Lecture 1: Classification metrics

Contents

Imports and LOs

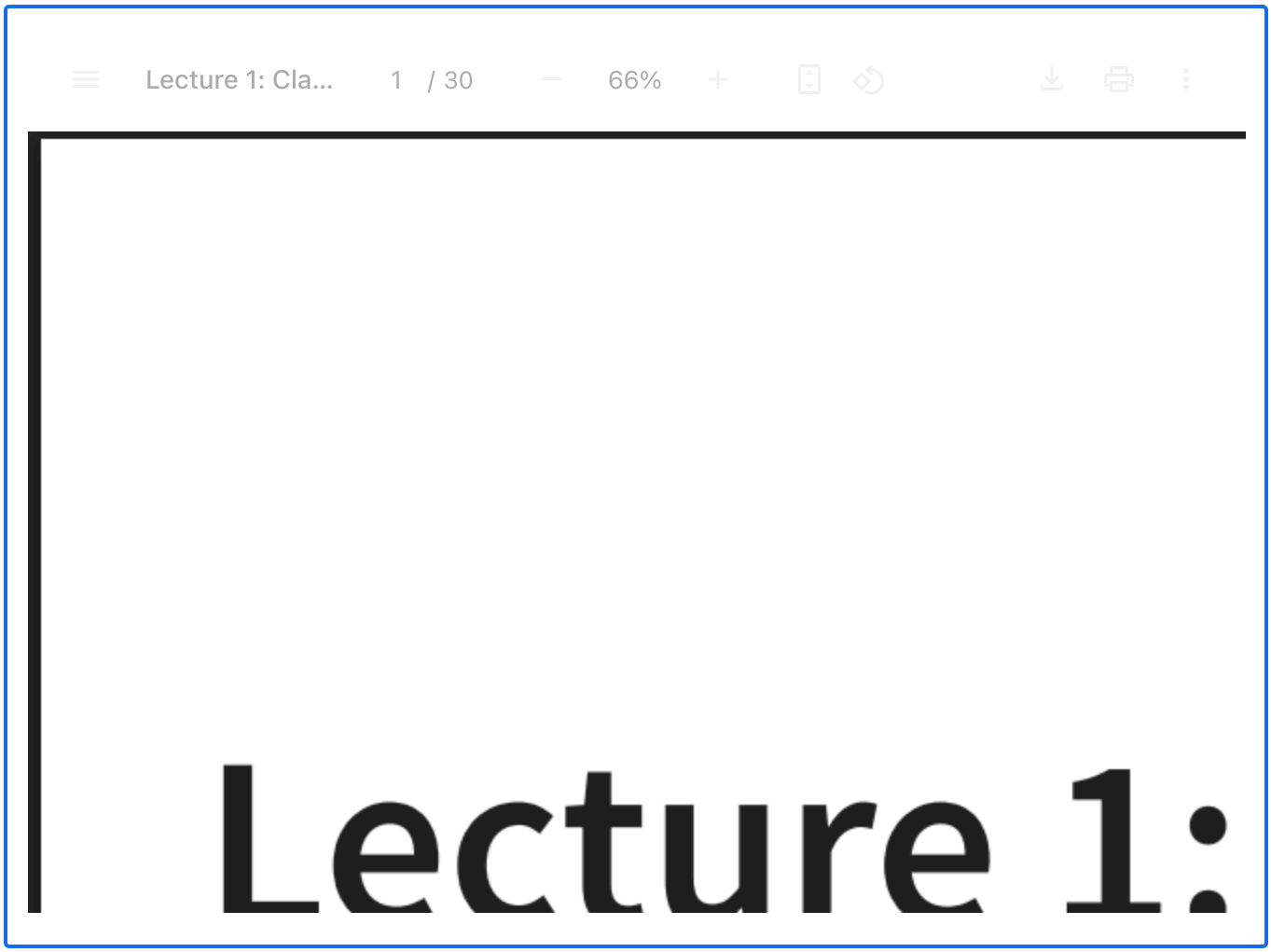
Learning outcomes

From this lecture, students are expected to be able to:

- Explain why accuracy is not always the best metric in ML.
- Explain components of a confusion matrix.
- Define precision, recall, and f1-score and use them to evaluate different classifiers.
- Broadly explain macro-average, weighted average.
- Interpret and use precision-recall curves.
- Explain average precision score.
- Interpret and use ROC curves and ROC AUC using `scikit-learn`.
- Identify whether there is class imbalance and whether you need to deal with it.
- Explain and use `class_weight` to deal with data imbalance.
- Assess model performance on specific groups in a dataset.

