Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa
Departamento Acadêmico de Informática
Curso Superior de Bacharelado em Ciência da Computação
Programa de Pós-Graduação em Ciência da Computação

ALUNO(S): Gustavo P. Ferraz e Vinicius A. Oliveira

1. INTRODUÇÃO

As bases de dados propostas pela orientadora deste trabalho consistem em amostras de diversos pacientes portadores de câncer, as bases são divididas pelo tipo, ou, lugar onde a doença se localiza no paciente, para a realização deste experimento, acabamos optando por escolher três bases, sendo elas sobre câncer pancreático, câncer de garganta, e leucemia. Dentro das bases de dados, estão, além dos milhares de genes que "codificam" a doença, temos o tipo, por exemplo, na base sobre leucemia, o campo "type" pode variar entre "CLL", ou em português, LLC, e "normal _B_ cell", ou, célula B normal, que caracteriza outra variante da doença. Diante disso, nosso experimento se resumiu a treinar com diferentes algoritmos de aprendizagem de máquina, inteligências artificiais para, a partir dos genes da base, conseguir prever o tipo de câncer.

2. ALGORITMOS DE APRENDIZAGEM DE MÁQUINA

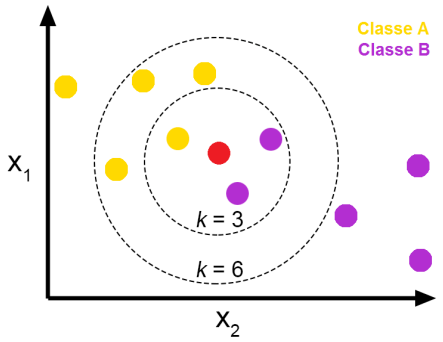
Neste experimento, fizemos uso de 4 algoritmos de aprendizagem de máquina, sendo eles o algoritmo KNN, a Árvore de Decisão, a rede neural MLP e a Regressão Linear.

2.1. Algoritmo KNN

O algoritmo KNN ou K-nearest neighbours, consiste de um método de aprendizado supervisionado que se baseia na ideia de proximidade, ou seja, elementos semelhantes estão próximos uns dos outros, isso é calculado no sentido literal, geralmente usa-se a distância Euclidiana.

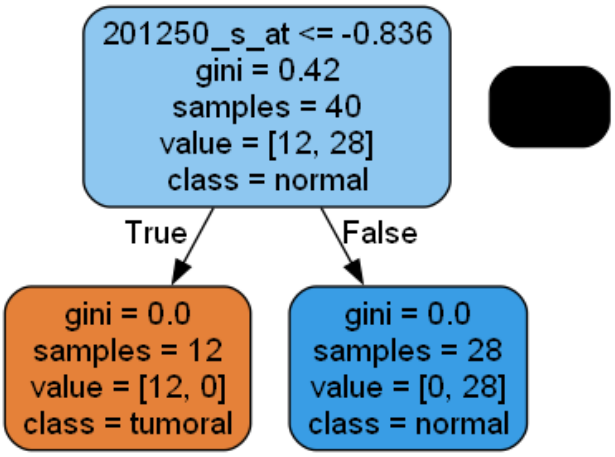
$$d(Euc) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Apesar de estarmos familiarizados com a distância Euclidiana entre pontos bi ou até mesmo tridimensionais, o algoritmo KNN tem a capacidade de lidar com múltiplas dimensões. Durante a programação do algoritmo, deve ser definido também o fator K, esse número determina quantos vizinhos o algoritmo usará como parâmetro, se o usuário colocar 3, o algoritmo considerará apenas os 3 vizinhos mais próximos, e então, verá qual é o rótulo mais comum entre eles.



2.2. Árvore de Decisão

A Árvore de Decisão opera da seguinte maneira, o algoritmo escolhe o atributo na base que melhor divide os elementos, ele faz isso geralmente pelo índice de Gini, que é um método matemático criado para medir a concentração de um atributo em um determinado grupo, por exemplo:



Em nosso experimento, por exemplo, foi o gene “201250_s_at”, nas folhas, o índice está 0.0 pois não há variação, o que indica que todos os elementos do grupo são da mesma classe.

