Projeto 1 de Aprendizado de Máquina
Grupo

Joao Victor de Lima Peixoto
Jose Douglas Pontes Silva
Marcos Heitor Carvalho de Oliveira
Mikael Vidal da Silva

Introdução

O projeto da cadeira de aprendizado de máquina visa aplicar as diferentes técnicas aprendidas na disciplina para fazer uma análise exploratória sobre a precisão (*precision*), cobertura (*recall*) e f1-score presentes no uso de cada algoritmo estudado em um *dataset* do mundo real.

Os algoritmos aprendidos na cadeira e testados no projeto foram:

- Árvores de Decisões
- Bayesiano Ingênuo
- Regressão Logística
- K Vizinhos

, onde cada algoritmo apresenta parâmetros e lógicas próprias que serão discutidas na fundamentação teórica deste relatório.

O conjunto de dados testado no projeto foi o Avila Data Set. Em resumo, esse conjunto de dados foi extraído através de 800 imagens da Bíblia de Avila, uma cópia em latim da Bíblia produzida no século XII em um local entre a Itália e a Espanha. No capítulo "Conjunto de dados", faremos uma descrição mais ampla de como esse dataset é dividido e também das suas features.

Por fim, mostraremos os resultados e conclusões obtidas dos nossos experimentos.

Objetivos

O objetivo geral do projeto é explorar os algoritmos aprendidos na cadeira de Aprendizagem de Máquina. Essa objetivo geral engloba os seguintes objetivos específicos:

- I. Explorar o conjunto de dados de Avila e verificar as features relevantes para cada método utilizado.
- II. Realizar o treinamento e classificação de cada um dos algoritmos aprendidos utilizando o Avila Data Set.
- III. Realizar uma análise exploratória de mudanças de hiper-parâmetros de cada algoritmo para buscar valores melhores de *precision*, *recall* e *f1-score.*

Fundamentação Teórica

Neste capítulo abordaremos alguns dos fundamentos básicos para que o leitor possa entender o projeto. Inicialmente falaremos sobre as métricas usadas para avaliar cada modelo e em seguida falaremos sobre cada modelo em si.

. Métricas

Em geral, quando estamos trabalhando com modelos de classificação, precisamos de algumas métricas para avaliar se a performance de um modelo é melhor do que a de outro modelo ao realizar uma classificação. Nesse contexto, precision, recall e f1-score servem como essas métricas para prover o quão bem um modelo é capaz de prever corretamente a classificação de uma instância com sua classe real. Além disso, essas métricas nos ajudam a comparar diferentes modelos ou parâmetros usados em algum classificador para otimizar sua performance.

. Precision:

A métrica de precisão (ou *precision*) mede a proporção de predições verdadeiras sobre todas as predições positivas feitas pelo modelo. A precisão é calculada como na fórmula abaixo:

$$\frac{TP}{TP + FP}$$

, onde TP é número de predições positivas enquanto FP é o número de predições falso-positivas. Uma precisão alta indica que o modelo é bom em identificar instâncias e tem uma taxa baixa de falso-positivos.

. Recall:

A métrica de cobertura (ou recall) mede a proporção de predições verdadeiras sobre todas as instâncias verdadeiramente positivas nos dados. A cobertura é calculada como na fórmula abaixo:

$$\frac{TP}{TP + FN}$$

, onde TP é número de predições positivas enquanto FN é o número de predições falso-negativas. Uma cobertura alta indica que o modelo é bom em identificar todas as instâncias positivas e tem uma taxa baixa de falso-negativos.

. F1-Score:

A métrica f1-score combina tanto a precisão como a cobertura em uma unica métrica. O f1-score pode ser calculada da seguinte forma:

$$2 * \frac{precision * recall}{precision + recall}$$

Essa métrica serve para avaliar a performance de um modelo onde ambas precisão e cobertura são desejadas.

. Algoritmos:

Dentro do assunto de aprendizado de máquina, os algoritmos que nós aprendemos servem como modelos de classificação. Esses modelos de classificação servem para prever uma classe categórica sobre uma instância de entrada levando em consideração suas features, ou parâmetros de entrada.

No caso do projeto, faremos um aprendizado supervisionado, onde usaremos um conjunto de dados já rotulados com a classe correta na entrada para treinar os modelos de classificação. Com isso, os modelos serão capazes de predizer a classe de dados ainda não vistos.

Existem muitos tipos de modelos de classificação na literatura. Entraremos agora em detalhe sobre os modelos de classificação usados no projeto.

. Decision Tree Classifier:

O classificador baseado em árvores de decisão (ou decision tree classifier) é um modelo que faz partições nas features em ramos que correspondem a um espaço que representa específica classe baseado em um conjunto de regras.

Existem várias formas de se dividir a árvore de decisão. Alguns dos parâmetros que valem a pena ser considerados são: a altura máxima da árvore, a estratégia de divisão de cada nó da árvore, o critério de divisão e o número máximo de features usadas. A altura máxima da árvore nos ajuda a decidir quantos critérios iremos considerar no nosso modelo, deixando assim a análise de cada feature mais complexa. Uma árvore curta dará respostas mais facilmente, no entanto, com menos precisão. A estratégia de divisão do nó nos ajuda a escolher o modo de melhor dividir a árvore, escolhendo métricas que priorizam a qualidade e ganho de informação do sistema.

. Naive Bayes:

Naive Bayes (ou classificador Bayesiano Ingênuo) é um modelo probabilístico que usa o teorema de Bayes para calcular a probabilidade das features da entrada pertencerem ou não a uma determinada classe.

Dado conjunto de entrada, com um vetor X de features e a classe Y, o modelo primeiro calcula as probabilidades à priori P(Y) para cada classe, o que representa a proporção de instâncias no conjunto de treinamento que pertencem a cada classe. Em sequência, o modelo calcula as probabilidades condicionais P(X | Y) para cada feature de entrada em cada classe, o que representa a probabilidade de observar cada valor da feature numa dada classe. Usando o teorema de Bayes, o modelo então calcula a probabilidade à posteriori P(Y | X) de cada classe dada a instância observada, o que indica a probabilidade de cada classe dado o vetor de features, então o modelo simplesmente seleciona a classe com maior probabilidade posterior.

. Logistic Regression:

A regressão logística (ou logistic regression) é um tipo de modelo de classificação linear utilizado comumente em problemas de classificação binária, onde o objetivo é prever um dos dois possíveis resultados com base em características de entrada. O modelo é chamado de "logístico" porque usa a função logística (também conhecida como função sigmoidal) para mapear a saída de uma função linear para um valor de probabilidade entre 0 e 1. No entanto, o modelo também pode ser usado para problemas com múltiplas classes, onde a função logística irá funcionar de maneira semelhante, mas agora calculando a probabilidade do vetor de entrada pertencer a uma das múltiplas classes possíveis.

