CAP4770 – Intro to Data Science

Classification

Prof. Ye Xia

MNIST Dataset

- Openml.org is a public repository for machine learning data and experiments, that allows everybody to upload open datasets.
- MNIST dataset 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.

  The following code loads the data from openml.org. It takes some time to complete.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'url'])
```

  - DESCR: describes the dataset
  - data: contains a DataFrame with one row per instance and one

column per feature

- target: contains an array with the labels

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

```
(70000, 784)
```

```
y.shape
```

```
(70000,)
```

- There are 70,000 images, and each image has 784 features.

  Each image is 28×28 pixels, and each feature represents one pixel's intensity, from 0 (white) to 255 (black).

- X is a DataFrame; y is a series where each entry is a string. Need some conversion.

```
X.head(2)
```

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

2 rows × 784 columns

```
y[0]
```

```
'5'
```

```
X=X.to_numpy()
```

```
import numpy as np
y = y.astype(np.uint8)
```

```
y[0]
```

```
5
```

`ndarray.astype`: copy the array as the type specified.

Let's take a peek at one digit from the dataset. All you need to do is grab an instance's feature vector, reshape it to a 28×28 array, and display it using Matplotlib's `imshow()` function, which displays data as an image.

```
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.figure(figsize = (3,3))
plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```

- More digits form MNIST:

- Recall that we are not supposed to look at the data much before splitting it into the training set and test set.

- 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:]
```

- The training set is already shuffled for us, which is good because this guarantees that all cross-validation folds will be similar (you don't want one fold to be missing some digits).

Training a Binary Classifier

- Let's first deal with a simpler problem and only try to identify one digit, the number 5.
- We need to create the labels for this classification task.

```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

- We will try Scikit-Learns `SGDClassifier` class.
  - This is not a single classifier. Depending on the hyperparameters `loss` (function) and `penalty`, it can be configured into one out of several types of linear classifiers, such as (linear) support vector machines (SVM) or logistic regression. The default is linear SVM.
  - It uses the stochastic gradient descent (SGD) optimization method

in the training process, which aims at solving the underlying optimization problem.

  - A plain gradient descent method computes the gradient (in each iteration step) using potentially all the training instances.
  - In contrast, SGD approximates the gradient using a single training instance (in each iteration).
  - SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. When the data is sparse, the classifiers can easily scale to problems with more than 10^5 training instances and more than 10^5 features.

- Let's create an SGDClassifier and train it on the whole training set.

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

- The SGDClassifier relies on randomness during training (hence the name “stochastic”). If you want reproducible results, you should set the `random_state` parameter.
- Try the trained model:

```
sgd_clf.predict([some_digit])
```

```
array([ True])
```

Performance Measures

- First, cross-validation:

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

```
array([0.95035, 0.96035, 0.9604 ])
```

- Accuracy (ratio of correct predictions) on the validation set is about $95 - 96\%$. Great, but...
- A classifier that classifies every instance as 'not-5' does well in accuracy too.

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

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

```
array([0.91125, 0.90855, 0.90915])
```

- Why? Only about 10% of the images are 5s; the other 90% are not 5.
- Accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with skewed datasets.
- In many binary classification scenarios, the positive instances are much fewer than the negative instances.

Confusion Matrix

- Idea: Count the number of times instances of class A are classified as class B for all classes A and B.
- This gives a $K \times K$ confusion matrix, where K is the number of classes.

  Each row represents an actual class, while each column represents a predicted class.

  Accuracy is part of the information provided by the confusion matrix.
- We will use the `cross_val_predict()` function to generate predictions (for the training set).
  - It does k-fold cross-validation. But, instead of returning a score on each validation set, it gives predictions on the validation set.
  - Since every fold is a validation set in some split, a prediction is made for every instance in the original training set (although the

model parameters are different for different training rounds).

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
```

```
array([[53892,   687],
       [ 1891,  3530]])
```

- The first row of this matrix is for non-5 images (the negative class).

  53892 of them were correctly classified as non-5s (true negatives), while the remaining 687 were wrongly classified as 5s (false positives).

- The second row is for the images of 5s (the positive class).

  1891 were wrongly classified as non-5s (false negatives), while the remaining 3530 were correctly classified as 5s (true positives).

	predicted neg.	predicted pos.
actual neg.	TN	FP
actual pos.	FN	TP

Precision and Recall

- A positive instance is thought to be associated with a significant event having occurred (e.g., a fire alarm, or a positive medical test).
- Ideally, you want to be able to identify all the positive instances (perfect recall), and when you classify an instance as 'positive', it is a correct decision (perfect precision).
- **Precision**: measures the accuracy of the positive predictions

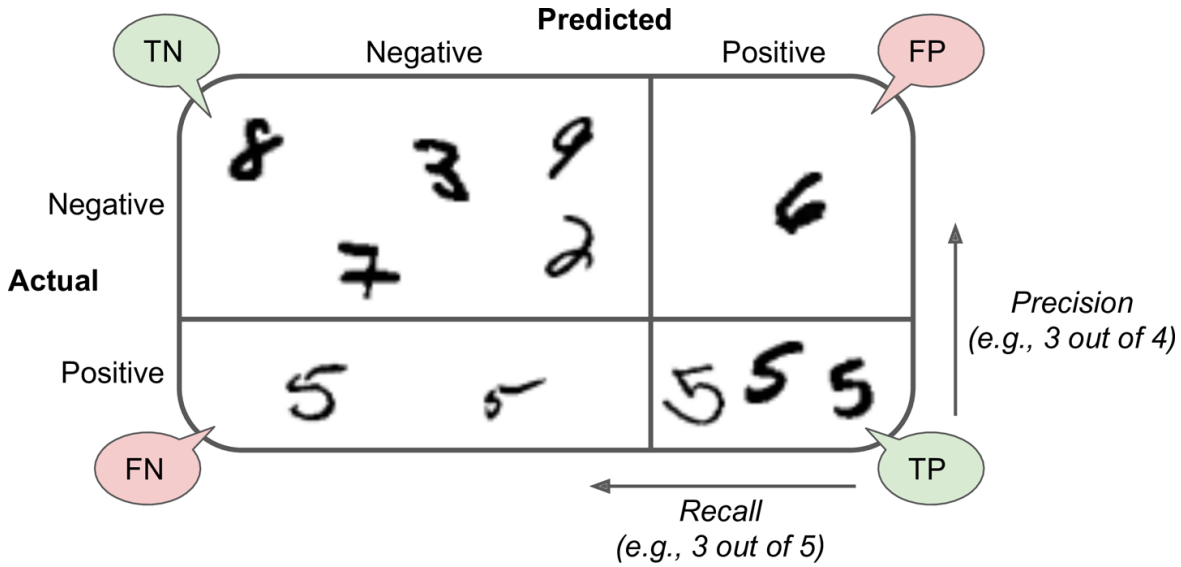
$$\text{precision} = \frac{TP}{TP + FP}.$$

  $TP + FP$ is the number of positive predictions.
- A trivial way to have perfect precision is to make one single positive prediction and ensure it is correct (precision $= 1/1 = 100\%$). But this would not be very useful, since the classifier would ignore all but one positive instance.

- **Recall:** also called **sensitivity** or the **true positive rate (TPR)**. Recall measures what portion of the actual positive instances is corrected identified (recalled). It is the ratio between the positive instances that are correctly detected by the classifier and the actual positive instances.

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

$TP + FN$ is the number of positive instances in the dataset.



