Pontifícia Universidade Católica
do Rio de Janeiro

Daniel da Rosa Marques

Um Metaclassificador para Encontrar as K-Classes mais Relevantes

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio.

Orientador: Prof. Eduardo Sany Laber

Rio de Janeiro
Novembro de 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Daniel da Rosa Marques

Um Metaclassificador para Encontrar as K-Classes mais Relevantes

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Eduardo Sany Laber
Orientador
Departamento de Informática – PUC-Rio

Prof. Ruy Luiz Milidiú
Departamento de Informática – PUC-Rio

Prof. Raul Pierre Renteria
Departamento de Informática – PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 24 de Novembro de 2015

Daniel da Rosa Marques

Graduou-se em Engenharia Eletrônica e da Computação na Universidade Federal do Rio de Janeiro. Trabalhou com desenvolvimento de soluções de informática em diversas empresas.

Ficha Catalográfica

Marques, Daniel da Rosa

Um Metaclassificador para Encontrar as K-Classes mais Relevantes / Daniel da Rosa Marques; orientador: Eduardo Sany Laber. – Rio de Janeiro : PUC-Rio, Departamento de Informática, 2015.

v., 56 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Aprendizado de Máquina. 3. Ranking. I. Laber, Eduardo Sany. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 620.11

Agradecimentos

Ao meu orientador Professor Eduardo Sany Laber pelo apoio e orientação no desenvolvimento deste trabalho.

À CAPES e à PUC–Rio, pelos auxílios concedidos, os quais possibilitaram a realização deste trabalho.

Ao Félix Ribeiro, minha maior fonte de inspiração e coragem, sem ele nada disso seria possível.

A minha mãe, irmã e família, sempre presentes e na torcida.

Aos funcionários do departamento de Informática pela a ajuda de costume.

Aos meus colegas da PUC–Rio, que também me ajudaram nessa jornada.

Resumo

Marques, Daniel da Rosa; Laber, Eduardo Sany. **Um Metaclassificador para Encontrar as K-Classes mais Relevantes**. Rio de Janeiro, 2015. 56p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Considere uma rede com k nodos que pode apresentar falhas ao longo de sua operação. Além disso, assuma que é inviável verificar todos os nodos sempre que uma falha ocorre. Motivados por este cenário, propomos um método que usa aprendizado de máquina supervisionado para gerar *rankings* dos nodos mais prováveis por serem responsáveis pela falha. O método proposto é um metaclassificador que pode utilizar qualquer tipo de classificador internamente, onde o modelo gerado pelo metaclassificador é uma composição daqueles gerados pelos classificadores internos. Cada modelo interno é treinado com um subconjunto dos dados. Estes subconjuntos são criados sucessivamente a partir dos dados originais eliminando-se algumas instâncias. As instâncias eliminadas são aquelas cujas classes já foram colocadas no ranking. Métricas derivadas da *Acurácia*, *Precision* e *Recall* foram propostas e usadas para avaliar este método. Utilizando uma base de domínio público, verificamos que os tempos de treinamento e classificação do metaclassificador são maiores que os de um classificador simples. Entretanto ele atinge resultados melhores em alguns casos, como ocorre com as árvores de decisão, que superam a acurácia do *benchmark* por uma margem maior que 5%.

Palavras-chave

Aprendizado de Máquina; Ranking.

Abstract

Marques, Daniel da Rosa; Laber, Eduardo Sany (Advisor). **A Metaclassifier for Finding the K-Classes most Relevants**. Rio de Janeiro, 2015. 56p. MsC Thesis – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Consider a network with k nodes that may fail along its operation. Furthermore assume that it is impossible to check all nodes whenever a failure occurs. Motivated by this scenario, we propose a method that uses supervised learning to generate *rankings* of the most likely nodes responsible for the failure. The proposed method is a meta-classifier that is able to use any kind of classifier internally, where the model generated by the meta-classifier is a composition of those generated by the internal classifiers. Each internal model is trained with a subset of the data created from the elimination of instances whose classes were already put in the ranking. Metrics derived from *Accuracy*, *Precision* and *Recall* were proposed and used to evaluate this method. Using a public data set, we verified that the training and classification times of the meta-classifier were greater than those of a simple classifier. However it reaches better results in some cases, as with the decision trees, that exceeds the benchmark accuracy for a margin greater than 5%.

Keywords

Machine Learning; Ranking.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Existem diversas tarefas importantes que o homem costuma desempenhar melhor do que a máquina. Entretanto, o aprendizado de máquina tem sido utilizado com sucesso para resolver algumas destas. Especialmente aquelas que exigem o processamento de uma grande quantidade de dados, o que seria inviável realizar manualmente. Exemplos destas tarefas são reconhecimento de imagens, tradução automática de idiomas, reconhecimento de fala, movimentação de autômatos, sistemas de recomendação, entre outras.

As soluções para algumas dessas tarefas utilizam conceitos de *ranking*. Por exemplo, a Figura 1.1 mostra o resultado de uma busca por páginas na *web*. Mecanismos de busca como esse são parte do cotidiano hoje em dia, eles utilizam *ranking* para criar listas de páginas relevantes para uma busca.

Motivado por isso, este trabalho se concentra em tarefas de *ranking*. Durante o projeto, criamos um metaclassificador para aprendizado de máquina supervisionado adequado para este tipo de tarefa. Este metaclassificador é capaz de utilizar internamente qualquer tipo de classificador encontrado na literatura.

1.1
Descrição do Problema

Para entender o problema de *ranking* no qual este trabalho se concentra, considere um exemplo onde temos uma rede R com n nodos. Imagine que esta rede eventualmente pode apresentar falhas em seus nodos, sendo que cada falha pode causar sérios danos à operação da rede. Para corrigi-las é necessário fazê-lo diretamente no nodo afetado pela falha. Isto por que não é possível estabelecer comunicação direta com ele, por que o mesmo está submerso por exemplo. Além disso, para corrigir uma falha rapidamente, é necessário determinar logo onde ela ocorreu. Para tanto poderíamos verificar simultaneamente todos os n nodos, por exemplo enviando equipes até os locais. Entretanto o custo do deslocamento até um dos nodos é muito elevado, sendo inviável checar toda a rede sempre que ocorre uma falha.

No âmbito do aprendizado de máquina tradicional os nodos da rede podem ser traduzidos em classes. Além disso, temos diversas estatísticas e

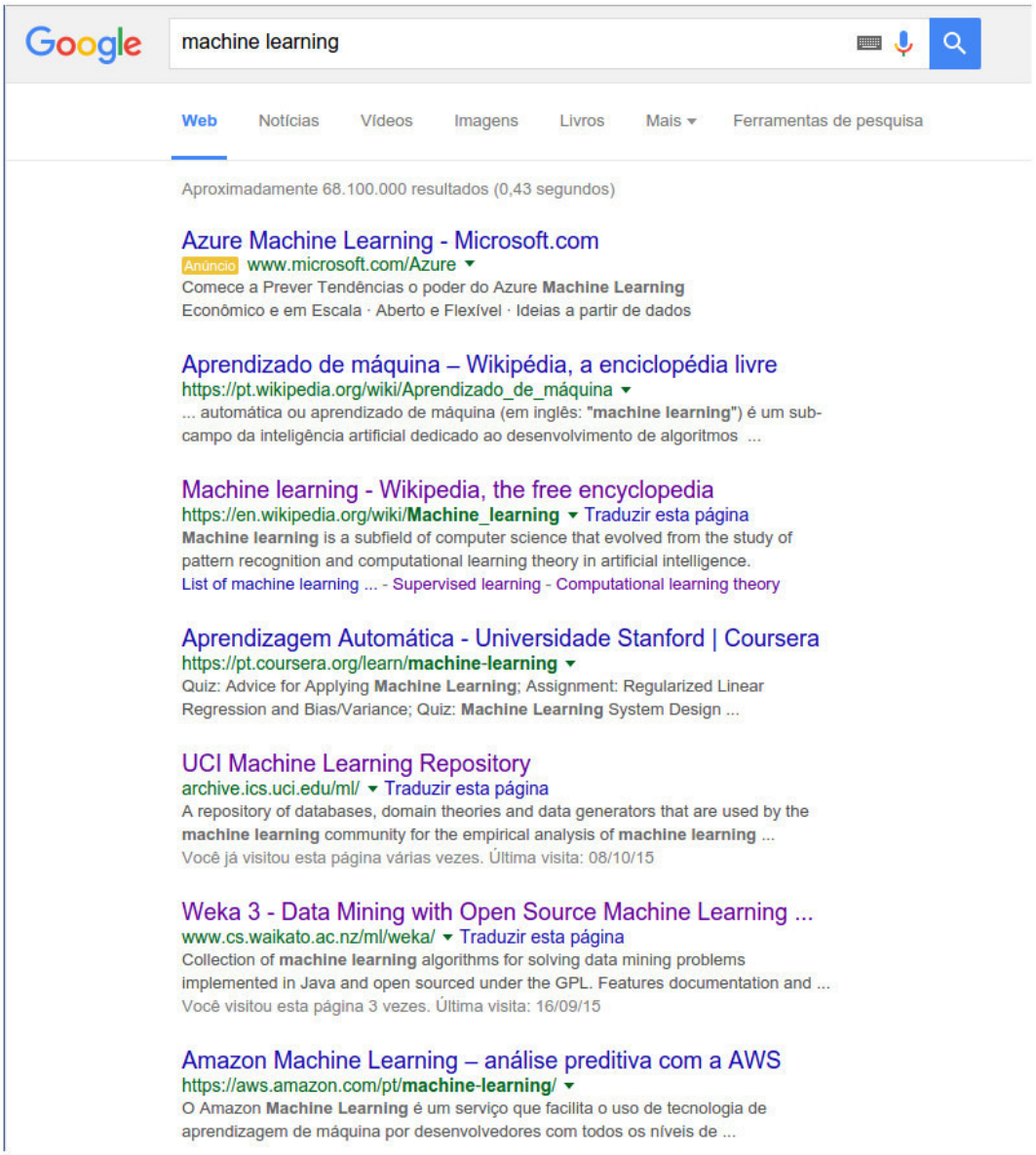

Figura 1.1: Exemplo de aplicação de ranking

medições sobre o estado da rede R ao longo do tempo. Estas podem ser utilizadas como atributos. Gostaríamos então de identificar automaticamente o nodo onde uma falha ocorre, provavelmente usando um classificador.

Tendo em vista a dificuldade de gerar essa informação com absoluta precisão, uma solução razoável é construir uma lista dos nodos mais prováveis onde a falha ocorreu. Quanto mais próximo da primeira posição da lista estiver o nodo verdadeiro, menor será o custo de reparo da rede. Embora o metaclassificador desenvolvido possa ser utilizado para tarefas de *ranking* de propósito geral, o problema definido acima será usado ao longo do trabalho como tarefa principal.

1.2 Contribuições

Neste trabalho propomos um método para geração de *rankings* capaz de utilizar qualquer tipo de classificador para esta tarefa. De fato, como já citamos, da forma que foi implementado ele funciona como um metaclassificador.

O método elimina ruído sucessivamente do conjunto de treino à medida que a lista de saída é construída. Isso foi feito por meio de filtragens aplicadas aos dados, que removem instâncias de classes já incluídas na lista. Desta forma, os classificadores internos do metaclassificador são treinados com conjuntos de treino filtrados de formas diferentes. Isto é, cada classificador gera um modelo único do qual algumas classes foram retiradas. Cada modelo é utilizado então para gerar um único elemento da lista de saída. Veremos que para classificar diversas instâncias, com listas de saída diferentes, uma grande quantidade de classificadores precisa ser gerada. Por exemplo, se o conjunto de dados tem sete classes diferentes o metaclassificador pode vir a ter mais de cem classificadores internos.

Além disso, o método foi desenvolvido em duas versões: estática e dinâmica. As duas versões diferem no momento no qual os classificadores internos são treinados.

A primeira versão treina todos os classificadores internos possíveis a priori. Isto é, gera todos os subconjuntos possíveis dos dados originais e treina um classificador para cada. Esta abordagem exigiu um tempo de treinamento maior. Isto por que, dependendo do conjunto de dados, a quantidade de classificadores possíveis pode ser grande. Todavia, uma vez que todos os classificadores foram treinados, o tempo de classificação das instâncias foi análogo ao de um classificador comum.

A segunda versão treina os classificadores à medida que o *ranking* é gerado. Nesta versão, somente os classificadores necessários para construção daquela lista são treinados. Com isso o método dinâmico conseguiu tempos de treinamento melhores do que a versão anterior. Por outro lado, seus tempos de classificação foram piores visto que o modelo é treinado durante a construção da lista de saída.

Para avaliar os resultados de problemas como o da rede, desenvolvemos para este trabalho um conjunto de métricas denominadas *k-Acurácia*, *k-Precision* e *k-Recall*. De forma geral essas métricas apontam um melhor resultado quanto mais bem posicionada estiver a classe verdadeira na lista de saída. Veremos em detalhes no Capítulo 4 como cada uma delas funciona.

O metaclassificador foi programado em Java e utiliza diretamente as classes para aprendizado de máquina do *Framework Weka* (Hall et al., 2009).

Com isso o ele é capaz de utilizar internamente a uma grande quantidade de classificadores diferentes disponíveis nesta ferramenta.

Utilizamos classificadores probabilísticos para criar o *Benchmark* dos testes de performance do método. Esses classificadores foram utilizados para construir as listas usadas como base para a avaliação dos modelos. Isso foi possível pois, eles podem gerar a probabilidade de cada classe para uma dada instância. A partir desta informação montamos um *ranking* de classes para a instância.

Durante a avaliação do método, executamos e analisamos testes do metaclassificador em combinação com classificadores conhecidos como o SVM, a Árvore de Decisão e o KNN. Observamos que o metaclassificador destacou-se nos testes com alguns deles, principalmente com a Árvore de Decisão. Além disso, discutimos os resultados do método quando aplicado à conjuntos de dados reais e sintéticos. Os tempos de execução também foram analisados. Dependendo do numero de classes envolvido o método proposto atinge tempos de execução razoáveis; ainda que mais lentos do que dos classificadores probabilísticos comuns.

