Classification

Data

- MNIST is a dataset consisting of 70,000 images of digits handwritten by high school students and employees of the US Census Bureau.
- This dataset is an excellent start to machine learning.
- The data can be obtained through Scikit-Learn's **fetch_openml()** function.
    - There's also **load_openml()**, which can load small toy datasets and **make_openml()**, which generates fake datasets.
- As always, we can plot the data with Pyplot to see what it looks like.
- We need to create the training set and the test set.
- Note: All functions and packages mentioned below come from Scikit-Learn unless specified otherwise.

Training a Binary Classifier

- We want to classify the images as if they're a certain digit or not (i.e., if our digit is 5, we will have two classes, five and non-five.)
- The **SGDClassifier()** function deals with training instances independently; it can handle large datasets very efficiently.
    - We can fit our training data into the classifier and use it to predict its digit classification.

Measuring Accuracy Using Cross-Validation

- We can evaluate our SGDClassifier using the **cross_val_score()** function, which uses k_fold cross-validation.
    - Make sure that the scoring parameter is set to accuracy.
- This score tells us the ratio of correct predictions.
- We can also use the **DummyClassifier()** class to classify every image in the most frequent class.
    - Since most images will belong to the negative class (i.e., non-five), the dataset is skewed, so we may need an alternative performance measure.

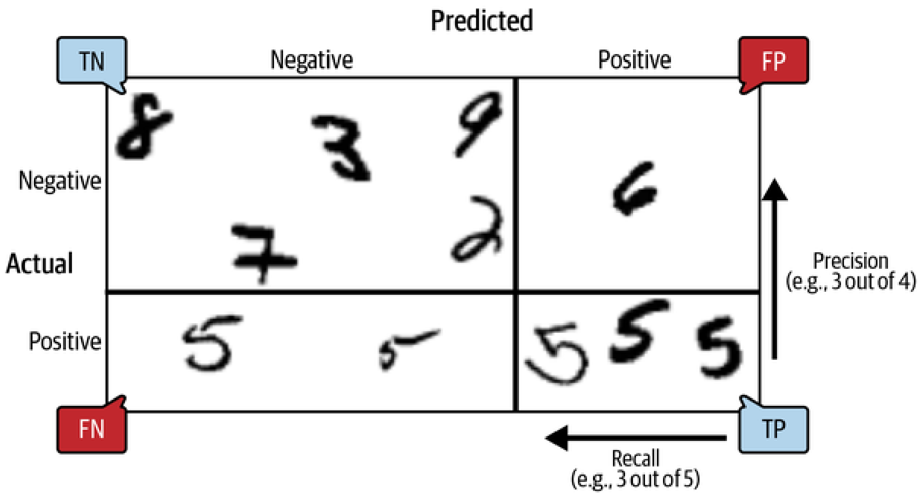
Confusion Matrices

- Confusion matrices count the number of instances of class A that are classified as class B for all A/B pairs.
    - For example, to know the number of times the classifier confused images of 8s with 0s, you would look at row #8, column #0 of the confusion matrix.
- First, we need a set of predictions. We can use the **cross_val_predict()** function using the training data.

    - This predict function runs k-fold cross-validation but returns the predictions instead of the evaluation scores.
    - This way, the model makes predictions on data it never saw during training.
- Each row in a confusion matrix represents an actual class, while each column represents a predicted class.
    -

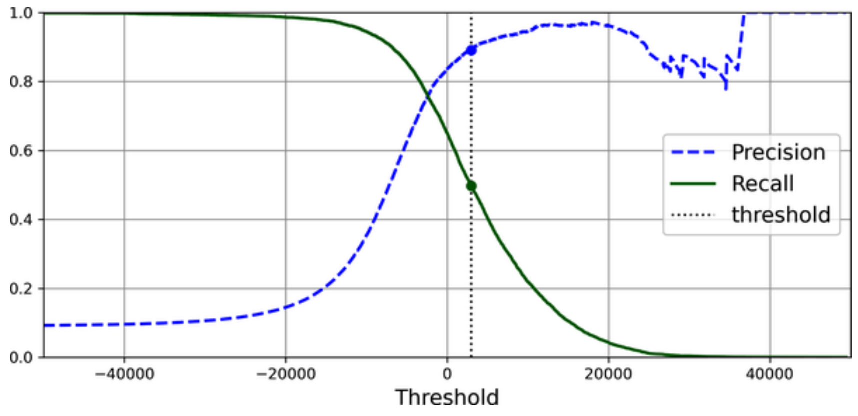
True Negatives	False Positives (Type I error)
False Negative (Type II error)	True Positive

    - A perfect classifier would have nonzero values only on the main diagonal (top left to bottom right.)
- Moreover, we can use precision and recall as more concise measures.
    - Precision is the accuracy of positive predictions: $precision = \frac{TP}{TP+FP}$
    - Recall or sensitivity is the ratio of positive instances that are correctly detected by the classifier: $recall = \frac{TP}{TP+FN}$



- We can use the **precision_score()** and **recall_score()** functions to calculate these two values.
- We can combine these two metrics into the F1 score, the harmonic mean of precision and recall.
    - The harmonic mean gives much more weight to low values.
    - $F1 = \frac{2}{\frac{1}{precision}+\frac{1}{recall}} = \frac{TP}{TP+\frac{(FN+FP)}{2}}$
    - The **f1_score()** function calculates this value for us.
    - Depending on your task, you may care more about getting more positive predictions – even if they are false positives – in which case, you would focus more on precision. The same logic applies to recall.
- One important detail to understand is that precision and recall are inversely related. This is called the Precision/Recall Trade-Off.
    - We can control this trade-off using a threshold. How do we decide which threshold to use?