```
from sklearn.metrics import precision_score, recall_score
precision_score(y_train_5, y_train_pred) # == 3530 / (3530 + 687)
```

```
0.8370879772350012
```

```
recall_score(y_train_5, y_train_pred) # == 3530 / (3530 + 1891)
```

```
0.6511713705958311
```

When the classifier claims an image represents a 5, it is correct only 83.7% of the time; and it detects 65.1% of the 5s.

- It is easy to have 100% recall: Just classify every instance as 'positive'; but the precision is most likely poor. We start to see the tension between precision and recall.
- The confusion matrix gives us more information than just precision and recall. It gives us estimates of all the conditional probabilities (see later).

Probability Version

- Suppose we select an instance randomly. Let

  $A \triangleq$ the event that the instance is (actually) positive

  $B \triangleq$ the event that the classifier predicts/classifies the instance as 'positive'.

  Then, we have the complement events:

  $A^c =$ the event that the instance is (actually) negative

  $B^c =$ the event that the classifier predicts/classifies the instance as 'negative'.

  Image the classifier/prediction is by testing of infection to COVID.

- The conditional probability $P(B|A)$ is called **sensitivity** (recall).
  - The name of sensitivity comes from thinking about test sensitivity, i.e., how good the test is in picking up an actual positive case.
  - Sensitivity is the **true positive rate** (of all actual positive cases).

- Given the sensitivity $P(B|A)$, we know the **false negative rate** $P(B^c|A)$ because $P(B^c|A) = 1 - P(B|A)$.
- High sensitivity means that the prediction/test can pick up (recall) positive instances with high probability.
- High sensitivity means low false negative rate.

- **Specificity** (usually used for medical test): is defined as $P(B^c|A^c)$. It is the conditional probability that given an instance is negative, the prediction result says 'negative'.
  - Specificity is the **true negative rate**.
  - Given the specificity $P(B^c|A^c)$, we know the **false positive rate** $P(B|A^c)$ because $P(B^c|A^c) = 1 - P(B|A^c)$.
  - High specificity means low false positive rate.
  - When a medical test for a medical condition has high specificity, it is highly specific for that condition in that it rarely gives positive results for actual negative cases.

- From the sensitivity $P(B|A)$ and specificity $P(B^c|A^c)$, we know the false negative rate $P(B^c|A)$ and the false positive rate $P(B|A^c)$. What about $P(A|B)$ and $P(A^c|B^c)$? (These measure the accuracies of positive or negative predictions, respectively.)

  By Bayes' theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|A^c)P(A^c)} \left(= \frac{P(A \cap B)}{P(B)} \right)$$

$$P(A^c|B^c) = \frac{P(B^c|A^c)P(A^c)}{P(B^c|A^c)P(A^c) + P(B^c|A)P(A)} \left(= \frac{P(A^c \cap B^c)}{P(B^c)} \right).$$

  To compute $P(A|B)$ or $P(A^c|B^c)$, we also need to know the prior $P(A)$ (and hence, $P(A^c)$). This is crucial!

- Conclusion: If we have the sensitivity (recall), specificity and the prior, we will know all the conditional probabilities.

Precision and Recall – To Continue

- In the probability context, precision is $P(A|B)$.
- From precision, we also have $P(A^c|B)$.
- From sensitivity $P(B|A)$, we have $P(B^c|A)$.
- From Bayes' theorem,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|A^c)P(A^c)}.$$

we get

$$P(B|A^c) = \frac{P(B|A)P(A)}{P(A^c)} \left(\frac{1}{P(A|B)} - 1 \right). \tag{1}$$

If we also have the priors $P(A)$ and $P(A^c)$, we will be able to compute the false positive rate $P(B|A^c)$, and then, the specificity $P(B^c|A^c)$.

- With the priors, we will also be able to compute $P(A|B^c)$ (no specific term

for this; it is $1-$ the accuracy of negative predictions):

$$P(A|B^c) = \frac{P(B^c|A)P(A)}{P(B^c|A)P(A) + P(B^c|A^c)P(A^c)}$$

- Conclusion: If we have the sensitivity (recall), precision and the prior, we will know all the conditional probabilities.

- Now, return to the case of the confusion matrix. The confusion matrix gives us the estimates of all the conditional probabilities.
- We can also compute $P(A)$ and $P(A^c)$ from the confusion matrix.

  In fact, only the training dataset is needed to compute the priors. (If we have split the training and test sets properly, the test set should have about the same ratio of positive and negative instances.)
- If there is an equal number of positive and negative training instances (so that $P(A) = P(A^c) = 1/2$), then knowing the precision and recall is enough to know everything else.

The F_1 Score

- It is often convenient to combine precision and recall into a single metric called the F_1 score.

  The F_1 score is the harmonic mean of precision and recall.

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

  The harmonic mean gives much more weight to the low value.

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
```

```
0.7325171197343846
```

- The F_1 score may not be always what you want. In some contexts

you mostly care about precision, and in other contexts you really care about recall.

Example 1: 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.

Example 2: Suppose you train a classifier to detect shoplifters in 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 some false alerts, but almost all shoplifters will get caught).

- Ideally, you always want to have perfect recall and perfect precision. Unfortunately, that is not possible in most situations. Increasing precision reduces recall, and vice versa.