Dado um conjunto de dados de treinamento com características de entrada X e rótulos de classe binários Y, o modelo primeiro aprende uma função que mapeia as características de entrada para um valor de saída contínuo. Essa função linear pode ser expressa como:

$$z = b0 + b1x1 + b2x2 + ... + bnxn$$

, onde z é a função linear das características de entrada, b0 é o termo de viés e cada b_x com x indo de 1 à n são os coeficientes (ou pesos) que correspondem a cada feature da entrada. A saída da função linear é então passada pela função logística, que mapeia a saída para um valor de probabilidade entre 0 e 1. O modelo então usa o valor de probabilidade para tomar uma decisão de classificação.

O modelo é treinado pela minimização de uma função de custo que mede o erro entre as probabilidades previstas e os rótulos de classe verdadeiros. Uma função de custo comumente usada para regressão logística é a função de perda de entropia cruzada binária.

. K-Neighbors:

O K neighbors (ou K-Vizinhos, ou até mesmo K-nearest neighbors: KNN) é um tipo de modelo de classificação não paramétrico e baseado em suas instâncias. O KNN faz suposições sobre as distribuições dos dados e pode trabalhar com features categóricas e numéricas. No KNN, a classificação é feita com base nos rótulos de classe dos k vizinhos mais próximos no espaço de características. No entanto, o valor de seu hiperparâmetro k deve ser escolhido com base no problema específico e nas características dos dados.

Dado uma nova instância para classificar, o modelo KNN calcula as distâncias entre a nova instância e todas as instâncias no conjunto de treinamento. Essa função de distância depende do problema selecionado. O KNN seleciona as k instâncias mais próximas da nova instância de entrada baseado na função escolhida e, assim, atribui a nova instância à classe que é mais frequente entre esses k vizinhos.

Uma das desvantagens do KNN é que pode ser computacionalmente caro para conjuntos de dados grandes, pois requer o cálculo das distâncias entre a nova instância e todas as instâncias no conjunto de treinamento. Além disso, KNN pode não funcionar bem em espaços de características de alta dimensão ou quando os dados são esparsos.

Conjunto de Dados

Os dados do projeto são referentes às informações extraídas das imagens de páginas copiadas da Bíblia de Avila. A análise paleográfica do manuscrito indicou a presença de 12 escrivães responsáveis pela cópia. A quantidade de páginas escritas por cada escrivão não é igual. Cada classe do dataset está dividida entre os tipos: A, B, C, D, E, F, G, H, I, W, X, e Y, onde cada classe faz referência à qual copiador foi responsável por aquele trecho / página.

Cada padrão de escrivão contém 10 features. Os dados foram normalizados usando o método Z-normalization método. Essa normalização é uma técnica usada para transformar um conjunto de características de entrada para ter uma média zero e um desvio padrão de um. Isso é feito subtraindo o valor médio de cada característica de cada ponto de dados e, em seguida, dividindo pelo desvio padrão desta característica. A normalização Z é comumente usada em aprendizado de máquina para garantir que todas as características de entrada estejam na mesma escala e tenham uma faixa de valores semelhante. Isso é particularmente importante para modelos que dependem de métricas baseadas em distância ou algoritmos de otimização, pois características com escalas ou variâncias maiores podem dominar o processo de tomada de decisão do modelo. A normalização Z pode ajudar a evitar esse problema e garantir que todas as características sejam igualmente importantes nas previsões do modelo.

O dataset foi dividido em duas partes, uma parte para treino, contendo 10430 amostras e outra parte para os testes, contendo 10437 amostras.

Metodologia

A metodologia utilizada neste projeto foi dividida em três etapas principais:

Realizamos o treinamento e classificação utilizando os valores default dos respectivos métodos para obter o valor primordial do precision, recall
e f1_score em cada um dos modelos propostos.

Realizamos uma análise exploratória em cada um dos hiperparâmetros para buscar otimizar os valores das métricas obtidas.

Por fim, utilizamos o Optuna, uma ferramenta que automatiza o processo de testagem de hiperparâmetros, para os modelos que tinham uma variação maior de parâmetros.
Para cada modelo de classificação diferente, alteramos os parâmetros que mais influenciavam na nossa classificação.

No caso do Decision Tree Classifier alteramos a altura máxima da árvore, a estratégia de divisão de cada nó da árvore, o critério de divisão e o número máximo de features usadas. A altura máxima tenta regular o feedback da resposta, enquanto a estratégia de divisão de cada nó diz como separar os nós baseado em suas features e critério de divisão.

No Naive Bayes Classifier, alteramos o parâmetro que regula a porção da maior variância de todas as características que é adicionada às variâncias para garantir a estabilidade dos cálculos. Apenas esse parâmetro é usado para a construção do modelo e foi o único variado.

No Logic Regression Classifier, alteramos os parâmetros de tolerância, penalidade, a função de regulação e o coeficiente de força de regulação. Cada parâmetro influencia no cálculo final do modelo, seja nas subdivisões espaciais para a entrada da classe ou como o modelo se comporta com entradas de pequena ordem.

Por fim, no KNN Classifier, testamos diferentes valores de k para o classificador e também a influência de que outras funções de distância faziam na composição do classificador.

A escolha de todos os hiperparâmetros foram feitos utilizando como o base f1_score em cima do conjunto de validação. Todos os testes foram utilizados usando a biblioteca do sklearn que disponibiliza todos esses classificadores e parâmetros estudados, além disso utilizamos o dataset do Avila em ambos os testes comentados.

Resultados

Observando o conjunto de dados

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	Class
0	0.266074	-0.165620	0.320980	0.483299	0.172340	0.273364	0.371178	0.929823	0.251173	0.159345	A
1	0.130292	0.870736	-3.210528	0.062493	0.261718	1.436060	1.465940	0.636203	0.282354	0.515587	A
2	-0.116585	0.069915	0.068476	-0.783147	0.261718	0.439463	-0.081827	-0.888236	-0.123005	0.582939	A
3	0.031541	0.297600	-3.210528	-0.583590	-0.721442	-0.307984	0.710932	1.051693	0.594169	-0.533994	A
4	0.229043	0.807926	-0.052442	0.082634	0.261718	0.148790	0.635431	0.051062	0.032902	-0.086652	F

Separando o conjunto de treinamento em TREINO 80% | VALIDAÇÃO 20%

```
In [19]: X_train,X_valid,y_train,y_valid = train_test_split(df.drop("Class",axis=1), df["Class"],test_size=0.2,random_state=2023)

In [20]: X_train.shape,y_train.shape,X_valid.shape,y_valid.shape

Out[20]: ((8344, 10), (8344,), (2086, 10), (2086,))
```

Primeira análise dos modelos com valores default's dos métodos utilizados pelo Scikit Learn para cada um dos classificadores:

Métricas para o conjunto de validação

	Precision	Recall	F1-Score
Decision Tree	97.56	97.56	**97.56**
Naive	48.69	28.04	31.27
Logistic	80.55	55.23	63.13
KNN	74.28	71.58	72.25

DecisionTreeClassifier parameters

De início já podemos observar que, utilizando apenas os valores defaults de cada um dos classificadores, o Decision Tree Classifier apresenta um melhor comportamento em relação à todas as métricas do que nos outros classificadores. Esse comportamento se deve ao fato de que as árvores de decisão tomam vantagem sobre a divisão das features para classificar o espaço de escrita de cada escrivão, tendo assim uma vantagem natural na classificação pela própria natureza dos dados.