1.3 Organização da Dissertação

Os capítulos deste texto estão organizados da forma a seguir. Primeiro são introduzidos os conceitos necessários de aprendizado de máquina para o entendimento do trabalho no Capítulo 2. Este capítulo discute o aprendizado supervisionado, algoritmos de aprendizado e avaliação de performance. Em seguida, no Capítulo 3, o método é discutido em detalhes. Depois os experimentos realizados para testar o método são detalhados e seus resultados são discutidos no Capítulo 4. Posteriormente, no Capítulo 5, uma compilação de trabalhos relacionados à este é apresentada. Por fim, o texto é concluído com nossas considerações finais sobre o trabalho.

2
Conceitos Básicos

Aprendizado de máquina é um dos campos de pesquisa da inteligência artificial e por sua vez da ciência da computação. Esta área de estudos se concentra na pesquisa e desenvolvimento de algoritmos que possam aprender tarefas automaticamente. Tipicamente estes algoritmos geram um *modelo matemático* a partir de um ou mais conjuntos de dados. O modelo é então responsável por desempenhar a tarefa, ou seja, não é necessário escrever um programa para tanto. Aprendizado de máquina tem inúmeras aplicações, entre elas podemos citar filtros de spam, reconhecimento ótico de caracteres, motores de busca, visão computacional, entre outras.

2.1
Aprendizado Supervisionado

Aprendizado supervisionado é inspirado na ideia de aprendizado por exemplos. Isto é, uma grande quantidade de exemplos é fornecida para o algoritmo no intuito de fazê-lo aprender. Todos os exemplos devem ter o mesmo formato. Cada um deles é composto por dois ou mais atributos, sendo que um dos atributos deve ser o alvo da tarefa de aprendizado. Os demais atributos devem constituir informações relevantes que permitam ao algoritmo aprender a tarefa. De forma geral temos que um conjunto com N exemplos é da forma $\{(x_1, y_1), ..., (x_N,\ y_N)\}$ tal que x_i representa o conjunto de atributos informativos do exemplo e y_i é seu atributo alvo.

O algoritmo de aprendizado deve gerar uma função $g : X \to Y$, onde X é o conjunto de entrada e Y o conjunto de saída. Uma vez que processou os exemplos, o algoritmo gera um modelo que representa a função g. Por fim, dados que ainda não foram vistos, desprovidos do atributo alvo, podem ser submetidos ao modelo. Com isso, ele pode executar automaticamente a tarefa para a qual foi treinado, e.g. determinar a qual classe o dado inédito pertence. Veja na Figura 2.1 um diagrama que ilustra o processo de aprendizado supervisionado.

Para exemplificar como funciona o aprendizado supervisionado, consideremos o problema da Figura 2.2. Neste queremos um modelo capaz de diferenciar entre círculos e triângulos. Ou seja, temos um problema de classificação

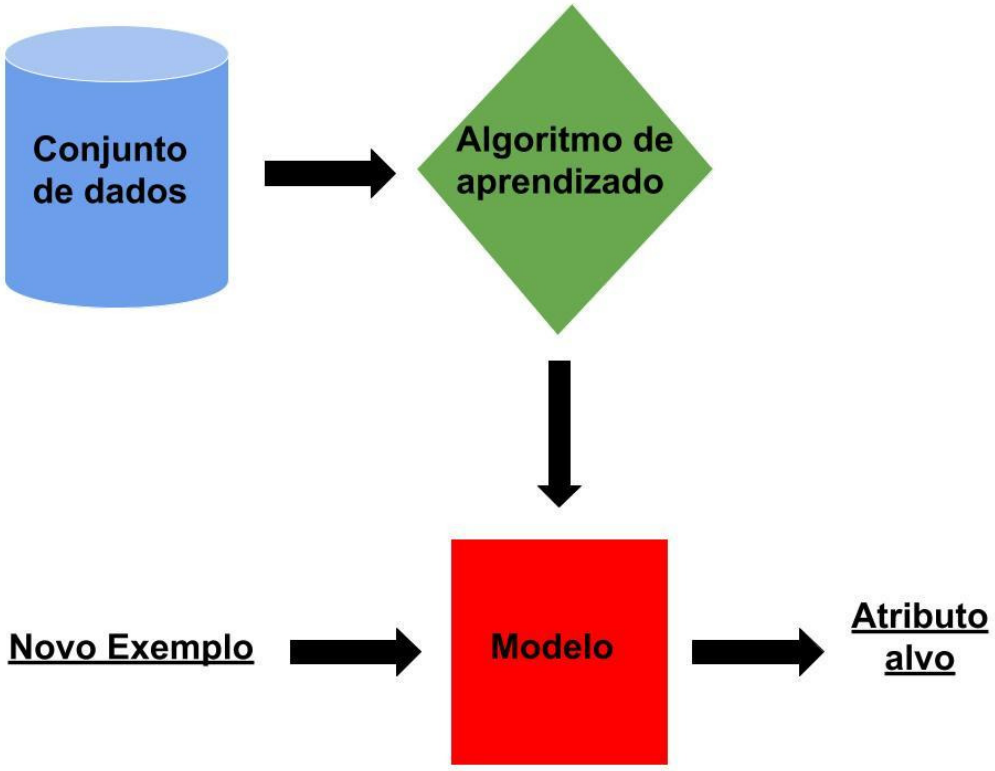

Figura 2.1: Fluxo do aprendizado supervisionado para classificação

binária. Para resolvê-lo precisamos de um modelo que seja capaz de classificar as duas formas geométricas. O algoritmo utilizado para gerar o modelo deve então aprender uma função *f(x)* que mapeia o vetor de entrada x em um único valor de saída. Esta função pode ser definida como:

$$f(x) = \begin{cases} 1 & \text{se } w \cdot x + b > 0 \\ 0 & \text{caso contrário} \end{cases}$$

onde,

- Os valores de saída de *f(x)* representam a resposta do classificador, por exemplo o valor 0 pode indicar um círculo e o valor 1 um triângulo
- x é um vetor de números reais que representam os valores de atributos de entrada, no exemplo podem ser as coordenadas dos centros de cada círculo e triângulo
- w é um vetor de números reais que representam os pesos conferidos aos atributos pelo algoritmo de aprendizado
- b é uma constante de bias

Com isso o problema se resume a calcular os valores do vetor de pesos w. Considere que temos disponível os dados dos diversos exemplos de círculos e triângulos vistos na Figura. Os exemplos envolvidos no processo de aprendizado

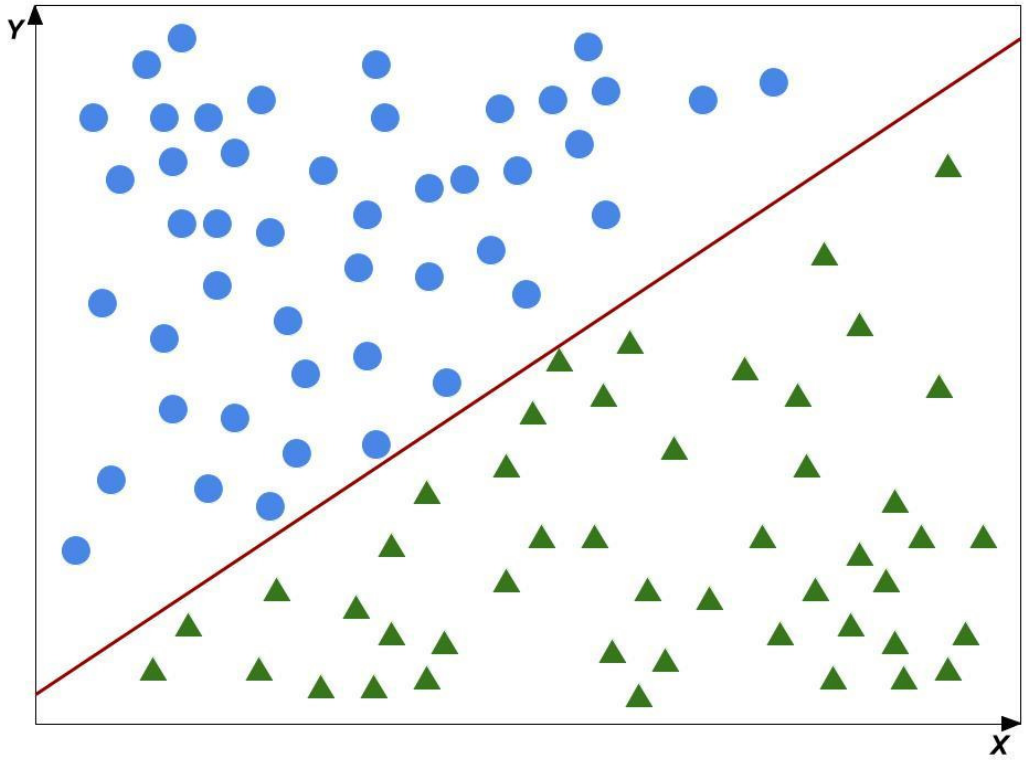

Figura 2.2: Exemplo de separador linear

são denominados *conjunto de dados*. Cada exemplo contém as coordenadas *(x,y)* mais o valor da classe (circulo ou triângulo). O algoritmo supervisionado simples proposto a seguir para resolver este problema é baseado no perceptron.

Algoritmo de aprendizado

Inicializa o vetor w com valores aleatórios. Para cada exemplo do conjunto de dados:

1. Calcula o valor de saída de *f(x)*
2. Calcula a diferença entre o valor obtido e o real (retirado do exemplo)
3. Corrige os valores do vetor de pesos w por valores proporcionais a esta diferença

No exemplo todos os atributos eram números reais. Porém, com a utilização de outros algoritmos os atributos podem ser de diversos tipos: nominal (uma lista de classes), numérico (um número inteiro ou real), uma data, etc.

As tarefas de aprendizado supervisionado mais clássicas são a classificação, vista no exemplo, e a regressão. Quando o alvo da tarefa é um atributo nominal dizemos que esta é uma classificação e quando é um número real dizemos que é uma regressão. Todavia existem outras tarefas nas quais o aprendizado supervisionado pode ser empregado, como é o caso do *ranking*.

Em um problema de *ranking*, queremos que o modelo gere uma lista ordenada de classes como saída. Para tanto, os exemplos fornecidos ao algoritmo de aprendizado também devem ter uma lista ordenada de classes ao invés de um atributo classe único. Isto é, em cada exemplo $(x_i,\ y_i)$ temos que y_i é da forma $[l_1, l_2, ..., l_k]$ onde k representa o número total de classes.

As Tabelas 2.1 e 2.2 dão exemplos de dados que podem ser usados para classificação comum e para *ranking* respectivamente. Os dois exemplos apresentam um certo número de atributos numéricos que podem ser usados pelos algorítimos de aprendizado. Porém eles diferem no atributo classe, o primeiro tem apenas uma classe enquanto o segundo uma lista.

Atributo 1	**Atributo 2**	...	**Atributo k**	**Classe**
3,14	166,55	...	-17	Nodo 10

Tabela 2.1: Exemplo de dados para classificação

Atributo 1	Atributo 2	...	Atributo z	Lista de Classes
42	-33,33	...	0,8	Nodo 2, Nodo 9, Nodo 5

Tabela 2.2: Exemplo de dados para *ranking*

2.1.1 Algoritmos de aprendizado

O perceptron, cujo princípio de funcionamento é o mesmo do algoritmo apresentado anteriormente, é um algoritmo de aprendizado supervisionado clássico encontrado na literatura (Rosenblatt, 1958). Ele serviu como base para métodos mais complexos como por exemplo o *Support Vector Machine* (SVM) (Chang et al., 2011). Entretanto, existem algoritmos baseados em outros princípios. Quando realizamos os experimentos deste trabalho procuramos comparar classificadores com princípios de funcionamento distintos. Foram usados os classificadores: SVM, *Naive Bayes* (Murphy, 2006), Arvore de Decisão (Quinlan, 1986), *Random Forest* (Breiman, 2001) e *k-nearest neighbors* (KNN) (Duda et al., 1973). Descreveremos a seguir de forma breve como esses algoritmos funcionam.

Support Vector Machines.

Como foi dito, o SVM é derivado diretamente do Perceptron. Ambos são classificadores binários lineares não probabilísticos. O SVM também pode ser usado para resolver problemas como o da Figura 2.2. Este classificador considera cada vetor de atributos de entrada x com d dimensões. Ele então calcula um hiperplano, com *d-1* dimensões, para separar os pontos no espaço d-dimensional dos exemplos. No exemplo da Figura temos que d é igual a 2 e portanto o hiperplano é uma reta. Este calculo ocorre iterativamente, a partir do conjunto de treino, de forma análoga ao algorítimo apresentado anteriormente. Além disso, o SVM procura encontrar o melhor hiperplano de separação para os pontos, i.e. aquele mais afastado de todos. Para tanto ele considera a existência uma *margem* que representa o afastamento do hiperplano para os pontos. Por fim ele calcula o hiperplano de margem máxima.

Naive Bayes.

Naive Bayes são classificadores probabilísticos baseados no teorema de Bayes. Este algoritmo cria um modelo de probabilidade condicional que recebe como entrada o vetor $x = (x_1, x_2, ..., x_d)$, com d atributos. Ele assume entretanto que estes atributos de entrada são independentes. Para classificar uma instância o modelo determina a probabilidade de cada classe C_i possível, dada por $p(C_i|x_1, x_2, ..., x_d)$, com a ajuda do teorema de Bayes. Por fim, o classificador utiliza as probabilidades para decidir qual classe escolher baseado

em uma regra preestabelecida, e.g. a classe mais provável.

Árvores de Decisão.

As Árvores de Decisão são modelos de predição que utilizam os atributos de entrada para tomar decisões e desta forma inferir a classe de uma instância. Sendo assim, as folhas da árvore representam classes enquanto os nodos representam decisões baseadas nos atributos. De forma geral, para construir um modelo a partir do conjunto de treino, a árvore de decisão escolhe a cada nodo o atributo que melhor divide o conjunto. Depois que o modelo está pronto, é possível utilizar os atributos de uma nova instância para percorrer a árvore de acordo com as decisões nos nodos. Com isso chega-se até uma folha que determinará a classe da instância. Note que tipos distintos de árvore podem utilizar critérios diferentes para a escolha do atributo decisório dos nodos. O *Ganho de Informação* por exemplo é empregado nas arvores de decisão do tipo C4.5 (utilizadas nos testes conduzidos neste trabalho).

