Grafos e Aprendizado de Máquina

Arthur Rodrigues Soares de Quadros[1]
Henrique Mendonça Castelar Campos[1]
João Pedro Torres de Souza Silva[1]
Sarah Luiza de Souza Magalhães[1]

[1]Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)

Abstract. *Creation of a library for manipulating graphs, with addition, alteration, and removal of vertices and edges, in Python programming language, using representations in Adjacency List, and Adjacency Matrix. Furthermore, there is an implementation of the machine learning semi-supervised algorithm proposed in the article "Learning with Local and Global Consistency", with tests equal to those performed in the article itself, combined with a case study.*

Resumo. *Criação de uma biblioteca para manipulação de grafos, com adição, alteração e remoção de vértices e arestas, na linguagem de programação Python, com uso das representações em Lista de Adjacência e Matriz de Adjacência. Além disso, há também uma implementação do algoritmo de aprendizado de máquina semi-supervisionado proposto no artigo "Learning with Local and Global Consistency", com testes iguais aos realizados no próprio artigo, aliados a um estudo de caso próprio.*

1. Introdução

Grafos são modelos matemáticos utilizados para representação de diferentes relações entre objetos, para isso, utiliza-se principalmente de elementos como vértices e arestas, cada qual podendo possuir rótulos e/ou pesos, sendo que as arestas podem ser também direcionadas ou não. Com todos esses fatores, é possível utilizar grafos para representação de quase qualquer coisa existente, desde dependências de pacotes de instalação à representação de mapas geográficos. Por isso, possuir um sistema de criação automática de grafos para possível representação de todos esses ambientes é de grande ajuda para solução de problemas reais.

O projeto em questão foi realizado na linguagem de programação *Python* por causa de seus múltiplos elementos nativos, sua facilidade de compreensão e sua abundância de recursos implementados por terceiros. Nesse projeto, foram utilizados, notavelmente: *Numpy*, para cálculo numérico e vetorial; *Scipy*, para cálculos mais específicos; *Pandas*, para representação interna dos dados usados no algoritmo de aprendizado de máquina; *Matplotlib*, para representação gráfica dos resultados; e *Scikit-Learn*, para uso da mesma base de dados que foi utilizada no artigo *Learning with Local and Global Consistency* [Zhou et al. 2003].

O algoritmo de aprendizado de máquina proposto, a partir daqui chamado de Inferência Transdutiva (aprendizado semi-supervisionado), foi implementado com o uso da biblioteca Pandas para representação computacional dos dados, com métodos adaptadores que transformam os grafos da biblioteca criada em um *DataFrame Pandas*. Esse

algoritmo proposto é do tipo de aprendizado semi-supervisionado, o que significa que ele utiliza dados que, em sua maioria, não possuem rótulos, sendo que, supondo que os dados utilizados possuam consistência, isto é, sejam próximos/semelhantes, baseando-se nesses poucos rótulos, é feita a classificação/agrupamento do restante dos dados (com os rótulos iniciais), inicialmente sem rótulos. Assim como mostrado na figura abaixo, retirada do *Kaggle Website*.

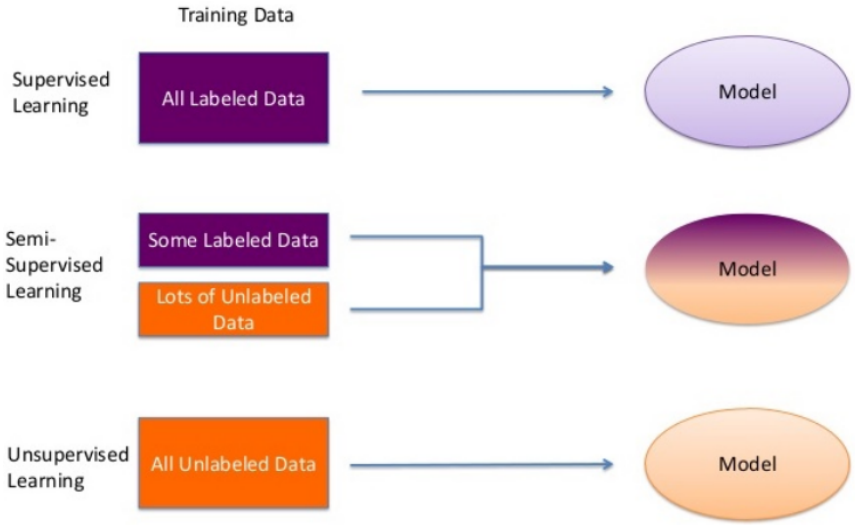

Figure 1. Explicação dos tipos de aprendizado de máquina

O aprendizado de máquina semi-supervisionado funciona da seguinte forma: inicialmente, os dados rotulados são utilizados para treinar o modelo; depois disso, o modelo é utilizado para rotular os dados não-rotulados; e no final, o modelo é treinado com ambos os conjuntos de dados. A principal diferença entre o aprendizado de máquina semi-supervisionado do supervisionado é que o semi-supervisionado trabalha com dois conjuntos de dados: um rotulado e um não rotulado. Esse tipo de aprendizado de máquina é útil quando se possui um conjunto de dados rotulados com um número pequeno de instâncias, e um conjunto de dados não-rotulados que poderia complementar o outro conjunto no treinamento do modelo.

