Relatório Final

Projeto Final da disciplina de Mineração de Dados – Aluno: Andrey Moutelik (arma2)

Dataset Swarm Behaviour Classification:
https://archive.ics.uci.edu/ml/datasets/Swarm+Behaviour

O CRISP-DM (Cross Industry Standard Process for Data Mining) consiste em um conjunto de boas práticas para se executar um projeto em Ciência de Dados. Essa metodologia apresenta 6 fases a serem seguidas:

1- Entendimento do negócio
2- Entendimento dos dados
3- Preparação dos dados
4- Modelagem dos dados
5- Avaliação do Modelo
6- Deployment

Para este projeto, foram utilizadas essas guidelides e sequencialmente, semana a semana, este relatório final foi tomando forma. Algumas considerações que devo fazer inicialmente são primeiro que, embora as três primeiras etapas sejam apresentadas em sequência, para cada modelo eu faço as etapas 4 e 5 antes de passar para o seguinte, de forma a melhor representar o processo de obtenção de hiperparâmetros. Além disso, não poderei ir muito a fundo na etapa de deployment, visto que não há um plano de lançamento deste projeto, cliente interessado ou necessidade de manutenção após o final da disciplina.

---

1 - Entendimento do Negócio

O comportamento de movimento coletivo, como o movimento de enxames de abelhas, bandos de pássaros ou cardumes de peixes, inspirou sistemas de enxame baseados em computador. Esses sistemas são amplamente utilizados no controle da formação de agentes, como veículos aéreos e terrestres, equipes de robôs de resgate e exploração de ambientes perigosos com grupos de robôs. O comportamento do movimento coletivo é fácil de descrever, mas altamente subjetivo de detectar. Humanos tem facilidade para fazer essa detecção, mas isso não é o caso para robôs. Dados coletados da percepção humana são, portanto, uma maneira de permitir que os métodos de aprendizado de máquina aprendam essa percepção.

Esses dados de reconhecimento de comportamento de movimento coletivo foram coletados por meio de uma pesquisa online. Nesta pesquisa, os participantes fornecem sua opinião sobre o comportamento de massas pontuais ‘boid’. Cada pergunta da pesquisa contém um pequeno vídeo (em torno de 10 segundos), capturado a partir de movimentos corporais simulados. Os participantes foram solicitados a arrastar um controle deslizante para rotular cada vídeo como “reunindo” ou “não reunindo”; “alinhado” ou “não alinhado”; e “agrupado” ou “não agrupado”. Ao calcular a média dessas respostas, três rótulos binários foram criados para cada vídeo.

Sobre o valor dos dados

Porque os dados são úteis?

Os humanos podem reconhecer comportamentos de movimento coletivo com muita facilidade, embora seja uma tarefa difícil para as máquinas. Na verdade, não existe um conjunto de medidas objetivas pelas quais uma máquina possa identificar tipos de movimento coletivo. A coleta de um conjunto de dados que captura a percepção humana do movimento coletivo ajudará a construir ou treinar máquinas que podem reconhecer automaticamente esses comportamentos, imitando a percepção humana do movimento coletivo.

Quem pode se beneficiar?

Esses dados ajudarão as indústrias que desejam usar robôs de enxame para reconhecer e controlar comportamentos de enxame automaticamente. Esse reconhecimento de comportamento coletivo tem aplicação potencial em áreas como aviação civil, defesa, robótica de enxame e missões de interação humano-enxame onde os humanos controlam grupos de veículos não tripulados. Eles também podem beneficiar pesquisadores de aprendizagem de máquina ou pesquisadores em inteligência de enxame que gostariam de uma rotulagem de verdade dos comportamentos de movimento coletivo.

Como os dados podem ser usados?

Esses dados podem ser usados para treinar algoritmos de aprendizado supervisionado para imitar a percepção humana do movimento de enxame. Programas treinados podem então ser incorporados em sistemas de aprendizado secundário para ajustar ou desenvolver o comportamento de enxame.

Usos futuros:

No futuro, é previsto que seja possível aplicar o aprendizado de transferência desse conjunto de dados para controlar robôs de enxame com uma variedade de dinâmicas comportamentais.

2- Entendimento dos Dados

O dataset utilizado no projeto, 'Swarm Behavior Data', contém três CSVs distintos: Aligned, Flocking e Grouped, cada um classificando os dados respectivamente em Aligned ou Not Aligned, Flocking ou Not Flocking, e Grouped ou Not Grouped.

Uma coisa que identifico logo que tornou este projeto certamente único foram as features:

---

xm e ym são a posição (X,Y) de cada boid;

xVeln e yVeln são o vetor de velocidade;

xAm e yAm são o vetor de alinhamento;

xSm e ySm são o vetor de separação;

xCm e yCm são o vetor de coesão;

nACm é o número de boids em um raio de Alinhamento/Coesão;

nSm é o número de boids no raio de Separação.

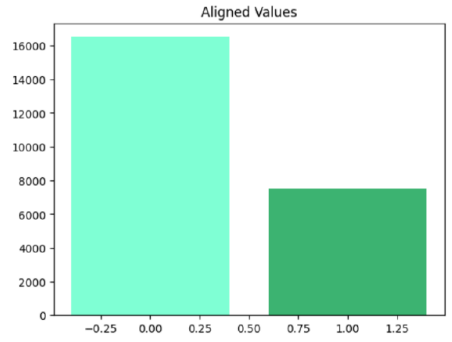
Esses atributos são repetidos para todos os m boids, onde m = 1,...,200. (12*200 = 2 400)

Resultando em **2400 FEATURES!**

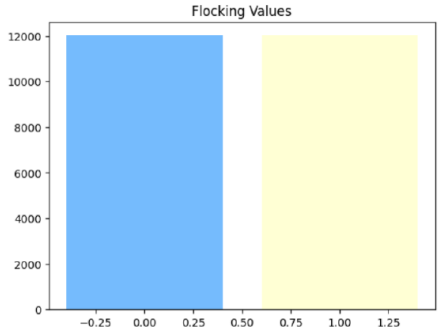
Eu nunca havia trabalhado com um problema de aprendizagem de máquina como esse, e o projeto como um todo se mostrou bem diferente devido a isso.

De acordo com o roteiro de EDA seguido, primeiramente é verificada a consistência dos dados: não há valores faltantes em nenhum dos datasets, e também não há valores anômalos ou linhas duplicadas. Uma etapa adicional que fiz foi verificar se os 3 CSVs eram, com exceção do atributo de classe na coluna final, idênticos. Em teoria isso deveria ser o caso, porém na prática, enquanto as entradas de 'Aligned' e 'Grouped' realmente são iguais, as de 'Flocking' são parcialmente diferentes. Esta diferença é significativa o suficiente para desconsiderar o dataset em comparações com os outros dois, a quantidade de dados distintos é muito grande.

