Fernando Costa Bertoldi

Detecção Automática de Termos Negados em Laudos Médicos

Florianópolis, Santa Catarina

18 de julho de 2011

Fernando Costa Bertoldi

Detecção Automática de Termos Negados em Laudos Médicos

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Orientador:
Aldo von Wangenheim

Co-orientador:
Rafael Andrade

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis, Santa Catarina

18 de julho de 2011

Agradecimentos

Agradeço primeiramente aos meus pais, que sempre me apoiaram e acreditaram em mim.

Agradeço também ao grupo Cyclops, pelo trabalho, aprendizado e amizades que me proporcionou.

Ao orientador Aldo von Wangenheim e Rafael Andrade, por sempre me guiarem e auxiliarem durante o desenvolvimento deste trabalho.

Ao Cloves Barcellos Langendorf, pelo apoio moral e companheirismo

Aos membros da banca, Alexandre Savaris e Rafael Andrade, pelo apoio, críticas e sugestões.

Resumo

Uma grande parte da informação médica está disponível em arquivos eletrônicos e há uma demanda crescente para que os sistemas de busca possuam maior precisão em suas respostas. A identificação de termos negados é de fundamental importância para a obtenção de precisão nas pesquisas. Este trabalho apresenta duas abordagens para a detecção automática de termos negados em laudos médicos, uma baseada na adaptação do algoritmo NegEx para a língua portuguesa, e outra usando aprendizado baseado em memória, e as compara.

Abstract

A great part of medical information is available in medical records, and there is a growing demand for search systems with greater precision in their answers. The identification of negated terms is an important factor precision-wise. This work presents and compares two approaches to the automatic identification of negated terms in medical records, one based on the adaptation of the NegEx algorithm to the portuguese language, and the other using Memory Based Learning.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Uma grande parte da informação médica está disponível em arquivos eletrônicos. Esses arquivos contêm informações valiosas que são utilizadas para diversas finalidades, como a realização de experimentos clínicos, a prática de medicina baseada em evidência, e pesquisa médica em geral (ROKACH; ROMANO; MAIMON, 2008). A maior parte dessas informações encontra-se em arquivos de texto livre e precisa ser transformada para um formato estruturado a fim de que possa ser utilizada. Devido ao contínuo crescimento e ao impacto dessas informações em pesquisas na área médica, há uma demanda crescente para que os sistemas de busca por informações possuam maior precisão em suas respostas. Para solucionar essa demanda, esses sistemas precisam ir além do nível superficial de recuperação de dados e processar também os recursos semânticos textuais, terminológicos e ontológicos da informação (ATHENIKOS; HAN, 2009). Por causa desse grande número de informações, a manipulação desses dados é um dos maiores desafios dos sistemas modernos de recuperação de informação na área de atenção à saúde. Nesse sentido, pesquisadores na área de recuperação de informação (Information Retrieval, ou IR) desenvolveram técnicas de processamento desses documentos para que documentos mais relevantes sejam retornados primeiro em resposta a consultas dos usuários.

Como resultado das pesquisas surgiram diversos aplicativos para a indexação, extração, e codificação de achados clínicos. A maioria dos sistemas se focaram na identificação de termos médicos usando técnicas tradicionais de IR, enquanto outros utilizaram modelos semânticos para melhorar a qualidade da pesquisa em bancos de dados médicos, p. ex., a ferramenta de recuperação de informações apresentada em (ANDRADE et al., 2009) utiliza, dentre outros métodos, indexação semântica e avaliação de similaridade.

1.1 Definição do Problema

Para que uma pesquisa em bases de dados biomédicas seja efetiva o contexto dos achados deve ser levado em conta. Em várias situações – como a seleção de pacientes para um ensaio clínico, ou a identificação de uma epidemia de alguma doença – as informações contidas no con-

texto dos termos são cruciais. Essas informações contextuais são importantes porque acabam mudando o sentido dos termos a que se referem. Por exemplo, uma pesquisa por pacientes com pneumonia onde o contexto é ignorado pode retornar casos que ocorreram no passado ('O paciente tem histórico de pneumonia') ou que ocorreram com um membro da família ('Histórico familiar de pneumonia'). Neste exemplo o não tratamento das características contextuais de temporalidade (ocorreu no presente ou passado) e experimentador (pessoa a quem o termo se refere) compromete a precisão da busca.

A negação é provavelmente a característica contextual mais importante em laudos médicos. Um estudo mostrou que a característica contextual Negação é a mais importante para classificar pacientes baseado na presença de síndrome respiratória aguda baixa (CHU; DOWLING; CHAPMAN, 2006). Assim como as características contextuais de temporalidade e experimentador, a negação interfere no sentido da frase. Por exemplo, considerando "O paciente reclama de náuseas e dores de cabeça"e "Ausência de sintomas, como náuseas e febre", ambas as frases possuem a palavra náuseas, porém somente a primeira frase é relevante em uma busca pela presença do sintoma náusea. Se a negação não é levada em conta, muitos dos laudos retornados serão irrelevantes e nesse caso, diminui-se a precisão dos resultados. Uma forma de obter resultados mais precisos é a utilização de um mecanismo de detecção de termos negados em laudos médicos e assim, obter resultados de pesquisa com maior qualidade.

Para resolver o problema da negação será criado um sistema de detecção de termos negados em laudos médicos em português.

1.2 Justificativa

A Rede Catarinense de Telemedicina (RCTM) é um projeto da Universidade Federal de Santa Catarina (UFSC) iniciado em 2005 com o objetivo de dar suporte à saúde através de um sistema descentralizado de telemedicina. A Figura 1.1 mostra o cenário atual dos municípios cobertos pela RCTM. O aplicativo central da RCTM é o Sistema de Telemedicina e Telessaúde de Santa Catarina (STT/SC). Dentre outras funcionalidades, o STT permite que profissionais da saúde localizados em diversas cidades do estado realizem exames médicos, como tomografias computadorizadas e eletrocardiogramas em suas cidades, e envie-os pela rede de telemedicina para servidores localizados no Hospital Universitário, onde os laudos serão emitidos por uma equipe de especialistas. Esses exames são armazenados em um banco de dados central, que em maio de 2011 possui em torno de 370.000 pacientes registrados e 930.000 exames de imagens e de análises clínicas armazenados (ANDRADE, 2011). armazenados. Mais de 1.200

profissionais da saúde têm acesso ao portal.

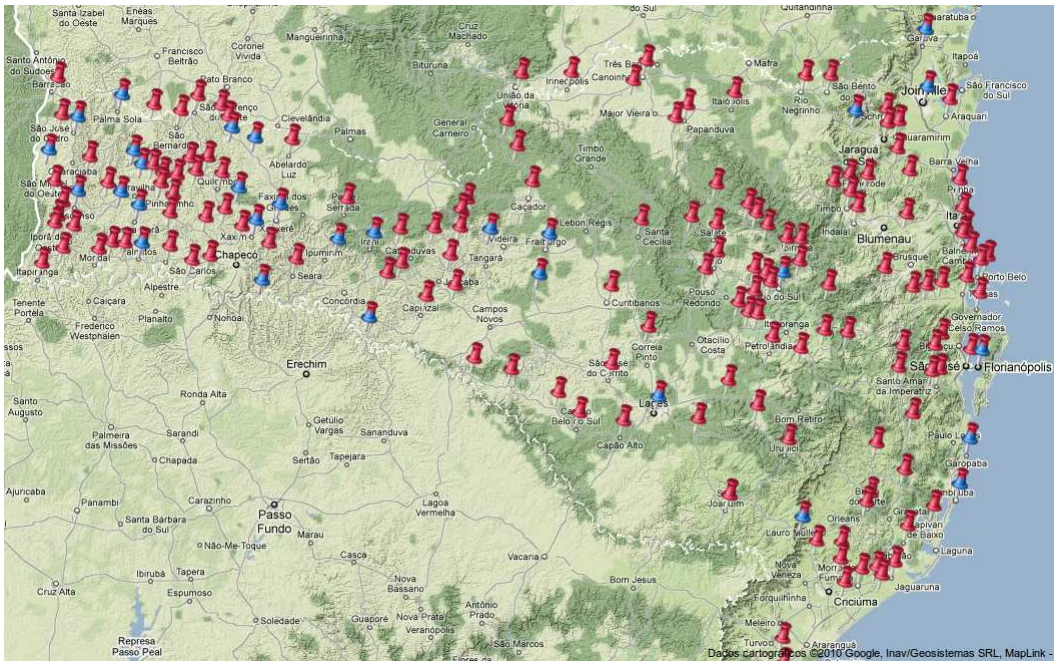

Figura 1.1: Mapa da Rede Catarinense de Telemedicina.

A fim de que esse grande volume de informação esteja disponível para pesquisa pelos profissionais da saúde, é necessário um mecanismo de busca. No entanto as ferramentas de busca que se utilizam de mecanismos tradicionais não provêem resultados de alta qualidade, isto é, dentro do contexto de avaliação de sistemas de recuperação de informação estes resultados possuem baixos índices de precisão e revocação. Um sistema de recuperação de informações de dados médicos está sendo desenvolvido no laboratório de Telemedicina com o objetivo de realizar pesquisas na base da STT mais eficientemente e com maior qualidade do que o sistema em uso (ANDRADE et al., 2009). Uma das questões que precisam ser abordadas pelo sistema é a identificação e indexação de termos negados.

Durante os últimos dez anos foram desenvolvidos vários métodos para determinar se um termo médico está negado (CHAPMAN, 2001; MUTALIK; DESHPANDE; NADKARNI, 2001; GORYACHEV et al., 2006; HARKEMA et al., 2009). Como não existem trabalhos na área de detecção de negação em textos médicos em português, foi necessário a adaptação de alguma técnica já existente para língua portuguesa.

1.3 Cenário de Aplicação - Estatísticas de Morbidade

Considere que um médico deseja saber a quantidade de portadores de doenças em relação à população de determinada região. Por exemplo, um usuário do sistema deseja saber qual a quantidade de pacientes com *"tireoidite"* nos laudos de seus exames. Uma pesquisa simples, sem o sistema de detecção de termos negados, retornará resultados irrelevantes. Por exemplo, em um laudo que contenha a frase *"o aspecto sugere tireoidite."*, não há certeza se há a presença de tireoidite, e portanto este é um caso que não pode ser contabilizado na estatística. Uma nova pesquisa, agora com o sistema de detecção de termos negados ativado, o laudo do exemplo anterior não seria retornado. O sistema economiza o tempo do usuário na medida em que ele não precisa verificar em cada laudo características contextuais que possam modificar o sentido do achado para poder excluir resultados irrelevantes.

Como resultado dessa pesquisa, o pesquisador poderá extrair dados e estatísticas que ajudarão a definir políticas de saúde para aquela determinada região.

1.4 Objetivos

1.4.1 Objetivo Geral

O objetivo do presente trabalho é a criação de um sistema de detecção de termos negados em laudos médicos.

1.4.2 Objetivos Específicos

- Implementar uma adaptação do algoritmo NegEx voltado para a língua portuguesa.
- Implementar um classificador de termos negados usando o paradigma de aprendizado baseado em memória.
- Avaliar e comparar as duas implementações para verificar qual delas possui a melhor performance.

2 *Revisão Bibliográfica*

A revisão bibliográfica foi feita utilizando o método de revisão sistemática proposto em (KITCHENHAM, 2004). Primeiramente são apresentados os conceitos fundamentais, e em seguida a revisão sistemática.

2.1 Conceitos Fundamentais

2.1.1 DeCS

O vocabulário estruturado e trilíngue DeCS - Descritores em Ciências da Saúde (DESCRITORES..., 2011) foi criado pela BIREME (BIREME, 2011) para servir como uma linguagem única na indexação de artigos de revistas científicas, livros, anais de congressos, relatórios técnicos, e outros tipos de materiais, assim como para ser usado na pesquisa e recuperação de assuntos da literatura científica nas fontes de informação disponíveis na Biblioteca Virtual em Saúde (BVS) como LILACS, MEDLINE e outras.

Foi desenvolvido a partir do MeSH - Medical Subject Headings (MESH, 2011) da U.S. National Library of Medicine (NLM) com o objetivo de permitir o uso de terminologia comum para pesquisa em três idiomas, proporcionando um meio consistente e único para a recuperação da informação independentemente do idioma.

Os conceitos que compõem o DeCS são organizados em uma estrutura hierárquica, permitindo a execução de pesquisas em termos mais amplos ou mais específicos ou todos os termos que pertençam a uma mesma estrutura hierárquica.

O DeCS é um vocabulário dinâmico totalizando 30.895 descritores, sendo destes 26.225 do MeSH e 4670 exclusivamente do DeCS. Existem 2032 códigos hierárquicos de categorias DeCS a 1479 descritores MeSH. As seguintes são categorias DeCS e seus totais de descritores: Ciência e Saúde (219), Homeopatia (1.944), Saúde Pública (3.491) e Vigilância Sanitária (828). O número é maior que o total, pois um descritor pode ocorrer mais de uma vez na hierarquia.

Por ser dinâmico, registra processo constante de crescimento e mutação, registrando a cada ano um mínimo de 1000 interações na base de dados dentre alterações, substituições e criações de novos termos ou áreas.

Os conceitos do vocabulário DeCS estão assim distribuídos (versão 2011):

- 26,4% referem-se a compostos químicos e drogas (categoria D), entendendo aqui tanto as drogas exógenas como as endógenas;
- 20,7% do total são de anatomia (categoria A), de organismos (categoria B) e de fenômenos e processos (categoria G);
- 13,2% do total são referentes a doenças (categoria C);
- as técnicas e equipamentos (categoria E), ciências afins (categorias F, H, I, J, K, L, M, N), características de publicações (categoria V) e áreas geográficas (categoria Z) representam juntas 20,6%;
- Saúde Pública (categoria SP), com um total de conceitos que representam 10,3% do total, Homeopatia (categoria HP) com 5,7%, Vigilância Sanitária (categoria VS) com 2,4% e Ciência e Saúde (categoria SH) com 0,6% dos conceitos. Estas quatro últimas categorias foram especialmente desenvolvidas para melhor representar a literatura gerada nos países da região.

2.1.2 Processamento de Linguagem Natural

Processamento de Linguagem Natural (PLN) é uma área da ciências da computação e linguística preocupada em estudar sistemas que analisam, tentam entender, ou produzem uma ou mais línguas humanas naturais, como o inglês, japonês, italiano, etc. O termo 'natural' neste contexto é usado para distinguir a escrita e fala humana de linguagens formais, como notações lógicas e matemáticas, ou linguagens de computador, como Java, Haskell, e C++. As tarefas realizadas pelo PLN envolvem desde as mais triviais, como a contagem de palavras e hifenização automática, até aplicações mais complexas, como a tradução em tempo real e a resposta automática a perguntas.

O que distingue essas aplicações de outros tipos de processamento de dados é o *conhecimento da linguagem*. Esse conhecimento pode ser dividido em 6 categorias (JURAFSKY et al., 2000):

Fonética e Fonologia o estudo dos sons linguísticos.

Morfologia o estudo dos componentes das palavras.

Sintaxe o estudo das relações estruturais entre as palavras.

Semântica o estudo do significado.

Pragmática o estudo de como o contexto contribui ao significado.

Discurso o estudo de unidades linguísticas maiores que uma elocução.

Um fato interessante a respeito das seis categorias de conhecimento linguístico é que a maioria das tarefas em processamento de linguagem resolvem ambiguidade em um desses níveis. Uma entrada é dita ambígua se múltiplas estruturas linguísticas podem ser derivadas a partir dela.

Duas visões sobre PLN

Historicamente as abordagens à área de PLN se dividem em dois paradigmas: simbólico e empírico.

1. O paradigma simbólico surgiu de duas linhas de pesquisa. A primeira foi o trabalho de Noam Chomsky e outros no desenvolvimento da teoria das linguagens formais durante o fim dos anos 50 e início dos anos 60, e no trabalho de vários linguistas e cientistas da computação em algoritmos de parsing. Um dos primeiros sistemas de parsing completo foi o Transformation and Discourse Analysis Project (TDAP) de Zelig Harris, implementado entre junho de 1958 e julho de 1959 na Universidade da Pensilvânia. A segunda linha de pesquisa foi a Inteligência Artificial, surgida em 1956 a partir de uma conferência na Universidade de Dartmouth que reuniu, entre outros, John McCarthy, Marvin Minsky, Allen Newell and Herbert Simon, figuras que viriam a ser líderes na pesquisa em IA. O início de IA foi marcado pela visão racionalista, caracterizada pela crença de que boa parte do nosso conhecimento é inato, derivado provavelmente da herança genética. Essa visão se reflete em sistemas com uma abordagem 'top-down', onde o conhecimento e sistemas de raciocínio são codificados de antemão, de modo a simular o conhecimento inato do ser humano.

2. O paradigma empírico, baseado no tratamento estatístico da linguagem, se tornou prevalente a partir dos anos 90 por vários motivos (DAELEMANS; BOSCH, 2005). Primeiramente, a capacidade de processamento e armazenamento dos computadores avançou de tal modo que a aplicação de métodos estatísticos de reconhecimento de padrões sobre

grandes volumes de texto e de fala se tornou viável. Em segundo lugar, tem havido um interesse, tanto por parte da academia como de empresas, pelo desenvolvimento de sistemas de PLN que tenham uma boa escalabilidade e que não necessitem de uma análise sintática e semântica completa do texto. E por último, métodos probabilísticos tiveram um grande sucesso nas áreas de reconhecimento de fala e recuperação de informação, e por isso foram transferidos para a área de PLN.

2.1.3 Part-of-Speech Tagging

Part-of-speech tagging (POS tagging ou POS), também chamado etiquetamento morfossintático, ou etiquetamento gramatical, é o processo de atribuir (ou etiquetar) à cada uma das palavras de um texto sua categoria morfossintático.

À primeira vista o etiquetamento morfossintático pode parecer uma tarefa trivial, bastando utilizar um dicionário de palavras e suas respectivas categorias. Contudo, algumas palavras podem ter categorias diferentes gramaticais em função do contexto em que aparecem. Este fato costuma ocorrer com frequência em linguagens naturais, onde uma grande quantidade de palavras são ambíguas. Por exemplo, a palavra 'nada' pode ser um substantivo ou uma forma do verbo 'nadar'.

A maioria dos algoritmos de POS se encaixa em duas categorias: baseado em regras e estocásticos. Etiquetadores baseados em regras envolvem grandes bancos de dados com regras de desambiguação que especificam, por exemplo, que uma palavra ambígua é um substantivo ao invés de um verbo se ela aparece depois de um determinante.

Etiquetadores estocásticos resolvem as ambiguidades utilizando um corpus de treinamento para computar a probabilidade de uma dada palavra ter uma categoria em um determinado contexto. Ultimamente eles têm se mostrado mais eficientes que os etiquetadores baseados em regras (HALTEREN; DAELEMANS; ZAVREL, 2001 apud ROKACH; ROMANO; MAIMON, 2008, p. 199–229). Dentre os modelos estocásticos mais usados podemos citar modelos de markov oculto, campo aleatório condicional (Conditional Random Field, ou CRF), e modelos baseados em memória. O aprendizando baseado em transformações é uma abordagem híbrida, que se utiliza de regras e um corpus de treinamento.

O etiquetamento morfossintático tem forte relação com a identificação de negações. De fato, o problema da negação pode ser considerado um caso especial de etiquetamento morfossintático (ROKACH; ROMANO; MAIMON, 2008), onde há duas classes – N para termos negados e P caso contrário – que são aplicadas a termos médicos. A aprendizagem baseada em

memória tem sido utilizada com sucesso em etiquetamento morfossintático (DAELEMANS et al., 1996).

2.1.4 Classificação de Textos

Classificação consiste em atribuir classes pré-definidas para uma dada entrada. Exemplos de aplicações da classificação de textos:

- filtragem de email para exclusão de spam.
- a classificação de notícias de jornal em tópicos como política, esporte, e ciência.
- decidir o sentido de uma palavra, p. ex., se em um determinado contexto a palavra 'banco' significa 'objeto para sentar' ou 'instituição que guarda dinheiro'. Esta tarefa também é conhecida como desambiguação lexical de sentido.

2.1.5 Aprendizado Baseado em Memória

O Aprendizado Baseado em Memória (ABM) é uma forma de aprendizado supervisionado e indutivo a partir de exemplos (DAELEMANS et al., 1996, p. 2). ABM é fundamentado na hipótese de que tarefas cognitivas são baseadas na similaridade entre novas situações e *representações armazenadas na memória de experiências anteriores*, ao invés de se aplicar *regras mentais* abstraídas de experiências anteriores (como no processamento baseado em regras). Esta abordagem surgiu em diferentes contextos sob diversos nomes, como raciocínio baseado em casos, baseado em exemplos, baseado em instâncias, e aprendizado preguiçoso (WANGENHEIM; WANGENHEIM, 2003; AHA, 1997).

Um sistema ABM, visualizado na Figura 2.1, contém dois componentes: um *componente de aprendizado*, baseado em memória, e um *componente de performance*, baseado em similaridade.

O componente de aprendizado é dito ser baseado em memória porque nele instâncias de treinamento são adicionadas à memória (também conhecida como *base de casos* ou de instâncias). Uma instância consiste de um vetor de tamanho fixo de pares atributo-valor, e um campo de informação contendo a classificação daquele vetor em particular.

No componente de performance a base de casos é usada para classificar novas instâncias; Durante a classificação, uma nova instância é apresentada ao sistema. A classe dessa instância é determinada com base na extrapolação dos exemplos mais *similares* na memória (os k-vizinhos

mais próximos). A similaridade entre uma instância X e as instâncias Y na memória é computada usando uma métrica de distância $\Delta(X,Y)$.

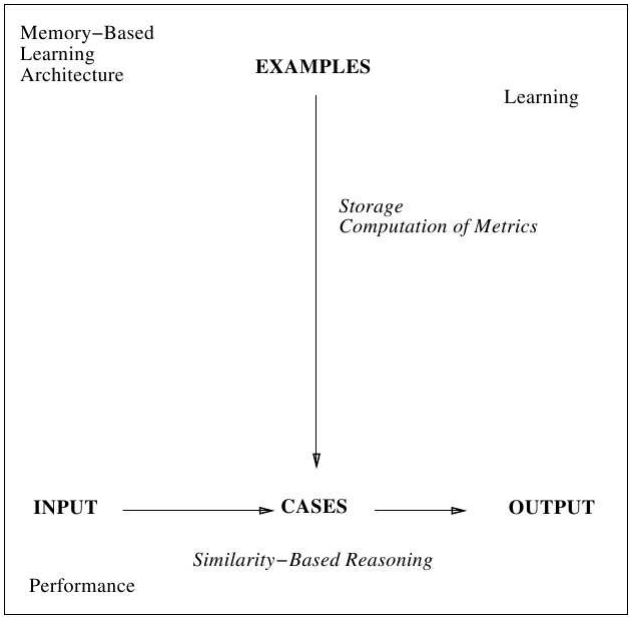

Figura 2.1: Arquitetura geral de um sistema ABM (DAELEMANS et al., 2010).

Medidas de Similaridade

Para vetores com atributos de valor simbólico a medida de similaridade mais simples é a *distância de hamming*, representada na Equação 2.1. $\Delta(X,Y)$ é a distância entre as instâncias X e Y, representadas por n atributos, e δ (Equação 2.2) é a distância para cada atributo. A distância é simplesmente a soma das diferenças entre atributos. O algoritmo $k-NN$ com esta métrica é chamado de **IB1**.

$$\Delta(X,Y) = \sum_{i=1}^{n} \delta(x_i, y_i) \tag{2.1}$$

onde:

$$\delta(x_i, y_i) = \begin{cases} 0 & \text{se } x_i = y_i \\ 1 & \text{se } x_i \neq y_i \end{cases} \tag{2.2}$$

A métrica de similaridade da Equação 2.1 conta o número de atributos que casam em ambas as instâncias. Na falta de informação sobre a relevância dos atributos, a distância de hamming se torna uma escolha razoável. No entanto, é possível adicionar conhecimento do domínio ao

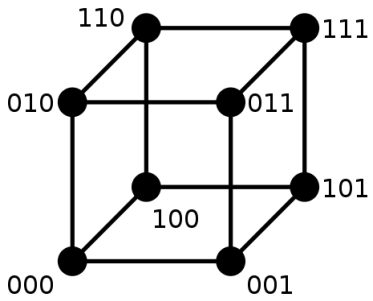

Figura 2.2: Cubo binário de três bits para achar a distância de Hamming (HAMMING..., 2006).

modelo através de pesos atribuídos aos atributos do vetor. Uma maneira de se conseguir isso é usar o **ganho de informação** de cada atributo como seu peso.

Na ponderação por **ganho de informação** (GI), em cada atributo é medido quanta informação ele contribui para a determinação da classe correta. O ganho de informação é medido calculando a diferença no grau de incerteza—também conhecida como **entropia**—com e sem o conhecimento do valor do atributo (Equação 2.3).

$$w_i = H(C) - \sum_{v \in V_i} P(v) \times H(C|v) \tag{2.3}$$

$$H(C) = -\sum_{c \in C} P(c) log_2 P(c) \tag{2.4}$$

Onde C é o conjunto de classes, V_i é o conjunto de valores para o atributo i, e $H(C)$ (Equação 2.4)

é a entropia das classes. As probabilidades são estimadas a partir das frequências relativas no conjunto de treinamento.

