Identificando *Commented-out Code* em Repositórios de Software

Vitor Botelho Vaz de Melo[1], **André Hora**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

{vitormelo,andrehora}@dcc.ufmg.br

Resumo. *Este trabalho realiza um estudo abrangente da prática commented-out code em diversos repositórios de software livre. Remover código através de comentários é uma má prática de programação e pode causar problemas como diminuição da legibilidade, perda de produtividade, entre outros. No trabalho foi desenvolvida uma ferramenta baseada em redes neurais recorrentes (LSTM) para identificar entre as linhas de comentários quais delas são códigos removidos por comentário. E por fim verificamos que 50% dos maiores e mais bem avaliados repositórios de software tem até 5,22% do total de comentários identificadas como commented-out code.*

1. Introdução

À medida que a Ciência da Computação avança e milhões de linhas de códigos são escritas, diariamente e em diversas linguagens de programação, os desenvolvedores percebem que, para alcançar sucesso a médio e longo prazo em seus projetos, é necessário utilizar-se, cada vez mais, das melhores ferramentas e práticas de programação.

Uma prática de programação muito comum ao se desenvolver softwares é a de ”remover” uma linha ou um bloco de código tornando-os comentários, sendo conhecida como commented-out code (Figura 1). Comentar uma linha de código pode ser útil, tanto que os próprios ambientes de programação (IDE’s e editores de texto) oferecem atalhos fáceis para isso. Contudo, esta prática se torna um verdadeiro problema quando estes códigos comentados estão inseridos em um sistema grande, complexo e mantido por diversas pessoas [Martin 2009].

```
61      h2 = hashlib.sha512(word2.encode('utf-8')).hexdigest()
62
63  #    print('Hash1 [{}]:\n{}\n\nHash2 [{}]:\n{}\n'.format(word1, h1, word2, h2))
64
65  #    # Use list so we can shuffle it
66  #    rez = []
67  #    for a, b in zip(h1, h2):
68  #        rez.append(chr(ord(a) ^ ord(b)))
69
70  #    #print(str(base64.a85encode(''.join(rez).encode())))
71
72  #    # Shuffle three times for good measure
73  #    random.shuffle(rez)
74  #    random.shuffle(rez)
75      random.shuffle(rez)
76
```

Figura 1. Commented-out code example

A inserção de código comentado nos arquivos fontes de um sistema pode causar redução da legibilidade, distração e perda de tempo. Martin, em seu livro clássico Clean Code, enfatiza *Few practices are as odious as commenting-out code. Don't do this!*, ele argumenta que outras pessoas não terão coragem de deletar este código por achar que ele pode ser importante [Martin 2009].

Além disso, quando um mantenedor se depara com códigos comentados ele pode ter uma série de dúvidas, como *"Por que este código está comentado? Isso é útil em alguma função? Este código está relacionado com determinada mudança?"*, entre outras questões específicas para cada sistema. Este tipo de reflexão pode no mínimo ser uma perda de tempo, ou até mesmo uma distração para introduzir bugs no sistema [Martin 2009].

Este trabalho tem como principal objetivo fazer um estudo abrangente desta prática em repositórios de código aberto. No entanto, existem poucos estudos científicos abordando esta problemática e não há, entre as ferramentas mais populares, uma solução que identifica automaticamente este tipo de comentário [Grijó and Hora 2017]. Portanto, este trabalho descreve o desenvolvimento de uma ferramenta apropriada utilizando técnicas de aprendizado de máquina. Em sequência é analisado os diversos repositórios de software.

2. Trabalhos Anteriores

GRIJÓ L. e HORA A. exploraram o problema da inserção de *commented-out code* e demonstraram seus interessantes resultados no artigo Minerando Código Comentado [Grijó and Hora 2017]. No trabalho foi desenvolvido um parser heurístico que identifica *commented-out code* para a linguagem java que obteve uma precisão de 83%. Com esta ferramenta foi identificado que alguns sistemas possuem como mediana 4,17% de taxa de commented-out code em relação ao total de comentários, com alguns sistemas chegando a 30%.

