantesDatabase”.

A Listagem 2 exibe o DAO da classe “Avaliacao”.

```
1 public class AvaliacaoDAO extends GenericDAO<Avaliacao> {
2
3    public AvaliacaoDAO(Context ctx) throws SQLException {
4        super(ctx, Avaliacao.class);
5    }
6
7    @Override
8    public boolean save(Avaliacao avaliacao) throws SQLException {
9        // Verifica se o usuário e estabelecimento obtidos no objeto de
10          parâmetro já existem no banco de dados.
11       Avaliacao avaliacaoTemp =
12              findByIdEstabelecimentoAndUsuario(avaliacao);
13
14       if (avaliacaoTemp == null){
15            // Se ainda não existir, insere.
16            avaliacao.setIdAvaliacao(
17                      getProximoId(avaliacao.getUsuario()) );
18            return super.insert(avaliacao);
19       } else {
20            // Se já existir, atualiza.
21            avaliacao.setIdAvaliacao(avaliacaoTemp.getIdAvaliacao());
22            return super.update(avaliacao);
23       }
24   }
25
26   private int getProximoId(String usuario) throws SQLException {
27        try {
28            QueryBuilder<Avaliacao, Integer> builder =
29                    getDao().queryBuilder();
30
31       /* Para gerar algo como:
32          SELECT MAX(IDAVALIACAO) FROM AVALIACAO
33             WHERE USUARIO = :usuario"*/
34            builder.selectRaw("MAX(" + IConstantesDatabase.AVALIACAO_ID
35 + ")").where().eq(IConstantesDatabase.AVALIACAO_USER, usuario);
36
37            // Executa a consulta
38            String[] resultado = getDao().queryRaw(
39                  builder.prepareStatementString()).getFirstResult();
40
41            if (resultado[0] != null ){
42                // Se encontrou o resultado, pega o maior + 1
43                return Integer.parseInt(resultado[0]) + 1;
44            } else {
45                // Se não, retorna 1
46                return 1;
47            }
48        } catch (Exception ex){
49            throw new SQLException("Erro: " + ex.getMessage());
50        }
51   }
52
53   private Avaliacao findByIdEstabelecimentoAndUsuario(Avaliacao obj)
54           throws SQLException {
55        try {
56            QueryBuilder<Avaliacao, Integer> builder = getDao().
57              queryBuilder();
58
59              /* Para gerar algo como:
```

```
60                  SELECT * FROM AVALIACAO
61                   WHERE USUARIO = :usuario
62                   AND IDESTABELECIMENTO = :estabelecimento */
63              builder.where()
64                .eq(IConstantesDatabase.AVALIACAO_USER, obj.getUsuario())
65                .and()
66                .eq(IConstantesDatabase.AVALIACAO_ESTABELECIMENTO,
67                    obj.getEstabelecimento().getIdEstabelecimento());
68
69              // Executa a consulta
70              List<Avaliacao> list = builder.query();
71              if (list.size() > 0) {
72                  return builder.query().get(0);
73              } else {
74                  return null;
75              }
76          } catch (Exception ex){
77              throw new SQLException("Erro: " + ex.getMessage());
78          }
79      }
80  }
```

**Listagem 2: DAO da classe "Avaliacao"**

Na linha 1 nota-se a declaração da classe, bem como uma relação de herança com a classe “GenericDAO”. Esta arquitetura de herança foi adotada visto que as interações com o banco de dados se darão através de ações genéricas, tais como salvar, atualizar, consultar e excluir. Portanto, os métodos responsáveis por estas ações ficam declarados e implementados na classe “GenericDAO” e as classes que herdam dessa os incorporam, sendo que, caso haja a necessidade, os métodos podem ser reescritos.

Na linha 8 temos a implementação do método para salvar as avaliações (“save()”). Como citado no parágrafo anterior, o método de salvamento foi definido como uma ação genérica. Porém no caso das avaliações foi optado por salvar apenas um registro por usuário e estabelecimento, ou seja, caso um usuário avalie um determinado estabelecimento mais de uma vez, apenas a última avaliação é mantida. Devido a essa necessidade de reescrever o método, tem-se a presença da *annotation* “Override”.

O método “save()” recebe como parâmetro um objeto do tipo “Avaliacao” e consulta no banco de dados do dispositivo, através da função “findByIdEstabelecimentoAndUsuario”, se já existe uma avaliação salva para o usuário e estabelecimento presentes no objeto obtido por parâmetro, conforme as linhas 11 e 12. A implementação do método “findByIdEstabelecimentoAndUsuario”, está descrita a partir da linha 53, onde é possível notar a presença da classe “QueryBuilder“ e seus métodos. Basicamente, esta classe serve para indicar instruções SQL.

Caso a função “findByIdEstabelecimentoAndUsuario” retorne um valor nulo, na linha 17 é consultado o próximo código identificador válido, através do método “getProximoID()” e

atribuído ao objeto que será persistido. Por sua vez, na linha 18 é invocado o método de inserção, declarado e implementado na classe “GenericDAO”.

Porém, caso a consulta indique que já existe uma avaliação para o usuário e estabelecimento obtidos do objeto de parâmetro, o valor do código identificador resultante da consulta é atribuído ao objeto que será persistido, e o método de atualização, presente na classe “GenericDAO” é chamado, conforme as linhas 21 e 22.