Sobre o estudo de caso e os testes padrão, foram realizados testes com os dados do *Scikit-Learn Make Moons*, que gera dois semi-círculos intercalados. Esse *dataset* é "de brinquedo", isto é, serve apenas para teste de algoritmos de clusterização. Além disso, o algoritmo foi aplicado em um grupo de dados diferente, com aplicação real, para testes do algoritmo.

2. Implementação

Como explicado anteriormente, as implementações foram feitas na linguagem de programação *Python*. Por sua simplicidade de uso e abundância de recursos, ela se torna uma ferramenta perfeita para desenvolvimento, principalmente, de modelos de Inteligência Artificial, que é o principal ponto desse projeto.

2.1. Biblioteca de grafos

Antes de explicar sobre o desenvolvimento da Inferência Transdutiva, é necessário descrever a biblioteca para manipulação de grafos que foi criada. A biblioteca, como é

de se esperar, é destinada para programadores utilizarem. Por isso, não há tratamento de exceções dentro dos métodos da biblioteca, sendo que tudo de errado que seja passado como parâmetro nos métodos implementados apenas levanta um erro, para o programador tratar como bem entender.

Todos os métodos da biblioteca, internamente, usam os endereços de memória de todas as instâncias para manipulação dos dados. O programador pode também utilizar as instâncias dos objetos (seus endereços de memória) se quiser, mas também é possível usar identificadores, seus rótulos, no caso de vértices, além de rótulos e extremidades (vértices), no caso de arestas.

Sobre as classes implementadas para a biblioteca, temos: *Vertex*, representação computacional de um vértice em um grafo; *Edge*, de uma aresta; *AdjacencyList*, de uma Lista de Adjacência; e *AdjacencyMatrix*, de uma Matriz de Adjacência.

2.1.1. *Vertex*

A classe *Vertex* possui os atributos peso e rótulo, além de um método que transforma a classe em um texto para mostrar seus atributos como o usuário desejar. Classe também descrita na imagem a seguir.

```
Vertex:

        Attributes with Get/Set:

        label: (string or integer, NonNullable) -> string if edge is labeled, defined by
                                                   user; else, integer defined by iterator.

        weight: (float, Nullable) -> float with weight if edge is
                                     weighted, defined by user; else, Null.

        Methods:

        to_string() -> return Vertex description as string.
```

Figure 2. Pseudo-código da implementação da classe *Vertex*

Mais especificamente, sobre a definição do rótulo (*label*) por iterador, essa é uma forma de identificar de maneira única os vértices de modo textual (facilmente utilizável pelo programador), como maneira alternativa ao uso do endereço de memória da instância das classes, apesar de que também é um método possível.

2.1.2. *Edge*

A classe *Edge* funciona da mesma forma que a classe *Vertex*, com a adição de uma instância da classe *Vertex* que funciona como o W de uma aresta V, W. Classe também descrita na imagem a seguir.

Sobre o iterador, ele funciona do mesmo modo que na classe *Vertex*. Além disso, é importante destacar que essa classe somente é utilizada na Lista de Adjacência da biblioteca.

```
Edge:

        Attributes with Get/Set:

        label: (string or integer, NonNullable) -> string if vertex is labeled, defined by
                                                   user; else, integer defined by iterator.

        weight: (float, Nullable) -> float with weight if vertex is
                                     weighted, defined by user; else, Null.

        to: (Vertex, NonNullable) -> Vertex instance; w from the edge e = {v, w}.

        Methods:

        to_string() -> return Edge description as string.
```

Figure 3. Pseudo-código da implementação da classe *Edge*

2.1.3. *_Graph*

Sobre a implementação dos grafos propriamente ditos, as implementações da Lista e Matriz de Adjacência possuem de modo agrupado tudo que elas têm em comum em uma classe "privada" *_Graph*. Essas definições são mostradas a seguir.

```
_Graph:

        Attributes with Get:

        n: (integer, NonNullable) -> indicates the amount of vertices in the graph.
        edges_counter: (integer, NonNullable) -> indicates the amount of edges in the graph.
        directed: (boolean, NonNullable) -> indicates if graph is directed or not.
        v_labeled: (boolean, NonNullable) -> indicates if vertices are labeled.
        v_weighted: (boolean, NonNullable) -> indicates if vertices are weighted.
        e_labeled: (boolean, NonNullable) -> indicates if edges are labeled.
        e_weighted: (boolean, NonNullable) -> indicates if edges are weighted.

        Methods:

        complete() -> suposing the graph is simple, return True if the graph is complete; else, False.
        empty() -> return True if the graph is empty; else, False.
```

Figure 4. Pseudo-código da implementação da classe *_Graph*

No grafo genérico, há todas as definições de propriedades gerais da estrutura, sendo que todas são só para visualização e operações internas, não sendo possível alterá-las depois de criado. Além disso, métodos gerais são somente os que não dependem de operações diretas nos vértices ou arestas, isto é, relacionados diretamente com os contadores n e *edges_counter*.