Este trabalho propõe uma extensão ou continuação desse estudo, a fim de validar e aprofundar o conhecimento sobre os resultados obtidos. Para isso, a primeira etapa foi desenvolver uma ferramenta com maior precisão na identificação de *commented-out code* e que seja multi linguagem.

3. Modelo de Identificação de *Commented-out Code*

Separar o que é código removido por comentário do que é comentário real, não é uma tarefa fácil. Se considerarmos ferramentas como parsers de linguagem ou mesmo expressões regulares, o identificador pode sofrer com algumas situações, como:

1 Códigos não finalizados
2 Códigos velhos e que não compila mais
3 Código misturado com textos, etc.

Além disso, essas ferramentas teriam que ser implementadas considerando uma linguagem alvo, requerindo um esforço muito alto de desenvolvimento de software para fazer um estudo multi-linguagem, consequentemente impossibilitando este estudo acadêmico. Por outro lado, modelos de classificação podem aprender a separar as linhas

de código de comentários a partir de exemplos [Duda et al. 2015]. E esses exemplos temos disponíveis em grande quantidade nos repositórios de código livre em ferramentas como o github[1].

3.1. Definição do Modelo de Classificação

Para realizar a tarefa de classificação foi utilizado uma rede neuronal profunda com a arquitetura LSTM (Long short-term memory) [Gers et al. 1999] (Figura 2). Essa é uma rede neural recorrente, o que permite criar uma estrutura ”temporal” de recebimento da entrada.

A entrada do modelo definido é o equivalente a uma linha de um código fonte, considerando que essa linha pode ser um comentário ou código. O modelo recebe essa linha caractere por caractere. A intuição por tras dessa escolha está no fato de que o código utiliza mais de caracteres especiais como ’{’, ’}’, ’(’, ’)’, ’=’, entre outros. Além disso, a arquitetura recorrente permite que a rede aprenda as estruturas sequenciais que os caracteres podem formar, como por exemplo a chamada de uma função: ”f(x)”.

O modelo conta ainda com uma camada de embedding que permite o aprendizado de uma representação distribuída dos caracteres [Mikolov et al. 2013]. Dessa forma os caracteres podem ter uma certa semântica ou peso para o modelo, facilitando o aprendizado como um todo. E ao final temos uma camada totalmente conectada e a camada de saída, com função de ativação sigmoid [Zurada 1992].

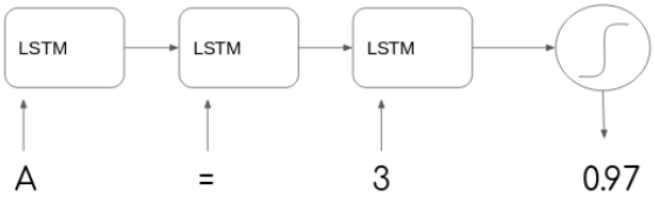


Figura 2. Arquitetura do modelo simplificada - no modelo desenvolvido cada caractere é recebido como entrada por uma LSTM de forma sequencial, ao final obtemos uma saída com a função de ativação sigmoid em que o resultado é uma predição para a linha de código entre 0 e 1, no qual consideramos que quanto mais próximo de 1, maior a probabilidade da linha ser realmente código

4. Obtenção e pré-processamento dos dados

Antes de treinar o modelo foi necessário a construção de uma base de dados de treino. Para a constituição da base foram realizadas as atividades descritas nas seções seguintes.

4.1. Download de Repositórios de Software

A primeira etapa foi realizar o download do código fonte de diversos repositórios. Para isso foi utilizado a api do github para escolher os repositórios de código aberto com uma boa evolução, que seja ativo, e de forma a construir uma base com uma variabilidade de estilo de código, e linguagem. Isso se traduziu nos seguintes critérios adotados para fazer o download dos dados:

[1]https://github.com/

- Pegar os repositórios mais populares por linguagem de programação (com mais estrelas)
- O primeiro commit deve ter mais de 2 anos
- O último commit deve ter menos de 1 mês

As linguagens utilizadas foram C, Java, Javascript e Typescript.

