Universidade Federal do Rio Grande do Sul
Instituto de Informática

Aprendizado de Máquina
INF01017

TREINAMENTO E VALIDAÇÃO DE MODELOS PREDITIVOS COM APRENDIZADO DE MÁQUINA

Gabriel Lando 291399
Juliane da Rocha Alves 285681

Porto Alegre **11 de abril de 2022**

SUMÁRIO

Introdução **2**

Metodologia **2**

Dataset 2

Algoritmos de classificação escolhidos 6

Random Forest 6

KNN 7

Logistic Regression 8

Implementação K-Fold cross validation 8

Implementação das métricas 12

Resultados **14**

Considerações finais **18**

1. Introdução

Nosso trabalho tem como objetivo desenvolver um modelo baseado em Aprendizado de Máquina para prever o se uma música lançada entre 2010 e 2019 no serviço de *streaming* de músicas *Spotify* se tornou um *hit* ou não a partir de um conjunto de características das faixas. A linguagem de desenvolvimento escolhida para o trabalho foi Python, utilizou-se as bibliotecas do Scikit-Learn para os modelos, e utilizou-se Numpy e Pandas para a manipulação dos dados.

2. Metodologia

2.1. Dataset

O dataset utilizado para o trabalho foi retirado do repositório do Kaggle (https://www.kaggle.com/theoverman/the-spotify-hit-predictor-dataset). A Tabela 1 refere-se aos atributos do dataset e o que cada atributo significa, de acordo com o repositório do Kaggle.

Tabela 1: Atributos do dataset

Atributo	Valor
track	O nome da música
artist	O nome do artista da música
uri	O identificador da música
danceability	A dançabilidade descreve o quão adequada uma faixa é para dançar com base em uma combinação de elementos musicais, incluindo tempo, estabilidade do ritmo, força da batida e regularidade geral. Um valor de 0.0 é o menos dançável e 1.0 é o mais dançável.
energy	A energia é uma medida de 0.0 a 1.0 e representa uma medida perceptiva de intensidade e atividade. Normalmente, as faixas energéticas parecem rápidas, altas e barulhentas. Por exemplo, o death metal tem alta energia, enquanto um prelúdio de Bach tem uma pontuação baixa na escala. As características perceptivas que contribuem para este atributo incluem faixa dinâmica, intensidade percebida, timbre, onset rate e entropia geral.
key	A chave geral estimada da faixa. Os inteiros

	são mapeados para pitches usando a notação padrão de Classe de Pitch. Por exemplo. 0 = C, 1 = C?/D?, 2 = D, e assim por diante. Se nenhuma chave foi detectada, o valor é -1.
loudness	O volume geral de uma faixa em decibéis (dB). Os valores de volume são calculados em média em toda a faixa e são úteis para comparar o volume relativo das faixas. A sonoridade é a qualidade de um som que é o principal correlato psicológico da força física (amplitude). Os valores típicos variam entre -60 e 0 db.
mode	Mode indica a modalidade (maior ou menor) de uma faixa, o tipo de escala da qual seu conteúdo melódico é derivado. Maior é representado por 1 e menor é 0.
speechiness	Speechiness detecta a presença de palavras faladas em uma faixa. Quanto mais exclusivamente falada a gravação (por exemplo, talk show, audiolivro, poesia), mais próximo de 1.0 o valor do atributo. Valores acima de 0.66 descrevem faixas que provavelmente são feitas inteiramente de palavras faladas. Valores entre 0.33 e 0.66 descrevem faixas que podem conter música e fala, seja em seções ou em camadas, incluindo casos como música rap. Os valores abaixo de 0.33 provavelmente representam músicas e outras faixas que não são de fala.
acousticness	A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic
instrumentalness	Prevê se uma faixa não contém vocais. Os sons “Ooh” e “aah” são tratados como instrumentais neste contexto. Faixas de rap ou de palavras faladas são claramente “vocais”. Quanto mais próximo o valor de instrumentalidade estiver de 1.0, maior a probabilidade de a faixa não conter conteúdo vocal. Valores acima de 0.5 destinam-se a representar faixas instrumentais, mas a confiança é maior à medida que o valor se aproxima de 1.0.