A Listagem 3, por fim, demonstra como o método de salvamento das avaliações é invocado.

```
1 private void avaliar(YesNo like) throws SQLException {
2    try {
3        // Cria objeto para persistir a avaliação no banco de dados
4        Avaliacao avaliacao = new Avaliacao();
5        // Valoriza os atributos
6        avaliacao.setEstabelecimento(estabelecimento);
7        avaliacao.setGostou(like);
8        avaliacao.setUsuario(
9            AppParaOndeIr.getInstance().getUser().getUsuario());
10
11       SimpleDateFormat formato =
12       new SimpleDateFormat("dd/MM/yyyy HH:mm:ss", Locale.getDefault());
13
14       avaliacao.setData(formato.format(new Date()));
15
16       // Cria objeto DAO para manipulação do banco
17       AvaliacaoDAO dao = new AvaliacaoDAO(this);
18       // Salva
19       dao.save(avaliacao);
20       // Mensagem de confirmação
21       MensagemUtils.gerarEExibirToast(this,
22           getString(R.string.msg_sucesso_avaliacao));
23
24       dao.close();
25       finish();
26    } catch (SQLException ex){
27        throw new SQLException(ex.getMessage());
28    }
29 }
```

**Listagem 3: Método de salvamento das avaliações**

Inicialmente vale destacar que esse método está presente na *activity* que ilustra os detalhes do estabelecimento. Na linha 4, temos a declaração e criação do objeto que representa as avaliações. Na sequência, os atributos do objeto são valorizados, tais como usuário, o estabelecimento que está sendo visualizado, o indicativo de que o usuário gostou ou não do local e a data da avaliação. Neste ponto é importante destacar o uso da classe “AppParaOndeIr” na linha 9. Tal classe possui uma relação de herança com a classe “Application” e foi desenvolvida para manter em memória algumas informações úteis ao aplicativo, como por exemplo, o usuário. Para poder manipulá-la, basta invocar o método

estático “getInstance()”, visto que para esta classe foi adotado o padrão de projeto “*Singleton”*.

Na linha 17, a classe DAO das avaliações é criada e, logo na sequência, o método de salvamento é invocado. Por fim, uma mensagem indicativa é exibida ao usuário, alertando sobre o sucesso ou falha no salvamento, e o DAO e a *activity* são encerrados.

Após as avaliações serem salvas e sincronizadas ao *web service*, a ação esperada pelo aplicativo é a solicitação de indicações de locais para visita, e quando esta é requisitada, um método no *web service* é disparado. A implementação deste método é apresentada na Listagem 4. Para facilitar a compreensão ao longo do trabalho, esse método será chamado de “método principal”.

```
1 @RequestMapping(value = "/", method = RequestMethod.POST,
2     produces = "application/json")
3 public String solicitaIndicacao(@RequestBody String json)
4           throws JSONException {
5
7     // Deserealiza o json do param em um objeto Usuario.
8     Usuario usuario = new Gson().fromJson(json,
9               new TypeToken<Usuario>() {}.getType());
10
11    List<Estabelecimento> retorno = new ArrayList<>();
12    // Consulta os usuários que já realizaram avaliações.
13    List<String> usuarios = avaliacaoDao.findUsuariosAvaliacao();
14    // Consulta as avaliações de cada usuário.
15    List<HashMap<String, Integer[]>> avaliacoes =
16          getAvaliacoes(usuarios, false);
17    // Consulta as avaliações positivas de cada usuário.
18    List<HashMap<String, Integer[]>> avaliacoesPositivas =
19          getAvaliacoes(usuarios, true);
20
21    // Aplica o cálculo do suporte até que existam itemsets.
22    List<Integer[]> itemsets = criaItemsetInicial();
23    do {
24          List<Integer[]> auxiliar = calculaSuporte(itemsets, usuarios,
25                avaliacoes, avaliacoesPositivas);
26          if (auxiliar.size() > 1){
27                itemsets.clear();
28                itemsets = atualizaItemset(auxiliar);
29          } else {
30                itemsets = auxiliar;
31          }
32    } while (itemsets.size() != 1);
33
35    /* Verifica se o usuário que está pedindo sugestões já fez alguma
36       avaliação. */
37    boolean calculaConfianca =
38           usuarioAvaliou(usuario.getUsuario(), usuarios);
39
40    if (calculaConfianca) {
42          // Divide o itemset resultante do suporte em conj menores.
43          List<Integer[]> itemsetsConfianca =
44              geraItemsetsConfianca(itemsets.get(0));
45
46          List<RegraAssociacao> regras = new ArrayList<>();
```

```
47          for (int i = 0; i < itemsetsConfianca.size(); i++) {
48                Integer[] itemset = itemsetsConfianca.get(i);
49                // Gera as regras de associação para cada conjunto
50                List<RegraAssociacao> regrasItemset =
51                    geraRegrasDeAssociacao(itemset);
52                for (RegraAssociacao regra : regrasItemset) {
53                      regra.dadosHashToList();
54                      // Calcula a confiança da regra.
55                      double confianca = calculaConfianca(regra,
56                                usuarios, avaliacoesPositivas);
57
58                      if (confianca >= Constantes.CONFIANCA_MINIMA) {
59                            regras.add(regra);
60                      }
61                }
62          }
63          /* Percorre as regras para eliminar os estabelecimentos que o
64             usuário já avaliou negativamente. */
65          for (int i = 0; i < regras.size(); i++) {
66                RegraAssociacao r = regras.get(i);
67                if (usuarioGostouDaCondicaoSe(usuario.getUsuario(),
68                            r.getListSe(), avaliacoesPositivas)) {
69                      List<Integer> entao = r.getListEntao();
70                      for (int j = 0; j < entao.size(); j++) {
71                            Estabelecimento estab =
72                                  estabDao.findOne(entao.get(j));
73                            addElemento(estab, retorno);
74                      }
75                }
76          }
77    } else {
78          /* Caso o usuário não tenha feito avaliações, retorna os
79             estabelecimentos que passaram pelo suporte. */
80          for (Integer[] itemset : itemsets) {
81                for (int i = 0; i < itemset.length; i++) {
82                      addElemento(estabDao.findOne(itemset[i]), retorno);
83                }
84          }
85    }
86    // Retorna o resultado no formato de um json
87    Gson gson = new Gson();
88    JsonElement element = gson.toJsonTree(retorno,
89          new TypeToken<List<Estabelecimento>>() {}.getType());
90    return element.getAsJsonArray().toString();
91 }
```