4.2. Separação de comentários e código

Nesta etapa pré processamos todos os arquivos fontes e os separamos em dois arquivos. Um arquivo restando somente linhas de código e outro somente comentários. Foi utilizado uma expressão regular para fazer essa separação (Figura 3).

```
(one_line_separator.*)
|('(?:\\[^']|\\'|.)*?')
|("(?:\\[^\"]|\\"|.)*?")
|(init_line_separator(?:.|[\n\r])*?end_line_separator)
```

Figura 3. Expressão regular utilizada para separar código fonte em comentários e código. Este regex tem três partes principais, a primeira identifica os comentários de uma linha, a segunda exclui os possíveis casos de uso do indicador de comentário dentro de uma string, e a terceira identifica os comentários de bloco

Na Figura 4 podemos observar um exemplo deste processamento. Repare que neste momento ainda não identificamos os códigos removidos por comentário.

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
//   InputStream resultsStream = formatter.getResultStream();
//   StreamReader reader = new StreamReader(resultsStream);
//   response.setContent(reader.read(formatter.getByteCount()));

 InputStreamResponse response = new InputStreamResponse();
 response.setBody(formatter.getResultStream(), formatter.getByteCount());
 //   InputStream resultsStream = formatter.getResultStrea
 //   StreamReader reader = new StreamReader(resultsStream
 //   response.setContent(reader.read(formatter.getByteCou
```

Figura 4. Exemplo de código fonte separado em duas partes (código e comentários)

4.3. Definição dos dados de treino

Após a separação em código e comentário foi definido o modelo final para os dados de treino, no qual para cada linha temos se é um comentário ’0’ ou código ’1’ (Tabela 1). Além disso, foi filtrado linhas com menos de 3 caracteres, ou vazias. Dessa forma evitamos muitas linhas que são só abertura ou fechamento de escopo, entre outras.

Temos ao final, portanto, uma base que usa diversas linhas de código, e diversos comentários reais. Dessa forma, o objetivo de nosso modelo vai ser simplesmente aprender o que é código e o que é um comentário, identificar os *commented-out code* será um passo secundário.

Tabela 1. Amostra dos dados de treino final

Language	Line	Is Code
Javascript	return output;	1.0
C++	if (visitable(sub)) {	1.0
C	loss of use, data, or profits; or business i...	0.0

Tabela 2. Proporção da base de dados por linguagem

Language	(%)
Java	25.4
Javascript	24.5
Typescript	18.0
C	23.4
C++	3.0