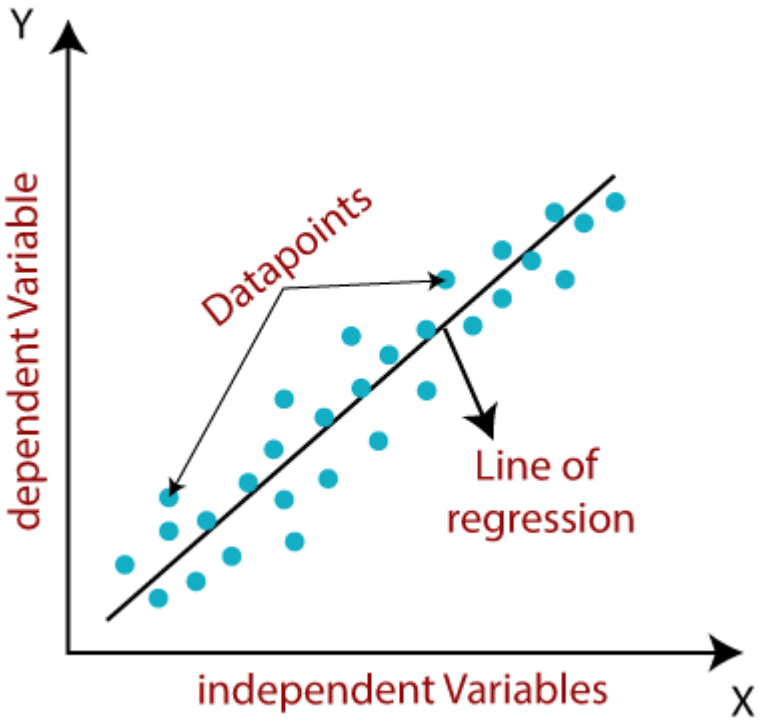
Na representação visual da árvore, cada ramificação de um nó indica um valor que ele pode assumir, esse processo é recursivo para cada ramificação, usando outros atributos para criar mais nós e ramos até que determinadas condições de parada sejam alcançadas.

A condição de parada de uma árvore de decisão pode variar, em nosso caso, consideramos a condição para quando não há nenhum novo ganho de informação a ser obtido, ou seja, o índice é zero, porém, em outras situações, é comum determinar uma profundidade máxima da árvore, para que a execução não se estenda, ou até mesmo quando todos os elementos possuem a uma mesma classe. Após o treinamento, quando um novo dado é apresentado, ele percorre a árvore até atingir uma folha.

2.3. Algoritmo de Regressão Linear

A regressão linear é um algoritmo de aprendizado de máquina que se enquadra na categoria de modelos supervisionados, sendo amplamente utilizado para análise e previsão de relações lineares entre variáveis. O

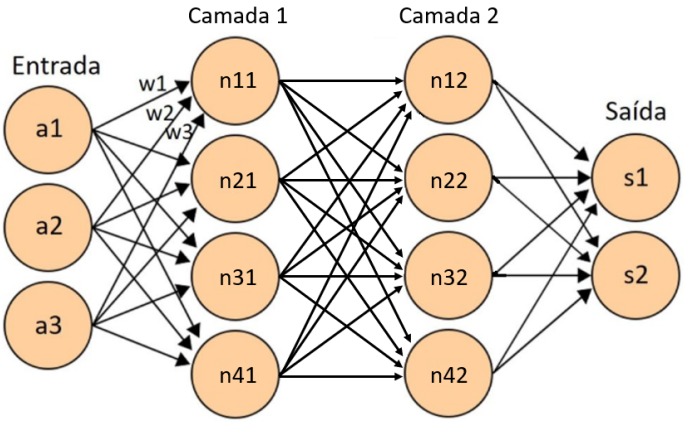
objetivo principal é encontrar a melhor linha que minimize a soma dos quadrados das diferenças entre as previsões do modelo e os valores reais observados. Em outras palavras, a regressão linear procura estabelecer uma equação linear que represente a relação entre a variável de entrada e a variável de saída. A equação resultante pode ser usada para fazer previsões sobre valores futuros com base nos dados de treinamento.



O processo de treinamento da regressão linear envolve ajustar os parâmetros da equação linear usando técnicas como o método dos mínimos quadrados. Essencialmente, o algoritmo busca encontrar os coeficientes que minimizam a soma dos erros ao quadrado. Uma vez treinado, o modelo pode ser usado para prever novos resultados com base em novas entradas. A simplicidade e interpretabilidade da regressão linear a tornam uma escolha comum em situações em que a relação entre variáveis pode ser aproximada por uma linha reta.