Random Forest.

O *Random Forest* é um método que utiliza um comitê de árvores de decisão para classificação. Isto é, durante o treinamento o algoritmo cria diversas árvores de decisão que compõem o modelo. Cada árvore contribui votando em uma classe e no final a classe mais votada é escolhida. Esse algoritmo utiliza *Bagging* para criar subconjuntos aleatórios do conjunto de treino. Cada subconjunto é utilizado para treinar uma das árvores de decisão. Além disso, o *Random Forest* também emprega o *Feature Bagging*, i.e. cada árvore utiliza somente um subgrupo aleatório dos atributos originais no treinamento.

k-Nearest Neighbors.

O algoritmo **KNN** apenas recebe os vetores de atributos de entrada do conjunto de treino em um primeiro momento. Para classificar uma instância ele simplesmente determina qual é a classe mais frequente entre os k exemplos mais próximos, onde k é definido pelo usuário. Este método pode utilizar diferentes métricas de proximidade para determinar os vetores próximos, uma das mais utilizadas é a distância euclidiana.

2.1.2 Avaliação de Desempenho

Para treinar e avaliar a performance de um algoritmo de aprendizado, tipicamente o conjunto de dados é dividido em dois. O primeiro, *conjunto de treino*, contém os exemplos que efetivamente serão usados para treinar o algoritmo. O restante, *conjunto de avaliação*, é usado para fornecer exemplos inéditos ao modelo. O resultado obtido quando o conjunto de avaliação é

submetido ao modelo é utilizando para calcular as métricas pertinentes. Essa divisão de dados é feita para garantir que o modelo foi capaz de generalizar e não apenas decorou o conjunto de dados (*overfitting*). Note que esta técnica de avaliação acaba por não utilizar uma parte dos dados (conjunto de avaliação) no treinamento. Isso pode criar dificuldades quando a quantidade de dados disponível for muito limitada.

Outra forma de avaliação importante é chamada *validação cruzada*. Neste caso o conjunto de dados é dividido em k subconjuntos. Estes podem ou não manter aproximadamente as mesmas proporções de classes do conjunto de dados original, dependendo do que se deseja fazer. A partir disso o algoritmo de aprendizado é executado k vezes, gerando um modelo diferente por vez. A cada iteração, aquele modelo é treinado com *k-1* subconjuntos como conjunto de treino e o restante (1 subconjunto) como conjunto de avaliação. Desta forma, no decorrer das k iterações do treinamento, todos os dados são usados. Por fim, os resultados de todos os modelos são combinados para gerar as estatísticas finais.

2.1.3 Métricas de Avaliação

Para explicar as métricas de avaliação suponha um exemplo onde temos um modelo *M*. Ele foi treinado para identificar se uma máquina apresenta falha ou não, baseando-se em uma série de medidas recolhidas sobre a mesma (atributos). Aplicamos então esse modelo à um conjunto de 100 instâncias e obtemos um resultado como o da Tabela 2.3.

	Real: com falha	**Real: sem falha**
Previsto: com falha	5 (*true positive*)	5 (*false positive*)
Previsto: sem falha	10 (*false negative*)	80 (*true negative*)

Tabela 2.3: Matriz de confusão do modelo *M*

Os dados da Tabela 2.3 estão separados em *true positive* (TP), *true negative* (TN), *false positive* (FP) e *false negative* (FN). Os dois primeiros indicam o número de instâncias corretamente classificadas e o restante a quantidade classificada incorretamente. Isto é, *TP* é o número de instâncias que foram classificadas como sendo da classe "com falha"e que realmente pertencem à essa classe. Já *TN* é o número de instâncias de outras classes, no exemplo temos somente "sem falha", que foram classificados corretamente. Em contraste, *FP* é a quantidade de instâncias que são da classe "com falha"mas que foram classificadas como sendo de outras classes. Por fim, *FP*

é a quantidade de instâncias da classe "sem falha"que foram erradamente classificadas como sendo "com falha".

A partir desses valores é possível gerar algumas das métricas mais usuais em aprendizado de máquina, as quais apresentaremos nas equações a seguir.

$$Acurácia = \frac{TP + TN}{TP + TN + FP + FN}$$
$$Precision_i = \frac{TP_i}{TP_i + FP_i}$$
$$Recall_i = \frac{TP_i}{TP_i + FN_i}$$

Conforme as métricas foram apresentadas temos que a *Acurácia* tem um valor único para o modelo. Contudo o *Precision* e o *Recall* se referem à uma classe por vez. Note que no exemplo temos duas classes: "com falha"e "sem falha". Logo, baseando-se nas equações, temos no exemplo da Tabela 2.3 um valor de *Acurácia* igual a 85%. Para classe "com falha"temos o *Precision* igual a 50% e *Recall* igual a 33,33%.

Notadamente cada uma dessas métricas é mais apropriada em casos distintos. A *Acurácia* talvez seja a métrica mais intuitiva e simples, visto que ela representa a proporção de elementos corretamente classificados. Mas nem sempre é possível avaliar um modelo da melhor maneira com ela. Considere por exemplo o caso de um modelo *M'* que simplesmente classifica as instâncias como *sem falha*. Ele atingiria uma *Acurácia* de 85% mas seria incapaz de identificar máquinas com falha, seu *Recall* para classe "com falha"seria zero.

Generalizações dessas métricas para problemas multiclasse encontradas na literatura são denominadas *Macro Precision* e *Macro Recall*. Neste caso as métricas são calculadas normalmente para cada classe, mas no final é tirada uma média aritmética. Apresentamos a seguir as fórmulas dessas outras formas de calcular as métricas. Nestas fórmulas considere que k é o número de classes do conjunto de dados e que TP_i, FP_i, e FN_i se referem à classe i.

$$Macro\ Precision = \frac{\sum_{i=1}^{k} \frac{TP_i}{TP_i + FP_i}}{k}$$
$$Macro\ Recall = \frac{\sum_{i=1}^{k} \frac{TP_i}{TP_i + FN_i}}{k}$$

A Tabela 2.4 mostra mais um exemplo, agora com três classes: A, B e C. Neste exemplo, para calcular o *Precision* para classe A, devemos proceder conforme demonstrado a seguir.

$$Precision_A = \frac{TP_A}{TP_A + FP_A}$$
$$Precision_A = \frac{30}{30 + 10 + 5}$$
$$Precision_A = 66,67\%$$

As demais métricas podem ser calculadas de forma análoga aplicando-se as fórmulas. Seus valores são apresentados na Tabela 2.5.

	Real: A	**Real: B**	**Real: C**
Previsto: A	30	10	5
Previsto: B	5	25	5
Previsto: C	5	0	15

Tabela 2.4: Exemplo com três classes

	Classe A	**Classe B**	**Classe C**	**Macro**
Precision	66.67%	71.43%	75%	71.03%
Recall	75%	71.43%	60%	68.81%

Tabela 2.5: Resultados do exemplo com três classes

Para compreender intuitivamente o *Precision* e o *Recall* podemos recorrer à conceitos de *Information Retrieval*. Neste contexto o *Precision* denota qual proporção dos elementos selecionados pelo modelo realmente são relevantes, no caso do exemplo da Tabela 2.3 máquinas com falha. Por outro lado, o *Recall* representa qual proporção dos elementos relevantes presentes nos dados foi selecionada pelo modelo.

Também é importante compreender essas métricas no contexto do problema da rede, onde precisamos enviar equipes para corrigir as falhas. Podemos entender o $Precision_i$ como o número de vezes que equipes são enviadas ao nodo i e ele falhou, dividido pelo número de vezes total que equipes são enviadas à este nodo. Por outro lado, o $Recall_i$ é o número de vezes que equipes são enviadas ao nodo i e ele falhou, dividido pelo número total de vezes que este nodo falhou.

3
Método Proposto

O método proposto deve receber como entrada um conjunto de treino, a partir do qual um modelo é construído. Quando uma nova instância é recebida este modelo pode ser usado para gerar uma lista ordenada de saída (ranking). Esta lista é composta pelos valores de classe (do conjunto de treino) mais prováveis para aquela instância, de forma que o primeiro valor é o mais provável.

O método proposto compreende um metaclassificador que por sua vez é composto por um conjunto de classificadores internos. Estes diversos classificadores são empregados em cascata para gerar a lista ordenada de saída. Note que, cada item da lista é dado por apenas um classificador interno.

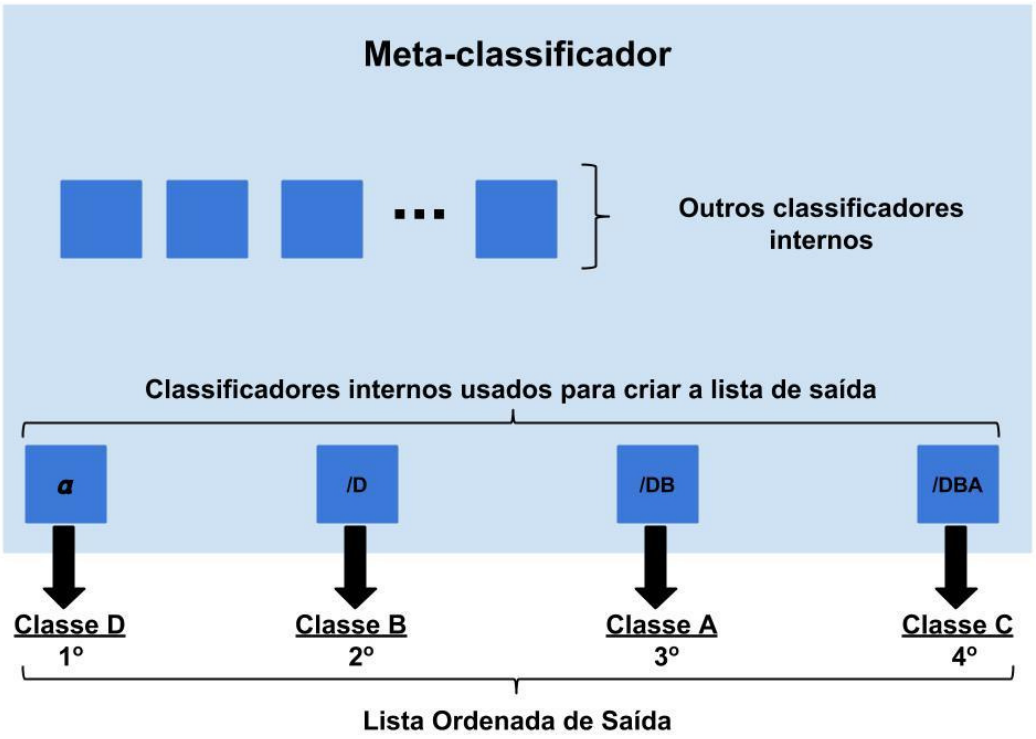

Figura 3.1: Classificadores em cascata e lista de saída.

A Figura 3.1 ilustra como a lista de saída é gerada pelos classificadores internos. Assuma que temos 5 classes distintas no conjunto de dados: A, B, C, D e E. No exemplo da Figura 3.1 queremos gerar uma lista de saída de tamanho

quatro, logo quatro classificadores internos são utilizados. O classificador inicial (alfa),que foi treinado com o conjunto de treino completo, sempre gera o primeiro item de uma lista. O metaclassificador precisa então escolher um de seus classificadores internos para gerar um próximo item. Como a Figura sugere, esta escolha depende de todos os elementos inseridos anteriormente na lista. Repare na notação utiliza para nomear os classificadores, as letras depois barra / indicam as classes retiradas do conjunto de treino original para treinar aquele classificador. Ou seja, o classificador /D foi treinado com um conjunto de treino filtrado de forma a excluir todas as instâncias da classe D, já o /BD excluiu aquelas das classes B e D e assim por diante. Desta forma, o classificador /BD foi usado para gerar o terceiro elemento da lista por que as classes B e D já tinham sido colocadas na mesma. Com isso o conjunto de treino utilizado para este classificador está livre da influência dessas classes.

3.1 Funcionamento do Metaclassificador

Seja T o conjunto de treino do metaclassificador e v o número de valores distintos que seu atributo classe pode assumir. Este conjunto será filtrado de formas diferentes e então utilizado para treinar os classificadores internos. Somente o classificador inicial, usado para gerar o primeiro elemento da lista de saída, é treinado com o conjunto de treino T completo. Qualquer outro classificador interno é treinado com uma versão filtrada de T. De forma geral, o classificador que será usado para classificar o item da lista na posição k, onde $k < v$, deve ser treinado com um conjunto de treino filtrado *k-1* vezes. Estas filtragens retiram sucessivamente do conjunto de treino as instâncias cujas classes já foram colocadas na lista.

No exemplo anterior, da Figura 3.1, o classificador /DBA é usado para gerar o quarto elemento da lista. Este classificador foi treinado com um conjunto filtrado 3 vezes (para retirar as instâncias com as classes A, B e D). O processo de treinamento e classificação para um classificador interno é ilustrado na Figura 3.4.

Note que o metaclassificador não foi desenvolvido para receber um conjunto de treino com uma lista ordenada no gabarito. Ele deve ser treinado com conjuntos que apresentam apenas um único valor no atributo classe. Apos o treinamento com um conjunto de dados deste tipo o metaclassificador é capaz de gerar uma lista ordenada de classes para uma nova instância.

3.2 Motivação

A principal motivação para utilização deste método é a eliminação de ruído dos dados. Como explicamos na seção anterior isso é feito através de filtragens sucessivas do conjunto de treino que eliminam as instâncias cuja classe já foi colocada na lista. Para entender melhor a vantagem desta abordagem considere o caso de conjuntos de dados desbalanceados. Isto é, que tem instâncias de diversas classes diferentes porém uma com a quantidade muito maior do que o restante.

A Figura 3.2 ilustra dois conjuntos de dados desbalanceados gerados aleatoriamente. Ambos tem a mesma quantidade de pontos, que estão agrupados de forma análoga em três grupos diferentes. Considere que os pontos pertencentes ao mesmo grupo tem também a mesma classe, denotada pela cor: azul, vermelho ou verde. Note que os dados são desbalanceados pois a classe verde tem uma quantidade de pontos muito maior do que as outras, são 3000 pontos verdes contra 250 azuis e 250 vermelhos.