De que formas podemos melhorar o atual modelo?

Primeiro começamos fazendo uma avaliação geral e mudando alguns hiperparâmetros para observar a variação de resultado da árvore. Para isso variamos cada um dos parâmetros comentados na metodologia e o resultados pode ser visto nas imagens abaixo:

In [27]:
```
clf_best = DecisionTreeClassifier(max_depth=12)
clf_best.fit(X_train, y_train)
display_metrics(clf_best, X_valid, y_valid)
```

	precision	recall	f1-score	support
A	0.87	0.96	0.91	850
B	1.00	1.00	1.00	2
C	1.00	0.73	0.84	26
D	0.77	0.83	0.80	69
E	0.92	0.83	0.87	231
F	0.91	0.84	0.88	379
G	0.92	0.75	0.83	89
H	0.97	0.86	0.91	112
I	1.00	0.98	0.99	180
W	1.00	0.88	0.93	8
X	0.92	0.91	0.92	92
Y	0.82	0.88	0.85	48
accuracy			0.90	2086
macro avg	0.93	0.87	0.89	2086
weighted avg	0.90	0.90	0.90	2086

Alterando apenas a altura máxima da árvore (max_depth), observamos que o valor geral das métricas do nosso modelo diminui. O que indica que apenas podar a árvore para ter uma resposta mais rápida não influencia tanto na qualidade do modelo.

In [28]:
```
clf_best = DecisionTreeClassifier(splitter="random")
clf_best.fit(X_train, y_train)
display_metrics(clf_best, X_valid, y_valid)
```

	precision	recall	f1-score	support
A	0.84	0.82	0.83	850
B	1.00	0.50	0.67	2
C	0.57	0.50	0.53	26
D	0.69	0.74	0.71	69
E	0.79	0.72	0.76	231
F	0.72	0.78	0.75	379
G	0.74	0.66	0.70	89
H	0.66	0.66	0.66	112
I	0.98	0.96	0.97	180
W	0.67	0.50	0.57	8
X	0.75	0.86	0.80	92
Y	0.80	0.81	0.80	48
accuracy			0.79	2086
macro avg	0.77	0.71	0.73	2086
weighted avg	0.79	0.79	0.79	2086

Ao alterarmos a estratégia de divisão de cada nó da árvore (splitter), podemos ver que uma divisão randômica dos dados para gerar os nós da árvore diminui bastante cada uma das

métricas do sistema, gerando assim um modelo que seja 20% menos eficiente do que nosso modelo básico.

```
In [29]: clf_best = DecisionTreeClassifier(criterion="entropy")
         clf_best.fit(X_train, y_train)
         display_metrics(clf_best, X_valid, y_valid)
```

	precision	recall	f1-score	support
A	0.99	0.99	0.99	850
B	1.00	0.50	0.67	2
C	1.00	0.85	0.92	26
D	0.99	1.00	0.99	69
E	0.98	0.97	0.98	231
F	0.99	0.99	0.99	379
G	0.98	0.98	0.98	89
H	0.96	0.96	0.96	112
I	0.99	1.00	1.00	180
W	1.00	0.75	0.86	8
X	0.98	0.98	0.98	92
Y	0.96	1.00	0.98	48
accuracy			0.99	2086
macro avg	0.99	0.91	0.94	2086
weighted avg	0.99	0.99	0.99	2086

Ao alterarmos o critério de divisão (criterion) do nosso modelo podemos observar que o critério que prioriza a entropia otimiza um pouco mais nossas métricas, visto que ao priorizar a entropia do sistema criamos uma árvore que valoriza mais as features do nosso problema.

In [30]:

```
clf_best = DecisionTreeClassifier(criterion="log_loss")
clf_best.fit(X_train, y_train)
display_metrics(clf_best, X_valid, y_valid)
```

	precision	recall	f1-score	support
A	0.99	0.99	0.99	850
B	1.00	0.50	0.67	2
C	1.00	0.85	0.92	26
D	0.99	0.99	0.99	69
E	0.98	0.97	0.98	231
F	0.99	0.99	0.99	379
G	0.98	0.98	0.98	89
H	0.96	0.96	0.96	112
I	0.99	1.00	1.00	180
W	1.00	0.75	0.86	8
X	0.97	0.97	0.97	92
Y	0.94	1.00	0.97	48
accuracy			0.98	2086
macro avg	0.98	0.91	0.94	2086
weighted avg	0.98	0.98	0.98	2086

Também temos uma melhora quando utilizamos o método de log_loss também temos uma otimização por não gerarmos uma árvore de qualquer maneira e sim uma que prioriza nosso ganho de informação.

In [32]:

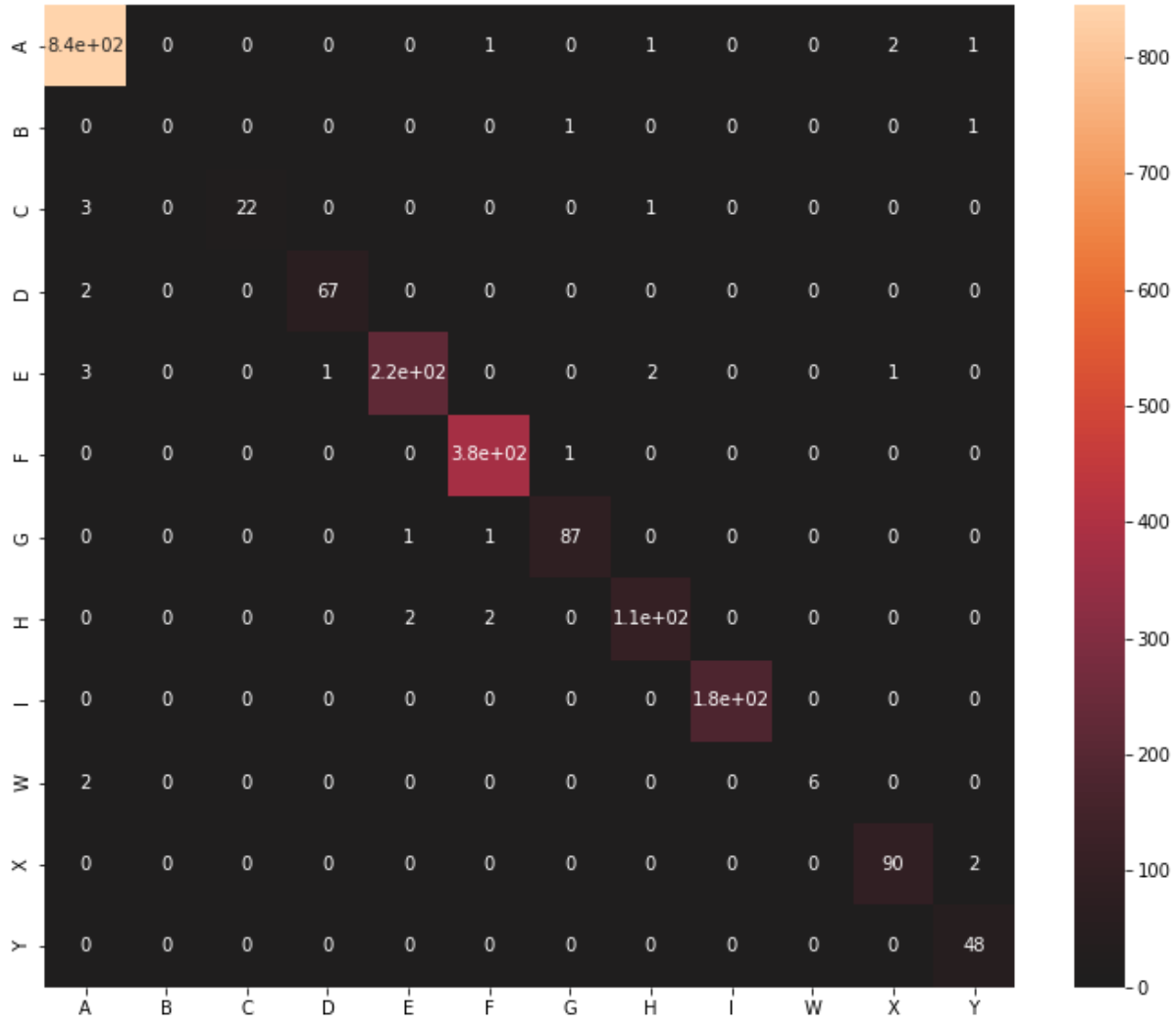
```
clf_best = DecisionTreeClassifier(max_features="log2")
clf_best.fit(X_train, y_train)
display_metrics(clf_best, X_valid, y_valid)
```

	precision	recall	f1-score	support
A	0.91	0.91	0.91	850
B	1.00	0.50	0.67	2
C	0.81	0.81	0.81	26
D	0.89	0.84	0.87	69
E	0.87	0.87	0.87	231
F	0.86	0.87	0.86	379
G	0.77	0.78	0.77	89
H	0.85	0.83	0.84	112
I	0.97	0.98	0.98	180
W	0.80	1.00	0.89	8
X	0.88	0.90	0.89	92
Y	0.88	0.88	0.88	48
accuracy			0.89	2086
macro avg	0.87	0.85	0.85	2086
weighted avg	0.89	0.89	0.89	2086

Ao alterarmos o número máximo de features usadas (max_features), podemos ver que ao considerarmos um número menor de features do que os usados antes, nossas métricas têm um desempenho pior. Isso se deve ao fato de que as features têm uma importância considerável no processo de classificação. Ao removermos essas informações, perdemos a precisão.

Melhor combinação de hiperparâmetros no testes realizados

	precision	recall	f1-score	support
A	0.99	0.99	0.99	850
B	0.00	0.00	0.00	2
C	1.00	0.85	0.92	26
D	0.99	0.97	0.98	69
E	0.99	0.97	0.98	231
F	0.99	1.00	0.99	379
G	0.98	0.98	0.98	89
H	0.96	0.96	0.96	112
I	1.00	1.00	1.00	180
W	1.00	0.75	0.86	8
X	0.97	0.98	0.97	92
Y	0.92	1.00	0.96	48
accuracy			0.99	2086
macro avg	0.90	0.87	0.88	2086
weighted avg	0.98	0.99	0.98	2086



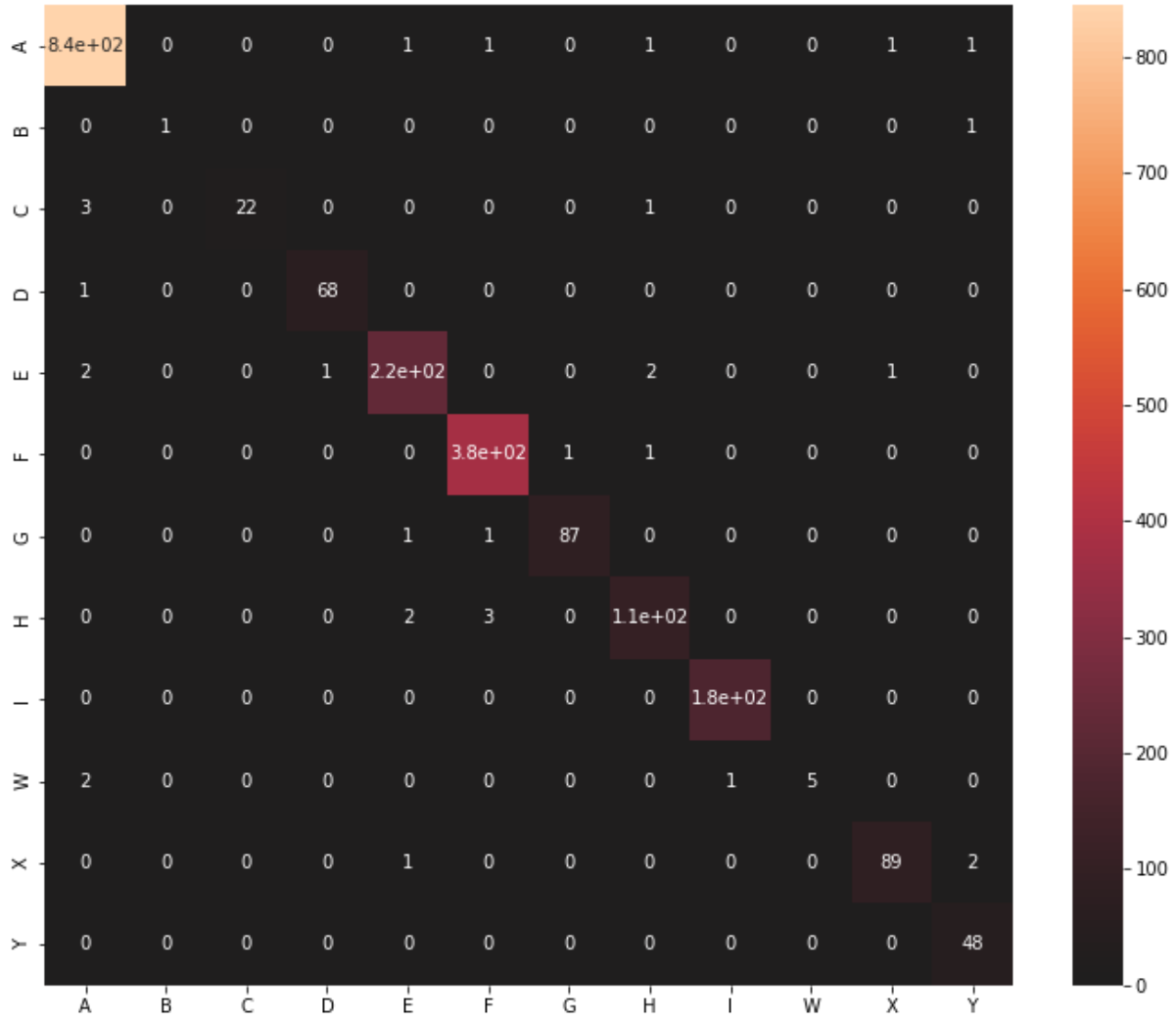
No processo final, podemos observar que ao combinarmos melhor a quantidade de parâmetros usados pelo modelo, podemos obter uma melhora no sistema de classificação devido à análise feita anteriormente nos parâmetros.