O ponderação por GI tende a "enfraquecer" valores com baixa frequência e com muita informação, e superestimar a relevância de atributos com um grande número de valores. Por exemplo, em um conjunto de dados de pacientes, onde há um atributo "ID do paciente", este atributo terá um valor de GI alto, porém ele não adiciona nenhuma generalização para novas instâncias. Para normalizar o ganho de informação de atributos com números variados de valores, Quinlan (QUINLAN, 1993 apud DAELEMANS et al., 2010) introduziu uma versão normalizada, chamada de **Taxa de Ganho**, que consiste no Ganho de Informação dividido por *si* (*split info*) (Equação 2.6), a entropia dos valores do atributo (Equação 2.5).

$$w_i = \frac{H(C) - \sum_{v \in V_i} P(v) \times H(C|v)}{si(i)} \tag{2.5}$$

$$si(i) = -\sum_{v \in V_i} P(v) log_2 P(v) \tag{2.6}$$

A escolha do tipo de representação para as instância em ABM é o fator chave para determinar a força desta ou daquela abordagem (DAELEMANS et al., 2010). Os atributos e categorias em tarefas de PLN são usualmente representados por rótulos simbólicos. Os exemplos de métricas vistos até agora se limitam a um casamento perfeito entre os vetores de atributo-valor, isto é, todos os valores de um atributo são considerados igualmente dissimilares. Porém há problemas, como p. ex. no domínio fonético, em que 'm' e 'n' são mais similares que 'b' e 'a'. Para esse propósito uma métrica foi definida por Stanfill e Waltz (STANFILL; WALTZ, 1986 apud DAELEMANS et al., 2010) e posteriormente melhorada por Cost e Salzberg (COST; SALZBERG, 1993 apud DAELEMANS et al., 2010). Ela é chamada de **(Modified) Value Difference Metric** (MVDM; Equação 2.7)

$$\delta(v_1, v_2) = \sum_{i=1}^{n} |P(C_i|v_i) - P(C_i|v2)| \tag{2.7}$$

Apesar de a métrica MVDM não computar explicitamente a relevância dos atributos, existe um efeito implícito de ponderação dos atributos. Se os atributos são muito informativos, suas probabilidades condicionais das classes serão em média bastante enviesadas. Isto implica que em média $\delta(v_1, v_2)$ será grande. Para atributos não-informativos, por outro lado, as probabilidades condicionais das classes serão bem uniformes, de forma que em média $\delta(v_1, v_2)$ será menor.

2.2 Revisão Sistemática

Este estudo foi feito com o objetivo de identificar a literatura existente sobre detecção automática de termos negados em textos médicos.

2.2.1 Questões de pesquisa

As questões de pesquisa abordadas neste estudo são:

Q1. Quais são as abordagens existentes sobre a detecção de termos negados?

Q2. Quais abordagens obtiveram melhor desempenho em termos de precisão, especificidade e sensitividade?

Q3. As abordagens exigem um conhecimento profundo da língua em que está sendo aplicada?

Com relação à questão 2, os parâmetros de precisão, especificidade, e sensitividade foram escolhidas porque são os mais comumente usados em pesquisas com modelos de PLN (MANNING; SCHUETZE, 1999, p. 267).

Levando em consideração a implementação e integração dos algoritmos com o sistema de recuperação de dados do STT, as seguintes perguntas foram feitas a fim de avaliar a viabilidade do uso de frameworks para construção de sistemas PLN:

Q2.1 Quais são os frameworks existentes na área de PLN?

Q2.2 Eles são integráveis com o sistema Lucene?

Q2.3 Quais linguagens de programação eles suportam?

Como este trabalho também involve a criação de um conjunto de treinamento e teste anotado, foram feitas perguntas em relação a ferramentas que auxiliam na anotação de corpus:

Q3.1 Quais ferramentas de anotação de corpus existem?

Q3.2 Quais ferramentas são mais fáceis de instalar?

Q3.3 Quais ferramentas são mais usáveis?

2.2.2 Critérios de Inclusão e Exclusão

Foram incluídos todos os artigos escritos na língua inglesa ou portuguesa relacionados a processamento de linguagem natural, recuperação de informação, e informática biomédica , revisados por pares, e que estejam em journals ou anais de congresso.

Artigos sem uma avaliação formal dos resultados foram excluídos.

2.2.3 Estratégia de Pesquisa

Os seguintes portais de pesquisa foram utilizados: IEEEXplore, ACM Digital Library, ScienceDirect. A pesquisa foi realizada no dia 8 de Outubro de 2010.

IEEEXplore:

“negation detection” OR (negation AND (“natural language processing” OR nlp OR “information retrieval” OR ir)) OR “clinical data mining” OR “text classification”

ACM Digital Library:

(“negation detection”) and (Keywords:negation OR Keywords:nlp OR Keywords:“information extraction” OR Keywords:“clinical data mining” OR Keywords:biomedical OR Keywords:medical OR Keywords:clinical)

ScienceDirect:

“negation detection” OR (negation AND (“natural language processing” OR nlp OR “information retrieval” OR ir)) OR “clinical data mining” OR “text classification”

2.2.4 Seleção dos Estudos

A pesquisa retornou no total 540 estudos. Desses, 10 foram selecionados como relevantes para a pesquisa. Houve casos onde dois artigos pelos mesmos autores referenciam um mesmo estudo, como foi o caso de (GINDL; KAISER; MIKSCH, 2008). Nesse caso somente um dos artigo foi selecionado.

2.2.5 Extração dos Dados

Para cada artigo selecionado para análise, foram extraídas as seguintes informações:

Estudo. Referência do artigo no qual o estudo é apresentado.

Tipo de abordagem. Se a abordagem é baseada em engenharia de linguagem (EC) ou aprendizado de máquina (AM).

Tamanho do conjunto de testes. Número de relatórios médicos usados como conjunto de teste. Os estudos onde o conjunto de teste é especificado em termos de sentenças estão sinalizados com a palavra ”sentença”.

Precisão. Precisão, em percentual, do estudo.

Especificidade. Especificidade, em percentual, do estudo.

Sensitividade. Sensitividade (revocação), em percentual, do estudo.

A tabela 2.1 apresenta os dados extraídos de cada estudo:

Tabela 2.1: Dados extraídos

Estudo	Tipo de abordagem	Tamanho do Conjunto de Testes	Precisão (%)	Especificidade (%)	Sensitividade (%)
(CHAPMAN, 2001)	EL	560 laudos	84,5	94,5	77,8
(MUTALIK; DESHPANDE; NADKARNI, 2001)	EL	30 sumários de alta e 30 notas cirúrgicas	97,7	97,7	95,3
(GORYACHEV et al., 2006) (NegEx)	EL	100 sumários de alta	84,8	94,2	94,5
(GORYACHEV et al., 2006) (NegExpander)	EL	100 sumários de alta	92,5	97	89,9
(GORYACHEV et al., 2006) (SVM)	AM	100 sumários de alta	87,9	95,2	84
(GORYACHEV et al., 2006) (Naïve Bayes)	AM	100 sumários de alta	80,4	92	76,1
(HUANG; LOWE,)	EL	120 laudos de radiologia	98,6	99,8	92,6
(ROKACH; ROMANO; MAIMON, 2008)	AM	1766 sentenças	94,6	-	96,7
(GINDL; KAISER; MIKSCH, 2008)	EL	14 diretrizes de prática clínical	67,5	-	83,5
(NASSIF et al., 2009)	EL	100 mamogramas	97,7	-	95,5
(UZUNER; ZHANG; SIBANDA, 2009)	EL	1954 relatórios de radiologia	93	-	74
(HARKEMA et al., 2009)	EL	120 laudos	94	-	92
(SKEPPSTEDT, 2010)	EL	436 sentenças de registros médicos	70	-	81

2.3 Estado da Arte

Nos últimos dez anos foram desenvolvidas técnicas e algoritmos para a detecção de negações em textos médicos (ROKACH; ROMANO; MAIMON, 2008; MUTALIK; DESHPANDE; NADKARNI, 2001; CHAPMAN, 2001; GORYACHEV et al., 2006).

Em (CHAPMAN, 2001) o grupo de trabalho liderado por Wendy Chapman desenvolveu o NegEx, um algoritmo baseado em expressões regulares para determinar se achados ou doenças mencionadas em relatórios médicos estão negados ou não. Esse trabalho parte do pressuposto

de que, a despeito da complexidade do problema da negação em processamento de linguagem natural, textos médicos em geral são menos ambíguos do que textos sem restrições.

Para a realização do estudo o algoritmo NegEx foi comparado com um algoritmo de base criado na Universidade de Pittsburgh. O algoritmo de base procura por 6 frases de negação e marca como negados todos os termos encontrados que estejam entre uma frase de negação e o fim da sentença. O NegEx foi projetado com base neste algoritmo simples, usando frases de negação adicionais e uma expressão regular mais detalhada. Ele funciona da seguinte maneira: na etapa de pré-processamento os laudos são divididos em sentenças, sinais de pontuação são removidos, e em cada sentença, os termos identificados na UMLS (Unified Medical Language System) que estão relacionados a achados e doenças são substituídos por seus respectivos identificadores na UMLS. Em seguida o NegEx recebe como entrada uma sentença pré-processada e tem como saída o status de negação de cada termo da UMLS na sentença. O algoritmo utiliza 35 frases de negação, divididas em três categorias. A primeira categoria, frases pseudo-negadas, são frases que parecem identificar negações, mas que de fato não são identificadores confiáveis de negação. Eles são usados para descartar falsos positivos. As outras duas categorias, pré-UMLS e pós-UMLS, são frases que ocorrem antes e depois dos termos que estão negando, respectivamente. As expressões regulares determinam o escopo da negação dentro da sentença. NegEx usa as seguintes expressões regulares:

```
<frase de negação pré-UMLS> [0-5 palavras] <termo UMLS>
```

e

```
<termo UMLS> [0-5 palavras] <frase de negação pós-UMLS>.
```

Caso a frase e uma das expressões regulares sejam relacionadas, os termos UMLS presentes na frase são identificados como negados.

No estudo do NegEx foram utilizados 2060 sumários de alta selecionados aleatoriamente e anonimizados. Deste conjunto de dados foram extraídos 1500 sumários para compor o conjunto de treinamento. Os sumários do conjunto de treinamento foram lidos para poder extrair as frases de negação e as expressões regulares. Os 560 sumários restantes formaram o grupo de teste. As 1000 sentenças do conjunto de teste contiveram 1235 ocorrências de termos UMLS. Para estabeler um "padrão áureo", contra o qual o NeGex pudesse ser comparado, três médicos julgaram as 1000 sentenças e marcaram cada termos da UMLS encontrado como (a)presente—o médico que fez o laudo constatou a presença do termo, (b)ausente—está explícito que o termo

está ausente no laudo, ou (c)ambíguo—não está claro se o termo está presente ou ausente. Cada médico avaliaram 400 das 1000 sentenças. Para determinar a confiabilidade inter-observadores (Kappa) 200 sentenças foram avaliadas por pares de médicos. Os avaliadores 1 e 2 julgaram 100 sentenças, e os avaliadores 2 e 3 julgaram outras 100 sentenças. Se dois avaliadores não concordassem com a avaliação de um termo em uma frase, esse termo é marcado como ambíguo.

No teste o algoritmo NegEx obteve uma especificidade de 94,5%, precisão de 84,5% e sensitividade de 77,8%, enquanto que o algoritmo de base obteve especificidade de 85,3%, precisão de 68,4%, e sensitividade de 88,3%. Comparando os resultados, nota-se uma melhora na especificidade e precisão. O principal motivo da melhora da precisão é a limitação do escopo da frase de negação. Por outro lado, esse escopo fez com que termos UMLS negados em frases longas não fossem corretamente negados, piorando assim a sensitividade em relação ao algoritmo de base.

Outro trabalho muito importante na área é o NegFinder, um sistema baseado em ferramentas de construção de compiladores (MUTALIK; DESHPANDE; NADKARNI, 2001). Ele consiste de vários componentes organizados em modo pipeline, isto é, a saída de um componente é a entrada do componente seguinte. Na primeira etapa os termos da UMLS no documento são identificados. Em seguida cada termo UMLS é substituído pelo seu UMLS ID. Na etapa seguinte o analisador léxico identifica palavras de negação e as classifica de acordo com a combinação de algumas características, tais como: a negação antecede ou sucede o conceito negado, ela pode negar vários conceitos. A sequência de tokens gerada pelo analisador léxico é passada para o parser, que então associa as palavras de negação com um ou mais conceitos UMLS. Na etapa de verificação o documento original é marcado com cores para facilitar na validação do resultado.

O NegFinder foi treinado com um conjunto de 40 documento médicos de diversas especialidades (relatórios de radiologia, notas cirúrgicas, sumários de alta) e testado com um conjunto de 60 documentos (30 sumários de alta e 30 notas cirúrgicas), onde os textos marcados foram inspecionados visualmente para poder quantificar o número de resultados falso positivos e falso negativos. No teste o algoritmo obteve especificidade de 91,8%, sensitividade de 95,7% e precisão de 91,8%. As palavras “no”, “not”, “denies/denied”, e “without” estavam presentes em 92,5% das negações. De acordo com o estudo, os erros cometidos pelo NegFinder incluem a falta de palavras de negação ou terminadores de negação, documentos com a linguagem fora dos padrões, uso de uma janela de três palavras para terminar o escopo de uma negação, negações duplas, entre outros.

No trabalho de Goryachev (GORYACHEV et al., 2006) são comparados quatro métodos de negação: dois deles são adaptações de sistemas já existentes, o NegEx e o NegExpander, e os

outros dois são classificadores baseados em aprendizado de máquina, o Naïve Bayes e o Support Vector Machine (SVM). A única modificação no NegEx foi a inclusão de dois tipos semânticos da UMLS para poder ampliar o número de achados identificados. O NegExpander faz parte do sistema de recuperação de informações InQuery pela Universidade de Massachusetts em Amherst. Ele utiliza frases conjuntivas para definir o escopo da negação. Frases conjuntivas são sintagmas nominais conectados por conjunções, tais como “e” e “mas”. A sua entrada são sentenças com termos da UMLS identificados e com etiquetas morfológicas. O algoritmo identifica as frases conjuntivas na sentença, e então procura por frases de negação dentro das frases conjuntivas. Se ao menos uma frase de negação é achada, todos os termos da UMLS dentro frase conjuntiva são negados.

Para as abordagens baseadas em aprendizado de máquina, os dois classificadores utilizados foram treinados em 1745 sumários de alta contendo achados e seus status de negação (afirmativo, negativo, e possível). Cada sumário de alta foi processado por um conjunto de ferramentas NLP para poder identificar os termos da UMLS. O pipeline de das ferramentas consiste de um tokenizador, um etiquetador morfossintático, um divisor de sentenças, um identificador de sintagmas nominais, e um identificador de conceitos UMLS. Antes de se treinar os classificadores, as sentenças que continham termos da UMLS passaram por uma etapa de pré-processamento, ilustrado na Figura 2.3. Usando a lista de termos disparadores do NeGex, termos de negação foram identificados e substituídos por tags correspondentes da forma [pre-neg], [pre-pos], [post-neg], e [post-pos]. Cada termo da UMLS foi substituído por uma tag [term]. Em seguida, um etiquetador substituiu as palavras restantes por sua categoria morfossintática correspondente. Usando as sentenças etiquetadas, um vetor de atributos foi extraído para cada termo UMLS, usando seis tags adjacentes de cada lado do conceito. Caso não houvessem tags em determinada posição, uma tag [null] era usada naquela posição. Dois modelos de classificação, o Naive Bayes e o SVM, foram alimentados com o conjunto de treinamento resultante.

O NegEx e o NegExpander obtiveram melhores resultados—precisão de 84,8% e 92,5%, revocação de 94,2% e 97%, respectivamente—do que as abordagens baseadas em aprendizado de máquina—Naive Bayes e SVM obtiveram precisão de 80,4% e 88%, revocação de 92% e 95,2%, respectivamente.

Huang e Lowe (HUANG; LOWE,) desenvolveram uma abordagem híbrida, combinando expressões regulares com parsing de linguagem natural. Eles partiram de duas hipóteses: 1) a informação contida em árvores sintáticas ajuda a identificar o escopo da negação e 2) em laudos de radiologia, uma gramática de negação pode ser derivada de um pequeno número de laudos e uma abordagem híbrida pode atingir boa performance usando esta gramática. Eles

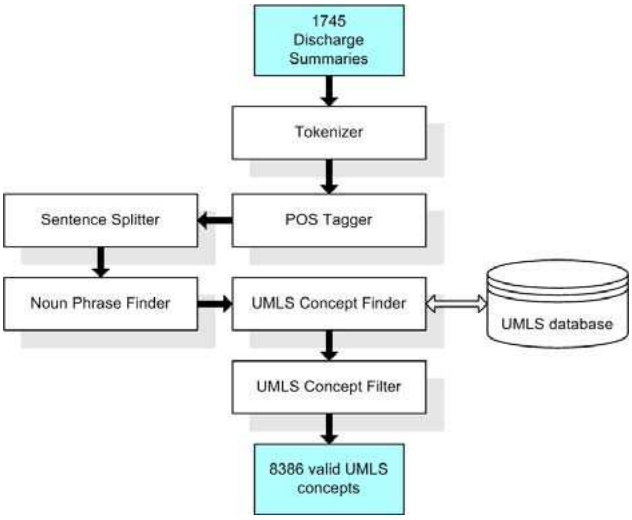

Figura 2.3: O processo de extração de termos em (GORYACHEV et al., 2006, p. 3)

utilizaram um parser da língua inglesa de alta precisão, o Stanford Parser (KLEIN; MANNING, 2003), para obter a árvore sintática de cada sentença. Através da inspeção manual da estrutura sintática de 30 laudos de radiologia, obteve-se uma classificação de tipos de negações, onde cada tipo é associado a uma expressão regular envolvendo estruturas sintáticas, tais como sintagmas nominais, preposicionais, e verbais. As 11 categorias pode ser vistas na Figura 2.4.

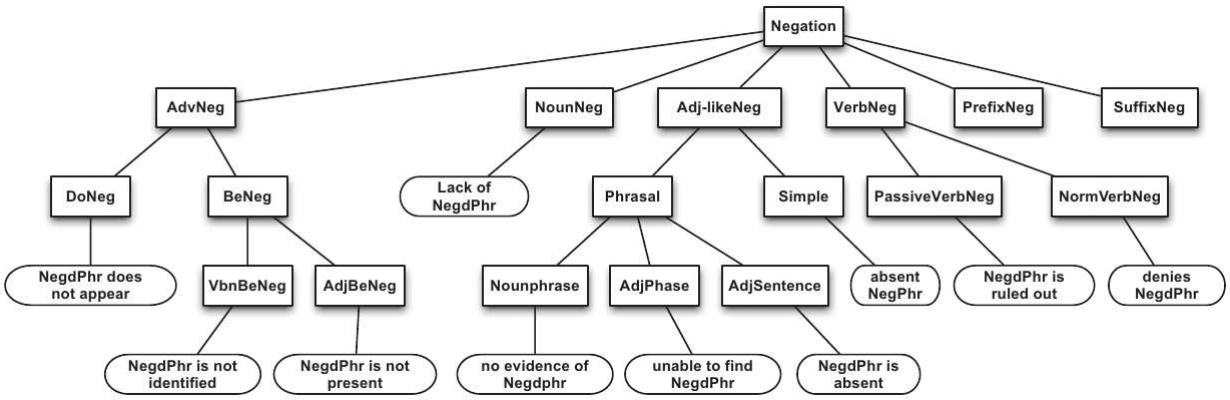

Figura 2.4: Árvore de classificação de negações. Classes são mostradas em retângulos, e exemplos nas caixas ovais (HUANG; LOWE, , p. 5).

A Figura 2.5 é um exemplo de árvore sintática resultante do parsing feito pelo Stanford Parser. Algoritmos que não usam informações estruturais de sentenças teriam dificuldade em detectar a negação, pois o sintagma “The previously identified isoechoic nodule on the right” e o sinal de negação “not” estão separados pelo sintagma “larger than 7mm”. No entanto, com a ajuda de uma árvore sintática, é possível ver que o sintagma verbal “is not seen...” nega seu sujeito, o sintagma nominal composto de outros dois sintagmas nominais, “The previously

identified isoechoic nodule on the right" e "larger than 7 mm". A abordagem híbrida foi aplicada em um conjunto de teste de 120 laudos, e obteve precisão de 98,6% e revocação de 99,8%.

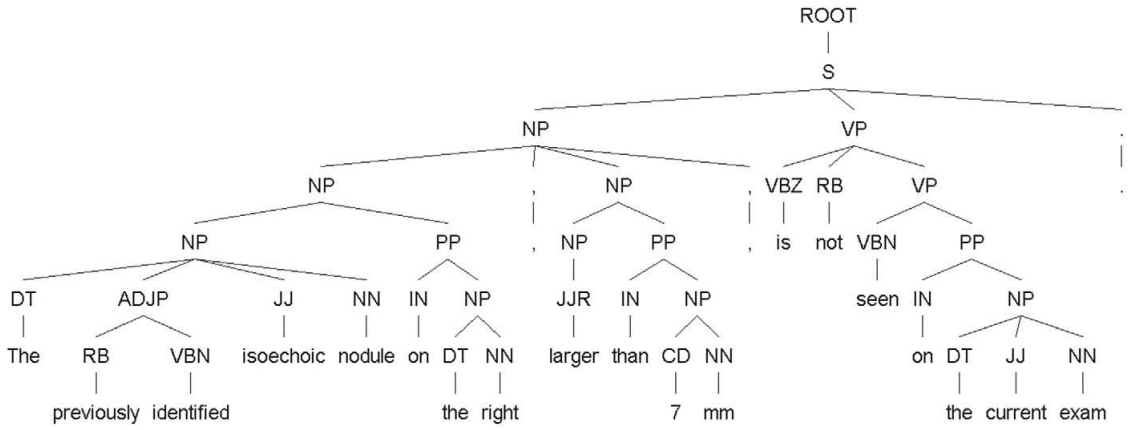

Figura 2.5: Um exemplo de negação onde o sintagma negado "The previously identified isoechoic nodule" está distante do sinal de negação "not" (HUANG; LOWE, , p. 5).

O trabalho de (ROKACH; ROMANO; MAIMON, 2008) apresentou um novo método de aprendizagem de expressões regulares para a identificação automática de contexto negativo em laudos clínicos. O processo de treinamento do classificador se divide em quatro etapas:

1. **Preparação do corpus:** normalização, indexação, e separação em sentenças;
2. **Aprendizado de padrões na forma de expressões regulares:** a criação automática de expressões regulares a partir do conjunto de treinamento. Dois algoritmos foram adaptados para a tarefa, o Longest Common Subsequence (LCS) e Teiresias;
3. **Seleção de padrões:** aplicação de heurísticas e técnicas de seleção de atributos para selecionar os melhores padrões;
4. **Treinamento do classificador:** o método usa árvores de decisão em forma de cascata, onde cada nodo de árvore é uma expressão regular.

O método obteve precisão de 94,6% e revocação 96,7%.

Em (GINDL; KAISER; MIKSCH, 2008) Gindl et al. desenvolveram o NegHunter, um algoritmo que classifica negações em diretrizes de prática clínica de acordo com tipos de negação levatados no trabalho. Da mesma forma que o NegEx, o NegHunter emprega uma lista de termos disparadores, porém com menos elementos. Cada termo disparador se classifica em uma das seguintes categorias: negação adverbial, negação intra-sintagma, negação preposicional, negação adjetiva, e negação verbal.Assim como em outros trabalhos já citados, achados

médicos da UMLS são mapeados no texto. Para o desenvolvimento do trabalho um conjunto de 18 diretrizes de prática clínica foram analizados, e para o teste outros 14 foram usados. O método obteve precisão de 67,49% e revocação de 83,51%.

Nassif et al. (NASSIF et al., 2009) desenvolveram um sistema para a recuperação de informação em mamogramas. Um módulo de detecção de negações faz parte desse sistema. Trata-se de um analisador léxico que procura por sinais de negação usando expressões regulares. Seguindo a abordagem de Gindl, as categorias de disparadores são a intra-sintagmas e adverbiais. O teste do sistema, realizado com um conjunto de teste de 100 mamogramas, obteve uma precisão de 97,7% e revocação de 95,5%.

Em (UZUNER; ZHANG; SIBANDA, 2009) os autores estudam duas abordagens para a classificação de asserções, o Extended NegEx (ENegEx) e o Statistical Assertion Classifier (StAC). A classificação de asserções consiste em determinar se um problema está presente, ausente, ou é incerto em um paciente, ou ainda se o problema é associado com outra pessoa que não o paciente (asserções de alter-associação). Os objetivos do estudo são:

1. estender pesquisas a detecção de negação e incerteza com a detecção de alter-associações;
2. determinar a contribuição do contexto léxico e sintático na classificação de asserções;
3. testar se uma abordagem baseada em aprendizado de máquina pode ser tão largamente aplicada quanto uma abordagem baseada em regras.