Cada grupo de pontos foi gerado seguindo uma distribuição normal bivariada. Apresentamos sua função densidade de probabilidade para um vetor aleatório *(x,y)* a seguir.

$$f(x,y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left\{-\frac{1}{2(1-\rho^2)}\left[\left(\frac{x-\mu_x}{\sigma_x}\right)^2 - 2\rho\left(\frac{x-\mu_x}{\sigma_x}\right)\left(\frac{y-\mu_y}{\sigma_y}\right) + \left(\frac{y-\mu_y}{\sigma_y}\right)^2\right]\right\}$$

onde,

- μ é o vetor de médias dado por

$$\begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix}$$

- ρ é a correlação entre x e y
- σ remete à matriz de covariâncias

$$\Sigma = \begin{pmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{pmatrix}$$

No primeiro cenário da Figura 3.2 os dados da distribuição que contem mais pontos (classe verde) estão afastados do restante. Em contraste, no segundo cenário a média da distribuição de pontos verdes está entre os centros das distribuições de pontos vermelhos e azuis. Depois de realizar testes com os

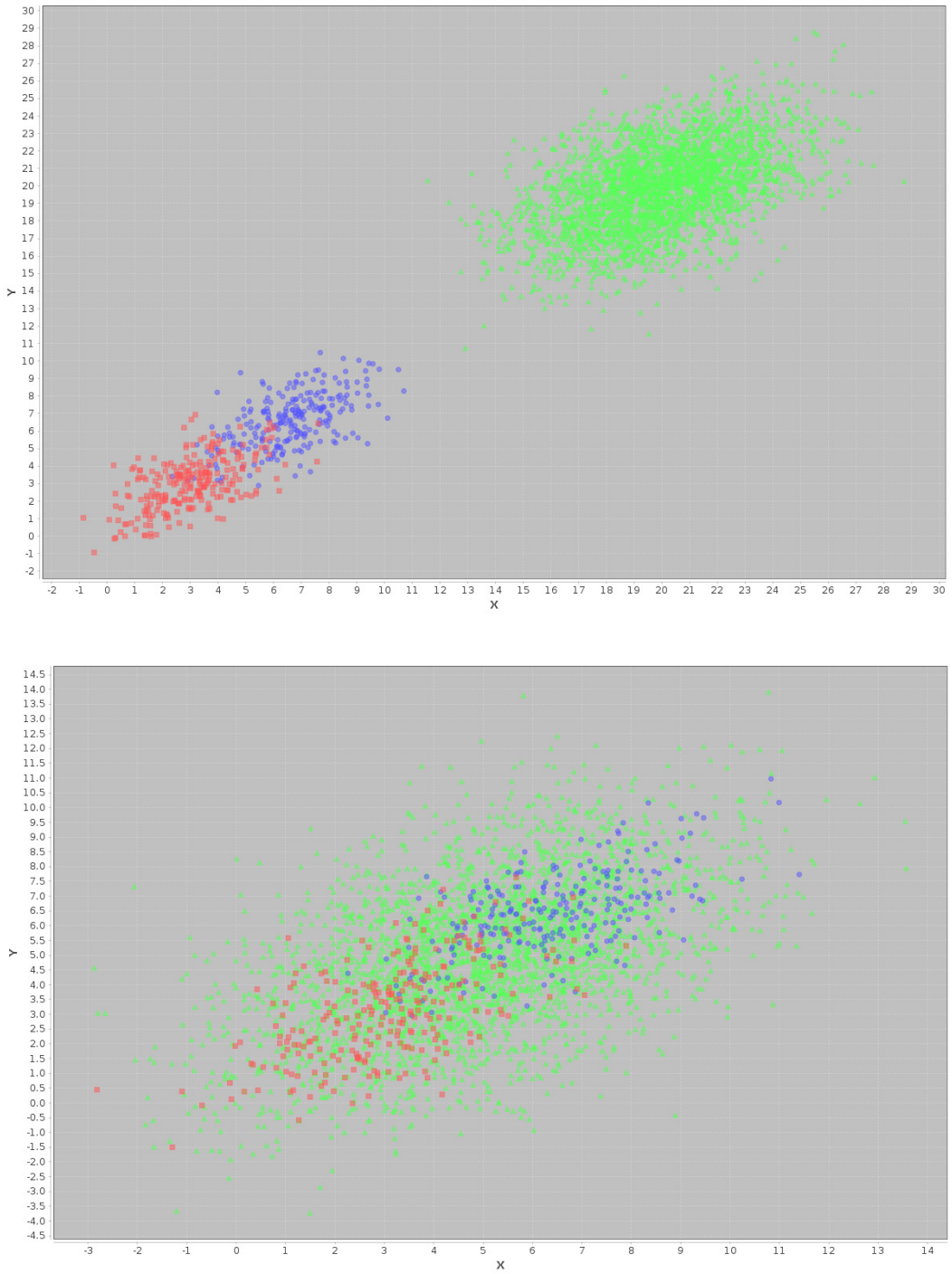

Figura 3.2: Conjuntos de dados desbalanceados.

dois conjuntos de dados percebemos que alguns classificadores, e.g. *k nearest neighbors*, tem acurácias mais elevadas no primeiro cenário. Notadamente, a presença da grande quantidade de pontos verdes próximos ao restante dificulta o aprendizado do classificador e piora o resultado final.

O método introduzido neste trabalho foi proposto no intuito de melhorar o resultado de casos como esse. Suponha que queremos montar uma lista

ordenada das cores mais prováveis para um ponto. De acordo com o que já foi apresentado sobre o método, depois de selecionar um ponto verde para a lista, instâncias dessa classe serão eliminadas do conjunto de treino. Desta forma o classificador utilizado para gerar o próximo elemento será treinado com um conjunto de treino sem a influência da classe mais numerosa (responsável pelo desbalanceamento). Veremos em mais detalhes no capítulo 4 os resultados obtidos por nossa abordagem.

3.3 Versões do método

O método proposto foi desenvolvido em duas versões: estático e dinâmico.

3.3.1 Método Estático

O método estático gera a priori todos os subconjuntos de classe que podem compor a lista de saída. Ele então treina todos os possíveis classificadores, cada um com sua versão filtrada do conjunto de treino, armazenando-os internamente. Este conjunto de classificadores internos é efetivamente o modelo gerado pelo metaclassificador. Este pode ser usado então para gerar rankings ao receber novas instâncias.

O pseudocódigo a seguir descreve o procedimento de treinamento e classificação para a versão estática do método.

Pseudocódigo: Funcionamento do Metaclassificador
Versão Estática

– *Treinamento*
Carrega o conjunto de treino completo
Gera todos os possíveis subconjuntos de classe (de tamanho k ou menor) sem repetições
Para cada combinação de classe que foi gerada faça:
    Copia o conjunto de treino completo
    Remove da cópia os exemplos dessas classes
    Treina um classificador com este conjunto de treino modificado
    Armazena internamente o modelo gerado
– *Classificação*
Para cada item *i* da lista de saída faça:
    Verifica as *i-1* classes que já foram colocadas na lista de saída

Recupera o modelo interno treinado sem essas classes
Utiliza este modelo para gerar o *i-ésimo* item da lista de saída

Note que, quanto mais valores a classe do conjunto de treino original puder assumir, mais classificadores internos comporão o modelo do metaclassificador. Sejam N o número de classificadores gerados pelo método estático, v o número de valores de classe distintos no conjunto universo e k o tamanho da lista que se deseja gerar. De forma geral, considerando que $k \leq v$, temos que:

$$N = \sum_{i=0}^{k} \binom{v}{i}$$

Desta forma, o número N de classificadores gerados aumenta muito com o tamanho k da lista. Com isso o método estático pode não ser indicado para problemas onde queremos gerar uma lista muito grande.

A Figura 3.3 ilustra os classificadores internos gerados quando as possibilidades de valor de classe são A, B, C e D. Na Figura as numerações não somente identificam cada classificador interno, elas denotam como o conjunto de treino foi filtrado para gerar aquele classificador. Além disso, a Figura divide os classificadores internos em camadas. Um classificador que pertence a camada k pode ser usado apenas para gerar um elemento na posição k da lista de saída.

A versão estática pode ser lenta durante a fase de treinamento, pois precisa treinar um grande número de classificadores antes de classificar qualquer nova instância. Esse ônus pode se tornar proibitivo se a quantidade de classes do conjunto de dados for grande, pois aumenta muito o número de classificadores (como vimos na equação anterior). Por outro lado, uma vez treinado o modelo pode ser usado para classificar diversas novas instâncias rapidamente.

3.3.2 Método Dinâmico

O método dinâmico constrói o modelo na medida do necessário, treinando apenas os classificadores requeridos para a construção da lista de saída para a instância em questão. Ou seja, o modelo é treinado ao mesmo tempo que a classificação de instâncias é feita. De forma análoga ao método anterior, os classificadores são armazenados internamente a medida que são treinados. Desta forma, ao construir a lista de saída, os classificadores já treinados são reutilizados. Sendo assim, um mesmo classificador nunca é treinado mais de uma vez.

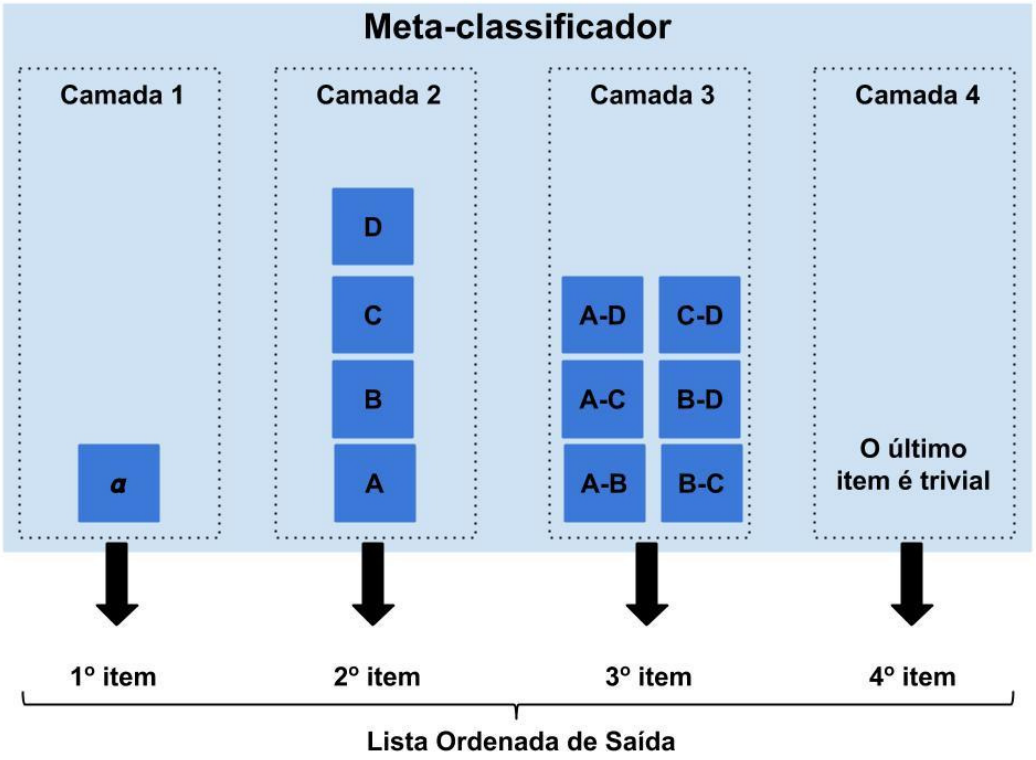

Figura 3.3: Camadas de classificadores internos.

O pseudocódigo a seguir descreve o procedimento de treinamento e classificação para a versão dinâmica do método.

Pseudocódigo: Treinamento de Classificador Interno
Versão Dinâmica

Carrega o conjunto de treino completo
Para cada item i da lista de saída (onde $1 \leq i \geq k$) faça:
    Verifica as *i-1* classes que já foram colocadas na lista de saída
    Copia o conjunto de treino completo
    Remove da cópia os exemplos das *i-1* classes já incluídas na lista
    Treina um classificador com o conjunto de treino modificado
    Armazena internamente o modelo gerado
    Utiliza este modelo para gerar o *i-ésimo* item da lista de saída

Esta segunda versão tende a ser mais rápida do que a anterior, visto que não precisa treinar todas as possíveis combinações de classificadores a priori. Entretanto, como o treinamento do modelo é feito ao mesmo tempo

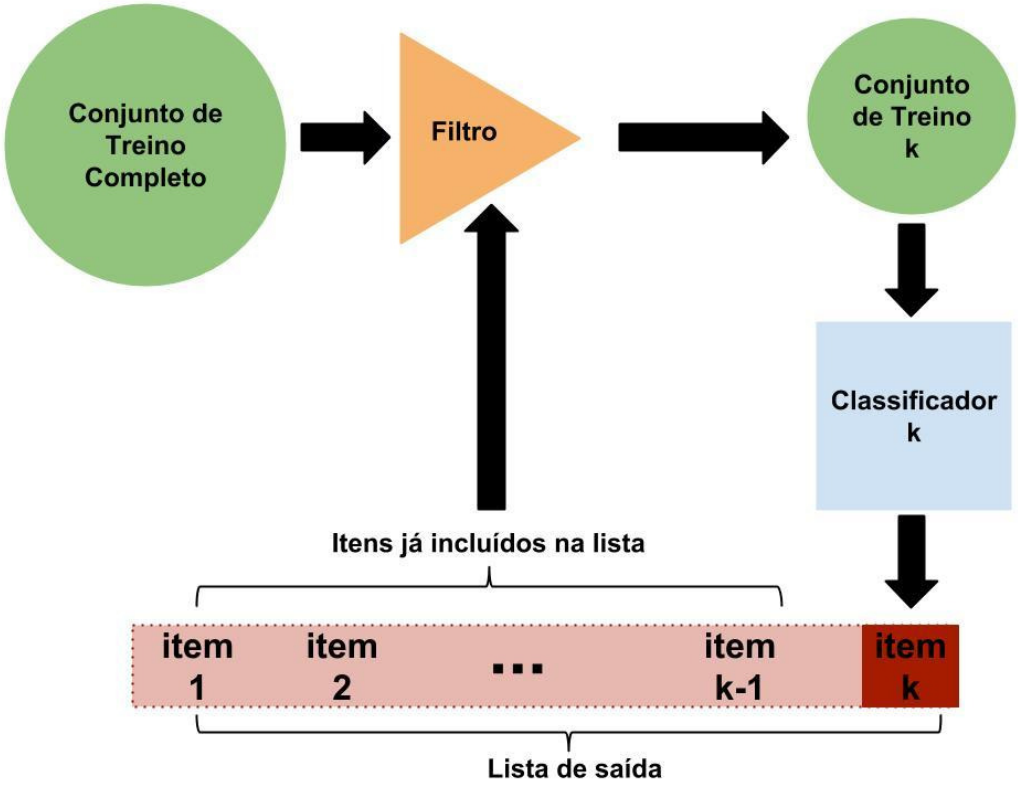

Figura 3.4: Filtragem do conjunto de treino e treinamento de um classificador.

que a classificação de instâncias, o tempo de classificação da versão dinâmica é maior do que a versão estática.