Precision/Recall Trade-Off

- Return to the confusion matrix:

$$\begin{array}{c} \\ \text{actual neg.} \\ \text{actual pos.} \end{array} \begin{array}{c} \begin{array}{cc} \text{predicted neg.} & \text{predicted pos.} \end{array} \\ \begin{pmatrix} \text{TN} & \text{FP} \\ \text{FN} & \text{TP} \end{pmatrix} \end{array}$$

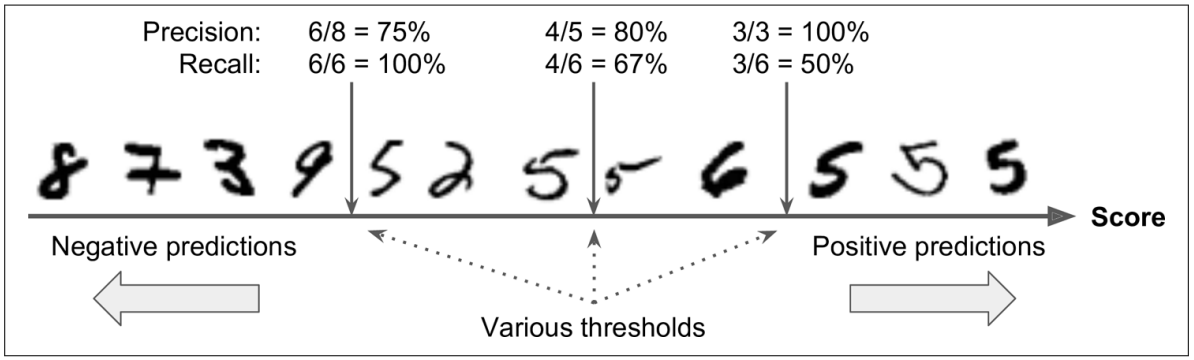
  For a given dataset, each row sums to a constant, regardless of which classifier is used.

- Because recall $= \frac{TP}{TP+FN}$ and $TP + FN$ doesn't change, an increase in the recall means that TP increases.

- Many classifiers are **threshold based**: Each instance has a score computed based on a decision function. If that score is greater than a threshold, the instance is assigned to the positive class; otherwise it is

assigned to the negative class.

- Recall decreases as the threshold increases: As you increase the threshold, you make fewer positive predictions and some of the positive instances may go from true positives to false negatives; then TP decreases.



- Precision tends to increase as the threshold increases, and the reason is more subtle.
  - Consider the scores of all the instances. If the classifier is doing the right things, the (actual) positive instances tend to have high scores and the negative instances tend to have low scores.

- As the threshold increases, the total number of positive predictions ($TP + FP$) decreases. However, among the positive predictions, the proportion of (actual) positive instances tends to increase, because their scores tend to be among the highest.
- In other words, (actual) positive instances are increasingly more concentrated among the positive predictions, as the threshold gets higher.
- The precision $\frac{TP}{TP+FP}$ measures that concentration.

- Scikit-Learn does not let you set the threshold directly, but it does give you access to the 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.
- You can use any threshold you want to make predictions based on those scores.

```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

```
array([2164.22030239])
```

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

- The `SGDClassifier` uses a threshold equal to 0, so the previous code returns the same result as the `predict()` method (i.e., True).

What Threshold to Use?

- First, use the `cross_val_predict()` function to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions.

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

- With these scores, use the `precision_recall_curve()` function to compute precision and recall for all possible thresholds.

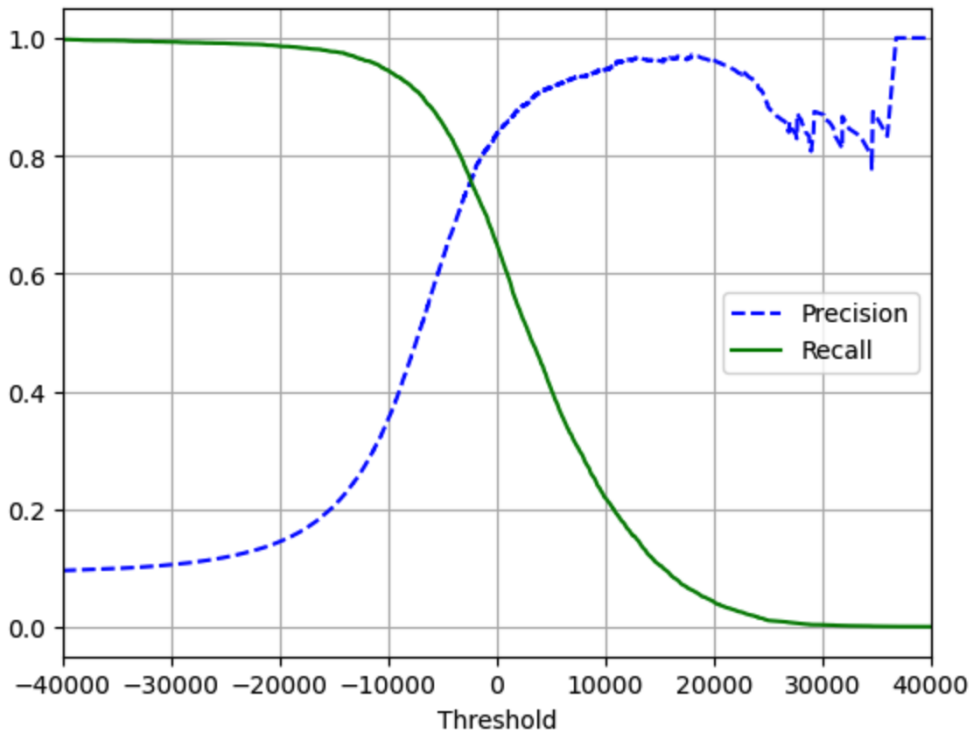
```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