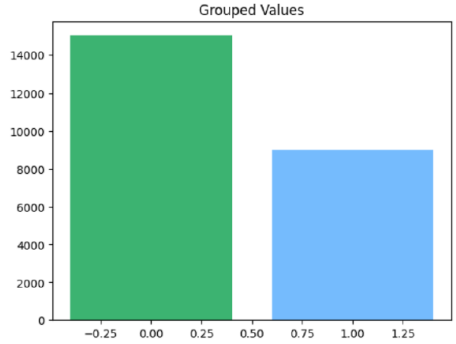
Em seguida, verifico o balanceamento dos três datasets com relação as suas respectivas classes:




```
0    16511
1     7505
Name: Class, dtype: int64
```




```
0    12008
1    12008
Name: Class, dtype: int64
```




```
0    15010
1     9006
Name: Class, dtype: int64
```

Vemos aqui ‘Aligned’ com um desbalanceamento de 69-31, ‘Flocking’ perfeitamente balanceado e ‘Grouped’ com um desbalanceamento um pouco menor, com 62,5% pertencendo a classe majoritária.

Alguns outros problemas emergem ao tentar seguir o roteiro de EDA:

Não é realista para este dataset fazer uma análise exploratória feature por feature como descrito no artigo recomendado. Temos simplesmente features demais, e eu as vejo praticamente como uma “caixa preta”, pois não posso inferir conclusões a partir delas. Nosso interesse é identificar o comportamento do todo, e, portanto o atributo do comportamento de um ‘boid’ fora do contexto geral não tem potencial de previsão nenhum.

Com relação a identificação de outliers, também não acredito que seja algo aconselhável devido ao contexto do problema. Os dados provêm das simulações usadas para identificação de padrões de comportamento de grupos, e eu não sei como determinar, para cada feature do dataset, o que seria um outlier ou não.

Uma ideia interessante que tentei explorar nessa etapa foi a de misturar as classes, fazer um classificador que ao invés de binário conseguisse identificar entre as seguintes possibilidades de classes:

0: Not Aligned, Not Flocking, Not Grouped

1: Aligned, Not Flocking, Not Grouped

2: Not Aligned, Flocking, Not Grouped

3: Not Aligned, Not Flocking, Grouped

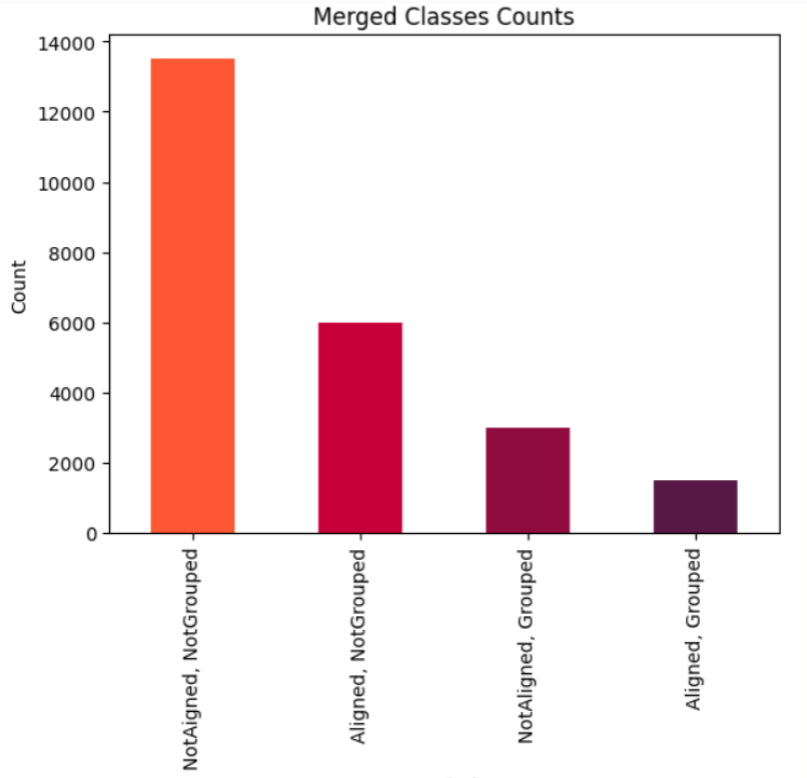
4: Aligned, Flocking, Not Grouped

5: Not Aligned, Flocking, Grouped

6: Aligned, Not Flocking, Grouped

7: Aligned, Flocking, Grouped

Devido a imparidade do dataset de ‘Flocking’ com os outros dois essa ideia logo foi impossibilitada, mas só por desencargo avaliei a possibilidade de fazer uma permutação só entre ‘Aligned’ e ‘Grouped’, checando o balanceamento das classes resultantes:



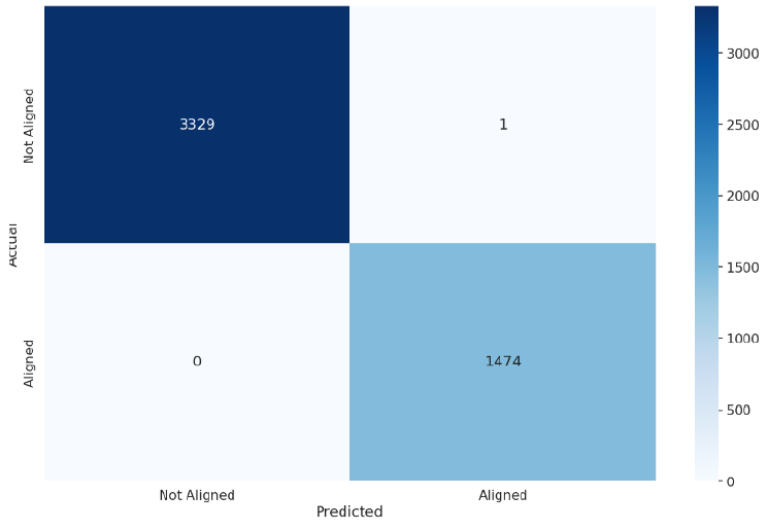
Vendo o resultado, o dataset me parece desbalanceado demais para apresentar um desempenho adequado para todas as classes. Por fins de simplicidade, a partir deste momento optei por escolher trabalhar com apenas um dos três datasets: O 'Aligned', por ser o mais desbalanceado. Devido as suas estruturas, todos os processos realizados nele podem ser replicados nos outros dois.