liveness	Detecta a presença de uma audiência na gravação. Valores mais altos de vivacidade representam uma probabilidade maior de que a faixa tenha sido executada ao vivo. Um valor acima de 0.8 fornece uma forte probabilidade de que a faixa esteja ativa.
valence	Uma medida de 0.0 a 1.0 que descreve a positividade musical transmitida por uma faixa. Faixas com alta valência soam mais positivas (por exemplo, feliz, alegre, eufórica), enquanto faixas com baixa valência soam mais negativas (por exemplo, triste, deprimida, irritada).
tempo	O ritmo geral estimado de uma faixa em batidas por minuto (BPM). Na terminologia musical, tempo é a velocidade ou ritmo de uma determinada parte e deriva diretamente da duração média da batida.
duration_ms	A duração da música em milisegundos.
time_signature	Uma assinatura de tempo geral estimada de uma faixa. A fórmula de compasso (medidor) é uma convenção de notação para especificar quantas batidas existem em cada compasso (ou medida).
chorus_hit	Esta é a melhor estimativa do autor de quando o refrão começaria para a faixa. É o timestamp do início da terceira seção do track (em milissegundos). Esse recurso foi extraído dos dados recebidos pela chamada da API para Análise de áudio dessa faixa específica.
sections	O número de seções que a faixa específica possui. Esse recurso foi extraído dos dados recebidos pela chamada da API para Análise de áudio dessa faixa específica.
target	É a classificação da música. Pode ser '0' ou '1'. '1' implica que esta música apareceu na lista semanal (emitido por Billboards) de faixas Hot-100 naquela década pelo menos uma vez e, portanto, é um 'hit'. '0' Implica que a faixa é um 'flop'.

Para a execução dos algoritmos de classificação, **não utilizou-se os atributos** *track, artist* e *uri*, por não conterem informações significativas para o problema em

questão. Para a execução da Random Forest, nenhum pré-processamento dos dados foi necessária, porém para executar o KNN e o Logistic Regression, foi aplicada uma função chamada *MinMaxScaler* da biblioteca do Scikit-Learn para normalizar os dados entre 0 e 1, por conta de como ambos os algoritmos funcionam.

Antes de aplicar os algoritmos, uma análise dos dados foi realizada. Primeiro verificou-se se existiam dados nulos no dataset. Como pode-se observar na Figura 1, todos os atributos estão completos, sem dados faltando. Além disso, o dataset apresenta 6398 amostras, sendo 3199 da classe 0 e 3199 da classe 1. Sendo assim, as classes são balanceadas.

```
Data columns (total 19 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   track             6398 non-null   object
 1   artist            6398 non-null   object
 2   uri               6398 non-null   object
 3   danceability      6398 non-null   float64
 4   energy            6398 non-null   float64
 5   key               6398 non-null   int64
 6   loudness          6398 non-null   float64
 7   mode              6398 non-null   int64
 8   speechiness       6398 non-null   float64
 9   acousticness      6398 non-null   float64
 10  instrumentalness  6398 non-null   float64
 11  liveness          6398 non-null   float64
 12  valence           6398 non-null   float64
 13  tempo             6398 non-null   float64
 14  duration_ms       6398 non-null   int64
 15  time_signature    6398 non-null   int64
 16  chorus_hit        6398 non-null   float64
 17  sections          6398 non-null   int64
 18  target            6398 non-null   int64
dtypes: float64(10), int64(6), object(3)
memory usage: 949.8+ KB
```

