Capítulo 3 – Classificação

Configuração

Importa alguns módulos comuns para garantir que o MatplotLib plote as figuras inline.

```
# importações comuns
import numpy as np
import os

# para tornar a saída deste notebook estável em todas as execuções
np.random.seed(42)

# Para plotar figuras padronizadas
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

MNIST

O conjunto de dados MNIST, é composto por 70.000 imagens de dígitos escritos à mão por estudantes do ensino médio e funcionários do US Census Bureau. Cada imagem é rotulada com o dígito que a representa.

```
# Importando o conjunto de dados MNIST
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
# desde o Scikit-Learn 0.24, fetch_openml() retorna um DataFrame do Pandas por padrão. Para evitar isso, usamos as_
mnist.keys()
```

Os conjuntos de dados carregados pelo Scikit-Learn geralmente possuem uma estrutura similar ao dicionário, incluindo:

- Uma chave **DESCR** que descreve o conjunto de dados;
- Uma chave **DATA** de dados que contém um array com uma linha por Instância em uma coluna por característica;
- Uma chave **TARGET** (alvo) contendo um Array com os rótulos.

Vejamos estes a arrays:

```
X, y = mnist["data"], mnist["target"]
X.shape # Dados
```

```
y.shape # Target
```

São 70.000 imagens, cada imagem possui 784 características (cada imagem tem 28 x 28 pixels). Cada característica representa a intensidade de um pixel, de 0 (preto) a 255 (branco).

Vamos dar uma olhada em um dígito do conjunto de dados. Utilizando a função *imshow()* do Matplotlib, você só precisa pegar um vetor de característica de uma Instância, remodelá-lo para um array de 28x28 e exibi-lo:

```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

um_digito = X[0]
um_digito_imagem = um_digito.reshape(28, 28)
plt.imshow(um_digito_imagem, cmap=mpl.cm.binary)

plt.show()
```

O dígito da imagem parece um cinco, é isso o que o rótulo nos diz:

```
y[0]
```

```
type(y[0])
```

```
# convertendo para inteiro
y = y.astype(np.uint8)
```

```
y[0]
```

```
type(y[0])
```

```
# Função que recebe um conjunto e plot a imagem
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = mpl.cm.binary,
               interpolation="nearest")
    plt.axis("off")
```

```
# Exemplo da chamada da função para plotar 4 digitod
for i in range(4):
  plot_digit(X[i])
  plt.show()
```

Criando o Conjunto de Treinamento e o Conjunto de Teste

O conjunto de dados MNIST já está dividido em um conjunto de Treinamento (as primeiras 60 mil imagens) e um conjunto de teste (as últimas 10 mil imagens).

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Vamos embaralhar o conjunto de Treinamento, isso garantirá que todos os subconjuntos da validação cruzada sejam semelhantes (sem que haja subconjunto faltando algum dígito).

```
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

Treinando um classificador binário

Vamos simplificar um problema por hora e tentar apenas identificar um dígito, por exemplo, o número 5. Este "5-detector" será um exemplo de classificador binário capaz de distinguir apenas entre duas classes: 5 e não-5. Vamos criar os vetores-alvo para esta tarefa de classificação:

```
1 # Verdadeiro para todos os 5s e falso para os outros dígitos
2 y_train_5 = (y_train == 5)
3 y_test_5 = (y_test == 5)
```

Agora vamos escolher um classificador e treiná-lo. Vamos começar usando um classificador de **Gradiente Descendente Estocástico(SGD)**, este classificador tem a vantagem de conseguir lidar eficientemente com conjuntos de dados muito grandes (o que também o torna adequado para aprendizado online).

```
1 # Vamos criar um SGDClassifier e treiná-lo em todo o conjunto de treinamento
2 from sklearn.linear_model import SGDClassifier
3 sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
4 sgd_clf.fit(X_train, y_train_5)
```

Observações:

- Você deve definir o parâmetro `random_state` se desejar resultados reprodutíveis.
- alguns hiperparâmetros terão um valor padrão diferente em versões futuras do Scikit-Learn, como `max_iter` e `tol`. Para ser à prova de futuro, definimos explicitamente esses hiperparâmetros para seus valores padrão futuros.

Usando o classificador treinado para detectar imagens do número 5:

```
1 sgd_clf.predict([um_digito])
```

```
1 if sgd_clf.predict([um_digito]):
2   print ('5')
3 else:
4   print ('nao-5')
```

Avaliando o classificador treinado para detectar imagens do número 5:

```
1 sgd_clf.score(X_test, y_test_5)
```

Exercício: tentar um classificador binário de outro número.

Medidas de desempenho

Medindo a precisão usando validação cruzada

Usaremos a função *cross_val_score()* para avaliar a **Acurácia** do modelo SGDClassifier com a utilização da **validação cruzada k-fold** com três partes (cv=3):

A **Acurácia** (taxa global de sucesso) é medido pela quantidade de acertos pelo número total de possibilidades (**classificações corretas/número total de classificações**). É a proporção de predições corretas, sem considerar o que é positivo e o que é negativo.

$$ACC = \frac{VP+VN}{VP+FP+VN+FN}$$

```
1 from sklearn.model_selection import cross_val_score
2 cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

Observação: Muito bom! acima de 95% de acurácia (taxa das previsões corretas) em todas as partes da validação cruzada. Mas, será tão bom mesmo? Vejamos um classificador muito fraco que apenas classifica cada imagem na classe "não-5":

Classificador bobo (dummy)

```
1 from sklearn.base import BaseEstimator
2 class Never5Classifier(BaseEstimator):
3     def fit(self, X, y=None):
4         pass
5     def predict(self, X):
6         return np.zeros((len(X), 1), dtype=bool)
```

```
1 never_5_clf = Never5Classifier()
2 cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

Observações:

Sim, mais de 90% de acurácia! Isso ocorre porque apenas cerca de 10% das imagens são "5", então, se o classificador sempre adivinhar que uma imagem não é 5, ele estará certo cerca de 90% das vezes.

Isso demonstra porque acurácia para os classificadores geralmente não é a medida preferencial de desempenho, especialmente quando você estiver lidando com conjuntos de dados assimétricos, ou seja, quando algumas classes forem muito mais frequente do que outras .

Matriz de Confusão

Uma maneira de avaliar bem melhor desempenho de um classificador é utilizar uma **Matriz de Confusão**. A ideia geral é contar o número de vezes que as distâncias da classe A são classificadas como classe B.

Para calcular a matriz de confusão primeiro você precisa ter um conjunto de previsões para que possam ser comparadas com os alvos reais. Para isso utilize a função *cross_val_predict()*.

Assim como a função *cross_val_score*, a função *cross_val_predict* realiza a validação cruzada k-fold, mas, em vez que retornar as pontuações da avaliação, ela retorna as previsões feitas em cada parte do teste.

```
1 # Criando o conjunto de previsões
2 from sklearn.model_selection import cross_val_predict
3 y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Utilize a função confusion_matrix() para obter a matriz de confusão. Apenas passe as classe-alvo (y_train_5) e as classes previstas (y_train_pred):

```
1 # Criando a Matriz de Confusão
2 from sklearn.metrics import confusion_matrix
3 confusion_matrix(y_train_5, y_train_pred)
```

Na matriz de confusão, cada **linha** representa uma **Classe Real**, enquanto cada **coluna** representa uma **Classe Predita**.

A primeira linha desta matriz considera imagens **não-5** (a **classe negativa**):

1. corretamente classificadas como **não-5** (**Verdadeiros Negativos**)
2. erroneamente classificadas com **5** (**Falsos Positivos**)

A segunda linha considera as imagens dos **5** (a **Classe Positiva**):

1. erroneamente classificadas como **não-5** (**Falsos Negativos**)
2. corretamente classificadas como **5** (**Verdadeiros Positivos**).

		Classe Predita	
		Negativa	Positiva
Classe Real	Negativa	VN	FP
	Positiva	FN	VP

Um classificador perfeito teria apenas Verdadeiros Negativos e Verdadeiros Positivos, então sua matriz de confusão teria valores diferentes de zero somente na sua diagonal principal (superior esquerda para inferior direita):

```
1 y_train_perfect_predictions = y_train_5  # caso tivessemos alcançado perfeição
2 confusion_matrix(y_train_5, y_train_perfect_predictions)
```

Precisão

A **Precisão** corresponde ao percentual de objetos positivos sobre o total predito de positivos.

$$\Pr = \frac{VP}{VP+FP}$$

```
1 # Calculo da Precisão com o método precision_score()
2 from sklearn.metrics import precision_score
3 precision_score(y_train_5, y_train_pred)
```

```
1 # Calculo da Precisão usando a Matriz de Confusão
2 cm = confusion_matrix(y_train_5, y_train_pred)
3 cm[1, 1] / (cm[0, 1] + cm[1, 1]) # VP/(VP+FP)
```

Recall

Recall (TVP, Sensibilidade ou Revocação) corresponde ao percentual de objetos positivos classificados corretamente como positivos.

$$TVP = \frac{VP}{VP + FN}$$

```
1 # Calculo do Recall com o método recall_score()
2 from sklearn.metrics import recall_score
3 recall_score(y_train_5, y_train_pred)
```

```
1 # Calculo do Recall usando a Matriz de Confusão
2 cm[1, 1] / (cm[1, 0] + cm[1, 1]) # VP/(VP+FN)
```

Observação: Agora, o classificador 5-detector não parece tão bom quando verificamos sua precisão/recall. Quando ele afirma que uma imagem representam um cinco, ele está correto menos que 75% das vezes (Precisão). Além disso, ele só detecta cerca de 80% dos 5 (Recall).

Medida-F

A **Medida-F** (Score-F) combina Precisão e Recall (TVP, Sensibilidade ou Revocação) de modo a trazer um número único que indique a qualidade geral do seu modelo.

$$F = \frac{2 \times Pr \times Re}{(Pr + Re)}$$

A Medida-F é a média harmônica da precisão e revocação. Como resultado, o classificador só obterá um Score-F alto se ambas as medidas foram altas.

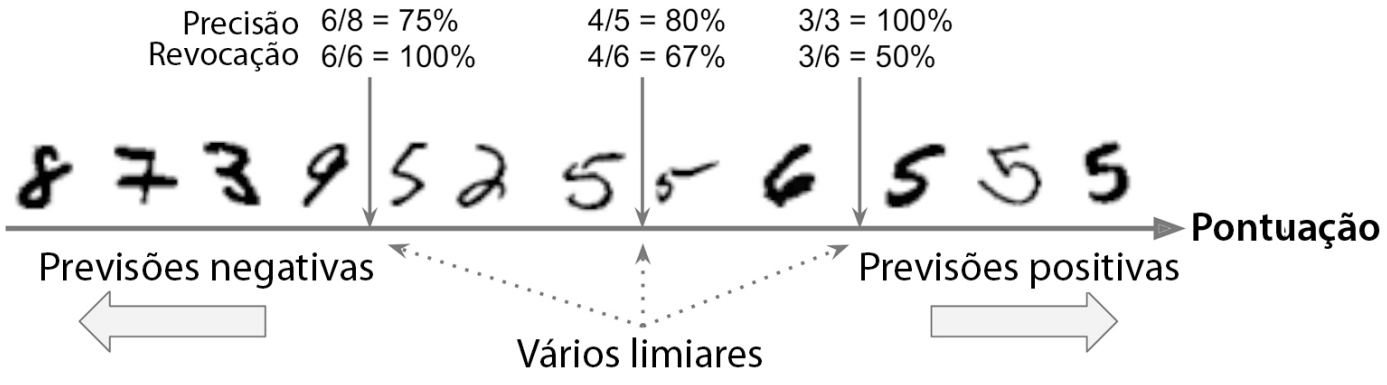
```
1 # Calculo da Medida-F com o método f1_score()
2 from sklearn.metrics import f1_score
3 f1_score(y_train_5, y_train_pred)
```

```
1 # Calculo da Medida-F usando a Matriz de Confusão
2 cm[1, 1] / (cm[1, 1] + (cm[1, 0] + cm[0, 1]) / 2) # 2*Pr*Re/(Pr+Re)
```

Relação Precisão/Revocação

A **Media-F** favorece classificadores com precisão e revogação similares. Isso nem sempre é o que você quer: em alguns contextos, sua preocupação é principalmente com a precisão, outros, você se preocupa com a revocação.

Infelizmente, não é possível ter os dois: aumentar a precisão reduz a revogação e vice-versa, isso é chamado de **Compensação da Precisão/Revocação**.



O Scikit-Learn não permite que você defina o limiar diretamente, mas lhe dá acesso as pontuações de decisão que ele utiliza para fazer previsões. Em vez de chamar o método *predict()* do classificador, você pode chamar o método *decision_function()*

que retorna uma pontuação para cada instância e, em seguida, fazer previsões com base nessas pontuações utilizando qualquer limiar desejado:

```
# Usando o método decision_function() para retornar uma pontuação para cada instância
y_scores = sgd_clf.decision_function([um_digito])
y_scores
```

```
threshold = 0
y_um_digito_previsto = (y_scores > threshold)
y_um_digito_previsto
```

```
if y_um_digito_previsto:
  print ('5')
else:
  print ('nao-5')
```

Observação: O SGDClassifier utiliza um limiar igual a zero, então, o código anterior retorna o mesmo resultado que o método *predict()* isto é **True**, aumentaremos o limiar:

```
threshold = 8000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

```
if y_some_digit_pred:
  print ('5')
else:
  print ('nao-5')
```

Observação: Isso confirma que aumentar o limiar diminui a revocação. A imagem realmente representa um 5, e o classificador o detecta quando o limiar é 0, mas, o perde quando o limiar sobe para 8000.

Então, como você pode decidir qual limiar utilizar? Para isso, utilizando novamente a função cross_val_predict(), você precisará primeiro obter as pontuações de todas as instâncias no conjunto de treinamento, mas desta vez especificando que deseja que ela retorne as pontuações da decisão em vez das previsões:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

Agora, utilizando a função *precision_recall_curve* com essas pontuações, você pode calcular a precisão e a revocação para todos os limiares possíveis:

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Apesar do código ser extenso, é importante ver as curvas

```
# Função para criação do Gráfico precision_recall_vs_threshold
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    # Configuração do Gráfico
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.legend(loc="center right", fontsize=16)
    plt.xlabel("Threshold", fontsize=16)
    plt.ylabel('Precision', fontsize=16)
    plt.grid(True)
    plt.axis([-50000, 50000, 0, 1])

# Obtendo os valores de recall e limiar para precisão de 90%
```

```
recall_90_precision = recalls[np.argmax(precisions >= 0.90)] # revocação da precisão de >= 90%
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)] # limiar da precisão de >= 90%

# Chamando a função para criação do Gráfico
plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)

# apontando a precisao e a revocação
plt.plot([threshold_90_precision, threshold_90_precision], [0., 0.9], "r:") # a precisao de 90% (linha vertical)
plt.plot([-50000, threshold_90_precision], [0.9, 0.9], "r:") # a precisao de 90%  (linha HS)
plt.plot([-50000, threshold_90_precision], [recall_90_precision, recall_90_precision], "r:")# a revocacao da precis

# marcando com bolinha vermelha
plt.plot([threshold_90_precision], [0.9], "ro") # a precisao de 90%
plt.plot([threshold_90_precision], [recall_90_precision], "ro") # a revocacao da precisao de 90%

plt.show()
```

Observação: Agora, você pode selecionar o valor do limiar que dá a melhor compensação de precisão/revocação para sua tarefa.

```
(y_train_pred == (y_scores > 0)).all() # o classificador que busca só acuracia usa limiar 0
```

Outra maneira de selecionar uma boa compensação de precisão/revocação é plotar a precisão diretamente contra a revogação:

```
# Função para criação do Gráfico da curva precision_vs_recall (PR)
def plot_precision_vs_recall(precisions, recalls):
    # Configuração do Gráfico
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])
    plt.grid(True)

# Chamando a função para criação do Gráfico
plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)

# apontando a precisao e a revocação
plt.plot([recall_90_precision, recall_90_precision], [0., 0.9], "r:") # a precisao de 90% (linha vertical)
plt.plot([0.0, recall_90_precision], [0.9, 0.9], "r:") # a precisao de 90%  (linha horizontal)
plt.plot([recall_90_precision], [0.9], "ro") # marcando com bolinha vermelha a precisao de 90%

plt.show()
```

Observação: É possível ver que a precisão realmente começa a diminuir acentuadamente em torno de 80% de revogação. Provavelmente você selecionará uma compensação de precisão/revocação antes dessa queda, por exemplo, cerca de 60% de revogação.

Então, vamos supor que você almeja 90% de precisão. Você procura a primeira plotagem e descobre que precisa utilizar um limiar de cerca de np.argmax(precisions >= 0.90).

```
np.argmax(precisions >= 0.90) # Obtendo a Precisão >=90
```

```
# Obtendo o valor do limiar
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
threshold_90_precision
```

Em vez de chamar o método *predict()* do classificador, você pode apenas executar este código para fazer previsões:

```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

Avaliando a Precisão e o Recall dessas previsões:

```
# Obtendo a Precisão
precision_score(y_train_5, y_train_pred_90)
```

```
# Obtendo o Recall
recall_score(y_train_5, y_train_pred_90)
```

A Curva ROC

A **curva das Características Operacionais do Receptor (ROC)**, é outra ferramenta comum utilizada com classificadores binários.

É muito semelhante a curva de precisão/revocação, mas, em vez de plotar precisão x revocação, a curva ROC plota a **Taxa de Verdadeiros Positivos (TPR, Recall ou Revocação) x Taxa de Falsos Positivos (FPR)**. O FPR é a razão de estâncias negativas incorretamente classificadas como positivas. O FPR é igual a **1 - a Taxa de Verdadeiros Negativos (TNR ou Especificidade)**. A Especificidade é a razão de instâncias negativas que são corretamente classificadas como negativas. Portanto, a curva ROC plota a **Sensibilidade (Recall) x (1 - Especificidade) (FPR)**.

Para plotar a curva ROC primeiro você precisa calcular a TPR e FPR para vários valores de limiares usando a função *roc_curve()*:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

E, então, você pode plotar a curva TPR x FPR usando o Matplotlib:

```
# Função para criação da curva ROC
def plot_roc_curve(fpr, tpr, label=None):
    # Configuração do Gráfico
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # dashed diagonal
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (Fall-Out)', fontsize=16)
    plt.ylabel('True Positive Rate (Recall)', fontsize=16)
    plt.grid(True)

# Chamando a função para criação da curva ROC
plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)

fpr_90 = fpr[np.argmax(tpr >= recall_90_precision)] # fpr do recall da precisao de 90%
plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:") # linha vertical vermelha
plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:") # linha horizontal vermelha
plt.plot([fpr_90], [recall_90_precision], "ro")# marcando com bolinha vermelha
plt.show()
```

Observação: Mais uma vez, existe uma compensação: quanto maior a revocação (TPR), mais Falsos Positivos (FPR) o classificador produz. A linha pontilhada representa a curva ROC de um classificador puramente aleatório; Um bom classificador fica o mais distante dessa linha possível (em direção ao canto superior esquerdo).

Uma maneira de comparar classificadores é medir a área abaixo da curva (**AUC**). Um classificador perfeito terá um **ROC AUC = 1**, enquanto um classificador puramente aleatório terá um **ROC AUC = 0,5**. O Scikit-Learn fornece uma função para calcular o **ROC AUC**:

```
# Calculando a área abaixo da curva ROC (ROC AUC)
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

Esse **AUC** pode sugerir que esse classificador é muito bom mas, mas não é verdade. A quantidade de positivos (5s) é muito mais baixa que a de negativos (Não-5s) então a curva ROC está nos dando a visão errada. Melhor considerar a curva PR, vista anteriormente. Caso haja preocupação com falso positivo, melhor ver a curva ROC.

Comparando curvas ROC

Vamos treinar um classificador **RandomForestClassifier** e comparar a sua **curva ROC** e a pontuação **ROC AUC** para o **SGDClassifier**.

Primeiro, você precisa obter pontuações para cada Instância no conjunto de Treinamento. Mas, devido ao modo como funciona a classe RandomForestClassifier ela não possui um método *decision_function()*.

Em vez disso, ela possui um método *predict_proba()*. Os classificadores do Scikit-Learn geralmente tem um ou outro. O método *predict_proba* retorna um array que contém uma linha por instância e uma coluna por classe, cada uma contendo **a probabilidade de instância dada pertencer à classe dada** (por exemplo, 70% de chance de a imagem representar um 5):

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=42)
# obtendo as pontuações para cada Instância no conjunto de Treinamento
y_probas_forest = cross_val_predict (forest_clf, X_train, y_train_5, cv=3, method="predict_proba")
```

Mas, para plotar uma curva ROC, você precisa de **pontuação**, não de **probabilidades**. Uma solução simples é **utilizar a probabilidade da classe positiva como a pontuação**:

```
y_scores_forest = y_probas_forest[:,1] # pontuação = probabilidade de classe positiva
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5, y_scores_forest)
```

Agora, você está pronto para plotar a curva ROC. É útil plotar a primeira curva ROC também para ver como ela se comparam:

```
# Plotando ambas curvas ROC
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()
```

Observação: Como você pode ver, a curva ROC do RandomForestClassifier parece bem melhor que a do SGDClassifier: se aproxima muito do canto superior esquerdo.

Como resultado, sua curva ROC AUC também será significativamente melhor:

```
roc_auc_score(y_train_5, y_scores_forest)
```

Exercício: Tente medir as pontuações da precisão e da revocação: você deve encontrar 98,5% de precisão e 82,8% de revocação, nada mal!

RESUMO Provavelmente, agora você já sabe como treinar classificadores binários, escolher a métrica apropriada para sua tarefa, avaliar seus classificadores utilizando a Validação Cruzada, selecionar a compensação da precisão/revocação que corresponde às suas necessidades e comparar vários modelos utilizando as curvas ROC e pontuações ROC AUC.

Classificação Multiclasse

```
from sklearn.svm import SVC
svm_clf = SVC(gamma="auto", random_state=42)
svm_clf.fit(X_train[:1000], y_train[:1000]) # y_train dessa vez, não y_train_5
svm_clf.predict([um_digito])
```

```
some_digit_scores = svm_clf.decision_function([um_digito])
some_digit_scores
```

```
np.argmax(some_digit_scores)
```

```
svm_clf.classes_
```

```
svm_clf.classes_[5]
```

```
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC(gamma="auto", random_state=42))
ovr_clf.fit(X_train[:1000], y_train[:1000])
ovr_clf.predict([um_digito])
```

```
len(ovr_clf.estimators_)
```

```
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([um_digito])
```

```
sgd_clf.decision_function([um_digito])
```

```
sgd_clf.score(X_train, y_train)
```

Aviso: as duas células a seguir podem levar cerca de 30 minutos para serem executadas ou mais, dependendo do seu hardware.

```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
sgd_clf.fit(X_train_scaled, y_train)
```

```
sgd_clf.score(X_train, y_train)
```

```
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
```

Análise de Erro

```
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

```
# desde sklearn 0.22, você pode usar sklearn.metrics.plot_confusion_matrix()
def plot_confusion_matrix(matrix):
    """essa funcao serve se voce prefere com cores e barra de cor"""
    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(111)
    cax = ax.matshow(matrix)
    fig.colorbar(cax)
```

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

+ Código + Texto

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```