- Plot:

```
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.legend(loc='best')
    plt.xlim(-40000,40000)
    plt.grid()
    plt.xlabel('Threshold')

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

`precisions[:-1]`: skip the last element. The length of `precisions` and `recalls` is 60001 each; the length of `thresholds` is 60000.

The last precision and recall values are 1 and 0, respectively and do not have a corresponding threshold.



- You can choose a threshold value that gives precision and recall values suitable for your application.
- The precision curve is not everywhere increasing, especially where the threshold values are high.

Why? At a high threshold, there are not many positive predictions; most of them are true positives; false positives are much fewer, if any.

Example: For the threshold value 20000, there are 241 positive predictions, 232 true positives, and 9 false positives.

Imagine that the few false positives are scattered among the true positives. As the threshold increases, there is a high chance that some true positives get removed from the set of positive predictions before any false positive is removed.

It is easy to see that the precision $= \frac{TP}{TP+FP} = \frac{1}{1+FP/TP}$ is an increasing function of TP when $FP > 0$. Therefore, the removal of true positives leads to a decrease in precision.

Since at a high threshold TP is not a big number, the decrease can be noticeable when it happens.

- You can also plot precision vs. recall:

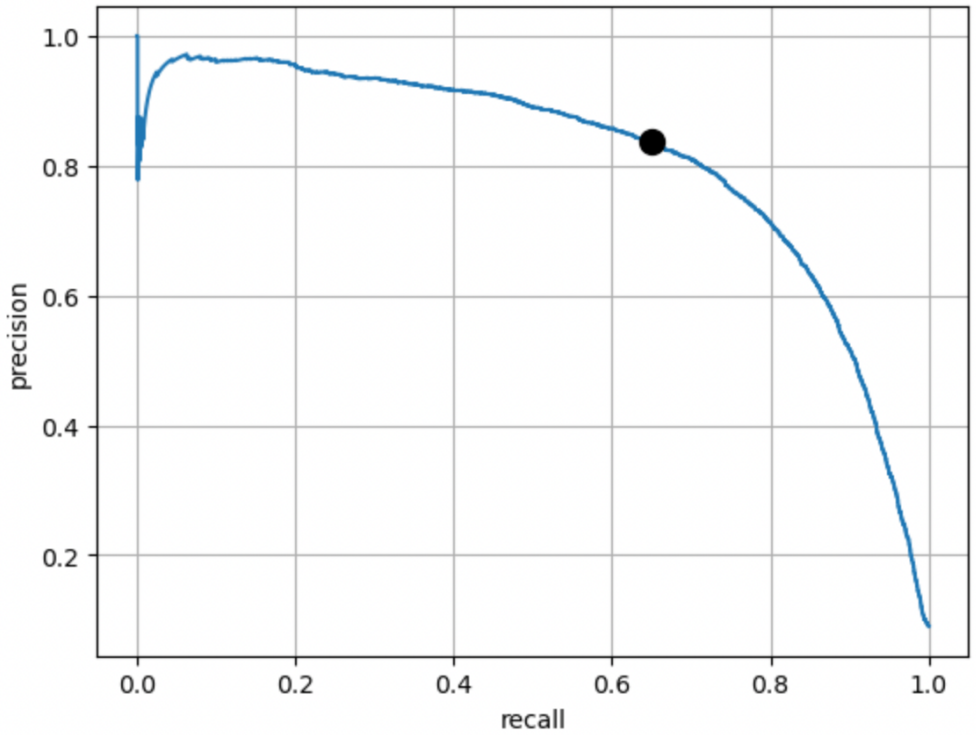


Figure 1: The black dot corresponds to the default threshold, 0.

- Notice the sharp drop of precision when the recall is around 0.8.
- With a high enough threshold, the precision can usually be made close to 1. But, the recall suffers.

- Example: Use the threshold for which precision $= 0.9$.

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

```
precision_score(y_train_5, y_train_pred_90)
```

```
0.9000345901072293
```

```
recall_score(y_train_5, y_train_pred_90)
```

```
0.4799852425751706
```

`np.argmax(arr)` gives the first index for which the maximum is achieved. Here, the array `arr` is `precisions >= 0.90`, which gives a boolean array.

The ROC Curve

- The receiver operating characteristic (ROC) curve is another common tool used with binary classifiers. (The term ROC came from radar field.)
- Intuition: You want to recall most positive instances (high recall). One way is to declare every instance positive; but that generates too many false positives.

  What you want is high recall and low FPR (which is an alternative, but related, goal to 'high recall and high precision').
- The ROC curve plots the true positive rate (i.e., recall or sensitivity) against the false positive rate (FPR).
- The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to $1 -$ the true negative rate (TNR).
- Another name for TNR is **specificity** (see earlier).

- Hence, the ROC curve plots sensitivity (recall) versus $1 -$ specificity.

```
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--') # Dashed diagonal
    plt.grid()
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate (Recall)')
    plt.plot(0, 1, marker="o", markersize=10, markeredgecolor="black", markerfacecolor="black")
    plt.plot(0.5, 0.5, marker="s", markersize=10, markeredgecolor="green", markerfacecolor="green")

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

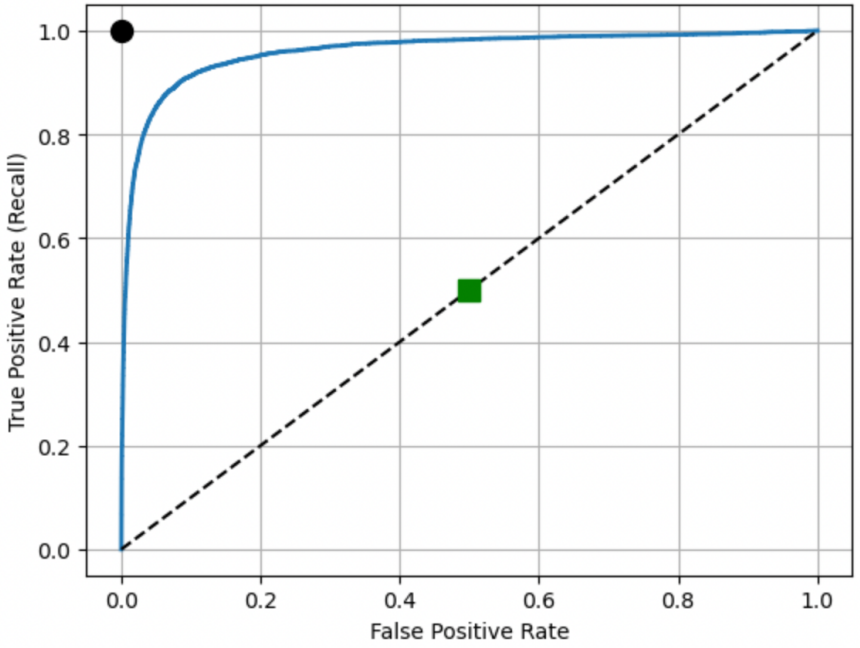


Figure 2: The black dot corresponds to the perfect classifier. The green square corresponds to a purely random classifier (coin tossing).

- One way to compare classifiers is to measure the area under the ROC curve (AUC).
  - As we get closer to the perfect classifier, the AUC approaches 1.

- A classifier where we assign each instance a score uniformly at random on the interval $[0, 1]$ will have a ROC AUC equal to 0.5 (the dash line).

  Why? Suppose the threshold is $t \in (0, 1)$. Take an arbitrary positive instance. When its score falls on $(t, 1)$, which has a probability $1 - t$, it will be classified as 'positive' (correctly); hence the recall (TPR) is $1 - t$.

  Similarly, for an arbitrary negative instance, when its score falls on $(t, 1)$, which has a probability $1 - t$, it will also be classified as 'positive' (falsely); hence the FPR is $1 - t$.

