Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Análise comparativa de técnicas de seleção de protótipos

Dayvid Victor Rodrigues de Oliveira

Trabalho de Graduação

Recife
22 de novembro de 2011

Universidade Federal de Pernambuco
Centro de Informática

Dayvid Victor Rodrigues de Oliveira

Análise comparativa de técnicas de seleção de protótipos

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Prof. Dr. George Darmiton*

Recife
22 de novembro de 2011

Eu dedico este trabalho a João Rodrigues de Silva, meu avô.

Agradecimentos

Agradeço a Deus.

What can I give back to God, for the blessings You pour out on me?
—BONO (Boston, 2001)

Resumo

RESUMO

Palavras-chave: PORTUGUES

Abstract

ABSTRACT

Keywords: INGLES

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

1.1 Motivação e Contextualização

Classificadores são (...)

1.1.1 Histórico

No final dos anos 50, surgiram os primeiros trabalhos de aprendizagem de máquina. De uma forma geral, elas consistiam em dar ao computador a habilidade de reconhecer formas. A partir daí, surgiram diversos problemas onde a aprendizagem de máquina atuava.

Um dos problemas que são importantes para esse trabalho, é o problema de classificação, que consiste em agrupar dados de acordo com suas características de forma que seja possível extrair informação útil desdes agrupamentos. Um outro problema é a discriminação, que consiste em achar uma forma de reconhecer um conceito, dado um conjunto de conceitos exemplos. O terceiro e último problema geral é o da generalização, que é o problema de como reduzir uma regra de forma a ser mais abrangente e menos custosa.

Reconhecimento de padrões ataca principalmente o problema da discriminação, tendo por objetivo classificar padrões, podendo ser os padrões pertencentes a qualquer domínio, como reconhecimento de digitais, gestos, escrita, fala, entre outros.

Todo sistema de reconhecimento de padrões utiliza um classificador para discriminar os padrões de teste. O quanto um dado classificador é eficiente é medido pela taxa de acerto média, pela variância, e pela eficiência em termos de custo computacional. Um classificador de aprendizagem baseada em instâncias muito utilizado é o *K-Nearest Neighbor*, KNN [PI69]. O KNN é muito utilizado por ser um método de aprendizagem supervisionado simples, e por possuir uma taxa de acerto relativamente alta. O conceito básico consiste em, dado um padrão x a ser classificado e um conjunto de padrões conhecidos T, obter os K elementos de T mais próximos de x, a classe de maior ocorrência, ou peso, entre os K elementos será a classe de x.

Algorithm 1 KNN

Require: K: uma lista
Require: T: conjunto de treinamento
Require: x: elemento para ser classificado
Require: L: uma lista

1. **for all** $t_i \in T$ **do**
2. $d_i = distance(t_i, x)$
3. adicione $(d_i, Classe(t_i))$ em L
4. **end for**
5. $Ordene(L)$ de acordo com as distâncias
6. obtenha os K primeiros elementos de L
7. **return** a classe de maior ocorrência, ou peso, entre os K

Conforme mostrado em Algorithm 1, o KNN é muito simples, porém, possui um custo alto, pois precisa visitar todos os elementos da base de dados para realizar uma classificação. Este problema é o mesmo problema de agrupamento (classificação) e generalização. Este problema é tratado na próxima subsessão.

1.1.2 Seleção de Protótipos

A estratégia do KNN, apesar de eficiente, possui algumas desvantagens. A primeira desvantagem é que o KNN é sensível á ruídos, para baixos valores de K. Outra desvantagem é que o KNN é custoso, pois precisa calcular a distância do padrão que se deseja classificar para cada um dos padrões da base de treinamento, com isso, o KNN torna-se lento em relação a outros classificadores.

Para resolver esse problema, surgiu a idéia de utilizar um conjunto menor, gerado a partir da base de dados original (conjunto de treinamento), que representem bem todas as classes, este processo é chamado de seleção de protótipos. A escolha desdes protótipos deve ser feita cuidadosamente, pois, é necessário que estes elementos possuam uma boa representatividade de todo o conjunto de treinamento. É importante também, que os protótipos não seja elementos ruidosos, pois isso aumentaria a taxa de erro do classificador.

Com os protótipos gerados é possível utilizar o *Nearest Prototype Classification*, NPC, que é utilizar protótipos gerados como treinamento do KNN. Assim a base de dados é reduzida, diminuindo o espaço de armazenamento e o tempo de processamento.

Além de possuir vantagens gerais como a diminuição do espaço de armazenamento e redução de esforço computacional para classificação, a seleção de protótipos pode ainda aumentar o desempenho no que se refere a taxa de acerto do classificador com a eliminação de ruídos e outliers, pois os protótipos aumentam a capacidade de generalização do classificador, levando a maiores taxas de acerto.

Algumas técnicas de seleção de protótipos selecionam instâncias que pertecem ao conjunto de treinamento, ou seja, elas escolhem, dentre as instâncias utilizadas, aquelas que julgam ser mais apropriadas para serem protótipos, estas técnicas são chamadas de técnicas puramente seletivas. Exemplos de técnicas seletivas são o *Edited Nearest Neighbor* [CPZ11], *Condensed Nearest Neighbor* [Har68], *Tomek Links* e *One-Sided Selection* [KM97].

Outra técnicas criam novos elementos durante o processo de redução, os protótipos são criados através de combinação entre as instâncias do conjunto de treinamento e ajustes relizados por meio de treinamento supervisionado. Entre estas técnicas estão o *Learning Vector Quantization 1, 2.1 e 3* [Koh88] e o *Self-Generating Prototypes* [FHA07].

Técnicas de seleção de protótipos também podem ser classificadas como determinísticas ou não deterministicas. Técnicas determinísticas são aquelas que, dada uma base de dados, sempre será gerado o mesmo conjunto de protótipos, independente da ordem em que o conjunto de treinamento é apresentado. Técnicas não determinísticas são aquelas que dependem em que o conjunto de treinamento é apresentado ou depende das instâncias pré-selecionadas para ajuste.

Cada uma das técnicas de seleção de protótipos aprensentam características próprias, sendo necessário uma análise do quanto cada uma destas técnicas é apropriada para uma dada base de dados. Algumas técnicas removem instâncias redundantes, outras, removem instâncias que estão na fronteira de classificação, e outras fazem uma combinação destas técnicas. Detalhes de algumas destas técnicas serão mostrados no próximo capítulo.

1.1.3 Bases Desbalanceadas

Em várias situações do mundo real, os classificadores precisam ser treinados com bases de dados que possue muito mais instâncias de uma de uma classe do que das outras classes, tais bases de dados são chamadas de bases desbalanceadas. Quanto maior a diferença entre a quantidade de instâncias de cada classe, maior o nível de desbalanceamento da base.