Concretamente, sejam k o tamanho da lista de saída, M o tempo médio de treinamento de um classificador interno e t o tempo médio de classificação de um único item por um classificador interno já treinado. Note que tipicamente temos que $M >> t$. Considere o caso onde o metaclassificador ainda não tem um classificador interno treinado. Usaremos este como limite superior para o tempo de construção da lista de saída para uma nova instância. Este tempo é calculado por $T = k(M + t)$. Por outro lado, o limite inferior para o tempo T ocorre no caso onde o metaclassificador já treinou a priori todos os classificadores internos necessários na construção da lista de saída de uma nova instância. Neste caso temos $T = kt$.

3.4 Vantagens e Desvantagens do método

O método proposto tem a versatilidade de permitir o uso que qualquer classificador internamente. Além disso, as sucessivas filtragens do conjunto de treino removem as instâncias com classes que não são mais pertinentes para construção da lista de saída. Estas duas características podem contribuir para melhoria do resultado final com (1) a escolha do classificador interno mais

adequado e (2) a remoção de ruído do conjunto de treino.

Uma desvantagem deste método é o alto custo computacional do treinamento dos classificadores internos, tanto em processamento quanto em memória. Dependendo da quantidade de valores de classe, o modelo do metaclassificador pode requerer o treinamento de centenas ou milhares de classificadores internos. Esta desvantagem pode vir a ser proibitiva para a versão estática do método.

A versão dinâmica mitiga este problema pois treina os classificadores internos somente quando são necessários. Desta forma ela economiza processamento em comparação com a versão estática. Ainda assim, pode ser necessário grandes quantidades de memória durante a execução do programa. Com isso podendo ser inviável a execução do mesmo na maioria das máquinas. Portanto, um gerenciamento de memória foi desenvolvido. Este garante que a memória alocada não superará um limiar máximo, especificado no momento da execução do programa.

A solução desenvolvida em Java para implementar o método proposto mantém um conjunto interno de objetos, que são classificadores já treinados. Com o gerenciamento de memória, sempre que a memória alocada pelo programa atingir um determinado patamar M, um número c de classificadores é removido do conjunto. Estes classificadores são removidos do menos usado para o mais usado. Tanto M quanto c podem ser ajustados pelo usuário no momento da execução do programa. Note que para que isto funcione M deve ser menor do que a quantidade máxima de memória disponível. Desta forma, em um momento posterior a exclusão destes classificadores internos, o *Garbage Collector* do Java liberará a memória. Com isso está garantido que o programa não poderá ficar com memória insuficiente para sua execução.

4
Estudo Experimental

Neste capítulo discutiremos os testes realizados para avaliar o desempenho do método proposto. Apresentaremos os conjuntos de dados usados e suas características gerais. Os algoritmos utilizados internamente no metaclassificador, assim como os *Benchmark* dos testes também serão mostrados. Por fim, discutiremos os resultados obtidos ao longo dos diversos testes.

4.1
O Framework Weka

Neste trabalho utilizamos o Framework *Weka* para construir o metaclassificador que implementa nosso método proposto. O *Weka* é uma coleção de algoritmos de aprendizado para tarefas gerais de mineração de dados (Hall et al., 2009). Ele contém ferramentas de preprocessamento, classificação, regressão, clusterização, etc. Ele pode ser utilizado para aplicar os algoritmos aos dados por meio de sua interface gráfica ou pode ser chamado diretamente de um código Java.

4.2
Benchmark dos Testes

O *Benchmark* dos testes permitirá uma comparação do método proposto com outra metodologia. Esta metodologia determina uma nota para cada classe e depois as ordena com base neste valor.

Para avaliar a performance do método proposto um *benchmark* específico é empregado em cada teste. Quando um algoritmo é utilizado internamente no metaclassificador, e.g. Árvore de Decisão, o resultado deste mesmo algoritmo sem o metaclassificador é usado como *benchmark*. Isso é possível pois as classes que implementam estes algoritmos, que fazem parte do *framework* Weka, são capazes de gerar uma distribuição de probabilidades de classes como saída. A lista de saída que servirá como *benchmark* do teste é então montada a partir desta distribuição de probabilidades. Isto é, as classes são colocadas na lista na ordem decrescente de probabilidade. Por exemplo, caso tenhamos as classes A, B, C e D com probabilidades 0.2, 0.25, 0.1 e 0.45 respectivamente, a lista de saída será D, B, A e C.

4.3 As métricas de k-Acurácia

Recorde o caso da rede R com n nodos que podem apresentar falhas. Imagine que sempre que a rede apresenta uma falha precisamos enviar uma ou mais equipes à nodos diferentes no intuito de corrigir o problema rapidamente. No pior caso, sem qualquer indicio de onde a falha ocorreu, enviaremos n equipes. Porém, se por exemplo estivermos na dúvida entre apenas três nodos, saberemos que somente três equipes serão necessárias.

As métricas apresentadas nessa seção foram desenvolvidas para avaliar a performance dos modelos de acordo com esse cenário. No caso da rede, precisamos de métricas que indiquem o grau de confiança que podemos ter ao enviar um certo número de equipes. Como foi discutido, o metaclassificador recebe uma instância como entrada, neste caso medições sobre o estado de R com ocorrência de falha. Ele então retorna uma lista ordenada com as k classes (nodos) mais prováveis. De acordo com o problema proposto, a posição da classe verdadeira (nodo com falha) nesta lista é o fator mais importante. Isto é, se a classe verdadeira aparece em uma das três primeiras posições da lista, este é o caso onde precisamos de apenas três equipes para efetuar o reparo.

Desta forma, a *k-Acurácia* é na verdade um conjunto de métricas. Elas variam de 1 até k; i.e. *1-Acurácia*, *2-Acurácia* até *k-Acurácia*; onde k denota o tamanho da lista retornada. Os valores das acurácias para uma dada instância dependem da posição da classe verdadeira na lista. Se ela está na posição i, onde $1 \leq i \leq k$, então as acurácias anteriores à i tem o valor zero e o restante o valor um. Com isso, a *i-Acurácia* indica a certeza da classe verdadeira aparecer na lista até a posição i. Ou, sob a ótica do problema da rede, a confiança em reparar a falha se enviarmos esse mesmo número de equipes.

Observe na Figura 4.1 diversos exemplos de listas e suas acurácias. Para estes exemplos imagine que temos um conjunto de dados com cinco classes diferentes: A, B, C, D e E. Considere também que a classe verdadeira em todos os casos ilustrados na Figura é A (em vermelho). Cada linha do exemplo refere-se então a lista retornada para uma instância distinta e suas consequentes acurácias.

Existe ainda uma diferença no cálculo da *k-Acurácia* para o *benchmark* dos testes. Como a lista de *benchmark* é construída a partir de uma distribuição de probabilidades, podem ocorrer empates. Quando a probabilidade da classe verdadeira está empatada com a de uma ou mais classes, múltiplas listas poderiam ser criadas a partir desta distribuição. Considere o caso onde temos as classes A, B e C com probabilidades 0.4, 0.4 e 0.2 respectivamente e a classe verdadeira é A. Com essas probabilidades podemos ter as listas de saída (1) A,

Instância	Lista Retornada					k-Acurácia				
						1	2	3	4	5
1	A	B	C	D	E	1	1	1	1	1
2	B	A	C	D	E	0	1	1	1	1
3	C	D	A	B	E	0	0	1	1	1
4	E	A	D	B	C	0	1	1	1	1
5	B	A	C	D	E	0	1	1	1	1
6	B	C	D	E	A	0	0	0	0	1

Figura 4.1: Exemplos de listas e suas acurácias.

B e C ou (2) B, A e C. No primeiro caso temos as acurácias de um a três iguais a um. No segundo caso temos a *1-Acurácia* igual a zero e o restante igual a um. Entretanto, por ter acesso às probabilidades, a métrica divide os valores. Teremos então *1-Acurácia* igual 0.5, *2-Acurácia* igual a 1 e *3-Acurácia* igual a 1.

Note que todos os cálculos citados até agora são feitos por instância. Ao gerar listas de saída para múltiplas instâncias, os valores obtidos para cada acurácia são somados, formando um valor total por *i-Acurácia*. Este valor total é então dividido pelo número de instâncias que foram ranqueadas e multiplicado por cem. Com isso os valores das acurácias apresentados neste trabalho estão na forma de percentuais.

Desta forma, podemos calcular os valores percentuais das acurácias ao longo de todos os exemplos da Figura 4.1. Os valores são apresentados na tabela 4.1.

Acurácia	**Valor Percentual**
1	16,67 %
2	66,67 %
3	83,33 %
4	83,33 %
5	100 %

Tabela 4.1: Valores percentuais das acurácias do exemplo da Figura 4.1

4.4 As métricas k-Precision e k-Recall

O *k-Precision* e o *k-Recall* são outros conjuntos de métricas que utilizamos na avaliação do método. Estes são inspirados nas métricas *Precision* e *Recall*, largamente utilizadas em aprendizado de máquina. Porém, da mesma forma como fizemos com as *k-Acurácias*, adaptamos essas métricas para avaliar listas de classes de acordo com a posição da classe verdadeira. Portanto, o *k-Precision* e o *k-Recall* também variam de 1 até k; i.e. *1-Precision*, *2-Precision* até *k-Precision*; onde k denota o tamanho da lista retornada.

Como sabemos, os valores dessas métricas são calculados a partir da contabilização dos *true positives (tp)*, *false positives (fp)* e *false negatives (fn)* (vide Capítulo 2). Estes por sua vez dependem da posição da classe verdadeira na lista. Se A é a classe verdadeira e ela aparece na posição i da lista, onde $1 \leq i \leq k$, então as posições anteriores à i contam como fn_A e o restante como tp_A. Além disso, para cada posição anterior à i também é contado um fp para a classe que foi prevista erradamente no lugar de A.

Da mesma forma que fizemos para as *k-Acurácias* o cálculo é feito de forma diferente para o *benchmark* dos testes. Lembre que quando a probabilidade da classe verdadeira está empatada com a de uma ou mais classes, múltiplas listas poderiam ser criadas a partir desta distribuição. Retornemos ao exemplo da seção anterior, onde tínhamos as classes A, B e C com probabilidades 0.4, 0.4 e 0.2 respectivamente e onde a classe verdadeira é A. Podemos ter duas possíveis listas de saída: (1) A, B e C ou (2) B, A e C. No primeiro caso as posições de um a três são contabilizadas como um tp_A cada. No segundo caso a primeira posição é contabilizada como um fn_A e um fp_B; e o restante das posições como tp_A. Com isso podemos fazer a contabilização de uma forma que leve em consideração todas as possíveis listas de saída. Podemos considerar a primeira posição como 0.5 tp_A, 0.5 fn_A e 0.5 fp_B e o restante conta simplesmente como um tp_A. Note que esse resultado é simplesmente uma média dos resultados das duas possíveis listas de saída.

Observe na Figura 4.2 um exemplo de conjunto de listas retornadas com a contagem de tp, fn e fp para cada classe. Neste exemplo temos um conjunto de dados com três classes (A, B e C). Em cada lista, a classe em vermelho indica a verdadeira.

O *k-Precision* e *k-Recall* para problemas multiclasse são baseados no *Macro Precision* e *Macro recall* explicados no Capítulo 2. Isto é, eles são uma média aritmética do *k-Precision* e do *k-Recall* de todas as classes.

Para efetuar o cálculo dessas métricas para o exemplo da Figura 4.2, primeiro é necessário contabilizar a quantidade de *true positives*, *false positives*

Instância	Lista Retornada	k-Precision/Recall 1	k-Precision/Recall 2
1	A B C	tp_A	tp_A
2	B A C	fn_A fp_B	tp_A
3	B C A	fn_A fp_B	fn_A fp_C
4	C B A	fn_A fp_C	fn_A fp_B
5	B A C	tp_B	tp_B
6	B C A	tp_B	tp_B
7	B A C	tp_B	tp_B
8	C B A	fn_B fp_C	tp_B
9	C B A	tp_C	tp_C
10	A B C	fn_C fp_A	fn_C fp_B

Figura 4.2: Exemplos de listas e suas contagens de tp, fn e fp.

e *false negatives* de cada classe. Esses valores são apresentados nas matrizes das Tabela 4.2, para as métricas de nível 1, e 4.3 para as de nível 2. Com os valores contabilizados em cada matriz é possível então calcular as métricas de nível 1 e 2 para cada classe, cujos resultados são apresentados na Tabela 4.4.