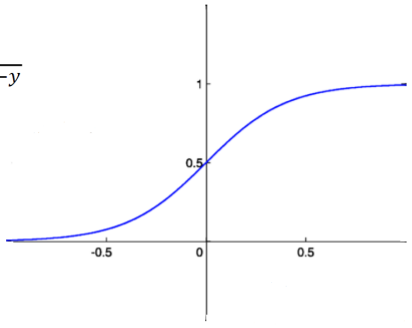
2.4. Rede neural artificial (Multi Layer Perceptron - MLP)

Uma MLP, ou Multi Layer Perceptron, é como o próprio nome diz, um perceptron multicamadas, sendo elas, a camada de entrada, a camada de saída e, entre elas, as camadas ocultas.



Em uma MLP, é possível ter N camadas ocultas, tudo depende da complexidade do problema que se quer resolver. O algoritmo funciona a partir de pesos e vieses, além de funções de ativação, inicialmente, na camada de entrada, a rede recebe os dados multiplicados pelo peso, esse valor então passa por uma função de ativação e, caso o resultado da função de ativação seja maior ou menor que o viés, o dado passa ou não para a próxima camada. Funções de ativação são funções matemáticas, geralmente não lineares na MLP, para permitir que a rede aprenda e represente relações complexas entre os dados, entre as funções de ativação mais comuns, temos a sigmóide,

$$p = \frac{1}{1 + e^{-y}}$$



e a ReLU, funções não lineares.

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

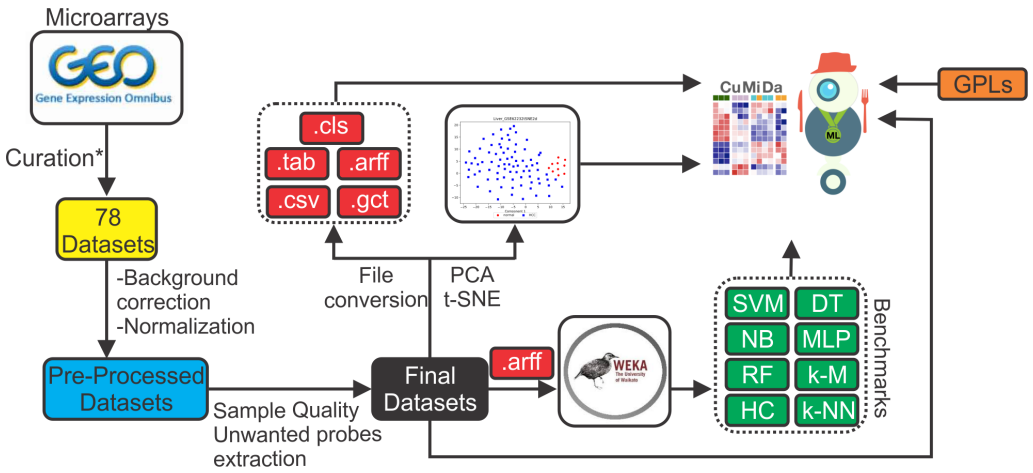
Depois de passar por uma função de ativação, caso uma entrada passe, prossegue até a próxima camada, e assim até atingir a camada de saída, isso se chama “forward propagation” ou, propagação direta, e é um dos conceitos da MLP. Caso atinja a camada de saída, os pesos e vieses permanecem os mesmos, porém, caso não, por meio de “backpropagation” ou, retropropagação, o algoritmo volta para alterá-los, e assim, treinar a IA.

Em alguns casos, podem ocorrer alguns problemas, como o Overfitting, ou, sobreajuste, isso ocorre quando a rede é complexa demais e fica então treinada demais e performa mal com novos dados, mesmo que seja boa em dados de treinamento. Ademais, pode ocorrer também o Underfitting, ou, sob ajuste, esse, o contrário, ocorre quando o modelo é muito simples e não consegue aprender com a base, geralmente performa mal tanto em dados de treino quanto em dados de teste.