Quando treinados com bases de dados desbalanceadas, classificadores sofrem uma redução da performace, e normalmente tendem a classificar mais padrões com as classes marjoritárias. Este é um problema grave, visto que, normalmente, a classificação de instâncias da classe minoritária é que são mais importantes (como exemplo, informações sobre doenças) [HKN07].

Da mesma forma que classificadores podem ser prejudicados por um desbalanceamento, técnicas de seleção de protótipos podem sofrer da mesma forma, selecionando muitas instâncias da classe marjoritária e poucas, ou nenhuma, da classe minoritária.

1.2 Objetivo

O objetivo deste trabalho é expor algumas técnicas de seleção de protótipos e avaliar seu desempenho em bases desbalanceadas. A avaliação de desempenho se refere a taxa de acerto utilizando bases de dados reais, e a disposição dos protótipos por meio de bases artificiais de diferentes níveis de desbalanceamento e sobreposição de classes.

Para que o trabalho seja mais objetivo, apenas os exemplos mais interessante serão citados para cada técnica, citando as maiores vantagens e desvantagens de cada técnica.

No final do trabalho, será possível identificar quais técnicas são mais apropriadas para bases de dados desbalanceadas, e a identificação de suas falhas possibilitará possíveis melhoras nestas técnicas para o caso em questão.

1.3 Estrutura do Trabalho

O restante deste trabalho possui uma sessão com detalhes sobre diferentes técnicas de seleção de protótipos, nesta, já serão citadas as características já conhecidas e algumas adaptações conhecidas para tratar de bases desbalanceadas, além de possuir ilustrações e pseudo-código de cada uma das técnicas.

Logo após, segue uma sessão mostrado casos de sucesso e falhas de cada técnica em bases de dados artificiais, e depois uma sessão mostrado os resultados em bases de dados reais.

As análises levarão em conta a disposição e a quantidade dos protótipos resultantes de cada técnica, além disso, a taxa de acerto dos protótipos em relação ao próprio conjunto de treinamento para analisar a representatividade será mostrada. Por fim, cada técnica será executada em bases de dados reais, onde será utilizada *K-Fold Cross-Validation* para calcular a taxa de acerto média de cada técnica.

CAPÍTULO 2

Técnicas de Seleção de Protótipos

Neste capítulo, serão explicadas as técnicas de seleção de protótipos analisadas neste trabalho. Cada uma das sessões abaixo aborda conceitos básicos, pseudo-código e características de cada técnica, assim como possíveis adaptações nos seus algoritmos para tratar bases desbalanceadas.

2.1 ENN

Edited Nearest Neighbor Rule[CPZ11] é uma técnica de seleção de protótipos puramente seletiva proposta por Wilson em 1976. De uma forma geral, esta técnica foi projetada para funcionar como um filtro de ruídos, eliminando instâncias na região de fronteira, região de alta susceptibilidade a erros.

Por atuar apenas na região de fronteira, esta técnica possui uma baixa capacidade de redução. Seu algoritmo mantém as instâncias que não estão localizadas nesta região, exceto no caso de instâncias com extrema probabilidade de erro.

O algoritmo do ENN está demonstrado em Algorithm 2.

Algorithm 2 ENN

[h]

Require: *list*: uma lista

1. **for all** instância e_i da base de dados original **do**
2. Aplique o KNN sobre e_i
3. **if** e_i foi classificado erroneamente **then**
4. salve e_i em *list*
5. **end if**
6. **end for**
7. Remova da base de dados todos os elementos que estão em *list*

O valor de K usado pelo KNN pode variar de acordo com o tamanho da base de dados, porém, tipicamente, utiliza-se K=3. Em geral, O valor de K é inversamente proporcional a quantidade de instâncias que serão eliminadas, ou seja, para que o filtro elimine todos os possíveis ruídos, deve-se utilizar K=1, mas com isso, elimina-se também instâncias não ruidosas.

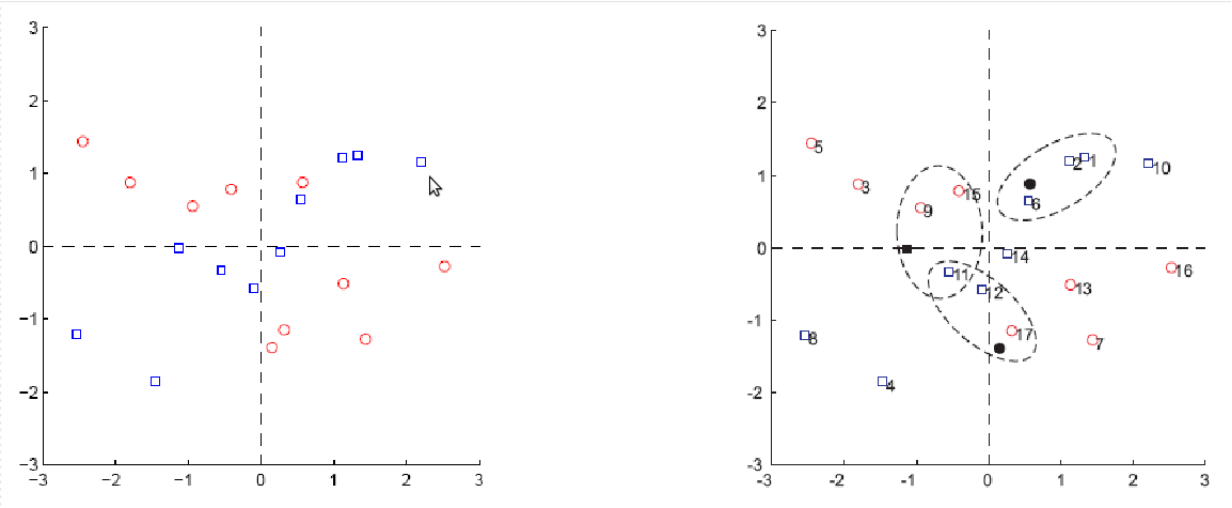

Figura 2.1 ENN aplicado com K=3

Na Figura 2.1 pode-se observar uma base de dados com duas classes. No primeiro gráfico da figura, observar-se a base de dados original, antes da aplicação do ENN. No segundo gráfico, foi aplicado o ENN, Algorithm 2, com K=3 sobre a base de dados. Os pontos pretos representam pontos que foram classificados erroneamente com a aplicação do KNN, a região circulada engloba os K elementos mais próximos do ruído. Mesmo um elemento que será posteriormente eliminado pode ser utilizado para eliminar ou manter outra instância, visto que as remoções são feitas apenas no fim da execução.

O mais interessante do caso acima é que, após a aplicação do ENN, as classes ficaram bem separadas pelos quadrantes pontilhados, mostrando a eficiência do ENN para a base de dados acima.

Uma vantagem do ENN é que ele independe da ordem que a base de dados foi apresentada, ou seja, o ENN aplicado a uma base de dados, com o mesmo valor de K, sempre terá o mesmo resultado.