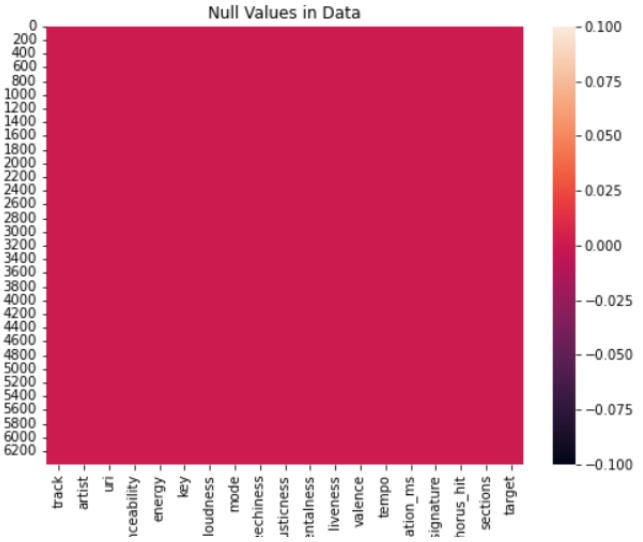

Figura 1: Informação de dados nulos

Uma análise da distribuição de cada um dos atributos e a relação de cada atributo dois a dois também foi realizada, e se encontra no arquivo *data_analysis.ipynb* (a imagem é muito grande e não coube no relatório). Para termos uma ideia se os algoritmos teriam alguma dificuldade para aprender a separar as classes, executamos um PCA com 2 Componentes Principais e com 3 Componentes Principais nos dados, utilizando a biblioteca do Scikit-Learn. Como podemos ver nas Figuras 2 e 3, as classes 0 e 1 são muito semelhantes entre si e se sobrepõem no plano, o que pode dificultar para os algoritmos ao tentar encontrar uma fronteira de decisão entre ambas as classes.

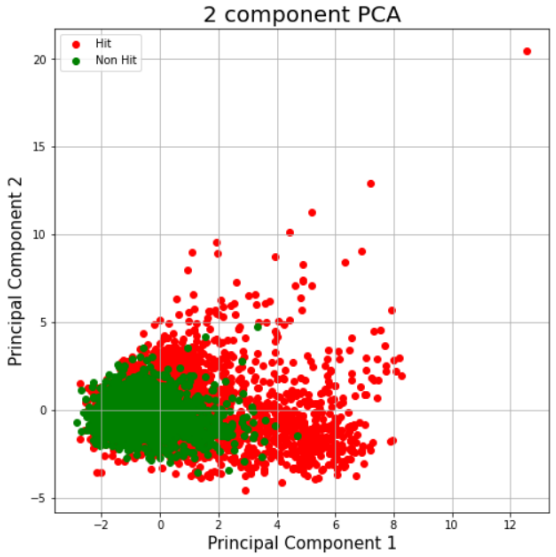

Figura 2: PCA com 2 Componentes Principais

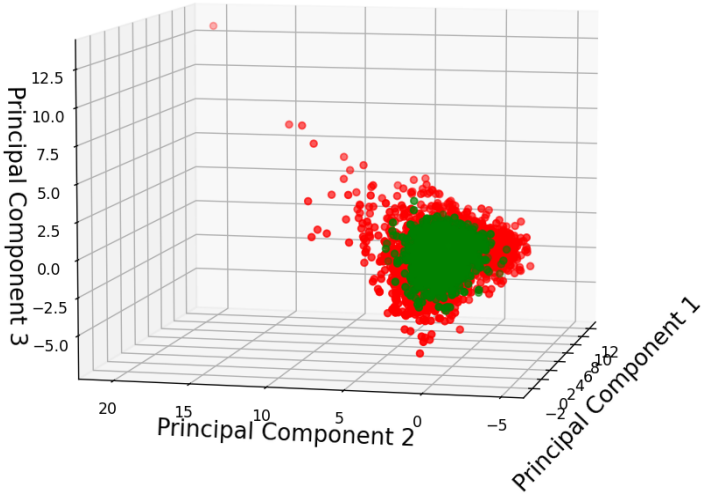

Figura 3: PCA com 3 Componentes Principais

2.2. Algoritmos de classificação escolhidos

Para a classificação dos hits, escolheu-se 3 algoritmos de classificação vistos em aula: Random Forest, K-Nearest Neighbors e Logistic Regression. As subseções a seguir explicam o motivo de cada escolha.

2.2.1. Random Forest

Random Forest foi escolhido pelo grupo como um dos classificadores por ser um essemble de Decision Trees, o que por si só resultaria em uma decisão de classifição melhor do que utilizar uma única Decision Tree, visto que agregaria o resultado de inúmeras árvores. Além disso, visto que o problema escolhido pelo grupo

não é um problema complexo, na hora de escolher os algoritmos, acreditou-se que uma Random Forest poderia lidar bem com o problema.

Para a aplicação do algoritmo, utilizou-se a biblioteca do Scikit-Learn, e os hiperparâmetros foram escolhidos através de inúmeras execuções, a fim de minimizar os falsos positivos. Iremos descrever melhor na seção de Resultados.

```
spotify_df = pd.read_csv("dataset-of-10s.csv")
features = ["danceability", "energy", "key", "loudness", "mode", "speechiness",
"acousticness", "instrumentalness", "liveness", "valence", "tempo",
"duration_ms", "time_signature", "chorus_hit", "sections"]

X = spotify_df.loc[:, features].values
y = spotify_df.loc[:, ["target"]].values

print("************************************")
print("****** Running a Random Forest ******")
print("************************************")
dtree = DecisionTreeClassifier(max_depth=10,
                               max_features="sqrt",
                               min_samples_leaf=10,
                               min_samples_split=25,
                               splitter="best",
                               criterion="gini")
```