Os dados do estudo foram obtidos de três corpus: 48 sumários de alta do Beth Israel Deaconess Medical Center, 142 sumários de alta do corpus Challenge, e 1.954 laudos de radiologia do Computational Medicine Center. Os problemas médicos, para o propósito do estudo, se referem a doenças e sintomas, correspondentes aos tipos semânticos da UMLS “funções patológica”, “doenças”, “sintomas”, entre outros (UZUNER; ZHANG; SIBANDA, 2009). Após os problemas serem identificados manualmente nos corpus por dois estudantes, uma enfermeira e um estudante classificaram cada problema como presente, ausente, incerto, ou pertencente a outra pessoa.

Para o estudo do ENegEx, uma versão própria do algoritmo foi implementada, com a adição de duas listas de termos disparadores para a identificação de alter-associações, uma para termos que ocorrem antes de um problema, e outra para termos que ocorrem após. O StAC consiste de um modelo do tipo Support Vector Machine (SVM) treinado sobre uma base de vetores de características extraídos dos corpus. De acordo com o artigo, o modelo SVM foi escolhido por sua capacidade de lidar com um grande número de conjuntos de características, e habilidade

em achar soluções globalmente ótimas (UZUNER; ZHANG; SIBANDA, 2009). Os vetores de características são compostos de 8 palavras (4 anteriores e 4 posteriores ao problema), verbos anteriores ou posteriores ao problema, e duas ligações, extraídas de um parser de gramática de ligações.

O ENegEx obteve precisão de 93% e revocação de 74% na identificação de problemas ausentes, enquanto que StAC obteve precisão de 82% e revocação de 89%.

Harkema et al. desenvolveram em 2009 o ConText, um algoritmo capaz de determinar se condições clínicas mencionadas em um laudo são negadas, hipotéticas, históricas, ou experimentadas por alguém que não o paciente (HARKEMA et al., 2009). Ele deriva do NegEx, e o extende em alguns pontos: primeiro, o escopo dos termos disparadores, no caso geral é maior, partindo do termo disparador e indo até o fim da sentença ou um termo de terminação, o que vier primeiro. Além disso, há três definições alternativas de escopo para termos disparadores específicos. Segundo, além da negação, ConText identifica valores para outras duas propriedades contextuais, a temporalidade e o experimentador. Temporalidade se refere a que momento no tempo a condição ocorreu. Seu valor padrão é *recente*, e seus outros possíveis valores são *histórico*—algo que ocorreu duas semanas antes da realização do exame– e *hipotético*. A propriedade contextual experimentador descreve quem possui a condição, o paciente ou outra pessoa. Por exemplo, na sentença "O pai do paciente possui um histórico de CHF", o valor do experimentador para a condição CHF é "outro".

O ConText foi testado com 120 laudos, coletados de uma base de seis tipos de laudos: radiologia, departamento de emergência, patologia cirúrgica, ecocardiograma, procedimentos operativos, e sumários de alta. O sistema obteve precisão de 94% e revocação de 92%.

No trabalho de Skeppstedt (SKEPPSTEDT, 2010) o algoritmo NegEx foi adaptado para a língua sueca. Os termos disparadores foram traduzidos para o sueco com a ajuda de um dicionário inglês-sueco e do site Google Translate, somando um total de 148 frases de negação traduzidas. Essa adaptação foi testada em um conjunto de 436 laudos médicos em sueco. Os resultados mostraram uma precisão de 70% e revocação de 81%.

3 *Metodologia*

Neste capítulo é exposta a metodologia utilizada para implementar e comparar os dois sistemas de extração de termos negados. Na seção ”Ferramentas”são descritas as ferramentas utilizadas para a realização deste projeto. A seção ”Pré-processamento”explica como os laudos são tratados para a retirada de informações indesejadas, antes de serem processados pelos algoritmos. Nas seções ”NegEx”e ”Algoritmo baseado em memória”é apresentada a implementação dos respectivos algoritmos. Em ”Método de Avaliação”é mostrado como foram construídas as bases de treinamento e teste, e quais métricas foram utilizadas para comparar os dois algoritmos.

3.1 Ferramentas

3.1.1 Apache Lucene

Apache Lucene é uma biblioteca de recuperação de informações escalável e de alta performance (MCCANDLESS; HATCHER; GOSPODNETIC, 2010) que permite adicionar recursos de busca em aplicações. Ele permite indexar e pesquisar qualquer dado que possa ser convertido para formato texto É um projeto maduro e de código livre, e faz parte Apache Software Foundation, licenciado sob a Apache Software License. Apesar de ser desenvolvido na linguagem Java, existem bindings e versões portadas para Perl, Python, Ruby, C/C++, PHP, e C#.

3.1.2 Apache Solr

Solr é um servidor de buscas de código aberto baseado no Lucene (APACHE. . . , 2011).

Solr é escrito em Java e roda como um servidor de buscas dentro de um contêiner de Servlets Java, tal como o Tomcat. Solr utiliza em seu núcleo a biblioteca Lucene para as funções de indexação e pesquisa. Além das funcionalidades do Lucene, ele oferece outras, tais como busca distribuída, navegação facetada, e integração com banco de dados. Solr se comunica com clientes através de APIs REST HTPP/XML e JSON, o que permite uma integração com praticamente qualquer linguagem de programação.

3.2 Pré-processamento

3.2.1 Normalização

O objetivo desta primeira etapa é a normalização dos laudos e a sua separação em sentenças. A normalização consiste na eliminação informação indesejada, como tags html, e correção de erros de codificação. Vários laudos da base de dados contém erros de codificação, onde o texto usando uma codificação é interpretado erroneamente como tendo outra codificação. Por exemplo, letras com sinais diacríticos codificadas em utf-8, como 'ã', quando interpretado em latin-1, transforma-se em 'Ã£'. Neste trabalho a correção é feita para caracteres acentuados. Os laudos analisados contém elementos de linguagem html, como quebra de linha (
), parágrafo (<p >), e caracteres codificados como character entity references (p. ex. **ã** representando **ã**). Verificou-se também que muitas sentenças não possuem ponto final. Nesses casos o elemento
funciona como ponto final, separando as sentenças. No entanto este padrão é somente uma heurística que usamos para separar sentenças; não se pode afirmar que ele valha para toda a base.

3.2.2 Separação em Sentenças

Após a divisão dos laudos pelas tags
, ainda há a possibilidade de haver mais de uma sentença entre tags. Uma primeira tentativa de dividir o texto em sentenças foi utilizar um segmentador de sentenças existente no pacote NLTK, o Punkt. Ele utiliza um algoritmo de aprendizado não-supervisionado, pré-treinado no corpus CETENFolha, que contém 321.000 tokens. Analisando uma amostra das sentenças segmentadas percebeu-se que houve muitos erros, como a divisão de sentenças no meio. Isso ocorreu por causa de pontuações que não são usadas para marcar o fim de sentença, como as usadas em abreviações. As abreviações usadas nos laudos não fazem parte do domínio do do corpus de treinamento, e portanto não foram previstas pelo segmentador.

Como alternativa foi criado um tokenizador de palavras baseado em expressões regulares que distingue entre pontuações usadas em abreviações e pontuações de fim de frase. Para sua criação foi compilada uma lista de abreviações utilizadas nos laudos de tomografia e de ultrassom, que foram incluídas na expressão regular:

```
self.token_pattern = re.compile(r""" \d+[.,]\w+ |
                                     \( |
                                     \) |
```

```
e/ou |
\w+(-\w+)+ |
obst\. | # ultrasound abbreviations
apres\. |
vol\. |
c\. |
g\. |
ii?\. |
x\. |
rep\. | # ecg abbrev.
repol\. |
obs\. |
\w+ |
\S+""", re.VERBOSE | re.IGNORECASE)
```

Dessa maneira quando ocorre um token ponto tem-se certeza de que é o fim de uma sentença.

3.2.3 Indexação dos achados

Os achados foram indexados com os descritores do DeCS. Pelo fato de o escopo dos descritores ser amplo, abrangendo variados conceitos da área da saúde, foi utilizado um subconjunto de descritores, consistindo das categorias B e C (organismos e doenças). Dentro do DeCS, estas categorias são as que mais próximo estão da definição de achados.

A indexação foi implementada utilizando o servidor de buscas Solr e o motor de buscas Lucene. Os descritores do DeCS são compostos de uma ou mais palavras, e por isso uma indexação baseada em palavras não é capaz de identificar todos os descritores do DeCS. Por conta disso, a indexação foi dividida em duas etapas—a construção do índice dos laudos e a procura por descritores DeCS nos índices. No arquivo `schema.xml` definimos os campos do documento—`id` e `laudo` neste estudo—e para cada campo definimos seu *tipo*, se ele é indexado, armazenado no Solr, entre outras configurações. A Figura 3.1 mostra a seção fields do arquivo `schema.xml` neste trabalho.

Na seção `types` (Figura 3.2) definimos o tipo do campo que armazena o texto do laudo. É nesta seção que definimos o analisador de cada campo. Duas cadeias de analisadores foram

```
<fields>

  <field name="id" type="string" indexed="true" stored="true" required="true" />
  <field name="laudo" type="laudoType" indexed="true" stored="true" termVectors="true" termPositions="true" termOffsets="true" />

</fields>
```

Figura 3.1: Parte do arquivo schema.xml onde os são especificados os campos a serem indexados

definidas, uma para a etapa de indexação e outra para a etapa de busca. A primeira consiste nas seguintes classes:

HTMLStripCharFilterFactory: retira elementos XML e character entities do stream de entrada.

WhitespaceTokenizerFactory: cria tokens, separando-os por espaço.

StopFilterFactory: descarta palavras comuns—como 'ele', 'o', 'este'—irrelevantes para a indexação.

WordDelimiterFilterFactory: divide palavras em sub-palavras, de acordo com certas regras, como a divisão de palavras hifenizadas.

LowerCaseFilterFactory: transforma todas as letras em minúsculas.

BrazilianStemFilterFactory: extrai o radical das palavras (`stemming`).

A cadeia de analisadores é um pouco diferente, excluindo o `HTMLStripCharFilterFactory` — uma pesquisa não possui elementos XML — e incluindo o `SynonymFilterFactory`, para a expansão da pesquisa com sinônimos das palavras. O arquivo com os sinônimos foi obtido do Thesaurus Eletrônico para o Português do Brasil (MAZIERO et al., 2008). DataImportHandler foi o plugin usado para poder indexar o banco de dados.

A API do Lucene foi utilizada para fazer um programa que, para cada descritor, procura no índice documentos que o contenham, e o destaque no laudo com tags `<achado>`. Uma busca, por exemplo, pelo descritor "cisto ovariano" retornaria o laudo "Ovário esquerdo de topografia e forma normais, porém com volume aumentado e com presença de imagem anecóica e homogênea, medindo 4,0 x 3,5 x 4,1 cm (volume de 30 cm3), compatível com <achado >cisto ovariano </achado >." O diagrama da Figura 3.3 mostra esse processo.

```
<fieldType name="laudoType" class="solr.TextField" positionIncrementGap="100" autoGeneratePhraseQueries="true">
  <analyzer type="index">
    <charFilter class="solr.HTMLStripCharFilterFactory"/>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords-br.txt" ignoreCase="true"/>
    <filter class="solr.WordDelimiterFilterFactory" catenateWords="1" catenateNumbers="1"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.BrazilianStemFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.SynonymFilterFactory" synonyms="synonyms-decs-lower.txt" ignoreCase="true" expand="true"/>
    <filter class="solr.StopFilterFactory" words="stopwords-br.txt" ignoreCase="true"/>
    <filter class="solr.WordDelimiterFilterFactory" catenateWords="1" catenateNumbers="1"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.BrazilianStemFilterFactory"/>
  </analyzer>
</fieldType>
```

Figura 3.2: Seção fields do arquivo schema.xml.

3.3 Adaptação do NegEx

O algoritmo NegEx foi escolhido para ser adaptado por ser o único detector de negações que possui uma implementação de código aberto. A princípio seria usada uma implementação na linguagem Python hospedada em (NEGEX, 2011), mas após a realização de alguns testes essa idéia foi descartada devido a presença de vários bugs no código. Por conta disso uma implementação própria foi criada e hospedada no site de hospedagens de projetos **Github** (FC-BERTOLDI. . . , 2011).

3.3.1 Tradução das frases de negação

Os termos disparadores para o português foram obtidos traduzindo os termos em inglês do NegEx original, localizados em (NEGEXTERMS, 2011). As traduções foram feitas com o auxílio de um dicionário inglês-português e do Google Translate (GOOGLE. . . , 2011). Além dos termos traduzidos, novos termos foram obtidos através de uma pesquisa nas bases de tomografia e ultrassom por palavras que indicam negação e possibilidade, como ”não”, ”sem”, ”ausência”, ”provável”, e pela leitura de uma amostra de laudos.

3.3.2 O algoritmo

NegEx usa expressões regulares e 5 listas de frases disparadoras para determinar se um achado médico está negado ou é hipotético em uma sentença. A primeira lista, frases pseudo-negadas, são frases que parecem identificar negações, mas que de fato não são identificadores confiáveis de negação. Eles são usados para descartar falsos positivos. A segunda e terceira lista, pré-negação e pós-negação, são frases que ocorrem antes e depois dos termos que estão

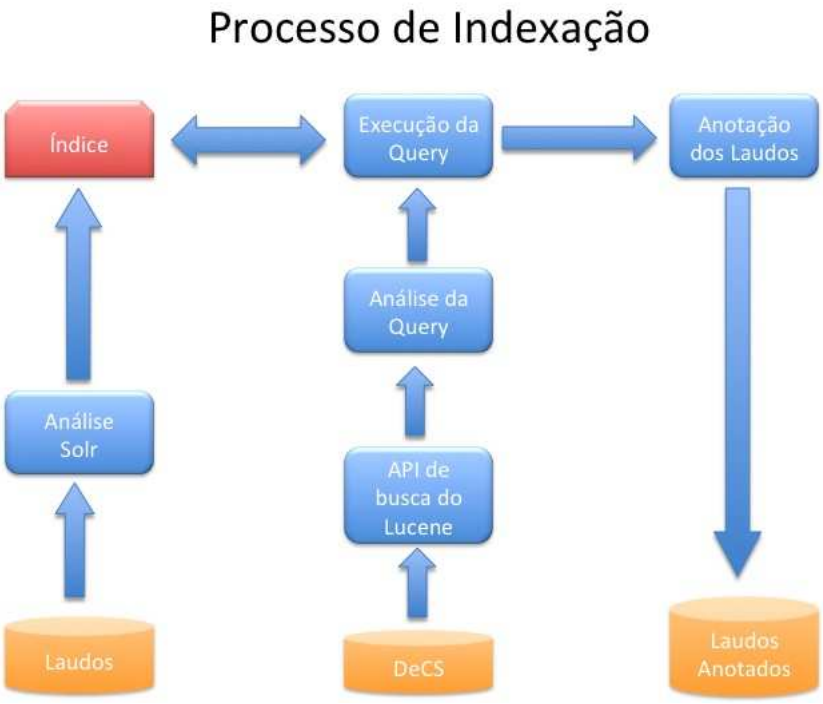

Figura 3.3: Diagrama do processo de indexação.

negando, respectivamente. A quarta e quinta, pré-possibilidade e pós-possibilidade, são frases indicadoras de linguagem especulativa, como p. ex. “provável” e “discutível”. As expressões regulares determinam o escopo dos termos disparadores dentro da sentença. NegEx usa as seguintes expressões regulares:

```
<frase de pré-negação ou pré-possibilidade> <$1> <achado indexado>

<achado indexado> <$2> <frase de pós-negação ou pós-possibilidade>
```

O símbolo $ pode representar um número específico ou não-específico de palavras. No caso da primeira expressão regular, $1 representa um número não-específico de palavras. Na segunda expressão regular, cinco palavras. Há uma sexta lista que consiste de termos terminadores de escopo. Termos indexados que estão dentro do escopo da frase disparadora são identificados como negados.

3.4 Aprendizado Baseado em Memória

A abordagem usando o aprendizado baseado em memória consiste em classificar os achados de uma sentença em *afirmativo*, *negativo*, ou *hipotético*.

Algorithm 1 Algoritmo NegEx

Input: termos_disp: uma lista de frases disparadoras
laudo: string de texto contendo um laudo
Output: laudo dividido em sentenças anotadas com os termos negados e hipotéticos identificados

normalizar(*laudo*) {eliminação de tags html e correção de erros de codificação}
sentencas ← *separar*(*laudo*) {tokenizador de sentenças divide o laudo em várias sentenças}
for all *sent* in *sentencas* **do**
  tokens ← *list*()
  tokenizar_achados(*tokens*) {tokenizar os achados}
  neg_tag(*tokens*) {tokenizar as frases disparadoras}
  tokenizar_palavras(*tokens*) {tokenizar as palavras que restaram}
  $frases\_disparadoras \leftarrow list()$
  achados ← *list*()
  for all *token* in *tokens* **do**
    if *token* é uma frase disparadora de pré-negação ou pré-possibilidade **then**
      definir escopo da frase disparadora, adicionando tokens a frente até que encontre uma das seguintes condições:
      -fim da sentença
      -termo de terminação
      -frase disparadora de negação ou pseudo-negação
      $frases\_disparadoras.adicionar(token)$
    else if *token* é uma frase disparadora de pós-negação o pós-possibilidade **then**
      definir escopo da frase disparadora, adicionando tokens antecedentes até que encontre uma das seguintes condições:
      -início da sentença
      -6 tokens já foram adicionados
      -termo de terminação
      $frases\_disparadoras.adicionar(token)$
    else if *token* é um achado **then**
      classificar o achado como 'afirmativo' por padrão
      achados.adicionar(*token*)
    end if
  end for
  $frases\_disparadoras.ordenar()$ {ordenar $frases\_disparadoras$ pela ordem de precedência. A ordem ascendente de precedência das frases disparadoras é: pós-possibilidade, pré-possibilidade, pós-negação e pré-negação.}
  for all $frase$ in $frases\_disparadoras$ **do**
    $frase\_offset \leftarrow frase.offset()$ {offset da frase disparadora na sentença, representado por uma tupla de tamanho 2}
    for all *achado* in *achados* **do**
      $achado\_offset \leftarrow achado.offset()$ {offset do achado na sentença, representado por uma tupla de tamanho 2}
      if $achado\_offset[0] >= frase\_offset[0]$ and $achado\_offset[1] <= frase\_offset[1]$ **then**
        {achado no escopo da frase}
        if $frase$ é pré-negação ou pós-negação **then**
          classificar *achado* como 'negativo'
        else if $frase$ é pré-possibilidade ou pós-possibilidade **then**
          classificar *achado* como 'especulativo'
        end if
      end if

Assim como o NegEx, o texto que servirá de entrada para o algoritmo de ABM precisa ser normalizado, isto é, ele passará pelas mesmas etapas de correção de erros de codificação, separação em sentenças, tokenização, e indexação. O resultado desse processamento é uma lista de frases tokenizadas. A princípio é possível definir o modelo do vetor que constituirá a base de casos como tendo um tamanho arbitrariamente grande (para poder abarcar frases de tamanho arbitrário) e tendo palavras como valores dos atributos do vetor. Esse modelo, no entanto, sofre de um problema pervasivo em PLN: o *problema dos dados esparsos*. Quanto maior o tamanho do vetor e número de possíveis valores de atributos, maior é a probabilidade de que uma instância qualquer não ocorra na base de casos. Devido a esse problema, é necessário reduzir a dimensão das frases e o número de valores possíveis; uma forma comum de se alcançar esse objetivo em PLN é utilizando um etiquetador morfossintático.

Para etiquetar as frases dos laudos foi utilizado o Memory-Based Part of Speech Tagger-Generator(Mbt) (DAELEMANS et al., 1996). O Mbt funciona tanto como um gerador de etiquetadores e como um etiquetador. O gerador de etiquetadores toma como entrada um conjunto de treinamento, composto por frases anotadas com a categoria morfossintática de cada palavra e, usando o algoritmo de ABM do TiMBL, gera um etiquetador. Para gerar o etiquetador foi utilizado o córpus Floresta (AFONSO et al., 2002), composto de 221.000 palavras e 9.266 frases compiladas dos jornais Folha de São Paulo e Público, anotadas com categorias morfossintáticas.

Após etiquetar as frases, foram identificadas frases de negação que aparecem antes de achados (como p. ex. *não identificamos*, *não se observou*), depois de achados (*ausente*, *desaparecido*), frases hipotéticas que aparecem antes e depois de achados (*não se pode afirmar*, *não sendo possível afastar*), usando a lista de termos disparadores do NegEx. As etiquetas correspondentes as palavras dos termos disparadores identificados foram substituídas pelas seguintes etiquetas: `<PREN>` para pré-negação, `<POST>` para pós-negação, `<PREP>` para pré-hipotético, e `<POSP>` para pós-hipotético.

Após essa etapa, um vetor de atributos é criado para cada achado na frase. Foram selecionadas 8 etiquetas adjacentes ao achado (4 antes do achado e 4 depois). Caso em alguma posição não haja etiqueta—o achado pode estar no começo da frase e assim, não há etiquetas antes do achado—é colocado em sua posição a etiqueta NULL. Por exemplo, a frase:

```
Presença de <achado tipo="afirmativo">feto único</achado>, em situação
longitudinal, apresentação pélvica e dorso à direita.
```

gera a seguinte instância:

```
NULL,NULL,n,prp,FIND,comma,prp,n,adj
```

3.5 Criação do padrão áureo

Para que os algoritmos sejam comparados foi necessário a criação de um conjunto anotado e validado por especialistas, chamado de "padrão ouro". Para o desenvolvimento desse estudo foi implementada uma ferramenta simples para que os especialistas pudessem fazer a validação. O desenvolvimento desse método de avaliação foi baseando nos estudos de Chapman (CHAPMAN, 2001, p. 4). 500 sentenças foram selecionadas, indexadas com termos do DeCS, e repassou-se subconjuntos sobrepostos dessas sentenças a 3 especialistas da área, que marcavam cada termo como

Presente o termo destacado está presente na sentença;

Ausente o termo destacados está ausente na sentença;

Hipotético há uma possibilidade de o termo existir no paciente, mas não há uma confirmação. Trata-se de uma linguagem especulativa.

Ambíguo não estava claro para o especialista se o termo está presente ou ausente.

Não-achado o termo não é um achado.

Cada especialista anotou 200 das 500 sentenças, sendo que 100 sentenças foram sobrepostas e avaliadas pelos 3 especialistas para determinar a melhor precisão das respostas. Os avaliadores 1 e 2 julgaram 50 sentenças sobrepostas, e os avaliadores 2 e 3 julgaram um conjunto separado de 50 sentenças sobrepostas, como pode ser visto na Figura 3.4. Se dois avaliadores não concordarem a respeito de um termo, esse termo é marcado como ambíguo, e a sentença onde ocorreu o termo é retirada do conjunto.

A Figura 3.5 mostra a interface de validação do especialista para análise das sentenças recuperadas pelo sistema, desenvolvida na linguaguem PHP. O objetivo dessa tela é agilizar o processo de validação por parte do especialista.

3.6 Conjuntos de teste e treinamento

A base de laudos de ultrassom é composta de 26862 laudos anonimizados que vem sendo alimentada desde 2005.

De uma base de 26862 laudos de ultrassonografia anonimizados, 2000 foram selecionados aleatoriamente. Em seguida, as sentenças dos laudos foram extraídas e indexadas com os termos

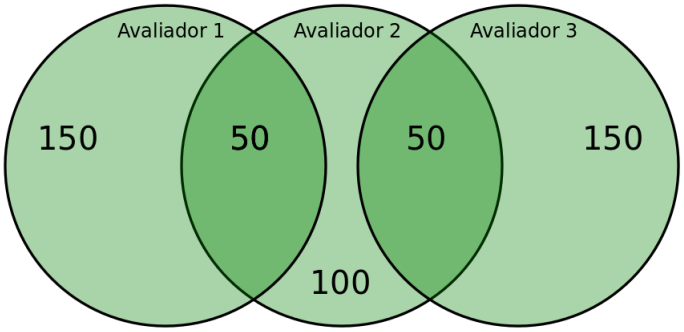

Figura 3.4: Diagrama de Venn que mostra a distribuição de sentenças entre os avaliadores.