Porém, o ENN também apresenta desvantagens, ele possui uma baixa capacidade de redução, pois elimina apenas ruídos, mantendo instâncias que são desnecessárias, que apresentam apenas redundância de informação. No caso da Figura 2.1, a base poderia ser representada por 4 instâncias bem posicionadas ou, por se tratar de uma técnica seletiva, com 8 instâncias, porém, o ENN manteve 13 instâncias, eliminando apenas 3.

Pelas suas características, normalmente o ENN é utilizado como método de pré-processamento da base de dados, eliminando apenas instâncias que apresentam alta probabilidade de serem ruídos.

No caso de bases desbalanceadas, o ENN pode tratar todas as instâncias da base minoritária como ruídos, caso a base seja altamente desbalanceada (isto será demonstrado posteriormente com exemplos), aumentando ainda mais o nível de desbalanceamento. Uma possível adaptação para o ENN em bases altamente desbalanceadas é eliminar apenas os elementos que sejam da classe marjoritária. O algoritmo adaptado está demonstrado em Algorithm 3.

Algorithm 3 ENN

[h][h]

Require: $list$: uma lista

1. **for all** instância e_i da base de dados original **do**
2. Aplique o KNN sobre e_i
3. **if** e_i foi classificado erroneamente **then**
4. **if** e_i for da classe marjoritária **then**
5. salve e_i em $list$
6. **end if**
7. **end if**
8. **end for**
9. Remova da base de dados todos os elementos que estão em $list$

Com este algoritmo adaptado, as instâncias da classe minoritária seriam mantidas, e a região delimitada por ela ficaria mais bem definida.

2.2 CNN

Condensed Nearest Neighbor [Har68] é uma técnica de seleção de protótipos puramente seletiva que tem como objetivo eliminar informação redundante. Diferentemente do ENN [CPZ11], o CNN não elimina instâncias nas regiões de fronteira, a técnica mantém estes elementos pois considera que estes "são os importantes"para distinguir entre duas classes.

A ideia geral do CNN é encontrar o menor subconjunto da base de dados original que, utilizando o 1-NN, classifica todos os padrões da base corretamente. Fazendo isso, o algoritmo elimina os elementos mais afastados da região de indecisão, da fronteira de classificação.

O algoritmo do CNN está descrito em Algorithm 4.

Algorithm 4 CNN

[h]

Require: $list$: uma lista

1. Escolha um elemento de cada classe $aleatoreamente$ e coloque-os em $list$
2. **for all** instância e_i da base de dados original **do**
3. $KNN(e_i, list)$
4. **if** e_i foi classificado erroneamente **then**
5. salve e_i em $list$
6. **end if**
7. **end for**
8. **return** $list$, os protótipos

Pode-se observar que este algoritmo possui uma abordagem totalmente diferente do ENN, pois ele começa com um conjunto mínimo de instâncias (uma de cada classe) e depois adiciona instâncias conforme a necessidade de mantê-las para que todos os elementos da base de dados original sejam classificados corretamente.

Um ponto importante na descrição do algoritmo, é a palavra *aleatoriamente*, o que significa que o CNN aplicado numa mesma base de dados com um mesmo valor de K para o KNN, nem sempre resulta nos mesmos protótipos. O primeiro fato para que isso ocorra é a seleção aleatória dos protótipos iniciais, a segunda é a ordem em que as instâncias são visitadas pelo algoritmo.

Existem algumas adaptações para o CNN, onde os protótipos iniciais são escolhidos utilizando técnicas como o SGP[FHA07] para obter as instâncias mais centrais. Modificações no CNN são muito comuns [Tom76], porém, mesmo com estas modificações, o CNN ainda não é determinístico, pois a ordem em que as instâncias são classificadas afeta o resultado final.

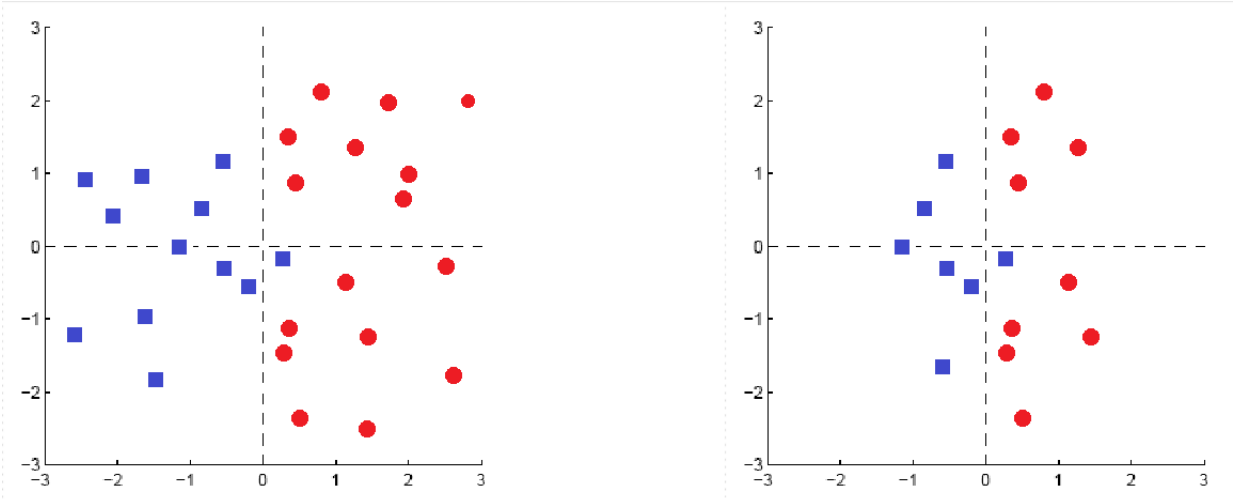

Figura 2.2 Exemplo da aplicação do CNN

No primeiro gráfico da figura 2.2 é mostrado uma base de dados qualquer, no segundo gráfico, é mostrado o resultado do CNN aplicado nesta base. Observa-se que uma grande quantidade de instâncias foi eliminada, mas praticamente todas as instâncias próximas da região de fronteira foram mantidas. Ainda existem instâncias redundantes nesta base, mas como citado anteriormente, isto acontece por conta da escolha aleatória dos protótipos iniciais e da ordem em que as instâncias são visitadas pelo algoritmo.