  For each threshold value, the recall (TPR) is equal to the FPR.

- Scikit-Learn provides a function to compute the ROC AUC:

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

```
0.9604938554008616
```

ROC Curve vs. Precision/Recall Curve

- We say you want high recall and low FPR. Earlier, we also say you want high recall and high precision. Given a fixed recall, is high precision the same as low FPR?

  From an earlier expression, we can relate FPR $P(B|A^c)$ with precision $P(A|B)$.

$$P(B|A^c) = \frac{P(B|A)P(A)}{P(A^c)} \left(\frac{1}{P(A|B)} - 1 \right).$$

  Consider a fixed, high recall $P(B|A)$. Suppose the precision is nearly 1. The term $\frac{1}{P(A|B)} - 1$ will be small. However, if $P(A)/P(A^c)$ is large, then FPR may be large.

- If there are many times more (actual) positive instances than negative instances, high precision does not necessarily imply low FPR (whereas low FPR does imply high precision).

  However, this situation may not happen often in applications; usually, the

positive instances are rarer than negative instances.

- Conversely, if there are much fewer (actual) positive instances than negative instances, high precision implies low FPR whereas low FPR does not necessarily imply high precision.
- If the numbers of (actual) positive instances and negative instances are comparable, then high precision and low FPR imply each other.
- As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you clearly care about the accuracy of positive predictions. Otherwise, use the ROC curve.
- 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): At high recall values, the FPR is low because of that. The precision may not be very high.
- In contrast, the PR curve (see earlier) shows that the classifier has room for improvement. The curve could be flatter (and closer to 1.0 in value) over a wider range of recall values.

RandomForestClassifier

- We will try `RandomForestClassifier` and compare its ROC curve and ROC AUC score to those of the `SGDClassifier`.
- First, you need to get score for each instance in the training set. But due to the way it works, `RandomForestClassifier` does not have a `decision_function()` method. Instead, it has a `predict_proba()` method.
- The `predict_proba()` method returns an array containing a row per instance and a column per class, each containing the probability that the given instance belongs to the given class (e.g., 70% chance that the image represents a 5).

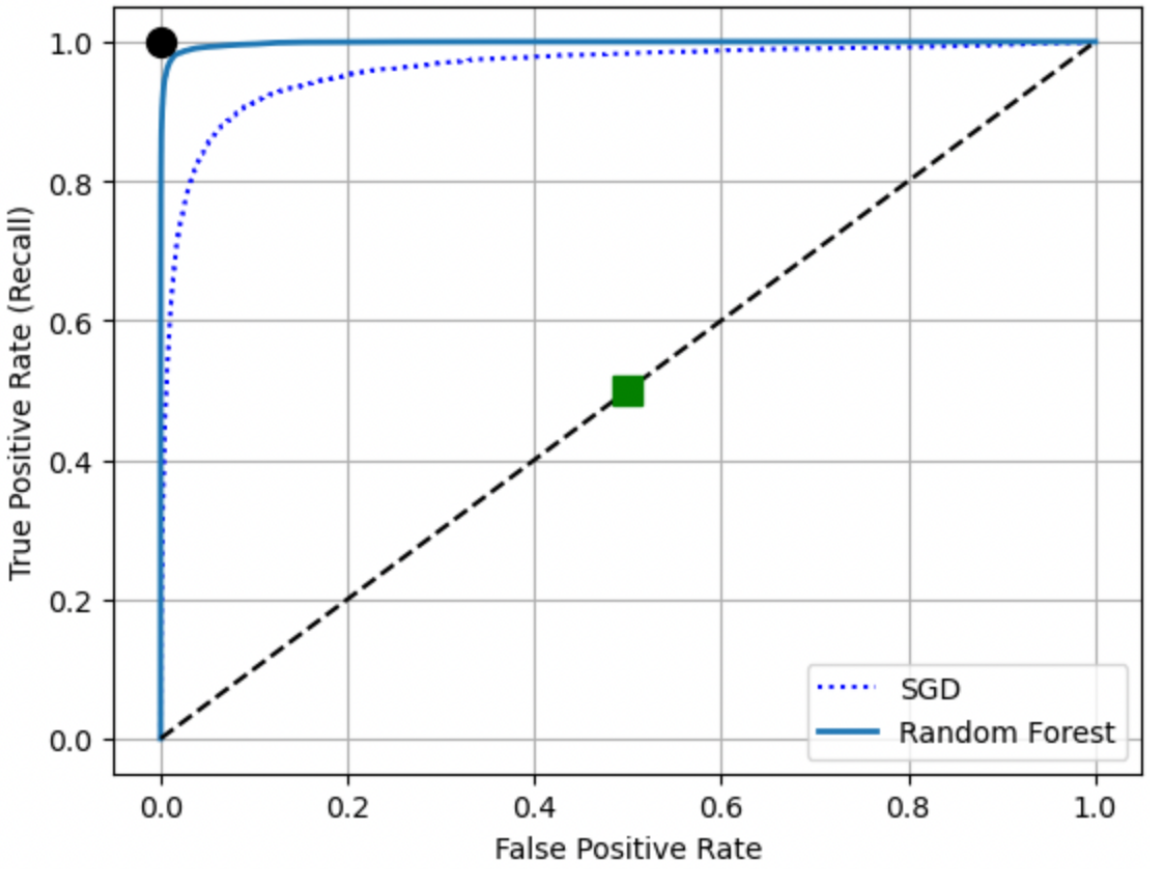
```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

- We will use the positive class's probability as the score.

```
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

- Then, plot:

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



- `RandomForestClassifier`'s ROC curve looks much better than the `SGDClassifier`'s. Its ROC AUC score is

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

```
0.9983436731328145
```

- The precision and recall values are:

```
y_rfc_pred = cross_val_predict(forest_clf, X_train, y_train_5, cv=3)
precision_score(y_train_5, y_rfc_pred)
```

```
0.9905083315756169
```

```
recall_score(y_train_5, y_rfc_pred)
```

```
0.8662608374838591
```

In comparison, they were 0.837 and 0.651 for `SGDClassifier`.

How is prediction made by `RandomForestClassifier`: If the probability for the positive class is greater than that for the negative class, the instance is predicted as positive. This corresponds using the threshold 0.5.

Multiclass Classification

- Some algorithms (such as Random Forest classifiers, and naive Bayes classifiers) are capable of handling multiple classes natively.
- Others (such as Logistic Regression or Support Vector Machine classifiers) are binary classifiers in their original form.
- However, there are various strategies by which you can build a multiclass classifier with multiple binary classifiers.
- **One-versus-the-rest** (OvR) strategy: also called one-versus-all. Train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on).