**Listagem 4: Método de geração de indicações de locais para visita**

O desenvolvimento do *web service* do aplicativo foi feito utilizando o *framework* Spring Boot, e na linha 1 já é possível visualizar a aplicação de um de seus conceitos, através da *annotation RequestMapping*. Por meio desta *annotation* é possível especificar o contexto no qual o método será exposto, isto é, a URL para invocar a função associada (atributo *value*), além de indicar características extras em relação ao seu acesso, como por exemplo, o método HTTP que deve ser utilizado (atributo *method*) e o conteúdo a ser produzido no retorno do método (atributo *produces*).

Na linha 3 do método principal, tem-se a declaração do mesmo, sendo que esse recebe como parâmetro um objeto do tipo texto, “anotado” com *RequestBody*. Tal *annotation* denota que o objeto deve ser vinculado ao corpo da requisição.

No método de geração de indicações, o objeto enviado no corpo da requisição possui como informação o usuário que está solicitando sugestões de locais para visita. Portanto, na linha 6 é feita a conversão desse valor para um objeto do tipo “Usuario”, facilitando assim a sua manipulação no restante do código através dos conceitos da orientação a objetos. Destaca-se que a conversão dos valores foi feita utilizando o método “fromJson()”, da biblioteca “GSON”.

Nas linhas 13, 15 e 18 do método principal são realizados processamentos a fim de valorizar os objetos que serão analisados na execução do algoritmo *A priori*. O primeiro deles se trata de uma consulta e é realizada através do método “findUsuariosAvaliacao()”. Esse método foi declarado na classe DAO das avaliações e tem como objetivo retornar uma lista com os usuários que já realizaram alguma avaliação. A implementação deste método é apresentada na Listagem 5.

```
1 @Query(value = "SELECT DISTINCT USUARIO "
2               + "  FROM AVALIACAO "
3               + " ORDER BY USUARIO", nativeQuery = true)
4 List<String> findUsuariosAvaliacao();
```

**Listagem 5: Método “findUsuariosAvaliacao()”**

Através da *annotation Query*, definimos a expressão SQL que o método executará. De forma geral, optou-se por trabalhar com instruções SQL puras para poder codificar instruções mais elaboradas e, por conta disso, devemos valorizar o campo *value* com a expressão SQL desejada e o campo *nativeQuery* com *true*.

Já o método “getAvaliacoes()”, utilizado no método principal após a execução da consulta anterior, retorna as avaliações efetuadas por cada usuário, no formato “chave x valor”. A Listagem 6 demonstra a implementação deste método.

```
1 private List<HashMap<String, Integer[]>> getAvaliacoes(
2     List<String> usuarios, boolean filtraParamGostou) {
3
4     List<HashMap<String, Integer[]>> retorno = new ArrayList<>();
5     for (String usuario : usuarios) {
6           /* IF ternário para consultar os estabelecimentos avaliados
7              pelo usuário. Ou consulta todos ou consulta apenas os que
8              foram avaliados positivamente */
9           List<Integer> estabelecimentos = filtraParamGostou ?
10                avaliacaoDao.findEstabsByUsuarioAndGostou(usuario, "S") :
11                avaliacaoDao.findEstabsByUsuario(usuario);
12
13          HashMap<String, Integer[]> hash =
14                new HashMap<String, Integer[]>();
```

```
15
16          // Atribui o resultado da consulta em um objeto hashMap.
17          hash.put(usuario, estabelecimentos.toArray(
18                  new Integer[estabelecimentos.size()]));
19
20          retorno.add(hash);
21      }
22      return retorno;
23  }
```

**Listagem 6: Método "getAvaliacoes()"**

Este método recebe como parâmetro a lista dos usuários que já realizaram avaliações e, em sua execução, itera sobre essa lista consultando no banco de dados as avaliações realizadas por esses. Aqui vale destacar que duas consultas podem ser executadas dependendo do valor recebido no parâmetro "filtraParamGostou". Caso esse parâmetro seja *true*, são consultadas apenas as avaliações em que o usuário indicou que gostou dos estabelecimentos. Do contrário, todas as avaliações do usuário são retornadas.

A partir disso, os códigos dos estabelecimentos retornados são transformados em um *array* de números inteiros, e por fim, tanto o usuário quanto suas avaliações são atribuídas a um objeto "HashMap", estrutura que armazena informações no formato "chave x valor".