Para o caso de estudo abordado neste trabalho, o CNN pode ser utilizado de forma adaptada. A adaptação consiste em manter todos os elementos da classe minoritária e o mínimo possível da classe marjoritária. O próprio CNN se encarrega de remover os elementos redundantes da classe marjoritária, assim, basta apenas selecionar todos os elementos da classe minoritária aos protótipos iniciais. Segue em Algorithm 5, o algoritmo desta adaptação:

Algorithm 5 CNN para bases desbalanceadas

[h]

Require: *list*: uma lista

1. Coloque todos os elementos da classe minoritária em *list*
2. **for all** instância e_i da base de dados original **do**
3. Aplique o KNN sobre e_i utilizando os elementos em *list* para treinamento
4. **if** e_i foi classificado erroneamente **then**
5. salve e_i em *list*
6. **end if**
7. **end for**
8. **return** base original - *list*

Com o algoritmo CNN adaptado para bases desbalanceadas, os elementos redundantes da classe marjoritária são removidos, e todos os elementos da classe minoritária são mantidos. Com esta adaptação, além da redução do número de instâncias, também é reduzido o nível de desbalanceamento da base de dados.

2.3 Tomek Links

Mantendo a mesma linha do ENN, Tomek Links é uma técnica de seleção de protótipos puramente seletiva que elimina os elementos das regiões de fronteiras e instâncias com probabilidade de serem ruído. Tomek Links podem ser definidos da seguinte forma: Dadas duas instâncias e_i e e_j, o par (e_i, e_j) é chamado de Tomek Link se não existe nenhuma instância e_k, tal que, para todo e_k $dist(e_i,e_j) < dist(e_i,e_k)$ e $dist(e_i,e_j) < dist(e_j,e_k)$. Segue o algoritmo detalhado em Algorithm 6:

Algorithm 6 Seleciona Tomek Links

[h]

Require: *list*: uma lista

1. **for all** instância e_i da base de dados original **do**
2. e_j = instância mais próxima de e_i
3. **if** instância mais p¿oxima de e_j for e_i **then**
4. **if** classe de e_i for diferente da classe de e_j **then**
5. salve o par (e_i, e_j) em *list*
6. **end if**
7. **end if**
8. **end for**
9. **return** *list*, Tomek Links

Os Tomek Links representam elementos da região de indecisão e prováveis ruídos, e a técnica de seleção de protótipos consiste em remover os Tomek Links da base de dados original. O algoritmo Tomek Links é apresentado em 7:

Algorithm 7 Tomek Links

[h]

Require: *list*: uma lista

1. *list* = *SelecionaTomekLinks* da base original
2. **for all** (e_i, e_j) em *list* **do**
3. remova e_i da base original
4. remova e_j da base original
5. **end for**
6. **return** base original filtrada

Enquanto o CNN remove os elementos que estão longe da região de indecisão, o Tomek Links remove os elementos que estão próximos desta região, o que causa uma maior separação entre as classes.

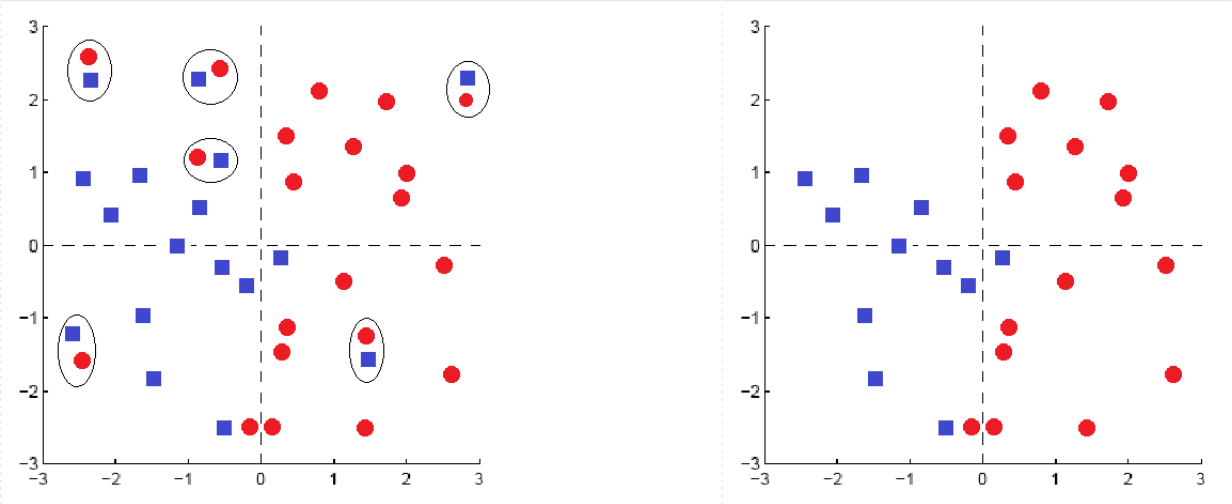

Figura 2.3 Exemplo da aplicação do Tomek Links

Na Figura 2.3 está exemplificada a aplicação do Tomek Links. No primeiro gráfico está a base original, e estão circulados os Tomek Links que, no segundo gráfico, foram removidos. No segundo gráfico, a base de dados foi filtrada e a maioria dos ruídos removidos. No exemplo desta figura, o Tomek Links fez uma separação entre as classes, eliminando parte de intersecção entre as mesmas.

Uma desvantagem do Tomek Links é que esta técnica elimina as duas instâncias presentes no Tomek Link, com isso, instâncias não ruidosas podem estar sendo removidas, instâncias estas que podem representar informação importante para a base de dados.

Observa-se facilmente que o Tomek Links pode remover todas as instâncias de regiões de indecisão, inclusive as instâncias da classe minoritária. Sendo assim, uma adaptação dos Tomek Links é eliminar apenas os elementos das classes marjoritárias. Nesse caso, ainda ocorre uma separação entre as classes, mas as instâncias dos Tomek Links que forem das classes mi-

noritárias são mantidas, diminuindo o nível de desbalanceamento. O algoritmo desta adaptação está demonstrado em Algorithm 8.

Algorithm 8 Tomek Links

[h]

Require: *list*: uma lista
1. *list* = *SelecionaTomekLinks* da base original
2. **for all** (e_i, e_j) em *list* **do**
3. **if** e_i for da classe marjoritária **then**
4. remova e_i da base original
5. **end if**
6. **if** e_j for da classe marjoritária **then**
7. remova e_j da base original
8. **end if**
9. **end for**
10. **return** base original - *list*

Com esta adaptação, a classe minoritária é mantida, evitando o aumento do desbalanceamento ou a remoção por alta probabilidade de ruído.

2.4 OSS

One-Sided Selection [KM97] é um método seletivo de seleção de protótipos, surgido pela combinação das técnicas CNN e Tomek Links. O algoritmo consiste na aplicação do CNN e depois da aplicação do Tomek Links como um filtro. O One-Sided Selection combina características das duas técnicas. A aplicação do CNN é feita para eliminar instâncias desnecessárias, redundantes, ou seja, instâncias que estão longe da fronteira de classificação. Já a aplicação do Tomek Links tem a função de remover elementos na fronteira de classificação, fazendo uma aparente separação das classes e removendo ruídos.