3. METODOLOGIA

3.1 Base de Dados

Para o decorrer do trabalho, utilizamos a base de dados da SBCB Lab (Structural Bioinformatics and Computational Biology Lab) chamada CuMiDa (Curated Microarray Database). Esta base de dados possui armazenada 78 arranjos de tipos diferentes de câncer. Para cada arranjo, há um processo complexo de validação das informações para que os dados sejam precisos.



Nesse processo, os microarrays são obtidos e avaliados. após as avaliações, há a obtenção dos 78 datasets. Após os datasets serem definidos, de forma manual, todos eles são avaliados por especialistas e normalizados para serem classificados como datasets pré-processados. Com isso, uma última etapa é feita para remoção de dados não desejados, assim, tendo a versão final do dataset para publicação.

Para exemplificar, segue a ilustração de um dos 78 arrays disponíveis na base de dados:

TYPE	GSE	GPL PLATFORM	SAMPLES	GENES	CLASSES	
Pancreatic	16515	570	51	54676	2	Download

ZEROR	SVM	MLP	DT	NB	RF	HC	KNN	K-MEANS
0.71	0.86	0.78	0.78	0.84	0.82	0.69	0.76	0.76

Deste array, utilizamos as informações “Type”, “Samples”, “Genes” e “Classes”, que são respectivamente, o tipo do câncer, o número de amostras que resulta no número de entradas, os genes que são a quantidade de informações por amostra e a classe que representa o número de variações que podem haver ao final do processo.

3.2 Pré-Processamento

Inicialmente determinamos como “target” a coluna “type”, portanto, todas as outras são consideradas “features”:

```
features = data.drop('type', axis=1)
target = data['type']
```

Nesse caso, o parâmetro “axis=1” indica que a operação deve ser executada nas colunas, e não nas linhas. A seguir, normalizamos os dados:

```
scaler = StandardScaler()
features_normalized = scaler.fit_transform(features)
```

```
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
```

Na primeira imagem, utilizamos o “StandardScaler”, que padroniza os dados de uma maneira que a média seja 0 e o desvio padrão 1, além disso, exploramos o uso do “MinMaxScaler”, esse, por vez, pega o valor máximo e considera como 1, o mínimo considera como 0, e padroniza todos os valores a partir disso.

Por fim, separamos os dados em dados de treino e dados de teste, utilizamos a proporção de 20% teste e 80% treino no KNN e na Árvore de Decisão, e 30% teste e 70% treino na MLP e Regressão Linear, a fim de obter melhores resultados, como foi observado.

```
X_train, X_test, y_train, y_test = train_test_split(X_normalized, y, test_size=0.3, random_state=42)
```

Na imagem, utilizamos a função "train_test_split()" da biblioteca SciKit Learn, a partir dos parâmetros "X_normalized" e y, a base é passada para a função, "test_size-0.3" indica que 30% da base deverá ser o tamanho do teste, por fim, "random_state=42" é a "seed" para um gerador de números aleatórios dentro da função, fixamos esse valor para que em todas as execuções as inicializações sejam as mesmas, assim, produzindo resultados mais consistentes, a escolha do número 42 não tem nenhum motivo.

3.3 Execução do Algoritmo

Para a execução do experimento, foi utilizado o editor de código VS Code, toda a programação foi feita utilizando a linguagem Python 3.11, com ajuda das bibliotecas Pandas e NumPy, para o tratamento dos dados, MatplotLib, para as representações visuais, Pydotplus e por fim a SciKit Learn, para tudo realizado ao aprendizado de máquina.

3.3.1 KNN

```
knn = KNeighborsClassifier(n_neighbors=3)

knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
```

Para o KNN, iniciamos instanciando a classe da SciKit Learn com o K sendo 3 ("n_neighbors"), seguindo na função "knn.fit(X_train, y_train)", o algoritmo vai calcular pontos multidimensionais para todas as amostras da base e calcular as distâncias entre eles, a fim de encontrar os 3 vizinhos mais próximos para ver qual é a classe mais comum, por meio de um array. Por fim, a função "knn.predict(X_test)" utiliza o que foi aprendido da função anterior para prever nos dados separados para teste.

3.3.2 Árvore de Decisão

```
model = DecisionTreeClassifier()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)
```

Novamente, inicia-se instanciando a classe da biblioteca SciKit Learn, em seguida, a função "model.fit(X_train, y_train)" para treinar o algoritmo, desta vez, de uma maneira diferente, pois os algoritmos não funcionam da

mesma maneira, após o treinamento, o algoritmo seleciona o atributo que melhor separa as classes, como mostrado previamente, a partir desse atributo, os novos dados são direcionados para uma das classes possíveis.