2.1.4. *AdjacencyList*

A Lista de Adjacência funciona estendendo tudo da classe privada *_Graph* e, por cima disso, implementa a estrutura que organiza os vértices e as arestas do grafo no formato de uma "lista". Mais especificamente, nessa implementação em *Python*, essa lista foi implementada como um dicionário, em que cada chave é uma instância da classe *Vertex*, e cada chave possui uma lista de implementações da classe *Edge*, isto é, cada vértice v

possui uma lista de vértices w, formando-se assim as arestas $\{v, w\}$. As definições são mostradas a seguir.

```
AdjacencyList extends _Graph:

        Attributes with Get:

        vertices: (list of "key -> values", NonNullable) -> list of vertices with its edges.
        keys: (Vertices, NonNullable) -> list of vertices outside of vertices dictionary.

        Methods:

        print() -> print the AdjacencyList.
        is_adjacent() -> test if two vertices are adjacent in AdjacencyList.
        to_csv() -> generate external file representing the AdjacencyList in GEPHI's CSV file model.
        to_gdf() -> generate external file representing the AdjacencyList in GEPHI's GDF file model.
        (CRUD operations)_vertex() -> all CRUD operations for vertices in AdjancencyList.
        (CRUD operations)_edge() -> all CRUD operations for edges in AdjacencyList.
```

Figure 5. Pseudo-código da implementação da classe *AdjacencyList*

Como é possível reparar na figura acima, as estruturas de grafos possuem métodos que externalizam a implementação para ser possível utilizar o grafo criado em softwares externos, nesse caso, seguindo os padrões de dados do aplicativo GEPHI para CSV e GDF.

2.1.5. *AdjacencyMatrix*

A Matriz de Adjacência funciona estendendo tudo da classe privada *_Graph* e, por cima disso, implementa a estrutura que organiza os vértices e as arestas do grafo no formato de uma "matriz". Mais especificamente, nessa implementação em *Python*, essa matriz foi implementada como um *array* n-dimensional através da biblioteca *Numpy*, em que as dimensões da matriz são $(n, n, 2)$, em que cada par $\{n, n\}$ possui dois valores, um que é o rótulo da aresta, outro que é o peso da aresta. As definições são mostradas a seguir.

```
AdjacencyMatrix extends _Graph:

        Attributes with Get:

        vertices: (Vertices, NonNullable) -> list of vertices instances in AdjacencyMatrix.
        matrix: (n-dimensional array) -> matrix with all edges instances in AdjacencyMatrix.

        Methods:

        print() -> print the AdjacencyMatrix.
        is_adjacent() -> test if two vertices are adjacent in AdjacencyMatrix.
        to_csv() -> generate external file representing the AdjacencyMatrix in GEPHI's CSV file model.
        to_gdf() -> generate external file representing the AdjacencyMatrix in GEPHI's GDF file model.
        (CRUD operations)_vertex() -> all CRUD operations for vertices in AdjancencyMatrix.
        (CRUD operations)_edge() -> all CRUD operations for edges in AdjacencyMatrix.
```

Figure 6. Pseudo-código da implementação da classe *AdjacencyMatrix*

Da mesma forma que a Lista de Adjacência transforma o grafo em CSVs e GDFs padrão GEPHI, a Matriz de Adjacência também faz.

2.2. Inferência Transdutiva

Inferência Transdutiva, como explicado anteriormente, é um modelo de aprendizado de máquina semi-supervisionado. Mais especificamente, o modelo implementado funciona com as ideias de "suposição de consistência dos dados" e propagação de labels, isto é, com base na pequena taxa de dados com o label conhecido e semelhança entre dados (consistência), as classes dos dados conhecidos são propagadas para os não conhecidos.

2.2.1. Como o algoritmo funciona

Com uma taxa de cerca de 10% dos dados com label, os primeiros 10% da amostra são rotulados, e os outros 90% receberão inferências a partir dos dados rotulados, considerando os seguintes pontos: pontos próximos entre si provavelmente possuirão rótulos iguais; e pontos na mesma estrutura também. Esses pontos são, em suma, o significado matemático de assumir consistência - pontos de mesma classe estão próximos entre si.

