André Felipe Santos
Celso Vital Crivelaro

Extrator de informações de textos jornalísticos sobre partidas de futebol

Orientador:
Prof. Dr. Ricardo Luis de Azevedo da Rocha

ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

São Paulo

2008

Dedicatória

Dedico esse trabalho à toda a minha família, especialmente ao meu pai, por todo o apoio, dedicação e motivação a mim durante toda a vida e principalmente durante toda a graduação.

Também dedico à Daniele Monteiro, pela pessoa que é na minha vida e que sempre me motivou nos momentos difíceis.

Celso Vital Crivelaro

Dedicatória

Dedico este trabalho à minha família e amigos, pela torcida, e em especial à minha mãe, por seu apoio e paciência durante todo o curso.

André Felipe Santos

Agradecimentos

Ao professor Ricardo Luis de Azevedo Rocha pelo apoio moral, pelo bom humor, pelo seu ensino e principalmente pela orientação do trabalho.

Ao Laboratório de Linguagens e Técnicas Adaptativas (LTA) pelo apoio ao trabalho, e especialmente ao Rodolpho Eckhardt pela sugestão de ferramentas.

Ao Departamento de Engenharia de Computação e Sistemas Digitais (PCS) pelo curso de Engenharia de Computação Cooperativo.

A Pablo Gamallo da Universidade de Santiago de Compostela pelo treinamento do Etiquetador, cujo trabalho nos ajudou muito nesse projeto.

Aos desenvolvedores das ferramentas utilizadas por serem softwares livres, sem as quais não haveria um projeto sem custos.

A César Vital Crivelaro pela revisão do texto.

A todos que apoiaram esse trabalho.

Resumo

O trabalho apresenta técnicas de processamento de linguagem natural para a retirada e busca de informações de textos jornalísticos de partidas de futebol.

O texto apresenta todas as fases do processamento de linguagem natural, descrevendo a teoria envolvida, modelagem, ferramentas e o processo utilisado.

Palavras-chave: linguagem natural, python, nltk, futebol, texto.

Abstract

This project features natural language processing techniques for information extraction from soccer journalistic texts.

This text presents all stages of natural language processing, describing the theory envolved, the modelling, the tools and process used.

Keywords: natural language, python, nltk, soccer, text.

Sumário

Lista de abreviações e siglas

Lista de Figuras

Lista de Tabelas

1 Introdução p. 13

1.1 Objetivos . . . . p. 14

1.2 Justificativa . . . . p. 14

1.3 Divisão do Documento . . . . p. 15

2 Referencial Teórico e Tecnológico p. 16

2.1 Construção da Ontologia . . . . p. 16

2.2 Processamento de Linguagem Natural . . . . p. 16

2.2.1 Análise Léxica . . . . p. 16

2.2.2 Análise Sintática . . . . p. 18

2.2.3 Semântica e Pragmática . . . . p. 18

2.3 Agentes . . . . p. 19

2.3.1 Agente reativo . . . . p. 19

2.3.2 Mecanismos de busca . . . . p. 20

3 Materiais e Métodos p. 21

3.1 Python . . . . p. 21

3.2 NLTK - Natural Language Toolkit . . . . p. 22

3.3 TreeTagger - Etiquetador Gramatical . . . . p. 23
3.4 Eclipse - IDE . . . . p. 23
3.5 Complemento de IDE - Pydev . . . . p. 24
3.6 Framework - Django . . . . p. 25
3.7 GTK+ - Gnome Toolkit . . . . p. 25
3.8 Glade . . . . p. 26
3.9 SVN - Controle de Versões . . . . p. 26
3.10 PostgreSQL . . . . p. 27
3.11 PgAdmin III . . . . p. 28

4 Implementação p. 30

4.1 Premissas e Escopo . . . . p. 30
4.1.1 Diretrizes do Trabalho . . . . p. 31
4.2 Arquitetura . . . . p. 32
4.2.1 Arquitetura Geral . . . . p. 32
4.2.2 Estrutura de Pacotes . . . . p. 32
4.3 Extrator de Informações . . . . p. 34
4.3.1 Analisador Léxical . . . . p. 34
4.3.2 Analisador Sintático . . . . p. 37
4.3.3 Analisador Semântico e Gerador da Estrutura . . . . p. 40
4.4 Estrutura de Dados - Persistência . . . . p. 46
4.5 Busca de Informações . . . . p. 48
4.5.1 Interface com usuário . . . . p. 48
4.5.2 Agente reativo . . . . p. 48
4.5.3 Módulo de processamento . . . . p. 52

5 Testes e Resultados p. 53

5.1 Extrator de Informações . . . . p. 53
5.1.1 Objetivos do Teste . . . . p. 53
5.1.2 Testes e Resultado . . . . p. 54
5.2 Buscador de Informações . . . . p. 55

6 Considerações Finais p. 56
6.1 Contribuições . . . . p. 58
6.2 Dificuldades no projeto . . . . p. 58
6.3 Trabalhos Futuros . . . . p. 59
6.3.1 Extensão . . . . p. 59
6.3.2 Adaptatividade . . . . p. 60
6.3.3 Desempenho . . . . p. 60

Referências Bibliográficas p. 61

Anexo A - Conjunto de *Tags* p. 63

Anexo B - Resultado dos testes com o Extrator p. 64

Anexo C - Relação de Equipes, Estádios e Cidades do Campeonato Paulista de 2008 p. 66

Lista de abreviações e siglas

BSD - Berkeley Software Distribution

GATE - General Architecture for Text Engineering

GIMP - GNU Image Manipulation Program

GNU - GNU is not Unix

GNOME - GNU Object Model Environment

GPL - GNU Public License

GTK - GIMP ToolKit

HTML - HyperText Markup Language

IDE - Integrated Development Environment

LISP - LISt Processing

LTA - Laboratório de Linguagens e Técnicas Adaptativas

NLTK - Natural Language Toolkit

NLP - Natural Language Processing

PCS - Departamento de Engenharia de Computação e Sistemas Digitais

POS - Part of Speech

RAD - Rapid application development

SVN - Subversion

USP - Universidade de São Paulo

XML - eXtensible Markup Language

Lista de Figuras

3.1 Logotipo da linguagem Python . . . . p. 22

3.2 IDE Eclipse . . . . p. 24

3.3 Logotipo do GTK+ . . . . p. 25

3.4 Tela do Glade . . . . p. 26

3.5 Logotipo do SVN . . . . p. 27

3.6 Logotipo do PostgreSQL . . . . p. 28

3.7 Tela do PgAdmin III . . . . p. 29

4.1 Arquitetura Geral do Sistema . . . . p. 32

4.2 Estrutura Geral de Pacotes . . . . p. 33

4.3 Arquitetura do Extrator de informações . . . . p. 34

4.4 Etiquetador Gramatical . . . . p. 35

4.5 Saída do TreeTagger . . . . p. 36

4.6 Refinamento Lexical . . . . p. 37

4.7 Funcionamento do Analisador Sintático . . . . p. 38

4.8 Analisador Semântico e Gerador da Estrutura . . . . p. 40

4.9 Exemplo de árvore de RESULTADO . . . . p. 41

4.10 Exemplo de Árvore de FUTEBOL . . . . p. 43

4.11 Modelo Entidade-Relacionamento . . . . p. 46

4.12 Página inicial da busca . . . . p. 49

Lista de Tabelas

3.1 Exemplos de tipos nativos do Python . . . . p. 22

3.2 Limites gerais do banco de dados PostgreSQL . . . . p. 28

4.1 *Tags* utilizadas para fazer o refinamento . . . . p. 36

4.2 Regras de formação da árvore sintática . . . . p. 39

5.1 Resultado da partida retirada por um Humano . . . . p. 54

5.2 Resultado da partida retirada pelo Extrator . . . . p. 54

5.3 Resultados dos testes . . . . p. 54

A.1 Relação da unidade morfológica com a *Tag* empregada pelo etiquetador . . . p. 63

B.1 Resultado da partida retirada por um Humano . . . . p. 64

B.2 Resultado da partida retirada pelo Extrator . . . . p. 65

B.3 Relações que foram extraídas dos textos . . . . p. 65

C.1 Relação da equipe com o seu estádio e a cidade de origem . . . . p. 66

1 *Introdução*

O processamento de linguagem natural tornou-se uma área muito explorada devido ao desejo da interatividade de comunicação do homem com a máquina usando a mesma linguagem utilizada para comunicar-se com outro ser humano.

O uso de processamento de linguagem natural tem como princípio a interação de máquinas com seres humanos tanto como no recebimento de comandos por voz, interpretar um texto ou até mesmo um comando escrito. Essa interação possui uma complexidade tão elevada que exige uma interdisciplinaridade entre computação e linguística para obter o resultado desejado.

As técnicas de processamento de linguagem natural realizam a ponte entre a linguagem natural e a linguagem de máquina. Assim, com essas técnicas, é possível tratar as palavras como informações, ou seja, podemos processá-las e criar relações lógicas cujas informações são compreendidas por computadores.

Assim, esse trabalho propõe a construção de agentes que possam fazer a extração de informações de uma partida de futebol de textos jornalísticos e, em conjunto, um Buscador para encontrar informações sobre os dados das partidas que são retiradas pelo Extrator.

A escolha de textos jornalísticos em língua portuguesa foi feita pelo desafio de escrever um Extrator na língua materna dos desenvolvedores. O desafio existe por haver poucas referências e poucas ferramentas para língua portuguesa em comparação à língua inglesa.

1.1 Objetivos

Os objetivos do trabalho são:

- Criar um extrator de informações sobre uma partida de futebol de texto jornalístico;
- Organizar as informações extraídas em uma estrutura de dados;
- Buscar e exibir a informação em um formato compreensível aos usuários.

Para atingir esses objetivos foi feito um estudo de processamento de linguagem natural e de técnicas de compilação para classificar as palavras e tirar significados de textos. Além disso, foi construído um módulo que também usa processamento de linguagem natural para buscar as informações extraídas e exibi-las em páginas web.

1.2 Justificativa

O trabalho justifica-se pelo desenvolvimento da *web semântica*, cujos serviços e páginas da Internet têm significado para computadores, conseguindo interagir e fazer operações sobre essa rede. O objetivo do trabalho é, portanto, construir os agentes que fazem tal extração. Para obter significado é necessário o uso intenso de metadados para estruturar a informação textual da Internet, que é uma informação desestruturada.

A escolha de textos jornalísticos foi feita pela qualidade do texto escrito, pois possui poucos erros de ortografia e pouca ambiguidade. Esse fator facilita muito a extração de informações.

O tema futebol, todavia, foi escolhido por gosto pessoal dos autores do trabalho e também pela facilidade de encontrar textos sobre partidas na Internet. Outro motivo também é o conhecimento dos termos do assunto, conhecidos como *jargões*, como *golear*, que significa "vitória com muitos gols".

1.3 Divisão do Documento

- Introdução: justificativa do nosso projeto, além da presente divisão do documento;
- Referencial teórico e tecnológico: teoria envolvida nos módulos do projeto;
- Materiais e métodos: componentes utilizados para o desenvolvimento do projeto e sua importância nos módulos;
- Implementação: descrição detalhada da implementação do projeto, mostrando cada fase do processamento de linguagem natural do Extrator de informações e do Buscador;
- Testes e resultados: plano de testes e análise dos resultados;
- Considerações finais: conclusões sobre o desenvolvimento do sistema.

2 *Referencial Teórico e Tecnológico*

Para o projeto foram analisadas algumas técnicas e tecnologias para viabilizar as especificações. Nesse capítulo serão abordadas as tecnologias escolhidas.

2.1 Construção da Ontologia

A Ontologia, segundo a filosofia grega antiga, é o estudo da natureza do ser, da sua existência e dos seus relacionamentos. Ela estuda o que existe ou que pode ser considerado como existente, como os seres se agrupam, quais as suas divisões e hierarquias e como se relacionam.

O termo foi adotado pela área de Inteligência Artificial para a representação do conhecimento sobre um domínio específico, que concorda com a idéia criada pelos filósofos.