3.3.3 Regressão Linear

```
model = LinearRegression()

model.fit(X_train , y_train)

y_pred = model.predict(X_test)
```

De início, a classe LinearRegression (Regressão Linear) é instanciada da biblioteca SciKit Learn (sklearn) que contém todo o método da regressão linear. Em seguida, o método fit é invocado do objeto model para que o treinamento seja iniciado. Ao final deste processo, a variável y_pred é declarada contendo o previsto em X_test,

para que o cálculo das métricas de avaliação do resultado possam ser feitas.

3.3.4 Rede neural artificial (MLP)

```
mlp = MLPClassifier(hidden_layer_sizes=(48, 48), max_iter=1000, learning_rate_init=0.05, activation="relu", random_state=1)

mlp.fit(X_train, y_train)

y_pred = mlp.predict(X_test)
```

Na classe MLPClassifier, utilizamos alguns parâmetros, são eles, “hidden_layer_sizes” é uma lista com valores numéricos, cada novo elemento significa uma nova camada oculta, no caso, duas camadas ocultas com 48 neurônios em cada uma, seguindo, “max_iter=1000”, aqui, limitamos o número máximo de iterações apenas para evitar que as execuções do algoritmo durem tempo demais. A “learning_rate_init” é a taxa de aprendizado, basicamente, ela é o ajuste a ser feito nos pesos e nos vieses cada vez que a rede errar, “activation=”relu”” é a função de ativação, que pode ser definida como:

```
def relu(self, x):
    return np.maximum(0, x)
```

Adiante, "random_state" é apenas a semente para o gerador aleatório dentro da função, não é obrigatório e a escolha do número 1 não tem nenhum motivo específico.

Seguindo, a função "fit" treina a rede a partir da base e da função de ativação (ReLU), aqui, a base de treino (X_train) percorre a rede enquanto a cada vez que um neurônio não é ativado, a partir da taxa de aprendizado os pesos e vieses são alterados. Enfim, na última linha a função "predict" atua na base de testes e tenta prever os resultados ao passar pela rede e atingir a camada de saída.

4. RESULTADOS OBTIDOS

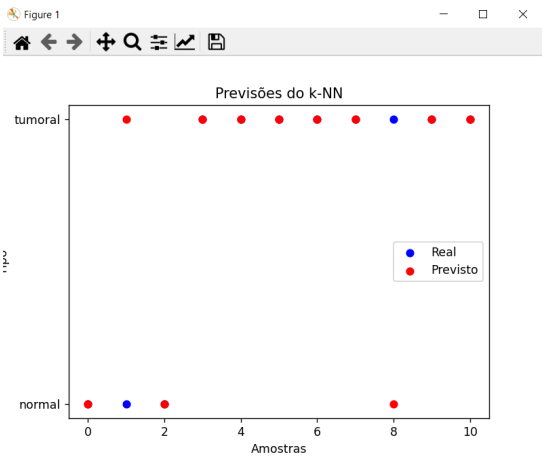
4.1 KNN

Para a visualização dos resultados obtidos, utilizamos o método accuracy_score, que avalia a igualdade entre dois parâmetros.

```
accuracy = accuracy_score(y_test, y_pred)
print(f"Precisão do modelo: {accuracy:.2f}")

plt.scatter(range(len(y_test)), y_test, color='blue', label='Real')
plt.scatter(range(len(y_pred)), y_pred, color='red', label='Previsto')
plt.xlabel('Amostras')
plt.ylabel('Tipo')
plt.title('Previsões do k-NN')
plt.legend()
plt.show()
```

Após a precisão ser calculada ela é representada em forma de output textual e posteriormente em formato de gráfico.

```
Acurácia do modelo: 0.82
```



4.2 Árvore de Decisão

```
accuracy = metrics.accuracy_score(y_test, y_pred)

dot_data = export_graphviz(model, out_file=None, feature_names=features.columns, class_names=target.unique(), filled=True, rounded=True)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
graph.write_png('tree.png')
accuracy = metrics.accuracy_score(y_test, y_pred)
print("Acurácia:", accuracy)
```

A fim de saber a precisão dos resultados, foi utilizada a função “accuracy_score”, a função recebe o “y_test” e “y_pred”, onde um é os dados reais e o outro os dados previstos, respectivamente.