2.2.2. KNN

A escolha do *KNN* se deve pelo fato de ter se observado nos dados, especialmente após a aplicação do PCA, de que os dados que foram classificados como *hits* estavam bem próximos um do outro no plano, assim como os dados classificados como não *hits*. Acreditou-se que pelo fato de os grupos de dados classificados como uma mesma classe estarem próximos um ao outro no plano, de que o algoritmo conseguiria aprender a classificar uma nova instância “olhando” apenas para os dados próximos.

Para a aplicação do algoritmo, utilizou-se a biblioteca do Scikit-Learn, e os hiperparâmetros foram escolhidos através de inúmeras execuções, a fim de minimizar os falsos positivos. Iremos descrever melhor na seção de Resultados. Antes da aplicação do *KNN*, executou-se a função do Scikit-Learn chamada *MinMaxScaler*, como mencionado na seção anterior.

```
# It is necessary to normalize the values for K-NN and Logistic Regression
scaler = MinMaxScaler()
scaler.fit(X)
X = scaler.transform(X)
```

```
print("**************************")
print("****** Running a KNN ******")
print("**************************")
knn = KNeighborsClassifier(n_neighbors=13)
```

2.2.3. *Logistic Regression*

A escolha do Logistic Regression se deve pelo fato de que, após visualizar os dados no plano com a aplicação do PCA, observou-se de que os dados classificados como *hits* e não *hits* se sobrepõem no plano, mesmo utilizando um PCA de 3 componentes. É muito difícil de encontrar uma linha de decisão visível que separe essas duas classes, então acreditou-se de que o Logistic Regression pudesse encontrar uma fronteira de decisão para a separação dos dados.

Para a aplicação do algoritmo, utilizou-se a biblioteca do Scikit-Learn, e os hiperparâmetros foram escolhidos através de inúmeras execuções, a fim de minimizar os falsos positivos. Iremos descrever melhor na seção de Resultados. Antes da aplicação do Logistic Regression, executou-se a função do Scikit-Learn chamada *MinMaxScaler*, como mencionado na seção anterior.

```
print("****************************************")
print("****** Running a Logistic Regression ******")
print("****************************************")
lr = LogisticRegression(tol=0.001, solver="newton-cg")
cv = CrossValidation(classifier=lr, k_folds=15, X=X, y=y)
cv.fit()
```

2.3. Implementação K-Fold cross validation

Para a implementação do *K-Fold cross validation* utilizou-se a explicação vista nas aulas assim como os slides como referência. Primeiramente o algortitmo verifica o número de instâncias total e o número de *folds*. Com essas informações, é calculado a quantidade de instancias que cada *fold* terá, arredondando-se o número para baixo. Dessa forma, o último *fold* a ser criado terá um pouco mais de instâncias, caso o número de *folds* ou o número de instâncias seja ímpar. Após esse cálculo, é verificado a proporção de cada classe. Para isso, utilizou-se uma função do *Numpy* que retorna os valores únicos encontrados em um determinado *array* e a frequência desses valores únicos. A proporção se dá pela fórmula:

$$(frequência\ da\ classe\ *\ número\ de\ instâncias)\ *\ 100$$

Os índices de cada instância de cada classe são armazenados em um dicionário, onde as chaves do dicionário são as classes e o valor armazenado em cada chave é um *array* que contém todos os índices das intâncias daquela determinada classe. Isso é feito de forma a facilitar a distribuição da escolha de cada instância de cada classe na hora de se criar cada *fold*, para que os *folds* possuam a mesma proporção de classes do *dataset* original.