A base de treino ficou com cerca de 12 milhões de linhas de código, com 19,5% sendo comentário e 80,5% código. Além disso, temos a seguinte proporção de linhas por linguagem.

E para cada linguagem temos a proporção de comentários e de código.

Tabela 3. Proporção de comentários e código por linguagem

Language	Código (%)	Comentário (%)
Java	79.4	20.6
Javascript	83.0	17.0
Typescript	70.8	29.2
C	86.9	13.1
C++	79.2	20.8

4.4. Treinamento e Avaliação

Nesta etapa os dados foram separados aleatoriamente em dados de treino (80%) e em dados de teste (20%). Após o treino durante 30 épocas, o modelo obteve uma acurácia de 98% tanto para os dados de treino quanto para os dados de teste.

Na figura 5 podemos observar como o modelo se comporta para diferentes níveis de treshold da saída. A ROC é quase perfeita, mostrando que o modelo separa muito bem os códigos e comentários. Podemos observar também a acurácia, a precisão e a revocação para alguns tresholds. Para o modelo final foi escolhido o treshold em 0.6, pois mantém a melhor acurácia e maior precisão.

Já com o treshold definido em 0.6 podemos observar na figura 6 a matriz de confusão, com os valores detalhados predição para uma amostra de trinta mil dados de teste. Em sequência podemos observar a acurácia, precisão e revocação para as diferentes linguagens, a linguagem Typescript se destaca com relação às outras, com uma acurácia e

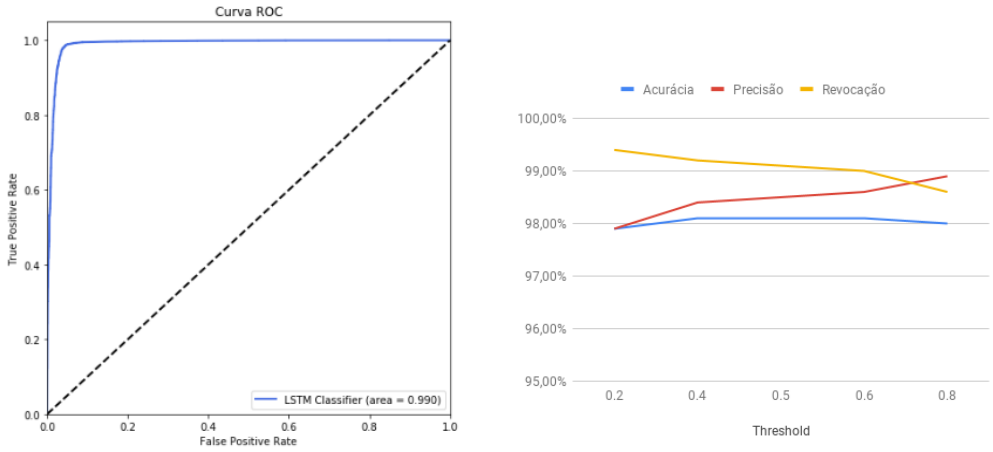


Figura 5. Acurácia, Precisão e Revocação para diferentes tresholds na saída e Curva ROC (Receiver Operator Curve)

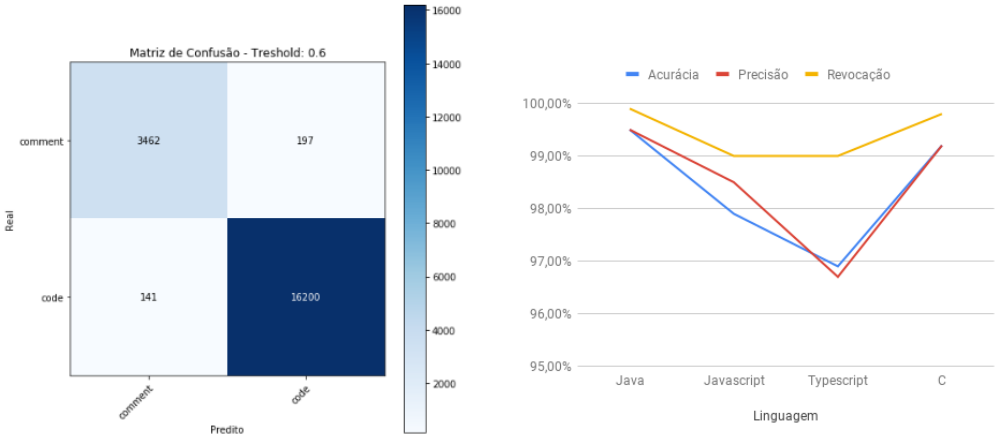


Figura 6. Matriz de confusão e estatísticas para as diferentes linguages

precisão mais próximas de 97%, porém esse valor faz sentido uma vez que essa linguagem tem uma proporção maior de comentários (Tabela 3). Além disso, o modelo erra mais comentários, classificando cerca de 5.6% dos comentários como código.

No entanto, quando olhamos para esses erros (Tabela 4) encontramos os *commented-out code*. Mostrando mais uma vez que o modelo foi capaz de aprender o que é código, e agora podemos separar estes ”ruídos” dos comentários. Além disso, a taxa de erro de 5.6% é uma boa aproximação da taxa de *commented-out code* nessa base, se aproximando da mediana de 4.17% encontrado por [Grijó and Hora 2017].

Tabela 4. Linhas consideradas comentários na base de dados que o modelo classificou como código

Line	Is code	Predito
system.out.println(sw.gettotaltimemillis());	0.0	0.981
mydomainclass myobject = unmarshaller.unmarshal(source);	0.0	0.995
public jmshandlermethodfactory myjmshandlermethodfactory()	0.0	0.998