  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.
- **One-versus-one** (OvO) strategy: 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.

If there are N classes, you need to train $N \times (N-1)/2$ classifiers. However, each classifier only needs to be trained on the part of the training set. For example, the classifier to distinguish 0s and 1s only needs to be trained on images of those two digits.

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. Roughly, this means

- Given an instance, you imagine it is the digit '0'. You run it through the classifiers 0-vs-1, 0-vs-2, ..., 0-vs-9, and count how many times the instance is classified as 0.
- Next, you imagine it is the digit '1'. You run it through the classifiers 1-vs-0, 1-vs-2, ..., 1-vs-9, and count how many times the instance is classified as 1.

  - Continue with the above procedure. In the end, the digit with the most wins will be declared as the predicted output.

- For most binary classification algorithms, OvR is preferred.
- Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set. For these algorithms OvO is preferred because it is faster to train many classifiers on small training sets than to train few classifiers on large training sets.
- Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvR or OvO, depending on the algorithm.

Multiclass Classification with SVM

- Let's try a Support Vector Machine classifier.

```
from sklearn.svm import SVC
svm_clf = SVC()
svm_clf.fit(X_train, y_train) # y_train, not y_train_5
svm_clf.predict([some_digit])
```

```
array([5], dtype=uint8)
```

- You can look at its scores. By default, for each image, there is one score for each class.

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

```
array([[ 1.72501977,  2.72809088,  7.2510018 ,  8.3076379 , -0.31087254,
         9.3132482 ,  1.70975103,  2.76765202,  6.23049537,  4.84771048]])
```

The calculation of the scores is a long story. There is a hyperparameter `decision_function_shape` and by default it is

set to ‘ovr’ (even though we are taking the ‘ovo’ strategy). This leads to one score for each class. The score for a class, say digit 5, is equal to the number of wins by digit 5 (against other nine digits) plus a confidence score whose value is in $(-1/3, 1/3)$ to break ties (another story).

If the hyperparameter `decision_function_shape="ovo"` (rather than the default `"ovr"`), there will be 45 scores for each image, one score for each of the 45 classifiers. See:

https://stackoverflow.com/questions/64561398/how-does-decision-function-in-scikit-learn-calculates-the-scores

- See which class gets the highest score:

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

```
5
```

```
svm_clf.classes_
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

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

```
5
```

The sixed score in the array is the highest. The corresponding class name is stored in the sixth location of the `classes_` variable. The class in that location is digit 5.

- If you want to force Scikit-Learn to use one-versus-one or one-versus-the-rest, you can use the `OneVsOneClassifier` or `OneVsRestClassifier` classes.

Simply create an instance and pass a classifier to its constructor (it does not even have to be a binary classifier).

```
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC())
ovr_clf.fit(X_train, y_train)
```

```
ovr_clf.predict([some_digit])
```

```
array([5], dtype=uint8)
```

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

```
10
```

There are 10 trained binary-classifier models.

Multiclass Classification with SGDClassifier

- `SGDClassifier` has built-in support for multiclass classification.

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

```
array([3], dtype=uint8)
```

  It has got the wrong classification result (the digit should be 5).

- Some details: By default, the hyperparameter `loss` is set to `"hinge"`. In this case, `SGDClassifier` is actually **linear** SVM.
- `SGDClassifier` supports multiclass classification by combining multiple binary classifiers in an OvR scheme.
- At testing time, it computes the confidence score (i.e. the signed distances to the hyperplane) for each classifier and choose the class

with the highest confidence.

- The `decision_function()` method returns one score per class.

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

```
array([[-31893.03095419, -34419.69069632,  -9530.63950739,
          1823.73154031, -22320.14822878,  -1385.80478895,
        -26188.91070951, -16147.51323997,  -4604.35491274,
        -12050.767298  ]])
```

We see that the next best score corresponds to the digit 5. Other digits have far lower scores.

- For a better evaluation of this classifier, you can use cross-validation. For instance, you can use `cross_val_score()` to evaluate the SGDClassifier's accuracy.

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

```
array([0.87365, 0.85835, 0.8689 ])
```

Here, it is OK to use accuracy (i.e., the fraction of correct predictions) as a metric because the dataset is no longer imbalanced. The numbers of instances are similar for all the 10 digits.

```
np.unique(y_train, return_counts=True)
```

```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8),
 array([5923, 6742, 5958, 6131, 5842, 5421, 5918, 6265, 5851, 5949]))
```

If you classify images randomly, you will only get about 10% accuracy.

- Accuracy around 86% is decent. But, it can be improved. For instance, simply scaling the inputs increases accuracy above 89%:

```
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([0.8983, 0.891 , 0.9018])
```

SGDClassifier vs. SVC

- `SGDClassifier` supports different families of models including **linear** Support Vector Machine (SVM) and logistic regression. One unifying feature is that it implements a stochastic gradient descent (SGD) algorithm to solve the underlying optimization problem for learning.
- The optimization objective is set up by specifying the loss function (`loss`) and penalty (`penalty`), e.g.,
  - `loss="hinge"`: (soft-margin) linear Support Vector Machine
  - `loss="log_loss"`: logistic regression
  - `penalty="l2"`: L2 norm penalty on the coefficients
  - `penalty="l1"`: L1 norm penalty on the coefficients
- By default, the hyperparameter `loss` is set to `"hinge"`. In this case, `SGDClassifier` is actually **linear** SVM. However, the underlying optimization algorithm is SGD, which is more scalable than what is used in the plain `SVC` class.

- On the other hand, the `SVC` class has the option to use **nonlinear** kernels, in which case the decision boundaries are nonlinear.

  The hyperparameter `kernel` can be set to `"linear", "poly", "rbf", "sigmoid", "precomputed"`. The default is `"rbf"`, the radial basis function.

- `SVC` uses different algorithms (not SGD) to solve the underlying optimization problem during learning. Internally, it uses `libsvm` and `liblinear` for optimization. As an example, the algorithm built into `libsvm` is a specialized decomposition algorithm for quadratic programming.

- Both `SGDClassifier` and `SVC` have their counterpart for regression. They are called `SGDRegressor` and `SVR`, respectively.

- There are several other implementations in Scikit-Learn for SVM, such as `LinearSVC` and `LinearSVR`.

Error Analysis

- Reminder: If this were a real project, you would continue to explore data preparation options, try out multiple models, shortlist the best ones and fine-tune their hyperparameters using `GridSearchCV`, and automate as much as possible.
- In the following, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.
- First, look at the confusion matrix.