3- Preparação dos Dados

Nesta etapa do projeto fui orientado a estabelecer um parâmetro base de performance, antes de começar o pré-processamento. Optei por aplicar os dados crus em dois modelos distintos: Regressão Logística e SVM. Temos como resultado:

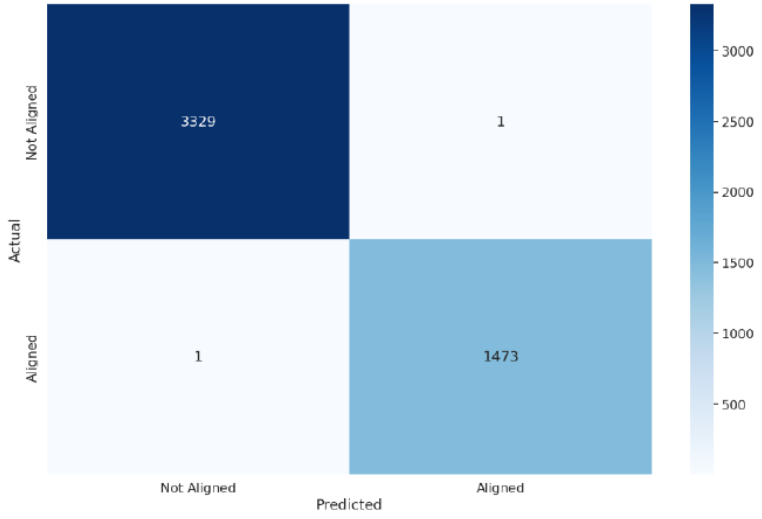
Regressão Logística

```
Training scores for each fold: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Testing scores for each fold: [1.0, 0.9994797086368367, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean testing score:, 1.0
Standard deviation of testing scores:, 0.000156
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804

Accuracy score:, 1.0
```



SVM

```
Training scores for each fold: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Testing scores for each fold: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean testing score:, 1.0
Standard deviation of testing scores:, 0.0
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804

Accuracy score:, 1.0
```



É evidente que num caso como esse está havendo overfitting. Eu gostaria de ter como provar isso, mas mesmo depois de horas de pesquisa não sei exatamente como. Assumo que seja mais devido aos indicadores (número absurdo de features, longo tempo de processamento..)

Enfim, a existência de overfitting é algo que eu devo ter como verdade se não tudo o que eu fizer daqui pra frente estaria apenas “piorando” o modelo.

Como adendo a esse quesito, eu gostaria de deixar explicito que testei esses modelos de diversas maneiras, com e sem cross validation, diferentes implementações.. Sempre com o mesmo resultado. Não acredito que haja data leakage, e também avaliei a correlação das features com as classes e não há problemas nesse aspecto.

Estabelecida nossa performance baseline, sigo para a preparação dos dados:

Train Test Split

Eu já começo o processo de preparação separando os dados em treino e teste.

```
✓ [86] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
0s
```

Estudos empíricos sugerem que um split de 70:30 ou 80:20 são ideais para melhores resultados. Considerando que utilizarei Undersampling na etapa seguinte e o subset de treinamento será reduzido, opto aqui por fazer o subset de teste 20% do total.

```
Shape of X_train: (19212, 2400)
Shape of X_test: (4804, 2400)
Shape of y_train: (19212,)
Shape of y_test: (4804,)
```

como referência, como ficou o shape dos dataframes.

Boas práticas sugerem a separação do dataset em subsets de treino e teste antes da etapa de balanceamento. A razão para isso é que essas técnicas, como oversampling e undersampling, são usadas para ajustar a distribuição das classes no conjunto de dados para resolver o desequilíbrio de classes. Se aplicarmos essas técnicas a todo o conjunto de dados (incluindo o conjunto de teste), corremos o risco de introduzir viés no conjunto de teste, o que pode levar a estimativas de desempenho excessivamente otimistas.

Ao usar técnicas de balanceamento apenas no conjunto de treinamento, garantimos que nosso modelo aprenda a generalizar para dados balanceados e evite o ajuste excessivo aos dados de treinamento desbalanceados. Em seguida, avaliamos o desempenho do nosso modelo no conjunto de teste independente, que é representativo da verdadeira distribuição dos dados no mundo real.

Balanceamento

Nesta etapa, faço o balanceamento do subset de treinamento, utilizando primeiramente Undersampling, reduzindo o número de elementos da classe mais representada, e em seguida Oversampling, aumentando o número da classe menos representada. Acredito que para o meu problema essa seja a melhor estratégia, a fim de não reduzir drásticamente o número de dados, mas também não gerar muitos dados ‘artificiais’.

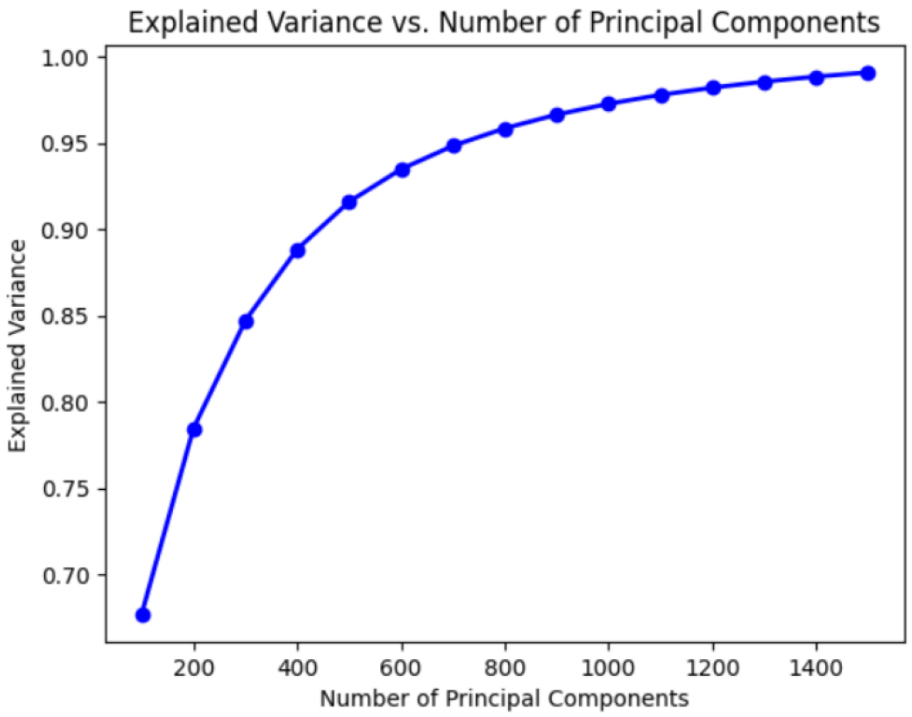
```
Number of samples in each class after undersampling:
0    8615
1    6031
Name: Class, dtype: int64
Number of samples in each class after oversampling:
0    8615
1    8615
Name: Class, dtype: int64
```

Para essa etapa, o parâmetro “sampling_strategy” do undersampling foi escolhido por um viés meu do que parecia mais adequado, não tendo necessariamente fundamentação. Isso não se mostrou um empecilho a acurácia no futuro.

Principal Component Analysis (PCA)

Por fim, aplico PCA afim de reduzir drasticamente o número de features do dataset. Isso é necessário para reduzir Overfitting do modelo, e para reduzir o tempo de processamento depois.

Normalizo os dados, e em seguida faço um gráfico de ‘Explained Variance’ x ‘Number of Principal Components’. Essa etapa é a que mais leva tempo (cerca de 6 minutos no colab). O número de componentes no gráfico varia de 100 a 1500, em intervalos de 100.



De acordo com o seguinte artigo sobre PCA no medium, eu devo almejar por uma métrica de 0.8, ou 80% de Explained Variance, a fim de reduzir overfitting.

https://towardsdatascience.com/dealing-with-highly-dimensional-data-using-principal-component-analysis-pca-fea1ca817fe6

So how can you tell how much information is retained in your PCA?