Foi optado por trabalhar com essa estrutura para facilitar a definição das transações e dos *itemsets*, conceitos importantíssimos para a execução do algoritmo *A Priori*. O Quadro 22 demonstra a estrutura que se visa obter.

| **Usuário** | **Códigos dos estabelecimentos avaliados (*itemsets*)** |
|---|---|
| teste1@gmail.com | [1, 2, 3, 4, 5] |
| teste2@gmail.com | [1, 3, 7, 8, 9] |

**Quadro 23: Estrutura do resultado do método "getAvaliacoes()"**

Na linha 18 do método principal, é apresentado o início da execução do *A priori*, com a criação dos *itemsets* iniciais, de tamanho um. O desenvolvimento deste método é apresentado na Listagem 7.

```
1 private List<Integer[]> criaItemsetInicial() {
2     List<Integer[]> retorno = new ArrayList<>();
3     // Consulta todos os estabelecimentos e os percorre.
4     for (Estabelecimento estab : estabDao.findAll()) {
5           // Adiciona o código do estab em um array.
6           Integer[] itemset = { estab.getIdEstabelecimento() };
7           retorno.add(itemset);
8     }
9     return retorno;
10 }
```

**Listagem 7: Método "criaItemsetInicial()"**

Para criar os *itemsets* iniciais, é efetuada uma consulta, no banco de dados, de todos os estabelecimentos e, na sequência, realizados *loops* para cada objeto retornado. A cada iteração, um *itemset* é criado e adicionado na lista de retorno.

Após definir os *itemsets*, são realizados *loops* sobre os conjuntos gerados, aplicando o cálculo do suporte para esses. O método que realiza esse cálculo é mostrado na Listagem 8.

```
1 private List<Integer[]> calculaSuporte (List<Integer[]> itemsets,
2     List<String> usuarios,
3     List<HashMap<String, Integer[]>> avaliacoes,
4     List<HashMap<String, Integer[]>> avaliacoesPositivas) {
5
6     List<Integer[]> retorno = new ArrayList<>();
7
8     // Percorre os itemsets obtidos por parâmetro.
9     for (int i = 0; i < itemsets.size(); i++) {
10          Integer[] itemset = itemsets.get(i);
11
12          // Conta o número total de avaliações do itemset do loop.
13          double total = contarAvaliacoes(itemset, usuarios, avaliacoes);
14
15          // Conta o número de avaliações positivas do itemset do loop
16          double totalSim = contarAvaliacoes(itemset, usuarios,
17                      avaliacoesPositivas);
18
19          // calcula suporte.
20          double resultado = totalSim / total;
21
22          // se resultar maior que o mínimo, add na lista de retorno.
23          if (resultado >= Constantes.SUPORTE_MINIMO) {
24                addElemento(itemset, retorno);
25          }
26    }
27    return retorno;
28 }
```

**Listagem 8: Método "calculaSuporte()"**

Para utilizar esse método, deve-se enviar como parâmetro uma lista contendo todos os *itemsets* definidos no formato de um *array*, os usuários que já realizaram avaliações, todas as avaliações realizadas por esses usuários e as avaliações positivas desses.

Primeiramente, são efetuadas iterações sobre os *itemsets* a fim de contar o seu número total de avaliações, bem como a quantidade de avaliações positivas. Quem realiza essa contagem é o método “contarAvaliacoes()”. Como a fórmula do suporte é definida por uma fração – conforme Figura 2 – obtemos os valores citados anteriormente e os aplicamos adequadamente. Caso o resultado seja maior que o suporte mínimo proposto (0.3), o *itemset* do *loop* é adicionado à lista de retorno. Por fim, apenas os *itemsets* cujo valor de suporte sejam maiores que o valor mínimo são retornados.

A cada ciclo de execução, os *itemsets* que passaram pelo cálculo do suporte são atualizados, objetivando gerar novos conjuntos para aplicar o cálculo do suporte novamente. O método responsável por esse processamento é o "atualizaItemset()", e esse é apresentado na Listagem 9.

```
1 private List<Integer[]> atualizaItemset(List<Integer[]> itemsets) {
2     Integer tamanhoAtual = itemsets.get(0).length;
3     List<Integer[]> novaLista = new ArrayList<>();
4
5     for (int i = 0; i < itemsets.size(); i++) {
6           for (int j = i + 1; j < itemsets.size(); j++) {
7                 // Pega-se 2 itemsets subsequentes para mesclá-los
8                 Integer[] itemsetLoopI = itemsets.get(i);
9                 Integer[] itemsetLoopJ = itemsets.get(j);
10                // Seta o tamanho do novo itemset. Sempre incrementa 1
11                Integer[] novoItemset = new Integer[tamanhoAtual + 1];
12
13                for (int k = 0; k < novoItemset.length - 1; k++) {
14                      // Atribui valores iniciais ao novo itemset.
15                      novoItemset[k] = itemsetLoopI[k];
16                }
17
18                int diferente = 0;
19                /* Verifica se os itemsets analisados possuem elementos
20                  em comum */
21                for (int l = 0; l < itemsetLoopJ.length; l++) {
22                      boolean encontrou = false;
23                      for (int m = 0; m < itemsetLoopI.length; m++) {
24                            if (itemsetLoopI[m] == itemsetLoopJ[l]) {
25                                  encontrou = true;
26                                  break;
27                            }
28                      }
29                     /* Caso verifique que um determinado elemento não
30                        seja comum aos dois itemsets, adiciona no array
31                          resultante. */
32                      if (!encontrou) {
33                            diferente++;
34                            novoItemset[novoItemset.length - 1] =
35                                 itemsetLoopJ[l];
36                      }
37                }
38
39                Arrays.sort(novoItemset);
40                if (diferente > 0) {
41                      addElemento(novoItemset, novaLista);
42                }
43          }
44    }
45
46    return novaLista;
47 }
```

**Listagem 9: Método "atualizaItemset()"**

Ao invocar esse método, deve ser enviada uma lista com os *itemsets* atuais do algoritmo. Sendo *n* o número de elementos dos *itemsets* da lista, o método realiza *loops* sobre

esses e sempre mescla dois deles para produzir novos com *n+1* elementos. Isto é, ao combinar os *itemsets* [1, 2] e [3, 4], são gerados novos de tamanho 3 com os elementos dos *itemsets* que foram combinados, produzindo, por exemplo, [1, 2, 3], [1, 2, 4] e etc. No final da execução é retornada uma lista com os novos *itemsets* gerados.