```
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
```

```
array([[5577,    0,   22,    5,    8,   43,   36,    6,  225,    1],
       [   0, 6400,   37,   24,    4,   44,    4,    7,  212,   10],
       [  27,   27, 5220,   92,   73,   27,   67,   36,  378,   11],
       [  22,   17,  117, 5227,    2,  203,   27,   40,  403,   73],
       [  12,   14,   41,    9, 5182,   12,   34,   27,  347,  164],
       [  27,   15,   30,  168,   53, 4444,   75,   14,  535,   60],
       [  30,   15,   42,    3,   44,   97, 5552,    3,  131,    1],
       [  21,   10,   51,   30,   49,   12,    3, 5684,  195,  210],
       [  17,   63,   48,   86,    3,  126,   25,   10, 5429,   44],
       [  25,   18,   30,   64,  118,   36,    1,  179,  371, 5107]])
```

Many digits are confused as 8.

5 and 3 are sometimes hard to tell apart.

- Matplotlib's `matshow()` gives an image representation of the confusion matrix.

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

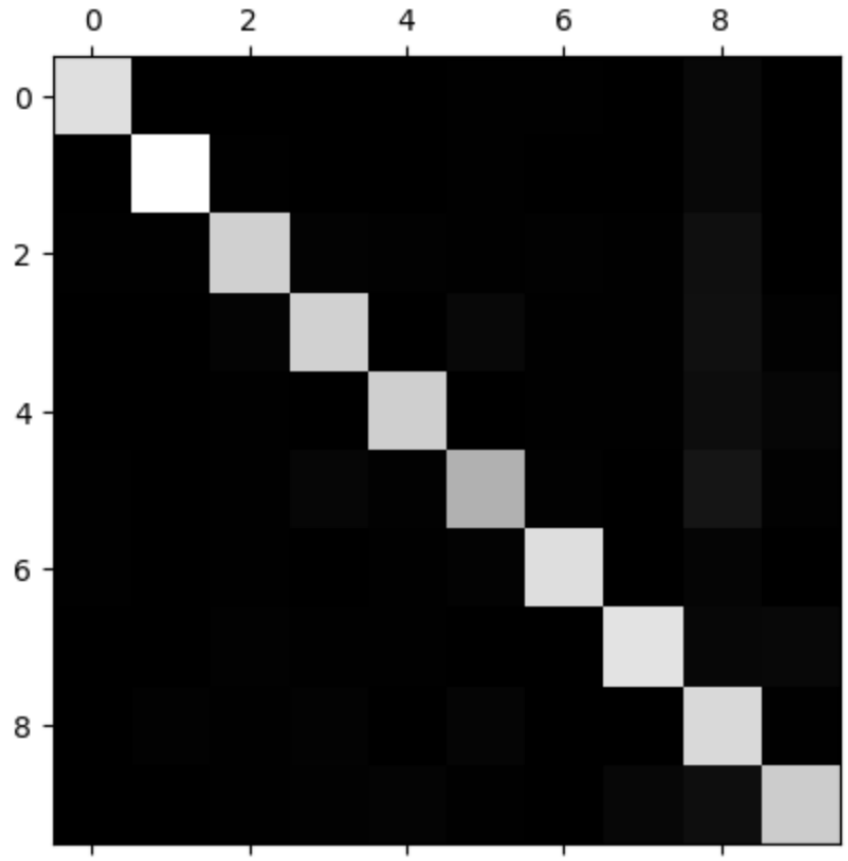
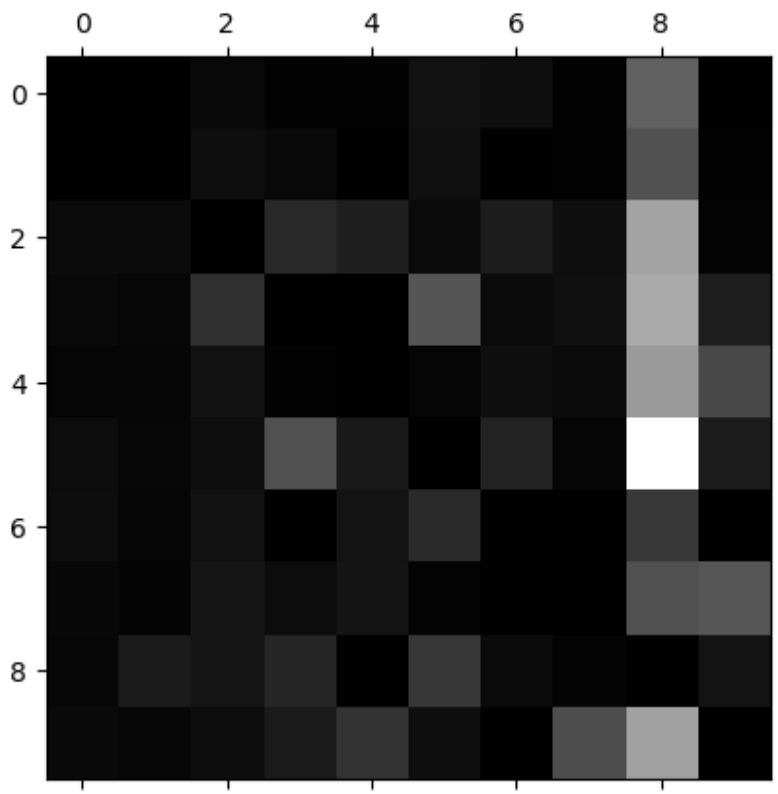


Figure 3: Lighter regions corresponds to bigger numbers.

- It is hard to see the details due to two problems: (i) The numbers were not normalized; (ii) The diagonal numbers are much larger than the off-diagonal numbers, which we are more interested in.
- We will normalize the numbers based on the row sums; and we will replace the diagonal entries with 0.

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```

In this case, you can check that filling diagonal entries with 0 matters more.



Many digits get classified as 8 wrongly (false 8s), especially 5. 3 and 5 get confused with each other. Others ...

- Analyzing the confusion matrix often gives you insights into ways to improve your classifier. It seems that your efforts should be spent on

reducing the false 8s.

- You could try to gather more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them from real 8s.
- You could engineer new features that would help the classifier. For example, write an algorithm to count the number of closed loops (e.g., 8 has two, 6 has one, 5 has none).
- You could preprocess the images (e.g., using Scikit-Image, Pillow, or OpenCV) to make some patterns, such as closed loops, stand out more.

- Next, you may analyze individual errors, say between 3s and 5s.