Os dados são representados como pontos geográficos, isto é, cada informação do X se torna uma das coordenadas de um ponto n-dimensional, isso significa que todos os dados utilizados são numéricos. Apesar disso, a proximidade dos dados não é meramente determinada a partir de um cálculo de distância entre os pontos. Essa proximidade é definida a partir de uma sequência de passos: cálculo da matriz de afinidade W entre todos os n dados; cálculo da matriz laplaciana S a partir da função $D - WD -$, D sendo a matriz diagonal em que de $W_{i,i}$ é igual a soma de todos os valores da linha i; a partir disso, é calculada a convergência $F*$, sendo que a resposta final da classificação é o argmax da matriz na linha i, para o dado na linha i da amostra utilizada para treino.

2.2.2. Sua implementação

Em relação a implementação da Inferência Transdutiva no projeto, ela é uma aplicação adaptada para uso em biblioteca do projeto [Coutinho 2018]. Nesse caso, o código adaptado permite que o algoritmo seja aplicado em qualquer base de dados em *Pandas* ou *Numpy* para classificação de dados. Essas adaptações são descritas a seguir.

2.2.3. *_PandasAdapter*

A classe *_PandasAdapter* é utilizada como um agrupador dos dados e informações relevantes para o algoritmo de inferência Transdutiva, por exemplo, desvio-padrão, dados com e sem label e quantidade de labels únicos. Os atributos são mostrados abaixo.

```
_PandasAdapter:

        Atributtes with Get/Set:

        std: (float, NonNullable) -> standard deviation defined by the user.
        data: (n-dimensional array, NonNullable) -> all training and testing data.
        n_labeled: (integer, NonNullable) -> amount of labeled data given the total amount and percentage.
        n_not_labeled: (integer, NonNullable) -> total amount - n_labeled.
        x: (n-dimensional array, NonNullable) -> x present in data attribute.
        y: (1-dimensional array, NonNullable) -> y present in data attribute.
        unique_labels: (integer, NonNullable) -> amount of classes available to predict.
```

Figure 7. Pseudo-código da implementação da classe *_PandasAdapter*

Com base nessa classe, a classe da implementação em si do método de aprendizado de máquina irá mapear os dados para os cálculos associados ao funcionamento do algoritmo descrito anteriormente.

2.2.4. *TransductiveInference*

Em relação à predição com o uso do algoritmo propriamente dito, a implementação usa uma classe chamada *TransductiveInference*. Nesse caso, essa classe guarda as matrizes calculadas no passo a passo do algoritmo e depois encontra seu *argmax* para a classificação final. Essa classe, além das matrizes, possui também uma instância interna do *_PandasAdapter* para cálculo de tudo necessário. Os métodos públicos da imagem são mostrados a seguir.

```
TransductiveInference:

        Methods:

        fit_predict() -> use all data available to fit according to the parameters determined in
                         _PandasAdapter and also makes the predictions for all data available. Return
                         a list of labels predicted in the same order as the original Y and also defines
                         colors to each unique label (up to 18) to plot in graph if the user wants to.

        plot() -> plot the results in a graph with multiple colors given pairs {x, y} and its label/color.
                  Only works with bidimensional arrays other than the label itself.
```

Figure 8. Pseudo-código da implementação da classe *TransductiveInference*

Como mostrado na figura acima, o treinamento e predição funcionam em conjunto, de modo mais semelhante a um algoritmo não-supervisionado. Isso faz com que seja inviável de se treinar o modelo com um grupo de dados e testar com um grupo separado, sendo que ele testará com o próprio grupo de treino (grupo rotulado) e complementar com o grupo de teste (grupo não rotulado).

3. Estudo de caso

Para testes do algoritmo, ele foi aplicado em duas diferentes bases de dados, a primeira, o algoritmo *Make Moons*, do *Scikit-Learn*, sendo a mesma utilizada no artigo original,

com resultados idênticos ao original. A seguir, são mostrados os resultados visuais da classificação da primeira iteração com desvio-padrão baixo (dados com consistência) e alto (dados sem consistência).

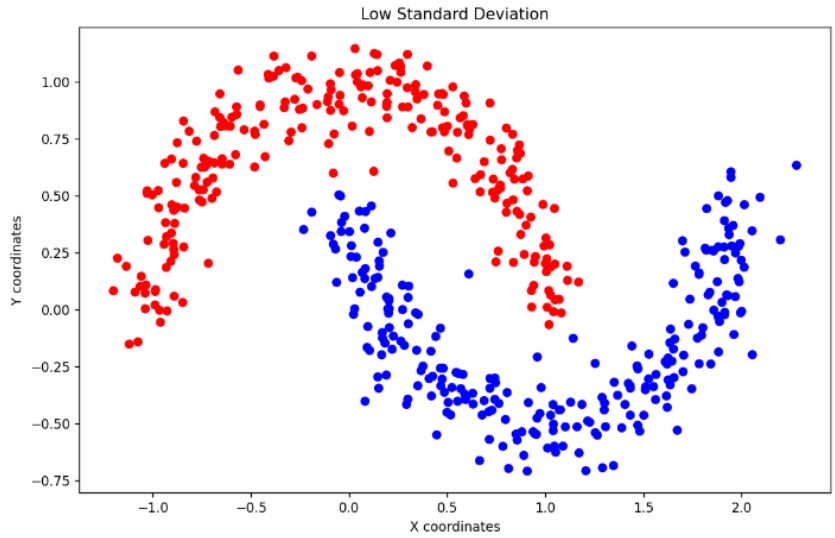

Figure 9. Predição sem convergência com desvio padrão baixo

Nessa execução do algoritmo, temos 99,0% de precisão, com 5 classificações erradas em um total de 500 possíveis. Com a suposição de consistência do algoritmo, quando o desvio padrão dos dados (consistência) é baixo, ele possui um bom desempenho em uma base de dados viável para aplicação de algoritmos de classificação.