Para a criação de cada *fold* verifica-se primeiro quantas instâncias de cada classe é necessário para manter a proporção. A fórmula para calcular o número de instâncias de cada classe se dá por:

$$(proporção\ da\ classe\ *\ \ tamanho\ do\ fold)/100$$

Assim como no cálculo do tamanho dos *folds*, arredonda-se esse valor para baixo. Para selecionar as instâncias para cada *fold*, escolhe-se o número de instâncias calculado pela fórmula acima de maneira randômica. Para isso, utilizou-se a função Random do próprio Pyhton, que retorna um valor aleatório de uma lista. A lista utilizada para se escolher aleatoriamente os valores são as listas armazenadas anteriormente com os índices das instâncias de cada classe. Dessa forma, por exemplo, se for preciso escolher 5 instâncias da classe 0, o *Random* irá retornar 5 índices aleatórios da lista de índices da classe 0. Os valores escolhidos são armazenados em uma outra lista, a fim de verificar se algum índice está sendo escolhido mais de uma vez. Caso isso ocorra, o *Random* irá escolher um ínicide qualquer da lista da classe até encontrar um que ainda não foi utilizado. Assim, cada instância estará presente somente uma vez em cada *fold*. A criação do último *fold* se dá pelas instâncias que não foram utilizadas em nunhum *fold* criado anteriormente.

Após finalizar a criação dos *folds*, um *fold* é escolhido para ser utilizado como teste, enquanto que os outros *folds* são utlizados para o treinamento. A cada iteração (*K* iterações), um *fold* diferente é escolhido para teste, garantindo assim que todos os *folds* são utilizados para o treinamento.

Ao fim de cada treinamento, é utilizado a classe *ConfusionMatrix* (criada pelo grupo) para calcular as métricas do modelo e imprimir a matriz de confusão. Quando todos os *folds* foram utilizados para o treinamento, é então calculado a média e o desvio padrão de todas as métricas de cada execução.

A classe *CrossValidation* recebe como parâmetro o classificador a ser utilizado (*classifier*), as instâncias a serem utilizadas (*X*) e o *array* de valores a serem preditos (*y*).

A seguir, temos o código utilizado para implementar a classe *CrossValidation*:

```
class CrossValidation:
    """
    This class implements the logic behind the Cross Validation. It will split in
K folds the X (values) and y (target)
```

```
    keeping the proportion between the classes in the folds.

    This class was implemented for Classification problems. It receives a Scikit
classifier and apply
    the cross validation method.
    """
    def __init__(self, classifier: BaseEstimator = None, k_folds: int = 5, X:
np.ndarray = None, y: np.ndarray = None) -> None:
        self.classifier = classifier
        self.k_folds = k_folds
        self.X = X
        self.y = y
        self.accuracy = []
        self.precision = []
        self.recall = []
        self.specificity = []
        self.f1_measure = []

    def fit(self) -> None:
        num_samples = len(self.y)
        folds_size = round(num_samples / self.k_folds)
        # Get the frequency of each class
        target, frequency = np.unique(self.y, return_counts=True)

        class_proportion = {}
        class_index = {}
        for ind in range(len(target)):
            # Calculates the classes proportion
            class_proportion[target[ind]] = (frequency[ind]/num_samples)*100
            # Gets all indexes for each class in the dataset
            class_index[target[ind]] = np.where(self.y == target[ind])[0]

        indexes_used = []
        X_folds = []
        y_folds = []
        for k in range(self.k_folds):
            X_fold = []
            y_fold = []
            print(f"Generating fold {k+1}")
            if k != self.k_folds-1:
                for key in class_proportion.keys():
                    # Calculates how many elements it needs to have in each fold
for each class, to keep the proportion
                    num_elements = round((class_proportion[key]*folds_size)/100)
                    # Gets random N (num_elements) elements from each class
                    ind = random.sample(class_index[key].tolist(), num_elements)

                    for i in ind:
                        # Verify if the index was already used
                        if i not in indexes_used:
```

```
                            X_fold.append(self.X[i].tolist())
                            y_fold.append(self.y[i][0])
                            indexes_used.append(i)
                        else:
                            # Calculates a new index if the index was already
used
                            new_i = i
                            while new_i in indexes_used:
                                new_i = random.sample(class_index[key].tolist(),
1)[0]

                            X_fold.append(self.X[new_i].tolist())
                            y_fold.append(self.y[new_i][0])
                            indexes_used.append(new_i)
                    # Append the fold
                    X_folds.append(X_fold)
                    y_folds.append(y_fold)
                else:
                    # The last fold will have the rest of the values that are not in
the previous folds
                    all_indexes = [ind for ind in range(num_samples)]
                    indexes_for_test = list(set(all_indexes) - set(indexes_used))
                    for ind in indexes_for_test:
                        X_fold.append(self.X[ind].tolist())
                        y_fold.append(self.y[ind][0])
                    # Append the last fold
                    X_folds.append(X_fold)
                    y_folds.append(y_fold)

            # Train the model K times (K = number of folds)
            for k in range(self.k_folds):
                X_train = []
                y_train = []
                X_test = []
                y_test = None

                # The K determines the fold to be used to test
                for values in X_folds[k]:
                    X_test.append(values)
                y_test = y_folds[k]

                print(f"Using the fold {k} for test")

                # The j determines the folds to be used to train. The fold used to
test is not used
                for j in range(self.k_folds):
                    if j != k:
                        print(f"Using the fold {j} for train")
                        for values in X_folds[j]:
                            X_train.append(values)
```

```
                    for y in y_folds[j]:
                        y_train.append(y)

            # Gets the shape of the training set
            shape = self._get_train_fold_shape(k, X_folds)

            # Reshape the train and test sets (n_samples, n_features)
            X_train = np.array(X_train).reshape((shape, self.X.shape[1]))
            y_train = np.array(y_train).reshape((shape,))

            X_test = np.array(X_test).reshape((self.X.shape[0] - shape,
self.X.shape[1]))
            y_test = np.array(y_test).reshape((self.X.shape[0] - shape,))

            self.classifier.fit(X_train, y_train)
            y_pred = self.classifier.predict(X_test)

            self.print_report(y_pred, y_test)

            print("******************************")

        self.print_final_report()

    def _get_train_fold_shape(self, k: int, X_folds: List) -> int:
        rows = 0
        for fold in range(self.k_folds):
            if fold != k:
                rows = rows + len(X_folds[fold])
        return rows
```