2.2 Processamento de Linguagem Natural

O processamento de linguagem natural é a principal proposta desse trabalho para a retirada de informações sobre as partidas de futebol. Assim, são mostradas algumas técnicas de processamento de linguagem natural junto com as ferramentas utilizadas no projeto.

2.2.1 Análise Léxica

O objetivo do analisador léxico é quebrar as partes do texto em pedaços (chamados de *tokens*) para poder classificá-los.

Tokenizador e Etiquetador Gramatical (*POS Tagger*)

A primeira fase do processamento é a quebra de um texto em *tokens*, realizada pelo *tokenizador*. Os *tokens* são unidades de texto que tem um sentido, ou seja, um item lexical, como uma palavra, um numeral ou uma pontuação.

O desafio da geração de *tokens* não apenas a separacão das palavras em que há espaços ou em que há pontuações, mas o *tokenizador* precisa interpretar uma unidade lexical para que o etiquetador ou *tagger* marque com a *tag*. A *tag* contém alguma classificação do *token* e, no caso de um etiquetador gramatical, a *tag* tem o signicado morfológico de tal.

Podemos ver essa dificuldade no caso das abreviações como 'S.P.' para 'São Paulo', em que os pontos não são pontos finais, mas pontos de abreviação. Outro caso é os numerais que usam pontos por questão estética, como o número '333.456', que é um *token* completo ao invés de ser três tokens '333', '.' e '456'.

Além disso, temos o problema de identificação de *tokens* em verbos e suas reduções. No caso do verbo *vencer* há variações de tempo e pessoa, como *venceram*, *venceu* e *vence*; o que poderia ser reduzido ao lexema *venc* para uma melhor análise.

Para o projeto foi utilizado um etiquetador gramatical chamando *TreeTagger* produzido pela Universidade de Stuttgart. Ele usa um recurso de Inteligência Artificial chamado 'árvore de decisão' para o aprendizado e para a classificação do *token*, tendo a dupla funcionalidade de *tokenizador* e etiquetador.

Foi utilizado um treinamento do *TreeTagger* realizado por Pablo Gamallo Otera da Universidade de Santiago de Compostela com um *corpus* da língua portuguesa. As *tags* usadas no treinamento são referenciadas no Anexo A.

Gazetteer - Listas de referências

As listas ajudam o analisador léxico a classificar os *tokens* por palavras já definidas. O *Gazetteer* é um dicionário com informações sobre lugares e nomes, cuja utilidade foi grande em mapas na antiguidade. No contexto de processamento de linguagem natural, o *Gazetteer* é uma lista de referências sobre qualquer assunto.

Para ajudar na analise léxica, o *token* é comparado com os elementos de cada lista e, estando presente nela, o *token* é etiquetado com o valor correspondente.

Exemplo de uma lista de times de futebol:

Palmeiras

São Paulo

Corinthians

Santos

2.2.2 Análise Sintática

A análise sintática reconhece uma sequência de palavras que formam um frase e constrói uma árvore de derivação, que mostra as relações entre as palavras que compoem essa frase.

Para isso, é preciso um analisador sintático ou *parser* que tenha acesso aos *tokens* produzidos pelo analisador léxico e uma gramática que contenha as regras de formação das frases.

As gramáticas são classificadas em quatro (CHOMSKY, 1959), segundo Noam Chomsky (1959):

- **Regular:** gramática restrita cujas regras de formação só permitem geração à esquerda;
- **Livre de contexto:** gramática que permite regras de geração em ambos os sentidos por ter um auto-recursivo central;
- **Sensível ao contexto:** gramática que consegue expressar-se dependendo do seu contexto, e a expressão não pode sofre nenhuma redução;
- **Irrestrita:** gramática sem restrições, sem nenhuma limitação imposta.

Para a simplicidade do projeto foi usada apenas uma gramática livre de contexto. Sendo (F) a frase que é composta pelo sintagma nominal (SN), que é o participante de uma ação ou argumento dos verbos; e um sintagma verbal (SV), que são as partes da frase iniciadas pelo verbo, temos a seguinte regra de formação:

F $\Rightarrow$ SN SV

SN $\Rightarrow$ Substantivo

SN $\Rightarrow$ Verbo SN

Com essa regra, o analisador sintático percorre a lista de *tokens* e procura as relações segundo a sua abordagem.

2.2.3 Semântica e Pragmática

A Semântica tem como objetivo estudar o significado das palavras e das estruturas; a Pragmática, estudar como o significado das expressões pode variar no seu contexto.

A semântica estuda a suas palavras e o desafio dos vários significados que elas podem ter, como *jogador*, que pode ser *jogador de futebol*, *jogador de vôlei* ou *jogador de pôquer*. Outras formas de ambiguidade podem ser vistas na construção sintática.

Para o projeto, o escopo do assunto foi delimitado para partidas de futebol do Campeonanto Paulista de 2008. Assim, o trabalho de interpretação fica simplificado para a extração e a busca de informações.

2.3 Agentes

Os agentes consistem em programas cujo comportamento é **racional**. Por racional, compreende-se que o agente agirá de forma a atingir o melhor resultado possível. No tópico de inteligência artificial, podemos dividir o software de acordo com duas divisões: quanto ao pensamento e quanto à forma de agir. (NORVIG; RUSSEL, 2002). Conforme a abordagem atual, decidimos, para a busca, utilizar um agente que age racionalmente, em contraste a um agente que pense, humana ou racionalmente, ou aja humanamente.

Decidimos esta abordagem em virtude do pensamento não ser devidamente demonstrado e a ação humana não ser necessariamente a melhor a ser tomada. Portanto, decidimos pela *ação racional*, que é a melhor possível, em função do ambiente a ser examinado e cursos de ação a serem tomados.

Pela complexidade de nosso trabalho, escolhemos utilizar um *agente reativo* para realizar as buscas. Em função das buscas estarem em um conjunto restrito, acreditamos que esta arquitetura seja o bastante, principalmente porque o universo em questão, as palavras da busca, é completamente observável, e o conjunto de possibilidades não é tão difícil de ser coberto.

2.3.1 Agente reativo

O agente reativo é um agente racional (NORVIG; RUSSEL, 2002), cujo comportamento é um pouco mais rico que o do agente orientado a tabelas, que verifica a entrada e tem necessidade de que *cada combinação* seja prevista. Mesmo em um universo limitado como o do Campeonato Paulista de 2008, há um grande número de combinações possíveis, o que torna essa abordagem impraticável.

Com relação aos agentes que levam em conta o histórico e o estado atual, decidimos que essa abordagem seria mais complexa que o realmente necessário. A justificativa para isso é de que grande parte das buscas não haverá necessidade de refinamento em virtude do tipo que será feito, conforme os resultados de jogos específicos.

O comportamento do agente reativo, ao contrário do agente orientado a tabelas, pode ser determinado em função de algumas abstrações. Em nosso caso, temos o interesse em retor-

nar confrontos, como no caso de entradas de dois times. Com a entrada de mais times, são retornados os possíveis confrontos, ou seja, as combinações dois a dois entre os mesmos. Se a entrada for um ou mais estádios, são retornados os confrontos realizados nos mesmos. São apenas exemplos, mas isso mostra que é uma abordagem viável e abrangente ao mesmo tempo.

2.3.2 Mecanismos de busca

Os mecanismos de busca atuais consistem em sistemas com três partes principais a saber: (LEVENE, 2005)

- *Web crawling*
- Indexação
- Busca

O *Web crawling* consiste em um programa que acessa continuamente links de páginas, analisando-as e obtendo as informações que serão enviadas para a indexação, que é a montagem de uma estrutura de dados com rápido acesso a uma grande quantidade de informação - obtida pelo *Web crawling*. A busca ocorre quando o usuário entra com termos a serem localizados em uma caixa de texto, esta em uma página web, que tem o papel de ser a interface do mecanismo de busca com o usuário.

Com relação ao nosso sistema, que inclui um módulo de busca, podemos dizer que é semelhante na medida em que temos um índice, que são as partidas e seus dados armazenados no banco de dados, sendo feita uma busca com base no agente reativo.

A *web crawling*, do ponto de vista de rastreamento de páginas na Internet, está fora do escopo de nosso projeto. Pode ser uma extensão possível, principalmente para se fazer um sistema que acompanhe um campeonato em andamento, ou mesmo para armazenar dados de campeonatos que já aconteceram e de atuais.

Por outro lado, as informações reunidas e a indexação das mesmas é um processo bem semelhante ao que nosso sistema realiza na extração de informações que são armazenadas em um banco de dados. Podemos, portanto, concluir que nosso sistema exerce o papel de um mecanismo de busca.

3 *Materiais e Métodos*

3.1 Python

Python (PYTHON..., 2008) é uma linguagem de programação de multi-paradigmas orientada a objetos e interpretada, que é suportada por várias plataformas, como Windows e Linux. Foi criada por Guido van Rossum em 1991.

A linguagem Python prioriza a legibilidade do código e a agilizade do produção em conjunto com uma sintaxe simples e precisa. Além disso, apresenta uma biblioteca-padrão vasta, oferecendo muitas ferramentas para o desenvolvimento de uma grande gama de programas, incluindo um bom número de estruturas de dados que são suficientes para grande parte de programação usual.

Um fator interessante que comprova a legibilidade dos códigos-fonte feitos em Python é que a sua sintaxe é baseada em identação para delimitar estruturas ao invés de usar chaves como nas linguagens derivadas do C:

```
def valor1():
    while True:
        try:
            c = int(raw_input('Primeiro Valor: '))
            return c
        except ValueError:
            print 'Inválido!'
```

Atualmente a linguagem Python é bastante usada para diversas finalidades pela sua portabilidade, desde software embarcado a servidores de aplicação. Ela é utilizada como linguagem de *script* em diversos sistemas, como o GIMP e o próprio GNOME, servidores de aplicação e computação gráfica.

A utilização da linguagem Python é preferencial pela grande legibilidade e por oferecer todas as possibilidades citadas, *frameworks* disponíveis e a vasta biblioteca-padrão. Outro fator importante é a quantidade de tipos de dados nativos que foram muito utilizados no projeto como os que representam conjuntos e tuplas:

Tipo	Descrição	Exemplo
str	string ou cadeia de caracteres	'futebol'
list	lista ordernada	['Palmeiras', 'São Paulo']
tuple	tuplas	('Palmeiras', 'Palestra Itália')
dict	conjunto associativo	{'Palmeiras':'Palestra Itália', 'São Paulo':'Morumbi'}
int	inteiro	0, 67, 379

Tabela 3.1: Exemplos de tipos nativos do Python

Para o projeto, foi utilizada a versão 2.5, apesar de já existir uma versão 2.6 que prepara os usuário para uma versão 3.0. O uso da versão antiga deve-se a sua estabilidade e bibliotecas já conhecidas. Caso haja migração para a versão 3.0, todos os projeto deverão ser revistos, pois a retrocompatibilidade não é garantida.

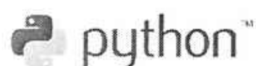

Figura 3.1: Logotipo da linguagem Python

3.2 NLTK - Natural Language Toolkit

O NLTK (NATURAL...,) é um grupo de módulos para a linguagem Python para o processamento de linguagem natural. Foi desenvolvido em conjunto com o curso de linguistica computacional da Univerdade da Pennsylvania em 2001. (BIRD; LOPER, 2004)

Ele contém módulos específicos como os *Tokenizadores* para quebra de texto, analisadores sintáticos, semânticos e, em conjunto, cerca de 40 *corpora* para fazer comparações e usar como parâmetros.

A estrutura do NLTK centra-se nesses quatro pontos:

- **Simplicidade**: o *framework* é construído de forma simples para que o usuário possar facilmente chamar as suas funções;
- **Consistência**: houve um esforço para que as estruturas de dados e interfaces dos módulos fosse consistentes;

- **Extensibilidade**: o *toolkit* acomoda novos componentes para replicar ou estender funcionalidades
- **Modularidade**: todos os componentes do NLTK são dividos em módulos com interfaces bem definidas.

No site do projeto há o livro *NLTK Book* (BIRD; LOPER; KLEIN, 2008) que contém exemplos do uso do NLTK com a teoria de processamento de linguagem natural.