Uma outra maneira de tentar modificar os hiperparâmetros é utilizando-se de uma ferramenta chamada de Optuna, que ajuda a definirmos alguns hiperparâmetros nos quais

ele irá testar automaticamente.

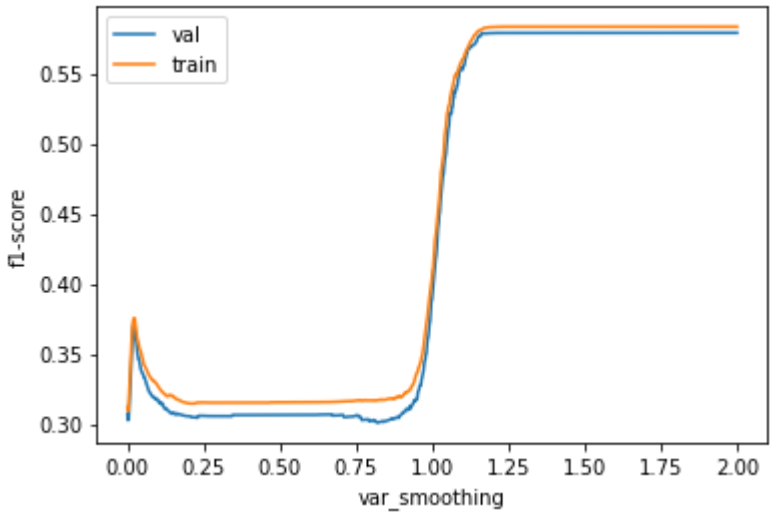
```
In [38]: clf_optuna_best = DecisionTreeClassifier(max_depth = 20, criterion="entropy",min_samples_split=2, min_samples_leaf=1)
         clf_optuna_best.fit(X_train, y_train)

Out[38]: DecisionTreeClassifier(criterion='entropy', max_depth=20)
         In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
         On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [39]: display_metrics(clf_optuna_best,X_valid,y_valid)
```

	precision	recall	f1-score	support
A	0.99	0.99	0.99	850
B	1.00	0.50	0.67	2
C	1.00	0.85	0.92	26
D	0.99	0.99	0.99	69
E	0.98	0.97	0.98	231
F	0.99	0.99	0.99	379
G	0.99	0.98	0.98	89
H	0.96	0.96	0.96	112
I	0.99	1.00	1.00	180
W	1.00	0.62	0.77	8
X	0.98	0.97	0.97	92
Y	0.92	1.00	0.96	48
accuracy			0.98	2086
macro avg	0.98	0.90	0.93	2086
weighted avg	0.98	0.98	0.98	2086



Podemos observar que, ao usarmos o Optuna, conseguimos otimizar ainda um pouco mais os parâmetros devido a busca pela melhor configuração dos parâmetros extras.

Naive Bayes:



Com o Naive Bayes, variamos o valor do smoothing e plotamos quais os valores da métrica de f1_score para cada valor testado. Pelo gráfico acima podemos observar que com um var_smoothing de aproximadamente 1.225 nós estabilizamos o f1_score do nosso modelo e deixamos ele mais otimizado.

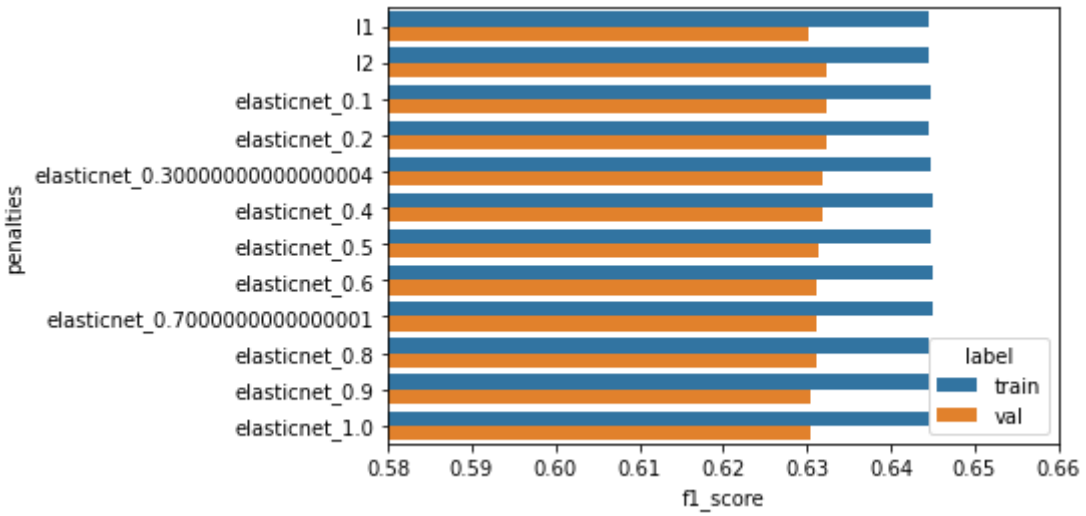
```
In [42]: clf = GaussianNB(var_smoothing = 1.225)
         clf.fit(X_train,y_train)
         print("acuracia =",clf.score(X_valid,y_valid))

         pred = clf.predict(X_valid)
         print("f1",f1_score(pred,y_valid,average="weighted"))
         print("recall",recall_score(pred,y_valid,average="weighted"))
         print("precision",precision_score(pred,y_valid,average="weighted"))
         print("-"*100)

         acuracia = 0.4074784276126558
         f1 0.5790190735694823
         recall 0.4074784276126558
         precision 1.0
         ----------------------------------------------------------------------------------------------------
```

```
In [43]: check_test(clf,X_test,y_test)

         Precision:  0.9997128540868042
         Recall:  0.410846028552266
         F1-score:  0.5821701553337219
         ------------------------------------------------
```

Logistic Regression:

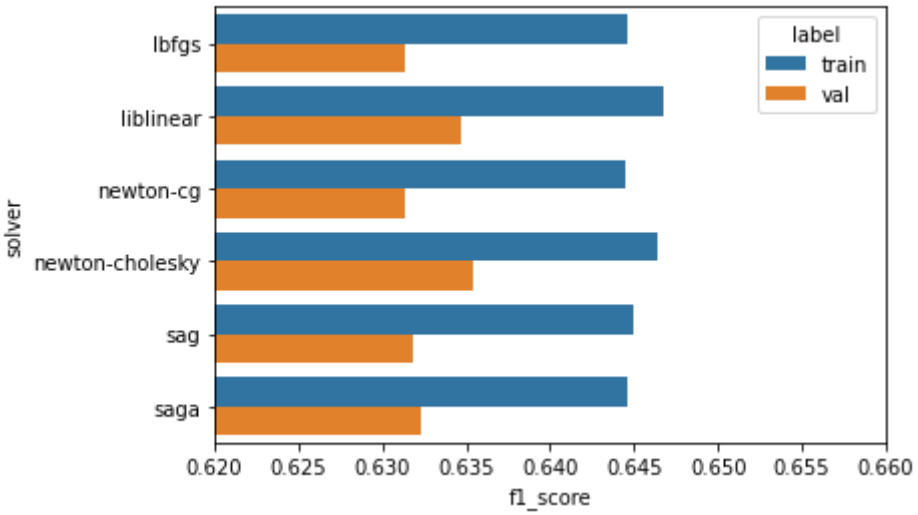
Para os testes a seguir, vamos testar individualmente cada hiperparâmetro, para cada hiperparâmetro testamos com um loop vários valores e plotamos gráficos para visualizar a progressão do f1 score para o conjunto de treino e de validação, para que baseado nos plots, possamos escolher o melhor valor para cada hiperparâmetro

.Check penalty:



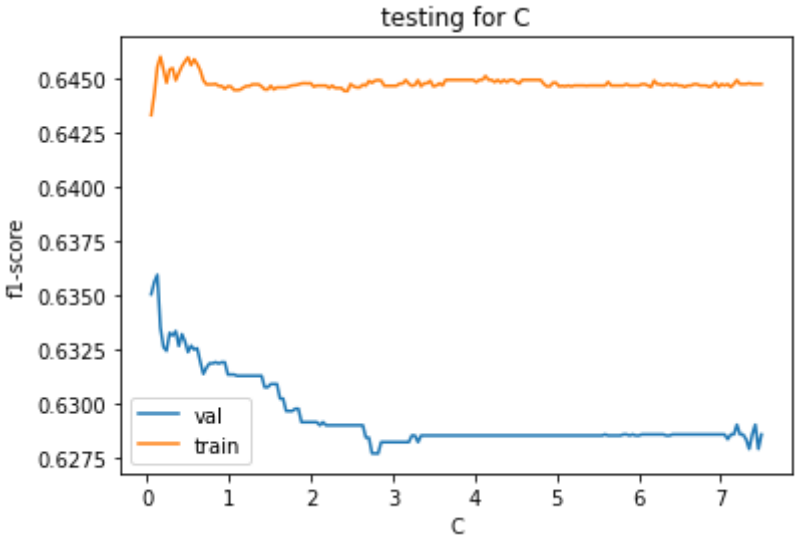
Pelo gráfico podemos observar que o método que teve maior desempenho no conjunto de validação foi utilizando a penalidade da norma L2.

.Check solver:



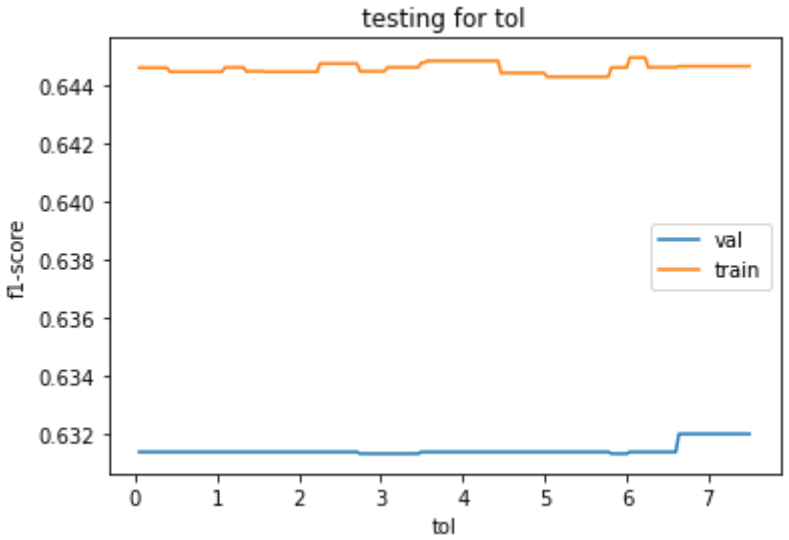
Pelo plot, podemos observar que o maior desempenho no conjunto de treinamento é o “liblinear”, no entanto, o que tem maior desempenho no conjunto de validação é o “newton-cholesky”.

.Check C:



Pelo coeficiente `C` de regulação, podemos observar que o valor que teve uma melhor métrica f1 com o conjunto de validação foi com o valor de 0.12.

.Check tol:



Aplicando a análise na tolerância podemos observar que o valor que teve a melhor métrica f1 no teste de validação foi o valor de 0.94.

```
[ ] clf =  Pipeline([('scaler', StandardScaler()), ('LR', LogisticRegression(penalty='l2',solver='newton-cholesky',C=0.12,tol=0.94,max_iter=10000))
    clf.fit(X_train,y_train)
    pred = clf.predict(X_valid)
    print("f1_val",f1_score(pred,y_valid,average="weighted"))
    pred = clf.predict(X_train)
    print("f1_train",f1_score(pred,y_train,average="weighted"))
```

```
check_test(clf,X_test,y_test)
```

```
Precision:  0.9291839021869868
Recall:  0.4786816134904666
F1-score:  0.6228920819461955
```

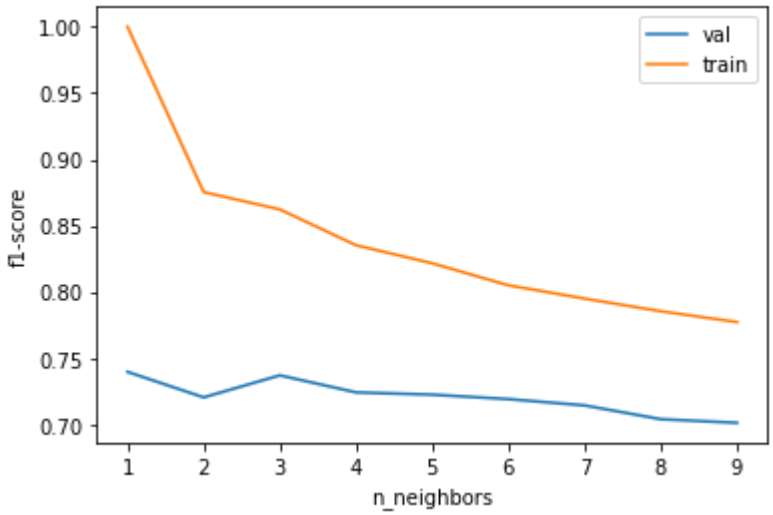
Agora aplicando todos os melhores parâmetros no nosso modelo podemos observar que no teste final tivemos uma melhora pequena nas nossas métricas, no entanto ainda sim conseguimos melhorar o modelo.

KNN:

.Parâmetros do KNN:

```
[ ]  def check_f1_test(clf,X_test,y_test):
       pred = clf.predict(X_test)
       print("f1",f1_score(pred,y_test,average="weighted"))
       print("recall",recall_score(pred,y_test,average="weighted"))
       print("precision",precision_score(pred,y_test,average="weighted"))
```

```
results_val = []
results_train = []
n_s = range(1,10)
for n_neighbors in n_s:
  clf = KNeighborsClassifier(n_neighbors = n_neighbors)
  clf.fit(X_train,y_train)
  pred = clf.predict(X_valid)
  results_val.append(f1_score(pred,y_valid,average="weighted"))
  pred = clf.predict(X_train)
  results_train.append(f1_score(pred,y_train,average="weighted"))

plt.plot(n_s,results_val,label='val')
plt.plot(n_s,results_train,label='train')
plt.ylabel("f1-score")
plt.xlabel("n_neighbors")
plt.legend()
```



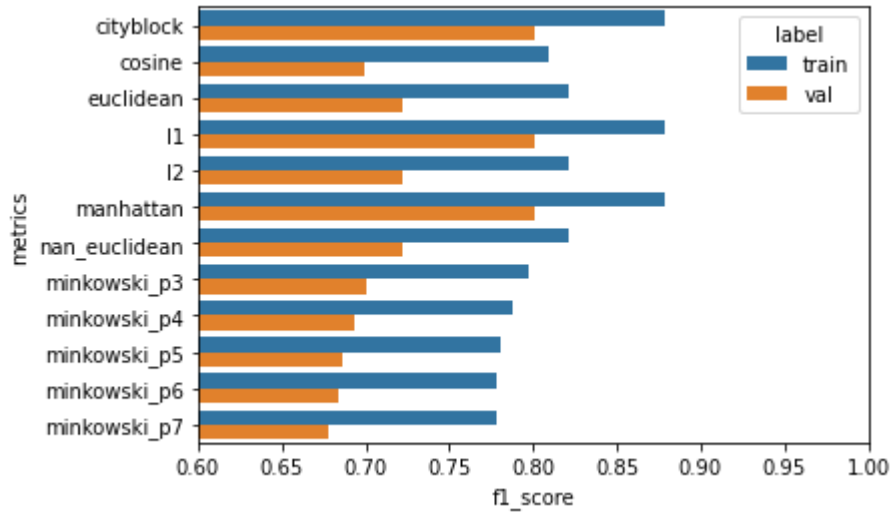
Observando o gráfica e analisando bem o modelo, podemos observar que com a fórmula de distância fixada como a distância euclidiana, o valor de `k` igual a 1 é o que apresenta a melhor métrica f1 e ainda consegue fazer as predições de forma mais simples.
Obs: como podemos constatar na documentação do sklearn, a métrica default é minkowski, mas como o padrão é 2 ela é equivalente a métrica euclidiana

. Testando outras métricas de distância:

```
[ ]  def make_df_metrics(methods,results_train,results_val):
       return pd.concat([pd.DataFrame({"x":methods,'y':results_train,'label':['train']*len(results_train)}),pd.DataFrame({"x":methods,'y':results_val,'label':['val']*len(results_val)})])
```

```
  results_val = []
  results_train = []
  metrics = ['cityblock','cosine','euclidean','l1','l2','manhattan','nan_euclidean']
  #'haversine'
  for x in metrics:
    clf = KNeighborsClassifier(metric = x)
    clf.fit(X_train,y_train)
    pred = clf.predict(X_valid)
    results_val.append(f1_score(pred,y_valid,average="weighted"))
    pred = clf.predict(X_train)
    results_train.append(f1_score(pred,y_train,average="weighted"))

  for x in range(3,8):
    metrics.append(f'minkowski_p{x}')
    clf = KNeighborsClassifier(metric = 'minkowski',p=x)
    clf.fit(X_train,y_train)
    pred = clf.predict(X_valid)
    results_val.append(f1_score(pred,y_valid,average="weighted"))
    pred = clf.predict(X_train)
    results_train.append(f1_score(pred,y_train,average="weighted"))

  df_metrics = make_df_metrics(metrics,results_train,results_val)
```

```
[ ]  sns.barplot(data=df_metrics,y='x',x='y',hue='label')
     plt.ylabel("metrics")
     plt.xlabel("f1_score")
     plt.xlim(0.6, 1.0)
```

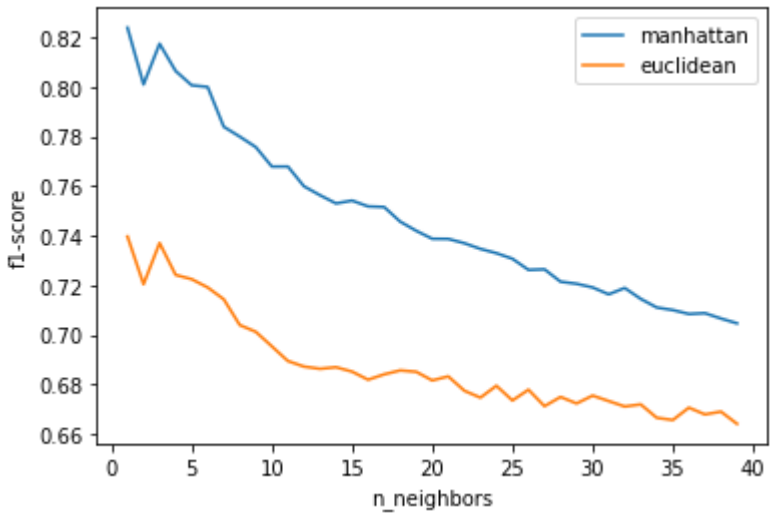



```
df_metrics.sort_values(by='y',ascending=False).groupby(by='label').head(4)
```

	x	y	label
0	cityblock	0.878461	train
3	l1	0.878461	train
5	manhattan	0.878461	train
2	euclidean	0.821596	train
5	manhattan	0.800712	val
3	l1	0.800712	val
0	cityblock	0.800712	val
2	euclidean	0.722539	val

Ao fixarmos o valor de `k` e variarmos as métricas de distância, podemos observar que tanto a distância de manhattan como a euclidiana apresentam bons resultados no nosso `dataset`, então nós fizemos uma análise mais profunda variando o `k` para essas duas distâncias.

```
manhattan_val = []
euclidean_val = []


n_s = range(1,40)
for n_neighbors in n_s:
  clf = KNeighborsClassifier(n_neighbors = n_neighbors,metric='manhattan')
  clf.fit(X_train,y_train)
  pred = clf.predict(X_valid)
  manhattan_val.append(f1_score(pred,y_valid,average="weighted"))

for n_neighbors in n_s:
  clf = KNeighborsClassifier(n_neighbors = n_neighbors,metric='euclidean')
  clf.fit(X_train,y_train)
  pred = clf.predict(X_valid)
  euclidean_val.append(f1_score(pred,y_valid,average="weighted"))

plt.plot(n_s,manhattan_val,label='manhattan')
plt.plot(n_s,euclidean_val,label='euclidean')

plt.ylabel("f1-score")
plt.xlabel("n_neighbors")
plt.legend()
```



Ao analisarmos o gráfico, podemos observar que para múltiplos valores de `k`, a distância de manhattan apresenta uma métrica f1 mais otimizada. Então utilizamos essa distância e o `k` igual a 1 para tentar otimizar o KNN.

```
clf = KNeighborsClassifier(n_neighbors = 1,metric='manhattan')
clf.fit(X_train,y_train)
check_f1_test(clf,X_test,y_test)
```

```
f1 0.828563428389104
recall 0.8293570949506563
precision 0.8292224138422384
```

Por fim, conseguimos otimizar ainda mais as métricas alterando os hiperparâmetros para os valores ótimos encontrados.

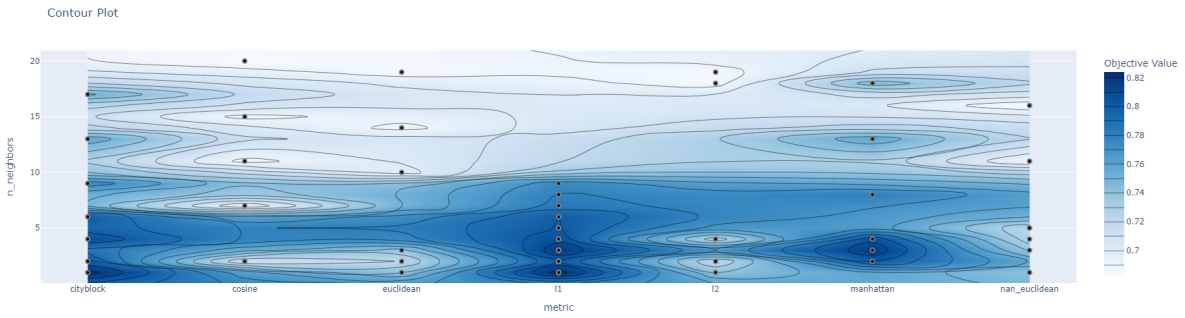
.Utilizando o Optuna para tentar encontrar os melhores parâmetros para utilizar no KNN:

```
metrics = ['cityblock','cosine','euclidean','l1','l2','manhattan','nan_euclidean']
def objective(trial):
    n_neighbors = trial.suggest_int("n_neighbors",1,20)

    metric = trial.suggest_categorical("metric",metrics)

    clf = KNeighborsClassifier(
                               n_neighbors=n_neighbors,
                               metric=metric
                               )
    clf.fit(X_train,y_train)
    pred = clf.predict(X_valid)
    return f1_score(pred,y_valid,average="weighted")

study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=100)

print(study.best_params)
```

```
{'n_neighbors': 1, 'metric': 'l1'}
```

Com o Optuna, obtivemos resultados semelhantes ao nosso estudo manual anterior, e ainda conseguimos plotar mais a importância dos hiperparâmetros no gráfico abaixo.

```
optuna.visualization.plot_contour(study, params=["metric", "n_neighbors"])
```



Podemos analisar visualmente o que foi feito no teste do Optuna o que foi feito anteriormente. As regiões mais escuras são aquelas que têm a melhor combinação de `k` e da métrica.

Conclusão

Podemos concluir então de que o estudo de cada hiperparâmetro afeta no uso final do modelo de classificação. Ao usarmos parâmetros que se comportam melhor com o `dataset` conseguimos obter métricas mais otimizadas em relação aos parâmetros padrões.

Em termos de comparação, o classificador da árvore de decisão teve um melhor desempenho quando comparado aos outros classificadores. Isso se deve ao fato desse modelo lidar bem com as divisões do espaço de features para cada um dos escrivãos do texto. Por outro lado, o algoritmo com o pior desempenho foi o de Naive Bayes. Esse modelo de classificação teve um desempenho ruim pois nem sempre assumir a independências da features irá gerar um bom resultado, as características de cada escrivão poderiam estar muito bem atribuídas de forma dependente uma das outras dentro do processo de escrita, o que faz com que o que assumimos no nosso modelo nem sempre seja verdade.