Quando não for mais possível aplicar o cálculo do suporte, ou seja, quando os *itemsets* não puderem mais ser atualizados e incrementados, esta etapa se encerra e o *itemset* resultante é mantido em memória para posteriores manipulações.

Na linha 31 do método principal, tem-se a execução da função "usuarioAvaliou()", que verifica se o usuário que requisitou sugestões de locais para visita já realizou alguma avaliação no aplicativo. A implementação desse método é ilustrada na Listagem 10.

```
1 private boolean usuarioAvaliou(String usuario, List<String> usuarios) {
2     boolean retorno = false;
3     for (int i = 0; i < usuarios.size(); i++) {
4           if (usuarios.get(i).equalsIgnoreCase(usuario)) {
5                 retorno = true;
6                 break;
7           }
8     }
9     return retorno;
10 }
```

**Listagem 10: Método "usuarioAvaliou()"**

Esse método apenas verifica se o usuário enviado por parâmetro está na lista de usuários que realizaram avaliações, sendo que essa também é obtida por parâmetro. Caso seja indicado que o usuário já realizou alguma avaliação, a segunda etapa do algoritmo se inicia: a geração das regras de associação. Do contrário, os estabelecimentos do *itemset* obtido na execução do cálculo do suporte são retornados ao aplicativo.

Ao iniciar a etapa da geração das regras de associação, o primeiro processamento realizado é o de separar os *itemsets* resultantes do cálculo do suporte em conjuntos menores. Quem realiza esse procedimento é o método "geraItemsetsConfianca()", e esse é exibido na Listagem 11.

```
1 private List<Integer[]> geraItemsetsConfianca (Integer[] itemset) {
2     List<Integer[]> retorno = new ArrayList<Integer[]>();
3     int controle = itemset.length;
4     int[] arrayBit = new int[controle];
5     // número limite de itemsets a serem gerados.
6     int qtdeItemsets = (int) (Math.pow(2, controle) - 1);
7     //inicializa o array de bits para utilizar nas validações posteriores
8     for (int i = 0; i < controle; i++) {
9           arrayBit[i] = 0;
10    }
11
12    for (int j = 1; j <= qtdeItemsets; j++) {
13          List<Integer> listaIds = new ArrayList<>();
14          /* invoca a função auxiliar para saber quais elementos do
```

```
15              itemset do parâmetro vao ser utilizados na geração dos
16              novos conjuntos */
17          atualizaBit(arrayBit, controle);
18          for (int k = 0; k < controle; k++) {
19                /* caso um determinado elemento do array auxiliar tenha
20                   o valor 1, pega-se o elemento de mesmo índice do
21                   itemset do parâmetro */
22                if (arrayBit[k] == 1) {
23                      listaIds.add(itemset[k]);
24                }
25          }
26          /* Só considera um itemset válido caso tenha mais de 1
27             elemento. */
28          if (listaIds.size() > 1) {
29                Integer[] ids = listaIds.toArray(
30                    new Integer[listaIds.size()]);
31                addElemento(ids, retorno);
32          }
33    }
34
35    return retorno;
36 }
```

**Listagem 11: Método "geraItemsetsConfianca()"**

Sendo *n* o número de elementos do *itemset* obtido por parâmetro, esse método produzirá todos os conjuntos possíveis com seus elementos, desde que o tamanho mínimo destes seja 2 e o máximo *n*. Para isso, foram necessárias algumas definições, tais como o número máximo de *itemsets* a serem gerados e um *array* auxiliar que possui *flags* indicando quais elementos do *itemset* devem ser utilizados na geração dos novos conjuntos.

Esse *array* auxiliar armazena apenas valores binários – 0 ou 1 – sendo que todos os seus elementos são inicializados em 0. A cada *loop* é invocado o método "atualizaBit()", que em seu processamento altera o valor dos elementos de 0 para 1, e logo na sequência interrompe a iteração. Por exemplo: ao enviar a esse método o *array* [0, 0, 0], ele se tornará [1, 0, 0]; o array [0, 0, 1], por sua vez, será transformado em [1, 0, 1], e assim por diante. O código deste método é apresentado na Listagem 12.

```
1 private void atualizaBit(int[] arrayBits, int tamanho) {
2     /* caso o array de bits tenha elemento com valor 0, troca para 1. Se
3        não, seta 0. */
4     for (int i = 0; i < tamanho; i++) {
5           if (arrayBits[i] == 0) {
6                 arrayBits[i] = 1;
7                 break;
8           }
9           arrayBits[i] = 0;
10    }
11 }
```

**Listagem 12: Método "atualizaBit()"**

Após invocar o “atualizaBit()”, o método “geraItemsetsConfianca()” utiliza o conteúdo do *array* auxiliar para determinar quais elementos do *itemset* serão usados no novo conjunto, isto é, caso um elemento do *array* auxiliar possua o valor 1, é obtido o elemento de mesmo índice do *itemset* obtido por parâmetro. Exemplificando, caso o *itemset* possua os elementos [3, 4, 5] e o *array* auxiliar tenha o conteúdo [1, 1, 0], serão obtidos os elementos dos índices 0 e 1 do *itemset* para gerar o novo conjunto, ou seja, o *itemset* gerado será [3, 4]. Esse ciclo se repete até que todos os conjuntos possíveis sejam produzidos.