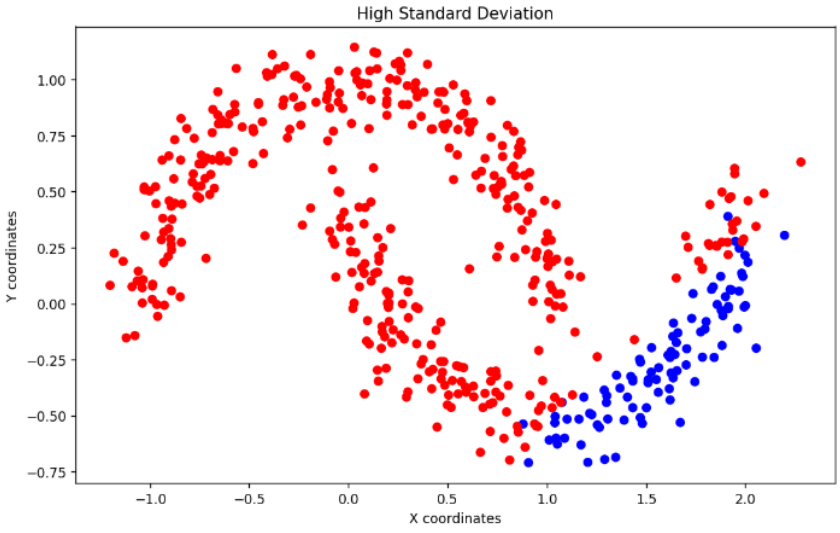

Figure 10. Predição sem convergência com desvio padrão alto

Enquanto isso, na execução do algoritmo com alto desvio padrão, temos 68,6% de precisão, com 157 classificações erradas em um total de 500 possíveis. Com a suposição de consistência do algoritmo, quando o desvio padrão dos dados (consistência) é alto, ele possui um mau desempenho mesmo que a base de dados seja viável para aplicação de algoritmos de classificação.

Além do Make Moons, foi também realizado um teste com base nos dados da RAIS (Relação Anual de Informações Sociais), que contém features de classificação de funcionários e empresas no Brasil. A maior proposta de aplicação desse modelo é a análise da discriminação salarial no Brasil para um possível melhor entendimento de sua distribuição salarial, dado que esse modelo considera dados próximos como sendo de mesma classe.

Com esse meio de funcionamento em mente, é possível tirar conclusões muito direcionadas a respeito dessa proposta. Para o melhor entendimento dessa conclusão, serão explicados primeiro, fatores da análise exploratória dos dados, e segundo, como foram aplicados os testes no algoritmo. A análise exploratória, com análises de distribuição de dados, distribuição salarial entre pessoas com diferentes combinações de fatores sociais, taxa de redução por "defeitos" (fatores sociais que sofrem estigmas) e testes de hipótese, aponta que os grupos sociais estigmatizados tendem a possuir uma redução salarial na área da computação de ao menos 9% em relação ao grupo que mais recebe em média, com os testes de hipótese confirmando que essa redução existente na amostra é expansível para a população inteira.

A Inferência Transdutiva foi aplicada a esses dados com diferentes combinações de features para fins de predição salarial (faixas de salário, variando entre 0 e 11, ou de 0 até 20 salários mínimos ou mais), algumas incluíndo somente features sociais, sendo elas, raça, sexo, deficiência e idade; outras, somente profissionais, como CBO, CNAE, nível de escolaridade e carga horária semanal; e outras, com diferentes combinações entre features profissionais e sociais. Os insights obtidos em relação a esse modelo com os dados da RAIS são voltados principalmente à existência do viés, isto é, através de diferentes abordagens, tudo aponta que os fatores sociais possuem tanto quanto senão mais influência em relação aos fatores profissionais na faixa salarial da pessoa.

Ao aplicar a Inferência Transdutiva nesses dados, é possível notar diversos padrões, principalmente a inabilidade do modelo de classificar os dados usando apenas as features profissionais CBO, CNAE, tempo na empresa, nível de escolaridade, município de trabalho e carga horária semanal, significando que os fatores sociais são a base do resultado da predição, sendo que, depois desses fatores, vêm os fatores profissionais, e isso gera uma matriz de confusão completamente enviesada, como mostrado a seguir.