	Real: A	**Real: B**	**Real: C**
Previsto: A	1	0	1
Previsto: B	2	3	0
Previsto: C	1	1	1

Tabela 4.2: Contabilização dos valores para métricas de nível 1 do exemplo da Figura 4.2

	Real: A	**Real: B**	**Real: C**
Previsto: A	2	0	0
Previsto: B	1	4	1
Previsto: C	1	0	1

Tabela 4.3: Contabilização dos valores para métricas de nível 2 do exemplo da Figura 4.2

Métrica	**Classe A**	**Classe B**	**Classe C**
1-Precision	50%	60%	33,33%
2-Precision	100%	66,67%	50%
1-Recall	25%	75%	50%
2-Recall	50%	100%	50%

Tabela 4.4: Contabilização dos valores das métricas de cada classe de nível 1 e 2 para o exemplo da Figura 4.2

Desta forma, podemos calcular os valores percentuais do *Macro k-Precision* e *Macro k-Recall* ao longo de todos os exemplos da Figura 4.2. Os valores obtidos são apresentados na tabela 4.5.

k	**Precision**	**Recall**
1	47.78 %	50 %
2	72.22 %	66.67 %

Tabela 4.5: Valores percentuais do *Macro Precision* e *Macro Recall* do exemplo da Figura 4.2

Para fazer esses cálculos utilizamos as fórmulas de *Macro Precision* e *Macro Recall* introduzidas no Capítulo 2. Por exemplo, para chegar ao valor do *Macro 1-Precision* da Tabela 4.5 devemos fazer os cálculos a seguir. Considere que N_A, N_B e N_C são as quantidades de instancias das classes A, B e C respectivamente.

$$\begin{aligned}\text{Macro 1-Precision} &= \frac{\text{1-}Precision_A + \text{1-}Precision_B + (\text{1-}Precision_C}{N_A + N_B + N_C} \\ &= \frac{50\% + 60\% + 33.33\%}{10} \\ &= 47.78\%\end{aligned}$$

4.5 Resultados dos Testes

Todos os testes reportados neste capítulo foram realizados com validação cruzada com partição em dez grupos. Novamente, os componentes do *framework* Weka foram utilizados para realizar as validações. Além disso, ao executar o programa 10 GB de memória são reservados para o *heap* da Máquina Virtual Java com o comando *java -Xmx10g*.

Todos os testes foram executados em máquinas virtuais no ambiente *Google Cloud Platform*. Estas máquinas tinham a seguinte configuração: sistema

operacional Linux Ubuntu 14.04, duas unidades de processamento (vCPU) e 13 GB de memória RAM.

Na Tabela 4.6 são apresentadas as configurações de algoritmos usados nos testes. Todas as classes utilizadas são do pacote *weka.classifiers*.

Algoritmo	**Classe Weka**	**Opções**
Árvore de Decisão	trees.J48	padrão
Naive Bayes	bayes.NaiveBayes	padrão
Support Vector Machine (SVM)	functions.SMO	padrão
Random Forest	trees.RandomForest	padrão
k vizinhos mais próximos (KNN)	lazy.IBk	K = 5, 7 e 9

Tabela 4.6: Configurações dos algoritmos

Na Tabela 4.7 são apresentados as características gerais dos conjuntos de dados utilizados nos testes. A maioria dos conjuntos de dados foi retirado do repositório *UCI Machine Learning Repository*, disponível na internet no endereço http://archive.ics.uci.edu/ml/. Além disso, em alguns casos o conjunto de dados foi preprocessado e reduzido para facilitar a realização dos diversos testes. A exceção é o conjunto de dados Data-Zero, este não foi retirado do mesmo repositório. O conjunto de dados Data-Zero representa a ocorrência de falhas em uma rede com diversos nodos. Seu atributo classe denota em qual nodo a falha ocorreu.

Conjunto de Dados	**Instâncias**	**Atributos**	**Valores de Classe**
Iris	150	5	3
Wine	178	14	3
Glass	214	10	7
Balance-Scale	625	5	3
Segment-Challenge	1500	20	7
Car	1728	7	4
Data-Zero	2846	202	42
Nursery	3330	9	5
Poker-Hand	3712	11	10
Covtype-01percent	5810	55	7

Tabela 4.7: Conjuntos de dados

4.5.1 Análise dos Tempos de Execução

A Tabela 4.8 ilustra os tempos médios de execução para o conjunto de dados *segment-challenge*, com uma lista de saída com tamanho três e validação cruzada com partição em dez grupos.

A coluna *Configuração* indica como a lista de saída foi construída. Isto é, o teste pode ter empregado o modelo gerado pelo *classificador* diretamente para construir a lista ou uma versão do método proposto (*estática* ou *dinâmica*). A coluna *Treino* informa o tempo médio de treinamento do modelo e a coluna *Teste* o tempo médio que o modelo levou para gerar as listas de saída para as instâncias. Estes valores foram obtidos com a média aritmética de cada Tempo ao longo das iterações da validação cruzada.

Note que, para todos os casos, temos que o tempo total do classificador é menor que os tempos das versões do método proposto. Este resultado já era esperado, visto que o método proposto precisa treinar diversos classificadores para construir a lista de saída. Além disso, a versão estática apresentou tempos totais maiores que a dinâmica. Isso ocorre pois a primeira precisa treinar todos os classificadores possíveis a partir do conjunto de treino, enquanto a segunda treina apenas aqueles que são efetivamente utilizados. Lembre que a versão dinâmica treina o modelo ao mesmo tempo que classifica as instâncias, portanto não é possível separar os tempos de treino e teste. Por fim, uma vez que ambas as versões do método incorrem o mesmo resultado, somente a versão dinâmica foi utilizada nos demais testes apresentados neste capítulo.

Algoritmo	**Configuração**	**Treino (ms)**	**Teste (ms)**	**Total (ms)**
Árvore de Decisão	classificador	91.85	7.89	99.74
Árvore de Decisão	estática	515.80	6.70	522.51
Árvore de Decisão	dinâmica	-	405.87	405.87
Naive Bayes	classificador	11.42	38.26	49.68
Naive Bayes	estática	99.01	40.43	139.44
Naive Bayes	dinâmica	-	100.95	100.95
SVM	classificador	240.37	4.18	244.55
SVM	estática	1646.29	6.45	1652.74
SVM	dinâmica	-	1340.57	1340.57
Random Forest	classificador	449.76	8.62	458.38
Random Forest	estática	5686.47	14.43	5700.89
Random Forest	dinâmica	-	3934.28	3934.28
KNN 5	classificador	0.89	23.76	24.66
KNN 5	estática	90.49	111.69	202.19
KNN 5	dinâmica	-	101.84	101.84

Tabela 4.8: Tempos médios de execução em milissegundos

4.5.2 Análise das Acurácias

A Tabela 4.9 exibe os valores de acurácia médios para cada algoritmo. Estes valores foram obtidos com a média aritmética de cada acurácia ao longo de todos os conjuntos de dados testados. Novamente, a coluna *Configuração*

indica a forma como a lista de saída foi gerada. O valor *classificador* significa que nestes testes usamos a distribuição de probabilidades do classificador para gerar a lista de saída, ou seja, é o *benchmark* daquele teste. Já o valor *metaclassificador* significa que o método proposto (versão dinâmica) foi utilizado. No intuito de facilitar a comparação, apresentamos as acurácias 1, 2 e 3 do classificador e do metaclassificador sempre em linhas subsequentes.

Além disso, é possível observar de forma rápida na tabela 4.10 quantas vezes cada configuração obteve o melhor resultado em cada acurácia. Note que a contagem da tabela 4.10 se refere aos testes individuais e não às médias, ou seja, um teste por combinação de algoritmo e conjunto de dados.

Algoritmo	**Configuração**	**1-Acurácia**	**2-Acurácia**	**3-Acurácia**
Árvore de Decisão	classificador	78.97	86.85	90.9
Árvore de Decisão	metaclassificador	79.1	91.37	96.37
NaiveBayes	classificador	71.7	85.44	93.68
NaiveBayes	metaclassificador	71.7	85.36	93.64
SVM	classificador	77.86	90.2	95.62
SVM	metaclassificador	77.82	90.16	95.71
RandomForest	classificador	84.24	93.74	97.75
RandomForest	metaclassificador	84.28	93.21	97.67
KNN 5	classificador	80.66	90.84	94.73
KNN 5	metaclassificador	80.45	91.3	96.24
KNN 7	classificador	80.17	90.97	95.3
KNN 7	metaclassificador	79.92	91.28	96.18
KNN 9	classificador	80.03	91.04	95.67
KNN 9	metaclassificador	79.93	90.56	95.93

Tabela 4.9: Valores de acurácia médios por algoritmo

Ganhador	**1-Acurácia**	**2-Acurácia**	**3-Acurácia**
Classificador	20	18	16
Metaclassificador	19	32	26
Empate	31	20	28

Tabela 4.10: Número de vezes que cada configuração ganhou

Observe na Tabela 4.9 que os valores da *1-Acurácia* são sempre muito similares para o classificador e o metaclassificador. Este resultado é esperado pois o primeiro modelo interno do metaclassificador (aquele que foi treinado com todo o conjunto de treino) é sempre igual ao modelo que gera a distribuição de probabilidades para o *benchmark*. Como foi explicado, quando ocorre empate de probabilidades na construção desse *benchmark*, a métrica *k-Acurácia* distribui o valor total entre as posições empatadas. Desta forma, as diferenças na *1-Acurácia* observadas na Tabela 4.9 são devidas ao tratamento diferente nestes casos de empate.

Ainda na Tabela 4.9, note que o metaclassificador destacou-se nos testes com o algoritmo Árvore de Decisão. Ele teve cerca de 4,5% de vantagem na *2-Acurácia* e 5,5% na *3-Acurácia*. Nos demais casos as diferenças nas acurácias foram muito pequenas, em alguns o classificador teve um resultado marginalmente melhor em outros o metaclassificador.

4.5.3 Análise do Precision e Recall

As Tabelas 4.11 e 4.12 exibem respectivamente os valores de *Macro k-Precision* e *Macro k-Recall* médios para cada algoritmo. Estes valores foram gerados calculando-se a média aritmética de cada métrica ao longo de todos os conjuntos de dados testados. Como anteriormente, a coluna *Configuração* denota a forma como a lista de saída foi gerada. Nesta coluna o valor *classificador* indica o *benchmark* do teste enquanto o valor *metaclassificador* o resultado do método proposto (versão dinâmica). Note que, os resultados das métricas 1, 2 e 3 do classificador e do metaclassificador são apresentados em linhas subsequentes.

Algoritmo	Configuração	1-Precision	2-Precision	3-Precision
Árvore de Decisão	classificador	68.34	79.13	86.26
Árvore de Decisão	metaclassificador	68.34	79.69	87.72
NaiveBayes	classificador	62.2	74.23	83.85
NaiveBayes	metaclassificador	62.2	74.06	83.66
SVM	classificador	64.97	76.2	85.36
SVM	metaclassificador	64.97	76.5	86.46
RandomForest	classificador	75.4	87.38	91.01
RandomForest	metaclassificador	75.4	83.18	91.12
KNN 5	classificador	70.46	82.86	84.32
KNN 5	metaclassificador	70.46	81.64	86.82
KNN 7	classificador	70.55	82.78	85.14
KNN 7	metaclassificador	70.55	82.48	86.72
KNN 9	classificador	70.72	82.63	84.86
KNN 9	metaclassificador	70.72	81.34	85.98

Tabela 4.11: Valores de *Macro k-Precision* médios por algoritmo

Além disso, é possível observar de forma rápida nas tabelas 4.13 e 4.14 quantas vezes cada configuração obteve o melhor resultado para cada métrica. Lembre que as contagens nestas tabelas se referem aos testes individuais, i.e., um teste por combinação de algoritmo e conjunto de dados.

É possível observar nas Tabelas 4.11 e 4.12 que os resultados do classificador (*Benchmark*) e o metaclassificador são muito próximos. Em alguns casos o metaclassificador ganha por margens maiores, como é o caso do KNN 5 na *3-Precision* e da Árvore de Decisão no *2-Recall* e *3-Recall*. Entretanto em

Algoritmo	Configuração	1-Recall	2-Recall	3-Recall
Árvore de Decisão	classificador	68.33	77.28	81.83
Árvore de Decisão	metaclassificador	68.33	79.75	87.38
NaiveBayes	classificador	62.08	74.43	84.15
NaiveBayes	metaclassificador	62.08	74.41	84.08
SVM	classificador	64.38	74.77	85.16
SVM	metaclassificador	64.38	74.88	85.82
RandomForest	classificador	71.87	83.16	89.69
RandomForest	metaclassificador	71.87	80.38	89.66
KNN 5	classificador	66.77	79.63	84.09
KNN 5	metaclassificador	66.77	78.33	86.01
KNN 7	classificador	65.91	79.49	84.75
KNN 7	metaclassificador	65.91	77.51	85.42
KNN 9	classificador	65.34	79.59	84.45
KNN 9	metaclassificador	65.34	74.66	84.68

Tabela 4.12: Valores de *Macro k-Recall* médios por algoritmo

Ganhador	1-Precision	2-Precision	3-Precision
Classificador	0	20	13
Metaclassificador	0	32	38
Empate	70	18	19

Tabela 4.13: *Macro k-Precision*: Número de vezes que cada configuração ganhou

Ganhador	1-Recall	2-Recall	3-Recall
Classificador	0	21	0
Metaclassificador	0	30	0
Empate	70	19	70

Tabela 4.14: *Macro k-Recall*: Número de vezes que cada configuração ganhou

outros casos ele perde, como ocorre com o *Random Forest* no *2-Precision* e no *2-Recall*. Nos demais casos as diferenças entre classificador e metaclassificador foram muito pequenas (inferiores à 1%).

4.6 Testes com Dados Sintéticos

Nos experimentos com conjuntos de dados retirados de bases públicas as métricas não indicaram um diferença clara entre o resultado do metaclassificador do *benchmark*. Em alguns casos o metaclassificador foi melhor, em outros o foi pior, geralmente com uma pequena margem de diferença. Entretanto, no intuito de entender melhor o comportamento do método proposto decidimos

conduzir uma série de testes com dados sintéticos. Estes dados foram construídos para emular características específicas nos dados, como por exemplo as quantidades de instâncias de cada classe balanceadas ou não. Desta forma pudemos utilizar as métricas *k-Acurácia*, *k-Precision* e *k-Recall* para entender como o método proposto responde à essas características dos conjuntos de dados.