5. Extração de informação

Nesta etapa podemos combinar as ferramentas desenvolvidas para calcular como a prática de remover código por comentário afeta os repositórios de software selecionados no github. A Figura 7 ilustra os passos de execução para extrairmos essas informações.

Na seção anterior descrevemos como foi feita a seleção de repositórios, como separamos um arquivo em linhas de código e comentário, e mostramos o Treinamento de um classificador para a identificação de *commented-out code* que será utilizado nessa fase. Outro passo importante nesta etapa será a variação da versão de um mesmo projeto, de forma que seja possível identificar como essa prática afetou os repositórios ao longo da evolução do código.

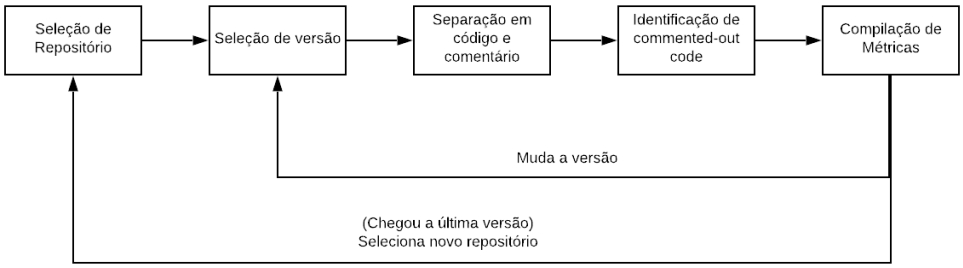


Figura 7. Pipeline de execução do algoritmo para identificar *commented-out code* e computar as métricas por projeto.

5.1. Seleção de versão

Foi possível selecionar as versões através do recurso *git tag*[2] que lista as versões que foram divulgadas para um repositório. Foram avaliadas seis versões de cada projeto, sendo que a primeira versão é a mais antiga e a sexta a mais nova.

5.2. Taxa de *Commented-out code*

Para compreender como a má prática de remover código comentado calculamos por projeto e por versão a taxa de *commented-out code* que seria:

$$rate = \frac{TotalCommentedOutCode}{TotalComentarios}$$

5.3. Evolução da taxa ao longo das versões

Para avaliar como a taxa de *commented-out code* varia ao longo das versões dos repositórios de software podemos observar a figura 8. A primeira vista chama a atenção alguns outliers identificados, que chegam a aproxidamente a 80% de *commented-out code*. No entanto, a mediana tem pouca variação e fica entre 5% e 6% aproximadamente. Além disso, podemos observar uma dimnuição na caixa, representando uma diminuição no 3º quartil. Isso pode indicar que projetos que tinham taxas maiores estão se preocupando mais com essa prática.

Este comportamento também foi identificado por um algoritmo de agrupamento (K-means [Hamerly and Elkan 2004]) sendo encontrado 3 grupos, como pode ser observado na figura 9. Foi identificado que cerca de 70% dos projetos tem um comportamento

[2]https://git-scm.com/book/en/v2/Git-Basics-Tagging

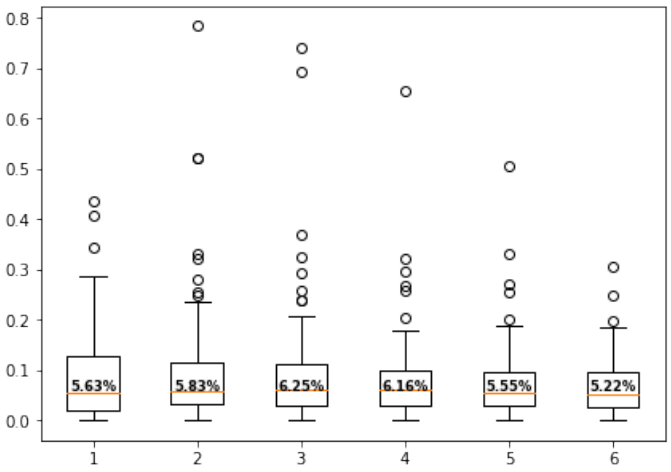


Figura 8. Gráfico boxplot da variação da taxa de *commented-out code* por projeto ao longo das versões