```
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)]
```

`X_aa` contains all the 3s that are classified as 3s.

`X_ab` contains all the 3s that are classified as 5s.

`X_ba` contains all the 5s that are classified as 3s.

`X_bb` contains all the 5s that are classified as 5s.

You can display a subset of each case, using Matplotlib's `imshow()`.

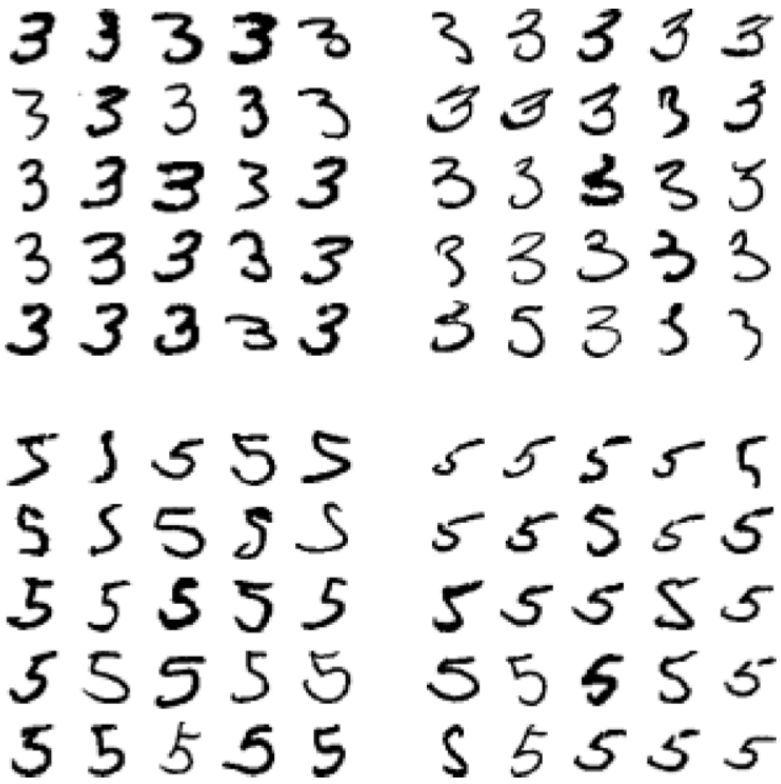


Figure 4: Two left blocks are classified as 3s; two right blocks are classified as 5s.

- `SGDClassifier` by default is a linear SVM, which is a linear model. When training finishes, each pixel gets a weight per class. For

classification of an image, it just sums up the weighted pixel intensities to get a score for each class.

Since 3s and 5s differ only by a few pixels, this model will easily confuse them.

- This classifier is quite sensitive to image shifting and rotation. One way to reduce the 3/5 confusion would be to preprocess the images to ensure that they are well centered and not too rotated.

Multilabel Classification

- In many applications, each instance may belong to multiple classes (as apposed to one class).
- Example: A book may be in one or several of the topic classes such as ‘religion’, ‘politics’, ‘finance’ or ‘education’.
- The encoding uses multiple binary values, one for each class.

  Example: A classifier has been trained to recognize three faces from photos: Alice, Bob, and Charlie. A photo with Alice and Charlie may have the output label $[1, 0, 1]$.
- Digit example:

```
from sklearn.neighbors import KNeighborsClassifier
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

This code creates a `y_multilabel` array containing two target labels for each digit image.

```
knn_clf.predict([some_digit])
```

```
array([[False,  True]])
```

Recall that `some_digit` is supposed to be 5.

How to Evaluate Multilabel Classification

- One approach is to measure the F_1 score for each individual target (or any other binary classifier metric discussed earlier), and then compute the average score across all targets.

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

```
0.976410265560605
```

- Or you can compute the weighted average of the F_1 scores. One simple option is to give each target a weight equal to its support (i.e., the number of instances with that target label). To do this, simply set `average="weighted"` in the preceding code.

Multiclass-Multioutput Classification

- Also known as Multioutput Classification or Multitask Classification.
- This is a generalization of the multilabel classification. Here, you have a multilabel for each instance and each entry of the multilabel (each target) has more than two values.
- Example: Classification of the properties “type of fruit” and “color” for a set of images of fruit. The property “type of fruit” has the possible classes: “apple”, “pear” and “orange”. The property “color” has the possible classes: “green”, “red”, “yellow” and “orange”.

  A multilabel consists of two entries, each representing one of the two properties.
- The following classifiers have built-in support for multioutput classifications:

  `tree.DecisionTreeClassifier`

`tree.ExtraTreeClassifier`

`ensemble.ExtraTreesClassifier`

`neighbors.KNeighborsClassifier`

`neighbors.RadiusNeighborsClassifier`

`ensemble.RandomForestClassifier`

- For classifiers that do not natively support multioutput classification, one can use Scikit-Learn's `MultiOutputClassifier` class and pass the chosen classifier as an argument.

  This strategy is to fit one classifier per target.

Multioutput Classification: Digit Image Cleanup

- Each instance's input is a noisy digit image, and the output is a clean digit image, represented as an array of pixel intensities, just like the MNIST images.

  Each multilabel has 784 entries, one for each pixel. Each pixel has 256 possible values.

  Note: This task can be accomplished with (multioutptut) regression. But, here we will use a multioutput classifier.

- Let us add noise to all the digit images in the MNIST dataset, we use the resulting images as the input. The output images are the original MNIST digit images.

```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

- Let us display a noisy digit after the added noise and compare it with the original one:

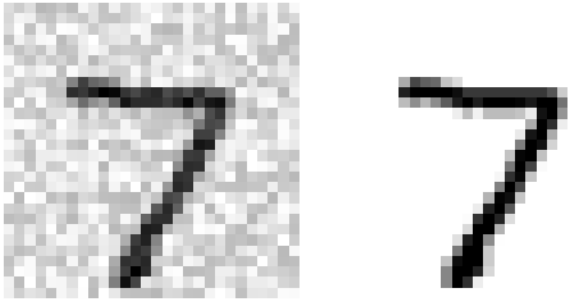


Figure 5: This is the first digit in the test set. The noisy image on the left is considered as an input; the image on the right is the correct output (label).

- We apply K-Nearest-Neighbor classifier to clean up the images.

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[0]])
```

- We can now display the cleaned up digit-7 (which is the predicted output).

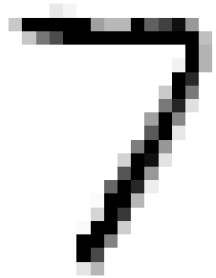


Figure 6: The cleaned-up digit-7.