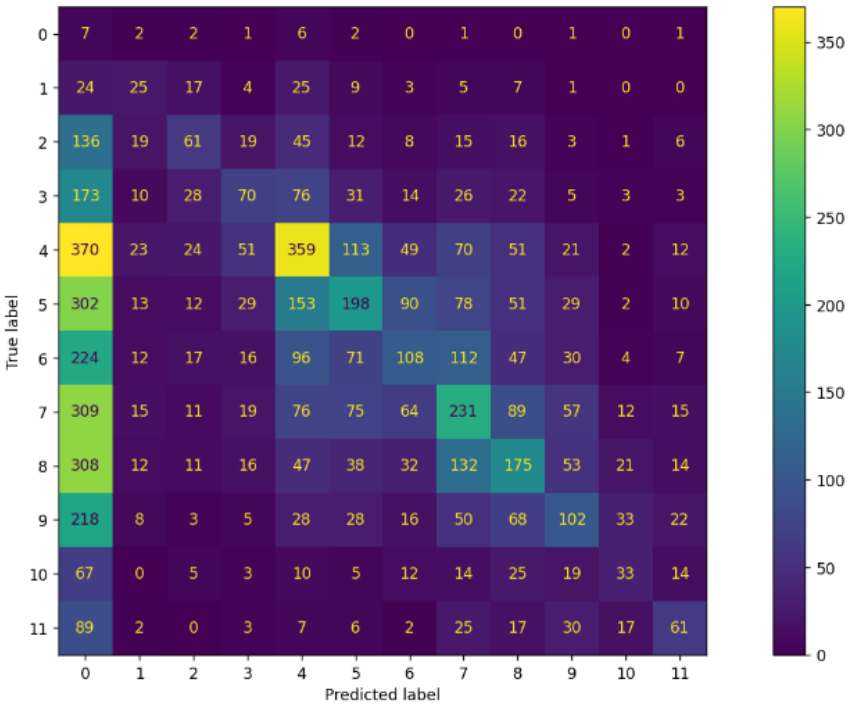

Figure 11. Predição salarial com somente fatores objetivos

Esse teste, em específico, mostra que, sem os fatores sociais e com base nas *features* existentes na base de dados RAIS, não é possível realizar uma predição precisa o suficiente. Isso acontece porque a Inferência Transdutiva supõe que dados semelhantes possuem proximidade entre si, mas, sem os fatores sociais, o modelo se depara com o fato de que há diversas pessoas (pontos geográficos) que possuem qualificações (coordenadas) idênticas ou muito próximas com faixas salariais completamente diferentes. Quando isso ocorre, a predição se torna completamente enviesada, dado que o modelo não encontrou nenhum padrão conclusivo para classificação adequada dos dados, no fim, classificando quase todos com a mesma resposta.

Além desse *insight*, é possível notar justamente o complemento do anterior: quando há dados sociais, o modelo consegue responder algo diferente de zero, mesmo que a precisão não seja muito alta, isto é, ele consegue ao menos encontrar um padrão entre os dados utilizando somente os sociais idade, raça, sexo e deficiência, como mostrado a seguir.

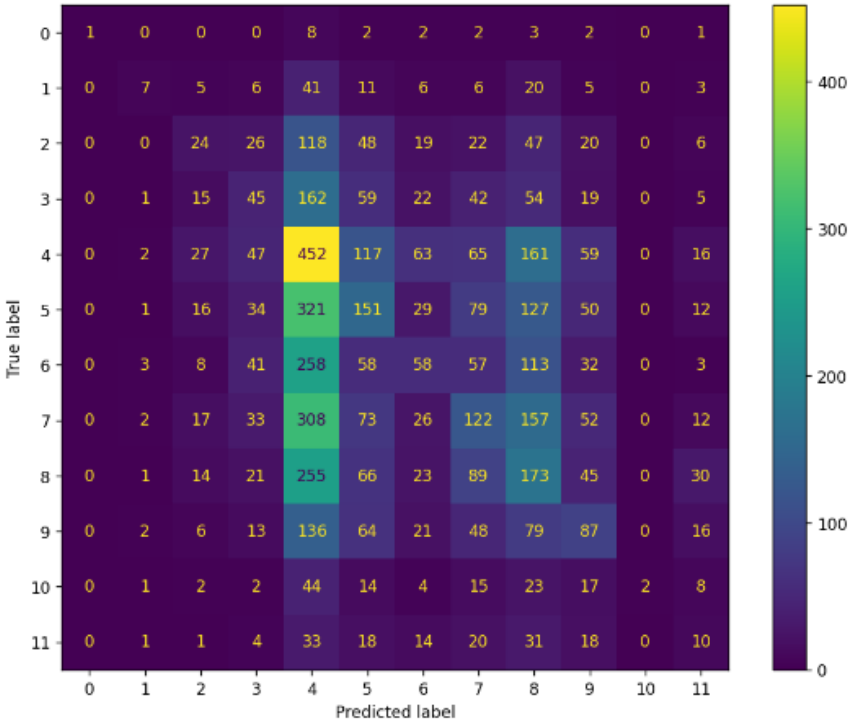

Figure 12. Predição salarial com somente fatores sociais

Em comparação aos modelos anteriores, o modelo com uma combinação entre os dois tipos de *features* conseguiu uma classificação melhor e mais distribuída entre as 12 classes. Mais especificamente, foi criado um modelo usando as informações CBO, tempo na empresa, nível de escolaridade, carga horária semanal, sexo, raça e idade com filtro para a área geral das “Ciências e Artes” do CBO. Os resultados obtidos podem ser observados a seguir.