We use *Explained Variance Ratio* as a metric to evaluate the usefulness of your principal components and to choose how many components to use in your model. The explained variance ratio is the percentage of variance that is attributed by each of the selected components. Ideally, you would choose the number of components to include in your model by adding the explained variance ratio of each component until you reach a total of around 0.8 or 80% to avoid overfitting.

No meu caso, opto por usar 250 componentes, com uma Explained Variance de ~82%. Reduzimos o drasticamente o dataset para 10% de sua dimensionalidade original, porém mantivemos 82% da variância.

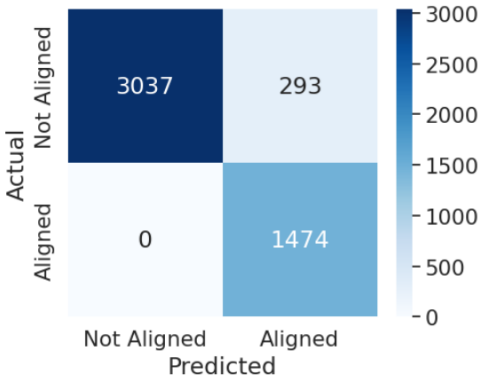
---

4 e 5 – Modelagem dos dados e Avaliação do Modelo

Por fins de simplicidade e redução da 'bagunça' e redundância de código, elaborei uma função que recebe um modelo e os conjuntos de treino e teste, faz a validação cruzada de acordo com as especificações do projeto (10-fold stratified cross validation), imprime a acurácia, o classification report e o desvio padrão, e retornar os dados relevantes na forma de um dicionário.

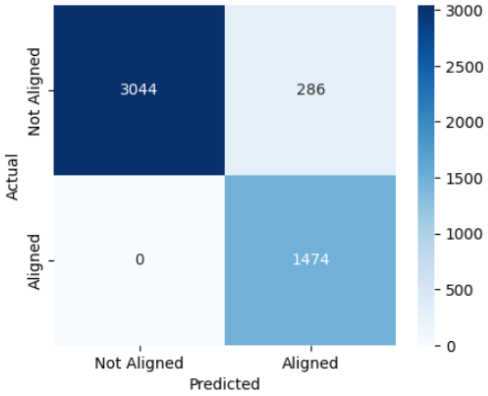
Nesta etapa do relatório apresento modelo por modelo as técnicas de variação paramétrica que utilizei para a identificação dos melhores hiperparâmetros. Adianto que em alguns casos o meu trabalho foi mais de generalização dos modelos, a fim da redução de overfitting.

Regressão Logística




```
Train scores: [0.999, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [0.999, 0.999, 0.999, 0.999, 1.0, 0.999, 1.0, 0.999, 0.997, 0.999]
Mean test score: 0.999
Standard deviation of test scores: 0.001

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.91      0.95      3330
           1       0.83      1.00      0.91      1474

    accuracy                           0.94      4804
   macro avg       0.92      0.96      0.93      4804
weighted avg       0.95      0.94      0.94      4804
```

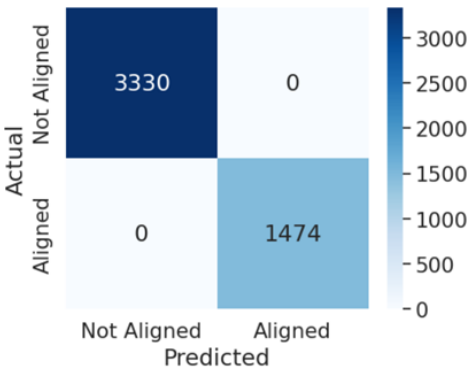
Este não foi um modelo requisitado pelas especificações, mas eu implementei na intenção de usar depois no comitê heterogêneo (e antes também no estabelecimento da performance baseline). Com relação ao ajuste de hiperparâmetros, explorei algumas alterações e tive uma leve melhora na acurácia trocando o 'C' de 1 para 10, já outras combinações de parâmetros explorados em grid search não se mostraram melhores que os valores default.




```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [1.0, 1.0, 1.0, 1.0, 1.0, 0.999, 1.0, 1.0, 1.0, 1.0]
Mean test score: 1.000
Standard deviation of test scores: 0.000

Accuracy: 0.940
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.91      0.96      3330
           1       0.84      1.00      0.91      1474

    accuracy                           0.94      4804
   macro avg       0.92      0.96      0.93      4804
weighted avg       0.95      0.94      0.94      4804
```

K-NN

Eu tinha preparado um algoritmo para escolher o melhor 'K' para o dataset, mas depois de rodar valores de ´1' a '31' ele retornou que o melhor valor de K é 1, e quando rodo o algoritmo tenho como resultados:




```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean test score: 1.000
Standard deviation of test scores: 0.000

Accuracy: 1.000
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```

Fiquei honestamente perplexo. Pelos meus conhecimentos o K do K-NN é o principal hiperparâmetro e mesmo com 1 o classificador é um 'fit perfeito'. O processo de seleção e ajuste de parâmetros, da forma como é descrito no artigo https://www.kdnuggets.com/2020/05/hyperparameter-optimization-machine-learning-models.html não faz sentido nesse caso, pois seria retornado apenas o K com o valor de 1 e os parâmetros em seu estado default.

O problema não está na função que fiz para rodar os modelos, pois uma implementação tradicional do knn também resulta na mesma acurácia.

Eu não sei para onde ir com esse modelo em específico. Poderia ajustar os parâmetros a fim de reduzir um suposto 'overfitting' (o que inclusive faço em modelos seguintes), mas com relação

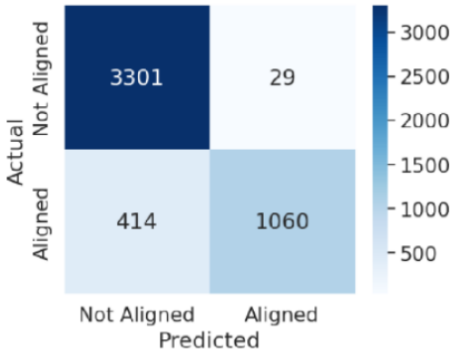
a identificação de parâmetros por grid search, random search ou outro método, neste caso não posso fazer nada.

LVQ

Após instalar o pacote de LVQ do sk_learn no colab (ele não está podendo ser importado da forma tradicional).

Uma primeira execução do LVQ resultou no seguinte classification report e matriz de confusão:

```
Train scores: [0.857, 0.834, 0.851, 0.851, 0.67, 0.851, 0.852, 0.831, 0.671, 0.826]
Mean train score: 0.809

Validation scores: [0.842, 0.835, 0.867, 0.843, 0.671, 0.844, 0.86, 0.82, 0.681, 0.835]
Mean validation score: 0.810

Test scores: [0.922, 0.914, 0.918, 0.915, 0.813, 0.915, 0.919, 0.91, 0.813, 0.908]
Mean test score: 0.895
Standard deviation of test scores: 0.041

Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.99      0.94      3330
           1       0.97      0.72      0.83      1474

    accuracy                           0.91      4804
   macro avg       0.93      0.86      0.88      4804
weighted avg       0.91      0.91      0.90      4804
```

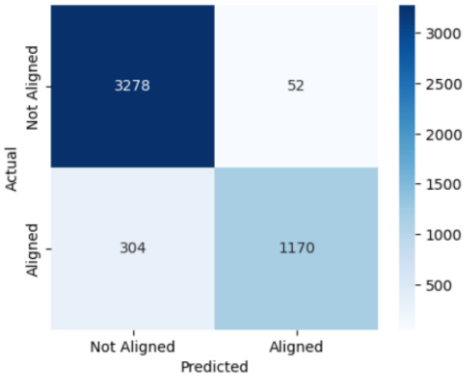


A fim de ajustar os hiperparâmetros (leve em conta que os parâmetros deste pacote de LVQ é um pouco diferente da documentação com a implementação “tradicional”), realizei um grid search com o seguinte dicionário:

```
param_grid = {
    'prototypes_per_class': [2, 3, 4],
    'beta': [1, 2, 3],
    'max_iter': [1000, 2500, 5000],
}
```

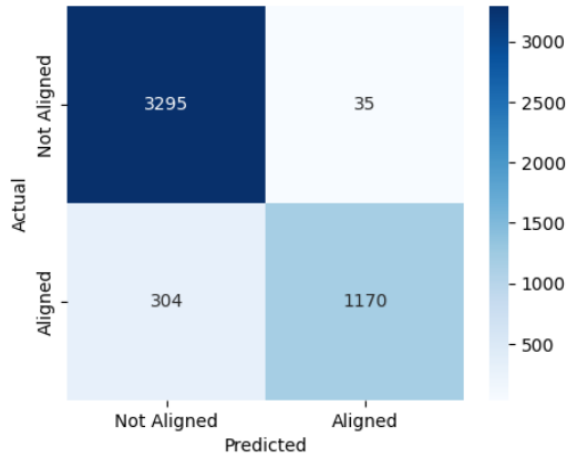
```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
Best parameters: {'beta': 3, 'max_iter': 2500, 'prototypes_per_class': 4}
Best score: 0.8560658338899029
```

Nesta execução eu usei apenas 3 como o valor para cross validation, a fim de reduzir um pouco o longo tempo da busca. Os valores e parâmetros foram escolhidos para a grid search com base nos valores comuns associados a eles.
De qualquer forma, usando os valores descobertos no grid search, executo o modelo novamente:




```
Train scores: [0.906, 0.852, 0.852, 0.906, 0.902, 0.9, 0.851, 0.85, 0.853, 0.851]
Mean train score: 0.872

Test scores: [0.891, 0.849, 0.86, 0.9, 0.894, 0.897, 0.858, 0.843, 0.862, 0.865]
Mean test score: 0.872
Standard deviation of test scores: 0.020

Accuracy: 0.926
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.98      0.95      3330
           1       0.96      0.79      0.87      1474

    accuracy                           0.93      4804
   macro avg       0.94      0.89      0.91      4804
weighted avg       0.93      0.93      0.92      4804
```

Temos notavelmente uma melhora na acurácia, o que é refletido na matriz de confusão.
Em seguida ainda alterei o parâmetro gtol para 1e-03, o que resultou em uma redução de acurácia para 88%, e em seguida testei como 1e-06. O modelo que levava cerca de 3 minutos para rodar levou 15 devido a isso, mas houve uma pequena melhora na acurácia:



Árvore de Decisão

Para este modelo, como de praxe, realizo uma execução sem determinar os parâmetros:

```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [0.999, 0.999, 0.999, 0.999, 0.999, 1.0, 0.999, 0.999, 0.999, 0.999]
Mean test score: 0.999
Standard deviation of test scores: 0.000

Accuracy: 0.949
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.93      0.96      3330
           1       0.87      0.99      0.92      1474

    accuracy                           0.95      4804
   macro avg       0.93      0.96      0.94      4804
weighted avg       0.95      0.95      0.95      4804
```

```
✓ [47] print("Maximum depth of decision tree:", dt.tree_.max_depth)
0s

Maximum depth of decision tree: 11
```

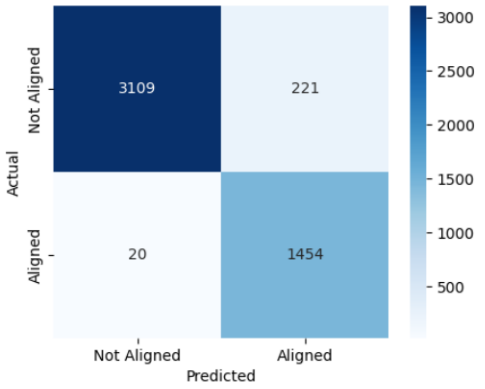
Temos uma acurácia de 94,9% e uma profundidade de 11.

Em seguida executo um Grid Search com valores e parâmetros comumente utilizados em árvores de decisão, e tenho o seguinte resultado:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

# Create a decision tree classifier object
dt = DecisionTreeClassifier(random_state=42)

# Define the grid search parameters
param_grid = {'max_depth': [5, 10, 20, 50, None],
              'min_samples_split': [2, 5, 10, 20],
              'min_samples_leaf': [1, 2, 4, 8],
              'max_features': ['sqrt', 'log2', None]}

# Perform the grid search
grid_search = GridSearchCV(dt, param_grid, cv=5, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)

# Print the best parameters and score
print("Best parameters found: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)
```

```
Fitting 5 folds for each of 240 candidates, totalling 1200 fits
Best parameters found:  {'max_depth': 20, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
Best score:  0.9992455020313408
```

Executando o modelo com os parâmetros encontrados, temos:




```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [0.999, 0.999, 0.999, 0.999, 0.999, 1.0, 0.999, 0.999, 0.999, 0.999]
Mean test score: 0.999
Standard deviation of test scores: 0.000

Accuracy: 0.949
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.93      0.96      3330
           1       0.87      0.99      0.92      1474

    accuracy                           0.95      4804
   macro avg       0.93      0.96      0.94      4804
weighted avg       0.95      0.95      0.95      4804
```

```
[52] print("Maximum depth of decision tree:", dt.tree_.max_depth)

Maximum depth of decision tree: 11
```

Considerando que os valores encontrados eram quase todos o padrão de inicialização de Decision Trees, o resultado que temos é exatamente igual ao que tínhamos antes, quando iniciamos a árvore sem parâmetro algum. O que pode ser feito agora é ajustar os parâmetros a fim de tornar o encaixe da árvore mais geral. Uma execução da árvore onde reduzo a profundidade máxima para 5, e dobro os valores de min_samples_leaf e min_samples_split termina com um resultado quase igual:

```
[71] from sklearn.tree import DecisionTreeClassifier

     dt = DecisionTreeClassifier(max_depth=5, min_samples_leaf=2, min_samples_split = 4, max_features = None, random_state=42)

     dt_dict = run_model(dt, X_train, y_train, X_test, y_test)
```

```
Train scores: [0.995, 0.995, 0.995, 0.996, 0.995, 0.996, 0.995, 0.994, 0.996, 0.995]
Mean train score: 0.995

Test scores: [0.994, 0.997, 0.994, 0.992, 0.994, 0.994, 0.992, 0.992, 0.995, 0.995]
Mean test score: 0.994
Standard deviation of test scores: 0.002

Accuracy: 0.949
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.93      0.96      3330
           1       0.87      0.99      0.92      1474

    accuracy                           0.95      4804
   macro avg       0.93      0.96      0.94      4804
weighted avg       0.95      0.95      0.95      4804
```