O OSS é muito utilizado para bases desbalanceadas, utilizando a adaptação do CNN, como monstrado no algoritmo 9.

Algorithm 9 One-Sided Selection

[h]

Require: *list*: uma lista
Require: $tomek_links_list$: uma lista
1. *list* = *CNNparabasesdesbalanceadas* sobre a base original
2. *list* = *TomekLinksparabasesdesbalanceadas* sobre *list*
3. **return** *list*

Observando o algoritmo, é fácil concluir que o One-Sided Selection é uma técnica apropriada para bases desbalanceadas. A aplicação do CNN adaptado elimina as instâncias redundantes da base marjoritária, colaborando para, além de diminuir a quantidade de instâncias

longe da fronteira de classificação, diminuir o nível de desbalanceamento entre as classes. Já a aplicação do Tomek Links adaptado, elimina instâncias da classe marjoritária na fronteira de classificação, colaborando para maior delimitação da classe minoritária.

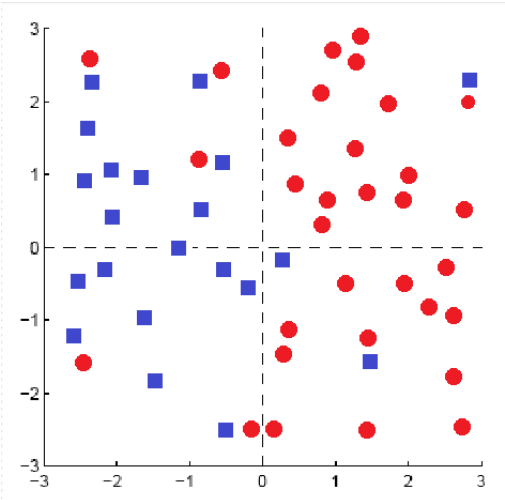

Figura 2.4 Base Original antes da aplicação do OSS

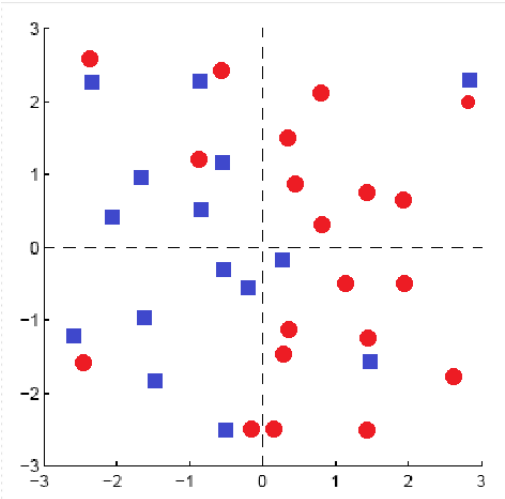

Figura 2.5 Aplicação da primeira etapa do OSS, o CNN

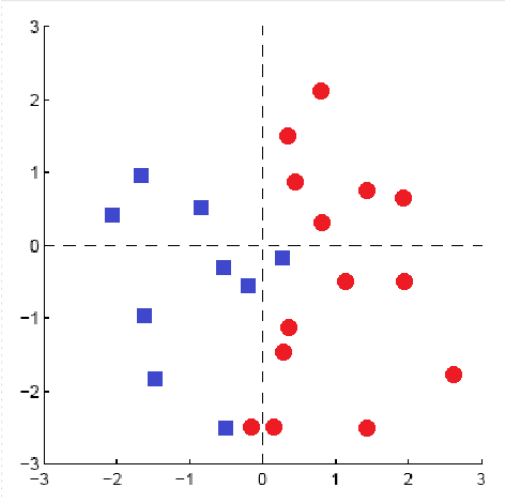

Figura 2.6 Resultado final do OSS, após aplicação da segunda etapa, o Tomek Links

Nas Figuras 2.4, 2.4 e 2.4 .

Uma desvantagem do One-Sided Selection é que ele não é determinístico, o CNN é não-determinístico e como o One-Sided Selection faz a aplicação dele, torna o mesmo não-determinístico. O OSS poderia ser feito aplicando-se outro algoritmo no lugar do CNN, podendo assim, torna-lo determinístico. Este trabalho, porém, não abordará adaptações para o OSS, pois o mesmo já é apropriado para base de dados, e ser ou não determinístico, apesar de ser levado em consideração, não faz parte do escopo deste trabalho.

2.5 LVQ

Learning Vector Quantization proposto por Kohonen [Koh88]. O Learning Vector quantization é um algoritmo supervisionado de síntese de protótipos, ou seja, cria novas instâncias baseadas em instâncias já existentes. A ideia básica do algoritmo é que dado um conjunto inicial de protótipos, o LVQ faz um ajuste dos protótipos, de forma a posicionar cada instância em um ponto que seja possível estabelecer uma função discriminante baseada nestes protótipos.

Uma desvantagem do LVQ é que a ordem das instâncias altera o resultado, ou seja, o algoritmo não é deterministico. Outra desvantagem é que, conforme será mostrado, o LVQ possui vários parâmetros, sendo necessário uma análise empírica dos valores apropriados para esses parâmetros.

Os protótipos iniciais podem ser escolhidos de qualquer forma, a idéia é que sejam protótipos que tenham boa representatividade da base de dados, mas também podem ser selecionados aleatoriamente, pois o próprio LVQ se encarrega de fazer os ajustes nestes protótipos.

2.5.1 LVQ 1

LVQ1 é a primeira versão do Learning Vector Quantization proposto por Kohonen. O algoritmo do LVQ1 basicamente seleciona alguns protótipos iniciais e ajusta esses protótipos utilizando

a base original. Quando uma instância da base original é classificada erroneamente pelos protótipos, afasta-se o protótipo mais próximo, e quando é classificada corretamente, aproxima-se. O algoritmo detalhado pode ser visto em 10.

Algorithm 10 LVQ 1

[h]

Require: *prototypes*: uma lista para os protótipos
Require: *selection*: um algorítmo para seleção dos protótipos iniciais
1. *prototypes* = *selection* (base original)
2. **while** *prototypes* não estiver sub-ajustado **do**
3. x = *ChooseOne* (base original)
4. e_i = *SelectNearestFrom*(*prototypes*, x)
5. **if** classe de $e_i \neq$ classe de x **then**
6. $e_i = e_i + \alpha(t) \times [x - e_i]$
7. **else**
8. $e_i = e_i - \alpha(t) \times [x - e_i]$
9. **end if**
10. **end while**
11. **return** *prototypes*

A vantagem do LVQ1 é que ele estabiliza durante o treinamento, porém, ele possui um grande número de passos. Para a maioria dos problemas, o LVQ 1 possui um resultado satisfatório, mas além da demora, é necessário escolher os parâmetros corretamente.