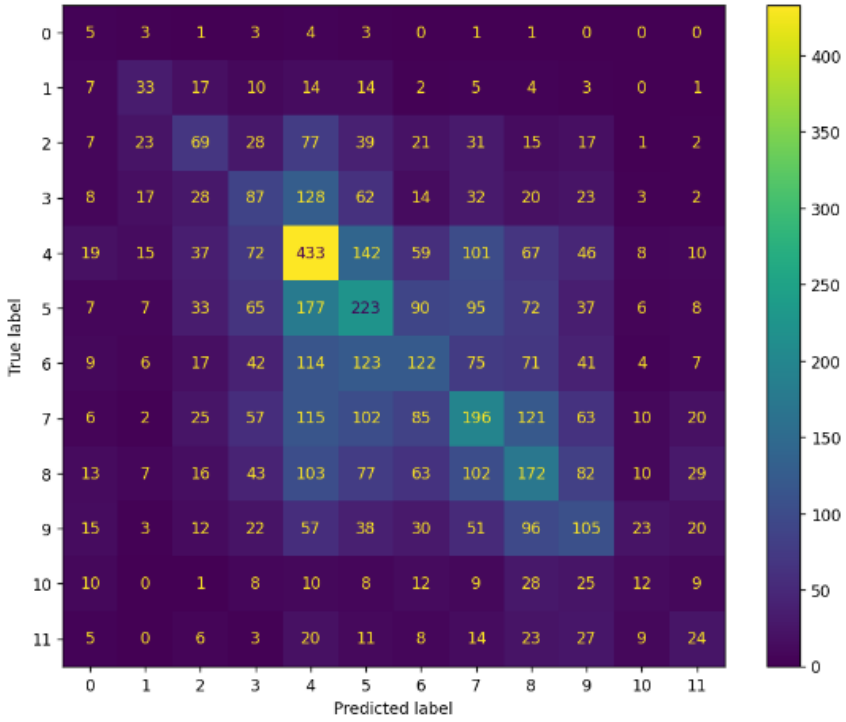

Figure 13. Predição salarial com fatores mistos

Com essas análises, é necessário reforçar que a precisão geral do modelo não é a maior preocupação do estudo de caso. O principal objetivo é a análise de discriminação salarial a partir da proposta do modelo avaliando como o agrupamento dos dados é feito a partir de cada combinação de *features*. Com esses experimentos, é possível afirmar que há um fator de peso consideravelmente alto na faixa salarial das pessoas no que diz respeito aos fatores sociais, dado que o agrupamento de dados não é feito de maneira padronizada quando são usados fatores objetivos; mas com sociais, sim. No primeiro teste com os dados da RAIS, é possível concluir que os fatores objetivos usados no modelo não são capazes de agrupar os dados de modo adequado. No segundo teste, somente com os fatores sociais, é notável o aumento da capacidade de agrupamento e padronização dos dados ignorando-se por completo os fatores objetivos do primeiro teste, mostrando que os fatores sociais possuem, de fato, grande impacto nas faixas salariais das pessoas, sendo possível classificar pessoas com base em "defeitos" melhor do que com base em qualificação profissional. No terceiro e último teste, com uma combinação de features dos dois anteriores, a capacidade de diferenciar as classes melhora em relação ao segundo teste, mostrando que os fatores objetivos possuem sim um nível de relevância na faixa salarial da pessoa, porém, os sociais são mais definitivos para a resposta do que os objetivos.

Além disso, é também possível subir a precisão da predição para cerca de 76% com esse algoritmo se a taxa de dados com *label* for aumentada para 80%, desvio-padrão de 0,5 e 15 iterações para convergência. Porém, dado que esse modelo não possui um método de predição sem envolver o treino, essa aplicação se torna inviável em um cenário real. A matriz de confusão da aplicação do modelo com as mesmas *features* usadas na última figura pode ser vista a seguir.