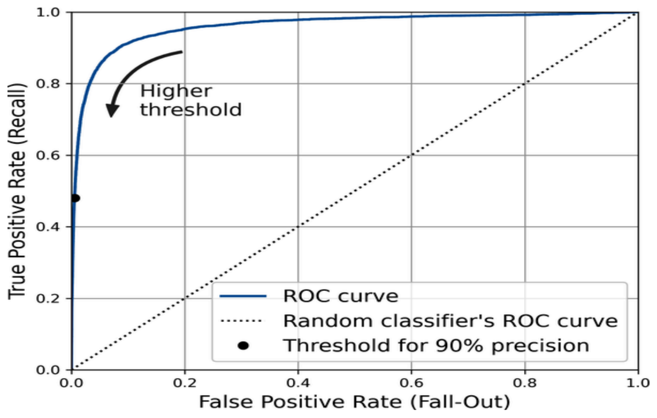
        - We want to use the cross_val_predict() function first to get the scores of all instances in the training set (set the method parameter to decision_function.)
        - We can then use the **precision_recall_curve()** function to compute the precision and recall for all possible thresholds. We can use this with Pyplot to see what this curve looks like.



        - If we want to find the threshold for a certain percentage of precision, we can use NumPy's **argmax()** method, which returns the first index of the maximum value.

The ROC Curve

- The receiver operating characteristic (ROC) curve plots the true positive rate (the recall) against the false positive rate.
- It is equal to 1 - the true negative rate (aka specificity), the ratio of negative instances correctly classified as negative.
- We can use the **roc_curve()** function to plot this curve and use it in conjunction with Pyplot.



- A good classifier will stay as far away from the dotted line as possible.
- We can compare classifiers by measuring the area under the curve (AUC) using the **roc_auc_score()** function.
    - A perfect classifier will have a ROC AUC of 1 while a truly random classifier will have a ROC AUC of 0.5.

    - 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. Otherwise, use the ROC curve.
- We can compare two classifiers using a PR curve or an ROC curve. For example, we can compare a **RandomForestClassifier()** with an SGDClassifier().
    - We can use the random forest classifier's **predict_proba()** to get class possibilities since it does not have a decision_function() like the SGD classifier.
    - Then, we can simply use cross-validation techniques, such as the cross-validation score or the PR curve, to compare their results.

Multiclass Classification

- Multiclass classifiers can distinguish between more than two classes. In our case, we need a system that classifies each digit image into one of 10 classes.
    - Some of Scikit-Learn's classifiers can handle multiple classes.
- One way is to train 10 binary classifiers, one for each digit. This is called a one-versus-the-rest (OvR) strategy.
- Another way is to train a binary classifier for every pair of digits. This is called a one-versus-one (OvO) strategy.
    - This means training N x (N - 1)/2 classifiers.
- Scikit-Learn automatically detects when we use a binary classification algorithm for a multiclass classification system. We can simply use the **SVC()** class to fit and predict.
- Alternatively, we can force Scikit-Learn to use one multiclass classification strategy.
    - For example, we can use the **OneVsRestClassifier()** function.
- Then, we can use cross-validation to check the algorithm's accuracy. If necessary, we can use a scaler.

Error Analysis

- Before reaching this step, we would normally evaluate other possible models and fine-tune their hyperparameters (with GridSearchCV) to see which model performs better.
- To check for any errors, we can use a confusion matrix (we can only use this matrix if we have predictions; if we don't, we should use the cross_val_predict() function.)
- We can use **ConfusionMatrixDisplay.from_predictions()** together with Pyplot to create a detailed confusion matrix.
    - Note that if the data is heavily skewed, we may have to normalize the matrix by dividing each value by the total number of images in the corresponding class.
    - To do this, we can simply set the normalize parameter to true.
- Note that algorithms cannot differentiate between handwritten digits the way our brains can. Algorithms are far more susceptible to classification errors.

Multilabel Classification

- In some cases, we want a classifier to output more than one class per instance.
    - For example, we may want to output more than just “True” and “False”.
- We can use the **KNeighborsClassifier()** to address these cases.
    - We may need to create an array with multiple labels. We can use **numpy.c_** to create this array.
    - Then, we can simply cross-validate and check the F1 score to evaluate this algorithm.
- It is possible to train a classifier to support multilabel classification.
    - We can use a classifier, like SVC, and train one model per label.
    - However, this will be difficult because we will only have access to one label and not the others, so we can’t use them simultaneously.
    - A solution could be organizing the model in a chain: when a model makes a prediction, it uses the input features plus all the predictions of the models that come before it in the chain.
    - We can simply use the **ClassifierChain()** function by feeding it another classifier, like the SVC.