Lecture slides

Section 1 slides **Section 2 slides**

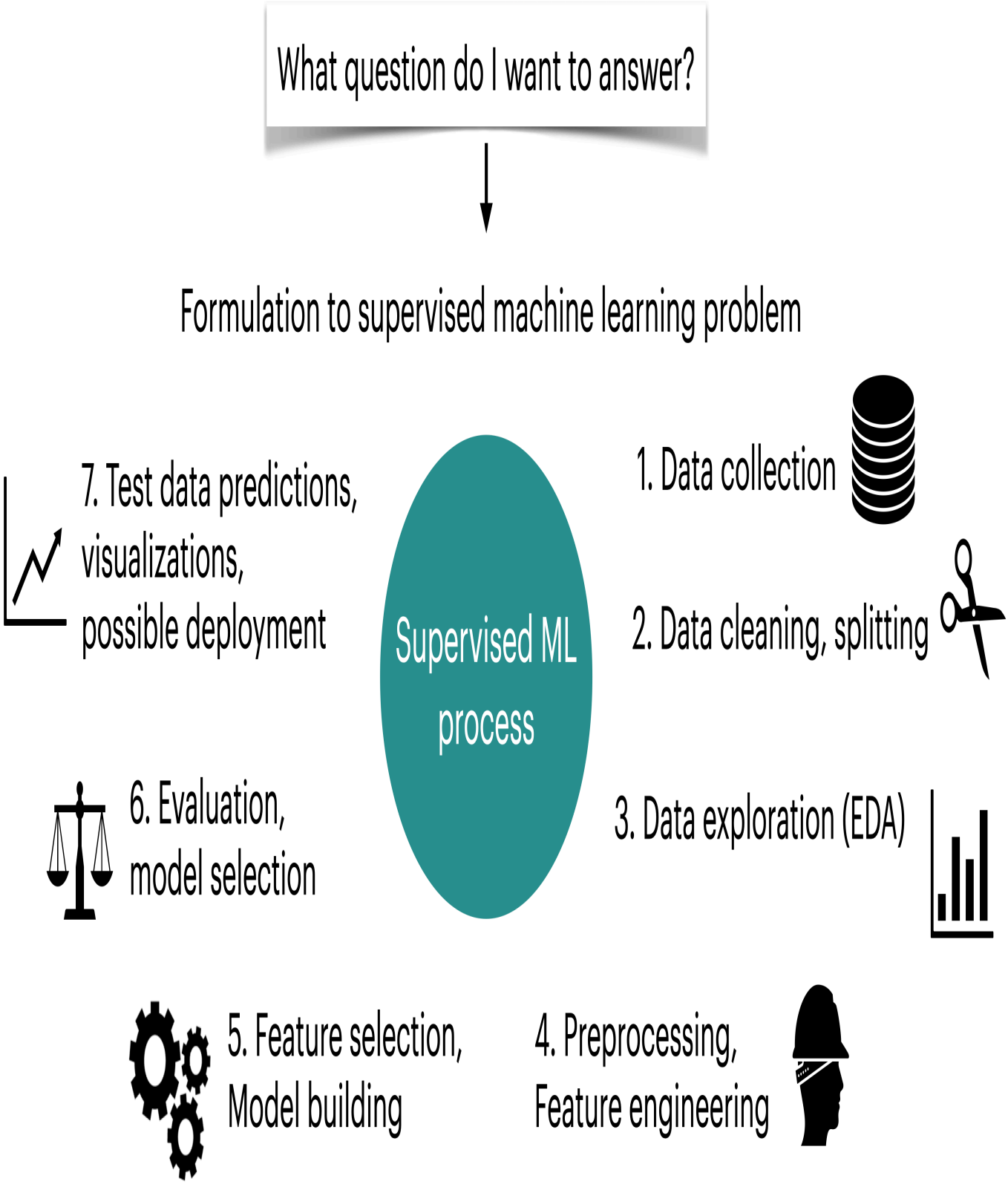


Imports

```
import os
import sys

sys.path.append("code/.")

import IPython
import matplotlib.pyplot as plt
import mglearn
import numpy as np
import pandas as pd
from IPython.display import HTML, display
from plotting_functions import *
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, cross_validate, train_tes
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler

# %matplotlib inline
pd.set_option("display.max_colwidth", 200)

from IPython.display import Image
# Ignore future deprecation warnings from sklearn (using `os` instead of `warn
import os
os.environ['PYTHONWARNINGS']='ignore::FutureWarning'
```

Machine learning workflow

- Here is a typical workflow of a supervised machine learning systems.
- In DSCI 571, we talked about data splitting, preprocessing, some EDA, model selection with hyperparameter optimization, and interpretation in the context of linear models.
- In DSCI 573, we will talk about, evaluation metrics and model selection in terms of evaluation metrics, feature engineering, feature selection, and model transparency and interpretation.



Evaluation metrics for binary classification:

Motivation

Dataset for demonstration

- Let's classify fraudulent and non-fraudulent credit card transactions using Kaggle's Credit Card Fraud Detection data set.

```
cc_df = pd.read_csv("data/creditcard.zip").assign(Class=lambda df: df['Class']
# Sorting columns so it is easier to read
cc_df = cc_df[['Class', 'Time', 'Amount'] + cc_df.columns[cc_df.columns.str.st

train_df, test_df = train_test_split(cc_df, test_size=0.3, random_state=111)

train_df
```

	Class	Time	Amount	V1	V2	V3	V4
64454	Non fraud	51150.0	1.00	-3.538816	3.481893	-1.827130	-0.573050
37906	Non fraud	39163.0	18.49	-0.363913	0.853399	1.648195	1.118934
79378	Non fraud	57994.0	23.74	1.193021	-0.136714	0.622612	0.780864
245686	Non fraud	152859.0	156.52	1.604032	-0.808208	-1.594982	0.200475
60943	Non fraud	49575.0	57.50	-2.669614	-2.734385	0.662450	-0.059077
...	...	...	...	...	...	...	...