4.6.1 Descrição dos conjuntos de dados Sintéticos

Todos os conjuntos de dados sintéticos que criamos são compostos por três classes: azul, vermelha e verde. Para gerar os pontos dessas classes utilizamos distribuições normais bivariadas. Veja a seção 3.2 para mais detalhes sobre esse tipo de distribuição. Sendo assim, cada instância tem dois atributos numéricos gerados aleatoriamente de acordo com a distribuição de sua classe. Utilizamos a mesma matriz de covariâncias nas distribuições das três classes (apresentada abaixo). Por outro lado, variamos as médias das distribuições e as quantidades de pontos de cada classe.

$$\text{Matriz de Covariâncias} = \begin{pmatrix} 2,5 & 1,5 \\ 1,5 & 2,5 \end{pmatrix}$$

Todos os conjuntos de dados criados tem 4500 pontos. Entretanto versões com proporções diferentes de instâncias das classes foram criadas: *balanceado*, *normal* e *desbalanceado*. A tabela 4.15 mostra as quantidades de instâncias de classes para as diferentes versões de dados. A Figura 4.3 apresenta um conjunto de dados de cada versão. Note que os três conjuntos de dados da Figura tem quantidades diferentes de instâncias de cada classe. Todavia as médias das distribuições de cada classe nos diferentes conjuntos de dados são as mesmas.

Conjunto de Dados	**Verde**	**Azul**	**Vermelho**
Balanceado (B)	1500	1500	1500
Normal (N)	2500	1000	1000
Desbalanceado (D)	4000	250	250

Tabela 4.15: Versões de quantidades de instâncias de cada classe

Além disso, os conjuntos de dados com a mesma distribuição também foram gerados em três versões diferentes. Estas versões se diferenciam pelo posicionamento relativo das distribuições de classes: *próxima*, *adjacente* e *distante*. A Figura 4.4 ilustra essas três versões para o conjunto de dados *desbalanceado*. Note que na versão *próxima* a média da classe dominante (verde) está entre as médias das outras duas classes. Nas versões *adjacente*

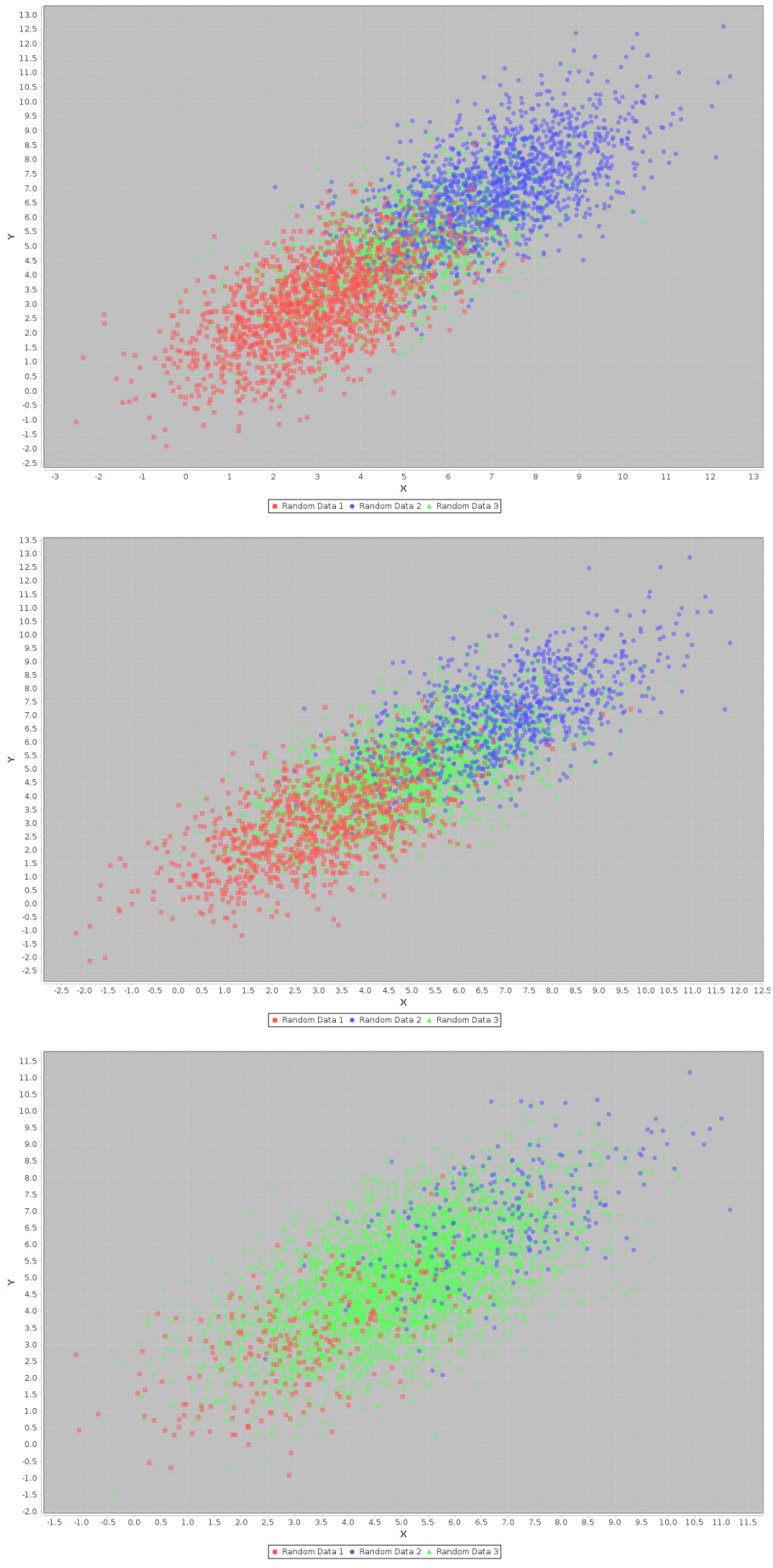

Figura 4.3: Conjuntos de dados sintéticos com diferentes quantidades de instâncias de cada classe

e *distante* a média da classe dominante é afastada sucessivamente das outras duas classes.

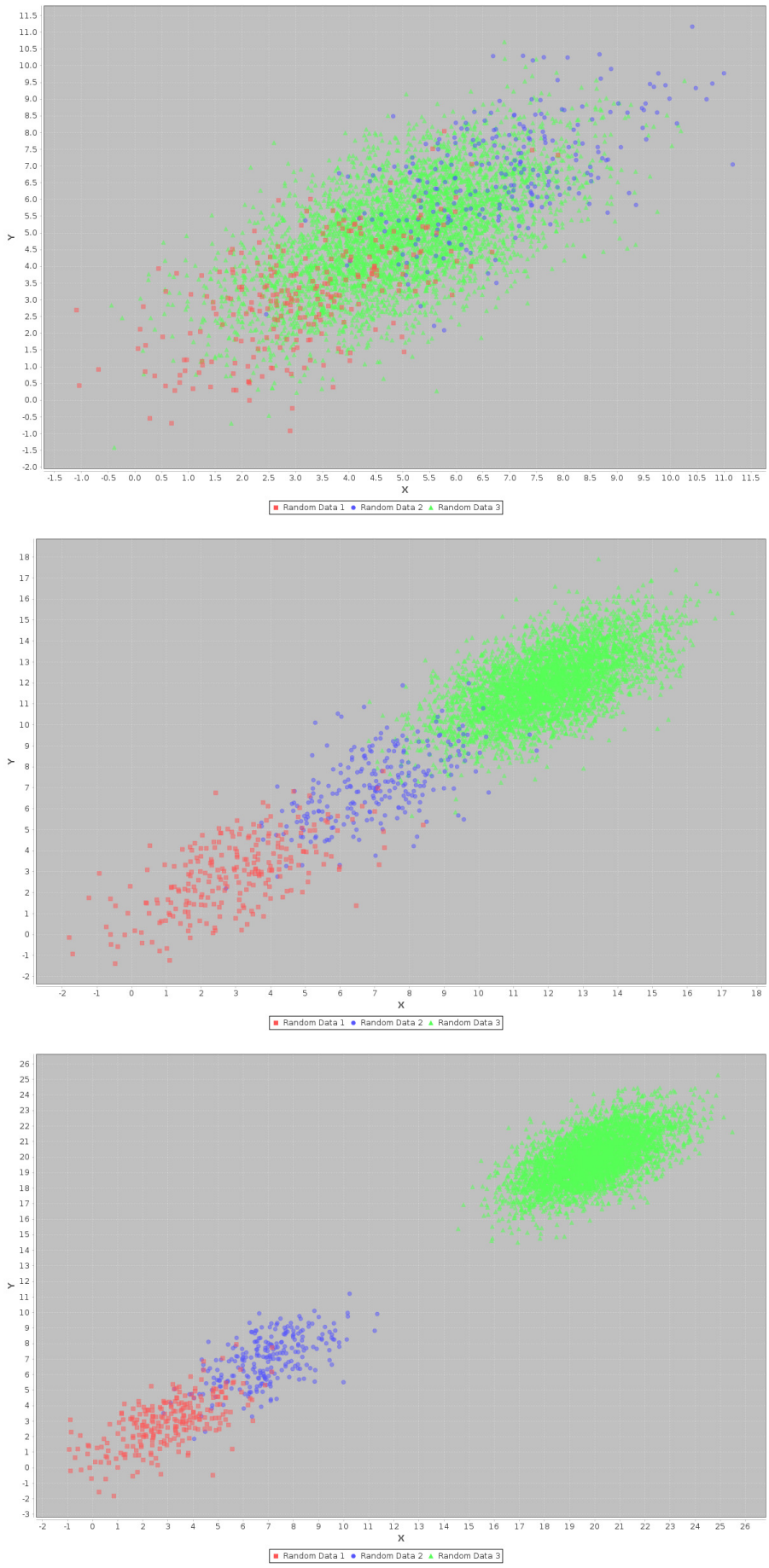

Figura 4.4: Conjuntos de dados sintéticos com a média da distribuição da classe dominante (verde) deslocada

4.6.2 Resultados dos Testes com os Dados Sintéticos

Utilizamos nove conjuntos de dados sintéticos diferentes nos testes apresentados nesta seção. Cada conjunto combina a característica *balanceado*, *normal* ou *desbalanceado* com uma das configurações de posições de distribuições de classe *próxima*, *adjacente* ou *distante*. Com isso pudemos testar o comportamento do método de duas formas. Na primeira verificamos como o método reage à variações no balanceamento entre classes. Na segunda estudamos a influência da posição das distribuições de classes, i.e. elas podem estar sobrepostas, ter apenas uma interseção parcial ou serem totalmente disjuntas.

Para medir os resultados utilizamos as mesmas métricas *k-Acurácia*, *Macro k-Precision* e *Macro k-Recall* apresentadas anteriormente. Lembre que essas duas últimas são respectivamente as médias aritméticas das *k-Precisions* e *k-Recalls* de todas as classes.

Separamos na Tabela 4.16 alguns dos resultados de destaque encontrados durante os experimentos com dados sintéticos. É importante notar que os resultados nesta tabela são, antes da barra, as diferenças entre os resultados do metaclassificador e do classificador simples. Depois da barra, é apresentado o resultado que o metaclassificador atingiu. Portanto um resultado positivo antes da barra significa que o método superou o *benchmark* por um percentual com aquele valor. Além disso, não apresentamos todos os resultados na Tabela 4.16, apenas aqueles que foram mais informativas neste teste.

Os resultados apresentados na Tabela 4.16 se referem à testes conduzidos com conjuntos de dados cuja posição das distribuições das classes era *próxima*. De fato os centros das distribuições de classes nesses conjuntos de dados são *vermelha* = (3,3), *verde* = (5,5) e *azul* = (7,7). A partir disso, variamos o balanceamento entre as classes ao longo do teste. A Tabela 4.16 identifica se os dados são balanceados (B), normais (N) ou desbalanceados (D), conforme foi detalhado na seção 4.6.1.

É possível observar na Tabela 4.16 que o algoritmo *Árvore de Decisão* ganha por uma margem maior que 6% no *2-Precision Macro* e no *2-Recall Macro* quando os dados são *desbalanceados*. Em contraste, neste mesmo cenário, o ganho é de apenas 1% para a métrica *2-Acurácia*.

Note que a *k-Acurácia* é uma medida geral do desempenho do modelo que simplesmente avalia a quantidade de instâncias corretamente classificadas. Desta forma, uma classe dominante com maior quantidade de instâncias, como é o caso da classe *verde*, tem maior influência no seu valor final. Por outro lado, o *Macro k-Precision* e *Macro k-Recall* são uma *média aritmética* das métricas de cada classe. Portanto, em um cenário com dados desbalanceados, essas

Algoritmo	**Dados**	**2-Acu.**	**2-Prec. (Macro)**	**2-Rec. (Macro)**
Árvore	B	0.24 / 94.16	0.32 / 94.12	0.33 / 94.16
Árvore	N	0.99 / 96.11	1.45 / 94.39	1.45 / 94.17
Árvore	D	1.04 / 98.71	6.8 / 92.81	6.27 / 92.27
KNN 5	B	3.46 / 94.22	7.92 / 94.23	8 / 94.22
KNN 5	N	2.78 / 95.96	5.41 / 94.53	4.4 / 94.01
KNN 5	D	1.69 / 98.76	7.06 / 92.53	9.9 / 92.53
R. Forest	B	0.99 / 93.76	1.64 / 93.75	1.64 / 93.76
R. Forest	N	0.53 / 95.67	1 / 94.22	1.09 / 93.7
R. Forest	D	0.67 / 98.73	3.37 / 92.4	3.61 / 92.4

Tabela 4.16: Valores percentuais de ganho do Metaclassificador em relação ao *Benchmark*

últimas tem a sensibilidade para capturar de forma mais evidente as variações de desempenho do modelo para todas as classes do conjunto de dados.

Seguindo com a mesma análise das as métricas *Macro Precision* e *Macro Recall*, observamos ganhos em outros algorítimos além das *Árvores de Decisão*. Para algoritmo *KNN 5* os ganhos foram de aproximadamente 7% e 10% e para o *Random Forest* de 3,5%. Desta forma podemos concluir que o método proposto foi eficiente para melhorar o resultado desses algorítimos em um cenário com dados desbalanceados e com distribuições de classes muito próximas.