Nesse caso, acredito que para compensar a profundidade reduzida a árvore ficou mais larga. O atributo que mais importa nessa execução, e que suponho que reduz ao máximo overfitting é o 'max_features', que reduz bastante a acurácia do modelo se trocado para 'sqrt' ou 'log2'. A fim de tornar o modelo da árvore de decisão mais generalista, optei por refazer o grid search sem 'None' como parâmetro para 'max_features'. Encontrei os seguintes parâmetros:

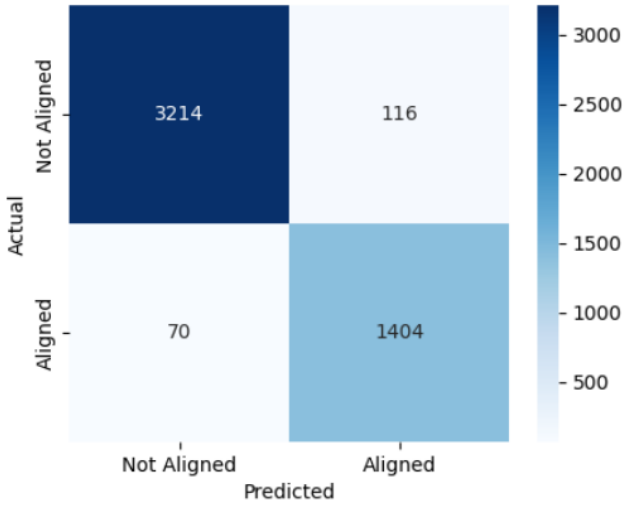
```
Fitting 5 folds for each of 160 candidates, totalling 800 fits
Best parameters found:  {'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 5}
Best score:  0.9954149738827626
```

```
dt = DecisionTreeClassifier(max_depth=20, min_samples_leaf=1, min_samples_split = 5, max_features = 'sqrt', random_state=42)

dt_dict = run_model(dt, X_train, y_train, X_test, y_test)
```

```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [0.995, 0.998, 0.995, 0.998, 0.994, 0.995, 0.997, 0.997, 0.995, 0.999]
Mean test score: 0.996
Standard deviation of test scores: 0.002

Accuracy: 0.961
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.97      0.97      3330
           1       0.92      0.95      0.94      1474

    accuracy                           0.96      4804
   macro avg       0.95      0.96      0.95      4804
weighted avg       0.96      0.96      0.96      4804
```

Curiosamente a acurácia resultante terminou sendo até um pouco maior. Segue também a matriz de confusão e a profundidade máxima da árvore.

```
[ ]  print("Maximum depth of decision tree:", dt.tree_.max_depth)

     Maximum depth of decision tree: 13

[ ]  plot_confusion_matrix(dt_dict, y_test, 1)
```



Uma coisa que tenho que comentar é que esta etapa do roteiro do projeto é impossível para mim:

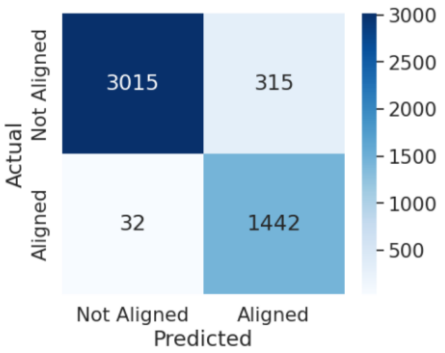
- Extrair uma árvore de decisão e identificar quais são os primeiros atributos utilizados na construção da árvore
  - Propor sugestões de decisões com base nos atributos encontrados

Não só estou trabalhando com um número absurdo de atributos, como também depois da aplicação do PCA estes não significam mais nada específico.

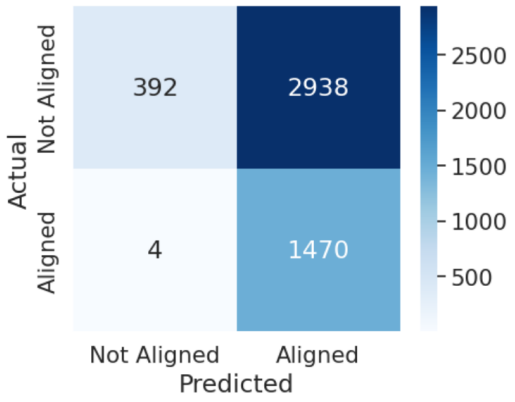
SVM

Com relação a SVM, confesso não ter realizado uma busca por hiperparâmetros como orientado. Primeiramente testei os diferentes kernels com o resto dos parâmetros default:

Linear, que apresentou um resultado razoável:



Poly, que depois de **23 minutos** de execução apresentou um resultado desastroso:




```
[50] from sklearn.svm import SVC

     svm = SVC(kernel='poly')

     svm_dict = run_model(svm, X_train, y_train, X_test, y_test)

     Train scores: [0.557, 0.559, 0.977, 0.559, 0.561, 0.562, 0.56, 0.559, 0.558, 0.558]
     Mean train score: 0.601

     Test scores: [0.553, 0.565, 0.981, 0.57, 0.561, 0.564, 0.554, 0.558, 0.555, 0.548]
     Mean test score: 0.601
     Standard deviation of test scores: 0.127

     Classification Report:
                   precision    recall  f1-score   support

                0       0.99      0.12      0.21      3330
                1       0.33      1.00      0.50      1474

         accuracy                           0.39      4804
        macro avg       0.66      0.56      0.36      4804
     weighted avg       0.79      0.39      0.30      4804
```

E então RBF, que foi um ‘fit perfeito’:

```
from sklearn.svm import SVC

svm = SVC(kernel='rbf', random_state=42)

svm_dict = run_model(svm, X_train, y_train, X_test, y_test)

Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [1.0, 0.999, 1.0, 1.0, 0.999, 1.0, 1.0, 1.0, 0.999, 0.999]
Mean test score: 1.000
Standard deviation of test scores: 0.000

Accuracy: 1.000
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```

Eu tinha lido em um artigo sobre SVM que o kernel rbf era propenso a overfitting. Mas devido a esse resultado, optei por explorar mais esse kernel, alterando os valores dos parâmetros ‘C’ e ‘gamma’ manualmente a fim de generalizar o modelo. Novamente, uma busca pelos melhores parâmetros nesse caso não faz muito sentido pois retornaria os valores padrão com o kernel rbf.

C = 0.1 e gamma = 0.01:

```
[57] from sklearn.svm import SVC

     svm = SVC(kernel='rbf', C= 0.1, gamma= 0.01, random_state=42)

     svm_dict = run_model(svm, X_train, y_train, X_test, y_test)
```

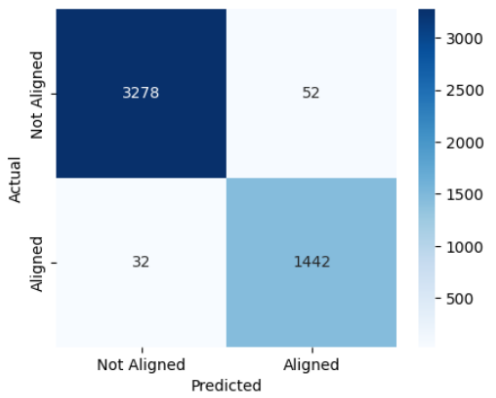
```
Train scores: [0.998, 0.998, 0.999, 0.998, 0.998, 0.998, 0.998, 0.998, 0.999, 0.998]
Mean train score: 0.998

Test scores: [0.998, 0.999, 0.998, 0.998, 0.998, 0.999, 0.999, 0.999, 0.997, 0.999]
Mean test score: 0.998
Standard deviation of test scores: 0.001

Accuracy: 1.000
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```

Redução do gamma para 0.001

```
[ ]  from sklearn.svm import SVC

     svm = SVC(kernel='rbf', C= 0.1, gamma= 0.001, random_state=42)

     svm_dict = run_model(svm, X_train, y_train, X_test, y_test)
```

```
Train scores: [0.873, 0.873, 0.873, 0.873, 0.871, 0.874, 0.872, 0.874, 0.874, 0.872]
Mean train score: 0.873

Test scores: [0.86, 0.873, 0.88, 0.864, 0.865, 0.868, 0.878, 0.873, 0.875, 0.885]
Mean test score: 0.872
Standard deviation of test scores: 0.007

Accuracy: 0.881
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.91      0.91      3330
           1       0.80      0.82      0.81      1474

    accuracy                           0.88      4804
   macro avg       0.86      0.86      0.86      4804
weighted avg       0.88      0.88      0.88      4804
```

Temos uma redução razoável na acurácia devido a essa mudança. Com a intenção de encontrar um balanço entre performance e generalização optei por aumentar o valor do gamma para 0.005:




```
16m  from sklearn.svm import SVC

     svm = SVC(kernel='rbf', C= 0.1, gamma= 0.005, random_state=42)

     svm_dict = run_model(svm, X_train, y_train, X_test, y_test)
```

```
Train scores: [0.983, 0.983, 0.983, 0.983, 0.983, 0.983, 0.983, 0.983, 0.983, 0.983]
Mean train score: 0.983

Test scores: [0.982, 0.983, 0.985, 0.981, 0.981, 0.979, 0.989, 0.981, 0.981, 0.985]
Mean test score: 0.983
Standard deviation of test scores: 0.003

Accuracy: 0.983
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.99      3330
           1       0.97      0.98      0.97      1474

    accuracy                           0.98      4804
   macro avg       0.98      0.98      0.98      4804
weighted avg       0.98      0.98      0.98      4804
```

Random Forest

Para Random Forest, tentei fazer um grid search inicialmente com diferentes valores para n_estimators, max_depth, min_samples_split e min_samples_leaf, mas logo percebi que provavelmente ia levar tempo demais. Uma execução do modelo sem hiperparâmetros resulta em:

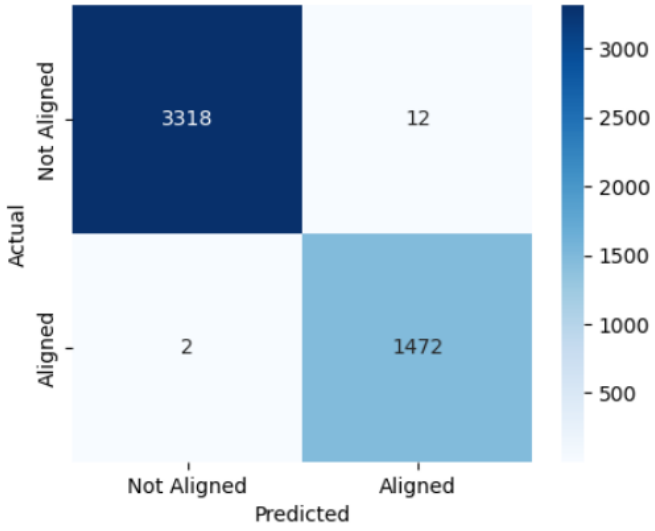
```
[79] from sklearn.ensemble import RandomForestClassifier

     rf = RandomForestClassifier()

     rf_dict = run_model(rf, X_train, y_train, X_test, y_test)
```

```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean test score: 1.000
Standard deviation of test scores: 0.000

Accuracy: 1.000
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```

E uma execução com parâmetros designados a reduzir overfitting resultam em algo bem semelhante:

```
[77] from sklearn.ensemble import RandomForestClassifier

     rf = RandomForestClassifier(n_estimators=100, random_state=42, max_depth=5, min_samples_split=4, min_samples_leaf=2, max_features = 'sqrt')

     rf_dict = run_model(rf, X_train, y_train, X_test, y_test)
```

```
Train scores: [0.998, 0.998, 0.999, 0.998, 0.998, 0.998, 0.999, 0.998, 0.998, 0.999]
Mean train score: 0.998

Test scores: [0.998, 0.998, 0.999, 0.997, 0.999, 0.996, 0.999, 0.998, 0.996, 0.998]
Mean test score: 0.998
Standard deviation of test scores: 0.001

Accuracy: 0.997
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       0.99      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```



Acredito que a perda mínima de acurácia seja compensada pelo tempo de execução menor e a maior generalização do modelo

MLP

Rodando sem determinação de parâmetros tenho como resultado:

```
[60] from sklearn.neural_network import MLPClassifier

     mlp = MLPClassifier(random_state=42)

     mlp_dict = run_model(mlp, X_train, y_train, X_test, y_test)
```

```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean test score: 1.000
Standard deviation of test scores: 0.000

Accuracy: 1.000
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```

Nunca pensei que ia ficar tão frustrado com esses classificadores “perfeitos”. Otimista com o tempo de execução do mlp (1 minuto), resolvi executar Random Search por parâmetros para ver no que dava.

Este foi o dicionário de parâmetros usado:

```python
# define the parameter space
param_dist = {'hidden_layer_sizes': sp_randint(50, 500),
              'activation': ['relu', 'tanh'],
              'solver': ['adam', 'lbfgs', 'sgd'],
              'alpha': [0.0001, 0.001, 0.01],
              'learning_rate': ['constant', 'adaptive'],
              'max_iter': sp_randint(500, 2000)}
```

DUAS HORAS DEPOIS, tive o seguinte resultado:

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best parameters:  {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': 320, 'learning_rate': 'constant', 'max_iter': 1595, 'solver': 'adam'}
Best score:  1.0
```

Best parameters: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': 320, 'learning_rate': 'constant', 'max_iter': 1595, 'solver': 'adam'}
Best score: 1.0

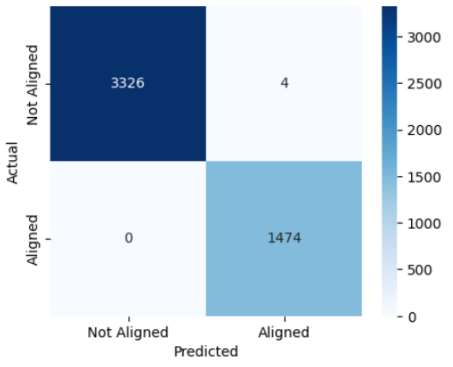
Sem surpresa nenhuma, o a execução do MLP com esses parâmetros resultou em 100% de acurácia.