Com a lista resultante do método “geraItemsetsConfianca()”, percorre-se cada elemento, a fim de gerar as regras de associação para os conjuntos separadamente, sendo que essa geração ocorre nas implementações dos métodos “geraRegrasDeAssociacao()”. Tal método foi reimplementado três vezes, utilizando-se do conceito de sobrecarga, e os seus códigos são apresentados nas listagens 13, 14 e 15.

```
1 private List<RegraAssociacao> geraRegrasDeAssociacao(Integer[] itemset)
2 {
3     if (itemset.length >= 2) {
4           List<RegraAssociacao> regras = new ArrayList<>();
5           Set<RegraAssociacao> setResultado = new HashSet<>();
6           Set<Integer> set = new HashSet<>(Arrays.asList(itemset));
7           Set<RegraAssociacao> setRegras =
8                 geraRegrasDeAssociacao(set);
9           geraRegrasDeAssociacao(set, setRegras, setResultado);
10
11          for (RegraAssociacao regraAssociacao : setResultado) {
12                regras.add(regraAssociacao);
13          }
14          return regras;
15    }
16    return null;
17 }
```

**Listagem 13: Primeira sobrecarga do método "geraRegrasDeAssociacao()"**

```
1 private Set<RegraAssociacao> geraRegrasDeAssociacao(
2        Set<Integer>itemset) {
3     Set<RegraAssociacao> retorno = new HashSet<>(itemset.size());
4     /* Atribui ao Set os elementos do itemset de parâmetro.
5        Ex: itemset [2,5] > Set se = [2,5]. */
6     Set<Integer> se = new HashSet<>(itemset);
7     Set<Integer> entao = new HashSet<>(1);
8     /* Vai percorrendo os elementos do itemset e os trocando de Set,
9        passando do "Se" para o "Então".
10       Ex: Set "Se" = [2,5]
11       Set "Então" = []
12       Depois do loop...
13       Set "Se" = [2]
14       Set "Então" = [5] */
15    for (Integer elemento : itemset) {
16          se.remove(elemento);
17          entao.add(elemento);
18          retorno.add(new RegraAssociacao(se, entao));
19          se.add(elemento);
```

```
20          entao.remove(elemento);
21      }
22      return retorno;
23  }
```

**Listagem 14: Segunda sobrecarga do método "geraRegrasDeAssociacao()"**

```
1  private void geraRegrasDeAssociacao(Set<Integer> itemset,
2            Set<RegraAssociacao> setRegras,
3            Set<RegraAssociacao> setResultado) {
4
5      Iterator<RegraAssociacao> it = setRegras.iterator();
6      /* Inicializa o setResultado */
7      while (it.hasNext()) {
8            RegraAssociacao regra = it.next();
9            setResultado.add(regra);
10     }
11
12     int k = itemset.size();
13     int m = setRegras.iterator().next().getHashSetEntao().size();
14     if (k > m + 1) {
15           /* Alterna os elementos da regra.
16               Ex: Regra "SE x, y ENTÃO z"
17               Transforma em ...
18               Regra "SE x ENTÃO y, z" */
19           Set<RegraAssociacao> novasRegras =
20                     moveUmElementoPraCondicaoEntao(setRegras);
21           Iterator<RegraAssociacao> iterator = novasRegras.iterator();
22           /* Adiciona novamente no setResultado. */
23             while (iterator.hasNext()) {
24                 RegraAssociacao regra = iterator.next();
25                 setResultado.add(regra);
26           }
27           geraRegrasDeAssociacao(itemset, novasRegras, setResultado);
28     }
29 }
```

**Listagem 15: Terceira sobrecarga do método "geraRegrasDeAssociacao()"**

O objetivo desses métodos é elaborar regras do tipo "Se X então Y" para os *itemsets*. Para facilitar o desenvolvimento, foi criada uma classe denominada de "RegraAssociacao" para representar a estrutura das regras. Essa classe possui atributos do tipo "Set", um indicando o parâmetro "Se" da regra e o outro representando o parâmetro "Então".

A partir dessa definição, os métodos de geração das regras realizam loops sobre os elementos do itemset e vão os adicionando em objetos do tipo "RegraAssociacao", de modo que um elemento não esteja nas duas condições da regra. Por exemplo, se uma regra possui no parâmetro "Se" o estabelecimento de código 1, obrigatoriamente o parâmetro "Então" não poderá conter esse mesmo estabelecimento.

Com o resultado obtido a partir desse método, é aplicado o cálculo da confiança para cada regra, verificando sua "validade", ou seja, se essa possui o valor de confiança maior que o mínimo proposto. O código deste método é ilustrado na Listagem 16.

```
1 private double calculaConfianca(RegraAssociacao regra,
2           List<String> usuarios,
3           List<HashMap<String, Integer[]>> avaliacoesPositivas) {
4
5     List<Integer> listSeEEntao = new ArrayList<Integer>();
6     List<Integer> listSe = new ArrayList<Integer>();
7
8     /* Divide os parâmetros SE e ENTAO da regra em duas listas.
9      Uma lista mantém só o SE da regra e a outra lista o SE e o ENTAO */
10     for (int i = 0; i < regra.getListSe().size(); i++) {
11          addElemento(regra.getListSe().get(i), listSeEEntao);
12          addElemento(regra.getListSe().get(i), listSe);
13    }
14
15    for (int i = 0; i < regra.getListEntao().size(); i++) {
16          addElemento(regra.getListEntao().get(i), listSeEEntao);
17    }
18
19    Integer[] seEEntao = listSeEEntao.toArray(
20           new Integer[listSeEEntao.size()]);
21
22    Integer[] se = listSe.toArray(new Integer[listSe.size()]);
23    /* conta as avaliações positivas dos elementos SE e ENTÃO juntos. */
24    double total = contarAvaliacoes(seEEntao, usuarios,
25                avaliacoesPositivas);
26    /* conta as avaliações positivas dos elementos SE */
27    double totalSe = contarAvaliacoes(se, usuarios, avaliacoesPositivas);
28    /* calcula confiança.
29    return total / totalSe;
30 }
```