2.4. Implementação das métricas

Para a implementação das métricas, utilizamos a linguagem *Python* e criamos uma classe chamada *ConfusionMatrix*. Essa classe recebe em seu construtor duas listas de dados, sendo a primeira contendo os valores reais, informando se aquele item foi ou não um *hit* e a segunda lista contendo os valores preditos pelo modelo.

Durante a inicialização da classe, a matriz de confusão é calculada. Para o cálculo, é criado uma matriz de tamanho 2x2, representando as combinações possíveis, onde para cada lista recebida, há apenas duas possibilidades, sendo elas 0 (para não *hit*) e 1 (para *hit*). Essa matriz de confusão gerada é salva em uma variável interna da classe, que será utilizada para o cálculo das métricas quando necessário.

O cálculo das métricas foi feito de acordo com as fórmulas apresentadas em aula. Para cada métrica, um método foi criado nessa classe.

A seguir, temos o código utilizado para implementar a classe *ConfusionMatrix*:

```python
class ConfusionMatrix:
    """
    This class implements the metrics calculation.

    During the class initialization, it will calculate the confusion matrix and
will store it internally.
    After that, all metrics will be available calling their respective methods.
    """

    def __init__(self, true: np.ndarray, pred: np.ndarray) -> None:
        self._true = true
        self._pred = pred
        self._calculate_confusion_matrix()

    # Calculate the confusion matrix using a N x N array as input,
    # based on the array used to initialize the class.
    # The final result will be a M x M array, based on the quantity
    # of different values. (E.g.: in a binary class, it will be a 2x2 array)
    def _calculate_confusion_matrix(self) -> None:
        k = len(np.unique(self._true))
        result = np.zeros((k, k))

        for i in range(len(self._true)):
            result[self._true[i]][self._pred[i]] += 1

        # Store each result in its own variable
        # to be accessible easily inside this class
        self._tp = result[1, 1]  # True Positive
        self._tn = result[0, 0]  # True Negative
        self._fp = result[0, 1]  # False Positive
        self._fn = result[1, 0]  # False Negative
        self._matrix = result  # The confusion matrix
        self._num = len(self._true)  # Quantity of elements

    # Return the confusion matrix, if necessary
    def get_confusion_matrix(self) -> np.ndarray:
        return self._matrix

    # Plot the confusion matrix using a predefined template
    def plot_confusion_matrix(self) -> None:
        print("       |     Predicted class ")
        print("------|---------------------")
        print("       |     |   +   |   -   ")
        print("True   |  +  | %5d | %5d " % (self._tp, self._fn))
        print("class  |  -  | %5d | %5d \n" % (self._fp, self._tn))

    # Calculate and return the accuracy
    def get_accuracy(self) -> float:
        return (self._tp + self._tn) / self._num
```

```
    # Calculate and return the precision
    def get_precision(self) -> float:
        return self._tp / (self._tp + self._fp)

    # Calculate and return the recall
    def get_recall(self) -> float:
        return self._tp / (self._tp + self._fn)

    # Calculate and return the specicicity
    def get_specificity(self) -> float:
        return self._tn / (self._tn + self._fp)

    # Calculate and return the F1-measure
    def get_f1_measure(self) -> float:
        return self._tp / (self._tp + 0.5 * (self._fp + self._fn))
```