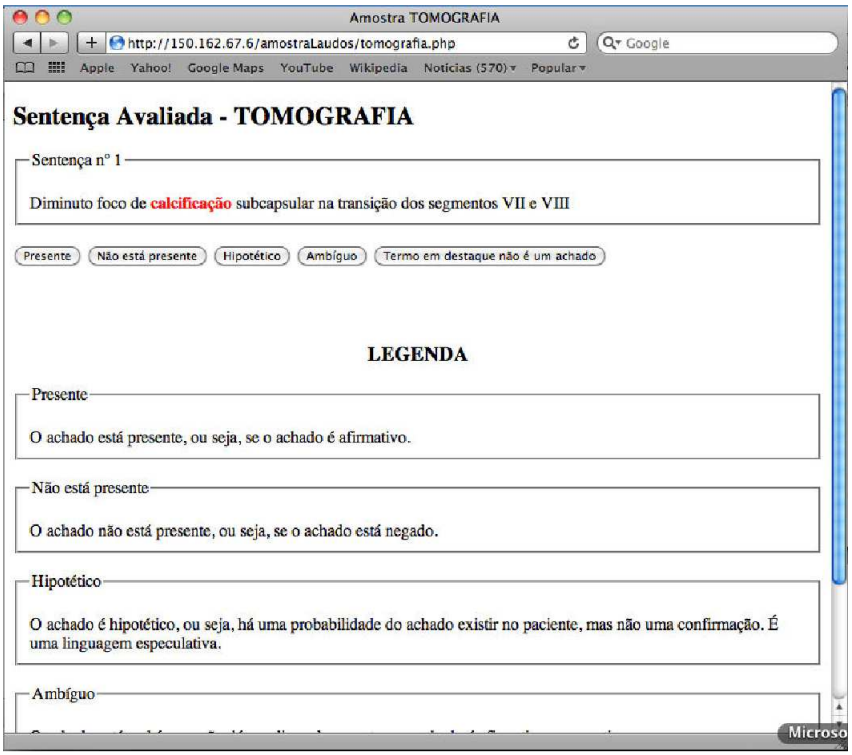

Figura 3.5: Interface para validação de especialistas em tomografia(ANDRADE, 2011). O sistema foi posteriormente adaptado para a validação de laudos de ultrassom.

do DeCS. Das sentenças que continham termos do DeCS, 500 sentenças foram aleatoriamente selecionadas, e divididas em dois grupos. O primeiro grupo, chamado de **conjunto de treinamento**, consiste de 350 sentenças lidas manualmente para se extrair termos disparadores de negação e hipótese. Ele também foi usado pelo ABM para construir a base de casos. O segundo grupo, o **conjunto de treinamento**, consiste das 150 sentenças restantes, que serão usadas para comparar o desempenho dos dois algoritmos.

3.7 Método de Avaliação e Métricas

O método de avaliação foi baseado no trabalho de Chapman em (HARKEMA et al., 2009, p. 5). Para cada achado classificado por um dos algoritmos, seu resultado é um *verdadeiro positivo* (VP) se o algoritmo muda corretamente a categoria padrão do achado; um *verdadeiro negativo* (VN) se corretamente não muda a categoria; um *falso positivo* (FP) quando a categoria padrão é mudada incorretamente; e um *falso negativo* (FN) quando a categoria padrão incorretamente não é mudada.

A partir dos valores de VP, VN, FP, e FN, são obtidas as seguintes métricas:

- **precisão** $= \frac{VP}{VP+FP}$.
- **revocação** $= \frac{VP}{VP+FN}$, também conhecida como recall.
- **medida-F** $= \frac{2\times p\times r}{p+r}$, onde p é a precisão e r é a revocação, é a média harmônica entre a precisão e revocação.
- **kappa** $\frac{P(a)-P(e)}{1-P(e)}$, onde $P(a)$ é a concordância relativa entre os avaliadores, e $P(e)$ é a probabilidade hipotética de concordância por acaso.

4 Resultados

4.1 Taxa de concordância entre anotadores

Das 500 sentenças anotadas, 111 contêm termos do DeCs que não foram considerados como achados pelos avaliadores (Tabela 4.1), e 5 sentenças foram consideradas ambíguas por pelo menos um dos avaliadores. Outras 53 foram consideradas ambíguas por discordância entre os avaliadores.

Termo DeCS	nº de ocorrências
Vesícula	86
Lobos	12
Rena	5
Doença	2
Cicatriz	2
Cálculos	1
Hipertrofia	1
Dor	1
Pielonefrite	1

Tabela 4.1: Termos do DeCS que não foram considerados como achados pelos avaliadores.

As sentenças anotadas em conjunto pelos avaliadores 1 e 2 tiveram o valor de Kappa de 0,2, e as sentenças anotadas pelos avaliadores 2 e tiveram um valor de Kappa de 0,15. Fazendo-se a média, temos $k_{medio} = 0,175$. Este é um valor muito abaixo do valor considerado de boa confiabilidade ($k > 0,8$, (KRIPPENDORFF, 1980 apud JURAFSKY et al., 2000, p. 313)).

A Tabela 4.2 contém as métricas de cada modelo testado. Em negrito estão destacados os melhores valores de medida-F obtidos. Pode-se notar que o Aprendizado Baseado em Memória obteve os melhores resultados, particularmente usando a métrica de distância MVDM e três vizinhos, com medida-F de 77%. O NegEx obteve um resultado muito semelhante, com uma medida-F de 75%.

As Tabelas 4.3, 4.4, e 4.5 mostram a matriz de confusão dos três melhores modelos: M.ig.k3, M.gr.k3, M.x2.k3, respectivamente. As três matrizes de confusão são idênticas, o que mostra que, com a métrica de distância MVDM, a escolha de peso é praticamente indiferente.

Modelo	**VP**	**VN**	**FP**	**FN**	**Precisão**	**Revocação**	**Medida-F**	**Acurácia**
NegEx	50	21	22	12	0,69	0,81	0,75	0,68
O.nw.k1	47	18	32	7	0,59	0,87	0,71	0,62
O.nw.k3	48	16	35	5	0,58	0,91	0,71	0,61
O.nw.k5	51	14	38	1	0,57	0,98	0,72	0,62
O.ig.k1	47	18	32	7	0,59	0,87	0,71	0,62
O.ig.k3	49	19	26	10	0,65	0,83	0,73	0,65
O.ig.k5	47	17	31	9	0,6	0,84	0,7	0,62
O.gr.k1	47	17	33	7	0,59	0,87	0,7	0,62
O.gr.k3	49	19	27	9	0,64	0,84	0,73	0,65
O.gr.k5	48	17	31	8	0,61	0,86	0,71	0,62
O.x2.k1	47	18	32	7	0,59	0,87	0,7	0,62
O.x2,k3	49	19	26	10	0,65	0,83	0,73	0,65
O.x2.k5	47	16	32	9	0,59	0,84	0,7	0,6
M.nw.k1	47	18	31	8	0,6	0,85	0,7	0,62
M.nw.k3	52	19	23	10	0,69	0,84	0,76	0,68
M.nw.k5	51	19	24	10	0,68	0,84	0,75	0,67
M.ig.k1	48	16	34	6	0,59	0,89	0,7	0,62
M.ig.k3	53	19	23	9	0,7	0,85	**0,77**	0,69
M.ig.k5	49	19	26	10	0,65	0,83	0,73	0,65
M.gr.k1	47	16	33	8	0,59	0,85	0,7	0,6
M.gr.k3	53	19	23	9	0,7	0,85	**0,77**	0,69
M.gr.k5	49	19	25	11	0,66	0,82	0,73	0,65
M.x2.k1	48	16	34	6	0,59	0,88	0,7	0,62
M.x2.k3	53	19	23	9	0,7	0,85	**0,77**	0,69
M.x2.k5	49	19	26	10	0,65	0,83	0,73	0,65

Tabela 4.2: Comparação entre os modelos, mostrando os valores de verdadeiro positivo (VP), verdadeiro negativo (VN), falso positivo (FP), falso negativo (FN), precisão, revocação, medida-f, e acurácia para cada um dos modelos testados.

		Algoritmo		
		Afirmativo	Hipotético	Negativo
Especialista	Afirmativo	19	4	0
	Hipotético	7	11	15
	Negativo	2	4	42

Tabela 4.3: Matriz de confusão do ABM com métrica de distância MVDM, peso ganho de informação, e 3 vizinhos mais próximos.

		Algoritmo		
		Afirmativo	Hipotético	Negativo
Especialista	Afirmativo	19	4	0
	Hipotético	7	11	15
	Negativo	2	4	42

Tabela 4.4: Matriz de confusão do ABM com métrica de distância MVDM, peso taxa de ganho, e 3 vizinhos mais próximos.

		Algoritmo		
		Afirmativo	Hipotético	Negativo
Especialista	Afirmativo	19	4	0
	Hipotético	7	11	15
	Negativo	2	4	42

Tabela 4.5: Matriz de confusão do ABM com métrica de distância MVDM, peso teste qui-quadrado, e 3 vizinhos mais próximos.

A Tabela 4.6 mostra a matriz de confusão do NegEx, e a Tabela 4.7 a precisão associada com as frases de disparo mais comuns. É possível ver que a frase "sem sinais", a mais comum, gerou um grande número de falsos positivos. Isso se deve à falta de concordância entre os anotadores com respeito a achados do tipo "sem sinais de ¡achado¿", onde alguns anotadores escolheram a opção negativa, e outros a opção hipotética, ou mesmo mudavam de idéia quanto a classificação durante a anotação.

		Algoritmo		
		Afirmativo	Hipotético	Negativo
Especialista	Afirmativo	21	2	0
	Hipotético	11	6	17
	Negativo	1	3	44

Tabela 4.6: Matriz de confusão do NegEx.

Frase_Disparadora	**VP**	**FP**	**Precisão**	**Nº_de_Ocorrências**
sem_sinais	27	11	0.71	38
sem_evidência	9	5	0.64	14
Ausência	5	0	1.00	5
possam	2	3	0.40	5
podem	2	0	1.00	2
Não_há	1	1	0.50	2
sugestiva	1	0	1.00	1
Não_há_evidência	1	0	1.00	1
pode	0	1	0.00	1
sugestivos	1	0	1.00	1
sugestivo	0	1	0.00	1
sem	1	0	1.00	1

Tabela 4.7: Frases de disparo identificadas pelo NegEx. Para cada frase de disparo é mostrado o número de verdadeiros e falsos positivos (VP, FP) associados a elas, a precisão, e o seu número de ocorrências no conjunto de testes.

Observando a Tabela 4.2 é possível notar que todos os modelos tiveram bons níveis de revocação, ao passo que a precisão dos modelos foi bastante ruim. A avaliação dos resultados foi bastante prejudicada pelo baixo grau de concordância entre os anotadores. Em muitos

casos (como p. ex. achados próximos da frase “sem sinais”) a base de treinamento possui classificações diferentes para instâncias parecidas ou mesmo iguais. Nessas situações o ABM obteve melhores resultados que o NegEx, pois ele escolhe a classificação mais frequente entre os vizinhos—daí a superioridade dos modelos que avaliam um maior número de vizinhos—enquanto que o NegEx emprega regras fixas, muitas vezes escolhendo a classificação menos frequente entre casos semelhantes.

4.2 Trabalhos Futuros

- Usar heurísticas e técnicas para a seleção dos atributos em ABM.
- Usar diferentes métricas de distância, como Mahalanobis.
- Usar outros modelos de aprendizado, como Support Vector Machines (SVM), Redes Neurais, Árvores de Decisão, Ensembles.
- Ampliar o Córpus.

5 *Conclusão*

Neste trabalho foram apresentadas duas abordagens para a detecção de termos negados em laudos médicos em português, uma baseada em um algoritmo existente, o NegEx, e outra baseada em Aprendizado Baseado em Memória. O trabalho mostrou que as duas abordagens possuem performances semelhantes. Apesar da performance de ambos os algoritmos não se enquadrar entre as melhores no estado-da-arte, este trabalho não representa suas qualidades intrísecas, dado que o experimento foi bastante prejudicado pelo baixo grau de concordância entre os anotadores do corpus usado no trabalho.

Referências Bibliográficas

AFONSO, S. et al. Floresta sintá (c) tica": a treebank for portuguese. In: CITESEER. *Proc. of the Third Intern. Conf. on Language Resources and Evaluation (LREC)*. [S.l.], 2002. p. 1698–1703.

AHA, D. W. Lazy learning: Special issue editorial. *Artificial Intelligence Review*, v. 11, 1997.

ANDRADE, R. *Um modelo para recuperação e comunicação do conhecimento em documentos médicos*. Tese (Doutorado), 2011.

ANDRADE, R. et al. An approach to semantic search in medical databases. In: . [S.l.: s.n.], 2009.

APACHE Solr. 2011. Disponível em: <http://lucene.apache.org/solr/>. Acesso em: 10 mar. 2011.

ATHENIKOS, S. J.; HAN, H. Biomedical question answering: A survey. *Computer methods and programs in biomedicine*, Elsevier, 2009. Disponível em: <http://www.cmpbjournal.com/article/S0169-2607%2809%2900287-9/abstract>.

BIREME. 2011. Disponível em: <http://regional.bvsalud.org/local/Site/bireme/P/descricao.htm>. Acesso em: 7 mai. 2011.

CHAPMAN, W. A simple algorithm for identifying negated findings and diseases in discharge summaries. *Journal of Biomedical Informatics*, Center for Biomedical Informatics, 8084 Forbes Tower, University of Pittsburgh, Pittsburgh, PA 15213, USA. chapman@cbmi.upmc.edu, v. 34, n. 5, p. 301–310, October 2001. ISSN 15320464. Disponível em: <http://dx.doi.org/10.1006/jbin.2001.1029>.

CHU, D.; DOWLING, J. N.; CHAPMAN, W. W. Evaluating the effectiveness of four contextual features in classifying annotated clinical conditions in emergency department reports. *AMIA ... Annual Symposium proceedings / AMIA Symposium. AMIA Symposium*, p. 141–145, 2006. ISSN 1942-597X. Disponível em: <http://view.ncbi.nlm.nih.gov/pubmed/17238319>.

COST, S.; SALZBERG, S. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine learning*, Springer, v. 10, n. 1, p. 57–78, 1993.

DAELEMANS, W.; BOSCH, A. van den. *Memory-Based Language Processing*. Cambridge, UK: Cambridge University Press, 2005. Hardcover. (Studies in Natural Language Processing). ISBN 0521808901. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0521808901>.

DAELEMANS, W. et al. Mbt: A memory-based part of speech tagger-generator. In: *PROC. OF FOURTH WORKSHOP ON VERY LARGE CORPORA*. [s.n.], 1996. p. 14–27. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.2610>.

DAELEMANS, W. et al. *TiMBL: Tilburg Memory-Based Learner*. [S.l.], 2010.

DESCRITORES em Ciências da Saúde. 2011. Disponível em: <http://decs.bvs.br/P/decsweb2011.htm>. Acesso em: 7 mai. 2011.

FCBERTOLDI Github. 2011. Disponível em: <https://github.com/fcbertoldi/tcc>. Acesso em: 7 mar. 2011.

GINDL, S.; KAISER, K.; MIKSCH, S. Syntactical negation detection in clinical practice guidelines. *Studies in health technology and informatics*, v. 136, p. 187–192, 2008. ISSN 0926-9630. Disponível em: <http://view.ncbi.nlm.nih.gov/pubmed/18487729>.

GOOGLE Translate. 2011. Disponível em: <http://translate.google.com>. Acesso em: 7 mar. 2011.

GORYACHEV, S. et al. *Implementation and evaluation of four different methods of negation detection*. [S.l.], 2006.

HALTEREN, H. V.; DAELEMANS, W.; ZAVREL, J. Improving accuracy in word class tagging through the combination of machine learning systems. *Computational linguistics*, MIT Press, v. 27, n. 2, p. 199–229, 2001. Disponível em: <http://portal.acm.org/citation.cfm?id=972669&dl=>.

HAMMING distance 3 bit binary — Wikipedia, The Free Encyclopedia. 2006. Disponível em: <http://en.wikipedia.org/wiki/File:Hamming_distance_3_bit_binary.svg>. Acesso em: 14 jul. 2011.

HARKEMA, H. et al. Context: an algorithm for determining negation, experiencer, and temporal status from clinical reports. *Journal of biomedical informatics*, Elsevier Science, San Diego, USA, v. 42, n. 5, p. 839–851, October 2009. ISSN 1532-0480. Disponível em: <http://dx.doi.org/10.1016/j.jbi.2009.05.002>.

HUANG, Y.; LOWE, H. J. A novel hybrid approach to automated negation detection in clinical radiology reports. Disponível em: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2244882/>.

JURAFSKY, D. et al. Speech and language processing. In: ____. [S.l.]: Prentice Hall New York, 2000. cap. 1, p. 4.

KITCHENHAM, B. *Procedures for performing systematic reviews*. [S.l.], 2004. v. 33.

KLEIN, D.; MANNING, C. D. Accurate unlexicalized parsing. In: *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*. Morristown, NJ, USA: Association for Computational Linguistics, 2003. p. 423–430. Disponível em: <http://dx.doi.org/10.3115/1075096.1075150>.

KRIPPENDORFF, K. *Content analysis: An introduction to its methodology*. [S.l.]: Sage Publications, Inc, 1980.

MANNING, C. D.; SCHUETZE, H. *Foundations of Statistical Natural Language Processing*. 1. ed. [S.l.]: The MIT Press, 1999. Hardcover. ISBN 0262133601.

MAZIERO, E. G. et al. A base de dados lexical e a interface web do TeP 2.0: thesaurus eletrônico para o Português do Brasil. In: ACM. *Companion Proceedings of the XIV Brazilian Symposium on Multimedia and the Web*. [S.l.], 2008. p. 390–392.

MCCANDLESS, M.; HATCHER, E.; GOSPODNETIC, O. *Lucene in Action*. 2. ed. Manning Publications, 2010. Paperback. ISBN 1933988177. Disponível em: <http://www.worldcat.org/isbn/1933988177>.

MESH. 2011. Disponível em: <http://www.nlm.nih.gov/mesh/>. Acesso em: 7 mai. 2011.

MUTALIK, P. G.; DESHPANDE, A.; NADKARNI, P. M. Use of general-purpose negation detection to augment concept indexing of medical documents: a quantitative study using the umls. *Journal of the American Medical Informatics Association : JAMIA*, Department of Diagnostic Raiology, Yale University School of Medicine, New Haven, Connecticut 06510, USA. Pradeep.Mutalik@yale.edu, v. 8, n. 6, p. 598–609, 2001. ISSN 1067-5027. Disponível em: <http://www.jamia.org/cgi/content/abstract/8/6/598>.

NASSIF, H. et al. Information extraction for clinical data mining: A mammography case study. In: . [s.n.], 2009. p. 37–42. Disponível em: <http://dx.doi.org/10.1109/ICDMW.2009.63>.

NEGEX. 2011. Disponível em: <http://code.google.com/p/negex>. Acesso em: 7 mar. 2011.

NEGEXTERMS. 2011. Disponível em: <http://code.google.com/p/negex/wiki/NegExTerms>. Acesso em: 7 mar. 2011.

QUINLAN, J. *C4. 5: programs for machine learning*. [S.l.]: Morgan Kaufmann, 1993.

ROKACH, L.; ROMANO, R.; MAIMON, O. Negation recognition in medical narrative reports. *Information Retrieval*, v. 11, n. 6, p. 499–538, December 2008. ISSN 1386-4564. Disponível em: <http://dx.doi.org/10.1007/s10791-008-9061-0>.

SKEPPSTEDT, M. Negation detection in swedish clinical text. In: ____. *Second Louhi Workshop on Text and Data Mining of Health Documents (Louhi-10)*. [S.l.: s.n.], 2010. p. 15–21.

STANFILL, C.; WALTZ, D. Toward memory-based reasoning. *Communications of the ACM*, v. 29, n. 12, p. 1213–1228, 1986.

UZUNER; ZHANG, X.; SIBANDA, T. Machine learning and rule-based approaches to assertion classification. *Journal of the American Medical Informatics Association*, Elsevier, v. 16, n. 1, p. 109–115, 2009.

WANGENHEIM, C. G. von; WANGENHEIM, A. von. *Raciocínio Baseado em Casos*. [S.l.]: Manole, 2003. ISBN 8520414591.

APÊNDICE A – Lista de termos disparadores

```
rule|type
onda T negativa|<PSEU>
sem evidência|<PREN>
não há evidência|<PREN>
sem sinal|<PREN>
com sinal|<PREN>
sem sinais|<PREN>
com sinais|<PREN>
ausência|<PREN>
não identificamos|<PREN>
não se identificam|<PREN>
não observamos|<PREN>
não se observou|<PREN>
não há|<PREN>
desaparecimento|<PREN>
sem|<PREN>
pode|<PREP>
podem|<PREP>
possa|<PREP>
possam|<PREP>
com ou sem|<PREP>
não parece|<PREP>
não parece ser|<PREP>
não sendo possível afastar|<PREP>
não se pode afirmar|<PREP>
não se pode descartar|<PREP>
não se pode descartar provável|<PREP>
```

```
não se pode descartar uma provável|<PREP>
não se pode excluir|<PREP>
não sendo possível afastar|<PREP>
porém não dá para descartar|<PREP>
podem ser|<PREP>
pode ser|<PREP>
provável|<PREP>
provavel|<PREP>
provavelmente|<PREP>
interrogável|<PREP>
provável|<PREP>
discutível|<PREP>
sugere|<PREP>
sugestiva|<PREP>
sugestivas|<PREP>
sugestivo|<PREP>
sugestivos|<PREP>
ausentes|<POST>
ausente|<POST>
desaparecido|<POST>
desaparecidos|<POST>
desaparecida|<POST>
desaparecidas|<POST>
\\?|<POSP>
sugestivo|<POSP>
porém|<CONJ>
no entanto|<CONJ>
contudo|<CONJ>
todavia|<CONJ>
entretanto|<CONJ>
```

APÊNDICE B – Código-Fonte

```
# -*- coding: utf-8 -*-

import re

# \S+ is used in the category group because the \w+ pattern doesn't match unicode characters in
python2.* and earlier
TAG_PATTERN = re.compile(r'<\s*achado\s*(?:tipo="(?P<category>\S+)")?>(?P<text>.*?)</\s*achado\s*>',
re.I)


class ClassificationEnum:
    """This class simulates an enumeration."""
    Affirmative, Negative, Speculative, Ambiguous = range(4)

CLASSIF_STR_DICT = {
    ClassificationEnum.Affirmative : 'afirmativo',
    ClassificationEnum.Negative : 'negativo',
    ClassificationEnum.Speculative : 'hipotético',
    ClassificationEnum.Ambiguous : 'ambíguo'
    }

DIACRITICAL_DICT = {
    'a': 'aáàâã',
    'á': 'aáàâã',
    'à': 'aáàâã',
    'â': 'aáàâã',
    'ã': 'aáàâã',
    'A': 'aáàâã',
    'Á': 'aáàâã',
    'À': 'aáàâã',
    'Â': 'aáàâã',
    'Ã': 'aáàâã',

    'e': 'eéèêẽ',
    'é': 'eéèêẽ',
    'è': 'eéèêẽ',
    'ê': 'eéèêẽ',
    'ẽ': 'eéèêẽ',
    'E': 'eéèêẽ',
    'É': 'eéèêẽ',
    'È': 'eéèêẽ',
    'Ê': 'eéèêẽ',
    'Ẽ': 'eéèêẽ',

    'i': 'iíìîĩ',
    'í': 'iíìîĩ',
    'ì': 'iíìîĩ',
    'î': 'iíìîĩ',
    'ĩ': 'iíìîĩ',
    'I': 'iíìîĩ',
    'Í': 'iíìîĩ',
    'Ì': 'iíìîĩ',
    'Î': 'iíìîĩ',
    'Ĩ': 'iíìîĩ',

    'o': 'oóòôõ',
    'ó': 'oóòôõ',
    'ò': 'oóòôõ',
    'ô': 'oóòôõ',
    'õ': 'oóòôõ',
    'O': 'oóòôõ',
    'Ó': 'oóòôõ',
    'Ò': 'oóòôõ',
    'Ô': 'oóòôõ',
    'Õ': 'oóòôõ',

    'u': 'uúùûũ',
    'ú': 'uúùûũ',
    'ù': 'uúùûũ',
    'û': 'uúùûũ',
    'ũ': 'uúùûũ',
    'U': 'uúùûũ',
```

```
    'Ú': 'uúùûũ',
    'Ù': 'uúùûũ',
    'Û': 'uúùûũ',
    'Ũ': 'uúùûũ',

    'c': 'cç',
    'ç': 'cç',
    'C': 'cç',
    'Ç': 'cç'
    }

def diacritical_charset_pattern(sentence):
    chars = list()
    for c in sentence:
        if c in DIACRITICAL_DICT:
            chars += '[' + DIACRITICAL_DICT[c] + ']'
        else:
            chars += c
    return ''.join(chars)
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import operator as op
import os
import functools
import subprocess

from tcc import common
from tcc.internals import merge_taglists
from tcc.preprocess import indexer, tokenizer
from tcc.negex import negex

AFFIRMATIVE, NEGATIVE, HYPOTHETIC, AMBIGUOUS = 'afirmativo', 'negativo', 'hipotético', 'ambíguo'
FEATURE_WINDOW = 4

def split_sent(sentence):
    """Splits the sentence by the number of findings in the sentence.
    Returns an empty string if there is no finding.

    """

    findings = list()
    match_iter = common.TAG_PATTERN.finditer(sentence)
    matches = list(match_iter)
    finding_start = 0
    cleaned_sentence = ''
    previous_match_end = 0
    # remove tags
    if len(matches) > 0:
        for match in matches:
            category = ' tipo="' + match.group('category') + '"' if match.group('category') else ''
            finding = match.group('text')
            cleaned_sentence += sentence[previous_match_end:match.start()] + finding
            previous_match_end = match.end()

            start = cleaned_sentence.index(finding, finding_start)
            end = start + len(finding)
            findings.append( (finding, category, start, end) )

            finding_start = end
            sentence_final_part = sentence[match.end():]

        cleaned_sentence += sentence_final_part
    else:
        return ''

    return [ cleaned_sentence[:s] + '<achado' + c + '>' + find + '</achado>' + cleaned_sentence[e:] for
find, c, s, e in findings ]

def remove_tag(sent, train=False):
    """Remove the <achado> tag from the sentence.
    Returns a tuple consisting of: the cleaned sentence; the finding token, with its slice indexes;
    and optionally the finding's category, if the train parameter is True.

    """
    match = common.TAG_PATTERN.search(sent)
    assert match, 'There\'s no tagged finding in the sentence "' + sent + '"\n'

    finding = match.group('text')
    cleaned_sent = sent[:match.start()] + finding + sent[match.end():]

    start = cleaned_sent.index(finding)
    end = start + len(finding)
    finding_token = (start, end, 'FIND')
    if train:
        category = match.group('category')
        return (cleaned_sent, finding_token, category)
    return (cleaned_sent, finding_token)


def str2tuple(t):
```

```
    sep = '//' if '//' in t else '/'
    return tuple(t.rsplit(sep, 1))


def main(sents, mbt_filename, option):
    """

    Arguments:
    sents: an iterable with sentences.
    mbt_filename: the tagger settings' file.
    option: train or test

    """
    splits_lists = (split_sent(s.strip()) for s in f)
    splits_lists = (s for s in splits_lists if s) # remove non-tagged lines
    sents = functools.reduce(op.concat, splits_lists, [])
    if option == 'train':
        removes = (remove_tag(s, train=True) for s in sents)
        removes = [(s, f, c) for s, f, c in removes if c != AMBIGUOUS] # remove ambiguous findings
        clean_sents, finding_tokens, classes = zip(*removes)
    else:
        removes = [remove_tag(s) for s in sents]
        clean_sents, finding_tokens = zip(*removes)

    tok = tokenizer.Tokenizer()
    span_t_sents = [tok.span_tokenize(s) for s in clean_sents]
    assert len(span_t_sents) == len(clean_sents)
    t_sents = list()
    # prepare Mbt data
    for ts, s in zip(span_t_sents, clean_sents):
        t_sent = [s[slice(*t)] for t in ts]
        t_sent.append('<utt>')
        t_sents.append(t_sent)

    assert len(t_sents) == len(clean_sents)

    # call Mbt -s mbt_filename < t_sents
    p = subprocess.Popen(['/usr/bin/Mbt', '-s', mbt_filename], stdin=subprocess.PIPE,
stdout=subprocess.PIPE)
    formatted_data = bytes('\n'.join(functools.reduce(op.concat, t_sents, [])), 'utf8')
    pos_tagged_data = p.communicate(formatted_data)[0]
    pos_tagged_sents = str(pos_tagged_data, 'utf8').split('<utt>')
    pos_tagged_sents = [s.strip() for s in pos_tagged_sents if s]
    pos_tagged_sents = [s for s in pos_tagged_sents if s]

    # print(pos_tagged_sents)
    assert_msg = 'len(pos_tagged_sents) == ' + str(len(pos_tagged_sents)) + '\nlen(clean_sents) == ' +
str(len(clean_sents))
    assert len(pos_tagged_sents) == len(clean_sents), assert_msg

    ## sys.stdout.writelines([s+'\n' for s in pos_tagged_sents])

    # transform sents into lists of token-tuples
    pos_tuple_sents = list()
    for sent in pos_tagged_sents:
        tokens = sent.split()
        pos_tuple_sents.append([str2tuple(t) for t in tokens])


    token_sents = list()
    assert len(span_t_sents) == len(pos_tuple_sents)
    for span_sent, tag_sent, clean_sent in zip(span_t_sents, pos_tuple_sents, clean_sents):

        assert len(span_sent) == len(tag_sent)
        for span_t, (tag_w, tag_t) in zip(span_sent, tag_sent):
            assert tag_w == clean_sent[slice(*span_t)]

        token_sents.append([(s, e, t) for (s, e), (w, t) in zip(span_sent, tag_sent)])

    feature_records = list()
    neg_tagger = negex.Tagger()
    assert len(clean_sents) == len(finding_tokens)
```

```
    for sent, token_sent, finding_tok in zip(clean_sents, token_sents, finding_tokens):
        neg_tags = neg_tagger.neg_tag(sent)
        final_sent_tokens = merge_taglists([finding_tok], neg_tags)
        # print(final_sent_tokens)
        final_sent_tokens = merge_taglists(final_sent_tokens, token_sent)
        # print(final_sent_tokens)

        finding_index = final_sent_tokens.index(finding_tok)
        final_sent_tags = (tok[2] for tok in final_sent_tokens)
        change_comma = lambda t: 'comma' if t == ',' else t
        final_sent_tags = [change_comma(tag) for tag in final_sent_tags]

        feat_previous = final_sent_tags[max(0, finding_index-FEATURE_WINDOW):finding_index]
        previous_remaining = FEATURE_WINDOW - len(feat_previous)
        if previous_remaining:
            feat_previous = previous_remaining * ['NULL'] + feat_previous

        feat_following = final_sent_tags[finding_index+1:finding_index+1+FEATURE_WINDOW]
        following_remaining = FEATURE_WINDOW - len(feat_following)
        if following_remaining:
            feat_following += following_remaining * ['NULL']

        feature_records.append(','.join(feat_previous + ['FIND'] + feat_following))

    if option == 'train':
        feature_records = (f + ',' + c for f, c in zip(feature_records, classes))

    sys.stdout.writelines([f+'\n' for f in feature_records])



if __name__ == '__main__':
    import sys
    args = sys.argv
    if len(args) < 3:
        print('\nModo de uso: ' + args[0] + ' train|test mbt_file [entrada]', file=sys.stderr)
        sys.exit(1)

    f = open(args[3], 'r') if len(args) >= 4 else sys.stdin
    mbt_filename = args[2]
    option = args[1]
    assert option == 'train' or option == 'test', 'Opção inválida, deve ser train ou test.\n'
    main(f, mbt_filename, option)
```

```python
# -*- coding: utf-8 -*-

def merge_taglists(taglist1, taglist2):
    """Returns sorted taglist1, with taglist2's tags inserted in taglist1.
    For each tag t in taglist2, t is inserted only if its span doesn't overlap
    with any tag of taglist1.

    """
    newlist1 = taglist1[:]
    rangesets1 = [set(range(t[0], t[1])) for t in taglist1]
    rangesets2 = [set(range(t[0], t[1])) for t in taglist2]
    for r2, t2 in zip(rangesets2, taglist2):
        if all(r2.isdisjoint(r1) for r1 in rangesets1):
            newlist1.append(t2)
            rangesets1.append(r2)

    newlist1.sort(key=lambda x: x[0])
    return newlist1


def abstract():
    import inspect
    caller = inspect.getouterframes(inspect.currentframe())[1][3]
    raise NotImplementedError(caller + ' must be implemented in subclass')

def to_unicode_or_bust(obj, encoding='utf-8'):
    if isinstance(obj, basestring):
        if not isinstance(obj, unicode):
            obj = unicode(obj, encoding)
    return obj
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import itertools, collections, psycopg2, sqlite3

from tcc import metrics

ANOT_CLASSIF = {
    1: 'afirmativo',
    2: 'negativo',
    3: 'hipotetico',
    4: 'hipotetico',
    5: 'ambiguo',
    6: 'nao_achado'
    }

def extract_achado():
    pass

def main():
    selectStr = 'SELECT at.id_anot_sqlite, at.avaliador, at.resposta \
        FROM amostra_ultrassom AS am \
        INNER JOIN anotacao_ultrassom AS at ON am.id = at.id_amostra_ultrassom \
        WHERE am.id IN (SELECT id_amostra_ultrassom \
                FROM anotacao_ultrassom \
                GROUP BY id_amostra_ultrassom \
                HAVING count(id_amostra_ultrassom) = 2) \
        ORDER BY am.id, at.avaliador;'

    conn = psycopg2.connect(host='150.162.67.6', database='fcbertoldi', user='pgsql')
    cur = conn.cursor()
    cur.execute(selectStr)

    AnotRecord = collections.namedtuple('AnotRecord', 'id_anot_sqlite avaliador resposta')
    queryResult = [AnotRecord._make(row) for row in cur.fetchall()]

    sqlite_conn = sqlite3.connect('/home/fernando/laudos_ultrassom.sqlite3')
    sqlite_cur = sqlite_conn.cursor()
    sqliteQueryStr = 'SELECT sd.id AS sd_id, s.id AS s_id, s.texto AS texto, sd.decs_id AS decs_id,
sd.offset AS offset, sd.length AS length \
    FROM sentencas AS s \
    INNER JOIN sents_decs AS sd ON s.id = sd.sentencas_id;'
    Sentences = collections.namedtuple('Sentences', 'sd_id s_id texto decs_id offset length')
    sqlite_cur.execute(sqliteQueryStr)
    sqliteQueryResult = [Sentences._make(r) for r in sqlite_cur.fetchall()]
    # dicionario 's_id.decs_id.offset' -> row
    sents_dict = dict()
    for tup in sqliteQueryResult:
        k = str(tup.s_id) + '.' + str(tup.decs_id) + '.' + str(tup.offset)
        sents_dict[k] = tup

    par1_av1 = list()
    par1_av2 = list()
    par2_av1 = list()
    par2_av2 = list()
    nao_achados = collections.Counter()
    sents_ambiguas = set()

    insertStr = 'INSERT INTO respostas (anot_id, resposta) VALUES (?,?);'
    # calcular as matrizes de confusão para os dois pares
    for k, g in itertools.groupby(queryResult, key=lambda row: row.id_anot_sqlite):

        group = list(g)
        assert len(group) == 2

        if ANOT_CLASSIF[group[0].resposta] == 'nao_achado' or ANOT_CLASSIF[group[1].resposta] ==
'nao_achado':
            sents_row = sents_dict[group[0].id_anot_sqlite]
            begin = sents_row.offset
            end = sents_row.offset + sents_row.length
            achado = sents_row.texto[begin:end]
            nao_achados[achado] += 1
```

```
            continue

        # se alguma das respostas for ambigua
        if ANOT_CLASSIF[group[0].resposta] == 'ambiguo' or ANOT_CLASSIF[group[1].resposta] ==
'ambiguo':
            amb_sent = sents_dict[group[0].id_anot_sqlite].texto
            sents_ambiguas.add(amb_sent)
            continue

        if group[0].avaliador == 1 and group[1].avaliador == 2:
            av1_resp = group[0].resposta
            par1_av1.append(ANOT_CLASSIF[av1_resp])
            av2_resp = group[1].resposta
            par1_av2.append(ANOT_CLASSIF[av2_resp])
            if av1_resp == av2_resp:
                resp = ANOT_CLASSIF[av1_resp]
                sents_row = sents_dict[group[0].id_anot_sqlite]
                sqlite_cur.execute(insertStr, (sents_row.sd_id, resp))

        elif group[0].avaliador == 2 and group[1].avaliador == 3:
            av1_resp = group[0].resposta
            par2_av1.append(ANOT_CLASSIF[av1_resp])
            av2_resp = group[1].resposta
            par2_av2.append(ANOT_CLASSIF[av2_resp])
            if av1_resp == av2_resp:
                resp = ANOT_CLASSIF[av1_resp]
                sents_row = sents_dict[group[0].id_anot_sqlite]
                sqlite_cur.execute(insertStr, (sents_row.sd_id, resp))

        else:
            msg = 'group[0]: ' + str(group[0]) + '\ngroup[1]:' + str(group[1])
            assert False, 'pares de avaliadores deveriam ser 1 e 2 ou 2 e 3.\n' + msg

    sqlite_conn.commit()
    cm1 = metrics.ConfusionMatrix(par1_av1, par1_av2)
    cm2 = metrics.ConfusionMatrix(par2_av1, par2_av2)

    print(cm1, cm2, nao_achados, len(sents_ambiguas), sep='\n')
    # TODO: atualizar tabela respostas


if __name__ == '__main__':
    main()
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os
import re
import sys
from collections import defaultdict, Counter

from tcc.common import TAG_PATTERN, ClassificationEnum, CLASSIF_STR_DICT
from tcc.preprocess.indexer import Annotator
from tcc.negex import negex

class ConfusionMatrix:
    def __init__(self, gold, test):
        assert len(gold) == len(test)
        self._classes = sorted(set(gold) | set(test))
        self._matrix = defaultdict(lambda: defaultdict(int))
        for g, t in zip(gold, test):
            self._matrix[g][t] += 1


    def __str__(self):
        max_len = len(max(*self._classes, key=len))
        line_offset = ' ' * (max_len + 1)
        labels = [c.rjust(max_len) for c in self._classes]
        s_list = list()

        for l in zip(*labels):
            s_list.append(line_offset)
            s_list.extend(' {:>3}'.format(el) for el in l)
            s_list.append('\n')

        s_list.append('\n')
        for k, l in zip(self._classes, labels):
            row = self._matrix[k]
            s_list.append(l)
            s_list.append(':')
            s_list.extend(' {:>3d}'.format(row[k2]) for k2 in self._classes)
            s_list.append('\n\n')

        s_list.append('\nrow: reference\ncolumn: test\n')

        return ''.join(s_list)

def extract_negex_reference(sentences):
    refs = list()
    for line in sentences:
        match_iter = TAG_PATTERN.finditer(line)
        for match in match_iter:
            category = match.group('category')
            refs.append(category)


    return refs


def extract_negex_test(sentences):
    neg_tagger = negex.Tagger()
    annotator = Annotator()
    split_sents_tuples = annotator.split_sents(sentences)
    split_sents_index, split_sents = zip(*split_sents_tuples)
    test_list = list()
    annot_sents = list()
    for line in split_sents:
        annotated_sent = neg_tagger.annotate(line)
        findings = annotated_sent.findings()
        classifications = [CLASSIF_STR_DICT[f.classification()] for f in findings]
        annot_sents.extend(annotated_sent for i in range(len(classifications)))
        test_list.extend(classifications)

    return test_list, annot_sents
```

```
def print_triggers_table(tp_counter, fp_counter):
    """Prints a table with information about each trigger found in the
    test, such as its precision, and the number of times identified in
    the test set.

    Arguments:
    tp_counter - a Counter of true positives
    fp_counter - a Counter of false positives

    """
    triggers = tp_counter.keys() | fp_counter.keys()
    triggers = sorted(triggers, key=lambda k: tp_counter[k] + fp_counter[k], reverse=True)
    s_list = list()
    s_list.append('{:40s}  TP  FP  Precisão  Nº de Ocorrências'.format('Frase Disparadora'))
    for t in triggers:
        num_occurrences = tp_counter[t] + fp_counter[t]
        precision = tp_counter[t] / num_occurrences
        s_list.append('{:40s}  {:>2d}  {:>2d}  {:>8.2f}  {:>17d}'.format(t, tp_counter[t], fp_counter
[t], precision, num_occurrences))

    print('\n'.join(s_list))

def print_findings_table(tp_counter, fp_counter, tn_counter, fn_counter):
    """Prints a table with information about findings found in the
    test, such as the number of false positives and true positives,
    and the number of occurrences

    Arguments:
    tp_counter - a Counter of true positives
    fp_counter - a Counter of false positives
    tn_counter - a Counter of true negatives
    fn_counter - a Counter of false negatives

    """
    triggers = tp_counter | fp_counter | tn_counter | fn_counter
    triggers = sorted(triggers, key=lambda k: tp_counter[k] + fp_counter[k] + tn_counter[k] +
fn_counter[k], reverse=True)
    s_list = list()
    s_list.append('{:40s}  Nº de Ocorrências  FP  FN'.format('Achado'))
    for t in triggers:
        num_occurrences = tp_counter[t] + fp_counter[t] + tn_counter[t] + fn_counter[t]
        s_list.append('{:40s}  {:>17d}  {:>2d}  {:>2d}'.format(t[8:-9], num_occurrences, fp_counter
[t], fn_counter[t]))

    print('\n'.join(s_list))

def main(sentences):
    """

    Arguments:
    sentences - a sequence of sentences.

    """
    gold = extract_negex_reference(sentences)
    test, annot_sents = extract_negex_test(sentences)
    assert len(test) == len(annot_sents)
    assert len(gold) == len(test), "\nlen(gold) == " + str(len(gold)) + "\nlen(test) == " + str(len
(test)) + "\n"

    trigger_tp_cnt = Counter()
    trigger_fp_cnt = Counter()
    finding_tp_cnt = Counter()
    finding_fp_cnt = Counter()
    finding_tn_cnt = Counter()
    finding_fn_cnt = Counter()
    for i, (g, t, a) in enumerate(zip(gold, test, annot_sents)):
        finding = a.findings()[0]
        findings_triggers = a.findings_triggers()
        print(a.sentence())
        print('FINDING: ', finding)
        print('GOLD: ', g)
        print('TEST: ', t)
```

```
        print('TRIGGERS:')
        for trig in a.triggers():
            print(trig.scope())

        print('\n')
        if g == "afirmativo" and t == "afirmativo":
            # TRUE NEGATIVE (TN)
            finding_tn_cnt[str(finding)] += 1

        elif g != "afirmativo" and t == "afirmativo":
            # FALSE NEGATIVE (FN)
            finding_fn_cnt[str(finding)] += 1

        elif g == t and (g == "negativo" or g == "hipotético"):
            # TRUE POSITIVE (TP)
            finding_tp_cnt[str(finding)] += 1
            trigger = findings_triggers[finding]
            trigger_tp_cnt[str(trigger)] += 1

        else:
            # FALSE POSITIVE (FP)
            finding_fp_cnt[str(finding)] += 1
            trigger = findings_triggers[finding]
            trigger_fp_cnt[str(trigger)] += 1

    true_positives = sum(finding_tp_cnt.values())
    false_positives = sum(finding_fp_cnt.values())
    true_negatives = sum(finding_tn_cnt.values())
    false_negatives = sum(finding_fn_cnt.values())
    precision = true_positives / (true_positives + false_positives)
    recall = true_positives / (true_positives + false_negatives)
    accuracy = (true_positives + true_negatives) / (true_positives + true_negatives + false_positives
+ false_negatives)
    f_measure = (2*precision*recall) / (precision + recall)

    print('TP = ', true_positives)
    print('TN = ', true_negatives)
    print('FP = ', false_positives)
    print('FN = ', false_negatives)
    print('PRECISION: {:.2f}'.format(precision))
    print('RECALL: {:.2f}'.format(recall), end='\n\n')
    print('ACCURACY: {:.2f}'.format(accuracy), end='\n\n')
    print('F-MEASURE: {:.2f}'.format(f_measure), end='\n\n')
    cm = ConfusionMatrix(gold, test)
    print('Confusion Matrix:', end='\n\n')
    print(cm)
    print('\n\n')
    print_triggers_table(trigger_tp_cnt, trigger_fp_cnt)
    print('\n\n')
    print_findings_table(finding_tp_cnt, finding_fp_cnt, finding_tn_cnt, finding_fn_cnt)



if __name__ == '__main__':
    import sys
    args = sys.argv
    if len(args) >= 2:
        sents = [l.strip() for l in open(args[1]).readlines()]
        main(sents)

    else:
        print('modo de uso: ' + args[0] + ' conjunto_de_teste', file=sys.stderr)
        sys.exit(1)
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import collections
import re


# html.entities eh novo na versao 3
try:
    from html.entities import name2codepoint
except ImportError:
    from htmlentitydefs import name2codepoint
# construcoes usadas para trabalhar com a base de laudos

# occurrences = dict(zip(name2codepoint.keys(), (0 for n in range(len(name2codepoint)))))

wrong_encodings = {
    'Ã§': 'ç',
    'Ã£': 'ã',
    'Ãµ': 'õ',
    'Ãi': 'á',
    'Ã©': 'é',
    'Ãª': 'ê',
    'Ã\xad': 'í',
    'Ã³': 'ó',
    'Ãº': 'ú',
    'Ã\xa0': 'à',
    'Ã¢': 'â',
    'Ã\x81': 'Á',
    'Ã\x8d': 'Í',
    'inespecï¿½ficas': 'inespecíficas',
    'taquicï¿½rdico': 'taquicárdico',
    'bradicï¿½rdico': 'bradicárdico',
    'Subepicï¿½rdica': 'Subepicárdica',
    '\xa0': ' '
    }

wrong_tags = {
    '</br>': '<br />',
    '<<br>': '<br />',
    '<div />': '',
    '<p/>': '</p>',
    '</p<p>': '</p>'
    }


def charent_occurrences(filename):
    """Returns a dictionary of occurrences of char entities in the file.

    Arguments:
    filename: name of the file
    """
    occurrences = collections.defaultdict(int)
    pattern = re.compile('&(' + '|'.join(name2codepoint.keys()) + ');')

    with open(filename, 'r') as f:
        for line in f:
            match_iter = pattern.finditer(line)
            for match in match_iter:
                match_str = match.group()
                char_entity = match_str[1:-1]
                occurrences[char_entity] += 1
    return occurrences

def tag_occurrences(filename):
    """Returns a dict of tags in the file 'filename', with the tag as key and its number of
occurrences as its value.

    Arguments:
    filename: name of the file
    """
    tags_occur = collections.defaultdict(int)
```

```
    pattern = re.compile('<.*?>')

    with open(filename) as f:
        for line in f:
            match_iter = pattern.finditer(line)
            for match in match_iter:
                tags_occur[match.group()] += 1

    return tags_occur


def wrong_encoding_words(filename):
    """Returns a set containing all two characters strings that begin with the letter Ã in the file.
    Two characters strings beginning with the letter Ã are generally the result of an encoding error.
    For example,
    'ç'.encode('latin_1').decode('utf8') -> 'Ã§'

    Arguments:
    filename - name of the file

    """
    atildes = set()
    with open(filename) as f:
        for line in f:
            next_start = 0
            atil_index = line.find('Ã', next_start)
            while atil_index != -1:
                next_start = atil_index+1
                if next_start >= len(line):
                    break
                atilde = line[atil_index:atil_index+2]
                atildes.add(atilde)
                atil_index = line.find('Ã', next_start)
    return atildes

def replacement_character_words(filename):
    """Returns a set of all the words in the file wich have the str ï¿½

    'ï¿½'.encode('latin_1').decode('utf8') -> unicode representational character
    This character is used when a byte sequence is not recognized as a character in its encoding.

    """

    repl_pattern = re.compile('\w*ï¿½\w*')
    repl_character_set = set()

    with open(filename) as f:
        for line in f:
            match_iter = repl_pattern.finditer(line)
            for match in match_iter:
                repl_character_set.add(match.group())

    return repl_character_set

class CharEntitySubstituter:
    def __init__(self, dict):
        self.dict = dict

    def __call__(self, matchobj):
        match_str = matchobj.group()
        char_entity = match_str[1:-1]
        code_point = self.dict[char_entity]
        return chr(code_point)

class WrongEncodingSubstituter:
    def __init__(self, dict):
        self.dict = dict

    def __call__(self, matchobj):
        match_str = matchobj.group()
        return self.dict[match_str]
```

```
class WrongTagSubstituter:
    def __init__(self, dict):
        self.dict = dict

    def __call__(self, matchobj):
        match_str = matchobj.group()
        return self.dict[match_str]

def substitute_wrong_tags(old_file, new_file, encoding='utf-8'):
    """Substitute wrong tags in the old_file, and save it in new_file.

    """
    pattern = re.compile('|'.join(wrong_tags.keys()))
    wTagSub = WrongTagSubstituter(wrong_tags)

    chunk_size = 2**20
    oldf = open(old_file, 'r', encoding=encoding)
    file_size = oldf.seek(0, 2)
    num_reads = file_size // chunk_size
    oldf.seek(0, 0)

    newf = open(new_file, 'w')
    i = 0
    while i < num_reads:
        s = oldf.read(chunk_size)
        substituted_str = pattern.sub(wTagSub, s)
        newf.write(substituted_str)
        i += 1
    # se ainda houver algo para ler, obte-lo
    if file_size % chunk_size != 0:
        s = oldf.read()
        substituted_str = pattern.sub(wTagSub, s)
        newf.write(substituted_str)

    oldf.close()
    newf.close()

def substitute_wrong_encoding(text):
    """Substitute some character sequences known to be with the wrong encoding in text.

    Arguments:
    text - a text iterable

    """
    pattern = re.compile('|'.join(wrong_encodings.keys()))
    wSub = WrongEncodingSubstituter(wrong_encodings)

    ## chunk_size = 2**20
    ## oldf = open(old_file, 'r')
    ## file_size = oldf.seek(0, 2)
    ## num_reads = file_size // chunk_size
    ## oldf.seek(0, 0)

    text_str = ''.join(text)
    return pattern.sub(wSub, text_str)

    ## i = 0
    ## while i < num_reads:
    ##     s = oldf.read(chunk_size)
    ##     substituted_str = pattern.sub(wSub, s)
    ##     sys.stdout.write(substituted_str)
    ##     i += 1
    ## # se ainda houver algo para ler, obtê-lo
    ## if file_size % chunk_size != 0:
    ##     s = oldf.read()
    ##     substituted_str = pattern.sub(wSub, s)
    ##     sys.stdout.write(substituted_str)

    ## oldf.close()

def unescape(text):
    def fixup(matchobj):
```

```
        txt = matchobj.group(0)
        if txt[:2] == '&#':
            # character reference
            try:
                if txt[:3] == '&#x':
                    return chr(int(txt[3:-1], 16))
                else:
                    return chr(int(txt[2:-1]))
            except ValueError:
                pass
        else:
            # named entity
            try:
                txt = chr(name2codepoint[txt[1:-1]])
            except KeyError:
                pass
        # leave as is
        return txt

    pat = re.compile(r'&#?\w+;')
    return pat.sub(fixup, text)


def substitute_char_entities(text):
    """Substitute html character entities in text for their corresponding unicode code points.

    Arguments:
    text - a text iterable

    """
    pattern = re.compile('&(' + '|'.join(name2codepoint.keys()) + ');')
    char_ents_in_text = set()

    text_str = ''.join(text)
    match_iter = pattern.finditer(text_str)
    for match in match_iter:
        match_str = match.group()
        char_entity = match_str[1:-1]
        char_ents_in_text.add(char_entity)

    ## oldf = open(old_file, 'r')
    ## for line in oldf:
    ##     match_iter = pattern.finditer(line)
    ##     for match in match_iter:
    ##         match_str = match.group()
    ##         char_entity = match_str[1:-1]
    ##         char_ents_in_text.add(char_entity)


    entdefs = {element: name2codepoint[element] for element in char_ents_in_text}
    entdefs['nbsp'] = ord(' ') # trata nbsp como espaço
    csub = CharEntitySubstituter(entdefs)

    ## chunk_size = 2**20
    ## file_size = oldf.tell()
    ## num_reads = file_size // chunk_size
    ## oldf.seek(0, 0)

    return pattern.sub(csub, text_str)

    ## i = 0
    ## while i < num_reads:
    ##     s = oldf.read(chunk_size)
    ##     substituted_str = pattern.sub(csub, s)
    ##     sys.stdout.write(substituted_str)
    ##     i += 1
    ## # se ainda houver algo para ler, obte-lo
    ## if file_size % chunk_size != 0:
    ##     s = oldf.read()
    ##     substituted_str = pattern.sub(csub, s)
    ##     sys.stdout.write(substituted_str)
```

```
    ## oldf.close()


if __name__ == "__main__":
    import sys
    args = sys.argv
    if len(args) >= 3:
        data = open(args[2], 'r')
        if args[1] == 'subst-charents':

            print(substitute_char_entities(data), end='')

        elif args[1] == 'subst-wrong-enc':

            print(substitute_wrong_encoding(data), end='')

        else:
            print('Argumento errado, deve ser subst-charents ou subst-wrong-enc')
            data.close()
            sys.exit(1)

        data.close()
    else:
        pass # modo de uso: preprocess.py [subst-charents|subst-wrong-enc] old_f new_f encoding
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import curses
import curses.wrapper
import itertools
import re
import sys

from tcc.preprocess import indexer

FINDING_PAT = re.compile(r'<achado>(.*?)</achado>', re.IGNORECASE)

def _remove_tag(sentence):
    """Returns in a triple the sentence without the <achado> tag, the start and end indexes of the finding in the returned sentence.

    """
    match = FINDING_PAT.search(sentence)
    finding = match.group(1)
    cleaned_sentence = sentence[:match.start()] + finding + sentence[match.end():]

    start = cleaned_sentence.index(finding)
    end = start + len(finding)
    return (cleaned_sentence, start, end)

def _add_sentence_to_screen(sentence, stdscr, y, x):
    cleaned_sent, f_start, f_end = _remove_tag(sentence)
    stdscr.move(y, x)
    stdscr.addstr(cleaned_sent[:f_start])
    stdscr.addstr(cleaned_sent[f_start:f_end], curses.A_REVERSE)
    stdscr.addstr(cleaned_sent[f_end:])


def main(stdscr, sample_sentences, corpus_file):
    """
    Arguments:
    sample_sentences - a sequence of sample sentences.

    """

    ## sample_sentences = open(sample_file, 'r').readlines()
    annotator = indexer.Annotator()
    annotator_l = annotator.split_sents(sample_sentences)
    annotator_l_iter = iter(annotator_l)

    annot_group = [list(g) for k, g in itertools.groupby(annotator_l, lambda x: x[0])]
    assert len(sample_sentences) == len(annot_group)
    annot_i = 0
    for s in sample_sentences:
        gr = annot_group[annot_i]
        assert s.count('<achado>') == len(gr)
        annot_i += 1

    num_of_buttons = 4
    button_space = 10
    sentence_y_offset = int(curses.LINES * 1/3)
    sentence_x_offset = button_space
    buttons_y_offset = int(curses.LINES * 2/3)
    buttons_x_offset = int(curses.COLS / num_of_buttons)

    try:
        index, sentence = next(annotator_l_iter)
    except StopIteration:
        sys.stderr.write('Arquivo de amostra está vazio.')
        sys.exit(1)
    _add_sentence_to_screen(sentence, stdscr, sentence_y_offset, sentence_x_offset)
    sent_count = 0
    stdscr.addstr(0, 0, str(sent_count))

    addstr_buttons_args = list()
    addstr_buttons_args.append((buttons_y_offset, 0*buttons_x_offset + button_space, 'afirmativo'))
```

```
    addstr_buttons_args.append((buttons_y_offset, 1*buttons_x_offset + button_space, 'negativo'))
    addstr_buttons_args.append((buttons_y_offset, 2*buttons_x_offset + button_space, 'hipotético'))
    addstr_buttons_args.append((buttons_y_offset, 3*buttons_x_offset + button_space, 'ambíguo'))

    y, x, text = addstr_buttons_args[0]
    # highlight the default choice
    stdscr.addstr( y, x, text, curses.A_REVERSE)
    stdscr.addstr( *addstr_buttons_args[1] )
    stdscr.addstr( *addstr_buttons_args[2] )
    stdscr.addstr( *addstr_buttons_args[3] )
    stdscr.move(buttons_y_offset, button_space)
    stdscr.refresh()

    annotated_sentences = list()
    selected_button = 0
    key_enter = 10

    while True:
        try:
            c = stdscr.getch()
            if 0<c<256 and chr(c) in 'Qq':
                break
            elif c == key_enter:
                annotated_sentences.append( (index, sentence, selected_button) )

                # clean previous sentence
                stdscr.addstr(sentence_y_offset, sentence_x_offset, len(sentence) * ' ')
                index, sentence = next(annotator_l_iter)
                sent_count += 1
                stdscr.addstr(0, 0, str(sent_count))
                _add_sentence_to_screen(sentence, stdscr, sentence_y_offset, sentence_x_offset)

            elif c == curses.KEY_RIGHT:
                stdscr.addstr( *addstr_buttons_args[selected_button] )
                selected_button = (selected_button+1) % num_of_buttons
                y, x, button_text = addstr_buttons_args[selected_button]
                stdscr.addstr( y, x, button_text, curses.A_REVERSE )
            elif c == curses.KEY_LEFT:
                stdscr.addstr( *addstr_buttons_args[selected_button])
                selected_button = (selected_button-1) % num_of_buttons
                y, x, button_text = addstr_buttons_args[selected_button]
                stdscr.addstr( y, x, button_text, curses.A_REVERSE )
            else:
                # ignore incorrect keys
                pass

            stdscr.move(y, x)
            stdscr.refresh()
        except StopIteration:
            break

    sent_groups = [ list(g) for k, g in itertools.groupby(annotated_sentences, lambda x: x[0]) ]
    assert len(sample_sentences) == len(sent_groups)

    index = 0
    for sent in sample_sentences:
        group = sent_groups[index]
        annotated_sent = annotator.annotate_sentence(sent, [tupl[2] for tupl in group])
        corpus_file.write(annotated_sent)
        index += 1



if __name__ == '__main__':
    args = sys.argv
    if len(args) >= 3:
        samples = open(args[1], 'r')
        corpus = open(args[2], 'w')
        curses.wrapper(main, list(samples), corpus)
        corpus.close()
    else:
        print('Modo de uso: ' + args[0] + ' arquivo_de_sentenças arquivo_de_saída.')
```

```
        sys.exit(1)
```

```
# -*- coding: utf-8 -*-
"""This module has the dictionaries used to index findings in the
medical reports."""

ULTRASOUND_DICT = {

    'insercao_posterior':
    r"""inser[çc][ãa]o
    [ \t]+
    posterior""",

    'AVF':
    r"""AVF""",

    'saco_gestacional':
    r"""saco
    [ \t]+
    gestacional""",

    'fundo_de_saco':
    r"""fundo
    [ \t]+
    de
    [ \t]+
    saco""",

    'cisto':
    r"""cistos?
    (?:[ \t]+
    (?:col[óo]ides|foliculares|lobulado))?""",

    'ecogenicidade':
    r"""ecogenicidade
    (?:[ \t]+
    (?:norma(?:l|is) |
    preservadas? |
    habitua(?:l|is) |
    heterog[êẽe]neas? |
    aumentadas?))?""",

    'colecoes':
    r"""cole[çc][õôo]es""",

    'indice':
    r"""[íi]ndice""",

    'liquido':
    r"""l[íi]quido
    (?:[ \t]+
    intra[ -]?articular)?""",

    'eco':
    r"""ecos?
    (?:[ \t]+
    (?:endometrial(?:[ \t]+homog[êẽe]neo) | amorfos?))?""",

    'dilatacao':
    r"""dilata[çc][ãâa]o
    (?:[ \t]+(?:das?|de)[ \t]+vias?[ \t]+
    (?:excretoras? | biliar(?:es)?))? |
    dilata[çc][õôo]es""",

    'litiase':
    r"""lit[íi]ases?
    (?:[ \t]+renal)?""",

    'nodulo':
    r"""n[óo]dulos?
    (?:[ \t]+dominante[ \t]+s[óo]lido)?""",

    'figado_homogeneo':
    r"""f[íi]gado[ \t]+homog[êẽe]neo""",
```

```
    'foco':
    r"""foco
    [ \t]+
    (?:hiper|eco)g[êêe]nico""",

    'circulacao_colateral':
    r"""circula[çc][ãâo]o
    [ \t+]
    colateral""",

    'calculo':
    r"""c[áa]lculos?""",

    'rotura':
    r"""rotura""",

    'calcificacao':
    r"""calcifica[çc] (?:[ãâa]o|[õôo]es)""",

    'microcalcificacao':
    r"""microcalcifica[çc] (?:[ãâa]o|[õôo]es)""",

    'derrame':
    r"""derrame
    (?:[ \t]+
    articular |
    peric[áa]rdico |
    pleural)?""",

    'fluxo':
    r"""fluxo""",

    'refluxo':
    r"""refluxo
    [ \t]+
    gastr?oesof[áa]gico""",

    'processo':
    r"""processos?
    [ \t]+
    (?:expansivos? |
    inflamat[óo]rios? |
    estil[óo]ides? |
    infecciosos?)""",

    'tendinose_do_supra_espinhal':
    r"""tendinose
    [ \t]+
    do
    [ \t]+
    supra[ -]?espinhal""",

    'sombra_acustica':
    r"""sombra
    [ \t]+
    ac[úu]stica""",

    'tireoidite':
    r"""tireoidite""",

    'colecistite':
    r"""colecistite""",

    'colecistolitiase':
    r"""colecistolit[íi]ase""",

    'substituicao_adiposa':
    r"""substitui[çc][ãâa]o
    [ \t]+
    adiposa""",
```

```
    'lesao_expansiva':
    r"""les(?:[ãâa]o | [õôo]es)
    [ \t]+
    expansivas?""",

    'feto_unico':
    r"""feto
    [ \t]+
    [úu]nico""",

    'gases_intestinais':
    r"""gases
    (?:[ \t]+
    em
    [ \t]+
    al[çc]as)
    [ \t]+
    intestinais""",

    'adenopatia':
    r"""adenopatias""",

    'batimentos_cardiacos':
    r"""batimentos
    [ \t]+
    card[íi]acos""",

    'imagem':
    r"""imagem
    (?:[ \t]+
    (?:hiper)?ecog[êẽe]nica)""",

    'hidronefrose':
    r"""hidronefrose""",

    'colelitiase':
    r"""colel[íi]t[íi]ase""",

    'conteudo':
    r"""conte[úu]do
    (?:[ \t]+l[íi]quido)?
    [ \t]+
    (?:sonolucente |
    heterog[êẽe]neo |
    sólido-c[íi]stico |
    anec[óo]ico |
    c[íi]stico |
    espesso)""",

    'sedimento':
    r"""sedimentos?""",

    'nefrectomia':
    r"""nefrectomia""",

    'mioma':
    r"""miomas?"""
    }

TOMO_DICT = {
    'deformidade_angular':
    r"""deformidade
    [ \t]+
    angular""",

    'nucleobasais_bilaterais':
    r"""nucleobasais
    [ \t]+
    bilaterais""",

    'infarto':
    r"""infartos?
```

```
    ([ \t]+
    terrr?itorial |
    lacunar |
    antigo
    )""",

    'aneurisma':
    r"""aneurisma""",

    'hematoma':
    r"""hematoma""",

    'contusao':
    r"""contus[ãâa]o
    (?:[ \t]+hemorr[áa]gica)""",

    'lacuna':
    r"""lacuna""",

    'massa':
    r"""massa""",

    'swelling':
    r"""swelling""",

    'leucoaraiose':
    r"""leucoaraiose""",

    'leucoencefalopaia':
    r"""leucoencefalopaia""",

    'cifose':
    r"""cifose""",

    'lordose':
    r"""lordose""",

    'escoliose':
    r"""escoliose""",

    'alteracoes_degenerativas_discais':
    r"""altera[çc][õôo]es
    [ \t]+
    degenerativas[ \t]+
    discais""",

    'aracnoidocele':
    r"""aracnoidocele""",

    'arteriectasia':
    r"""arteriectasia""",

    'artrodese':
    r"""artrodese""",

    'ateromas':
    r"""ateromas?""",

    'hemorragia':
    r"""hemorragia
    (?:[ \t]+
    intra[ -]?ventricular)""",

    'estenose':
    r"""estenose""",

    'sinusopatia':
    r"""sinusopatia""",

    'otomastoidopatia':
    r"""otomastoidopatia""",
```

```
    'calcificacao':
    r"""calcifica[çc](?:[ãâa]o|[õôo]es)""",

    'craniectomia':
    r"""craniectomia""",

    'dilatacao_ventricular':
    r"""dilata[çc][ãâa]o
    [ \t]+ventricular""",

    'desvio_do_septo_nasal':
    r"""desvio
    [ \t]+
    d[oe]
    [ \t]+
    septo
    [ \t]+
    nasal""",

    'encefalomalacia':
    r"""encefalomal[áa]cia""",

    'enfisema':
    r"""enfisema""",

    'espessamento':
    r"""espessamentos?
    [ \t]+
    (?:mucosos? |
    parieta(?:l|is))""",

    'focos_contusionais_hemorragicos':
    r"""focos
    [ \t]+
    contusionais
    [ \t]+
    hemorr[áa]gicos""",

    'herniae':
    r"""h[ée]rnia(?:[çc][ãâa]o)?
    [ \t]+
    subfalcina""",

    'hidrocefalia':
    r"""hidrocefalia""",

    'hipoatenuacao':
    r"""hipoatenua[çc][ãâa]o
    [ \t]+
    (?:sub)?cortical""",

    'atrofia':
    r"""atrofia
    [ \t]+
    (?:cortical |
    encefálica)?""",

    'microangiopatia':
    r"""microangiopatia
    (?:[ \t]+
    difusa |
    (?:sub)?cortical)?""",

    'linfonodomegalia':
    r"""linfonodomegalia"""
    }
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import csv
import re
import itertools

from tcc.preprocess.finding_dicts import ULTRASOUND_DICT, TOMO_DICT
from tcc.common import ClassificationEnum

class ReportEnum:
    """An emulation of an enumeration."""
    ECG, ULTRASSOM, TOMO = range(3)


class FindingIndexer:

    def _pattern_list(self, d):
        pat_l = list()
        for k, v in d.items():
            pattern = re.compile(r"""(?P<id>\b""" + v + r"""\b)""", re.IGNORECASE | re.VERBOSE)
            pat_l.append(pattern)

        return pat_l


    def __init__(self):
        self.ultrasound_patterns = self._pattern_list(ULTRASOUND_DICT)
        self.tomo_patterns = self._pattern_list(TOMO_DICT)

        # WORKAROUND WHILE I DON'T MAKE AN ECG_DICT: ecg patterns file
        ecg_achados = open('/home/fernando/tcc/Corpus/ECG/ecg_achados.txt', 'r')
        ecg_achados_stripped = [pattern_str.strip() for pattern_str in ecg_achados]
        ecg_dict = dict(zip(ecg_achados_stripped, ecg_achados_stripped))
        self.ecg_patterns = self._pattern_list(ecg_dict)



    def index_sentence(self, sentence, patterns):
        """Returns a tuple where the first element is a boolean
        indicating whether or not a finding was matched, and the
        second is the sentence, indexed or not.

        """
        matched = False
        for pattern in patterns:
            if pattern.search(sentence):
                matched = True
                sentence = pattern.sub(r'<achado>\g<id></achado>', sentence)

        return (matched, sentence)


    def index_sentences(self, sents, report_enum):
        indexed_list = list()
        not_indexed_list = list()

        if report_enum == ReportEnum.ECG:
            patterns = self.ecg_patterns
        elif report_enum == ReportEnum.ULTRASSOM:
            patterns = self.ultrasound_patterns
        elif report_enum == ReportEnum.TOMO:
            patterns = self.tomo_patterns

        for sent in sents:
            indexed, sentence = self.index_sentence(sent, patterns)
            indexed_list.append(sentence) if indexed else not_indexed_list.append(sentence)

        return indexed_list, not_indexed_list


    def index_sentences_csv(self, corpus_file, indexed_file, not_indexed_file, report_enum):
```

```
        """Reads sentences from the csv file named corpus_file, and
        saves the indexed sentences in the file named
        indexed_file. Sentences which were not indexed are saved in
        the file named not_indexed_file.

        Arguments:
        corpus_file: corpus filename
        indexed_file: indexed filename
        not_indexed_file: not indexed filename

        """
        sents = list()
        csv_reader = csv.reader(open(corpus_file, 'r'), delimiter='|', quoting=csv.QUOTE_NONE,
escapechar='\\')
        for row in csv_reader:
            try:
                sents.append(row[1].strip())
            except IndexError:
                pass

        indexed, not_indexed = self.index_sentences(sents, report_enum)
        ind = open(indexed_file, 'w')
        not_ind = open(not_indexed_file, 'w')
        ind.writelines(s+'\n' for s in indexed)
        not_ind.writelines(s+'\n' for s in not_indexed)


    def index_sentences_file(self, corpus_file, indexed_file, not_indexed_file, report_enum):
        sents = list(s.strip() for s in open(corpus_file, 'r'))

        indexed, not_indexed = self.index_sentences(sents, report_enum)
        ind = open(indexed_file, 'w')
        not_ind = open(not_indexed_file, 'w')
        ind.writelines(s+'\n' for s in indexed)
        not_ind.writelines(s+'\n' for s in not_indexed)



def sentence_set(csv_file):
    """Returns a sorted list of all the kinds of sentences in the file named csv_file.

    Arguments:
    csv_file: csv filename

    """
    sents_set = set()
    for row in csv.reader(open(csv_file, 'r'), delimiter='|', quoting=csv.QUOTE_NONE, escapechar='\\'):
        try:
            sents_set.add(row[1])

        except IndexError as e:
            print(e.message())

    whitespace_pattern = re.compile(r'[ \t]+')
    keys = [ whitespace_pattern.sub(' ', sent.strip().lower()) for sent in sents_set ]
    sents_dict = dict(zip(keys, sents_set))

    return [sents_dict[key] for key in sorted(sents_dict)]


class AchadoSubst:


    def __init__(self, annotations):
        self.index = 0
        self.annotations = annotations

    def __call__(self, matchobj):
        affirmative = 'afirmativo'
        negative = 'negativo'
        speculative = 'hipotético'
        ambiguous = 'ambíguo'
```

```
        annotation = self.annotations[self.index]
        if annotation == ClassificationEnum.Affirmative:
            verdict = affirmative
        elif annotation == ClassificationEnum.Negative:
            verdict = negative
        elif annotation == ClassificationEnum.Speculative:
            verdict = speculative
        elif annotation == ClassificationEnum.Ambiguous:
            verdict = ambiguous
        else:
            raise ValueError('"annotation" parameter must be one of the ClassificationEnum values.')

        self.index += 1
        return '<achado tipo="' + verdict + '">'


class Annotator:



    def generate_annotator_sentences(self, sentence):
        """Returns a list of annotated sentences. If a sentence has n
        <achado> tags, that sentence will be repeated n times in the
        return list, where each one of these repetitions has only one tag,
        namely the n^th in the sentence.

        """

        pattern = re.compile(r'<achado[^>]*>(.*?)</achado>')
        findings = list()
        match_iter = pattern.finditer(sentence)

        finding_start = 0
        cleaned_sentence = ''
        previous_match_end = 0
        # remove tags
        for match in match_iter:

            finding = match.group(1)
            cleaned_sentence += sentence[previous_match_end:match.start()] + finding
            previous_match_end = match.end()

            start = cleaned_sentence.index(finding, finding_start)
            end = start + len(finding)
            findings.append( (finding, start, end) )

            finding_start = end
            sentence_final_part = sentence[match.end():]

        cleaned_sentence += sentence_final_part

        return [ cleaned_sentence[:s] + '<achado>' + find + '</achado>' + cleaned_sentence[e:] for
find, s, e in findings ]


    def annotate_sentence(self, sentence, annotations):
        """Returns the annotated sentence. For each finding in theIt raises a ValueError if the
sentence does not have a <achado> tag.

        Arguments:
        sentence: a sentence to be annotated.
        annotations: a sequence of Verdicts.

        Pre-condition: the number of findings in the sentence must be equal to the length of
annotations

        """
        assert sentence.count('<achado>') == len(annotations)
```

```
        pattern = re.compile(r'<achado>')
        achadoSubst = AchadoSubst(annotations)
        return pattern.sub(achadoSubst, sentence)


    def generate_annotator_file(self, sample_file, annotator_file):
        """Generates a csv file with sentences from the indexed file
        named annotator_file. The file has two columns, the first one
        being the sentences' number in the sample_file, and the other
        being the sentence. If a sentence has n <achado> tags, that
        sentence will be repeated n times in the file named
        annotator_file, where each one of these repetitions has only
        one tag, namely the n^th in the sentence.

        Arguments:
        sample_file: name of the sample input file
        annotator_file: nameo of the generated file

        """
        annotator_writer = csv.writer(open(annotator_file, 'w'), delimiter='|',
quoting=csv.QUOTE_NONE, escapechar='\\')
        count = 0
        with open(sample_file, 'r') as f:
            for line in f:
                annotator_writer.writerows( [ [count, sent] for sent in
self.generate_annotator_sentences(line) ] )
                count += 1


    def split_sents(self, sample_sentences):
        """Splits the sentences.

        Arguments:

        sample_sentences - a sentences iterable

        """
        annotator_l = list()
        for i, line in enumerate(sample_sentences):
            stripped_line = line.strip()
            annotator_sentences = self.generate_annotator_sentences(stripped_line)
            sentence_tuples = zip([i] * len(annotator_sentences), annotator_sentences)
            annotator_l.extend(sentence_tuples)

        return annotator_l


def merge_annotation_sents(annot_sents):
    finding_pattern = re.compile(r'<achado(?:\s+tipo="\w+"\s*)?>(.*?)</achado>', re.I)
    whitespace_pattern = re.compile(r'[ \t]+')
    remove_tags = lambda x: finding_pattern.sub(r'\1', x)
    def sent_key(s):
        without_tag = remove_tags(s)
        return whitespace_pattern.sub(' ', without_tag)

    groups = list()
    clean_sents = list()
    data = sorted(annot_sents, key=sent_key)
    for k, v in itertools.groupby(data, sent_key):
        clean_sents.append(k)
        groups.append(list(v))

    sample = list()
    for i, g in enumerate(groups):
        sent = clean_sents[i]
        sub_args = list()
        for s in g:
            match = finding_pattern.search(s)
            sub_args.append( (match.group(1), match.group()) )

        for arg in sub_args:
            sent = re.sub(arg[0], arg[1], sent, 1)
```

```
        sample.append(sent)
    return sample

def sort_sents(sents):
    """Returns a sorted list of the sentences.

    Argument:
    sents - an iterable of the sentences.

    """
    attr_pat = re.compile(r'tipo="\w+"', re.IGNORECASE)
    keyfunc = lambda s: attr_pat.sub('', s)
    return sorted(sents, key=keyfunc)

def csv2txt(csv_data):

    tipo = '', 'afirmativo', 'negativo', 'hipotético', 'ambíguo'
    achado_pat = re.compile(r'<\s*achado\s*>')
    def subst(s, n):
        return achado_pat.sub(lambda matchobj: '<achado tipo="' + tipo[n] + '">', s)

    csv_reader = csv.reader(csv_data, delimiter='|', quoting=csv.QUOTE_NONE, escapechar='\\')
    sents = list()
    next(csv_reader)
    for row in csv_reader:
        try:
            number_clas = int(row[1])
            if number_clas >= 5 or number_clas <= 0:
                continue
            sent = row[0].strip()
            sent = subst(sent, number_clas)
            sents.append(sent+'\n')
        except IndexError:
            pass