105747	Non fraud	69669.0	43.05	1.222065	-0.557475	0.231881	-0.054993
102486	Non fraud	68226.0	49.00	-0.378726	0.176414	2.038723	1.075694
135892	Non fraud	81448.0	9.23	-2.201029	2.135296	-0.082015	-0.079166
10196	Non fraud	15772.0	87.00	-1.998332	-0.137916	2.196376	1.961751
129900	Non fraud	79237.0	1.00	1.245801	-0.448697	0.882059	-0.711534

199364 rows × 31 columns

- Good size dataset
- For confidentially reasons, it only provides features transformed with PCA (V1, V2, etc), which is a popular dimensionality reduction technique.
- A timestamp is also provided, as well as the amount of the transaction.
- The class is the target where 0 means “not fraudulent” and 1 means “fraudulent”

EDA

```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 199364 entries, 64454 to 129900
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Class   199364 non-null  object 
 1   Time    199364 non-null  float64
 2   Amount  199364 non-null  float64
 3   V1      199364 non-null  float64
 4   V2      199364 non-null  float64
 5   V3      199364 non-null  float64
 6   V4      199364 non-null  float64
 7   V5      199364 non-null  float64
 8   V6      199364 non-null  float64
 9   V7      199364 non-null  float64
 10  V8      199364 non-null  float64
 11  V9      199364 non-null  float64
 12  V10     199364 non-null  float64
 13  V11     199364 non-null  float64
 14  V12     199364 non-null  float64
 15  V13     199364 non-null  float64
 16  V14     199364 non-null  float64
 17  V15     199364 non-null  float64
 18  V16     199364 non-null  float64
 19  V17     199364 non-null  float64
 20  V18     199364 non-null  float64
 21  V19     199364 non-null  float64
 22  V20     199364 non-null  float64
 23  V21     199364 non-null  float64
 24  V22     199364 non-null  float64
 25  V23     199364 non-null  float64
 26  V24     199364 non-null  float64
 27  V25     199364 non-null  float64
 28  V26     199364 non-null  float64
 29  V27     199364 non-null  float64
 30  V28     199364 non-null  float64
dtypes: float64(30), object(1)
memory usage: 48.7+ MB
```

```
train_df.describe()
```