3. Resultados

Para a execução de cada algoritmo, foi definido o número de *k-folds* como 15. Sendo assim, cada algoritmo foi executado 15 vezes, tendo 14 grupos de treinamento e 1 grupo de teste para validação em cada iteração. O valor escolhido para o número de *folds*, assim como os hiperparâmetros dos modelos executados, visam diminuir o número de falsos positivos dos resultados. Nas primeiras tentativas de execução dos modelos, observou-se, pela matriz de confusão, que muitas instâncias que não são *hits* estavam sendo classificadas como *hits*. A fim de minimizar o problema, executou-se inúmeras vezes os algoritmos alternando-se os hiperparâmetros e o número de *folds*, até obtermos um resultado relativamente satisfatório que melhorasse a precisão dos modelos.

Os hiperparâmetros finais da *Random Forest* foram: número máximo de árvores igual a 10 (max_depth=10), o número mínimo de amostras necessárias para dividir um nó interno igual a 25 (min_samples_split=25), o número mínimo de amostras necessárias para estar em um nó folha (min_samples_leaf=10), o número de features a serem considerados ao procurar a melhor divisão como sendo a raiz quadrada do número total de features (max_features="sqrt") e a função de ganho utilizou-se a Gini (criterion="gini").

O hiperparâmetro final do *KNN* foi o número de vizinhos igual a 13. Já os hiperparâmetros finais do *Logistic Regression* foram: tolerância para o critério de parada de 0.001 e o algoritmo de otimização do problema foi o *newton-cg*.

O número de *folds* final para cada algoritmo foi de 15. Tentou-se diferentes números de folds, assim como os diferentes valores de hiperparâmetros, porém 15 foi o valor ideal encontrado para todos os algoritmos que minimizava o número de falsos positivos.

Os resultados de cada uma das 15 execuções (*k-folds* = 15), médias e desvio padrão para os algoritmos *Random Forest, K-Nearest Neighbors* e *Logistic Regression* podem ser vistos, respectivamente, nas Tabelas 1, 2 e 3. Já as Tabelas 4, 5 e 6 apresentam, respectivamente, a matriz de confusão da última execução dos algoritmos citados anteriormente. Foi decidido exibir apenas a matriz de confusão da última execução de cada algoritmo para minimizar a quantidade de dados no relatório.

Tabela 1: Resultados do algoritmo Random Forest

	Random Forest				
	Acurácia	**Precisão**	**Recall**	**Especificidade**	**F1-measure**
Exec. 1	0.80607	0.78355	0.84579	0.76636	0.81348
Exec. 2	0.81075	0.78059	0.86449	0.75701	0.8204
Exec. 3	0.8271	0.80702	0.85981	0.79439	0.83258
Exec. 4	0.83178	0.81416	0.85981	0.80374	0.83636
Exec. 5	0.75467	0.72428	0.82243	0.68692	0.77024
Exec. 6	0.8271	0.78689	0.8972	0.75701	0.83843
Exec. 7	0.76168	0.72222	0.85047	0.6729	0.78112
Exec. 8	0.78972	0.77679	0.81308	0.76636	0.79452
Exec. 9	0.78271	0.75745	0.83178	0.73364	0.79287
Exec. 10	0.81075	0.77593	0.87383	0.74766	0.82198
Exec. 11	0.79673	0.76349	0.85981	0.73364	0.80879
Exec. 12	0.82009	0.79654	0.85981	0.78037	0.82697
Exec. 13	0.8014	0.77922	0.84112	0.76168	0.80899
Exec. 14	0.81308	0.77686	0.8785	0.74766	0.82456
Exec. 15	0.82759	0.80645	0.86207	0.7931	0.83333
Média	0.80408	0.77676	0.85467	0.7535	0.81364
Des. Padrão	0.02276	0.02586	0.02093	0.0353	0.02012