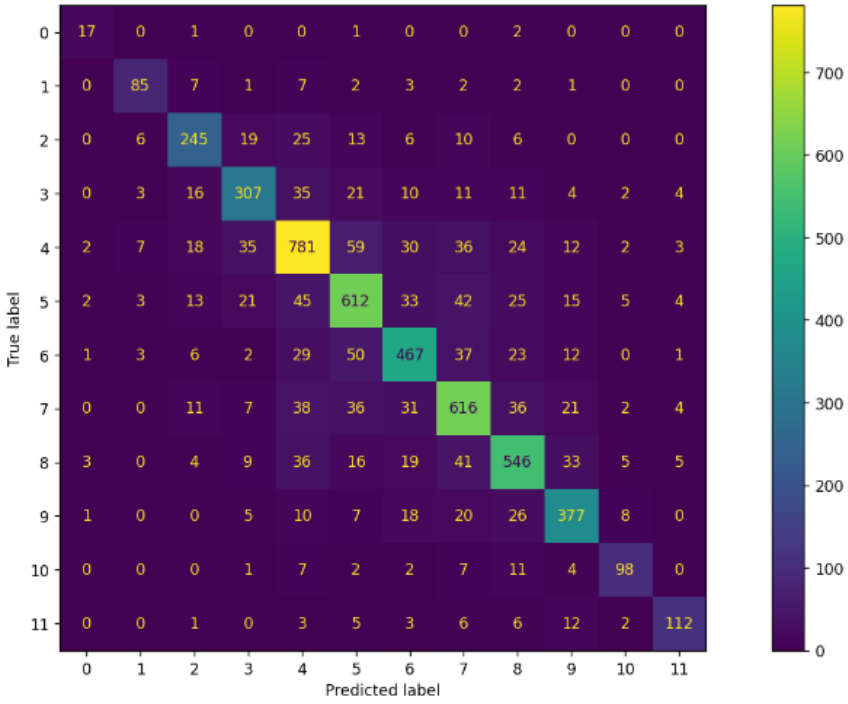

Figure 14. Predição salarial com convergência e grupo rotulado maior

Essa aplicação, apesar da alta precisão para 12 classes, não possui um resultado de alta confiança pois: esse modelo precisa de dados de treino juntamente com os dados de teste, ou seja, com a taxa de dados com *label* igual a 80%, seria necessária a resposta antecipada de 80% dos dados que se deseja encontrar a resposta, sendo que isso não é muito lógico; e também há uma possibilidade da alta precisão ocorrer por causa de uma lembrança das respostas de uma grande parte dos dados. Por isso, apesar de ser possível obter uma alta precisão na predição com esse modelo, os resultados não são completamente confiáveis nesse aspecto. Por essa razão, a análise dos resultados fica limitada à avaliação de viés do modelo através dos dados da RAIS.

4. Conclusão

A elaboração da biblioteca de grafos não é a principal proposta desse projeto, porém, essa implementação possui as qualidades de ser facilmente utilizável em qualquer contexto, com bom controle de erros por parte do programador, além de flexibilidade com forma de uso e possível expansão para uso do grafo resultado em aplicações externas como GEPHI, com parsers da estrutura gerada para arquivos CSV, para grafos mais simples, e gdf, para grafos mais complexos. Apesar disso, essa implementação se mostra consideravelmente pesada em comparação a possibilidades mais simples, como o tratamento de uma lista de adjacência como sendo uma "lista de nós que apontam para uma lista de nós", por exemplo. Em geral, a matriz de adjacência tende a ser de uso mais adequado, por ser mais leve e mais rápida, por usar ferramentas associadas ao numpy sempre que possível, em comparação à lista de adjacência, que não usa nada além de listas nativas em *Python*.

O estudo de caso da análise de discriminação e viés social com a RAIS aponta que, baseando-se nesses dados e nos resultados da aplicação da Inferência Transutiva: fatores sociais possuem considerável impacto na faixa salarial da pessoa; fatores objetivos

são relevantes para a predição salarial, porém, como "segundo plano" aos sociais, o que significa que, sozinhos, não possuem muito uso para o preditor; e que fatores sociais, com os dados da RAIS, são mais relevantes para a determinação da faixa salarial da pessoa do que fatores objetivos sozinhos. Além disso, com os testes de hipótese também apontando que há discriminação social nos dados, os resultados dos experimentos e análises combinados apontam que há sinais de discriminação social na distribuição salarial das pessoas que trabalham no Brasil.

Outro *insight* relevante em relação ao estudo de caso é sobre como a discriminação social é propagada para o modelo. Na aplicação que usa somente *features* sociais, é importante notar que o agrupamento aproximado que o modelo faz com base em somente fatores sociais possui uma precisão consideravelmente alta dentro das capacidades dessa base de dados. Isso mostra que fatores sociais, quando não tratados de modo a remover o preconceito existente nos dados (apontado através de diversos métodos estatísticos), propagam sim a discriminação para o modelo preditor, tornando-o preconceituoso, igual seus dados de base.

References

Coutinho, F. (2018). Learning with local and global consistency algorithm. Disponível em: `https://github.com/provezano/lgc`. Acesso em 22 de junho de 2023.

Zhou, D., Bousquet, O., Lal, T., Weston, J., and Schölkopf, B. (2003). Learning with local and global consistency. *Advances in neural information processing systems*, 16.