Outro conjunto notável de resultados que podem ser vistos na Tabela 4.16 são aqueles referentes aos dados perfeitamente balanceados. Note que neste contexto o algoritmo *KNN 5* exibe ganhos de cerca de 8% nas métricas *Macro k-Precision* e *Macro k-Recall*. Este cenário de testes é especial pois representa uma tarefa mais difícil para muitos algoritmos. Lembre que neste caso as médias das distribuições das três classes do conjunto de dados estão muito próximas e portanto estas tem uma grande interseção no espaço amostral (vide primeira imagem da Figura 4.3). Este resultado demonstra que o método proposto também melhorou o desempenho desses algoritmos neste caso.

Por fim, ainda é possível observar ganhos de 3,5% a 5,5% no *Macro k-Precision* e *Macro k-Recall* para o algoritmo *KNN 5* nos testes com dados do tipo *normal*. Recorde que estes são um meio termo entre os dados *balanceados* e *desbalanceados*. Ganhos menores são observáveis na mesma situação para o algorítimo *Random Forest*. Novamente este resultado reforça que o método proposto melhora o resultado desses algoritmos para conjuntos de dados cujas classes tem distribuições com médias muito próximas e que portanto

se sobrepõem.

Como citamos, diversos outros testes com dados sintéticos foram feitos mas mostramos apenas os resultados de interesse nesta seção. Realizamos os mesmos testes discutidos aqui com dados onde as distribuições das classes estão mais afastadas (*adjacente* e *distante*), conforme ilustrado na Figura 4.4. Porém nestes demais cenários o metaclassificador não se destacou de maneira substancial. Embora ele tenha apresentado algum ganho na maioria dos casos, estes são iguais ou inferiores a 1%.

5
Trabalhos Relacionados

A *Classificação em Múltiplas Classes* representa um importante subgrupo das tarefas de classificação abordadas pelo aprendizado de máquina. Diferente dos problemas clássicos de classificação, neste caso cada instância está associada à mais de uma classe (Tsoumakas e Katakis, 2007). Além disso, é possível que as classes tenham correlações entre si de forma a influenciar no resultado final, aumentando a dificuldade da tarefa. De maneira formal, a tarefa de classificação em múltiplas classes requer que o vetor de atributos de entrada x seja mapeado em um vetor de saída y com valores binários que representam as classes. Isto é, x é da forma $[a_1, a_2, ..., a_n]$, onde a_j representa um dos n atributos, e y da forma $[c_1, c_2, ..., c_k]$ onde c_j pode assumir o valor zero ou um, indicando se aquela classe está relacionada à instância.

O método proposto nesta dissertação figura entre diversos outros que procuram solucionar problemas de classificação em múltiplas classes. Ainda mais especificamente aqueles voltados para *ranking*, que é uma das tarefas cada vez mais proeminentes em aprendizado supervisionado.

5.1
Métodos de Transformação do Problema

Entre as soluções encontradas na literatura podemos destacar ainda aquelas que fazem a transformação do problema (Tsoumakas e Katakis, 2007). Isto é, elas executam uma transformação do problema de classificação em múltiplas classes de forma que ele possa ser solucionado com uma ou mais classificações em uma classe. Como o método proposto nesta dissertação também se baseia em transformação do problema (filtragens sucessivas do conjunto de dados), a seguir apresentaremos alguns destes métodos.

5.1.1
Métodos Clássicos

O método de transformação de problema mais famoso é chamado de *Binary Relevance* (Tsoumakas e Katakis, 2007, Godbole e Sarawagi, 2004, Zhang e Zhou, 2005). Este método se baseia em transformar um problema de classificação em múltiplas classes em diversos problemas de classificação bi-

nários. Ou seja, se existem $|L|$ classes diferentes no conjunto de dados, $|L|$ classificadores diferentes são treinados. Cada um deles é responsável pela classificação binária referente à uma das $|L|$ classes do conjunto de dados. Note porém que este método não é capaz de modelar qualquer tipo de correlação entre as classes, portanto não pode ser usado para gerar listas ordenadas por relevância.

Label Powerset é mais um método clássico encontrado na literatura (Tsoumakas e Vlahavas, 2007, Read et al., 2008). Este método se baseia em transformar cada possível combinação entre as $|L|$ classes em novas classes atômicas. Com isso o problema pode ser resolvido por um classificador para múltiplas classes comum. Além disso, este método leva diretamente em consideração a correlação entre classes. Todavia, a desvantagem do *Label Powerset* é sua complexidade de tempo no pior caso.

5.1.2 Outros Métodos

O método denominado *RAkEL* (*RAndom k labELsets*) é baseado em *Label Powerset* (Tsoumakas et al., 2007). Sua proposta é segmentar aleatoriamente o conjunto de dados de acordo com as classes. Desta forma diversos subgrupos menores (*labelsets*) podem ser criados. Estes subgrupos também podem ou não ter classes em comum. Um classificador é treinado para cada subgrupo utilizando o método *Label Powerset* citado anteriormente. Para classificar uma nova instância o método combina os resultados de todos os classificadores.

Existem diversos métodos que transformam um conjunto de dados com múltiplas classes por exemplo em outro com uma única classe por exemplo (Boutell et al., 2004, Chen et al., 2007). Nestes métodos a proposta é que cada exemplo (x,Y) seja decomposto em $|Y|$ exemplos (x, l) onde $l \in Y$. Com isso um único classificador probabilístico pode ser treinado por meio do conjunto de dados transformado. É possível então classificar uma nova instância a partir da distribuição de probabilidades devolvida por tal classificador. As classes pertinentes podem ser selecionadas entre aquelas com probabilidade superior a um dado patamar, e.g. 50 %.

Entre os modelos de transformação de problema também podemos citar o de classificadores em cascata (Read et al.,2009). No caso de um conjunto de dados D com $|L|$ valores de classe diferentes, $|L|$ classificadores são utilizados em cascata. O conjunto de dados de cada classificador é acrescido sucessivamente de um novo atributo binário, que informa o resultado do classificador anterior. Com isso os classificadores em cascata são capazes de utilizar, em certo nível, a informação de interdependência entre classes. Notadamente, a

ordem dos classificadores impacta na acurácia final. Para resolver este problema o artigo introduz o esquema de conjuntos de classificadores em cascata. Neste caso, subconjuntos aleatórios de D são utilizados para treinar cada classificador. Além disso, cada classificador é treinado com uma ordem aleatória de classes. Ao classificar um novo exemplo a lista de saída é dada pela votação por comitê. Desta forma o conjunto de classificadores em cascata é capaz de gerar um ranking diretamente.

Ranking by pairwise comparison é um método que transforma um conjunto de dados com múltiplas classes em diversos conjuntos de dados com a classe binária (Hüllermeier et al., 2008, Mencia and Fürnkranz, 2008, Furnkranz et al., 2008). Um novo conjunto é criado para cada par de classes do conjunto de dados original. Este encorpora os exemplos que contemplam uma classe ou a outra, mas não as duas. Um classificador binário é então treinado para cada um desses conjuntos no intuito de aprender a diferenciar entre as duas classes em questão. Por fim, esses classificadores são empregados em conjunto para gerar a lista ordenada de saída referente à uma nova instância.

6
Conclusão

Retornemos agora ao problema da rede R com n nodos que podem apresentar falhas. Considere que temos conjuntos de dados com medições sobre o funcionamento da rede os quais podemos utilizar para aprendizado supervisionado. Imagine que quando ocorre uma falha é necessário deslocar equipes para efetuar os reparos, mas essa operação tem alto custo. Também não é aceitável que a rede fique inoperante por muito tempo. No pior caso teríamos que enviar n equipes para verificar todos os nodos rapidamente.

O ideal neste cenário seria descobrir o nodo falho e dirigir apenas uma equipe ao local, minimizando assim o custo e tempo de reparo. Mas recorde que é muito difícil prever, com total certeza, onde uma falha ocorre nesta rede usando aprendizado de máquina tradicional. Por isso propomos a utilização de *ranking* para criar listas com os nodos mais prováveis onde a falha ocorreu. Se garantirmos que o nodo com falha está entre as k primeiras posições da lista, saberemos que apenas k equipes precisam ser enviadas para resolver o problema.

Note que as métricas *k-Acurácia*, *k-Precision* e *k-Recall*, propostas neste trabalho, capturam exatamente esta problemática. Por exemplo, se temos um valor de *3-Acurácia* de 100% isso garante que se enviarmos 3 equipes, uma à cada nodo das três primeiras posições do *ranking*, conseguiremos encontrar e corrigir a falha.

Como vimos, o método proposto foi implementado na forma de um metaclassificador que pode utilizar qualquer outro classificador internamente. Exploramos essa característica durante os estudos experimentais, onde o metaclassificador foi testado em conjunto com diversos classificadores amplamente conhecidos.

Outra característica importante do método proposto é o tratamento dado ao conjunto de treino. Isto é, à medida que o metaclassificador gera o *ranking*, ele também remove dos dados as instâncias das classes que já foram colocadas na lista. Desta forma o classificador interno responsável por gerar o próximo elemento da lista é treinado com dados sem a influência das classes anteriores.

Durante o estudo experimental conjuntos de dados sintéticos foram construídos para avaliar essa característica. Alguns desses dados simulavam

situações onde a influência de uma classe poderia prejudicar o desempenho de um classificador, e.g. dados desbalanceados com uma classe dominante. Neste contexto observamos ganhos de desempenho do método proposto em relação o *Benchmark*.

Os testes conduzidos neste trabalho também utilizaram diversos dados retirados de bases públicas. De forma geral, conseguimos identificar que o método proposto contribui para a melhoria do desempenho de alguns classificadores na geração de *rankings*. Em grande parte dos casos essas melhorias foram pequenas, da ordem de 1%. Porém podemos destacar que com os algoritmos *Árvore de Decisão* e *k-Nearest Neighbors* ganhos maiores foram medidos. Para reforçar esta conclusão exploramos métricas além da *k-Acurácia*. Durante o estudo com dados sintéticos, utilizando o *Macro k-Precision* e o *Macro k-Recall*, pudemos observar com maior clareza esses ganhos de desempenho.

Não obstante o foco deste trabalho tenha sido o problema da rede, outras tarefas relevantes tem características análogas. Muitas destas são aplicações de comum utilização do público geral atualmente como o *ranking* de páginas na *web*, sistemas de recomendação de produtos e sistemas de reconhecimento de voz e imagens. Portanto, o metaclassificador talvez possa ser eficaz em alguns desses casos. Além disso, note que as métricas desenvolvidas se mostraram pertinentes na avaliação de modelos aplicados à problemas de *ranking*. Por isso, elas podem contribuir de forma geral para a análise de problemas desse tipo. Sendo assim, o que foi produzido neste trabalho, tanto para a melhoria de geração de *rankings* como para a análise de desempenho dos modelos, tem potencial para ser aplicado na solução de diversos problemas reais.

Referências bibliográficas

BOUTELL, M.; LUO, J.; SHEN, X. ; BROWN, C.. **Learning multi-label scene classification**. Pattern Recognition, 37:1757–1771, 2004.

BREIMAN, L.. **Random forests**. Machine learning, 45(1):5–32, 2001.

CHANG, C.-C.; LIN, C.-J.. **Libsvm: A library for support vector machines**. ACM Transactions on Intelligent Systems and Technology (TIST), 2(3):27, 2011.

CHEN, W.; YAN, J.; ZHANG, B.; CHEN, Z. ; YANG, Q.. **Document transformation for multi-label feature selection in text categorization**. Artificial Intelligence, p. 451–456, 2007.

DUDA, R. O.; HART, P. E. ; OTHERS. **Pattern classification and scene analysis**, volumen 3. Wiley New York, 1973.

FÜRNKRANZ, J.; HÜLLERMEIER, E.; MENCIA, E. L. ; BRINKER, K.. **Multilabel classification via calibrated label ranking**. Machine Learning, 2008.

GODBOLE, S.; SARAWAGI, S.. **Discriminative methods for multi-labeled classification**. PAKDD '04: 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining, p. 22–30, 2004.

HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B.; REUTEMANN, P. ; WITTEN, I. H.. **The weka data mining software: An update**. SIGKDD Explorations, 11, 2009.

HÜLLERMEIER, E.; FÜRNKRANZ, J.; CHENG, W. ; BRINGER, K.. **Label ranking by learning pairwise preferences**. 7th IEEE International Conference on Data Mining, 2008.

MENCIA, E. L.; FÜRNKRANZ, J.. **Pairwise learning of multilabel classifications with perceptrons**. IEEE International Joint Conference on Neural Networks (IJCNN-08), p. 2900–2907, 2008.

MURPHY, K. P.. **Naive bayes classifiers**. University of British Columbia, 2006.

QUINLAN, J. R.. **Induction of decision trees**. Machine learning, 1(1):81–106, 1986.

READ, J.; PFAHRINGER, B. ; HOLMES, G.. **Multi-label classification using ensembles of pruned sets**. ICDM'08: Eighth IEEE International Conferenceon Data Mining, p. 995–1000, 2008.

READ, J.; PFAHRINGER, B.; HOLMES, G. ; FRANK, E.. **Classifier chains for multi-label classification**. Proc 13th European Conference on Principles and Practice of Knowledge Discovery in Databases and 20th European Conference on Machine Learning 2009, 2009.

ROSENBLATT, F.. **The perceptron: a probabilistic model for information storage and organization in the brain.** Psychological review, 65(6):386, 1958.

TSOUMAKAS, G.; KATAKIS, I.. **Multi-label classification: An overview**. International Journal of Data Warehousing and Mining, 3:1–13, 2007.

TSOUMAKAS, G.; VLAHAVAS, I. P.. **Random k-l406–417abelsets: An ensemble method for multilabel classification**. ECML '07: 18th European Conference on Machine Learning, p. 406–417, 2007.

TSOUMAKAS, G.; KATAKIS, I. ; VLAHAVAS, I.. **Random k-labelsets: An ensemble method for multilabel classification**. ECML PKDD, the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, 2007.

ZHANG, M.-L.; ZHOU, Z.-H.. **A k-nearest neighbor based algorithm for multi-label classification**. GnC '05: IEEE International Conference on Granular Computing, p. 718–721, 2005.