Classification Concepts

Lecturer: Ngo Huy Bien
Software Engineering Department
Faculty of Information Technology
University of Science
Ho Chi Minh City, Vietnam
nhbien@fit.hcmus.edu.vn

Objectives

- To *write* a machine learning program recognizing a handwritten digit
- To explain classification *terminologies* via examples

Contents

References

1. Aurelien Geron (2017). Hands on Machine Learning with Scikit Learn and TensorFlow. O'Reilly Media.

The MNIST Dataset [1]

- The *MNIST dataset*, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau.
- Each image is *labeled* with the digit it represents.
- *Scikit-Learn* provides many helper functions to download popular datasets. MNIST is one of them.

Downloading The Dataset

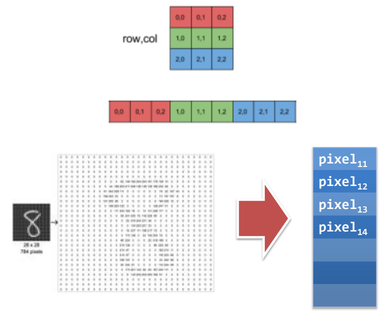
```
import numpy as np
from sklearn.datasets import fetch_openml

def sort_by_target(mnist):
    reorder_train = np.array(sorted([(target, i) for i, target in 
enumerate(mnist.target[:60000])]))[:, 1]
    reorder_test = np.array(sorted([(target, i) for i, target in 
enumerate(mnist.target[60000:])]))[:, 1]
    mnist.data[:60000] = mnist.data[reorder_train]
    mnist.target[:60000] = mnist.target[reorder_train]
    mnist.data[60000:] = mnist.data[reorder_test + 60000]
    mnist.target[60000:] = mnist.target[reorder_test + 60000]

try:
    mnist = fetch_openml('mnist_784', version=1, cache=True)
    mnist.target = mnist.target.astype(np.int8) # fetch_openml() returns targets 
as strings
    sort_by_target(mnist) # fetch_openml() returns an unsorted dataset
except ImportError:
    from sklearn.datasets import fetch_mldata
    mnist = fetch_mldata('MNIST original')
```

Loading Data

```
X, y = mnist["data"], mnist["target"]
print('X.shape:', X.shape)
print('y.shape:', y.shape)
```

```
X.shape: (70000, 784)
y.shape: (70000,)
```

Reshaping Matrix To Vector

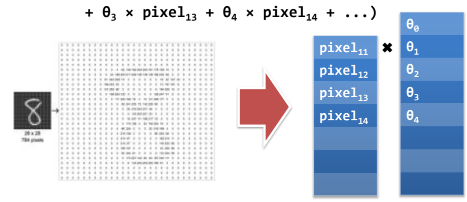


Visualizing The Data

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
print(y[36000])
some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap =
matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```

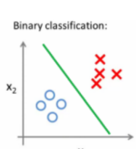
Selecting a Model

- $\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$
- $\text{housing\_price} = \theta_0 + \theta_1 \times \text{median\_income} + \theta_2 \times \text{total\_rooms} + \theta_3 \times \text{households}$
- $\text{number} = f(\theta_0 + \theta_1 \times \text{pixel}_{11} + \theta_2 \times \text{pixel}_{12} + \theta_3 \times \text{pixel}_{13} + \theta_4 \times \text{pixel}_{14} + \ldots)$



Binary Classifier

- $\text{number}_5 = f(\theta_0 + \theta_1 \times \text{pixel}_{11} + \theta_2 \times \text{pixel}_{12} + \theta_3 \times \text{pixel}_{13} + \theta_4 \times \text{pixel}_{14} + \ldots)$
  $= 1$ if the image is number 5
  $= 0$ otherwise



Preparing Training Data

- The MNIST dataset is actually already split into a *training set* (the first 60,000 images) and a *test set* (the last 10,000 images):

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

# Shuffle the training set;
# this will guarantee that all cross-validation folds will be similar
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
print ('X_train.shape:', X_train.shape)
print ('y_train.shape:', y_train.shape)
print ('X_test.shape:', X_test.shape)
print ('y_test.shape:', y_test.shape)
```

```
X_train.shape: (60000, 784)
y_train.shape: (60000,)
X_test.shape: (10000, 784)
y_test.shape: (10000,)
```

Preparing Train Data for Number 5

```
# Only try to identify one digit—for example, the number 5
y_train_5 = (y_train == 5) # True for all 5s, False for all
other digits.
y_test_5 = (y_test == 5)
print ('y_train:', y_train)
print ('y_train_5:', y_train_5)
```

```
y_train: [2 1 7 ... 5 9 8]
y_train_5: [False False False ...  True False False]
```

The Class **SGDClassifier**

- This classifier has the advantage of being capable of handling very large datasets efficiently.
- This is in part because SGD deals with training instances independently, one at a time (which also makes SGD well suited for online learning).
- https://scikit-learn.org/stable/modules/sgd.html
- The class SGDClassifier implements a plain stochastic gradient descent learning routine which supports different *loss functions* and *penalties* for classification.
- The concrete loss function can be set via the loss parameter.

Training a **SGDClassifier** for Number 5

```
from sklearn.linear_model import SGDClassifier
# Stochastic Gradient Descent (SGD) classifier
# The SGDClassifier relies on randomness during training (hence
# the name “stochastic”). If you want reproducible results, you
# should set the random_state parameter.
# See also https://scikit-learn.org/stable/modules/sgd.html
# for loss functions and penalties.

sgd_clf_5 = SGDClassifier(max_iter = 1000, tol = 3, random_state
= 42)
sgd_clf_5.fit(X_train, y_train_5)
```

```
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=1000,
       n_iter=None, n_iter_no_change=5, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=42, shuffle=True, tol=3,
       validation_fraction=0.1, verbose=0, warm_start=False)
```

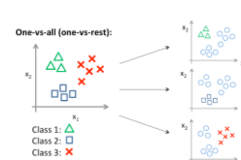
Making Prediction

```
print ('Prediction:', sgd_clf_5.predict([some_digit]))
#The classifier guesses that this image represents a 5
(True).
```

```
Prediction: [ True]
```

One-Versus-All Strategy

- One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train *10 binary classifiers*, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on).
- Then when you want to classify an image, you get the decision *score from each classifier* for that image and you select the class whose classifier outputs the highest score.
- This is called the *one-versus-all (OvA) strategy* (also called one-versus-the-rest).



Implementing OvA With **`SGDClassifier`**

- <u>Under the hood</u>, Scikit-Learn actually trained *10 binary classifiers*, got their decision scores for the image, and selected the class with the highest score.

```
sgd_clf = SGDClassifier(max_iter = 1000, tol = 3,
random_state = 42)
sgd_clf.fit(X_train, y_train) # y_train, not y_train_5
print ('Prediction:', sgd_clf.predict([some_digit]))
```

```
Prediction: [5]
```

Printing The Classifier Properties

```
print('Classes:', sgd_clf.classes_)
print('Class 5:', sgd_clf.classes_[5]) # the class at
index 5 happens to be class 5
some_digit_scores =
sgd_clf.decision_function([some_digit])
print('All 10 scores:', some_digit_scores)
print('Max score:', np.argmax(some_digit_scores))
```

```
Classes: [0 1 2 3 4 5 6 7 8 9]
Class 5: 5
All 10 scores: [[-23641.49767859 -65644.73846393 -16200.9203688
-9416.25775593
  -45647.6328221   17596.51581023 -36571.49746203 -
22054.48774638
  -24696.14657217 -33091.093484  ]]
Max score: 5
```

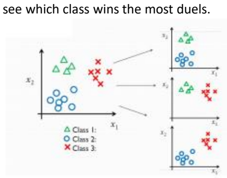
Implementing OvA With **`RandomForestClassifier`**

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(n_estimators = 10,
random_state=42)
forest_clf.fit(X_train, y_train)
print ('Prediction:', forest_clf.predict([some_digit]))
print ('Number of classes:', len(forest_clf.estimators_))
print ('Prediction probabilities:',
forest_clf.predict_proba([some_digit]))
```

```
Prediction: [5]
Number of classes: 10
Prediction probabilities: [[0.  0.  0.  0.2 0.  0.8 0.  0.  0.  0. ]]
```

One-Versus-One Strategy

- Another strategy is to train a binary classifier for *every pair of digits*: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on.
- This is called the *one-versus-one (OvO) strategy*.
- If there are N classes, you need to train N × (N – 1) / 2 classifiers. For the MNIST problem, this means training 45 binary classifiers!
- When you want to classify an image, you have to run the image through *all 45 classifiers* and see which class wins the most duels.



Implementing OvO With **`SGDClassifier`**

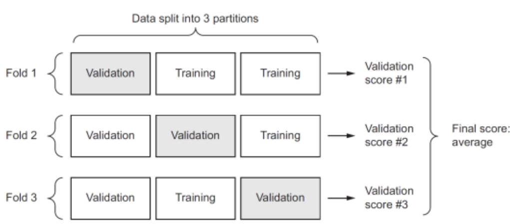
```
from sklearn.multiclass import OneVsOneClassifier
sgd_clf = SGDClassifier(max_iter = 1000, tol = 3,
random_state=42)
ovo_sgd_clf = OneVsOneClassifier(sgd_clf)
ovo_sgd_clf.fit(X_train, y_train)
print ('Prediction:', ovo_sgd_clf.predict([some_digit]))
print ('Number of classes:',
len(ovo_sgd_clf.estimators_))
```

```
Prediction: [5.]
Number of classes: 45
```

OvA vs. OvO

- The main advantage of OvO is that each classifier only needs to be *trained on the part of the training set* for the two classes that it must distinguish.
- Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set, so for these algorithms OvO is preferred since it is *faster* to train many classifiers on small training sets than training few classifiers on large training sets.
- For most binary classification algorithms, however, *OvA is preferred*.

3-Fold Cross-Validation Review



Implementing Cross-Validation

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone
# K-fold crossvalidation means splitting the training set
# into K-folds (in this case, three), then making predictions
# and evaluating them on each fold using a model trained on the
remaining folds
skfolds = StratifiedKFold(n_splits=3, random_state=42)
for train_index, test_index in skfolds.split(X_train, y_train):
    clone_sgd_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train[test_index]

    clone_sgd_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_sgd_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print('Ratio of correct predictions:', n_correct / len(y_pred))
```

```
Ratio of correct predictions: 0.8837732453509298
Ratio of correct predictions: 0.879793989699485
Ratio of correct predictions: 0.8657298594789219
```

Using **cross_val_score**

```
from sklearn.model_selection import cross_val_score
print('Accuracy scores:', cross_val_score(sgd_clf,
X_train, y_train, cv=3, scoring="accuracy"))
```

```
Accuracy scores: [0.88377325 0.87979399 0.86572986]
```

Comparing **SGDClassifier** With **RandomForestClassifier**

```
print('sgd_clf accuracy scores:\t',
cross_val_score(sgd_clf, X_train, y_train, cv=3,
scoring="accuracy"))
print('forest_clf accuracy scores:\t',
cross_val_score(forest_clf, X_train, y_train, cv=3,
scoring="accuracy"))
```

```
sgd_clf accuracy scores:        [0.88377325
0.87979399 0.86572986]
forest_clf accuracy scores:     [0.9385123
0.94264713 0.93984098]
```

`SGDClassifier` for Number 5 Accuracy

```
sgd_clf_5 = SGDClassifier(max_iter = 1000, tol = 3,
random_state = 42)
print('sgd_clf_5 accuracy scores:',
cross_val_score(sgd_clf_5, X_train, y_train_5, cv=3,
scoring="accuracy"))
```

```
sgd_clf_5 accuracy scores: [0.9549  0.9674  0.96335]
```

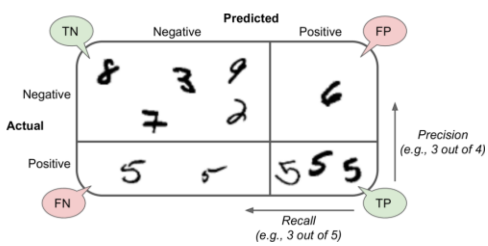
A Very Dumb Classifier

```
from sklearn.base import BaseEstimator
# A very dumb classifier that just classifies every single image in the
“not-5” class
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)

never_5_clf = Never5Classifier()
print ('never_5_clf accuracy scores:', cross_val_score(never_5_clf,
X_train, y_train_5, cv=3, scoring="accuracy"))
```

```
never_5_clf accuracy scores:
[0.90995 0.9085  0.9105 ]
```

- It has *over 90% accuracy*!
- This is simply because only about 10% of the images are 5s, so if you always guess that an image is not a 5, you will be right about 90% of the time.

Confusion Matrix



$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN}$$

Implementing Confusion Matrix

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

# cross_val_predict() performs K-fold cross-validation,
# but instead of returning the evaluation scores,
# it returns the predictions made on each test fold.
# The function computes for each fold the predictions and concatenates
them.
y_train_pred = cross_val_predict(sgd_clf_5, X_train, y_train_5, cv=3)
print('Confusion Matrix:')
print(confusion_matrix(y_train_5, y_train_pred))
```

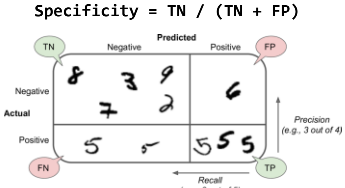
```
Confusion Matrix:
[[53398  1181] # non-5 images (the negative class)
 [ 1106  4315]] # images of 5s (the positive class)
```

True negatives (Ok) **53398**	**False positives (Not good)** **1181**
False negatives (Not good) **1106**	**True positives (Ok)** **4315**

Precision, Recall & Specificity

True negatives	**False positives**
False negatives	**True positives**

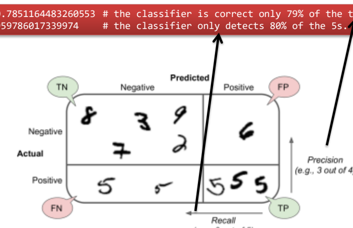
```
Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
Specificity = TN / (TN + FP)
```



Implementing Precision & Recall

```
from sklearn.metrics import precision_score, recall_score
print('Precision:', precision_score(y_train_5, y_train_pred))
print('Recall:', recall_score(y_train_5, y_train_pred))
```

```
Precision: 0.7851164483260553 # the classifier is correct only 79% of the time.
Recall: 0.7959786017339974    # the classifier only detects 80% of the 5s.
```



The Dumb Classifier Precision & Recall

```
from skl
never_5_
never_5_
print('P
never_5_
print('R
never_5_
```

```
_score, recall_score

edict(X_train)
_train_5,

_5,
```

```
Precisio
Recall:
```

F_1 Score

- The *F_1 score* is the harmonic mean of precision and recall.
- Whereas the regular mean treats all values equally, the harmonic mean gives much *more weight to low values*.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

- The classifier will only get a high F_1 score if *both recall and precision* are high.
- The F_1 score favors classifiers that have *similar precision and recall*.

Implementing F_1 Score

```
from sklearn.metrics import f1_score
print('F1 score:', f1_score(y_train_5, y_train_pred))
```

```
F1 score: 0.7905102134285976
```

Making Prediction & Decision Threshold

```
sgd_clf_5.fit(X_train, y_train_5)
print ('Prediction:', sgd_clf_5.predict([some_digit]))
```

```
Prediction: [ True]
```

```
some_digit_score = sgd_clf_5.decision_function([some_digit])
print ('some_digit score:', some_digit_score)
threshold = 0 # The SGDClassifier uses a threshold equal to 0
y_some_digit_pred = (some_digit_score > threshold)
print ('prediction with threshold = 0:', y_some_digit_pred)
```

```
some_digit score: [269.06360646]
prediction with threshold = 0: [ True]
```

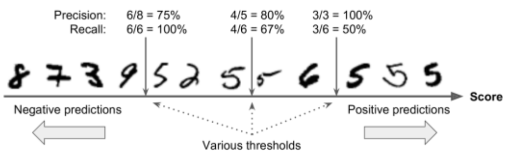
Making Prediction Using Another Decision Threshold

```
some_digit_score = sgd_clf_5.decision_function([some_digit])
print ('some_digit score:', some_digit_score)
threshold = 200000 # The SGDClassifier uses a threshold equal to
200000
y_some_digit_pred = (some_digit_score > threshold)
print ('prediction with threshold = 200000:', y_some_digit_pred)
# The image actually represents a 5,
# and the classifier detects it when the threshold is 0,
# but it misses it when the threshold is increased to 200,000.
```

```
some_digit score: [269.06360646]
prediction with threshold = 200000: [False]
```

`SGDClassifier` Classification Decisions

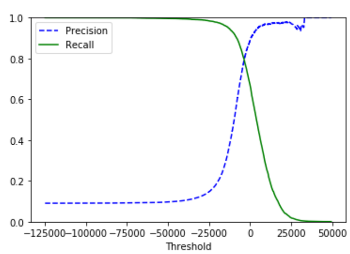
- Let's look at how the SGDClassifier makes its *classification decisions*.
- For each instance, it computes a score based on a *decision function*, and if that score is greater than *a threshold*, it assigns the instance to the positive class, or else it assigns it to the negative class.



Plotting Precision And Recall Versus Decision Threshold: Code

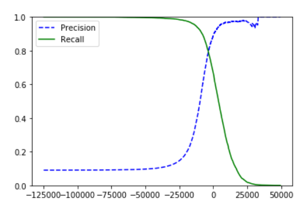
```
from sklearn.metrics import precision_recall_curve
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                              method="decision_function")
precisions, recalls, thresholds = precision_recall_curve(y_train_5,
y_scores)

def plot_precision_recall_vs_threshold(precisions, recalls,
thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.ylim([0, 1])

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```

Plotting Precision And Recall Versus Decision Threshold: Graph



Selecting A Decision Threshold

- Let's suppose you decide to *aim for 90% precision*.
- You *look up* the first plot (zooming in a bit) and find that you need to use a threshold of about 4500.



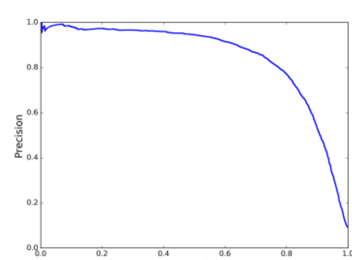
Verifying Decision Threshold Selection

```
y_train_pred_90 = (y_scores > 4500)
print ('Precision score:',
precision_score(y_train_5, y_train_pred_90))
print ('Recall score:', recall_score(y_train_5,
y_train_pred_90))
```

```
Precision score: 0.9062591830737584
Recall score: 0.5688987271721084
```

Precision/Recall Tradeoff

- *Increasing precision reduces recall*, and vice versa.
- If someone says "let's reach 99% precision," you should ask, "at what recall?"



High Precision or High Recall?

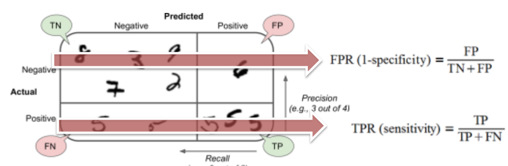
	Actual - Cancer	Actual - NOT Cancer	Total
Predicted - Cancer	TP = 20	FP = 70	90
Predicted - NOT Cancer	FN = 10	TN = 200	210
Total	30	270	300

```
Precision = TP / (TP + FP) = 20/90
Recall = TP / (TP + FN) = 20/30
```

When To Use Accuracy?
When To Use Precision/Recall?

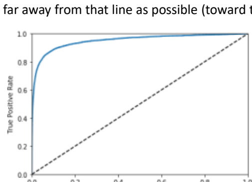
The False Positive Rate

- The *FPR* is the ratio of negative instances that are incorrectly classified as positive.
- It is equal to one minus the *true negative rate*, which is the ratio of negative instances that are correctly classified as negative.
- The TNR is also called *specificity*.



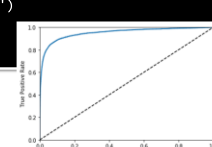
The ROC Curve

- The *receiver operating characteristic (ROC) curve* is another common tool used with binary classifiers.
- The ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate*.
- *The higher* the recall (TPR), *the more* false positives (FPR) the classifier produces.
- The *dotted line* represents the ROC curve of a purely **random** classifier; a good classifier stays as far away from that line as possible (toward the top-left corner).



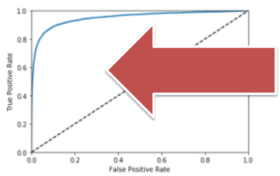
Plotting The ROC Curve

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

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr, tpr)
plt.show()
```



Area Under The Curve

- One way <u>to compare classifiers</u> is to measure the *area under the curve (AUC)*.
- A *perfect classifier* will have a <u>ROC AUC equal to 1</u>, whereas a *purely random classifier* will have a <u>ROC AUC equal to 0.5</u>.



Computing ROC AUC

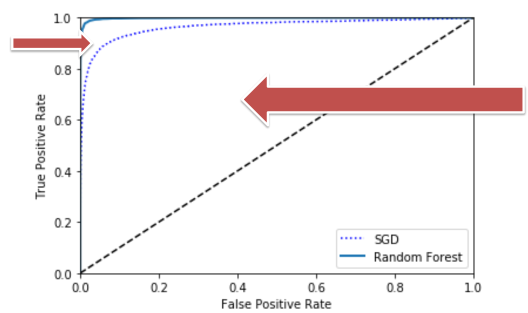
```
from sklearn.metrics import roc_auc_score
print ('Area Under The Curve: ',
roc_auc_score(y_train_5, y_scores))
```

```
Area Under The Curve:  0.9631306882158759
```

Implementing ROC AUC Comparison

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(n_estimators = 100,
random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train,
y_train_5, cv=3,
                                    method="predict_proba")
y_scores_forest = y_probas_forest[:, 1] # score = probability of
positive class
fpr_forest, tpr_forest, thresholds_forest =
roc_curve(y_train_5,y_scores_forest)

plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend()
plt.show()
```

Comparing ROC AUC



`RandomForestClassifier` Scores

```
print('ROC AUC score:', roc_auc_score(y_train_5,
y_scores_forest))
y_train_pred_forest = cross_val_predict(forest_clf,
X_train, y_train_5, cv=3)
print ('Precision score:', precision_score(y_train_5,
y_train_pred_forest))
print ('Recall score:', recall_score(y_train_5,
y_train_pred_forest))
```

```
ROC AUC score: 0.9984095747726475
Precision score: 0.990530303030303
Recall score: 0.8682899833978971
```

Normalizing Image Inputs

- Data normalization is an important step which ensures that *each input parameter* (pixel, in this case) has a similar data distribution.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
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")
```

```
array([0.91026795, 0.90974549, 0.90768615])
```

Error Analysis: Confusion Matrix

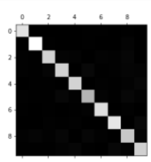
Rows represent actual classes, while columns represent predicted classes

```
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
# Rows represent actual classes, while columns represent predicted
classes
conf_mx
```

```
array([[5721,    2,   23,   13,   11,   54,   48,   10,   37,    4],
       [   2, 6478,   44,   30,    6,   39,    6,   10,  115,   12],
       [  52,   40, 5328,  100,   87,   25,   99,   51,  159,   17],
       [  50,   36,  137, 5359,    1,  227,   35,   52,  134,  100],
       [  19,   23,   36,    9, 5386,   10,   55,   34,   81,  189],
       [  69,   39,   32,  195,   68, 4606,  108,   30,  181,   93],
       [  34,   27,   40,    2,   35,   87, 5639,    8,   46,    0],
       [  25,   19,   74,   26,   56,   12,    6, 5806,   14,  227],
       [  50,  157,   85,  178,   12,  156,   57,   27, 4994,  135],
       [  45,   28,   28,   92,  152,   37,    2,  205,   73, 5287]],
      dtype=int64)
```

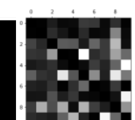
Image Representation Of The Confusion Matrix

```
# image representation of the confusion matrix
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
# The 5s look slightly darker than the other digits,
# which could mean that there are fewer images of 5s in the dataset or
that
# the classifier does not perform as well on 5s as on other digits.
```



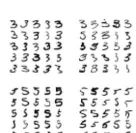
Another Image Representation Of The Confusion Matrix

```
# Divide each value in the confusion matrix
# by the number of images in the corresponding class
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums

# Fill the diagonal with zeros to keep only the errors
np.fill_diagonal(norm_conf_mx, 0)

# Plot the result
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
# Again, rows represent actual classes, while columns represent
predicted classes
# The columns for classes 8 and 9 are quite bright,
# which tells you that many images get misclassified as 8s or 9s.
# Conversely, some rows are pretty dark, such as row 1:
# this means that most 1s are classified correctly
```



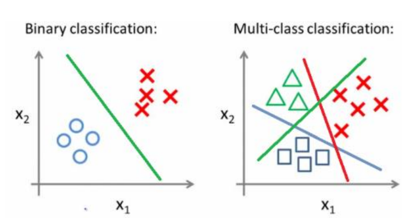
Analyzing Individual Errors

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = matplotlib.cm.binary, **options)
    plt.axis("off")

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()
```

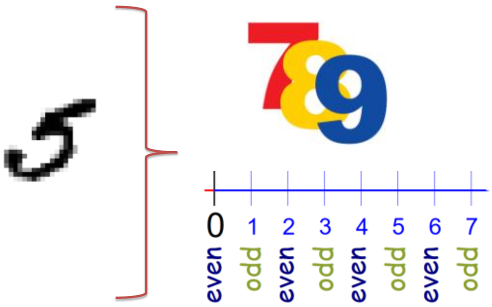


The two 5×5 blocks on the left show digits classified as 3s, and the two 5×5 blocks on the right show images classified as 5s.

Binary vs. Multiclass Classification



Multilabel Classification



Implementing Multilabel Classification

```
from sklearn.neighbors import KNeighborsClassifier

# Create a y_multilabel array containing two target labels for each
digit image:
# the first indicates whether or not the digit is large (7, 8, or 9) and
# the second indicates whether or not it is odd.
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

# Create a KNeighborsClassifier instance
# (which supports multilabel classification, but not all classifiers do)
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)

print ('some_digit > 7, some_digit is odd:',
knn_clf.predict([some_digit]))
```

```
some_digit > 7, some_digit is odd: [[False  True]]
```

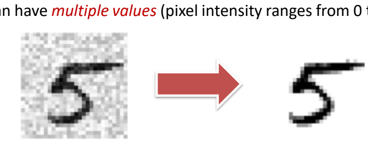
Evaluating A Multilabel Classifier

```
y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_train, cv=3)
f1_score(y_train, y_train_knn_pred, average="macro")
```

```
CPU issue!
```

Multioutput Classification

- It is simply a generalization of multilabel classification where *each label can be multiclass* (i.e., it can have more than two possible values).
- Let's build a system that *removes noise* from images.
- It will take as *input* a noisy digit image, and it will (hopefully) *output* a clean digit image, represented as an array of pixel intensities.
- The classifier's output is *multilabel* (one label per pixel) and each label can have *multiple values* (pixel intensity ranges from 0 to 255).

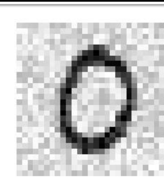


Adding Noise To MNIST Images

```
import numpy.random as rnd
# Create the training and test sets by taking the MNIST images
# and adding noise to their pixel intensities using NumPy's
randint() function
noise = rnd.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = rnd.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Visualizing a Test Image (!)

```
some_index = 300
digit_with_noise = X_test_mod[some_index]
digit_with_noise_image = digit_with_noise.reshape(28, 28)
plt.imshow(digit_with_noise_image, cmap = matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```



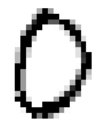
Implementing Multioutput Classification

```
from sklearn.neighbors import KNeighborsClassifier
# Create a KNeighborsClassifier instance
# (which supports multioutput classification, but not all
classifiers do)
knn_clf = KNeighborsClassifier()
# Train the classifier and make it clean this image
knn_clf.fit(X_train_mod, y_train_mod)
some_index = 300
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
```

Making Prediction

```
some_index = 300
clean_digit = knn_clf.predict([X_test_mod[some_index]])
clean_digit_image = clean_digit.reshape(28, 28)
plt.imshow(clean_digit_image, cmap = matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```



Thank You for Your Time