Um dos parâmetros é o $\alpha(t)$, uma constante de ajuste, que serve para aproximar ou afastar os protótipos. Este afastamento ou aproximação é regulado pelo valor de $\alpha(t)$, sendo $0 < \alpha(t) < 1$. Percebe-se que $\alpha(t)$ foi colocado como uma função. Normalmente, essa função é uma exponencial descrescente, e o algoritmo termina quando $\alpha(t)$ se torna insignificante.

Outra questão do LVQ1 é escolher a quantidade de protótipos iniciais adequada, visto que, esta quantidade não é alterada durante toda a execução do algoritmo.

No caso de bases desbalanceadas, pode-se utilizar fatores de ajustes diferenciados para cada classe, ou escolher uma quantidade aproximada de cada classe para os protótipos iniciais. Fazendo estas adaptações, o LVQ1 poderá ter resultados melhores para bases desbalanceadas.

2.5.2 Optimized-learning-rate LVQ

Optimized-learning-rate LVQ [colocar referencia aqui http://www.springerlink.com/content/n72865x1t57q187 é uma versão otimizada do LVQ1, proposto para aumentar a velocidade de convergência do LVQ1. O modelo consiste basicamente em cada protótipo ter taxas de aprendizado individuais, a dinâmica da taxa de aprendizado consiste no aumento da mesma caso o protótipo esteja classificando corretamente e na diminuição, caso contrário.

$$\alpha(t) = \frac{\alpha(t-1)}{1 + s(t) \times \alpha(t-1)}$$

$$s(t) = +1, \text{ se } x \text{ é classificado corretamente}$$
$$s(t) = -1, \text{ se } x \text{ é classificado erroneamente}$$
$$0 < \alpha(t) < 1$$

Com esta alteração do valor de $\alpha(t)$ faz com que o LVQ convirja mais rapidamente, tornando o algoritmo OLVQ mais viável em termos de performace e mantendo as características do LVQ1.

2.5.3 LVQ 2.1

Kohonen propos duas novas versões melhoradas do LVQ, uma delas é o LVQ 2.1. Esta nova versão do LVQ faz atualização nos dois protótipos mais próximos desde que as condições de ajuste sejam atendidas.

A ideia do LVQ 2.1 é ajustar apenas os protótipos próximos das fronteiras de classificação, região de indecisão. Para evitar uma divergência entre estes protótipos, foi introduzida a Regra da Janela.

Diz-se que um elemento está na janela quando ele obedece a regra da janela, isso acontece quando um elemento está na região de indecisão.

Dado um elemento x, diz-se que ele está na janela se:

$$\min \frac{d_i}{d_j}\frac{d_j}{d_i} > s, \text{ onde } s = \frac{1-w}{1+w}$$
$$e_i \text{ e } e_j \text{ são os protótipos mais próximos de } x$$
$$d_i \text{ é a distância de } x \text{ para } e_i$$
$$d_j \text{ é a distância de } x \text{ para } e_j$$
$$w \text{ é a largura relativa}$$

Algorithm 11 LVQ 2.1

[h]

Require: *prototypes*: uma lista para os protótipos

1. *prototypes* = $LVQ1$ (base original)
2. **while** *prototypes* não estiver sub-ajustado **do**
3. x = *ChooseOne* (base original)
4. e_i, e_j = $SelectNearestsFrom(prototypes, x)$
5. **if** $CaiuNaJanela(x, e_i, e_j)$ **then**
6. **if** $Classe(e_i) \neq Classe(e_j)$ **then**
7. **if** $Classe(e_i) = Classe(x)$ **then**
8. $e_i = e_i + \alpha(t) \times [x - e_i]$
9. $e_j = e_j - \alpha(t) \times [x - e_i]$
10. **else**
11. $e_i = e_i - \alpha(t) \times [x - e_i]$
12. $e_j = e_j + \alpha(t) \times [x - e_i]$
13. **end if**
14. **end if**
15. **end if**
16. **end while**
17. **return** *prototypes*

A algoritmo de LVQ 2.1 é aplicado depois do LVQ 1, mas ele ajusta dois protótipos a cada iteração. Esta técnica não faz ajustes de protótipos se x não estiver na janela, se nenhum dos protótipos forem da classe de x ou se os protótipos forem da mesma classe. Pode-se ver o algoritmo detalhado em 11.

Enquanto o LVQ1 provoca o afastamento dos protótipos nas regiões de indecisão, o LVQ 2.1 reduz esse afastamento atuando apenas sobre protótipos vizinhos pertencentes a classes diferentes.

Uma desvantagem do LVQ 2.1 é que além do custo ser maior, a aplicação do mesmo pode sobre-ajustar os protótipos nas regiões de indecisão. Para diminuir esse sobre-ajuste, foi criado o LVQ 3, abordado na próxima sessão.

2.5.4 LVQ 3

A segunda melhora proposta por Kohonen foi o LVQ 3. Este método tenta evitar o sobre-ajuste do LVQ 2.1 atuando também quando o elemento já está sendo classificado corretamente pelos dois protótipos mais próximos, aproximando ambos da instância utilizada para ajuste. Além disso, a terceira versão do LVQ introduz um fator de estabização ε.

Algorithm 12 LVQ 3

[h]

Require: *prototypes*: uma lista para os protótipos

1. *prototypes* = $LVQ1$ (base original)
2. **while** *prototypes* não estiver sub-ajustado **do**
3. x = *ChooseOne* (base original)
4. e_i, e_j = $SelectNearestsFrom(prototypes, x)$
5. **if** $CaiuNaJanela(x, e_i, e_j)$ **then**
6. **if** $Classe(e_i) \neq Classe(e_j)$ **then**
7. **if** $Classe(e_i) = Classe(x)$ **then**
8. $e_i = e_i + \alpha(t) \times [x - e_i]$
9. $e_j = e_j - \alpha(t) \times [x - e_i]$
10. **else**
11. $e_i = e_i - \alpha(t) \times [x - e_i]$
12. $e_j = e_j + \alpha(t) \times [x - e_i]$
13. **end if**
14. **else if** $Classe(e_i) = Classe(e_j) = Classe(x)$ **then**
15. $e_i = e_i + \varepsilon \times \alpha(t) \times [x - e_i]$
16. $e_j = e_j + \varepsilon \times \alpha(t) \times [x - e_i]$
17. **end if**
18. **end if**
19. **end while**
20. **return** *prototypes*

O fator de estabilização serve para suavizar o ajuste quando os protótipos já estão classificando corretamente x. O valor de ε deve ser tal que $0 < \varepsilon < 1$.

Assim como as outras versões do LVQ, o LVQ 3 é robusto, mas seu maior problema é determinar o valor de tantos parâmetros como α, ε e w. Porém, a maior desvantagem é não ser possível saber com certeza quando os protótipos foram ou não sobre-ajustados.