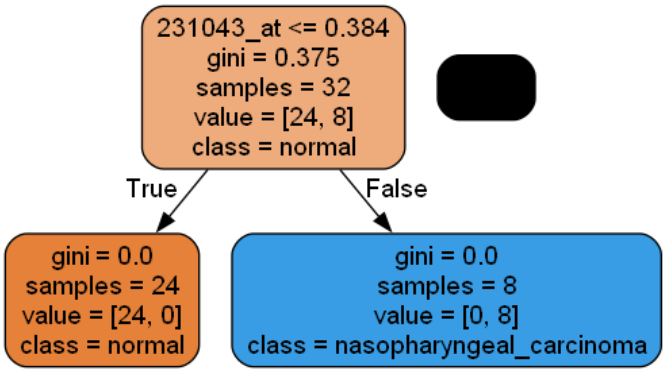
```
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

A função compara a quantidade de elementos em comum nas duas listas e retorna um valor de 0 a 1 que pode ser interpretado como a porcentagem.

```
Acurácia: 0.875
```

```
dot_data = export_graphviz(model, out_file=None, feature_names=features.columns, class_names=target.unique(), filled=True, rounded=True)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
graph.write_png('tree.png')
```

Por fim, geramos uma imagem de uma árvore para representação visual dos resultados.



A raiz mostra qual foi o gene determinante, o índice de Gini geral, o número de amostras, a distribuição dessas amostras e a classe dominante.

4.3 Regressão Linear

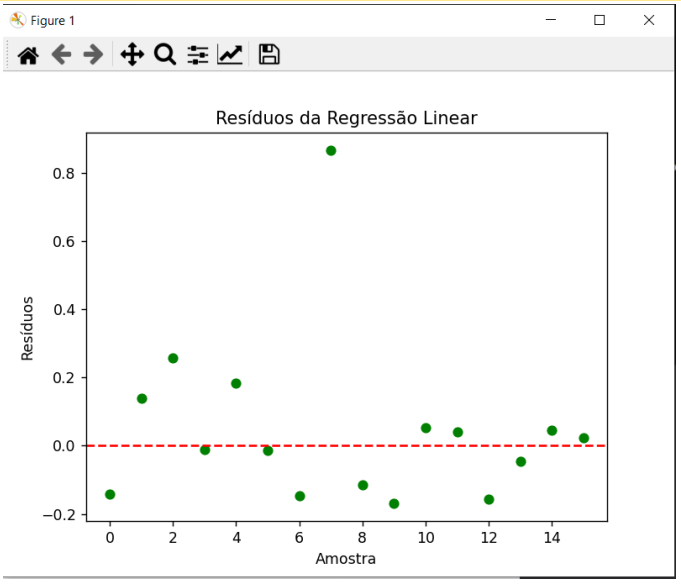
```python
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
print("Erro Quadrático Médio (MSE):", mse)
print("Erro Absoluto Médio (MAE):", mae)

residuos = y_test - y_pred
plt.scatter(range(len(residuos)), residuos, color='green')
plt.axhline(y=0, color='red', linestyle='--')
plt.xlabel('Amostra')
plt.ylabel('Resíduos')
plt.title('Resíduos da Regressão Linear')
plt.show()
```

Para o algoritmo de regressão linear, foram utilizados outros métodos de avaliação, o erro quadrático médio e o erro absoluto médio. Com isso, obtemos a diferença entre os valores observados e os valores de previsão, quanto menor, mais próximo dos valores esperados via EQM. Já o EAM retorna o número absoluto das somas das diferenças.

$$EQM = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$EAM = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- n representa o número de observações.
- y_i são os valores reais.
- $\hat{y}_i$ são os valores preditos pelo modelo.



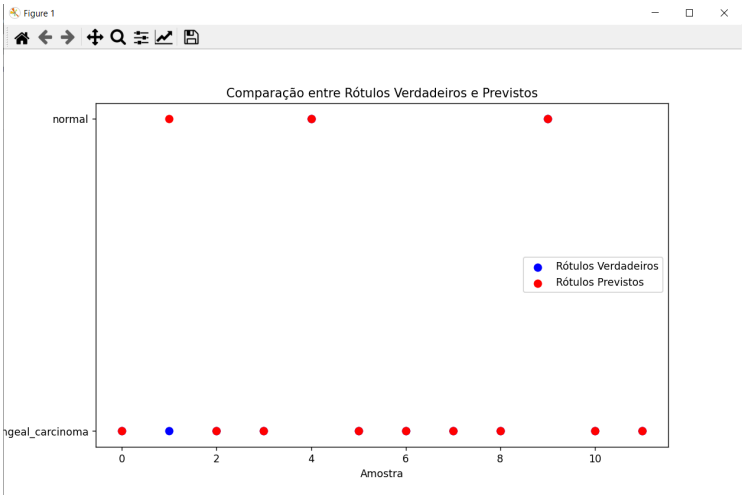

