Tatiana Escovedo

85 Followers About Follow

Implementando um Modelo de Classificação no Scikit-Learn*

Tatiana Escovedo Mar 19 · 15 min read

O que é o Scikit-Learn?

O *Scikit-Learn* é uma das bibliotecas de *Machine Learning* mais conhecidas e utilizadas do *Python*, dentre as diversas existentes, elaborada em código aberto e desenvolvida para suportar e possibilitar o treino de diversas técnicas de estatística e Machine Learning, para aprendizagem supervisionada e não supervisionada.

Brucher começou a trabalhar no projeto como parte de sua tese de doutorado. Pesquisadores do INRIA (*Institut National de Recherche en Informatique et en Automatique*), assumiram a liderança do projeto, em 2010, lançando sua primeira release em fevereiro do mesmo ano. As posteriores releases tem sido publicadas em intervalos aproximados de 3 meses, havendo atualmente uma grande e próspera comunidade internacional que desenvolve melhorias e correções.

O *Scikit-Learn* fornece ferramentas importantes para os vários momentos do ciclo de projetos de *Machine Learning*, como:

- ***Datasets*:** disponibiliza alguns *datasets* que podem ser baixados para o projeto com poucos comandos, como o *dataset* **Iris**, um dos mais conhecidos da área de reconhecimento de padrões.
- **Pré-processamento de dados:** fornece diversas técnicas de preparação de dados, como normalização e *encoding*.
- **Modelos:** implementa diversos modelos de *Machine Learning*, tais como Regressão Linear, SVM e *Random Forest*, possibilitando o ajuste, avaliação e seleção do melhor modelo para o problema.

Sua API é uniforme, limpa e simplificada, além de possuir uma documentação online muito completa. Estes são exemplos de vantagens significativas para o profissional de *Machine Learning*, pois uma vez entendido o uso básico e a sintaxe para um determinado modelo, a mudança para um modelo diferente será muito simples. Os modelos de *Machine Learning* implementados no *Scikit-Learn* estão listados na sua documentação oficial. A seguir, é apresentado um resumo de suas características mais relevantes:

- **Consistência:** todos os objetos compartilham uma interface comum desenhada a partir de um conjunto limitado de métodos, com documentação consistente.
- **Inspeção:** todos os valores de parâmetros especificados são expostos como atributos públicos;
- **Hierarquia Limitada de Objetos:** somente algoritmos são representados por classes *Python*; os conjuntos de dados são representados em formatos padrão

- **Composição:** sempre que possível, as tarefas de *Machine Learning* são expressas como sequências de algoritmos mais fundamentais;
- **Padrões Sensíveis:** quando os modelos requerem parâmetros especificados pelo usuário, a biblioteca define um valor padrão apropriado.

Estes princípios fazem com que o *Scikit-Learn* seja de fácil utilização em termos práticos, assim que esses princípios básicos são entendidos. A seguir, apresentaremos rapidamente alguns conceitos sobre representação de dados, a *API Estimator* e validação de modelos no *Scikit-Learn*. Finalmente, teremos um exemplo prático para ilustrar todos os conceitos teóricos apresentados neste artigo.

Representação de dados

Os dados precisam estar representados de forma adequada para que o computador os entenda, e o principal objetivo de *Machine Learning* é a criação de modelos a partir de bases ou conjuntos de dados (*datasets*). O *Scikit-Learn* pode trabalhar com dados como tabelas, ou como matrizes de atributos (características ou *features*) e de valores de saída (valores alvo, *targets*, rótulos ou *labels*). A clássica representação bidimensional (2D) como tabela contém todos os exemplos (instâncias, registros). Nela, as linhas são compostas por colunas contendo os atributos e o valor de saída esperado.

Uma outra representação de dados, mais comumente utilizada no *Scikit-Learn* é o formato de matrizes. Nesta representação, geralmente utilizamos a variável **X** para o armazenamento dos atributos. Esta matriz terá a forma [n_exemplos, n_atributos] e frequentemente terá o formato de um arranjo *NumPy* ou um *DataFrame Pandas*, apesar de alguns modelos *Scikit-Learn* também aceitarem matrizes esparsas *SciPy*.

Ainda nesta representação, geralmente utilizamos a variável **Y** para representar a matriz unidimensional que armazenará os valores de saída, com a forma [n_exemplos]. Esta matriz geralmente terá o formato de um arranjo *NumPy* ou de uma série *Pandas*. Alguns estimadores tratam valores de múltiplos destinos na forma de arranjos bidimensionais do tipo [n_exemplos, n_destinos], mas também é possível trabalhar o caso do arranjo destino unidimensional, muito comum. Esta matriz irá conter números reais para problemas de Regressão, ou inteiros para problemas de Classificação (ou qualquer outro conjunto discreto de valores). Já para tarefas de aprendizagem não supervisionada, esta matriz não precisa ser especificada.

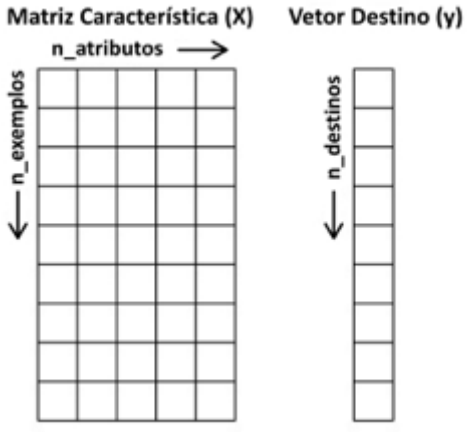


Figura 1 — Esquema visual da representação matricial.

A API Estimator

Os algoritmos de *Machine Learning* no *Scikit-Learn* sempre serão implementados utilizando a *API Estimator*, que em geral, segue as etapas:

1. Escolha uma classe de modelo importando a classe apropriada do *Scikit-Learn*.
2. Escolha os hiperparâmetros do modelo instanciando esta classe com os valores desejados.
3. Organize os dados em uma matriz de recursos **X** e vetor de destino **Y**.
4. Ajuste o modelo aos seus dados chamando o método ***fit()*** da instância do modelo.
5. Aplique o modelo aos novos dados:

No aprendizado supervisionado, geralmente estamos interessados na predição de rótulos para dados desconhecidos. Para tal, use o método ***predict()***.

No aprendizado não supervisionado, geralmente estamos interessados em descobrir associações ou propriedades dos dados usando. Para tal, use o método ***transform()*** ou o método ***predict()***.

Validação de Modelos

Quando vamos treinar e validar nosso modelo de aprendizado supervisionado, uma das alternativas seria utilizar a estratégia *train-test-split*, que consiste em separar nosso *dataset* em dois subconjuntos menores para serem utilizados como dados de treino e dados de validação (ou teste). Uma razão comumente utilizada nessa estratégia é

treinar o modelo e o conjunto de 30% seria utilizado para validar o modelo.

Uma alternativa a essa estratégia é a técnica de *Cross Validation*, ou Validação Cruzada. De forma simplificada, esta técnica consiste em dividir o conjunto de dados em um número **n** de subconjuntos (partições ou *folders*) contendo aproximadamente o mesmo número de elementos. Durante esta estratégia, escolhe-se 1 dos **n** *folders* para validação, treina-se o modelo nos **n-1** *folders* restantes, validando-se o modelo no *folder* escolhido inicialmente. Este processo é repetido, escolhendo-se um novo *folder* diferente dos que já foram escolhidos para a validação do modelo e treinando-o com os demais, e continua até que se tenha selecionado cada um dos *folders* para validação. O resultado será um *array* contendo **n** medidas resultantes da avaliação, do qual geralmente extraímos a sua média para calcular o desempenho do modelo. As técnicas de validação cruzada mais utilizadas são as que utilizam 5 ou 10 partições. Apesar da validação cruzada em geral ser mais adequada para validação de modelos do que a estratégia *train-test-split*, ela pode ser difícil de ser utilizada quando dispomos de poucos dados para treinamento.

Prática: Projeto de Classificação Binária usando o Scikit-Learn

A seguir, apresentamos um exemplo prático de um Projeto de Classificação Binária para demonstrar algumas das principais funcionalidades do *Scikit-Learn*. Passaremos pelas etapas de Carga de Dados, Análise de dados, Pré-processamento de dados, Construção de modelos de Classificação e Finalização do modelo. Obviamente, este exemplo não tem o intuito de ser exaustivo, uma vez que existem inúmeros outros recursos disponíveis na biblioteca que não serão demonstrados. Sugerimos que você explore a documentação do *Scikit-Learn* para se aprofundar em outros exemplos. Você pode acompanhar o exemplo digitando os comandos na IDE de sua preferência

Passo 1 — Definição do Problema

O *dataset* usado neste projeto será o ***Pima Indians Diabetes***, proveniente originalmente do Instituto Nacional de Diabetes e Doenças Digestivas e Renais. Seu objetivo é prever se um paciente desenvolverá ou não diabetes, com base em certas medidas de diagnóstico médico. Este *dataset* é um subconjunto do *dataset* original e está disponível tanto no *Kaggle* como em diversas URLs. Nele, todos os pacientes são mulheres com pelo menos 21 anos de idade e de herança indígena Pima.

paciente teve, seu IMC, nível de insulina, idade e assim por diante. Para este problema, você irá importar os seguintes pacotes:

```
# Imports
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import ConfusionMatrixDisplay

from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix

from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

Passo 2 — Carga de Dados

Usando *Pandas*, iremos importar o dataset diretamente de uma URL onde ele se encontra disponível, e o carregaremos para a variável *dataset*, conforme mostrado no código abaixo. Iremos especificar nomes para as colunas e também indicaremos que não há informações de cabeçalho (*header*), para que o primeiro exemplo não seja considerado o nome da coluna. Com o *dataset* carregado, iremos explorá-lo um pouco.

```
# Carrega arquivo csv usando Pandas usando uma URL
# Informa a URL de importação do dataset
```

```
# Informa o cabeçalho das colunas
colunas = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

# Lê o arquivo utilizando as colunas informadas
dataset = pd.read_csv(url, names=colunas, skiprows=0, delimiter=',')
```

Passo 3 — Análise de Dados

Como o foco deste artigo é a biblioteca *Scikit-Learn*, não iremos realizar muitas análises deste *dataset*. Vamos apenas exibir as dimensões do *dataset*, os tipos de cada atributo e as primeiras linhas.

```
>>> # dimensões do dataset
>>> print(dataset.shape)
(768, 9)

>>> # tipos de cada atributo
>>> print(dataset.dtypes)
preg int64
plas int64
pres int64
skin int64
test int64
mass float64
pedi float64
age int64
class int64
dtype: object

>>> # primeiras linhas do dataset
>>> print(dataset.head())
preg plas pres skin test mass pedi age class
0 6 148 72 35 0 33.6 0.627 50 1
1 1 85 66 29 0 26.6 0.351 31 0
2 8 183 64 0 0 23.3 0.672 32 1
3 1 89 66 23 94 28.1 0.167 21 0
4 0 137 40 35 168 43.1 2.288 33 1
```

É uma boa prática usar um conjunto de teste (na literatura também chamado de conjunto de validação), uma amostra dos dados que não será usada para a construção do modelo, mas somente no fim do projeto para confirmar a precisão do modelo final. É um teste que podemos usar para verificar o quão boa foi a construção do modelo, e para nos dar uma ideia de como o modelo irá performar nas estimativas em dados não vistos. Usaremos 80% do conjunto de dados para modelagem e guardaremos 20% para teste, usando a estratégia *train-test-split*, já explicada anteriormente. Primeiramente, iremos sinalizar quais são as colunas de atributos (**X**, as 8 primeiras) e qual é a coluna das classes (**Y** — a última). Em seguida, especificaremos o tamanho do conjunto de teste desejado e uma semente (para garantir a reprodutibilidade dos resultados). Finalmente, faremos a separação dos conjuntos de treino e teste por meio do comando *train_test_split*, que retornará 4 estruturas de dados: os atributos e classes para o conjunto de teste e os atributos e classes para o conjunto de treino.

OBS: Nesta etapa, poderíamos, se necessário, realizar algumas operações de preparação de dados, como, tratamento de valores *missings* (faltantes), limpeza de dados, transformações, entre outras.

```
# Separação em conjuntos de treino e teste
array = dataset.values
X = array[:, 0:8].astype(float)
Y = array[:, 8]
test_size = 0.20
seed = 7
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=test_size,
random_state=seed)
```

Passo 5 — Modelos de Classificação

Passo 5.1 — Criação e avaliação de modelos: linha base

Não sabemos de antemão quais modelos desempenharão bem neste conjunto de dados. Assim, será usada a validação cruzada *10-fold* (já detalhada anteriormente) e serão avaliados diversos modelos usando a métrica de acurácia. Inicialmente configuram-se os parâmetros de número de *folds* e métrica de avaliação.

```
scoring = 'accuracy'
```

Em seguida, vamos criar uma linha base de desempenho para esse problema, verificando vários modelos diferentes com suas configurações padrão. Utilizaremos os modelos de Regressão Logística, K-vizinhos mais próximos (KNN), Árvores de Classificação (CART), *Naive Bayes* (NB) e Máquinas de vetores de suporte (SVM).

```
# Criação dos modelos
models = []
models.append(('LR', LogisticRegression(solver='newton-cg')))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
```

Agora vamos comparar os resultados modelos criados, treinando-os com os dados do conjunto de treino e utilizando a técnica de validação cruzada. Vamos definir, inicialmente, uma semente global para este bloco. Para cada um dos modelos criados, executaremos a validação cruzada e, em seguida, exibiremos a acurácia média e o desvio padrão de cada um.

```
np.random.seed(7) # definindo uma semente global

# Avaliação dos modelos
results = []
names = []
for name, model in models:
kfold = KFold(n_splits=num_folds)
cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
results.append(cv_results)
names.append(name)
msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
print(msg)
```

Os resultados dos modelos foram:

```
CART: 0.687335 (0.052398)
NB: 0.750820 (0.050575)
SVM: 0.757271 (0.047915)
```

Estes resultados sugerem que a Regressão Logística, o *Naive Bayes* e o SVM têm potencial de serem bons modelos, porém, vale observar que estes são apenas valores médios de acurácia, sendo prudente também observar a distribuição dos resultados de cada *fold* da validação cruzada. Faremos isto comparando os modelos usando *boxplots*.

```
# Comparação dos modelos
fig = plt.figure()
fig.suptitle('Comparação dos Modelos')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

A Figura 2 ilustra o resultado da execução deste bloco de código:

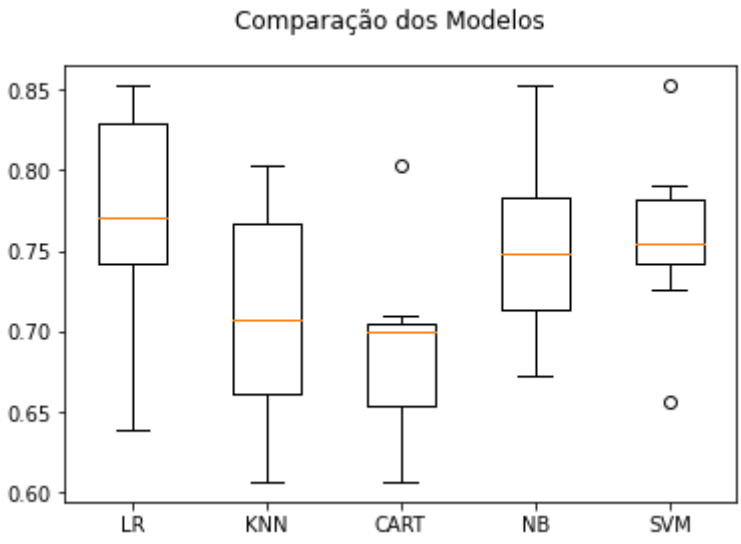


Figura 2 — Resultado da comparação dos modelos.

A seguir, repetiremos este processo usando uma cópia padronizada do conjunto de dados de treinamento para verificar se as diferentes distribuições dos dados brutos impactam negativamente a habilidade de alguns modelos.

Como suspeitamos que as diferentes distribuições dos dados brutos possam impactar negativamente o resultado de alguns modelos, vamos agora utilizar uma cópia padronizada do *dataset*: os dados serão transformados de modo que cada atributo tenha média 0 e desvio padrão 1, e executaremos novamente os mesmos modelos.

O *Scikit-Learn* fornece um poderoso utilitário de *pipeline* para ajudar a automatizar os fluxos de trabalho de *Machine Learning*. Os *pipelines* permitem que uma sequência linear de transformações de dados seja encadeada, culminando em um processo de modelagem que pode ser avaliado em seguida. O objetivo é garantir que todas as etapas do *pip*eline sejam restritas aos dados disponíveis para a avaliação, como o conjunto de dados de treinamento ou cada *fold* do procedimento de validação cruzada. Assim, vamos usar *pipelines* que padronizam os dados e já constroem o modelo para cada *fold* de teste de validação cruzada. Dessa forma, poderemos obter uma estimativa justa de como cada modelo treinado com dados padronizados irá se comportar com dados não vistos.

OBS: Você pode aprender mais sobre os *pipelines* do *Scikit-Learn* consultando a documentação oficial .

```
np.random.seed(7) # definindo uma semente global

# Padronização do dataset
pipelines = []
pipelines.append(('ScaledLR', Pipeline([('Scaler', StandardScaler()), ('LR',
LogisticRegression(solver='newton-cg'))])))
pipelines.append(('ScaledKNN', Pipeline([('Scaler', StandardScaler()),('KNN',
KNeighborsClassifier())])))
pipelines.append(('ScaledCART', Pipeline([('Scaler', StandardScaler()),('CART',
DecisionTreeClassifier())])))
pipelines.append(('ScaledNB', Pipeline([('Scaler', StandardScaler()),('NB',
GaussianNB())])))
pipelines.append(('ScaledSVM', Pipeline([('Scaler', StandardScaler()),('SVM',
SVC())])))
results = []
names = []
for name, model in pipelines:
```

```
results.append(cv_results)
names.append(name)
msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
print(msg)
```

Os resultados dos modelos foram:

```
ScaledLR: 0.778424 (0.061895)
ScaledKNN: 0.718297 (0.071563)
ScaledCART: 0.684082 (0.052059)
ScaledNB: 0.750820 (0.050575)
ScaledSVM: 0.752565 (0.064295)
```

Analisando os resultados, vemos que aparentemente a padronização de dados não fez muita diferença nos valores de acurácia. Podemos escolher alguns modelos para verificar se outras configurações de hiperparâmetros geram resultados melhores. Faremos isto a seguir.

Passo 5.3 — Ajuste dos modelos

Passo 5.3.1 — Ajuste do KNN

Vamos começar ajustando hiperparâmetros como o número de vizinhos e as métricas de distância para o KNN. Para tal, tentaremos todos os valores ímpares de **k** entre 1 a 21 e as métricas de distância euclidiana, *manhattan* e *minkowski*. Cada valor de **k** e de distância será avaliado usando a validação cruzada *10-fold* no conjunto de dados padronizado, que mostrou resultados um pouco melhores do que os dados originais.

O *Scikit-Learn* fornece um recurso para avaliar facilmente diversas variações de hiperparâmetros de algoritmos, a função *GridSearchCV*. Para tal, vamos utilizar esta função passando como parâmetros o modelo, o conjunto de parâmetros que queremos variar, a métrica de avaliação utilizada e o número de *folds* utilizados na validação cruzada.

OBS: Você pode aprender mais sobre a funcionalidade *GridSearchCV* do *Scikit-Learn* na documentação oficial .

```
# Tuning do KNN
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)

k = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
distancias = ["euclidean", "manhattan", "minkowski"]
param_grid = dict(n_neighbors=k, metric=distancias)

model = KNeighborsClassifier()
kfold = KFold(n_splits=num_folds)

grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring,
cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
print("Melhor: %f usando %s" %(grid_result.best_score_, grid_result.best_params_))

means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
print("%f (%f): %r" % (mean, stdev, param))
```

Os resultados foram (exibindo apenas as primeiras linhas, por motivos de legibilidade):

```
Melhor: 0.747329 usando {'metric': 'manhattan', 'n_neighbors': 17}
0.702036 (0.075358): {'metric': 'euclidean', 'n_neighbors': 1}
0.734611 (0.068286): {'metric': 'euclidean', 'n_neighbors': 3}
...
```

Os resultados mostram que a melhor configuração encontrada utiliza distância de *manhattan* e k = 17. Isto significa que o algoritmo fará previsões usando as 17 instâncias mais semelhantes.

Passo 5.3.2 — Ajuste do SVM

Iremos ajustar dois dos principais hiperparâmetros do algoritmo SVM: o valor de C (o quanto flexibilizar a margem) e o tipo de *kernel* utilizado. No *Scikit-Learn*, o padrão

hiperparâmetros, e cada combinação de valores será avaliada usando a função *GridSearchCV*, além de aplicarmos os modelos nos dados padronizados, como fizemos anteriormente para o KNN. Neste caso, não iremos utilizar a versão padronizada dos dados, mas sim a versão original, que produziu melhores resultados.

```
np.random.seed(7) # definindo uma semente global

# Tuning do SVM

c_values = [0.1, 0.5, 1.0, 1.5, 2.0]
kernel_values = ['linear', 'poly', 'rbf', 'sigmoid']
param_grid = dict(C=c_values, kernel=kernel_values)

model = SVC()
kfold = KFold(n_splits=num_folds)

grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring,
cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
print("Melhor: %f com %s" % (grid_result.best_score_,grid_result.best_params_))

means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
print("%f (%f): %r" % (mean, stdev, param))
```

Os resultados foram (exibindo apenas algumas linhas por motivos de legibilidade):

```
Melhor: 0.776838 usando {'C': 0.5, 'kernel': 'linear'}
0.771920 (0.053452): {'C': 0.1, 'kernel': 'linear'}
0.745822 (0.042116): {'C': 0.1, 'kernel': 'poly'}
...
```

Podemos ver que a configuração que alcançou a maior acurácia foi o modelo que utilizou *kernel* linear e C = 0,1. Apesar desta acurácia ter sido um pouco melhor do que

Passo 6 — Finalização do Modelo

Até aqui, verificamos que a Regressão Logística foi o modelo que mostrou melhor acurácia para o problema. A seguir, finalizaremos este modelo, treinando-o em todo o conjunto de dados de treinamento (sem validação cruzada) e faremos predições para o conjunto de dados de teste que foi separado logo no início do exemplo, a fim de confirmarmos nossas descobertas.

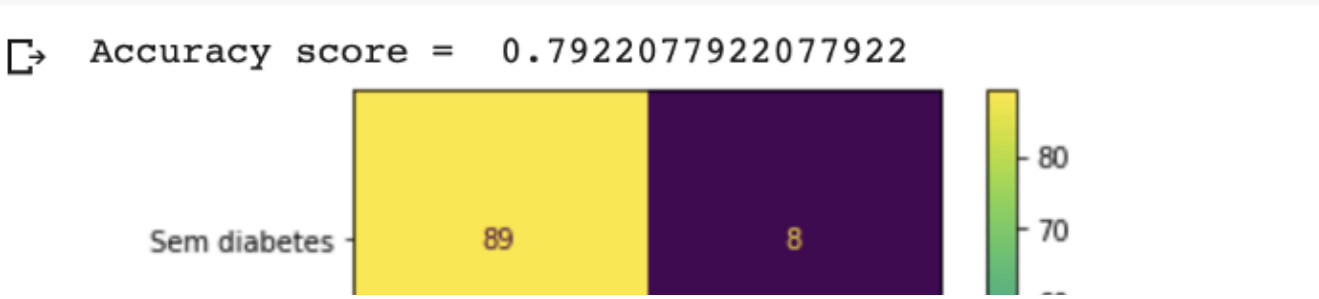
Como para este modelo a padronização dos dados de entrada não produziu resultados diferentes, iremos utilizar a versão original do *dataset*. Exibiremos a acurácia, a matriz de confusão e um relatório de Classificação com diversas métricas, disponível com a utilização da função *classification_report*, como ilustra a Figura 3.

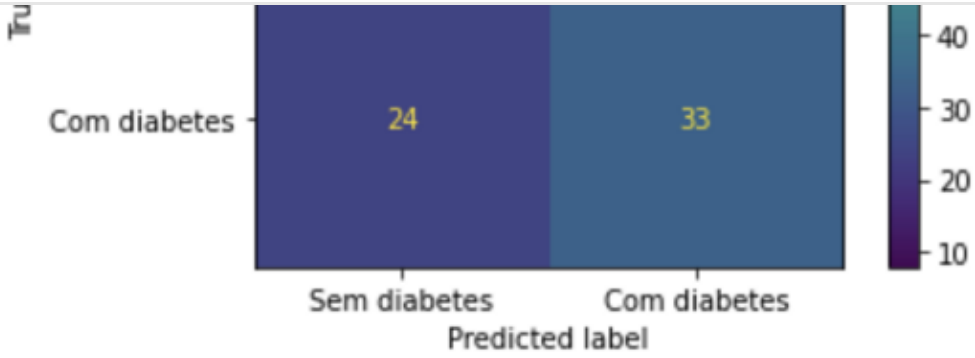
```
np.random.seed(7) # definindo uma semente global

# Preparação do modelo
model = LogisticRegression(solver='newton-cg')
model.fit(X_train, Y_train)

# Estimativa da acurácia no conjunto de teste
predictions = model.predict(X_test)
print("Accuracy score = ", accuracy_score(Y_test, predictions))

# Matriz de confusão
cm = confusion_matrix(Y_test, predictions)
labels = ["Sem diabetes", "Com diabetes"]
cmd = ConfusionMatrixDisplay(cm, display_labels=labels)
cmd.plot(values_format="d")
plt.show()
print(classification_report(Y_test, predictions, target_names=labels))
```





	precision	recall	f1-score	support
Sem diabetes	0.79	0.92	0.85	97
Com diabetes	0.80	0.58	0.67	57
accuracy			0.79	154
macro avg	0.80	0.75	0.76	154
weighted avg	0.79	0.79	0.78	154

Figura 3 — Resultados do modelo.

Por meio do conjunto de testes, verificamos que alcançamos uma acurácia de 79,22% em dados não vistos. Este resultado foi ainda melhor do que as nossas expectativas, pois no treinamento alcançamos a acurácia de 78%. Valores semelhantes são esperados quando este modelo estiver executando em produção e fazendo predições para novos dados.

É importante ressaltar que este exemplo prático não buscou ser exaustivo, apresentando apenas uma parte dos muitos recursos disponíveis na biblioteca *Scikit-Learn* para que você compreenda em linhas gerais o seu funcionamento. Por exemplo, poderíamos ter realizado mais análises exploratórias dos dados, testado outras operações de pré-processamento de dados, utilizado outros valores de hiperparâmetros e, ainda, experimentado outros modelos de classificação, como os *ensembles*. Recomendamos que você explore a documentação disponível e incremente este exemplo com novas possibilidades.

**Este texto foi escrito em colaboração com Carlos Eduardo Silva Castro e Cassius T. C. Mendes, e faz parte do livro colaborativo Jornada Python — Uma Jornada imersiva na aplicabilidade de uma das mais poderosas linguagens de programação do mundo, previsto para lançamento ainda em 2021 pela editora Brasport.*

Machine Learning Scikit Learn Aprendizado De Máquina Classificação Python