2.6 SGP

Self-Generating Prototypes [FHA07] é uma técnica de síntese de protótipos muito completa. Sua maior vantagem é que, enquanto muitas técnicas de seleção de protótipos dependem da escolha correta do número de protótipos iniciais ou possuem muitos parâmetros, o SGP encontra a quantidade de protótipos e a localização de cada uma em sua fase de treinamento, sem supervisão humana.

A ideia principal do SGP é formar um certo número de grupos e eleger representantes para esses grupos. Conforme necessário, o algorítmo divide os grupos ou move instâncias de um grupo para outro.

Inicialmente, para cada classe, é criado um grupo contendo todas as instâncias daquela classe. Com os grupos feitos, são obtidos os centróides de cada grupo e estes são chamados representantes do grupo. Depois, para cada grupo, faça os passos abaixo até que não haja mais

alterações em nenhum grupo.

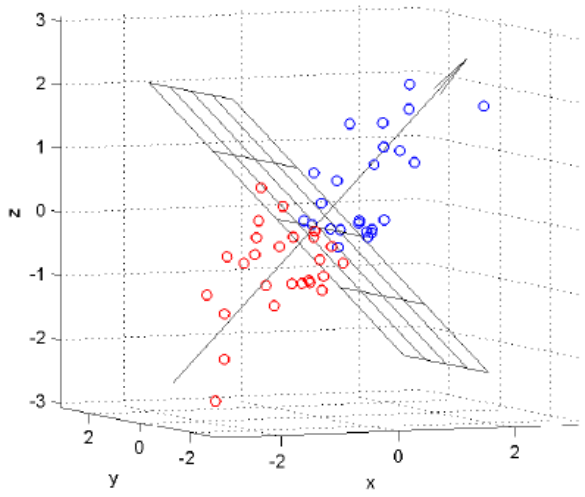

Figura 2.7 Etapa da divisão de grupos. Figura obtida de [dSPC08]

- Se para todos os padrões de um grupo o protótipo mais próximo é o centróide do grupo, então nenhuma operação é realizada.
- Se para todos os padrões de um grupo o protótipo mais próximo é de uma classe diferente da do grupo, ele é dividido em dois subgrupos 2.6. Essa divisão é feita separando os padrões pelo hiperplano que passa pelo centroide do grupo, e cujo vetor normal é a primeira componente principal gerada pelos padrões do grupo.
- Se para alguns padrões de um grupo o protótipo mais próximo é diferente do centróide, mas da mesma classe, esses padrões são deslocados do grupo original para o grupo do protótipo mais próximo
- Se para alguns padrões de um grupo o protótipo mais próximo não é o centróide e é de uma classe diferente, estes padrões são removidos do grupo original e formam um novo grupo, sendo o centróide computado como um novo protótipo.

No final de cada iteração, o centróide de cada grupo é computado novamente. O processo se repete até que não haja alterações em mais nenhum grupo. Uma descrição mais formal do algoritmo é descrita em 13

Algorithm 13 SGP 1

[h]

Require: PS: uma lista para os representantes
Require: GS: uma lista de grupos de instâncias
Require: T: uma lista de duplas de instâncias

for all classe C **do**
  $G = \bigcup$ instâncias da classe C
  $Adicione(G, GS)$
  $Adicione(Centroide(G), PS)$
end for
$count = 1$
while $count \neq 0$ **do**
  $count$ = $Quantidade(GS)$
  $Limpe(T)$
  for all G em GS **do**
    P = Representante de G
    for all e_i em $group$ **do**
      $NearestP$ = $1NN(e_i, PS)$
      $Adicione((e_i, NearestP), T)$
    end for
    if $\forall\ e_i, NearestP$ em T, $NearestPS$ = P **then**
      $count$ = $count - 1$
    else if $\forall\ e_i, NearestP$ em T, a $Classe(NearestP) \neq Classe(P)$ **then**
      $Vector = PrimeiraComponentePrincipal(G)$
      $Hiperplano$ = hiperplano que passa pelo centróide de G e cujo vetor normal é a $Vector$.
      Divida G em 2 grupos, instâncias acima e abaixo de $Hiperplano$
      Atualize GS e PS.
    else if $\exists\ e_i, NearestP$ em T tal que $NearestP \neq P$ e $Classe(NearestP) = Classe(P)$ **then**
      Remova e_i de G e adicione a grupo de $NearestP$.
      Atualize GPS e PS.
    else if $\exists\ e_i, NearestP$ em T tal que o $Classe(NearestP)! = Classe(P)$ **then**
      Remova e_i de G.
      Crie um novo grupo contendo as instâncias removidas.
      Atualize PS e GS,
    end if
  end for
end while
return PS

Observando o algoritmo do SGP1 13, percebe-se que apesar do conceito ser bem simples, esta técnica possui alguns passos complexos, sendo necessário conhecimentos sobre extração de características [REFERÊNCIA PCA AQUI]. Principal Component Analysis é a técnica uti-

lizada para traçar o vetor perpendicular ao hiperplano 2.6.

A maior vantagem do SGP1 é que, conforme citado anteriormente, ele é não depende de parâmetros como quantidade de protótipos iniciais nem valores específicos. Por ser um algoritmo deterministico, o SGP1 executado numa mesma base de dados, sempre gerará os mesmos protótipos. Estes protótipos gerados possuem excelente representatividade do conjunto de treinamento, tanto que, utilizando-se os protótipos gerados como treinamento de um KNN, e classificando-se todas as instâncias de treinamento do SGP1, a taxa de acerto é de 100%. Claro que o treinamento não pode ser utilizado para teste, mas isto mostra a boa representatividade das instâncias geradas pelo SGP1.

Porém, o SGP1 também apresenta desvantagens. Uma delas é que o SGP1 é muito custoso, exigindo, em geral, um treinamento mais longo que outras técnicas.

Outra desvantagem é que o SGP1 é sensível a ruídos, pois, se necessário, o algoritmo criará um grupo com apenas uma instância. No caso de um ruído, isto será muito desvantajoso, considerando que, em experimentos diversos, o SGP1 conseguiu reduzir em mais de 100 vezes o tamanho da base de dados.

Apesar das desvantagens citadas, bases desbalanceadas podem ser beneficiadas com o SGP1, considerando que ele considera os agrupamentos de classes, e não a quantidade de instâncias em cada classe.

2.7 SGP 2

O Self-Generating Prototypes 2 [FHA07]é uma versão melhorada do SGP1 que reduz ainda mais a quantidade de protótipos e é menos sensível aos ruídos e outliers. Para fazer essas melhorias, o SGP2 possui uma etapa de *merge* e *prunning*.