```
Erro Quadrático Médio (MSE): 0.06167688434267136
Erro Absoluto Médio (MAE): 0.15078077389570035
```

4.4 MLP

```python
precisao = mlp.score(X_test, y_test)
print("Precisão:", precisao)

plt.figure(figsize=(10, 6))

plt.scatter(range(len(y_test)), y_test, label='Rótulos Verdadeiros', color='blue', s=50)
plt.scatter(range(len(y_test)), y_pred, label='Rótulos Previstos', color='red', s=50)

plt.title("Comparação entre Rótulos Verdadeiros e Previstos")
plt.xlabel("Amostra")
plt.ylabel("Rótulo")
plt.legend()
plt.show()
```

Na rede neural, o cálculo da precisão não foi feito da mesma maneira que os anteriores, a função “score” prevê os rótulos, nota-se que a função “accuracy_score” recebe como parâmetros “y_pred” e “y_test”, ou seja, compara o previsto com o real, já a função usada, recebe “X_test” e “y_test”, então, antes de retornar a acurácia, a função utiliza a função “predict” para obter o resultado.

A seguir, adicionamos uma representação visual dos resultados.




```
Precisão: 0.9166666666666666
```

5. CONCLUSÃO

Diante dos testes efetuados, escolhendo uma única base para todos os algoritmos, o KNN performou relativamente bem e obteve 82% de precisão, no mesmo modelo a Árvore de Decisão obteve 90,09%, a Regressão Linear um erro quadrático médio de 0.1319, como o valor que estamos tentando prever é de 1, isso caracteriza uma precisão de aproximadamente 86%, e a MLP 81,5%. A partir desses dados, fica claro que a Árvore de Decisão, além de ser relativamente mais simples que os outros, principalmente a MLP, tem uma performance superior, além disso, a MLP não performou muito bem, possivelmente por causa do tamanho das camadas em relação ao tamanho da entrada, ou até mesmo, possivelmente pelo número de iterações ter sido limitado a 1000.

Apesar de obter resultados não tão esperados, esse experimento serviu para ilustrar que nem sempre o algoritmo mais complexo é o melhor para solucionar qualquer problema, uma vez que, além desse teste, a MLP performou abaixo em outras bases e a Árvore de Decisão teve o melhor desempenho.

6. REFERÊNCIAS

SBCB Lab - Structural Bioinformatics and Computational Biology Lab. Curated Microarray Database (CuMiDa). Porto Alegre: SBCB Lab, 2022; Disponível em: https://sbcb.inf.ufrgs.br/cumida

SCIKIT-LEARN DEVELOPERS. Scikit-Learn Site: Documentation, 2019. P 2160. Disponível em: <https://scikit-learn.org/0.21/_downloads/scikit-learn-docs.pdf>. Acesso em: 24 de nov. de 2023.

DidaticaTech: O que são Redes Neurais e Deep Learning? Disponível em: https://didatica.tech/introducao-a-redes-neurais-e-deep-learning/. Acesso em: 24 de nov. de 2023.

JavaTPoint: Linear Regression in Machine Learning. Disponível em: https://www.javatpoint.com/linear-regression-in-machine-learning. Acesso em: 24 de nov. de 2023.

Italo José. Medium: Knn (K-Nearest Neighbors) #1. Disponível em: https://medium.com/brasil-ai/knn-k-nearest-neighbors-1-e140c82e9c4e. Acesso em: 24 de nov. de 2023.