3.3 TreeTagger - Etiquetador Gramatical

TreeTagger é um POS Tagger independente de linguagem desenvolvido pelo Instituto de Linguística Computacional da Universidade de Stuttgart. Foi usado com sucesso para línguas como Inglês, Francês, Alemão, Português, Espanhol, Italiano, Grego e Chinês. É facilmente adaptável à outras línguas se houver um um *corpus* já etiquetado com uma lista de *tags*. (TREETAGGER...,) (SCHMID, b) (SCHMID, a)

É um etiquetador que usa árvores de decisão no seu treinamento com um *corpus* já etiquetado. Para esse projeto, foi usada uma versão com um *corpus* de língua portuguesa já treinado por Pablo Gamallo da Universidade de Santiago de Compostela. (GAMALLO,)

3.4 Eclipse - IDE

A IDE Eclipse (ECLIPSE...,) é um ambiente de programação de licença livre que permite o desenvolvimento de software em várias linguagens de programação, sendo Java a principal. Ele é mantido pela fundação Eclipse que inicialmente foi criada por um consórcio das empresas Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft e Webgain.

Contendo uma boa usabilidade, a IDE Eclipse possui recursos de complementação de código, sugestões e detecção de erros. Além disso, contém muitos *plugins* que permitem estender as funcionalidades da IDE. Assim, a Eclipse suportas as linguaens de programação Java, C/C++, Python, Ruby etc.

Esses *plugins* têm a vantagem de implementar novas funcionalidades à IDE Eclipse, o que permite o uso de várias funcionalidades integradas, como editor de Latex e controlador de versões SVN.

Esse ambiente foi escolhido ao projeto pela familiaridade dos desenvolvedores com a ferramenta e o vasto conhecimento que há na comunidade de software livre sobre os *plugins*.

Figura 3.2: IDE Eclipse

Foi usada a versão 3.4 que tem o nome de Ganymede.

3.5 Complemento de IDE - Pydev

Pydev (PYDEV. . . ,) é um *plugin* para o Eclipse cujo objetivo é integrar a linguagem Python à IDE Eclipse.

Com ele pode-se desenvolver módulos e pacotes Python e obter os benefícios de depuração de código que o Eclipse fornece. Outro benefício é as outras extensões de controle de versão que integram a plataforma.

3.6 Framework - Django

Django (DJANGO. . . ,) é um *framework* para desenvolvimento web para a linguagem de programação Python. Ele inclui todos os componentes relevantes a este desenvolvimento, incluindo a camada de persistência e exibição, permitindo que se desenvolva um aplicativo web com arquitetura de camadas bem definidas. Atualmente, é recomendado pelo próprio criador da linguagem, Guido van Rossum, para este tipo de desenvolvimento. (HOLOVATY, 2006)

Sua utilização ajuda na interface do buscador, e, mais importante, a camada de persistência, para armazenar os resultados da extração no banco de dados.

3.7 GTK+ - Gnome Toolkit

GTK+ (THE...,) é um *toolkit* multi-plataforma para a construção de interfaces gráficas. Seu nome completo é **GIMP Toolkit**.

Apesar de ser construída em C, o GTK tem ligações com muitas outras linguagens de programação como o próprio C, Java, Python, Ruby, entre outras..

Em relação à parte gráfica, o GTK+ tem recursos como temas, integração com a base gráfica do Sistema Operacional, conhecido como *look-and-feel*, suporte a internacionalização e localização etc. Ele implementa as prinicipais interfaces gráficas conhecidas como botões, janelas, displays, menus, toolbars, caixas de texto, layouts etc.

Figura 3.3: Logotipo do GTK+

3.8 Glade

Glade (GLADE...,) é um construtor de interface gráfica para GTK e GNOME licenciado pela GNU GPL e é distribuído na versão 3.4. Por esse motivo é uma ferramenta RAD pois propicia o rápido desenvolvimento de interface gráfica.

O arquivo de desenho é gravado no formato XML e pode ser usado em várias linguagens de programação, especialmente Python.

3.9 SVN - Controle de Versões

Para o controle de versões dos códigos é usado o Subversion, também conhecido como SVN. (SUBVERSION,)Essa ferramenta é utilizada para o controle de versões do código, que pode ser alterado pelos 2 membros da equipe ao mesmo tempo e também pela necessidade de replicação por segurança, ou seja, evitar perdas de código ou voltar para um versão anterior. Também tem recursos de travamento de arquivo, impedindo a sua edição e *merge* de versões.

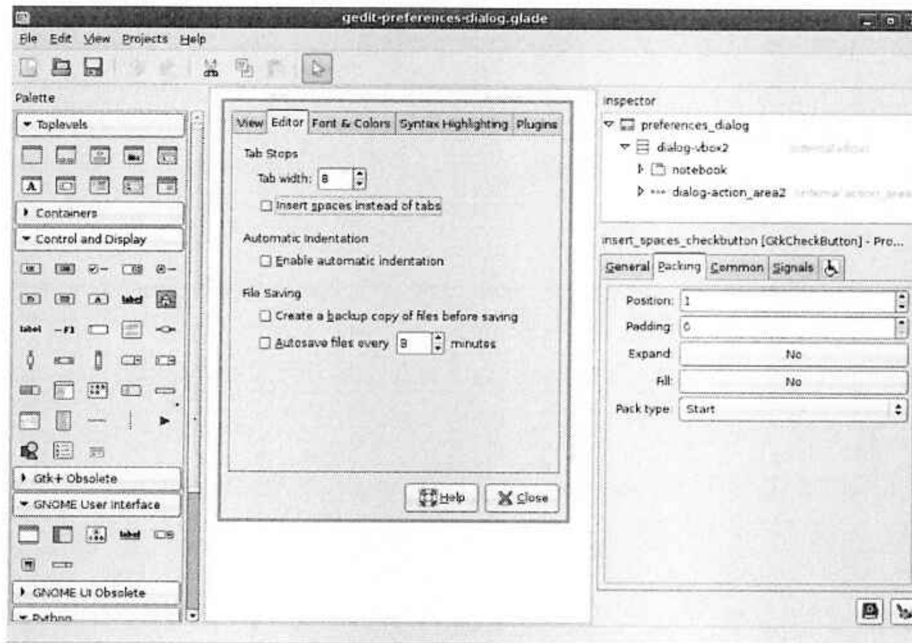

Figura 3.4: Tela do Glade

Funciona as principais plataformas como Unix, Win32, BeOS, OS/2, MacOS X e Linux e é mantido pela comunidade chamada Tigris.org. É usadas nos principais projetos de código aberto como Apache Software Foundation, KDE, GNOME, Free Pascal, FreeBSD, GCC, Python, Django, Ruby, Mono, SourceForge.net, entre outros.

Uma necessidade é um servidor para armazenamento das versões, e para isso foi utilizado o serviço de hospedagem do Google (GOOGLE. . . ,). Para a integração com a IDE, foi utilizado um *plugin* do Eclipse chamado SubEclipse (SUBCLIPSE,).

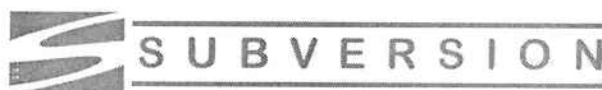

Figura 3.5: Logotipo do SVN

3.10 PostgreSQL

PostgreSQL (POSTGRESQL. . . ,) é um sistema gerenciador de banco de dados relacional distribuído em código aberto. Ele é famoso pelos seus recursos e por ser um dos bancos de dados mais usados na comunidade de software livre.

Ele é totalmente compatível com as plataformas UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), Linux e Windows e tem ligações com as principais linguagens de programação como Java (JDBC), ODBC, Perl, Python, Ruby, C, C++, PHP, Lisp, Scheme, Qt

e tantas outras.

É totalmente ACID, ou seja, respeita atomicidade, consistência, integridade e durabilidade. Além de suportar vários tipos de campos como binário, imagens, audio e vídeo, além dos tipos definidos em ANSI-SQL92/99 (ANSI-SQL92/99,) como INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP.

O PostgreSQL também roda *store procedures* nas mais variadas linguagens de programação como Java, Perl, Python, Ruby, Tcl, C/C++ e o próprio PL/pgSQL.

Para ter-se uma idéia da robustez do PostgreSQL, aqui mostra-se uma tabela com as informações sobre armazenamento:

Limite	**Valor**
Tamanho máximo de um banco de dados	Sem limites
Tamanho máximo de uma tabela	32 TB
Tamanho máximo de uma linha	1.6 TB
Tamanho máximo de um campo	1 GB
Quantidade máxima de linhas por tabela	Sem limites
Quantidade máxima de colunas por tabela	250 – 1600
Quantidade máxima de índices por tabela	Sem limites

Tabela 3.2: Limites gerais do banco de dados PostgreSQL

Os recursos mais atraentes são a grande capacidade de armazenamento, criação de índices e desempenho competitivo. Também contém muito conhecimento distribuído pela comunidade, o que o tornou muito atrativo ao projeto.

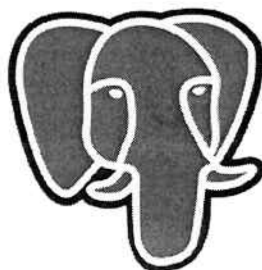

Figura 3.6: Logotipo do PostgreSQL

Para o projeto foi usada a versão 8.3.

3.11 PgAdmin III

PgAdmin (PGADMIN. . . ,) gráfica para o gerencimanento do banco de dados PostgreSQL.

Nele é possível inserir consultas SQL via interface gráfica e obter as respostas sobre essa mesma interface. Também é possível criar novas tabelas e banco de dados e fazer as operações que o PostgreSQL fornece.

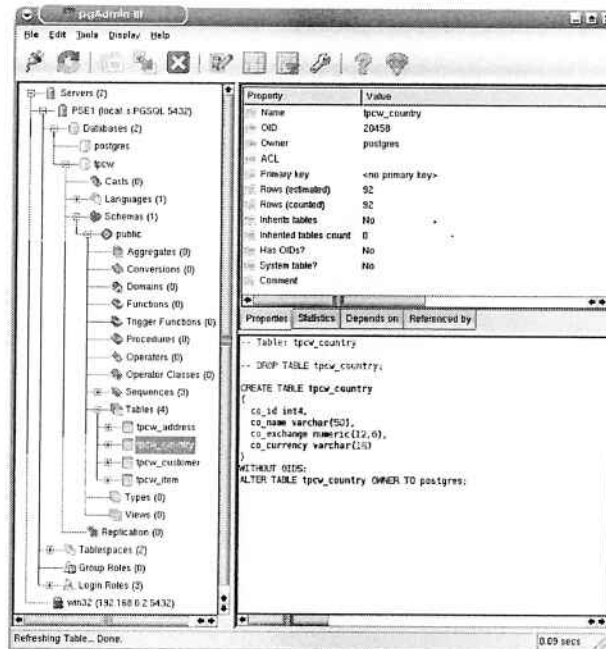

Figura 3.7: Tela do PgAdmin III

4 *Implementação*

O método proposto nesse projeto mostra como fazer a extração de informações de um texto que, no caso, é sobre partidas de futebol. O sistema inteiro pode ser extensível para novas funcionalidades ou poder ser modificado para outros assuntos, como voleibol; ou assuntos que não envolvam esportes.

O agente é implementado para automatizar a extração de informações de um texto jornalístico e guardar numa estrutura de dados, o que permite uma busca facilitada. As capacidades do sistema podem ser descritas como:

1. Utilizar técnicas de processamento de linguagem natural para identificar as equipes, os gols e o estádio onde ocorre a partida;
2. Fazer o relacionamento, consolidação e armazenar essas informações em um banco de dados;
3. Possibilitar a busca das informações das partidas realizadas usando uma interface web.

4.1 Premissas e Escopo

O trabalho parte da premissa que os textos jornalísticos de futebol sejam bem escritos, ou seja, que não tenham erros de ortografia e que a construção sintática não tenha ambiguidades que comprometam a análise de frases que contenham a relação entre equipes.