    return sents


if __name__ == '__main__':
    import sys
    args = sys.argv
    if len(args) >= 3:
        if args[1] == 'csv2txt':
            csv_file = open(args[2], 'r')
            sys.stdout.writelines( s for s in csv2txt(csv_file) )
            csv_file.close()

        elif args[1] == 'sort':
            sents_file = open(args[2], 'r')
            sorted_sents = sort_sents(sents_file)
            sents_file.close()
            sys.stdout.writelines(sorted_sents)

        elif args[1] == 'merge':
            sents_file = open(args[2], 'r')
            merged_sents = merge_annotation_sents(sents_file)
            sents_file.close()
            sys.stdout.writelines(merged_sents)
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import csv
import codecs
import re
import sys

from BeautifulSoup import BeautifulSoup, Tag, NavigableString

from tcc.internals import to_unicode_or_bust

def unicode_csv_reader(csv_data, dialect=csv.excel, **kwargs):
    """
    Generator that wraps csv.reader to handle unicode csv data. The parameter csv_data must be encoded
in UTF-8 or printable ASCII to be safe.
    """
    # csv.py doesn't do Unicode.
    csv_reader = csv.reader(csv_data, dialect=dialect, **kwargs)
    for row in csv_reader:
        # decode UTF-8 to Unicode, cell by cell:
        yield [unicode(cell, 'utf-8') for cell in row]

# for row in unicode_csv_reader(open('laudos.csv', 'r'), delimiter='|', quoting=csv.QUOTE_NONE,
escapechar=r'\'):


def remove_html_tags(report):
    """Remove html tags from the report and returns a list of sentences from it. It uses the following
heuristics: the <br> tag works as a period,
    signaling the end of a sentence. A text within a <p> tag is also a sentence. A <br> has higher
precedence than a <p>, which means that a <br> inside a <p>
    will split <p>'s text in two sentences. All other html tags are meaningless.

    To reach this, it uses the following algorithm: first, for each <p> tag P, add two <br> tags, one
at P's previousSlibing position, and the other at P's nextSibling position.
    Then insert all sub-elements of P in P's parent, and then remove P. After this we'll only have
text and <br> elements in the BeautifulSoup parse tree.

    Arguments:
    report: string containing the report

    """
    try:
        soup = BeautifulSoup(report)
    except:
        sys.stderr.write('nao conseguiu parsear o seguinte laudo: \n' + report)
        return list()

    # process <p> tags
    p_tags_list = soup.findAll('p')
    for el in p_tags_list:
        el_parent = el.parent
        index = el_parent.contents.index(el)
        br1 = Tag(soup, 'br')
        br2 = Tag(soup, 'br')
        el_parent.insert(index, br1)
        el_parent.insert(index+2, br2)


    # remove all tags which are not <br>
    remove_list = soup.findAll(lambda tag: tag.name != u'br')
    for el in remove_list:
        el_parent = el.parent
        index = el_parent.contents.index(el)
        for sub_el in reversed(el.contents[:]):
            el_parent.insert(index, sub_el)
        el.extract()

    # get all NavigableString elements
    sentences = soup.findAll(text=True)
    return sentences
```

```
if __name__ == "__main__":
    import sys
    args = sys.argv
    if len(args) >= 2:
        data = open(args[1]).read()
    else:
        data = sys.stdin.read()

    u_data = to_unicode_or_bust(data)
    reports = u_data.split(u'\n')
    csv_writer = csv.writer(sys.stdout, delimiter='|', quoting=csv.QUOTE_NONE, escapechar='\\',
lineterminator='\n')
    for (id, report) in enumerate(reports):
        csv_writer.writerows([id, codecs.encode(line, 'utf8')] for line in remove_html_tags(report))
```

```python
# -*- coding: utf-8 -*-

import re

class Tokenizer:
    def __init__(self):
        self.finding_pat = re.compile(r'<achado\b[^>]*>(.*?)</achado>')
        self.token_pattern = re.compile(r""" \d+[.,]\w+ |
                                            \( |
                                            \) |
                                            e/ou |
                                            \w+(-\w+)+ |
                                            obst\. | # ultrasound abbreviations
                                            apres\. |
                                            vol\. |
                                            c\. |
                                            g\. |
                                            ii?\. |
                                            x\. |
                                            rep\. | # ecg abbrev.
                                            repol\. |
                                            obs\. |
                                            \w+ |
                                            \S+""", re.VERBOSE | re.IGNORECASE)


    def span_tokenize(self, sent):
        return [ m.span() for m in self.token_pattern.finditer(sent) ]


    def tokenize_findings(self, sent, slices=[]):

        new_slices = slices
        if len(slices) > 0:

            end = len(sent)
            for i, span in reversed(list(enumerate(slices[:]))):
                start = span[1]
                interval = sent[start:end]
                if len(interval) > 0:
                    new_slices[i+1:i+1] = [ (start + m.start(), start + m.end(), 'FIND') for m in
self.finding_pat.finditer(interval) ]

                end = span[0]
            interval = sent[0:end]
            if len(interval) > 0:
                new_slices = [ (m.start(), m.end(), 'FIND') for m in self.finding_pat.finditer
(interval) ] + new_slices
        else:
            new_slices = [ (m.start(), m.end(), 'FIND') for m in self.finding_pat.finditer(sent) ]

        return new_slices

class SentenceTokenizer:
    def __init__(self):
        self._tokenizer = Tokenizer()

    def tokenize(self, text):
        word_tokenized = self._tokenizer.span_tokenize(text)
        sep_indexes = [token[1] for token in word_tokenized if text[slice(*token)] in ['.', '?', '!']]
        sents = list()
        prev = 0
        for idx in sep_indexes:
            sents.append(text[prev:idx].strip())
            prev = idx
        if len(text[prev:]):
            sents.append(text[prev:].strip())

        return sents
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import re
import itertools
import os.path
import csv

from tcc import internals
from tcc.common import ClassificationEnum, CLASSIF_STR_DICT, diacritical_charset_pattern
from tcc.preprocess import tokenizer


# for overlapping scopes, the ascending order of precendence is POSP, PREP, POST, and PREN.
PRECEDENCE_DICT = {
    'POSP' : 0,
    'PREP' : 1,
    'POST' : 2,
    'PREN' : 3
    }

TRIGGER_TAG_DICT = {
    'POSP' : ClassificationEnum.Speculative,
    'PREP' : ClassificationEnum.Speculative,
    'POST' : ClassificationEnum.Negative,
    'PREN' : ClassificationEnum.Negative
    }


class Finding:
    def __init__(self, text, start, end, classification=None):
        self._text = text
        self._start = start
        self._end = end
        # affirmative by default
        self._classification = ClassificationEnum.Affirmative if classification is None else
classification

    def __str__(self):
        return self._text

    def start(self):
        return self._start

    def end(self):
        return self._end

    def classification(self):
        return self._classification

    def set_classification(self, classification):
        self._classification = classification

class Trigger:
    def __init__(self, text, tag):
        self._text = text
        self._tag = tag
        self._classification = TRIGGER_TAG_DICT[self._tag]

    def __str__(self):
        return self._text

    def set_scope(self, scope):
        self._scope = scope

    def set_span(self, s, e):
        self._span = (s, e)

    def scope(self):
        return self._scope

    def span(self):
```

```
        return self._span

    def tag(self):
        return self._tag

    def classification(self):
        return self._classification

class AnnotatedSentence:
    """This class wraps a sentence with information about the
    classification of each of its findings. It also has a sequence of
    the negation and hypothetical triggers in the sentence.

    """
    def __init__(self, sent, findings, triggers, findings_triggers_dict):
        """
        Arguments:

        findings - sequence of findings
        triggers - sequence of triggers
        findings_triggers_dict - a dictionary that maps a finding to its classification-defining
trigger
        """
        self._sentence = sent
        self._findings = findings
        self._triggers = triggers
        self._findings_triggers = findings_triggers_dict

    def sentence(self):
        return self._sentence

    def findings(self):
        return self._findings

    def triggers(self):
        return self._triggers

    def findings_triggers(self):
        return self._findings_triggers

    def __str__(self):
        s = self._sentence
        for f in self._findings:
            start = f.start() + 7
            s = s[:start] + ' tipo="' + CLASSIF_STR_DICT[f.classification()] + '"' + s[start:]
        return s


class Tagger:
    def __init__(self, rules=None, pos_trigger_window=6, precedence=PRECEDENCE_DICT,
trigger_tag_d=TRIGGER_TAG_DICT):
        self.pos_trigger_window = pos_trigger_window
        self.word_tokenizer = tokenizer.Tokenizer()
        self.precedence = precedence
        self.trigger_tag_dict = trigger_tag_d

        self.whitespace_pat = re.compile(r'\s+')
        self.termination_tag_pat = re.compile(r'PSEU|PREN|PREP|POST|POSP|CONJ', re.IGNORECASE)
        self.trigger_tag_pat = re.compile(r'PREN|PREP|POST|POSP', re.IGNORECASE)
        self.filler = '_'
        self.rules = list()
        self.triggers = list()

        if not rules:
            triggers_path = os.path.join(os.path.dirname(__file__), 'termos_disparadores.txt')
            rules = open(triggers_path)

        csv_rule = csv.reader(rules, delimiter='|', quoting=csv.QUOTE_NONE, escapechar='\\')
        next(csv_rule) # skip header
        for rule in csv_rule:
            # Create rules patterns
            trigger = rule[0].strip()
```

```
            trigger_tokens = trigger.split()
            trig = r'\s+'.join(diacritical_charset_pattern(tok) for tok in trigger_tokens)
            rule.append( re.compile(r'\b(' + trig + r')\b', re.IGNORECASE) )
            self.rules.append(rule)


    def _trig_text(self, sent):
        return sent[6:-7]

    def _pren_scope(self, spans, i, token_window):
        if token_window <= 0:
            scope_end = len(spans)
        else:
            scope_end = i + token_window
        trigger_scope = list()
        for tok in spans[i+1:scope_end+1]:
            tagname = tok[2]
            # end the scope if token is termination term or another negation or pseudo-negation term
            if self.termination_tag_pat.match(tagname):
                break

            trigger_scope.append(tok)

        return trigger_scope

    def _prep_scope(self, spans, i, token_window):
        if token_window <= 0:
            scope_end = len(spans)
        else:
            scope_end = i + token_window
        trigger_scope = list()
        for tok in spans[i+1:scope_end+1]:
            tagname = tok[2]
            # end the scope if token is termination term or another negation or pseudo-negation term
            if self.termination_tag_pat.match(tagname):
                break

            trigger_scope.append(tok)

        return trigger_scope

    def _post_scope(self, spans, i):
        trigger_scope = list()
        scope_begin = max(0, i-self.pos_trigger_window)
        for tok in spans[scope_begin:i]:
            tagname = tok[2]
            if self.termination_tag_pat.match(tagname):
                # discard previous tokens from the scope
                trigger_scope = list()
            else:
                trigger_scope.append(tok)

        return trigger_scope

    def _posp_scope(self, spans, i):
        trigger_scope = list()
        scope_begin = max(0, i-self.pos_trigger_window)
        for tok in spans[scope_begin:i]:
            tagname = tok[2]
            if self.termination_tag_pat.match(tagname):
                # discard previous tokens from the scope
                trigger_scope = list()
            else:
                trigger_scope.append(tok)

        return trigger_scope



    def _scope(self, spans, i, tagname, token_window):
        if tagname == 'PREN':
            return self._pren_scope(spans, i, token_window)
```

```
        elif tagname == 'PREP':
            return self._prep_scope(spans, i, token_window)
        elif tagname == 'POST':
            return self._post_scope(spans, i)
        elif tagname == 'POSP':
            return self._posp_scope(spans, i)

    def neg_tag(self, sent):
        neg_tags = list()
        rangesets = list()
        for rule in self.rules:
            rule_tag = rule[1].strip()[1:-1]
            rule_tag = rule_tag.upper()
            if rule_tag != 'CONJ':
                intervals = [m.span() for m in rule[2].finditer(sent)]
                for i in intervals:
                    neg_rangeset = set(range(*i))
                    if all(neg_rangeset.isdisjoint(r) for r in rangesets):
                        # neg_rangeset doesn't overlap with any trigger, thus it can be added to the
list of triggers
                        neg_tags.append((i[0], i[1], rule_tag))
                        rangesets.append(neg_rangeset)

        neg_tags.sort(key=lambda x: x[0])
        return neg_tags


    def annotate(self, sentence, token_window=-1):
        """Returns the sentence annotated with negation classification.

        Arguments:
        sentence - string to be tagged.
        token_window - the number of tokens, to the left or right of the trigger, inside the scope of
the trigger term.
        If it is less than or equal zero, the scope goes until the end or beginning of the sentence.

        """
        self.original_sentence = sentence
        all_spans = list()
        # tokenize all findings
        all_spans = self.word_tokenizer.tokenize_findings(sentence, all_spans)

        # Identify triggers and tag them
        neg_tags = self.neg_tag(sentence)
        all_spans = internals.merge_taglists(all_spans, neg_tags)

        # tokenize the rest
        token_spans = self.word_tokenizer.span_tokenize(sentence)
        all_spans = internals.merge_taglists(all_spans, [(s, e, 'TOKE') for s, e in token_spans])

        triggers = list()
        findings = dict()
        for i, tok in enumerate(all_spans):
            tagname = all_spans[i][2]
            trigger_tag_match = self.trigger_tag_pat.search(tagname)
            if trigger_tag_match:
                trigger_scope = self._scope(all_spans, i, tagname, token_window)
                if len(trigger_scope) > 0:
                    scope_start = trigger_scope[0][0]
                    scope_end = trigger_scope[-1][1]
                    name = sentence[all_spans[i][0]:all_spans[i][1]]
                    trig = Trigger(name, tagname)
                    trig.set_scope(sentence[scope_start:scope_end])
                    trig.set_span(scope_start, scope_end)
                    triggers.append(trig)
            elif tagname == 'FIND':
                f_start = all_spans[i][0]
                f_end = all_spans[i][1]
                f_text = sentence[f_start:f_end]
                f_key = (f_start, f_end)

                findings[f_key] = Finding(f_text, f_start, f_end, ClassificationEnum.Affirmative)
```

```
        t_keyfunc = lambda t: self.precedence[t.tag()]
        triggers.sort(key=t_keyfunc)
        t_group = list()
        for k, g in itertools.groupby(triggers, t_keyfunc):
            t_group.append(list(g))

        findings_triggers = dict()
        # for overlapping scopes, the ascending order of precendence is POSP, PREP, POST, and PREN.
        for g in t_group:
            for t in g:
                t_span = t.span()
                for k, finding in findings.items():
                    if k[0] >= t_span[0] and k[1] <= t_span[1]:
                        # in the scope of t
                        finding.set_classification(t.classification())
                        findings_triggers[finding] = t


        return AnnotatedSentence(sentence, list(findings.values()), triggers, findings_triggers)

def annotate(input_data, output):
    triggers_path = os.path.join(os.path.dirname(__file__), 'termos_disparadores.txt')
    trigger_terms = open(triggers_path)
    tagger = Tagger(trigger_terms)
    annot_sents = list()

    for sent in input_data:
        annot_sent = tagger.annotate(sent.strip())
        annot_sents.append(str(annot_sent))

    output.writelines( s + '\n' for s in annot_sents )


def features(input_data, output):
    word_tokenizer = tokenizer.Tokenizer()
    writer = csv.writer(output)
    triggers_path = os.path.join(os.path.dirname(__file__), 'termos_disparadores.txt')
    trigger_terms = open(triggers_path)
    tagger = Tagger(trigger_terms)


    for sent in input_data:
        all_spans = list()
        # tokenize all findings
        all_spans = word_tokenizer.tokenize_findings(sent, all_spans)

        # Identify triggers and tag them
        all_spans = tagger.neg_tag(sent, all_spans)

        # tokenize the rest
        all_spans = word_tokenizer.span_tokenize(sent, all_spans)

        writer.writerow([tok[2] for tok in all_spans])



if __name__ == '__main__':
    import sys
    args = sys.argv
    if len(args) >= 3:
        option = args[1]
        filename = args[2]
        if option == 'annotate':

            sents = open(filename, 'r')
            annotate(sents, sys.stdout)
            sents.close()
```

```
    elif option == 'features':

        sents = open(filename, 'r')
        features(sents, sys.stdout)
        sents.close()

    else:
        sys.stderr.write('opção ' + option + ' não existe.\n')
        sys.exit(1)

else:
    sys.stderr.write('modo de uso: ' + args[0] + ' conjunto_de_teste.\n')
    sys.exit(1)
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import unittest

from tcc import feature_extract

class FeatureExtractTest(unittest.TestCase):

    def setUp(self):
       self.input = 'Eu tenho <achado tipo="afirmativo">pneumonia</achado>, mas não <achado tipo="negativo">catapora</achado>'

    def test_split(self):
        expected = [
            'Eu tenho <achado tipo="afirmativo">pneumonia</achado>, mas não catapora',
            'Eu tenho pneumonia, mas não <achado tipo="negativo">catapora</achado>'
            ]

        mestre_splinter = feature_extract.split_sent(self.input)
        self.assertEqual(mestre_splinter, expected)

    def test_remove_tag(self):
        expected = ('Eu tenho pneumonia, mas não catapora', (9, 18, 'FIND'), 'afirmativo')
        tupl = feature_extract.remove_tag('Eu tenho <achado tipo="afirmativo">pneumonia</achado>, mas não catapora', train=True)
        self.assertEqual(tupl, expected)

if __name__ == '__main__':
    unittest.main()
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import re
import unittest

from tcc import indexer

class IndexerUtilsTest(unittest.TestCase):

    def test_merge(self):
        sents = [
            'Observa-se <achado>imagem ecogênica</achado> em região interpolar do rim direito, medindo 1,1cm no maior diâmetro, com sombra acústica posterior, consistente com litíase.\n',
            'Observa-se imagem ecogênica em região interpolar do rim direito, medindo 1,1cm no maior diâmetro, com <achado>sombra acústica</achado> posterior, consistente com litíase.\n',
            'Observa-se imagem ecogênica em região interpolar do rim direito, medindo 1,1cm no maior diâmetro, com sombra acústica posterior, consistente com <achado>litíase</achado>.\n',
            '<achado>Cisto</achado> no dorso da mão direita.\n',
            'Observou-se epsódios de <achado>refluxo gastoesofágico</achado> em --- minutos de pesquisa.\n',
            'Não há <achado>nódulos</achado> sólidos ou císticos.\n',
            'Ausência de <achado>nódulo dominante sólido</achado> e/ou cístico de permeio ao parênquima glandular na mama esquerda.\n',
            'Dilatação das vias biliares intra e extra-hepáticas. O colédoco mede cerca de 10,0 mm. Em seu terço distal na cabeça pancreática, observa-se cálculo que mede 7,7 mm. Próximo a este <achado>cálculo</achado>, observa-se ainda outra imagem de menor ecogenicidade e contonos mal definidos, que pode representar lama biliar.\n',
            'Dilatação das vias biliares intra e extra-hepáticas. O colédoco mede cerca de 10,0 mm. Em seu terço distal na cabeça pancreática, observa-se cálculo que mede 7,7 mm. Próximo a este cálculo, observa-se ainda outra imagem de menor <achado>ecogenicidade</achado> e contonos mal definidos, que pode representar lama biliar.\n'
            ]
        expected = set([
            'Observa-se <achado>imagem ecogênica</achado> em região interpolar do rim direito, medindo 1,1cm no maior diâmetro, com <achado>sombra acústica</achado> posterior, consistente com <achado>litíase</achado>.\n',
            '<achado>Cisto</achado> no dorso da mão direita.\n',
            'Observou-se epsódios de <achado>refluxo gastoesofágico</achado> em --- minutos de pesquisa.\n',
            'Não há <achado>nódulos</achado> sólidos ou císticos.\n',
            'Ausência de <achado>nódulo dominante sólido</achado> e/ou cístico de permeio ao parênquima glandular na mama esquerda.\n',
            'Dilatação das vias biliares intra e extra-hepáticas. O colédoco mede cerca de 10,0 mm. Em seu terço distal na cabeça pancreática, observa-se <achado>cálculo</achado> que mede 7,7 mm. Próximo a este cálculo, observa-se ainda outra imagem de menor <achado>ecogenicidade</achado> e contonos mal definidos, que pode representar lama biliar.\n'
            ])

        merged = indexer.merge_annotation_sents(sents)
        m_zip = zip(merged, expected)
        message = '\n' + '\n'.join(m+e for m, e in m_zip)
        self.assertEqual(expected, set(merged), msg=message)


    def test_sort(self):
        unordered_sents = [
            'Dilatação das vias biliares intra e extra-hepáticas. O colédoco mede cerca de 10,0 mm. Em seu terço distal na cabeça pancreática, observa-se <achado>cálculo</achado> que mede 7,7 mm. Próximo a este cálculo, observa-se ainda outra imagem de menor <achado>ecogenicidade</achado> e contonos mal definidos, que pode representar lama biliar.\n',
            'Ausência de <achado>nódulo dominante sólido</achado> e/ou cístico de permeio ao parênquima glandular na mama esquerda.\n',
            'Observa-se <achado>imagem ecogênica</achado> em região interpolar do rim direito, medindo 1,1cm no maior diâmetro, com <achado>sombra acústica</achado> posterior, consistente com <achado>litíase</achado>.\n',
            '<achado>Cisto</achado> no dorso da mão direita.\n',
            'Não há <achado>nódulos</achado> sólidos ou císticos.\n',
            'Observou-se epsódios de <achado>refluxo gastoesofágico</achado> em --- minutos de pesquisa.\n',
            ]
```

```
        expected = [
            '<achado>Cisto</achado> no dorso da mão direita.\n',
            'Ausência de <achado>nódulo dominante sólido</achado> e/ou cístico de permeio ao
parênquima glandular na mama esquerda.\n',
            'Dilatação das vias biliares intra e extra-hepáticas. O colédoco mede cerca de 10,0 mm. Em
seu terço distal na cabeça pancreática, observa-se <achado>cálculo</achado> que mede 7,7 mm. Próximo a
este cálculo, observa-se ainda outra imagem de menor <achado>ecogenicidade</achado> e contonos mal
definidos, que pode representar lama biliar.\n',
            'Não há <achado>nódulos</achado> sólidos ou císticos.\n',
            'Observa-se <achado>imagem ecogênica</achado> em região interpolar do rim direito, medindo
1,1cm no maior diâmetro, com <achado>sombra acústica</achado> posterior, consistente com
<achado>litíase</achado>.\n',
            'Observou-se epsódios de <achado>refluxo gastoesofágico</achado> em --- minutos de
pesquisa.\n',
            ]

        sorted_sents = indexer.sort_sents(unordered_sents)
        m_zip = zip(sorted_sents, expected)
        message = '\n' + '\n'.join(s+e for s, e in m_zip)
        message += '\nsorted:\n\n' + ''.join(s for s in sorted_sents)
        message += '\nexpected:\n\n' + ''.join(s for s in expected)
        self.assertEqual(expected, sorted_sents, msg=message)


    def test_csv2txt(self):
        csv_data = [
            'sentenca|resposta',
            'Relação córtico-medular dos rins mantidas, sem <achado>hidronefrose</achado>. Com foco
ecogênico no complexo ecogênico central|2',
            '12 -<achado>Índice</achado> de resistência da artéria hepática de|5',
            'Não há <achado>dilatação das vias biliares</achado> intra ou extra-hepáticas.|2'
            ]

        expected = [
            'Relação córtico-medular dos rins mantidas, sem <achado tipo="negativo">hidronefrose</
achado>. Com foco ecogênico no complexo ecogênico central\n',
            'Não há <achado tipo="negativo">dilatação das vias biliares</achado> intra ou extra-
hepáticas.\n'
            ]

        txt = indexer.csv2txt(csv_data)
        m_zip = zip(txt, expected)
        message = '\n' + '\n'.join(s+e for s, e in m_zip)
        self.assertEqual(expected, txt, msg=message)



if __name__ == '__main__':
    unittest.main()
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import unittest