Tabela 2: Resultados do algoritmo K-Nearest Neighbors

	K-Nearest Neighbors				
	Acurácia	**Precisão**	**Recall**	**Especificidade**	**F1-measure**
Exec. 1	0.78271	0.71841	0.92991	0.63551	0.81059
Exec. 2	0.75701	0.70073	0.8972	0.61682	0.78689
Exec. 3	0.72664	0.66899	0.8972	0.55607	0.76647
Exec. 4	0.79907	0.73529	0.93458	0.66355	0.82305
Exec. 5	0.79439	0.73684	0.91589	0.6729	0.81667
Exec. 6	0.80374	0.74621	0.92056	0.68692	0.82427
Exec. 7	0.78972	0.72464	0.93458	0.64486	0.81633
Exec. 8	0.78271	0.72	0.92523	0.64019	0.80982
Exec. 9	0.79673	0.73432	0.92991	0.66355	0.82062
Exec. 10	0.77804	0.71174	0.93458	0.6215	0.80808
Exec. 11	0.80374	0.73381	0.95327	0.65421	0.82927
Exec. 12	0.77336	0.71273	0.91589	0.63084	0.80164
Exec. 13	0.81542	0.75665	0.92991	0.70093	0.83438
Exec. 14	0.76869	0.71375	0.8972	0.64019	0.79503
Exec. 15	0.7931	0.72624	0.94089	0.64532	0.81974
Média	0.78434	0.72269	0.92378	0.64489	0.81085
Des. Padrão	0.02126	0.02005	0.01609	0.03269	0.01703

Tabela 3: Resultados do algoritmo Logistic Regression

	Logistic Regression				
	Acurácia	**Precisão**	**Recall**	**Especificidade**	**F1-measure**
Exec. 1	0.8014	0.75904	0.88318	0.71963	0.81641
Exec. 2	0.79206	0.749	0.8785	0.70561	0.8086
Exec. 3	0.8014	0.73455	0.94393	0.65888	0.82618
Exec. 4	0.81308	0.768	0.8972	0.72897	0.82759
Exec. 5	0.82243	0.75746	0.9486	0.69626	0.84232
Exec. 6	0.81075	0.76494	0.8972	0.7243	0.82581
Exec. 7	0.81075	0.75285	0.92523	0.69626	0.83019
Exec. 8	0.81075	0.75676	0.91589	0.70561	0.82875
Exec. 9	0.80607	0.7417	0.93925	0.6729	0.82887
Exec. 10	0.78505	0.72263	0.92523	0.64486	0.81148
Exec. 11	0.8271	0.76515	0.94393	0.71028	0.84519
Exec. 12	0.78972	0.72628	0.92991	0.64953	0.81557
Exec. 13	0.81542	0.75665	0.92991	0.70093	0.83438
Exec. 14	0.74766	0.69203	0.89252	0.6028	0.77959
Exec. 15	0.79803	0.73004	0.94581	0.65025	0.82403
Média	0.80211	0.74514	0.91975	0.68447	0.823
Des. Padrão	0.01844	0.02006	0.02329	0.03496	0.01523

Tabela 4: Matriz de confusão da execução 15 do algoritmo Random Forest

Random Forest	**Classe predita**		
Classe verdadeira		**+**	**-**
	+	175	28
	-	42	161

Tabela 5: Matriz de confusão da execução 15 do algoritmo K-Nearest Neighbors

K-Nearest Neighbors	**Classe predita**		
Classe verdadeira		**+**	**-**
	+	191	12
	-	72	131

Tabela 6: Matriz de confusão da execução 15 do algoritmo Logistic Regression

Logistic Regression	**Classe predita**		
Classe verdadeira		**+**	**-**
	+	192	11
	-	71	132

4. Considerações finais

Como foi observado na análise dos dados e também durante o treinamento dos modelos, pelo fato de que os dados acabam se sobrepondo no plano e tendo características um tanto quanto parecidas entre as classes, os algoritmos acabaram classificando muitas instâncias que não eram *hits* como *hits*. Além disso, os algoritmos apresentaram métricas muito similares, porém o *KNN* foi o que possui o menor valor de acurácia e precisão. Como estávamos interessados em minimizar os falsos positivos, a precisão é a métrica que mais nos importou, dessa forma a Random Forest acabou se sobressaindo em relação aos outros modelos, tendo uma precisão média de 77%. Também podemos observar, nos três modelos executados, que não tivemos um problema tão grande em relação aos falsos negativos, visto que o Recall, em média, foi maior que 80%.

O maior desafio que o grupo enfrentou foi de conseguir um valor de Precisão maior para o problema. Apesar de testar inúmeras vezes os modelos com hiperparâmetros diferentes, a maior precisão que se conseguiu foi de 77%.