O futebol é um assunto vasto cujas notícias saem nos jornais impressos e eletrônicos, como troca de jogadores, contusões em treinos ou também sobre a alta cúpula. Esse trabalho vai ser restrito apenas aqueles textos que saem logo após a partida, com o resultado e os seus principais lances. Mesmo assim, não é esperado que o texto contenha as informações nos formatos exatos ou que não seja ambiguo, e não é esperaod que em todos os textos possam ser retiradas informações úteis.

Esse trabalho não tem o objetivo de criar um agente que procure informações nas páginas da internet, mas que apenas os textos sem nenhuma marcação de página (*tags* HTML ou metadados) estejam disponíveis para processamento.

Quanto às buscas efetuadas pelo usuário, tomamos como hipótese que o usuário tem interesse principal nas equipes, antes das cidades. Por outro lado, quando o mesmo digitar uma equipe, há um grande número das mesmas no campeonato paulista que apresentam o mesmo nome da cidade. Assim, nestes casos, decidimos por supor que o usuário deseja saber dados relativos à equipe.

A entrada buscada será bem limitada, apenas algumas categorias serão utilizadas, em sua maioria as equipes de futebol, ou, em raras ocasiões, cidades das partidas ou os estádios. Por isso, levamos em conta as técnicas de Inteligência Artificial, um agente reativo que não leva em conta buscas anteriores será utilizado, não havendo necessidade de armazenamento das buscas realizadas, visando otimizar o resultado.

Para o mecanismo de busca, o próprio servidor de testes do Django foi utilizado. Para um site em produção, os criadores do *framework* recomendam que seja utilizado um servidor que suporta alta carga de requisicóes, como o *Apache*. Como o objetivo de nosso trabalho não envolve publicar uma página web ao público, foi decidido usar o servidor do Django.

4.1.1 Diretrizes do Trabalho

Para que esse trabalho pudesse ser mantido e expandido, foram usadas as seguintes diretrizes:

- O trabalho foi divido em módulos que pudessem ser feitas modificações e expansões que não afetassem os outros módulos. Com isso, a construção do projeto foi divida em vários módulos, como a tokenizador, o refinamento lexical, analisador sintático, analisador semântico etc.
- Outra diretriz importante é o uso de software livre e conhecido pela comunidade (distribuídos sob a licença GPL ou BSD). Com esse benefício, qualquer software fornecedor a esse projeto pode ser modificado e devolvido à comunidade que o mantém com as alterações propostas.

Por esse motivo foi usado o NLTK, o *framework* para o desenvolvimento de aplicações que tratam linguagem natural que tem licença GPL, uma comunidade ativa, que possui um livro aberto para consulta pública na Internet.

4.2 Arquitetura

Nessa seção demonstraremos como o projeto foi dividido, mostrando a arquitetura dos seus componentes e os seus pacotes.

4.2.1 Arquitetura Geral

O sistema desenvolvido é divido em dois grande módulos: o módulo extrator de informações, cujo objetivo é extrair as informações do texto jornalístico e montar uma estrutura de dados, e o módulo de cusca de informações.

Figura 4.1: Arquitetura Geral do Sistema

Para intermediar esses dois grandes módulos, a estrutura de dados é gravada em um banco de dados, que é preenchido pelo Extrator e é consultado depois pelo Extrator e pelo Buscador.

4.2.2 Estrutura de Pacotes

O sistema pode ser divido nos seguintes pacotes:

- **Extrator de Informações:** contém os módulos de processamento de linguagem natural para extração de informações;
- **Buscador de Dados:** contém módulos de interface com o usuário via web e processamento de busca;
- **Django:** contém os módulos do *framework* web.

Essa seção trata os pacotes do Extrator e do Buscador que foram desenvolvidos no projeto.

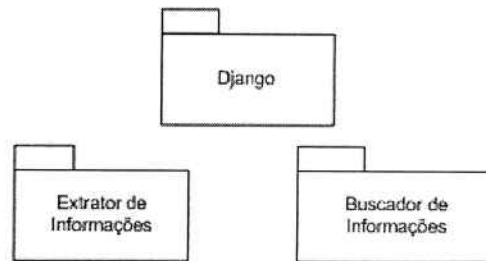

Figura 4.2: Estrutura Geral de Pacotes

Pacote Extrator de Informações

O pacote Extrator de Informações está divido nos módulos de processamento de linguagem natural. Ele contém os três módulos:

- **Analisador Léxico**: extração de *tokens* e etiquetação;
- **Analisador Sintático**: extração de construções sintáticas;
- **Analisador Semântico**: retirada de informações das construções sintáticas e contrução das estruturas de dados.

Pacote Buscador de Informações

O pacote Buscador de Informações, mais simples que o Extrator, tem dois módulos:

- **Agente**: agente reativo, processando a busca e obtendo resultados;
- **Processamento de resultado**: módulo de construção da página de resultados da busca.

4.3 Extrator de Informações

Aqui nesse texto há informações do subsistema extrator de informações cujo objetivo é criar um processamento de linguagem natural sobre um texto jornalístico e extrair informações sobre uma partida.

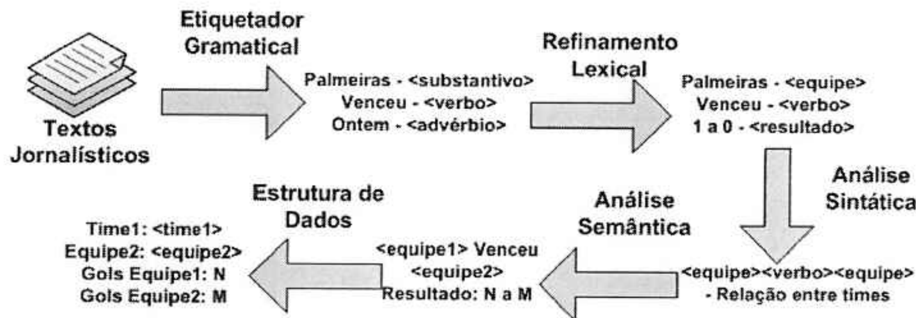

Figura 4.3: Arquitetura do Extrator de informações

4.3.1 Analisador Léxical

O analisador léxico do sistema quebra o texto em unidades, ou seja, palavras ou símbolos que contenham algum sentido são classificados como *tokens*, marcando-os com a sua função morfológica.

A primeira fase da análise léxica é uso do *tokenizer* e o etiquetador. Assim foi utilizado o *TreeTagger*, que retorna os *tokens* com as suas respectivas *tags*. Por exemplo, *"jogador"* será classificado como **"NOM"** (as outras *tags* são apresentadas no Anexo A). O *TreeTagger* contém um treinamento para língua portuguesa realizado por Pablo Gamallo da Universidade de Compostela na Espanha.

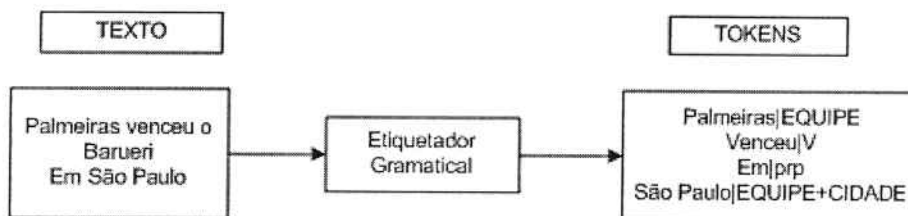

Figura 4.4: Etiquetador Gramatical

Assim, temos a saída conforme a figura a baixo:

```
reading parameters ...
tagging ...
O        DET     o
Palmeiras        NOM     palmeira
venceu   V       vencer
o        DET     o
Barueri  NOM     barueri
por      PRP     por
1        CARD    @card@
a        PRP     a
0.       CARD    @ord@
finished.
```

Figura 4.5: Saída do TreeTagger

Com o resulado da saída do *TreeTagger*, foi produzida na linguagem Python uma lista com as tuplas (*token*,*tag*) para cada *token* retirado do texto.

O próximo passo da análise léxica é o refinamento dos *tokens* usando um *Gazetteer* ou listas de palavras de um determinado conceito. No projeto, foram usadas três listas: lista de equipes, lista de cidades e lista de estádios. A maior dificuldade encontrada para o uso destas listas foi a existência de equipes e cidades com o mesmo nome. Para contornar esse problema foi nessária uma classificação especial.

Para o tema, foi necessário identificar as equipes de futebol, os estádios e as cidades onde os jogos ocorreram. Assim, foram criadas as seguintes *tags*:

Tag	**Significado**
EQUIPE	Equipe de Futebol
CIDADE	Cidade
ESTADIO	Estádio
EQUIPE-CIDADE	Equipe ou Cidade
DATA	Data

Tabela 4.1: *Tags* utilizadas para fazer o refinamento

A *tag* EQUIPE-CIDADE foi feita para as equipes de futebol que contêm os mesmos nomes que suas respectivas cidades; por exemplo, São Paulo e Barueri.

Assim, os conteúdos dessas listas são considerados palavras reservadas e o refinamento lexical marca esses *tokens* com a sua respectiva *tag*.

Para fazer tal refinamento foi preciso o uso de uma estrutura de dados chamada "pilha". A pilha é usada para o caso de palavras reservadas que são compostas. No momento que um *token*

é achado e o seu início combina com alguma palavra reservada, o *token* é empilhado e essa é operação é feita conforme os tokens são achados. Ao mesmo tempo, um *buffer* é preenchido, tornando-se a base de comparação com a palavra reservada.

Caso o *buffer* seja exatamente igual à palavra reservada, essa palavra é etiquetada com a *tag* correspondente, esvaziando-se e a pilha e o *buffer*. Caso não seja mais parte da palavra reservada, o *buffer* é esvaziado e ocorre o desempilhamento da pilha que devolve a lista de saída os *tokens* com as *tags* que foram etiquetadas inicialmente pelo *TreeTagger*.

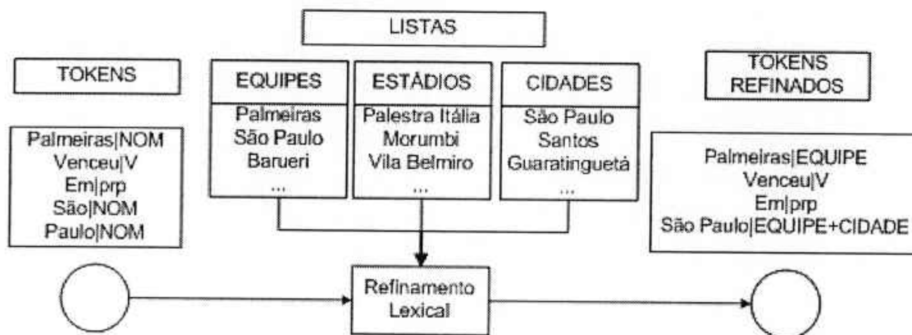

Figura 4.6: Refinamento Lexical

4.3.2 Analisador Sintático

A próxima fase do processamento de linguagem natural é a análise sintática que procura estruturas que contenham sequências de *tokens* definidas.

Após os *tokens* serem etiquetados pelo analisador léxico, o analisador sintático irá incorporá-los em estruturas de árvores, ou seja, a saída do analisador sintático é uma grande árvore com as classificações das estruturas.

O projeto não tem o objetivo de classificar todas as estruturas da língua portuguesa, mas apenas as estruturas que tem relacionamento com partidas de futebol.

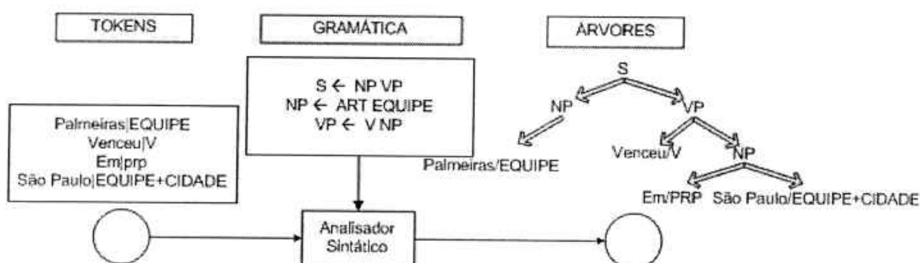

Figura 4.7: Funcionamento do Analisador Sintático

Para obter essa estrutura de árvores, é necessária a aplicação de uma gramática tendo, como símbolos terminais, as *tags* que representam os seus *tokens*.