Quando dois grupos A e B são da mesma classe e para todas as instâncias de A o segundo protótipo mais próximo é o representante de B, e para todas as instâncias de B o segundo protótipo mais próximo é o representante de A, os grupos A e B podem ser fundidos, e será computado um novo protótipo para este novo grupo maior. Esta etapa é chamada de *merge*, e ela é responsável por reduzir ainda mais a quantidade de protótipos do SGP1. O *prunnig* consiste em remover grupos onde o segundo protótipo mais próximo de todos os padrões de um certo grupo possuem a mesma classe, neste caso, tanto as instâncias quanto o representante do grupo são eliminados.

Para evitar o sobre-ajuste e perda de generalização introduziu-se dois parâmetros ao SGP2 chamados de R_n e R_s. O R_n representa um limite para o tamanho relativo de um grupo em relação ao maior grupo. Por exemplo, se R_n é 0.1 e o tamanho do maior grupo é 200, todos os grupos com tamanho menor que $20(0.1 \times 200)$ são descartados. O segundo parâmetro R_s é um limiar para a taxa de classificações incorretas de um grupo. Se o número de padrões classificados erroneamentes em um grupo dividido pelo tamanho do grupo é menor que R_s, então o grupo permanece inalterado. Este parâmetros reduzem o número de protótipos e aumentam a capacidade de generalização. Pereira e Cavalcanti recomendam $0.01 < R_n < 0.2$ e $0.01 < R_s < 0.2$ [dSPC08].

Algorithm 14 SGP 2

[h]

Require: PS: uma lista para os representantes
Require: GS: uma lista de grupos de instâncias
Require: T: uma lista de duplas de instâncias

1. **for all** classe C **do**
2. $G = \bigcup$ instâncias da classe C
3. $Adicione(G, GS)$
4. $Adicione(Centroide(G), PS)$
5. **end for**
6. $count = 1$
7. **while** $count \neq 0$ **do**
8. $count$ = $Quantidade(GS)$
9. $Limpe(T)$
10. **for all** G em GS **do**
11. P = Representante de G
12. **for all** e_i em $group$ **do**
13. $NearestP = 1NN(e_i, PS)$
14. $Adicione((e_i, NearestP), T)$
15. **end for**
16. **if** Quantidade de tuplas em T onde $NearestP \neq P \leq R_s$ **then**
17. $count$ = $count - 1$
18. **else if** $\forall\ e_i, NearestP$ em T, a $Classe(NearestP) \neq Classe(P)$ **then**
19. $Vector = PrimeiraComponentePrincipal(G)$
20. $Hiperplano$ = hiperplano que passa pelo centróide de G e cujo vetor normal é a $Vector$.
21. Divida G em 2 grupos, instâncias acima e abaixo de $Hiperplano$
22. Atualize GS e PS.
23. **else if** $\exists\ e_i, NearestP$ em T tal que $NearestP \neq P$ e $Classe(NearestP) = Classe(P)$ **then**
24. Remova e_i de G e adicione a grupo de $NearestP$.
25. Atualize GPS e PS.
26. **else if** $\exists\ e_i, NearestP$ em T tal que o $Classe(NearestP)! = Classe(P)$ **then**
27. Remova e_i de G.
28. Crie um novo grupo contendo as instâncias removidas.
29. Atualize PS e GS,
30. **end if**
31. **end for**
32. **end while**
33. $LargeGroupCount$ = Quantidade de instâncias do maior grupo em GS
34. **for all** G, P in GS, PS **do**
35. $CurrentGroupCount$ = Quantidade de instâncias de G.
36. **if** $CurrentGroupCount/LargeGroupCount \leq R_n$ **then**
37. Remova G de GS e P de PS.
38. **end if**
39. **end for**
40. **return** PS

O algoritmo SGP2 14 é mais custoso e demorado que o SGP1, porém, em experimentos práticos, o SGP2 se mostrou mais eficiente, diminuindo a quantidade de protótipos ainda mais que o SGP1 e ainda assim, aumentando a taxa de acerto.

Porém, um defeito do SGP2 é que ele não leva em conta bases desbalanceadas. Dependendo da distribuição das instâncias, pode acontecer de, na fase de *prunning* o SGP2 remover o representante da classe minoritária, podendo até remover todos os protótipos desta classe. Isto é perceptível, considerando o valor de $R_n = 0.15$, bases onde a classe minoritária possui apenas 3% das instâncias da base, mesmo com a base marjoritária multimodal, provavelmente, após o *prunning*, todos os protótipos serão da classe marjoritária.

Referências Bibliográficas

[CPZ11] Ruiqin Chang, Zheng Pei, and Chao Zhang. A modified editing k-nearest neighbor rule. *JCP*, 6(7):1493–1500, 2011.

[dSPC08] Cristiano de Santana Pereira and George D. C. Cavalcanti. Prototype selection: Combining self-generating prototypes and gaussian mixtures for pattern classification. In *IJCNN*, pages 3505–3510. IEEE, 2008.

[EJJ04] Andrew Estabrooks, Taeho Jo, and Nathalie Japkowicz. A multiple resampling method for learning from imbalanced data sets. *Computational Intelligence*, 20(1):18–36, 2004.

[FdJH09] Alberto Fernández, María José del Jesús, and Francisco Herrera. Hierarchical fuzzy rule based classification systems with genetic rule selection for imbalanced datasets. *Int. J. Approx. Reasoning*, 50(3):561–577, 2009.

[FHA07] Hatem A. Fayed, Sherif Hashem, and Amir F. Atiya. Self-generating prototypes for pattern classification. *Pattern Recognition*, 40(5):1498–1509, 2007.

[Har68] P. E. Hart. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14:515–516, 1968.

[HKN07] Jason Van Hulse, Taghi M. Khoshgoftaar, and Amri Napolitano. Experimental perspectives on learning from imbalanced data. In Zoubin Ghahramani, editor, *ICML*, volume 227 of *ACM International Conference Proceeding Series*, pages 935–942. ACM, 2007.

[KM97] Miroslav Kubat and Stan Matwin. Addressing the curse of imbalanced training sets: One-sided selection. In Douglas H. Fisher, editor, *ICML*, pages 179–186. Morgan Kaufmann, 1997.

[Koh86] Teuvo Kohonen. Learning vector quantization for pattern recognition. Report TKK-F-A601, Laboratory of Computer and Information Science, Department of Technical Physics, Helsinki University of Technology, Helsinki, Finland, 1986.

[Koh88] Teuvo Kohonen. Learning vector quantization. *Neural Networks*, 1, Supplement 1:3–16, 1988.

[PI69] E. A. Patrick and F. P. Fischer II. A generalization of the k-nearest neighbor rule. In *IJCAI*, pages 63–64, 1969.

[Sav] Sergei Savchenko. http://cgm.cs.mcgill.ca/ .

[TC67] P.E. Hart T.M. Cover. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, IT-13:21 – 27, 1967.

[Tom76] I. Tomek. Two Modifications of CNN. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(2):679–772, 1976.

[WP01] G. Weiss and F. Provost. The effect of class distribution on classifier learning, 2001.