**Listagem 16: Método "calculaConfianca()"**

Assim como ocorre no cálculo do suporte, a confiança da regra é definida através de uma fração – exibida na Figura 3. Por conta disso, esse método obtém os elementos contidos nos parâmetros “Se” e “Então” da regra enviada por parâmetro, os converte em objetos do tipo *array* e conta as avaliações onde esses elementos foram classificados positivamente. Após, aplica esses valores na fórmula da confiança e retorna o resultado do cálculo.

Caso a regra possua uma confiança superior a mínima proposta (0.8), essa regra é considerada válida e adicionada em uma lista a parte. A etapa do cálculo da confiança é finalizada quando todos os conjuntos forem analisados e todas as regras de associação passarem pela fórmula.

A partir do resultado obtido na etapa anterior, é realizado um “refinamento” no algoritmo, isto é, uma implementação extra que visa obter resultados que se assemelham com os padrões de avaliação dos usuários. Para isso, inicia-se uma iteração nas regras válidas e é verificado se os estabelecimentos presentes no parâmetro “SE” da regra já foram avaliados positivamente pelo usuário, sendo que tal verificação é realizada pelo método “usuarioGostouDaCondicaoSe()”. A implementação deste método é exibida na Listagem 17.

```
1 private boolean usuarioGostouDaCondicaoSe(String usuario,
2           List<Integer> paramSe,
3           List<HashMap<String, Integer[]>> avaliacoesPositivas) {
4
5     int count = 0;
6     for (int i = 0; i < avaliacoesPositivas.size(); i++) {
7           Integer[] estabsAv =
8                 avaliacoesPositivas.get(i).get(usuario);
9           if (estabsAv!= null) {
10                /* Percorre a lista do parâmetro SE da regra e verifica
11                   se este está presente nas avaliações positivas do
12                   usuário. */
13                for (int j = 0; j < paramSe.size(); j++) {
14                      for (int k = 0; k < estabsAv.length; k++) {
15                            if (paramSe.get(j) == estabsAv[k]) {
16                                  count++;
17                            }
18                      }
19                }
20                break;
21          }
22    }
23    /* Se todos os elementos do SE estiverem nas avaliações positivas,
24       retorna verdadeiro.*/
25    return count == paramSe.size();
26 }
```

**Listagem 17: Método "usuarioGostouDaCondicaoSe()"**

# 5 CONCLUSÃO

Apesar da vasta gama de metodologias existentes que aplicam os conceitos de regras de associação, a utilização é bem maior no cunho empresarial, visando obtenção de lucros maiores, assim como a geração de estratégias de marketing para promover marcas e produtos. Este trabalho, no entanto, propôs o uso das regras de associação em um escopo onde os usuários comuns – sem vínculos comerciais com algum estabelecimento – podem usufruir das informações geradas.

Inicialmente foi destacada a evolução dos algoritmos aplicados a mineração de dados, citando de maneira introdutória como cada um deles funciona, bem como seus prós e contras. A partir desse contexto, foram adentradas referências ao *A priori*, algoritmo que guiou o desenvolvimento do restante do trabalho. Sua escolha ocorreu devido a efetividade e ampla utilização, além de se encaixar no contexto do presente trabalho.

Como resultado, foi desenvolvido um aplicativo móvel e um *web service* REST Java, cuja finalidade é sugerir locais gastronômicos para visitação, aplicando regras de associação.

Durante a codificação do trabalho, além do processamento comum do algoritmo *A priori*, foram adicionados alguns conceitos idealizados pelo autor do trabalho, com a finalidade de refinar os resultados gerados pelo algoritmo. Esse fato ocasionou algumas dificuldades, gerando a necessidade de pesquisas mais aprofundadas sobre a linguagem Java e a lógica que poderia ser utilizada nos métodos dos códigos-fonte. No entanto, isso acabou se mostrando algo assertivo pois intensificou a pesquisa do autor sobre os assuntos aplicados no trabalho, além de propor uma opção a mais para a geração de regras de associação.

Outra dificuldade encontrada no desenvolvimento foi a utilização de bibliotecas de terceiros para a realização de algum processamento, como por exemplo, a ORMLite, biblioteca para persistência de dados. Devido ao baixo conhecimento do autor, no início do trabalho, a respeito dessa biblioteca, as formas de implementação ainda não eram conhecidas e, por consequência, *bugs* surgiram. Tais problemas foram solucionados a partir de leituras na documentação da biblioteca.

A respeito do consumo do algoritmo *A priori*, vale destacar que a escolha dos valores de parâmetro (suporte 0.3 e confiança 0.8) se deu devido ao tamanho da base de dados explorada. Como não foi possível disponibilizar o aplicativo para ser consumido por terceiros, os experimentos foram realizados a partir de experiências do próprio autor e a base de consumo não ficou tão extensa. Ainda assim, foi possível analisar que em uma base de dados

pequena, a escolha por parâmetros de valores mais elevados tende a diminuir o número dos resultados, visto que a busca por associações se torna mais intensa. Por conta disso, esse projeto utilizou-se de valores mais moderados.