Para a estrutura de frase com relacionamento entre equipes de futebol foi definifida para o projeto estrutura baseada nas equipes e nos verbos. Por exemplo: "Palmeiras venceu o Barueri por 2 a 0" seria equivamente a uma estrutura do tipo " EQUIPE VERBO ARTIGO EQUIPE PREPOSICAO RESULTADO ".

Com o objetivo de pegar frases simples como a acima foi desenvolvida a seguinte gramática:

<table>
<tr><th>Não-Terminal</th><th>Regra de Formação</th></tr>
<tr><td>RESULTADO</td><td><CARD><PRP|NOM|V><CARD></td></tr>
<tr><td>RESULTADO_EXPLICITO</td><td><EQUIPE|EQUIPE-CIDADE><RESULTADO>
<EQUIPE|EQUIPE-CIDADE></td></tr>
<tr><td>NP</td><td><PRP+P|PRP+DET|PRP|DET|ADJ>*?
<EQUIPE-CIDADE|EQUIPE|CIDADE|ESTADIO>+
<ADJ>?<PRP+P|PRP+DET|PRP>?<RESULTADO>?</td></tr>
<tr><td>VP</td><td><V>+<ADV>*<NP>*</td></tr>
<tr><td>NPREAL</td><td><NP><CONJ>*<NP>*</td></tr>
<tr><td>FUTEBOL</td><td><NPREAL>*<VP></td></tr>
</table>

Tabela 4.2: Regras de formação da árvore sintática

Com as regras da Tabela 4.2, é possível perceber que o não-terminal principal da árvore é FUTEBOL, pois contém uma frase simples referente ao resultado de uma partida de futebol relacionando uma equipe à outra.

Outras estruturas são mapeadas como RESULTADO que contém o resultado em gols da partida e a estrutura RESULTADO_EXPLICITO que além do resultado em gols, tem a relação das equipes e os seus gols na partida, o que facilita ainda mais a interpretação para a formação da estrutura de dados na análise semântica.

Para conseguir esse objetivo foi usado um recurso no NLTK, o ***RegexpParser***, em que o argumento de entrada é uma expressão regular com a gramática proposta (conforme a tabela gramatical anterior).

A saída dessa análise é uma estrutura de dados em formato de árvore sintática, no mesmo formato que uma estrutura de árvore representada na linguagem funcional LISP:

```
(S
  (FUTEBOL
    (NPREAL (NP Corinthians/EQUIPE))
    (VP perde/V (NP para/PRP Noroeste/EQUIPE)))
  e/CONJ
  (FUTEBOL (VP fica/V fora/V))
```

```
das/PRP+DET
semifinais/NOM
Do/NOM
UOL/NOM
Esporte/NOM
Em/NOM
(NPREAL (NP SãoPaulo/EQUIPE-CIDADE))
. . .
```

4.3.3 Analisador Semântico e Gerador da Estrutura

O analisador semântico transforma as árvores sintáticas em informação estruturada pela interpretação dessa estrutura. Para conseguir esse objetivo, o analisador semântico trata cada parte da estrutura das árvores e os *tokens* produzidos.

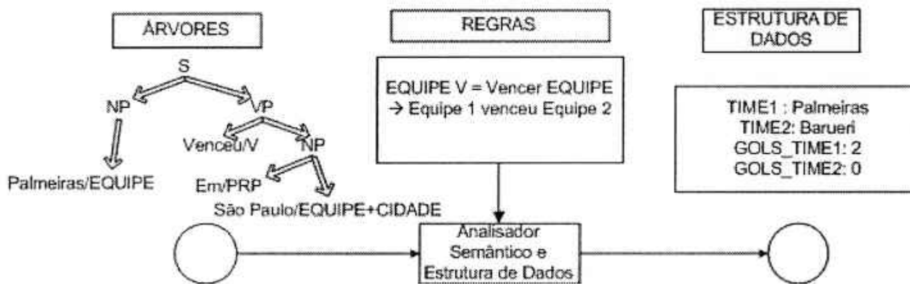

Figura 4.8: Analisador Semântico e Gerador da Estrutura

As árvore sintáticas são pecorridas usando o algoritmo de pré-ordem que pega o nó raiz da árvore e em seguida o nó da folha da esquerda e percorre todos os nós até o nó da folha da direita. Isso é importante pois, na língua portuguesa escrita, seguimos o padrão de escrever da esquerda para a direita, tanto na ordem de escrita das palavras quanto na ordem de escrita das frases.

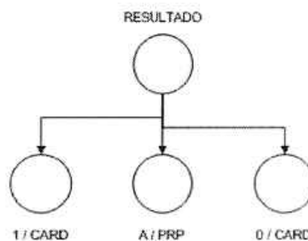

Figura 4.9: Exemplo de árvore de RESULTADO

O objetivo de verificar o nó da raiz é verificar qual estrutura é analisada. No caso da árvore

de RESULTADO podemos verificar facilmente o resultado do jogo, pois vem no formato "1 a 1", "1 x 1" ou "1 - 1".

No caso da estrutura sintática RESULTADO_EXPLICITO a vantangem é maior ainda, pois temos as equipes e o resultado em uma estrutura de fácil reconhecimento, como em "Palmeiras 1 X 0 Barueri". Com isso, podemos preencher a estrutura de dados com os dados imediatamente.

Para o caso que não ter estrutura, ao pecorrer as folhas das árvores, o analisador pega o *token* com a sua classificação e, ao achar uma equipe, preenche as equipes da estrutura de dados **JogoDeFutebol** a medida que é encontrado no texto. Esse método se mostrou eficiente, pois normalmente as equipes que jogaram a partida são as primeiras citadas no texto.

Com esse método pegamos as equipes, o resultado e o estádio onde ocorreu a partida. Normalmente, apenas com esse processamento, a maioria dos dados seria preenchida. Entretando, como a retirada apenas da primeira ocorrência de equipes está sujeita a muitos erros, como pegar equipes que não jogaram a partida, foram citadas no texto antes de alguma que realmente jogou a partida.

Na segunda parte do processamento semântico é feita uma análise estatística que utiliza as quantidade de citações no texto.

Para garantir que essas equipes selecionadas são as corretas, o analisador semântico guarda qualquer referência de equipe encontrada. No final, ele conta as referências achadas de equipes e compara com as encontradas pelo primeiro processamento. Confirmadas as referências, são gravadas na estrutura de dados; caso não, são trocadas pelas referências mais achadas. Assim, temos um contator de referências às equipes.

Todas essas informações são gravadas em uma estrutura de dados que chamamos de **JogoDeFutebol**, que contém o nome das duas equipes, os gols das equipes no jogo, o estádio e a data do jogo. Podemos ver isso no código abaixo:

```
jogoDeFutebol = {'equipe1':'',
                 'equipe2':'',
                 'gols1':0,
                 'gols2':0,
                 'estadio':'',
                 'data':''}
```

Assim, na primeira parte da conferência, é verificado se as duas equipes mais referenciadas são as equipes que foram encontradas na busca inicial. Parte-se do princípio que as duas equi-

pes que jogaram a partida são as mais referênciadas no texto, pois outras equipes também são referenciadas por causa de jogos anteriores ou futuros.

Podemos ver isso no algoritmo abaixo:

```
if self.jogoDeFutebol['equipe1'] == equipePrimeira
         and self.jogoDeFutebol['equipe2'] <> equipeSegunda:
   self.jogoDeFutebol['equipe2'] = equipeSegunda
elif self.jogoDeFutebol['equipe1'] <> equipePrimeira
         and self.jogoDeFutebol['equipe2'] == equipeSegunda:
   self.jogoDeFutebol['equipe1'] = equipePrimeira
elif self.jogoDeFutebol['equipe1'] == equipeSegunda
         and self.jogoDeFutebol['equipe2'] <> equipePrimeira:
   self.jogoDeFutebol['equipe2'] = equipePrimeira
elif self.jogoDeFutebol['equipe1'] <> equipeSegunda
         and self.jogoDeFutebol['equipe2'] == equipePrimeira:
   self.jogoDeFutebol['equipe1'] = equipeSegunda
elif self.jogoDeFutebol['equipe1'] <> equipeSegunda
         and self.jogoDeFutebol['equipe2'] <> equipePrimeira
         and self.jogoDeFutebol['equipe1'] <> equipePrimeira
         and self.jogoDeFutebol['equipe2'] <> equipeSegunda:
   self.jogoDeFutebol['equipe1'] = equipePrimeira
   self.jogoDeFutebol['equipe2'] = equipeSegunda
```

Caso a *equipe1* da estrutura de dados JogoDeFutebol seja igual a equipe mais referenciada, verifica se a *equipe2* é igual a segunda mais referenciada. Caso não seja, há a troca. Caso a *equipe1* é igual a segunda mais referenciada, verifica se a *equipe2* é igual a primeira mais referenciada. Esse algoritmo é analogamente aplicado, executando primeiro a verificação da *equipe2*. Esse princípio é muito importante, pois são verificadas as equipes pela quantidade de vezes que elas aparecem no texto.

O mais importante é o caso de nenhuma das equipes forem escolhidas entre as mais referenciadas, pois, pelo nosso algoritmo, elas são totalmente descartadas e trocadas para as mais referenciadas. A partir disso, a análise semântica vai para a próxima fase, que é a procura de relações entre as equipes pela busca de construções sintáticas de frases.

Levando em consideracão que o analisador semântico ache uma árvore que contenha a estrutura FUTEBOL, ele verifica essa árvore pela unidade do verbo. Caso o verbo seja relacionado

aos verbos *empatar*, *vencer* ou *derrotar*, o analisador vai procurar no sintagma nominal as duas equipes e armazenar em outra estrutura, que é chamamada de **relacao**.

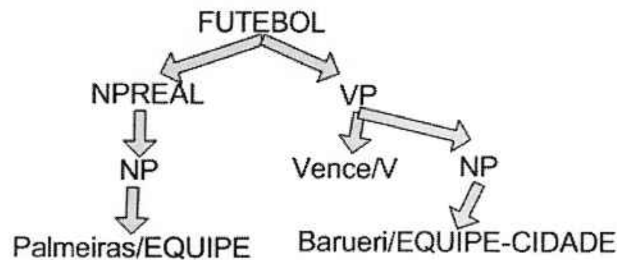

Figura 4.10: Exemplo de Árvore de FUTEBOL

Código da estrutura de relação entre equipes:

```
relacao = {'equipe1':'',
           'relacao':'',
           'equipe2':''}
```

Essa estrutura é utilizada para corrigir eventuais erros que são preenchidos na estrutura **JogoDeFutebol**, pois a estrutura **JogoDeFutebol** pode ser preenchida por equipes erradas pelos dois processamento citados anteriormente, ou o resultado pode estar com valor trocado; ou seja, a equipe vitoriosa pode ser marcada com os gols da equipe derrotada.

Encontrada uma relação entre as equipes, é verificado se as equipes estão corretas, ou seja, se as equipes de **JogoDeFutebol** são as mesmas de **Relacao**. Caso não estejam, faz-se a troca. Podemos ver isso no algoritmo abaixo:

```
if len(self.relacao['relacao']) > 0:
  if self.jogoDeFutebol['equipe1'] == self.relacao['equipe1']:
    if self.jogoDeFutebol['equipe2'] <> self.relacao['equipe2']
          and len(self.relacao['equipe2']) > 0:
      self.jogoDeFutebol['equipe2'] = self.relacao['equipe2']
  elif self.jogoDeFutebol['equipe2'] == self.relacao['equipe2']:
    if self.jogoDeFutebol['equipe1'] <> self.relacao['equipe1']
          and len(self.relacao['equipe1']) > 0:
      self.jogoDeFutebol['equipe1'] = self.relacao['equipe1']
  elif len(self.relacao['equipe2']) >0 and len(self.relacao['equipe1'])>0:
     self.jogoDeFutebol['equipe1'] = self.relacao['equipe1']
     self.jogoDeFutebol['equipe2'] = self.relacao['equipe2']
```

Nesse código é verificado se a *equipe1* da estrutrura **JogoDeFutebol** é igual a *equipe1* da estrutura **relacao**; verifica-se também se a *equipe2* é diferente nas duas estruturas. Caso seja, faz-se a troca. Assim, o acerto do extrator é incrementado ao descobrir as equipes corretas. Analogamente, é feito com a *equipe2* e, caso as equipes da estrutrura **JogoDeFutebol** forem totalmente diferentes de **relacao** e **relacao** tiver as duas equipes preenchidas, então essas duas equipes têm prioridades de escolha, sendo substituídas em **JogoDeFutebol**.

Para verificar se os gols atribuídos às equipes forem corretos (a equipe vitoriosa fez mais gols e a derrotada, menos) é verificada a relação e a quantidade de gols atribuída à equipe. Assim, temos o código:

```
if self.relacao['relacao'] == 'venceu' or self.relacao['relacao'] == 'vence'
            or self.relacao['relacao'] == 'goleia'
            or self.relacao['relacao'] == 'derruba':
    if self.relacao['equipe1'] == self.jogoDeFutebol['equipe1']:
      if int(self.jogoDeFutebol['gols1']) < int(self.jogoDeFutebol['gols2']):
         temp = self.jogoDeFutebol['gols1']
         self.jogoDeFutebol['gols1'] = self.jogoDeFutebol['gols2']
         self.jogoDeFutebol['gols2'] = temp
elif self.relacao['equipe1'] == self.jogoDeFutebol['equipe2']:
        temp = self.jogoDeFutebol['gols1']
        self.jogoDeFutebol['gols1'] = self.jogoDeFutebol['gols2']
        self.jogoDeFutebol['gols2'] = temp
```

No caso de vitória, se a *equipe1* que venceu (ou seja, a relação são verbos de vitória) tem menos gols, eles são invertidos entre as equipes e, se foram invertidos os nomes das equipes na estrutura de dados **JogoDeFutebol** e **relacao**, os gols também serão invertidos. Analogamente é feito dessa maneira caso haja derrota da equipe1.

No caso de empate entre as equipes se os gols não forem iguais, é atribuído o valor '0' aos gols de ambas as equipes, o que pelo menos ameniza o erro de uma empate com uma equipe com mais gols. Essa forma de análise mostrou que o número de erros que o analisador semântico gerou diminuiu em relação à busca apenas estatística.

Assim, conseguimos fazer o objetivo do Extrator, que é a extracão dos dados mais importantes de uma partida e futebol; que são as equipes, o resultado e as partidas.

4.4 Estrutura de Dados - Persistência

Para modelar a persistência de dados, foi feito o seguinte Diagrama Entidade-Relacionamento:

- **Equipe:** equipes que participaram do campeonato. É preenchida na criação do sistema.
- **Cidade:** cidades onde ocorrem os jogos. É preenchida na criação do sistema.
- **Estadio:** estádio onde ocorrem os jogos que são ligados às cidades. É preenchido na criação do sistema.
- **Partida:** contém as partidas do campeonato com as equipes e o resultado do jogo. É preenchida pelo Extrator de Informações.

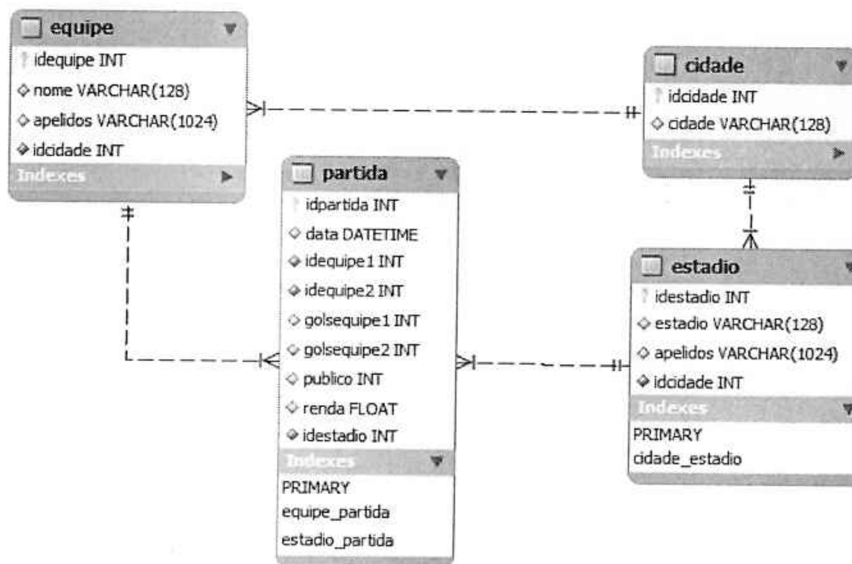

Figura 4.11: Modelo Entidade-Relacionamento

As tabelas Equipe, Cidade e Estadio são utilizadas pelos Extrator de Informações para uso de refinamento lexical, pois essas tabelas já estão preenchidas com os dados do Campeonato Paulista de 2008. Elas também são utilizadas pelo Buscador que usa também o analisador lexical para classificação dos dados de entrada do usuário.

A tabela Partida, porém, contém as partidas do campeonado que são o resultado da extração de dados feita pelo Extrator dos textos. Nela, nenhum dado é obrigatório, pois o extrator por não achar uma equipe que participou da partida ou o estádio, por exemplo. Essa tabela também é usada na consulta de informações, que é feita pelo Buscador, a partir das relações lógicas que ele faz com os dados das outras três tabelas.

Nos dois módulos foi usado o mapeamento objeto-relacional que é disponível no *framework* Django. Dessa forma, as aplicações têm uma camada de modelo de dados,que torna as funções de consulta e criação simples, apenas usando programação em linguagem Python.

4.5 Busca de Informações

O módulo de busca, conforme apresentado anteriormente, é o responsável pelo processo da busca do usuário, devolvendo as partidas mais příximas das partidas procuradas.

Diferentemente do Extrator, este módulo está mais próximo do usuário final, existindo uma grande interação com o *framework* Django para a obtećão da entrada de busca e a exibição do resultado.

4.5.1 Interface com usuário

Com o servidor iniciado, o usuário poderá acessar a página inicial do sistema com uma caixa de texto em um *form* com o método *post*.

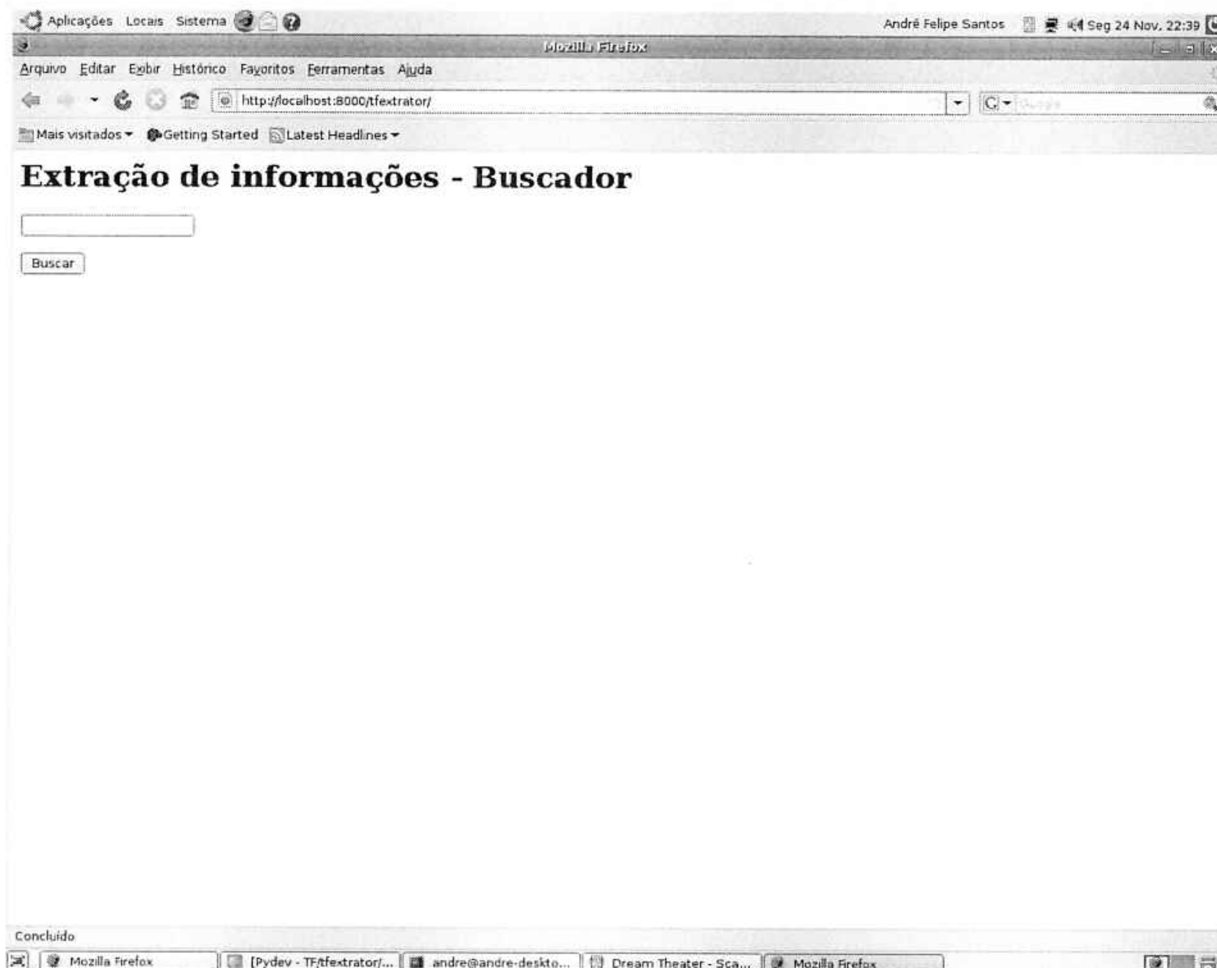

Figura 4.12: Página inicial da busca

Este post chama uma ação 'busca', que é reconhecida pelos *URLs* do servidor, chamando o método 'busca' na classe *views.py*, que é a classe padrão do Django para tratar requisições

HTTP. Este método instancia o agente reativo, alimentando-o com o conteúdo da caixa de texto preenchida pelo usuário.

4.5.2 Agente reativo

Ao ser instanciado, o agente tem como atributo a *string* escrita pelo usuário. Com base no princípio do reuso de código, o módulo de análise lexical é reutilizado sobre essa entrada, sendo feitas as mesmas operaćões em um texto jornalístico.

Por outro lado, a lista de resultados da análise lexical oferece algumas complicações para as buscas. Com base nas premissas apresentadas, foi feita uma simplificação, substituindo a classificação “EQUIPE-CIDADE” por “EQUIPE”, que ocorre pela ambiguidade apresentada por diversas equipes que têm como nome a cidade de origem, como São Paulo, São Caetano, Rio Claro etc. O código utilizado para fazer a simplificação é bem simples:

```
for i in range(len(lista)):
    if lista[i][1] == "EQUIPE-CIDADE":
        lista[i] = (lista[i][0], "EQUIPE")
return lista
```

A lista simplificada é separada em listas de equipes, cidade e estádio. A necessidade dessa separação é para a aplicaćão da lógica do agente reativo. Ele simplesmente adiciona em uma nova lista cada *token* cujo tipo seja idêntico ao solicitado.

```
resultado = []
for i in lista:
    if i[1] == tipo:
        resultado.append(i[0])
return resultado
```

Partindo da premissa que os usuários do sistema entrarão com termos facilmente identificados, ou seja, dificilmente fugirão do escopo de *estádio*, *equipe* ou *cidade*; foram decididos oito casos possíveis de busca do usuário:

- Busca apenas por equipes;
- Busca apenas por estádios;

- Busca apenas por cidades;
- Busca por equipes e estádios;
- Busca por equipes e cidades;
- Busca por estádios e cidades;
- Busca pelos três tipos de termos;
- Busca inválida: nenhum termo foi reconhecido como válido.

Para o caso da busca inválida, o sistema retorna uma mensagem avisando o usuário que devem ser colocados os termos *equipes*, *cidades* ou *estádios*.

Com relação às buscas válidas, o agente faz uso da API de acesso ao banco do Django, buscando as partidas conforme as listas dos tipos de termos. Para as buscas que envolvem equipes, havendo duas ou mais, é montada uma lista com pares das equipes, para que sejam recuperadas partidas do banco de dados que envolvam cada um desses pares de equipe. No caso de apenas uma equipe ser detectada, são recuperadas todas as partidas que ela participa.

Quanto às buscas que envolvem estádios e cidades são recuperadas as partidas que cada estádio ou cidade mencionados participa. O único ponto a ser mencionado é que, se o estádio e cidade possuírem nomes iguais, os jogos que cumprirem esse critério devem ser relacionados primeiro no resultado.

Caso a busca por um conjunto de equipes, estádios ou cidades haja um pareamento dos times, busca-se as partidas que contenham os times e tenham ocorrido nos estádios ou cidades solicitados. Caso não haja correspondência, faz-se somente a busca por equipes.

O estudo da busca feita pelo usuário consiste em analisar as listas resultantes da análise lexical, simplificação e separação. Verifica-se o comprimento das listas separadas: caso sejam diferentes de zero, é um indicador que aquele tipo foi solicitado pelo usuário. Para o melhor entendimento do texto anterior, é apresentado um pseudocódigo no trecho a seguir:

```
se (tamanho da lista de equipes != 0 e
            o das listas de estádios e cidades == 0)
    buscar partidas por pares de equipes
senão
    se (tamanho da lista de estádios != 0 e
            o das listas de equipes e cidades == 0)
```

```
        buscar partidas por estádio
senão
    se (tamanho da lista de cidades != 0 e
            o das listas de equipes e estádios == 0)
        buscar partidas por cidades
senão
    se (tamanho da lista de equipes e estádios !=0 e
            o da lista de cidades == 0)
        buscar partidas por pares de equipes e por estádios
senão
    se (tamanho da lista de equipes e cidades !=0 e
            o da lista de estádios == 0)
        buscar partidas por pares de equipes e por cidades
senão
    se (tamanho da lista de estádios e cidades != 0 e
           o da lista de equipes == 0)
        buscar partidas por estádios e cidades
senão
    se (tamanho da lista de equipes, estádios e cidades !=0)
        buscar partidas por equipes, estádios e cidades
senão
    emitir mensagem de busca inválida
```

No caso de buscas que utilizam os três tipos distintos de termos, a priorização dos resultados é feita da seguinte forma:

- Partidas envolvendo as equipes solicitadas, no estádio que tenham o m esmo nome com cidade apresentada;
- Partidas envolvendo as equipes solicitadas, no estádio, independente da cidade;
- Partidas envolvendo as equipes solicitadas, independente do estádio e da cidade.

A partir desta análise feita pelo agente são recuperadas partidas, ordenadas conforme a explicacáo anterior. Esta lista ordenada de partidas é enviada para o módulo de processamento para a construção da página de resultado.

4.5.3 Módulo de processamento

De posse da lista de partidas recuperadas pelo agente, o módulo de processamento acessa cada item dela, montando uma página *HTML* que contém uma tabela indicando as datas, equipes, gols, público e renda do estádio. Essa página é novamente enviada para a classe *views.py*, que envia para o browser do usuário a página de resultado, ainda chama o método 'busca'.

A construção desta página começa pela construção das partes de *Head* e *Title*, além do início da tabela, indicando os tipos de dados a serem apresentados. A seguir, itera-se por cada item da lista, adicionando uma nova linha na tabela, colocando os dados de partida de acordo com o que foi indicado antes do começo da iteração. Com o fim, a tabela é fechada, adicionando-se um *link* para voltar à pagina inicial do mecanismo de busca.

5 *Testes e Resultados*

Nessa seção mostraremos os testes e os resultados da implementação do projeto usando textos reais de sites da Internet e comparando-os com os dados obtidos.

5.1 Extrator de Informações

5.1.1 Objetivos do Teste

O objetivo do Extrator é retirar três grupos de informações: quais equipes jogaram, qual foi o resultado do jogo em gols e em qual estádio foi o jogo. Além dessas informações explícitas, o Extrator deve procurar a relação implícita das equipes; ou seja, qual equipe foi vitoriosa, se houve empate ou qual foi a derrotada.

Para isso, foram usados quinze textos sobre partidas futebol, sendo esses textos inseridos no formato de texto puro, ou seja, sem formatação de estilo, *tags* HTML ou imagens para não interferir no processamento de linguagem natural, colocando metadados que não são tratados.

Esses textos foram retirados de páginas do site UOL (Universo OnLine) sobre o Campeonato Paulista de 2008. Eles são escritos por vários jornalistas diferentes e os textos tratam principalmente de jogos das últimas rodadas do Campeonato. Eles foram escolhidos porque contém mais informações; como mais citações às outras equipes que não jogaram a partida e mais relações entre as equipes do que os textos das primeiras rodadas do Campeonado.

O processo de teste se baseia na retirada das informações, tanto por um leitor humano quanto pelo Extrator. O objetivo desse teste é comparar essas duas informações, partindo do princípio que a informação compreendida pelo ser humano é a correta.

Outro foco da análise é comparar se o resultado obtido pelo Extrator condiz com a relação encontrada; ou seja, se uma equipe venceu, logo ela deve estar no resultado com mais gols.

Por fim, é feito um quadro comparativo com a taxa de acerto para os dados retirados nesses quinze textos (A tabela completa pode ser achada no Anexo B).

5.1.2 Testes e Resultado

Como exemplo dos testes, foram colocados dois resultados dos textos analisados, podendo ver o resultado extraído por um ser humano:

Teste	Equipe 1	Equipe 2	Gols 1	Gols 2	Estádio
1	Palmeiras	Barueri	3	0	Arena Barueri
6	Corinthians	Noroeste	2	3	Alfredo de Castilho

Tabela 5.1: Resultado da partida retirada por um Humano

Com a análise do Extrator, temos o seguinte resultado para esses dois textos:

Teste	Equipe 1	Equipe 2	Gols 1	Gols 2	Estádio
1	Palmeiras	Barueri	3	0	Arena Barueri
6	Corinthians	Noroeste	2	3	Alfredo de Castilho

Tabela 5.2: Resultado da partida retirada pelo Extrator

É possível ver que, nesses testes, os resultados foram bem sucedidos, pois os dados foram exatamente os mesmos dos dados retirados pelo humano. No Anexo B, há uma tabela com as informações dos quinze textos.

Não foram todos os testes que tiveram resultados tão completos e satisfatórios como os apresentados acima. Na partida do teste 9, que foi entre Juventus e Rio Preto, o texto se mostrou ambiguo até mesmo ao leitor humano, pois houve muita dificuldade para saber qual foi o resultado do jogo e o Extrator não conseguiu retirar essa informação. Mas a informação da estrutura que o Juventus perdeu a partida foi o diferencial, pois sabemos o resultado do jogo mesmo sem os gols.

Em outros casos, apenas uma das equipes foi detectada, pois havia poucas referencias às equipes que jogaram e a relação encontrada apenas mostrava qual a equipe foi vencedora.

Com isso, temos a seguinte tabela com os resultados dos testes:

Grupo de teste	Acerto
Acertou as 2 equipes que jogaram	86.67%
Acertou pelo menos 1 equipe que jogou	100.00%
Acertou o resultado da partida	93.33%
Acertou o estádio	88.89%
Acertou a relação	66.67%

Tabela 5.3: Resultados dos testes

A tabela acima mostra que as técnicas usadas foram bem sucedidas, pois acertou sempre em pelo menos uma das equipes que jogou a partida e acertou em 86.67% as duas. Outro fato

interessante é o acerto do resultado do jogo junto ao acerto de qual time venceu. Nesse caso, o acerto potencializado pela extração estatística das equipes.

Entretanto, o acerto de quem ganhou ou quem perdeu foi feito principalmente pelo acerto da relação entre as equipes. Em 66.67% dos casos o Extrator descobriu quem ganhou a partida ou quem perdeu. Isso é muito importante, pois essa informação é uma relação efetiva entre as equipes que jogaram e podemos fazer correções dos dados extraídos anteriormente pelo método de procura e estatístico, ocorrido porque, na maioria do texto, temos estruturas do tipo "*Equipe 1 vence Equipe 2*"..

Mas podemos ver que os resultados não foram bons para empates. Isso foi observado porque nenhuma relação foi achada entre as equipes que foram citadas no texto. Foi observado no textos em que há empate da partida que não há estruturas comuns devido ao uso de gírias, críticas ou outros termos à uma equipe por ter cedido o empate.

Apesar desses problemas citados, podemos ver o Extrator é bastante confiável pois retornou o mesmo dado observado pelo ser humano na maioria dos casos.

5.2 Buscador de Informações

O objetivo do buscador é, a partir de uma entrada do usuário, identificar os tipos de termos desejados e, a partir dos mesmos, obter os dados das partidas que correspondem à entrada, sendo exibidos em uma página web.

Para os testes, são utilizados sete grupos de entradas que correspondam aos usos típicos do sistema por um usuário que queira saber informações a respeito de partidas. é desejável que cada uma das possibilidades de busca apresentadas na seção de implementação seja utilizada.

Nestes grupos de entrada estão presentes entradas comuns, como se espera de um usuário médio solicitando dois times, um estádio ou cidade. Foram colocadas também entradas mais complexas, principalmente que envolvam ambiguidades, ou uma quantidade maior de termos a serem localizados.

Os resultados esperados são páginas *HTML* com as partidas entre as equipes na entrada (se houver), com estádios e cidades; ordenadas conforme o critério apresentado na seção de implementação. Havendo equipes, os confrontos em estádios e cidades especificados devem aparecer primeiro, seguidos dos confrontos onde não haja essa correspondência.

6 *Considerações Finais*

O trabalho apresentou a extração e busca de informações de textos jornalísticos sobre futebol usando técnicas de processamento de linguagem natural. O objetivo foi a criação de agentes que possam extrair informações de textos e fazer buscas com eficiência e precisão de informações.

Ao observar a quantidade de informação que um texto informativo carrega, a importância desse tipo de processamento é mostrada, pois os seres humanos usam a linguagem natural para comunicação e um dos principais meios é o texto. Por esse meio, pode-se retirar informações que são objetivos do agente desse projeto.

Na parte de implementação foi exposta a forma de construção dos agentes. No Extrator foi exposta todas as etapas do processamento de linguagem natural, mostrando como foi feito o analisador léxico, sintático e semântico, saindo no final uma estrutura de dados com os resultados das partidas de futebol.

Como apresentado na seção Resultados e Testes, foi possível verificar que o Extrator teve resultados satisfatórios, pois em boa parte dos textos foi possível retirar o mínimo de informações para saber o que aconteceu na partida de futebol. O mais importante foi que a taxa de acertos foi alta, mostrando que o Extrator de dados é confiável.

Quanto à busca de informações, esse projeto apresentou uma implementação de busca de informações. Foi aproveitado o agente reativo, uma técnica simples, que se aproveita de uma abstracão dos casos possíveis, mostrando-se eficiente para este uso.

A arquitetura utilizada pode ser facilmente modificada para outros domínios de textos por um conjunto limitado de termos de interesse. Para um conjunto maior, a mesma abordagem pode ser utilizada, mas a complexidade envolvida é sensivelmente maior.

Para a melhoria do projeto, podem ser utilizadas técnicas mais avançadas de processamento de linguagem natural para obter mais informações e informações mais precisas.

A grande vantagem mostrada nesse projeto foi a sua divisão em módulos que torna possível

mudanças pontuais e expansões do projeto. Apesar do foco ser em partidas de futebol, o projeto pode ser modificado facilmente para tratar outros temas como voleibol ou retirar informações sobre o mercado financeiro.

Apesar dos problemas do projeto e de acurácia dos agentes, eles se mostraram grandes utilitários para extração de informação não-estruturada, que é muito comum em documentos ou em páginas da Internet.