import tcc.internals

class InternalsTest(unittest.TestCase):

    def test_merge_neg_tags_and_findings(self):
        sentence = 'Ausência de sinais de rotura, calcificações ou derrame articular.'
        neg_tags = [(0, 8, 'PREN')] # Ausência
        finding_tok = [(22, 28, 'FIND')] # rotura
        expected = [(0, 8, 'PREN'), (22, 28, 'FIND')]
        res = tcc.internals.merge_taglists(finding_tok, neg_tags)
        self.assertEqual(res, expected)

if __name__ == '__main__':
    unittest.main()
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import unittest
import re

from tcc.common import ClassificationEnum, CLASSIF_STR_DICT
from tcc.negex import negex

class NegexTest(unittest.TestCase):

    def setUp(self):
        self.tagger = negex.Tagger()
        self.sent2 = 'Ausência de <achado tipo="negativo">nódulo dominante sólido</achado> e/ou
císticos de permeio ao parênquima glandular mamário.'


    def test_neg_tag(self):
        expected = [(0, 8, 'PREN')]
        neg_tags = self.tagger.neg_tag(self.sent2)
        message = '\n\nexpected = ' + str(expected) + '\nactual = ' + str(neg_tags)
        self.assertEqual(neg_tags, expected, msg=message)


    def test_prep(self):
        prep_sent = 'Bloqueio ramo E com provável <achado>fibrose Inferior</achado>'
        start, end = re.search('<achado>(.*?)</achado>', prep_sent).span()
        annotated_sentence = self.tagger.annotate(prep_sent)
        finds = annotated_sentence.findings()
        self.assert_(len(finds) == 1, 'There must be only one finding in the sentence')
        cls = finds[0].classification()

        message = '\n\n' + '\n'.join('escopo de "' + str(t) + '": ' + t.scope() for t in
annotated_sentence.triggers())

        self.assertEqual(CLASSIF_STR_DICT[cls], 'hipotético', msg=message)


    def test_pren(self):
        pren_sent = 'Sem sinais de <achado>cardiopatia</achado>.'
        annotated_sentence = self.tagger.annotate(pren_sent)
        finds = annotated_sentence.findings()
        self.assertTrue(len(finds) == 1, 'There must be only one finding in the sentence')
        cls = finds[0].classification()

        self.assertEqual(CLASSIF_STR_DICT[cls], 'negativo')

    def test_post(self):
        post_sent = '<achado>Nódulos intracranianos</achado> ausentes.'
        annotated_sentence = self.tagger.annotate(post_sent)
        finds = annotated_sentence.findings()
        self.assertTrue(len(finds) == 1, 'There must be only one finding in the sentence')
        cls = finds[0].classification()

        self.assertEqual(CLASSIF_STR_DICT[cls], 'negativo')



if __name__ == '__main__':
    unittest.main()
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import unittest
from tcc.preprocess import sentence_chunker


class ParseLaudosTest(unittest.TestCase):

    def test_report_with_br_tag(self):
        report = "Ritmo sinusal bradicárdico<br />Alterações difusas e inespecíficas da repolarização ventricular<br />"
        expected = [u"Ritmo sinusal bradicárdico", u"Alterações difusas e inespecíficas da repolarização ventricular"]
        self.assertEqual(expected, sentence_chunker.remove_html_tags(report))

    def test_report_with_p_tag(self):
        report = "<p>ritmo sinusal</p><p>eixo qrs desviado para esquerda</p><p>sobrecarga do VE</p>\n"
        expected = [u"ritmo sinusal", u"eixo qrs desviado para esquerda", u"sobrecarga do VE", u"\n"]
        self.assertEqual(expected, sentence_chunker.remove_html_tags(report))

    def test_report_with_br_inside_p_tag(self):
        report = "<p>ritmo sinusal</p><p>sobrecarga de camaras esquerdas</p><p>isquemia subepicardica lateral e<br> lateral-alta</p>"
        expected = [u"ritmo sinusal", u"sobrecarga de camaras esquerdas", u"isquemia subepicardica lateral e", u" lateral-alta"]
        self.assertEqual(expected, sentence_chunker.remove_html_tags(report))

if __name__ == '__main__':
    unittest.main()
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import unittest

from tcc.preprocess import tokenizer

class TokenizerTest(unittest.TestCase):

    def setUp(self):
        self.sent = 'Ausência de <achado tipo="negativo">nódulo dominante sólido</achado> e/ou
císticos de permeio ao parênquima glandular mamário.'

    def test_span_tokenize(self):
        expected = [(0, 8),
                    (9, 11),
                    (12, 18),
                    (19, 28),
                    (29, 35),
                    (36, 40),
                    (41, 49),
                    (50, 52),
                    (53, 60),
                    (61, 63),
                    (64, 74),
                    (75, 84),
                    (85, 92),
                    (92, 93)]
        tok = tokenizer.Tokenizer()
        test_sent = 'Ausência de nódulo dominante sólido e/ou císticos de permeio ao parênquima
glandular mamário.'
        tokens = [(t[0], t[1]) for t in tok.span_tokenize(test_sent)]

        message = '\n\nExpected:\n' + '\n'.join(self.sent[t[0]:t[1]] for t in expected)
        message += '\n\nTokenized:\n' + '\n'.join(self.sent[t[0]:t[1]] for t in tokens)

        self.assertEqual(expected, tokens, msg=message)

    def test_tokenize_findings(self):
        expected = [(12, 68, 'FIND')]
        tok = tokenizer.Tokenizer()
        slices = tok.tokenize_findings(self.sent)

        self.assertEqual(expected, slices)


if __name__ == '__main__':
    unittest.main()
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sqlite3, psycopg2, collections, random, sys


def insert_amostra(row, cursor):
    insertAmostraStr = 'INSERT INTO amostra_ultrassom (sentenca) VALUES (%s);'
    selectAmostraIdStr = 'SELECT max(id) FROM amostra_ultrassom;'

    original_sent = row.sentenca
    start_tag = '<achado>'
    end_tag = '</achado>'
    achado_start = row.offset
    achado_end = row.offset + row.length
    sent_l = list()
    sent_l.append(original_sent[:achado_start])
    sent_l.append(start_tag)
    sent_l.append(original_sent[achado_start:achado_end])
    sent_l.append(end_tag)
    sent_l.append(original_sent[achado_end:])
    sentence = ''.join(sent_l)

    cursor.execute(insertAmostraStr, (sentence,))
    cursor.execute(selectAmostraIdStr)
    return cursor.fetchone()[0]

def insert_anotacao(row, cursor, id_amostra, avaliador):
    insertAnotacaoStr = 'INSERT INTO anotacao_ultrassom (id_amostra_ultrassom, avaliador,
id_anot_sqlite) VALUES (%s, %s, %s);'
    id_anot_sqlite = str(row.s_id) + '.' + str(row.decs_id) + '.' + str(row.offset)
    cursor.execute(insertAnotacaoStr, (id_amostra, avaliador, id_anot_sqlite))


def main():
    sample = None
    numSampled = 500
    Sentences = collections.namedtuple('Sentences', 's_id decs_id sentenca offset length')
    conn = sqlite3.connect('/home/fernando/laudos_ultrassom.sqlite3')
    with conn:
        cur = conn.cursor()
        sentencesQueryStr = 'SELECT sd.sentencas_id AS s_id, sd.decs_id AS decs_id, s.texto AS
sentenca, sd.offset AS offset, sd.length AS length \
        FROM sentencas AS s \
        INNER JOIN sentencas_decs AS sd ON s.id = sd.sentencas_id \
        WHERE sd.selecionada = 0 \
        ORDER BY sd.sentencas_id'
        cur.execute(sentencesQueryStr)
        queryResult = [Sentences._make(row) for row in cur.fetchall()]

        # selecionar 500 sentenças aleatoriamente
        random.shuffle(queryResult)
        sample = random.sample(queryResult, numSampled)

        # atualizar registros selecionados em sentencas_decs
        for row in sample:
            updateStr = 'UPDATE sentencas_decs SET selecionada = 1 WHERE decs_id = ? AND sentencas_id
= ? AND offset = ?'
            cur.execute(updateStr, (row.decs_id, row.s_id, row.offset))

    cur.close()
    conn.close()
    if not sample:
        print('A amostra não foi coletada.', file=sys.stderr)
        sys.exit(1)

    assert len(sample) == numSampled
    # separar as sentenças de cada avaliador
    num_avaliador_sents = 200
    intersecting_sample_size = num_avaliador_sents // 4

    av1_end = num_avaliador_sents
```

```
    av2_start = av1_end - intersecting_sample_size
    av2_end = av2_start + num_avaliador_sents
    av3_start = av2_end - intersecting_sample_size

    avaliador1 = sample[:av1_end]
    avaliador2 = sample[av2_start:av2_end]
    avaliador3 = sample[av3_start:]


    assert len(avaliador1) == num_avaliador_sents
    msg = '\nlen(avaliador2) == ' + str(len(avaliador2))
    assert len(avaliador2) == num_avaliador_sents, msg
    msg = ' len(avaliador3) é igual a ' + str(len(avaliador3))
    assert len(avaliador3) == num_avaliador_sents, msg
    av2_set = set(avaliador2)
    assert len(set(avaliador1) & av2_set) == intersecting_sample_size
    assert len(av2_set & set(avaliador3)) == intersecting_sample_size

    # insere na base do postgres
    insertAmostraStr = 'INSERT INTO amostra_ultrassom (sentenca) VALUES (%s);'
    selectAmostraIdStr = 'SELECT max(id) FROM amostra_ultrassom;'
    insertAnotacaoStr = 'INSERT INTO anotacao_ultrassom (id_amostra_ultrassom, avaliador,
id_anot_sqlite) VALUES (%s, %s, %s);'
    conn = psycopg2.connect(host='150.162.67.6', database='fcbertoldi', user='pgsql')
    try:
        cur = conn.cursor()
        for i, sent in enumerate(sample):
            if i < av2_start:
                id_amostra_ultrassom = insert_amostra(sent, cur)
                insert_anotacao(sent, cur, id_amostra_ultrassom, 1)

            elif av2_start <= i < av1_end:
                id_amostra_ultrassom = insert_amostra(sent, cur)
                insert_anotacao(sent, cur, id_amostra_ultrassom, 1)
                insert_anotacao(sent, cur, id_amostra_ultrassom, 2)

            elif av1_end <= i < av3_start:
                id_amostra_ultrassom = insert_amostra(sent, cur)
                insert_anotacao(sent, cur, id_amostra_ultrassom, 2)

            elif av3_start <= i < av2_end:
                id_amostra_ultrassom = insert_amostra(sent, cur)
                insert_anotacao(sent, cur, id_amostra_ultrassom, 2)
                insert_anotacao(sent, cur, id_amostra_ultrassom, 3)

            elif i >= av2_end:
                id_amostra_ultrassom = insert_amostra(sent, cur)
                insert_anotacao(sent, cur, id_amostra_ultrassom, 3)

        conn.commit()
    except psycopg2.Error:
        conn.rollback()
        raise
    finally:
        conn.close()



if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from collections import namedtuple
import csv
import functools
import itertools
import operator as op
import os
import re
import subprocess
import sys

import psycopg2

from tcc.common import TAG_PATTERN, ClassificationEnum, CLASSIF_STR_DICT, DIACRITICAL_DICT,
diacritical_charset_pattern
from tcc.negex import negex
from tcc.preprocess import encoding_handlers, tokenizer
from tcc.preprocess.indexer import ReportEnum


class IndexedSentence:
    def __init__(self, sent):
        self._sentence = sent
        self._indexes = list()
        self._rangesets = list()
        self._idx_classif_dict = dict()
        self._idx_id_dict = dict()

    def __str__(self):
        return self._sentence

    def sentence(self):
        return self._sentence

    def indexes(self):
        self._indexes.sort(key=lambda t: t[0])
        return self._indexes

    def idx_classif_dict(self):
        return self._idx_classif_dict

    def idx_id_dict(self):
        return self._idx_id_dict

    def insertClassification(self, idx, classif):
        if idx not in self._indexes:
            raise LookupError

        self._idx_classif_dict[idx] = classif

    def insertIndexId(self, idx, idx_id):
        if idx not in self._indexes:
            raise LookupError

        self._idx_id_dict[idx] = idx_id

    def insertIndex(self, index):
        index_range = set(range(index[0], index[1]))
        if all(index_range.isdisjoint(r) for r in self._rangesets):
            self._indexes.append(index)
            self._rangesets.append(index_range)
            return True
        else:
            return False # failed to insert index

def index_decs(laudo_groups, conn):
    """Index the sentences with DeCS descriptors.

    Argumentos:
```

```
    laudo_groups - sequence of sequences, where each sequence is the set of all sentences in a laudo.
    conn - a psycopg2 connection object
    """
    DecsRecord = namedtuple('DecsRecord', 'id descritor')


    cur = conn.cursor()
    cur.execute("SELECT id, descritor FROM descritor.decs WHERE id LIKE 'A%' OR id LIKE 'B%' OR id
LIKE 'C%' OR id LIKE 'D%' OR id LIKE 'E%'")
    decs_query_result = [DecsRecord._make(row) for row in cur.fetchall()]
    # sort by the number of words in the descriptor, in descending order
    decs_query_result.sort(key=lambda row: len(row.descritor.split()), reverse=True)
    decs_patterns = list()
    for row in decs_query_result:
        regex_str = r'\b' + r'\s+'.join(diacritical_charset_pattern(tok) for tok in row.descritor.split
()) + r'\b'
        decs_patterns.append(re.compile(regex_str, re.I))

    indexed_laudos = list()
    for laudo_l in laudo_groups:
        indexed_laudo = list()
        for sent in laudo_l:
            indexed_sent = IndexedSentence(sent)
            for decs_row, decs_pat in zip(decs_query_result, decs_patterns):
                for match in decs_pat.finditer(sent):
                    index = (match.start(), match.end())
                    insert_index_success = indexed_sent.insertIndex(index)
                    if insert_index_success:
                        indexed_sent.insertIndexId(index, decs_row.id)

            indexed_laudo.append(indexed_sent)

        indexed_laudos.append(indexed_laudo)

    return indexed_laudos

def index_ecg(laudo_groups):
    """Index electrocardiographic sentences with descriptor from SBC

    Argumentos:

    laudo_groups - sequence of sequences, where each sequence is the set of all sentences in a
electrocardiographic laudo.

    """
    # TODO: popular tabela 'termo_cotidiano' nesta função, e inserir os ids com o método insertIndexId
()

def insert_tags(s, indexes):
    indexes.sort(key=lambda t: t[0])
    previous_i_end = 0
    tagged_sent_l = list()
    for i in indexes:

        assert i[0] >= 0 and i[1] >= 0, 'index ' + str(i) + 'anomalous.'

        tagged_sent_l.append(s[previous_i_end:i[0]])
        tagged_i = '<achado>' + s[slice(*i)] + '</achado>'
        tagged_sent_l.append(tagged_i)
        previous_i_end = i[1]

    tagged_sent_l.append(s[previous_i_end:])
    tagged_sent = ''.join(tagged_sent_l)
    without_tags = re.sub(r'<achado>(?P<finding>[^>]*)</achado>', r'\g<finding>', tagged_sent)
    assert without_tags == s, '"' + without_tags + '" is different from original sentence "' + s +
'"\ntagged_sent: ' + tagged_sent + '\n' + repr(indexes)
    return tagged_sent

def main(csv_data, laudo_type):
    """Gera o arquivo xml de laudos separados por sentença. Esta
    função assume que o arquivo csv consiste de duas colunas, a
    primeira com o índice do laudo, e a segunda com o laudo em si.
```

```
    Argumentos:

    csv_data - um iterable de dados em formato csv.
    laudo_type -  um LaudoEnum indicando o tipo de laudo do csv.

    """

    charent_corrected_str = encoding_handlers.substitute_char_entities(csv_data)
    wrongenc_corrected_str = encoding_handlers.substitute_wrong_encoding(charent_corrected_str)
    data = wrongenc_corrected_str.split('\n')
    csv_reader = csv.reader(data, delimiter='|', quoting=csv.QUOTE_NONE, escapechar='\\')
    laudos = list()
    next(csv_reader) # csv header
    for row in csv_reader:
        try:
            laudos.append(row[1])
        except IndexError:
            pass

    # eliminate repeated sentences
    laudos = [k for k, g in itertools.groupby(laudos)]

    formatted_data = bytes('\n'.join(laudos), 'utf8')
    # call python2.*
    sub_env = dict()
    sub_env['PYTHONPATH'] = os.getcwd()
    p = subprocess.Popen(['python2.6', 'tcc/preprocess/sentence_chunker.py'], stdin=subprocess.PIPE,
stdout=subprocess.PIPE, env=sub_env)
    sent_chunker_data = p.communicate(formatted_data)[0]
    sent_chunker_sents = str(sent_chunker_data, 'utf8').split('\n')
    sent_chunker_sents = (s for s in sent_chunker_sents if s) # remove empty lines

    csv_reader = csv.reader(sent_chunker_sents, delimiter='|', quoting=csv.QUOTE_NONE, escapechar='\\')
    csv_reader = list(csv_reader)
    laudo_groups = list()
    for k, g in itertools.groupby(csv_reader, key=lambda row: row[0]):
        laudo_groups.append(list(sent[1] for sent in g))

    tok = tokenizer.SentenceTokenizer()
    tokenized_laudo_groups = list()
    for laudo_group in laudo_groups:
        tokenized_laudo = (tok.tokenize(sent) for sent in laudo_group)
        tokenized_laudo_groups.append(functools.reduce(op.concat, tokenized_laudo, []))

    conn = psycopg2.connect(database='buscas', user='postgres', host='150.162.67.6')
    if laudo_type == 'ecg':
        indexed_laudos = index_ecg(tokenized_laudo_groups)
    else:
        indexed_laudos = index_decs(tokenized_laudo_groups, conn)

    neg_tagger = negex.Tagger()
    for idx_laudo in indexed_laudos:
        for idx_sentence in idx_laudo:
            # surround each index with <achado> tags
            original_sent = idx_sentence.sentence()
            tagged_sent = insert_tags(original_sent, idx_sentence.indexes())
            assert len(re.findall(r'<achado>', tagged_sent)) == len(idx_sentence.indexes()), str
(idx_sentence) + '\n' + tagged_sent
            tagged_sent_indexes = list()
            for m in re.finditer(r'<achado>[^>]*</achado>', tagged_sent):
                tagged_sent_indexes.append((m.start(), m.end()))

            index_dict = dict(zip(tagged_sent_indexes, idx_sentence.indexes()))

            annotated_sent = neg_tagger.annotate(tagged_sent)
            findings = annotated_sent.findings()
            assert len(idx_sentence.indexes()) == len(findings), 'len(idx_sentence.indexes()) == ' +
len(idx_sentence.indexes()) + '\n' + 'len(findings) == ' + len(findings)
            for finding in findings:
                tagged_sent_idx = (finding.start(), finding.end())
                index = index_dict[tagged_sent_idx]
```

```
                idx_sentence.insertClassification(index, CLASSIF_STR_DICT[finding.classification()])

            classif_dict = idx_sentence.idx_classif_dict()
            for idx in idx_sentence.indexes():
                assert idx in classif_dict.keys(), str(idx_sentence) + '\n' + tagged_sent + '\n' + '\n
\n'.join(original_sent[slice(*i)] for i in idx_sentence.indexes())


    cur = conn.cursor()
    insert_laudo_l = list()
    values = dict()
    for idx_laudo in indexed_laudos:
        text_column = ''.join(str(idx_sent) for idx_sent in idx_laudo)
        cur.execute('SELECT nextval(%s);', ('laudo_id_seq',))
        laudo_id = cur.fetchone()[0]
        cur.execute('INSERT INTO laudo (id, texto) VALUES (%s, %s);', (laudo_id, text_column))
        for idx_sent in idx_laudo:
            cur.execute('SELECT nextval(%s);', ('sentenca_id_seq',))
            sentenca_id = cur.fetchone()[0]
            values['id_sentenca'] = sentenca_id
            sent_str = str(idx_sent)
            cur.execute('INSERT INTO sentenca (id, id_laudo, texto) VALUES (%s, %s, %s);',
(sentenca_id, laudo_id, sent_str))

            idx_id_dict = idx_sent.idx_id_dict()
            idx_classif_dict = idx_sent.idx_classif_dict()
            for index in idx_sent.indexes():
                values['id_termo_cotidiano'] = idx_id_dict[index]
                values['id_decs'] = idx_id_dict[index]
                values['inicio'] = index[0]
                values['fim'] = index[1]
                if laudo_type == 'ecg':
                    cur.execute('INSERT INTO sentenca_termo_cotidiano (id_sentenca,
id_termo_cotidiano, inicio, fim) VALUES (%(id_sentenca)s, %(id_termo_cotidiano)s, %(inicio)s, %
(fim)s);', values)
                else:
                    cur.execute('INSERT INTO sentenca_decs (id_sentenca, id_decs, inicio, fim) VALUES
(%(id_sentenca)s, %(id_decs)s, %(inicio)s, %(fim)s);', values)

                classif = idx_classif_dict[index]
                idx_str = sent_str[slice(*index)]
                if classif == 'negativo':
                    cur.execute('SELECT id FROM expressao_negada WHERE texto = %s', (idx_str,))
                    en_row = cur.fetchone() # fetchone() returns None if there are no rows
                    if not en_row:
                        cur.execute("SELECT nextval(%s);", ('expressao_negada_id_seq',))
                        expr_negada_id = cur.fetchone()[0]
                        cur.execute('INSERT INTO expressao_negada (id, texto) VALUES (%s, %s);',
(expr_negada_id, idx_str))
                    else:
                        expr_negada_id = en_row[0]

                    values['id_expressao_negada'] = expr_negada_id
                    cur.execute('INSERT INTO sentenca_expressao_negada (id_sentenca,
id_expressao_negada, inicio, fim) VALUES (%(id_sentenca)s, %(id_expressao_negada)s, %(inicio)s, %
(fim)s);', values)

                elif classif == 'hipotético':
                    cur.execute('SELECT id FROM expressao_hipotetica WHERE texto = %s', (idx_str,))
                    eh_row = cur.fetchone() # fetchone() returns None if there are no rows
                    if not eh_row:
                        cur.execute("SELECT nextval(%s);", ('expressao_hipotetica_id_seq',))
                        expr_hipotetica_id = cur.fetchone()[0]
                        cur.execute('INSERT INTO expressao_hipotetica (id, texto) VALUES (%s, %s);',
(expr_hipotetica_id, idx_str))
                    else:
                        expr_hipotetica_id = eh_row[0]

                    values['id_expressao_hipotetica'] = expr_hipotetica_id
                    cur.execute('INSERT INTO sentenca_expressao_hipotetica (id_sentenca,
id_expressao_hipotetica, inicio, fim) VALUES (%(id_sentenca)s, %(id_expressao_hipotetica)s, %
(inicio)s, %(fim)s);', values)
```

```
    conn.commit()




if __name__ == "__main__":
    args = sys.argv
    if len(args) >= 3:
        laudo_type = args[1]
        data = open(args[2], 'r')
        if laudo_type == 'ecg':
            laudo_type = ReportEnum.ECG
        elif laudo_type == 'ultrassom':
            laudo_type = ReportEnum.ULTRASSOM
        elif laudo_type == 'tomo':
            laudo_type = ReportEnum.TOMO
        else:
            print('Argumento ', laudo_type, ' é inválido. Opções válidas: ecg, ultrassom, ou tomo.',
file=sys.stderr)
            sys.exit(1)

        print(main(data, laudo_type), end='')
    else:
        print('Faltou parâmetros\nModo de uso : ', args[0], ' tipo_de_laudo [arquivo_csv]\nlaudos
podem ser ecg, ultrassom ou tomo', file=sys.stderr)
        sys.exit(1)
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os, subprocess, sqlite3, csv, itertools, functools
import operator as op

from collections import namedtuple
from tcc.preprocess import encoding_handlers, tokenizer

def main():
    conn = sqlite3.connect('/home/fernando/laudos_ultrassom.sqlite3')
    cur = conn.cursor()
    cur.execute('SELECT id, laudo FROM laudos;')

    LaudosRecord = namedtuple('LaudosRecord', 'id laudo')
    laudos_query_result = [LaudosRecord._make(row) for row in cur]
    laudos_query_result = [row for row in laudos_query_result if row.laudo] # remove empty laudos
    laudos_corrigidos = list()
    for r in laudos_query_result:
        charent_corrected_str = encoding_handlers.substitute_char_entities(r.laudo)
        wrongenc_corrected_str = encoding_handlers.substitute_wrong_encoding(charent_corrected_str)
        laudos_corrigidos.append((r.id, wrongenc_corrected_str))

#    formatted_data = bytes('\n'.join(r[1] for r in laudos_corrigidos), 'utf8')

    sub_env = dict()
    sub_env['PYTHONPATH'] = os.getcwd()
    laudo_groups = list()
    for laudo in laudos_corrigidos:
        # call python2.*
        formatted_data = bytes(laudo[1], 'utf8')
        p = subprocess.Popen(['python2.6', 'tcc/preprocess/sentence_chunker.py'],
stdin=subprocess.PIPE, stdout=subprocess.PIPE, env=sub_env)
        sent_chunker_data, stderrdata = p.communicate(formatted_data)
        if stderrdata:
            print(stderrdata, laudo)

        sent_chunker_sents = str(sent_chunker_data, 'utf8').split('\n')
        sent_chunker_sents = (s for s in sent_chunker_sents if s) # remove empty lines

        csv_reader = csv.reader(sent_chunker_sents, delimiter='|', quoting=csv.QUOTE_NONE,
escapechar='\\')
        csv_reader = list(csv_reader)
        laudo_groups.append(list(sent[1] for sent in csv_reader))

    ## for k, g in itertools.groupby(csv_reader, key=lambda row: row[0]):
    ##     laudo_groups.append(list(sent[1] for sent in g))

    ## print('len(laudo_groups) = ', len(laudo_groups))
    ## print('len(laudos_corrigidos) = ', len(laudos_corrigidos))
    assert len(laudo_groups) == len(laudos_corrigidos)

    tok = tokenizer.SentenceTokenizer()
    tokenized_laudo_groups = list()
    for laudo_group in laudo_groups:
        tokenized_laudo = (tok.tokenize(sent) for sent in laudo_group)
        tokenized_laudo_groups.append(functools.reduce(op.concat, tokenized_laudo, []))

    assert len(tokenized_laudo_groups) == len(laudos_corrigidos)

    db_ids = (row[0] for row in laudos_corrigidos)
    for laudo_group, db_id in zip(tokenized_laudo_groups, db_ids):
        for sent in laudo_group:
            cur.execute('INSERT INTO sentencas VALUES (NULL, ?, ?)', (db_id, sent))

    conn.commit()


if __name__ == '__main__':
    main()
```