O tamanho do número de hidden layers me parecia excessivo. Optei novamente então em tentar reduzi-lo e ver no que dá:

```python
[ ] from sklearn.neural_network import MLPClassifier

    mlp = MLPClassifier(activation= 'relu', alpha= 0.1, hidden_layer_sizes= 32, learning_rate='constant', max_iter=1500, solver = 'adam', random_state=42)

    mlp_dict = run_model(mlp, X_train, y_train, X_test, y_test)
```

```
Train scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean train score: 1.000

Test scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Mean test score: 1.000
Standard deviation of test scores: 0.000

Accuracy: 0.999
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```

Utilizando hidden_layer_sizes como 32 ao invés de 320 temos uma perda mínima de acurácia, acredito então que por ser mais generalista esta é uma representação melhor do modelo.



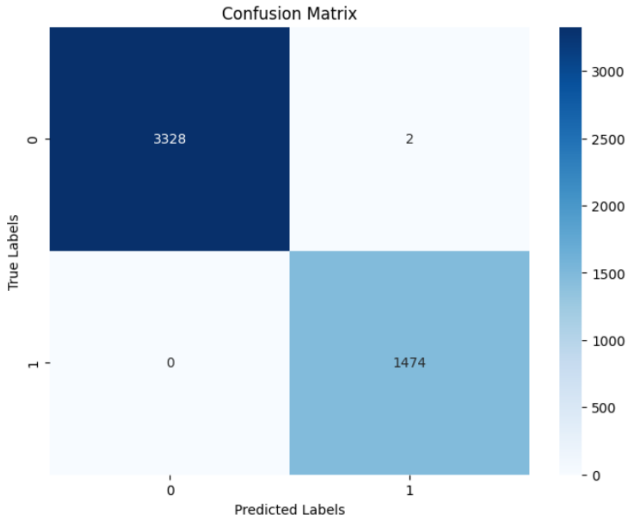
Comitê de Redes Neurais

O desempenho da rede neural MLP foi excepcional para a base de dados de teste. Assume-se, portanto, que um comitê delas apenas reforce ainda mais a acurácia. Para tornar esse modelo interessante o que fiz foi iniciar as redes neurais do comitê com hiperparâmetros aleatórios e um range relativamente pequeno de hidden_layer_sizes:

```
# Create and train an individual model with randomized hyperparameters
model = MLPClassifier(hidden_layer_sizes=(np.random.randint(16, 32),),
                      activation=np.random.choice(['relu', 'tanh']),
                      max_iter=np.random.choice([500, 1000, 1500]))
```

Como imaginei, a acurácia deste modelo está entre as melhores:

```
Accuracy: 0.9995836802664446
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3330
           1       1.00      1.00      1.00      1474

    accuracy                           1.00      4804
   macro avg       1.00      1.00      1.00      4804
weighted avg       1.00      1.00      1.00      4804
```

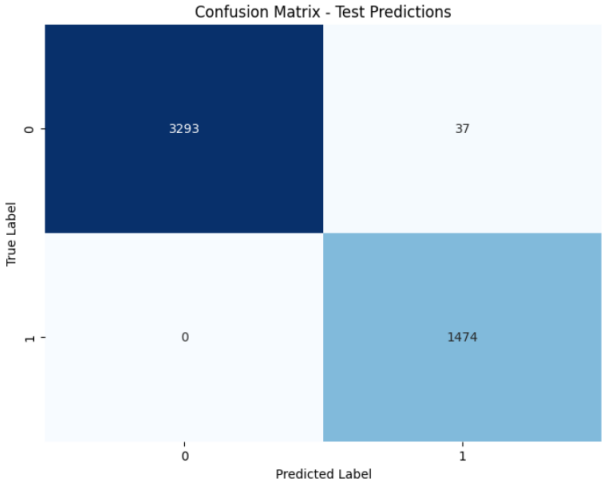


Não acredito que há muito mais que eu possa fazer com relação a ele.

Comitê Heterogêneo:

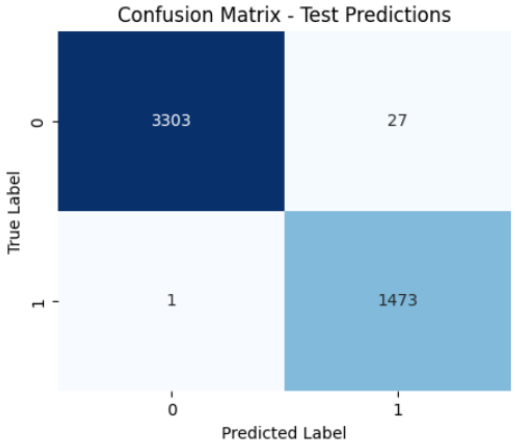
Para o comitê heterogêneo, eu optei por usar três modelos que apresentaram performances resultantes razoáveis ao longo do projeto: Regressão Logística, Árvore de Decisão e SVM.

Simplesmente executando o comitê sem ajustar os hiperparâmetros, e com ‘hard’ voting temos como resultado:

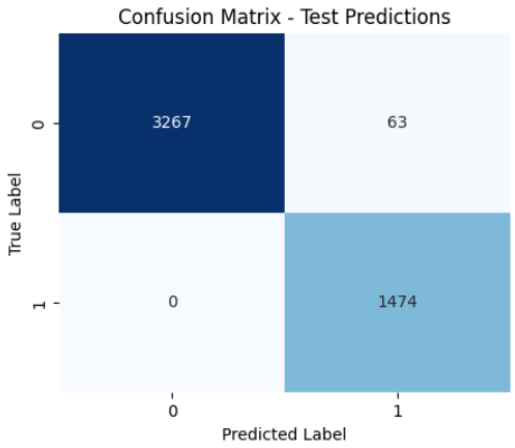



```
Accuracy: 0.9922980849292257
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      3330
           1       0.98      1.00      0.99      1474

    accuracy                           0.99      4804
   macro avg       0.99      0.99      0.99      4804
weighted avg       0.99      0.99      0.99      4804
```

Aplicando os hiperparâmetros que encontrei para cada um dos três modelos, temos uma pequena melhora na acurácia:



Em seguida, eu experimentei o modelo com ‘soft’ voting, mudando também o parâmetro ‘probability’ do SVM para True para que o ensemble funcionasse. Com isso há um aumento considerável da carga computacional devido ao SVM; A execução levou 35 minutos e o resultado do comitê foi pior, como pode ser visto nessa matriz:



Como um último experimento, tentei adicionar meu modelo de LVQ ao comitê. A performance porém também piorou devido a isso: