Performance Measures

Monday, March 2, 2020 4:15 PM

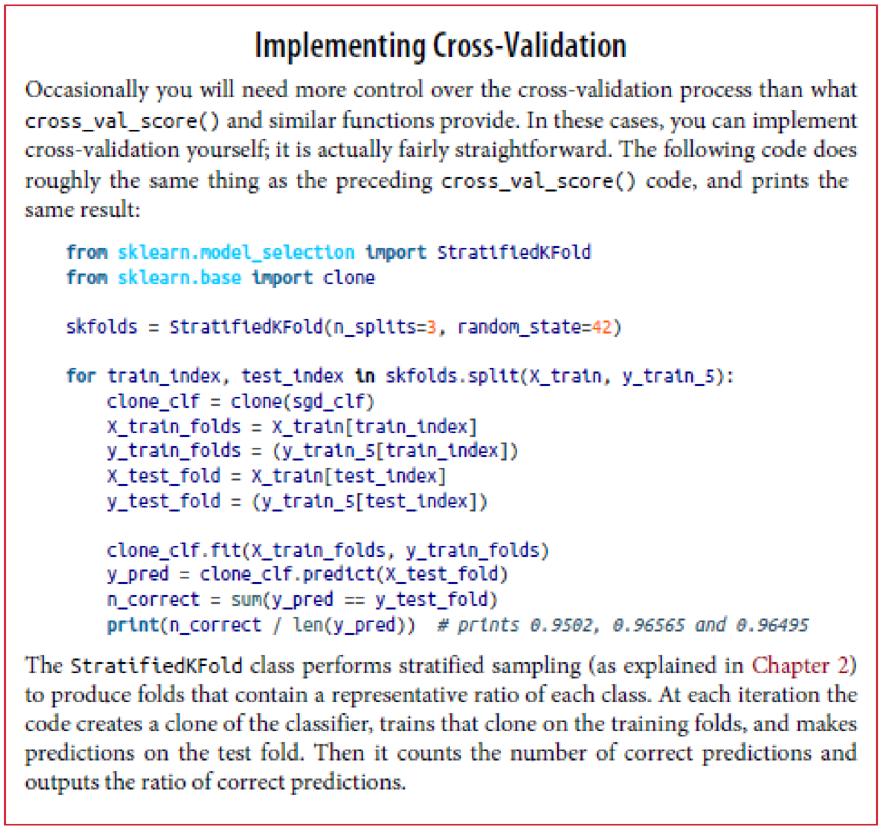
Note Written by: Zulfadli Zainal

Performance measure for classification is much more complex than regressor performance measures.

Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in Chapter 2.

Implementing Cross Validation: The following code roughly done the same thing as the preceding cross_val_score() code, and prints the same result:



Implementing Cross-Validation

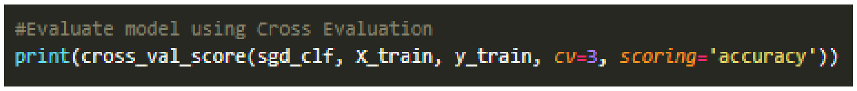
Occasionally you will need more control over the cross-validation process than what `cross_val_score()` and similar functions provide. In these cases, you can implement cross-validation yourself; it is actually fairly straightforward. The following code does roughly the same thing as the preceding `cross_val_score()` code, and prints the same result:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = (y_train_5[train_index])
    X_test_fold = X_train[test_index]
    y_test_fold = (y_train_5[test_index])

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))  # prints 0.9502, 0.96565 and 0.96495
```

The `StratifiedKFold` class performs stratified sampling (as explained in Chapter 2) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

Now, lets measure our SGD Classifier model accuracy using cross_val_score()

```
from sklearn.model_selection import cross_val_score
```



```
#Evaluate model using Cross Evaluation
print(cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring='accuracy'))
```

```
from sklearn.model_selection import cross_val_score

#Evaluate model using Cross Evaluation
print(cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring='accuracy'))
```

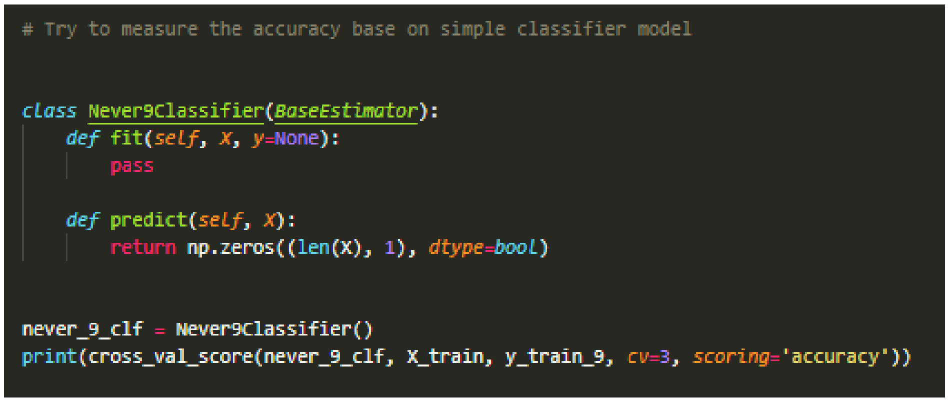
Result:

[0.87035 0.86345 0.85575]

The accuracy is around 85% - 87%

Lets try to compare the accuracy with more simple classifier - And test our number 9 sample accuracy.

```
from sklearn.base import BaseEstimator
```



```
# Try to measure the accuracy base on simple classifier model


class Never9Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass

    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)


never_9_clf = Never9Classifier()
print(cross_val_score(never_9_clf, X_train, y_train_9, cv=3, scoring='accuracy'))
```

```
from sklearn.base import BaseEstimator

# Try to measure the accuracy base on simple classifier model

class Never9Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)

never_9_clf = Never9Classifier()
print(cross_val_score(never_9_clf, X_train, y_train_9, cv=3, scoring='accuracy'))
```

Result:

[[0.90255 0.9011 0.8989]]

The accuracy is around 89% - 90%

The accuracy is different. This is because the number of datasets that containing 9 is small compared to the total datasets. This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with unbalanced datasets (i.e., when some classes are much more frequent than others).

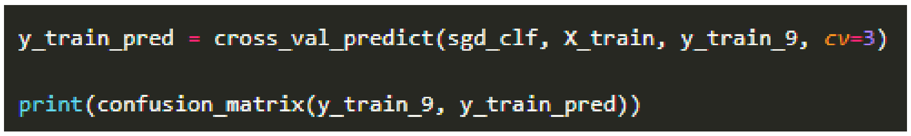
Confusion Matrix

Another way to evaluate performance of classifier -> Use confusion Matrix

Mechanism: The general idea is to count the number of times instances of class A are classified as class B.

To compute:

1. You need to have set of predictions
2. This need to be compared with actual targets
3. We can use cross validation function (cross_validation_predict) to apply this.

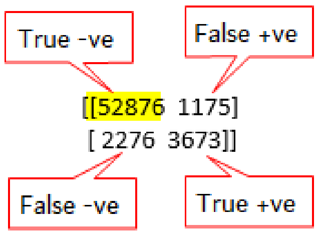


```
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_9, cv=3)

print(confusion_matrix(y_train_9, y_train_pred))
```

```
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_9, cv=3)
print(confusion_matrix(y_train_9, y_train_pred))
```

Result:

```
[[52876  1175]
 [ 2276  3673]]
```



52876: Correctly classified as non 9
1175: Wrongly classified as 9
2276: Wrongly classified as non 9
3673: Correctly classified as 9

If perfect predictions, it will look like this:

```
[[54579, 0],
[ 0, 5421]]
```

Confusion Matrix -> Give you a lot of information (But complicated and confusing) -> Overcome this using Precision!

Equation 3-1. Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

TP -> True Positive
FP -> False Positive

However if the number of sample is very small (Eg: 1/1), you can still get 100% precision. To overcome this issue, usually precision is being used together with Recall.

Equation 3-2. Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

FN -> False Negative

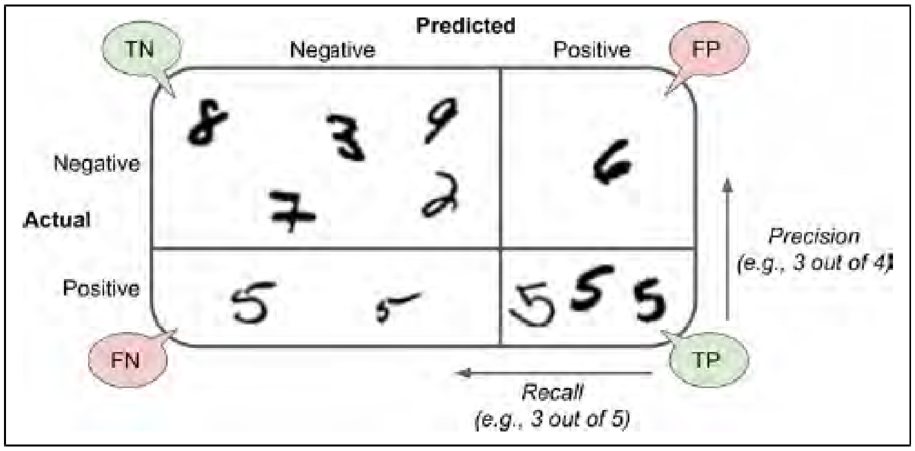
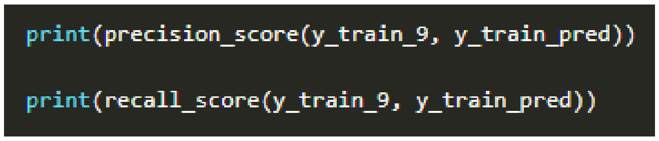


Figure 3-2. An illustrated confusion matrix

Precision and Recall

Scikit Learn provide a precision and recall functions to calculate the classifier metrics.



```
print(precision_score(y_train_9, y_train_pred))

print(recall_score(y_train_9, y_train_pred))
```

```
print(precision_score(y_train_9, y_train_pred))
print(recall_score(y_train_9, y_train_pred))
```

Result:

(Confusion Matrix - Result from before script)
[[52641 1410]
[2258 3691]]

Precision Score: 0.7235836110566556
Recall Score: 0.6204404101529669

This results means ->

1. It is only correct to predict 72% of the time
2. Also, it only detects 62% of the 9s

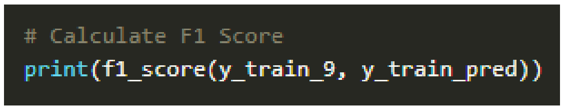
Often: People combine precision & recall -> Call it F1 Score

F1 Score is the harmonic means of precision + recall

Equation 3-3. F_1 score

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

Scikit learn also provide calculation for F1 score:



```
# Calculate F1 Score
print(f1_score(y_train_9, y_train_pred))
```

```
# Calculate F1 Score
print(f1_score(y_train_9, y_train_pred))
```

[[52573 1478]
[1634 4315]]

Precision Score: 0.7448644916278266
Recall Score: 0.7253319885695075
F1 Score: 0.7349684891841254

Read this: Important!

The F1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters on surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught). Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the precision/recall tradeoff.

Precision and Recall Tradeoff

To understand this tradeoff -> Need to understand how SGD classifier makes its decision.

How SGD Classifier works:

1. For each instance, it computes a score based on decision function
2. If the score > threshold … It assigns threshold to a +ve Class
3. Else, it assigns to a -ve Class

Example -> To Detect Actual 5
Left (Lowest Score), Right (Highest Score)

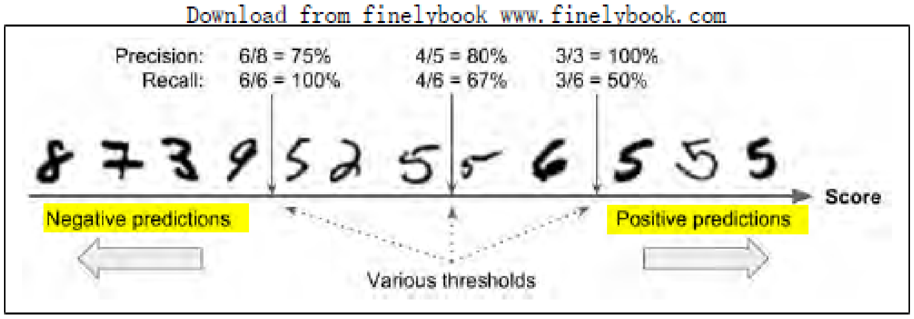

Precision: 6/8 = 75% 4/5 = 80% 3/3 = 100%
Recall: 6/6 = 100% 4/6 = 67% 3/6 = 50%
Score
Negative predictions
Positive predictions
Various thresholds

Figure 3-3. Decision threshold and precision/recall tradeoff

If Central Arrow in the middle

So,

Precision = 4/5 = 80% TP/(TP + FP) --> 4 True +ve (Actual 5) & 1 False +ve (Actual 6)
Recall = 4/6 = 67% TP/(TP + FN) --> Out of Out of 6 (Actual 5), it can only detect 4. FN = 2 (Actual 5 in left threshold).

If move threshold to the right (Precision Increase, Recall Reduces)

Precision = 3/3 = 100%
Recall = 3/6 = 50%

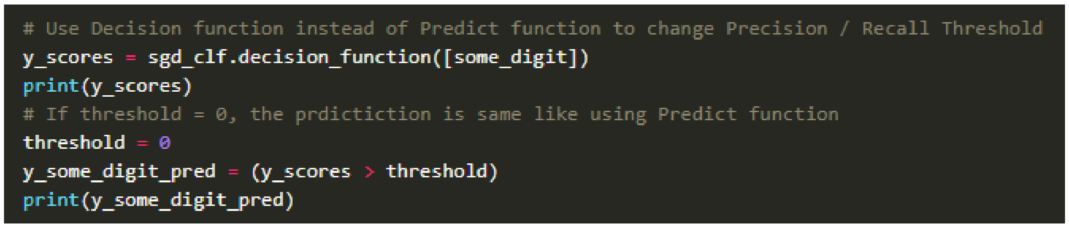
If move threshold to the left (Precision Reduces, Recall Increase)

Precision = 6/8 = 75%
Recall = 6/6 = 100%

How can we change this threshold?

Scikit-Learn not allowing to change the threshold directly: But we can access decision scores that it uses to make predictions.

Instead of calling the classifier's predict() method, you can call its decision_function() method, which returns a score for each instance, and then make predictions based on those scores using any threshold you want:

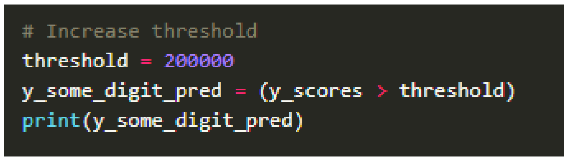


```
# Use Decision function instead of Predict function to change Precision / Recall Threshold
y_scores = sgd_clf.decision_function([some_digit])
print(y_scores)
# If threshold = 0, the prdictiction is same like using Predict function
threshold = 0
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

```
# Use Decision function instead of Predict function to change Precision / Recall Threshold
y_scores = sgd_clf.decision_function([some_digit])
print(y_scores)
# If threshold = 0, the prdictiction is same like using Predict function
threshold = 0
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

Result:
[True]

Increase Threshold:



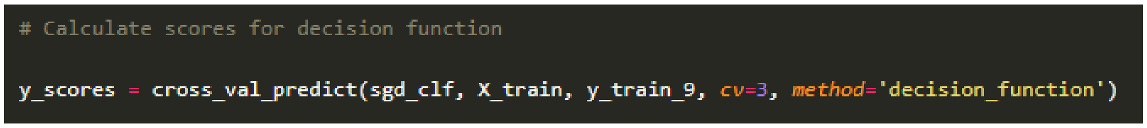
```
# Increase threshold
threshold = 200000
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

```
# Increase threshold
threshold = 200000
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

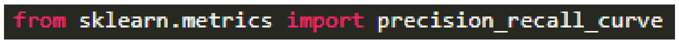
Result:
[False]

This confirms that raising the threshold decreases recall. The image actually represents a 9, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 200,000.

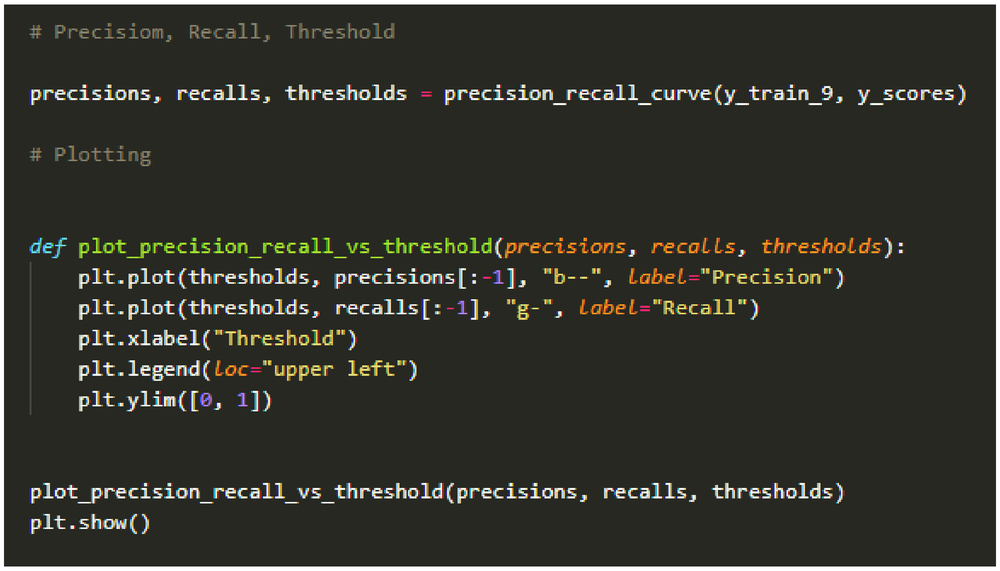
So how to decide which threshold to use? >> Use Matplotlib and plot Precisions vs Recall



```
# Calculate scores for decision function

y_scores = cross_val_predict(sgd_clf, X_train, y_train_9, cv=3, method='decision_function')
```



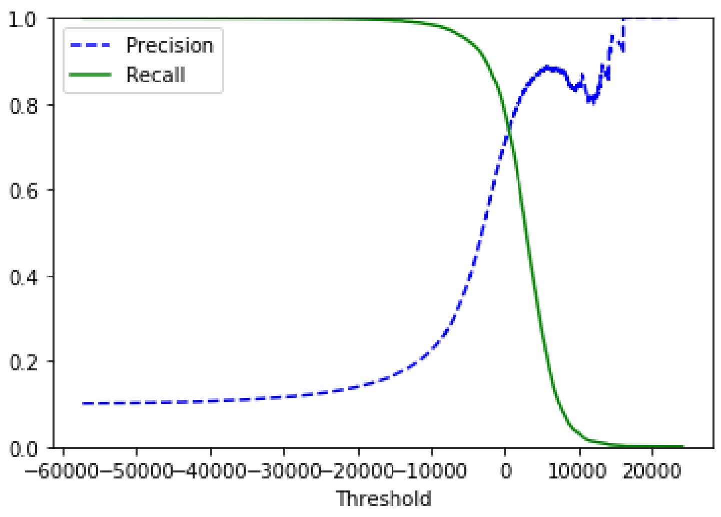
```
from sklearn.metrics import precision_recall_curve
```



```
# Precisiom, Recall, Threshold

precisions, recalls, thresholds = precision_recall_curve(y_train_9, y_scores)

# Plotting


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()
```

```
# Calculate scores for decision function

y_scores = cross_val_predict(sgd_clf, X_train, y_train_9, cv=3, method='decision_function')

# Precisiom, Recall, Threshold

from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_9, y_scores)

# Plotting

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()
```

Result:



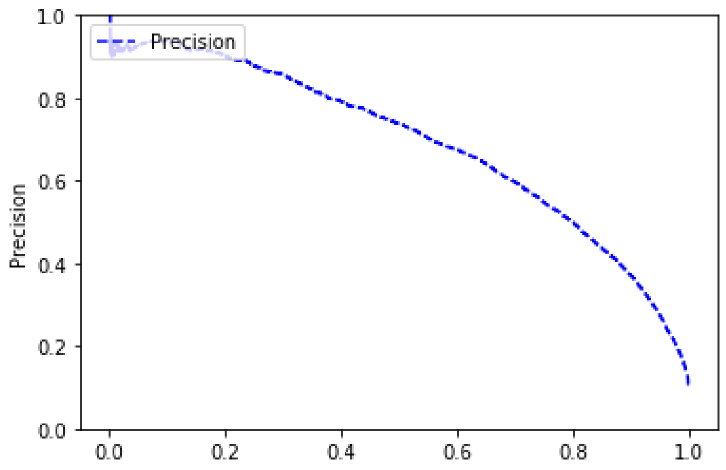
Now you can simply select the threshold value that gives you the best precision/recall tradeoff for your task.

If we plot Precision vs Recall -> We can get better idea what kind of trade-off are we expecting.

```
# Calculate scores for decision function

y_scores = cross_val_predict(sgd_clf, X_train, y_train_9, cv=3, method='decision_function')
```

```
from sklearn.metrics import precision_recall_curve
```

```
# Plotting


def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b--", label="Precision")
    plt.xlabel("Recall")
    plt.ylabel('Precision')
    plt.legend(loc="upper left")
    plt.ylim([0, 1])


plot_precision_vs_recall(precisions, recalls)
plt.show()
```



Recall

So, if someone says "let's reach 99% precision," you should ask, "at what recall?"

The ROC Curve

ROC: Receiver Operating Characteristics (Another common tools used with binary classifiers).

Very similar to Precision/Recall curve.

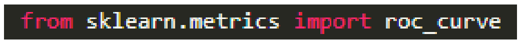
Difference :

Precision/Recall: Plot Precision vs Recall
ROC Curve: Plot True Positive Rate vs False Positive Rate

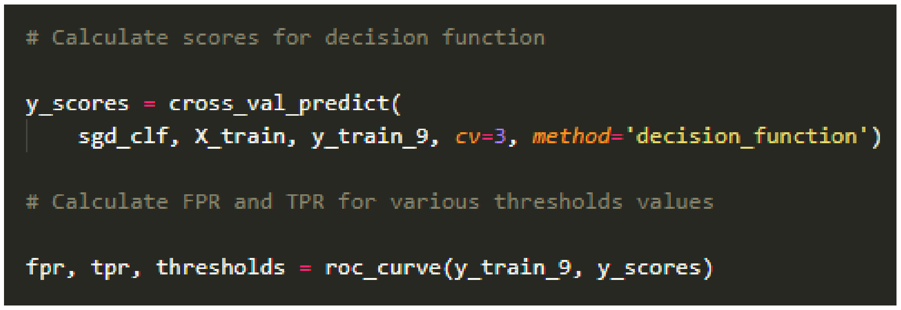
False Positive Rate: Ratio of negative instances that are incorrectly classified as positive.
It is equal to one minus the true negative rate, which is the ratio of negative instances that are correctly classified as negative. The TNR is also called specificity. Hence the ROC curve plots sensitivity (recall) versus 1 – specificity.

To plot ROC curve:-

1. Compute TPR & FPR



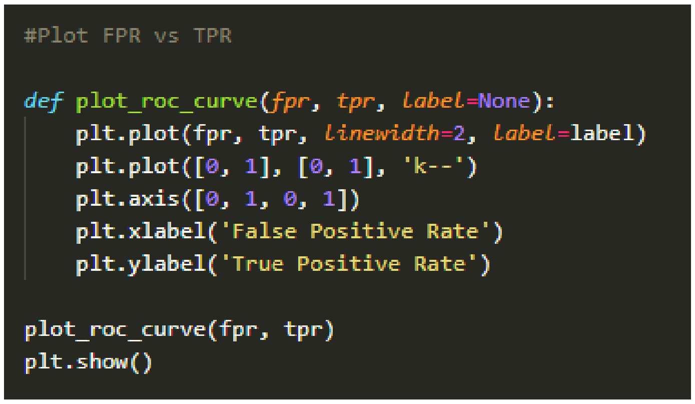
```python
from sklearn.metrics import roc_curve
```



```python
# Calculate scores for decision function

y_scores = cross_val_predict(
    sgd_clf, X_train, y_train_9, cv=3, method='decision_function')

# Calculate FPR and TPR for various thresholds values

fpr, tpr, thresholds = roc_curve(y_train_9, y_scores)
```

```python
# Calculate scores for decision function

from sklearn.metrics import roc_curve

y_scores = cross_val_predict(
    sgd_clf, X_train, y_train_9, cv=3, method='decision_function')

# Calculate FPR and TPR for various thresholds values

fpr, tpr, thresholds = roc_curve(y_train_9, y_scores)
```

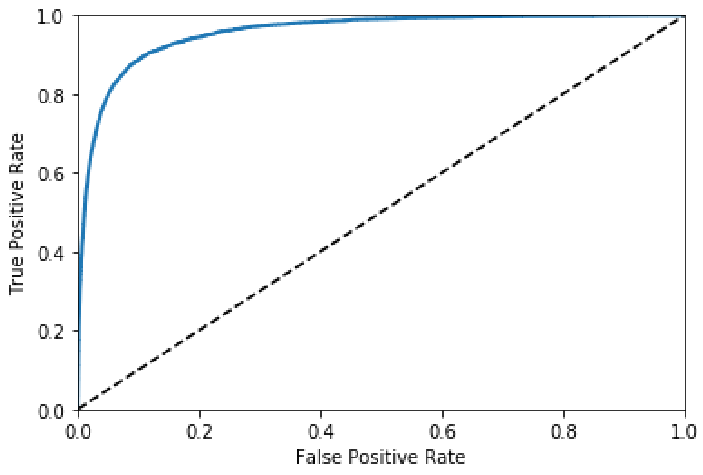
2. Plot FPR & TPR



```python
#Plot FPR vs TPR

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()
```

```python
#Plot FPR vs TPR

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()
```

[Result]

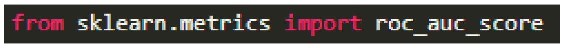


ROC method also have a Tradeoff:

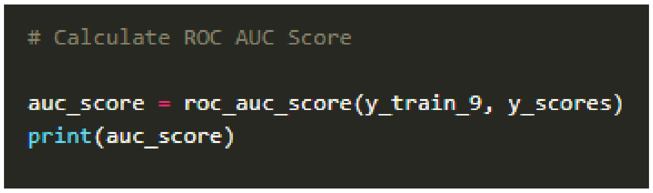
The higher the recall (TPR), the more false positives (FPR) the classifier produces. The dotted line represents the ROC curve of a purely random classifier; a good classifier stays as far away from that line as possible (toward the top-left corner).

One way to compare classifiers: Calculate AUC (Area under Curve)

1. A perfect classifier will have a ROC AUC equal to 1
2. Whereas a purely random classifier will have a ROC AUC equal to 0.5.



```
from sklearn.metrics import roc_auc_score
```

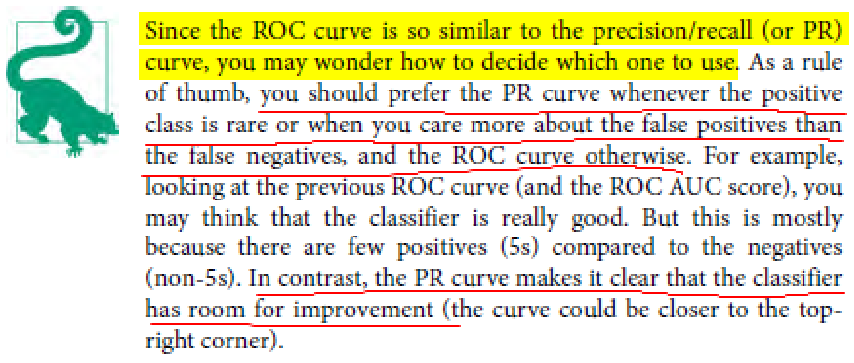


```
# Calculate ROC AUC Score

auc_score = roc_auc_score(y_train_9, y_scores)
print(auc_score)
```

```
# Calculate ROC AUC Score

from sklearn.metrics import roc_auc_score

auc_score = roc_auc_score(y_train_9, y_scores)
print(auc_score)
```

[Result]
0.9579232676469721 (Almost perfect = near to 1)

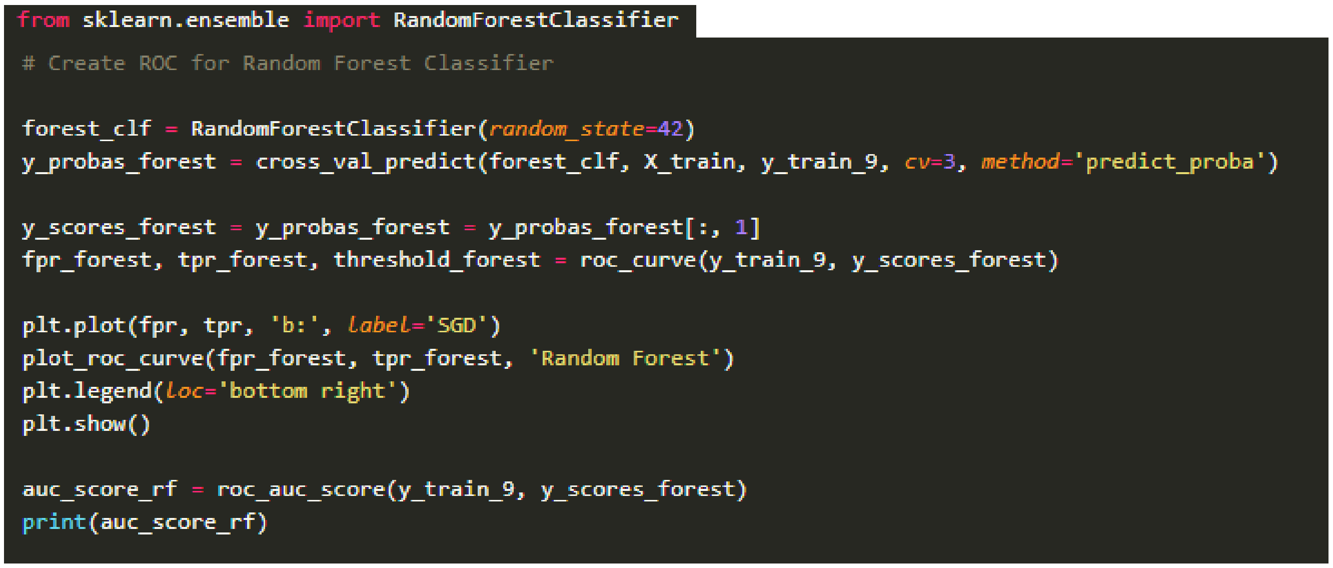
Read this!



Since the ROC curve is so similar to the precision/recall (or PR) curve, you may wonder how to decide which one to use. As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives, and the ROC curve otherwise. For example, looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement (the curve could be closer to the top-right corner).

Compare SGDClassifier & RandomForestClassifier

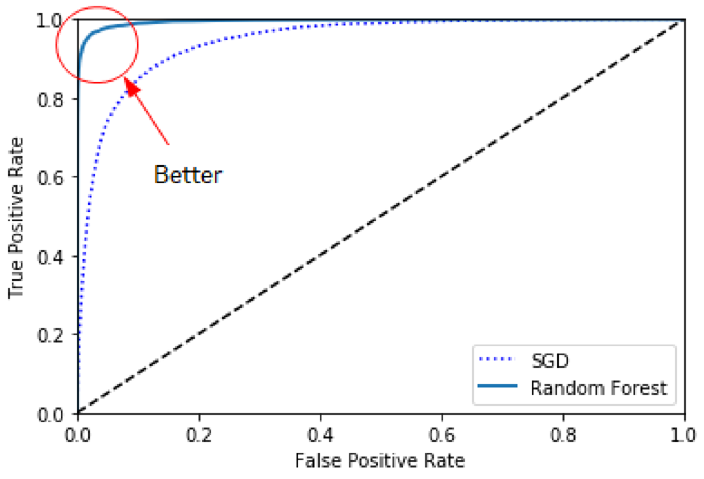
Compare ROC Curves between 2 types of classifiers.



```
from sklearn.ensemble import RandomForestClassifier
# Create ROC for Random Forest Classifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_9, cv=3, method='predict_proba')

y_scores_forest = y_probas_forest = y_probas_forest[:, 1]
fpr_forest, tpr_forest, threshold_forest = roc_curve(y_train_9, y_scores_forest)

plt.plot(fpr, tpr, 'b:', label='SGD')
plot_roc_curve(fpr_forest, tpr_forest, 'Random Forest')
plt.legend(loc='bottom right')
plt.show()

auc_score_rf = roc_auc_score(y_train_9, y_scores_forest)
print(auc_score_rf)
```

```
from sklearn.ensemble import RandomForestClassifier

# Create ROC for Random Forest Classifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_9, cv=3, method='predict_proba')
y_scores_forest = y_probas_forest = y_probas_forest[:, 1]
fpr_forest, tpr_forest, threshold_forest = roc_curve(y_train_9, y_scores_forest)

plt.plot(fpr, tpr, 'b:', label='SGD')
plot_roc_curve(fpr_forest, tpr_forest, 'Random Forest')
plt.legend(loc='bottom right')
plt.show()

auc_score_rf = roc_auc_score(y_train_9, y_scores_forest)
print(auc_score_rf)
```

[Result]
0.9953840296246363 (Better than SGD)



The RandomForestClassifier's ROC curve looks much better than the SGDClassifier's: it comes much closer to the top-left corner. As a result, its ROC AUC score is also significantly better.