Como expectativas futuras, espera-se melhorar a usabilidade do aplicativo, ampliar as categorias de locais sugeridos para visita, introduzir algumas funcionalidades extras ao aplicativo, como exemplo, traçar uma rota até o local sugerido e tornar o processamento no *web service* mais performático. A ampliação da base de dados a ser explorada também se torna uma meta, visto que dessa maneira os resultados apresentados podem trazer informações mais ricas.

# REFERÊNCIAS

AGRAWAL, R.; IMIELINSKI, T.; SWAMI, A. **Mining associations rules between sets of items in large databases.** Apresentado a ACM SIGMOD International Conference on Management of Data. Nova York, USA, 1993.

AGRAWAL, R; SRIKANT, R. **Fast Algorithms for mining association rules.** Apresentado a 20º International Conference on Very Large Data Bases. San Francisco, USA, 1994.

AGRAWAL R; SRIKANT, R. Mining sequential patterns. Apresentado a 11º Iternational Conference on Data Engineering. Taipen, Taiwan, 1995.

BERRY, Michael J. A; LINOFF, Gordon. **Data mining techiques for marketing, sales and Customer Support.** John Wiley e Sons, Inc., 1997.

BORGES, Priscilla. **11 mil candidatos do SISU mudaram de Estado para estudar em 2011.** Disponível em: <http://ultimosegundo.ig.com.br/educacao/11+mil+candidatos+do+sisu+mudaram+de+estado+para+estudar+em+2011/n1597035293078.html>. Acesso em 19 de março de 2015.

CAMILO, Cássio Oliveira; SILVA, João Carlos da. **Mineração de Dados: Conceitos, Tarefas, Métodos e Ferramentas.** Relatório Técnico. Agosto, 2009.

CARVALHO, Deborah Ribeiro. **Árvore de decisão/algoritmo genético para tratar o problema de pequenos disjuntos em classificação de dados.** Tese de Doutorado. Rio de Janeiro, 2005.

COMACHIO, Vanderson. **Funcionamento de Banco de Dados em Android: Um estudo experimental usando SQLite**. Tese de Mestrado. Medianeira, 2011.

CORTÊS, Sérgio da Costa; PORCARO, Rosa Maria; LIFSCHITZ, Sérgio. **Mineração de Dados – Funcionalidades, Técnicas e Abordagens**. 2002.

FAYYAD, U. M. **From data mining to knowledge discovery: an overview. In: Advances in knowledge discovery and data mining**. California: AAAI/The MIT, 1996. p.1-34.

GALVÃO, Noemi Dreyer; MARIN, Heimar de Fátima. **Técnica de mineração de dados: uma revisão da literatura**. São Paulo – Set/Out 2009.

GIROLLETE, Rowan Ben-Hur Andrighetti. **Android: Visão Geral**. Tese de Especialização. Cascavel, 2012.

GOLDSCHMIDT, Ronaldo; PASSOS, Emmanuel. **Data Mining - Um guia prático**. Rio de Janeiro: Elsevier, 2005.

LECHETA, Ricardo. **Google Android: Aprenda a criar aplicações para dispositivos móveis com o Android SDK**. 3º Edição. Novatec, 2013.

MELANDA, Edson Augusto. **Pós-processamento de Regras de Associação.** Tese de Doutorado. São Carlos, 2004.

MOBILEIN. **Conceitos chave do Android – parte 1 – Camadas**. 2010. Disponível em <http://mobilein.com.br/?p=55>. Acesso em 18 de março de 2015.

**Mobile/Tablet Top Operating System Share Trend.** Disponível em: <https://netmarketshare.com/>. Acesso em 26 de junho de 2015.

OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas Operacionais**. Volume III. 03/12/2001. Disponível em: <http://www.das.ufsc.br/~romulo/artigos/Romulo-Carissimi-Simao-Rita2001.pdf>. Acesso em 18 de março de 2015.

PRADO, Sérgio. **Introdução ao funcionamento interno do Android**. 15/05/2011. Disponível em: <http://sergioprado.org/introducao-ao-funcionamento-interno-do-android/ >. Acesso em 19 de março de 2015.

REZENDE, S. O. **Sistemas Inteligentes: Fundamentos e Aplicações**. 1º Edição. Barueri, SP: Manole, 2003.

REZENDE, S.O; PUGLIESI, J. B; MELANDA, E. A; PAULA, M. F. **Mineração de Dados.** 1º Edição, Capítulo 12, pp. 307-335.

*

SARAWAGI, S.; THOMAS, S. AGRAWAL, R.. **Integrating association rule mining with relational database systems: Alternatives and implications.** ACM SIGMOD Rec. 1998.

SFERRA, Heloisa Helena; CORREA, Ângela M. C. Jorge. **Conceitos e Aplicações de Data Mining**. Revista Ciência & Tecnologia, PP. 19-34. 2003.

**SISU provoca aumento de alunos de diferentes estados em federais.** Disponível em: <http://g1.globo.com/jornal-da-globo/noticia/2014/01/sisu-provoca-aumento-de-alunos-de-diferentes-estados-em-federais.html>. Acesso em 19 de março de 2015.

TOSIN, Carlos. **Conhecendo o Android.** Disponível em: <http://www.dicas-l.com.br/arquivo/conhecendo_o_android.php>. Acesso em 02 de junho de 2015.

VIEIRA, Henrique. **Android é o sistema operacional mais utilizado no planeta.** Disponível em <http://blog.maisestudo.com.br/android-e-o-sistema-operacional-mais-utilizado-no-planeta/>. Acesso em 05 de junho de 2015.

WEIS, Sholon M.; INDURKHYA, Nitim. Predilect Data Mining. Morgan Kaufmann Publishers, Inc., 1999.