bem próximo ao da mediana identificada no gráfico acima, e os outliers identificados aparecem em um grupo com cerca de 4% dos projetos, percebemos que o comportamento deles foi de aumentar a taxa nas versões 2 e 3, e diminuir em sequência. Ainda identificamos um terceiro grupo que apresenta uma taxa mais constante, porém mais alta, com os centros próximos de 15%.

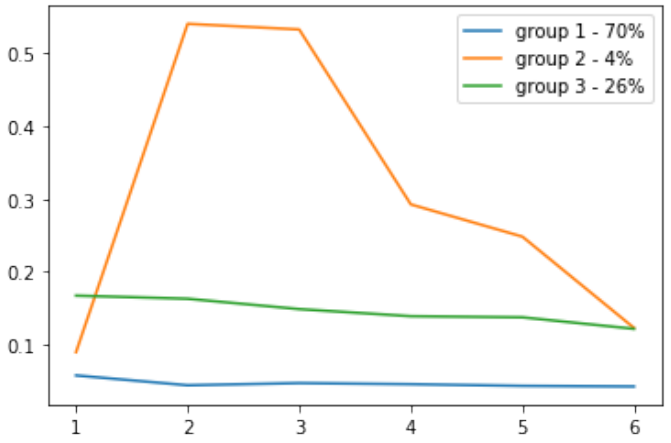
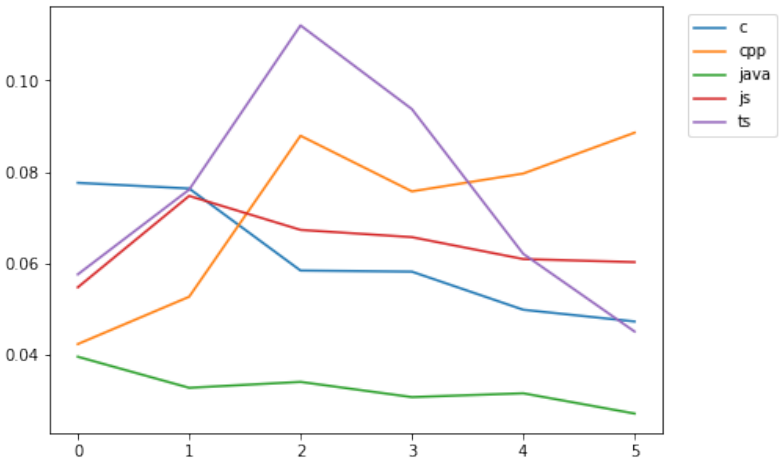


Figura 9. Representação dos grupos identificados pelo algoritmo K-means para k=3

5.4. Avaliação por linguagem de programação

Na Figura 10 podemos observar como a taxa de *commented-out code* varia para as diferentes linguagens avaliadas. Neste gráfico é mostrado a mediana da taxa de commented-out code dos projetos por linguagem. É observado que as taxas variam bastante para diferentes linguagens, no qual a linguagem Typescript teve o maior pico, mas melhorou muito na última versão. Java manteve a melhor taxa em todas as versões e sempre com uma tendência de diminuir, enquanto que C++ começou com a segunda menor taxa mas acabou com a maior.



groups

Figura 10. Variação da taxa avaliada por linguagem

6. Considerações Finais

Os resultados mostraram que as taxas de *commented-out code* podem ser bem heterogênias quando analisamos diferentes projetos e linguagens. No geral foi observado uma diminuição da taxa nas últimas versões dos projetos o que está de acordo com as observações em [Grijó and Hora 2017]. Espera-se que cada vez mais os desenvolvedores de software se preocupem em não manter essa prática, e que os repositórios de software tenham cada vez menos *commented-out code* em suas linhas de código. Como trabalho futuro seria importante entender a relação que essa prática tem com outras métricas ou práticas identificadas pela engenharia de software e pela mineração de repositórios de software como LOC, quantidade de commits, etc. [Tornhill 2015].

Referências

Duda, R. O., Hart, P. E., and Stork, D. G. (2015). *Pattern Classification*.

Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm.

Grijó, L. and Hora, A. (2017). Minerando codigo comentado. In *6th Brazilian Workshop on Software Visualization, Evolution and Maintenance*, pages 1–7. VEM.

Hamerly, G. and Elkan, C. (2004). Learning the k in k-means. In *Advances in neural information processing systems*, pages 281–288.

Martin, R. C. (2009). *Clean Code: a handbook of agile software craftsmanship*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Tornhill, A. (2015). *Your Code as Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs*.

Zurada, J. M. (1992). *Introduction to artificial neural systems*, volume 8. West publishing company St. Paul.