6.1 Contribuições

Nesse projeto foram desenvolvidas técnicas de processamento de linguagem natural e retirada de informações de textos.

As principais contribuições desse trabalho foram na criação de agentes que trabalham com processamento de linguagem natural, tanto para a busca quanto para a extração de dados.

A especificação e projeto do Extrator e de todas as fases do processamento de linguagem natural mostram a importância do contexto nessa extração.

Por fim, o uso de ferramentas de código livre e amplamente conhecidas pela comunidade como Python, NLTK, PostgreSQL e Django possibilita a continuação do projeto e as sua expansão.

6.2 Dificuldades no projeto

No projeto ouve muita dificuldade de definir o escopo. Incialmente foi pensada a idéia de serem retiradas muitas informações sobre os jogos como jogadores, jogadas da partida, renda, entre outros. Mas, devido à complexidade do trabalho e a falta de conhecimento sobre o projeto, decidiu-se reduzir o escopo apenas às informações básicas da partida.

Outro problema de escopo foi o interesse da busca em apresentar adaptavidade, principalmente para aprender apelidos de equipes e jogadores. Infelizmente, a complexidade envolvida para isso tornou esse desejo inviável para a implementação do projeto.

Também não foi tratado no projeto a questão do tratamento de cidades onde ocorreram as partidas. A complexidade desse tratamento se mostrou grande, pois gerava muita confusão entre as cidades e as equipes que tinham o mesmo nome da cidade. Por isso, foi decidido remover esse tratamento do projeto.

Durante a análise, outro problema encontrado foi o aprendizado teórico sobre o assunto de processamento de linguagem natural, pois muito do material encontrado era assunto avançado ou pouco relavante ao projeto. Era necessário solidificar os conceitos e isso foi conseguido com o livro do NLTK.

O *framework* também foi outro grande empecilho ao projeto. Inicialmente foi proposto o uso do *framework* GATE (General Architecture for Text Engineering) feito em Java. Mas tal *framework* se mostrou muito complexo ao projeto por tratar várias etapas do processamento de linguagem natural e ter pouco suporte à língua portuguesa; logo, a linguagem de programação

Java se mostrou pouco atrativa. Assim, decidiu-se usar o NLTK e Python, que agilizaram muito o processo. O etiquetador morfológico também foi uma dificuldade grande no projeto, pois é muito raro encontrar tal material pronto sobre o assunto em língua portuguesa.

Durante a implementação foram necessários vários refinamentos ao analisadores das ferramentas, pois eles tinham o resultado muito abaixo do esperado. O analisador que precisou maior tratamento foi o léxico devido às limitações do etiquetador.

Quanto ao mecanismo de busca, houve uma dificuldade por decidir qual modelo utilizar. Na referência usada há um grande número de técnicas. O modelo escolhido (agente reativo) apresentou a adequação necessária e complexidade aceitável à implementação. A dificuldade nesta etapa foi o levantamento de técnicas por sua extensão.

O *framework* Django, mesmo sendo bem documentado e de código aberto, apresentou alguma dificuldade para o uso de sua API de persistência para módulos que não estejam diretamente envolvidos com páginas web.

6.3 Trabalhos Futuros

O trabalho de extração e busca de informações tem muito o que ser extendido e melhorado, apresentamos nesta seção algumas sugestões de por onde pode-se prosseguir.

6.3.1 Extensão

Das extensões possíveis ao sistema, podemos começar pelo aumento do domínio das informações extraídas. No presente momento, obtem-se as equipes que jogaram, o resultado da partida e estádio onde ela aconteceu. Pode-se acrescentar a capacidade de identificar jogadores participantes, lances de importânca tais como faltas, cartões, impedimentos.

Pode-se associar o sistema a um sistema de *web crawling*, para a obtenção automática de textos, aumentando a gama de dados disponíveis à extração, podendo não só obter dados sobre campeonatos presentes, como de campeonatos anteriores, mantendo um banco de dados histórico e crescente sobre partidas de futebol.

Quanto à extração de textos em outras línguas, como inglês e espanhol, é necessário refazer a lista de termos relevantes que orientam a extração. É também necessário refazer a parte de programação orientada à gramática em função da língua do texto de onde se quer extrair as informações. O mesmo procedimento pode ser utilizado para se mudar o domínio das

informações, como notícias a respeito de economia, ou saúde pública.

6.3.2 Adaptatividade

Apesar de poder ser considerada uma extensão do trabalho, acreditamos que pelas possibilidades, vale a pena destacar a importância da adaptatividade. O projeto permite muitas melhorias com o uso destas técnicas.

Do ponto de vista da extração, podemos aumentar a quantidade de termos que o sistema considera relevantes para a obtenção de dados da partida, como apelidos das equipes e estádios, podendo inclusive localizar cada vez mais termos relevantes às partidas, em função da frequência com que os mesmos aparecem nos textos.

Quanto à busca, o agente reativo também permite a inclusão de adaptatividade pela maneira como foi implementado. Através de técnicas de programação reflexiva, que Python permite, pode-se incluir novos casos de busca em função de novas entradas que o usuário coloque, como a busca por participações de jogadores, jogos que tenham tido certo árbritro entre outras possibilidades.

6.3.3 Desempenho

Para tornar o sistema mais viável a ser disponibilizado ao público, é necessário que haja uma pesquisa maior sobre o etiquetador. O POS Tagger, apesar dos resultados interessantes, não apresenta um bom desempenho para ser utilizado em larga escala. Um texto de 6 parágrafos leva em torno de 5 segundos para ser plenamente etiquetado. Considerando que é utilizado também no processo de busca, isso torna a solução atual inviável para o grande público.

O ponto do etiquetador é um dos grandes gargalos do sistema, mas o trabalho todo foi orientado à obtenção de resultados, nem tanto quanto ao desempenho. Para melhorá-lo de maneira geral, pode-se pesquisar outras técnicas que permitam o processamento linguístico mais rapidamente. Do ponto de vista da implementação como um todo, pode-se considerar a recodificação em Java ou C, ou mesmo considerar a utilização de compilação JIT do sistema através do Psyco.

É necessário também adaptar o sistema para que seja executado com sucesso por um servidor voltado a ambientes de produção, como o Apache.

Referências Bibliográficas

ANSI-SQL92/99. Disponível em: <http://www.ncb.ernet.in/education/modules/dbms/SQL99/ansi-iso-9075-5-1999.pdf>.

BIRD, S.; LOPER, E. Nltk: The natural language toolkit. In: *The Companion Volume to the Proceedings of 42st Annual Meeting of the Association for Computational Linguistics.* Barcelona, Spain: Association for Computational Linguistics, 2004. p. 214–217.

BIRD, S.; LOPER, E.; KLEIN, E. *Natural Language Processing — Analyzing Text with Python and the Natural Language Toolkit.* [S.l.: s.n.], 2008.

CHOMSKY, N. On certain formal properties of grammars. 1959.

DJANGO - The Web framework for perfectionists with deadlines. Disponível em: <http://www.djangoproject.com/>.

ECLIPSE.ORG home. Disponível em: <http://www.eclipse.org>.

GAMALLO, P. *Tree-Tagger*. Disponível em: <http://gramatica.usc.es/ gamallo/tagger.htm>.

GLADE User Interface Builder. Disponível em: <http://glade.gnome.org/>.

GOOGLE Code - Project Hosting. Disponível em: <http://code.google.com/hosting/>.

HOLOVATY, A. *Guido likes Django*. 2006. Disponível em: <http://www.djangoproject.com/weblog/2006/aug/07/guidointerview/>.

LEVENE, M. *An Introduction to Search Engines and Web Navigation.* [S.l.]: Pearson Education, 2005.

NATURAL Language Toolkit. Disponível em: <http://nltk.sf.net>.

NORVIG, P.; RUSSEL, S. *Artificial Intelligence: A Modern Approach.* [S.l.]: Prentice Hall, 2002.

PGADMIN III: PostgreSQL administration and management tools. Disponível em: <http://www.pgadmin.org/>.

POSTGRESQL: The world's most advanced open source database. Disponível em: <http://www.postgresql.org/>.

PYDEV - Python Development Extension. Disponível em: <http://pydev.sourceforge.net>.

PYTHON Programming Language – Official Website. novembro 2008. Online. Disponível em: <http://www.python.org>.

SCHMID, H. Improvements in part-of-speech tagging with an application to german.

SCHMID, H. Probabilistic part-of-speech tagging using decision trees.

SUBCLIPSE. Disponível em: <http://subclipse.tigris.org/>.

SUBVERSION. Disponível em: <http://subversion.tigris.org/>.

THE GTK+ Project. Disponível em: <http://www.gtk.org/>.

TREETAGGER - a language independent part-of-speech tagger. Disponível em: <http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html>.

ANEXO A – Conjunto de Tags

Tabela A.1: Relação da unidade morfológica com a *Tag* empregada pelo etiquetador

Tipo de *token*	***Tag***
Adjetivo	ADJ
Advérbio	ADV
Determinante	DET
Número Cardinal / Ordinal	CARD
Nome Comum / Próprio	NOM
Pronome	P
Preposição	PRP
Verbo	V
Interjeição	I
Vírgula	VIRG
Separadores de orações	SENT

ANEXO B – Resultado dos testes com o Extrator

Tabela B.1: Resultado da partida retirada por um Humano

Teste	Equipe 1	Equipe 2	Gols 1	Gols 2	Estádio
1	Palmeiras	Barueri	3	0	Arena Barueri
2	Guaratinguetá	Ituano	2	1	–
3	Palmeiras	Ponte Preta	1	0	Moisés Lucareli
4	Portuguesa	Sertãozinho	1	0	Frederico Dalmazo
5	Guarani	Rio Preto	2	0	Brinco de Ouro
6	Corinthians	Noroeste	2	3	Alfredo de Castilho
7	Rio Claro	Ponte Preta	1	1	–
8	Guaratinguetá	Mirassol	3	1	–
9	Juventus	Rio Preto	0	1	–
10	Portuguesa	Rio Preto	3	3	–
11	Palmeiras	Juventus	4	0	Santa Cruz
12	Corinthians	Bragantino	1	1	Morumbi
13	Marília	São Paulo	3	2	Bento de Abreu
14	Santos	Rio Preto	1	2	Anísio Haddad
15	São Caetano	Rio Claro	2	1	–

Tabela B.2: Resultado da partida retirada pelo Extrator

Teste	Equipe 1	Equipe 2	Gols 1	Gols 2	Estádio
1	Palmeiras	Barueri	3	0	Arena Barueri
2	Guaratinguetá	Palmeiras	2	1	–
3	Ponte Preta	Palmeiras	0	1	Moisés Lucareli
4	Portuguesa	Sertãozinho	1	0	Frederico Dalmazo
5	Guarani	Rio Preto	2	0	Brinco de Ouro
6	Corinthians	Noroeste	2	3	Alfredo de Castilho
7	Rio Claro	Ponte Preta	1	1	–
8	Guaratinguetá	Mirassol	3	1	–
9	Juventus	Rio Preto	0	0	–
10	Palmeiras	Rio Preto	3	3	–
11	Palmeiras	Juventus	4	0	Santa Cruz
12	Corinthians	Bragantino	1	1	–
13	Marília	São Paulo	3	2	Bento de Abreu
14	Rio Preto	Santos	2	1	Anísio Haddad
15	São Caetano	Rio Claro	2	1	–

Tabela B.3: Relações que foram extraídas dos textos

Teste	Equipe 1	Relação	Equipe 2
1	Palmeiras	vence	Barueri
2	Guaratinguetá	vence	–
3	Palmeiras	vence	–
4	Portuguesa	vence	–
5	Guarani	vence	–
6	Corinthians	perde	Noroeste
7	–	perdeu	–
8	Guaratinguetá	vence	–
9	Juventus	perde	Rio Preto
10	–	–	–
11	Palmeiras	goleia	–
12	–	–	–
13	–	–	–
14	Santos	perde	–
15	São Caetano	vence	Rio Claro

Células que contém '–' significam que o dado não pode ser obtido, tanto pela compreensão humana quanto pelo processamento do Extrator.