	Time	Amount	V1	V2	
count	199364.000000	199364.000000	199364.000000	199364.000000	199364.0
mean	94888.815669	88.164679	0.000492	-0.000726	0.0
std	47491.435489	238.925768	1.959870	1.645519	1.5
min	0.000000	0.000000	-56.407510	-72.715728	-31.8
25%	54240.000000	5.640000	-0.918124	-0.600193	-0.8
50%	84772.500000	22.000000	0.018854	0.065463	0.1
75%	139349.250000	77.150000	1.315630	0.803617	1.0
max	172792.000000	11898.090000	2.451888	22.057729	9.3

8 rows × 30 columns

- We do not have categorical features. All features are numeric.
- We have to be careful about the `Time` and `Amount` features.
- We could scale `Amount` to the same scale as the V1, V2, etc variables.
- Do we want to scale time?
  - We could, but in this lecture, we'll just drop the Time feature.
  - The temporal features we are interested in would be things like what time each day (or week, or month) a fraudulent transaction occurred and we will learn more about this type of preprocessing later.

Let's separate `X` and `y` for train and test splits.

```
X_train_big, y_train_big = train_df.drop(columns=["Class", "Time"]), train_df[
X_test, y_test = test_df.drop(columns=["Class", "Time"]), test_df["Class"]
```

- It's easier to demonstrate evaluation metrics using an explicit validation set instead of using cross-validation.
- So let's create a single validation set here just for learning purposes.
- Our data is large enough so it shouldn't be a problem.

```
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_big, y_train_big, test_size=0.3, random_state=123
)
```

Baseline

```
dummy = DummyClassifier()
dummy.fit(X_train, y_train)
dummy.score(X_valid, y_valid)
```

```
0.9982945995652901
```

Observations

- `DummyClassifier` is getting 0.998 cross-validation accuracy!!
- Should we be happy with this accuracy and deploy this `DummyClassifier` model for fraud detection?

Let's explore that question by looking at the proportion of each class:

```
train_df["Class"].value_counts(normalize=True)
```

```
Class
Non fraud    0.9983
Fraud        0.0017
Name: proportion, dtype: float64
```

- We have a rather large class imbalance.
- We have MANY non-fraud transactions and only a handful of fraud transactions.
- So in the training set, `most_frequent` strategy is labeling 199,025 (99.83%) instances correctly and only 339 (0.17%) instances incorrectly.
- Is this what we want?
- The "fraud" class is the important class that we want to spot.

Which such a high scoring baseline model, it becomes tricty to make any substantial improvements. Let's for example scale the features and try `LogisticRegression`.

```
pipe = make_pipeline(StandardScaler(), LogisticRegression())
pipe.fit(X_train, y_train)
pipe.score(X_valid, y_valid)
```

```
0.9992141782310651
```

```
y_train
```

```
80437     Non fraud
60984     Non fraud
128056    Non fraud
71109     Non fraud
172062    Non fraud
            ...
41235     Non fraud
264678    Non fraud
24998     Non fraud
241690    Non fraud
233612    Non fraud
Name: Class, Length: 139554, dtype: object
```

- We are getting a slightly better score with logistic regression, but is it a meaningful improvement?
- What score should be considered a good score in this case?
- Remember that the `score` method by default returns accuracy, which is defined as the proportion of correct predictions: $\$accuracy = \frac{\text{number of correct predictions}}{\text{total number of predictions}}\$$
- Is accuracy a good metric here?
- Is there anything more informative than accuracy that we can use?

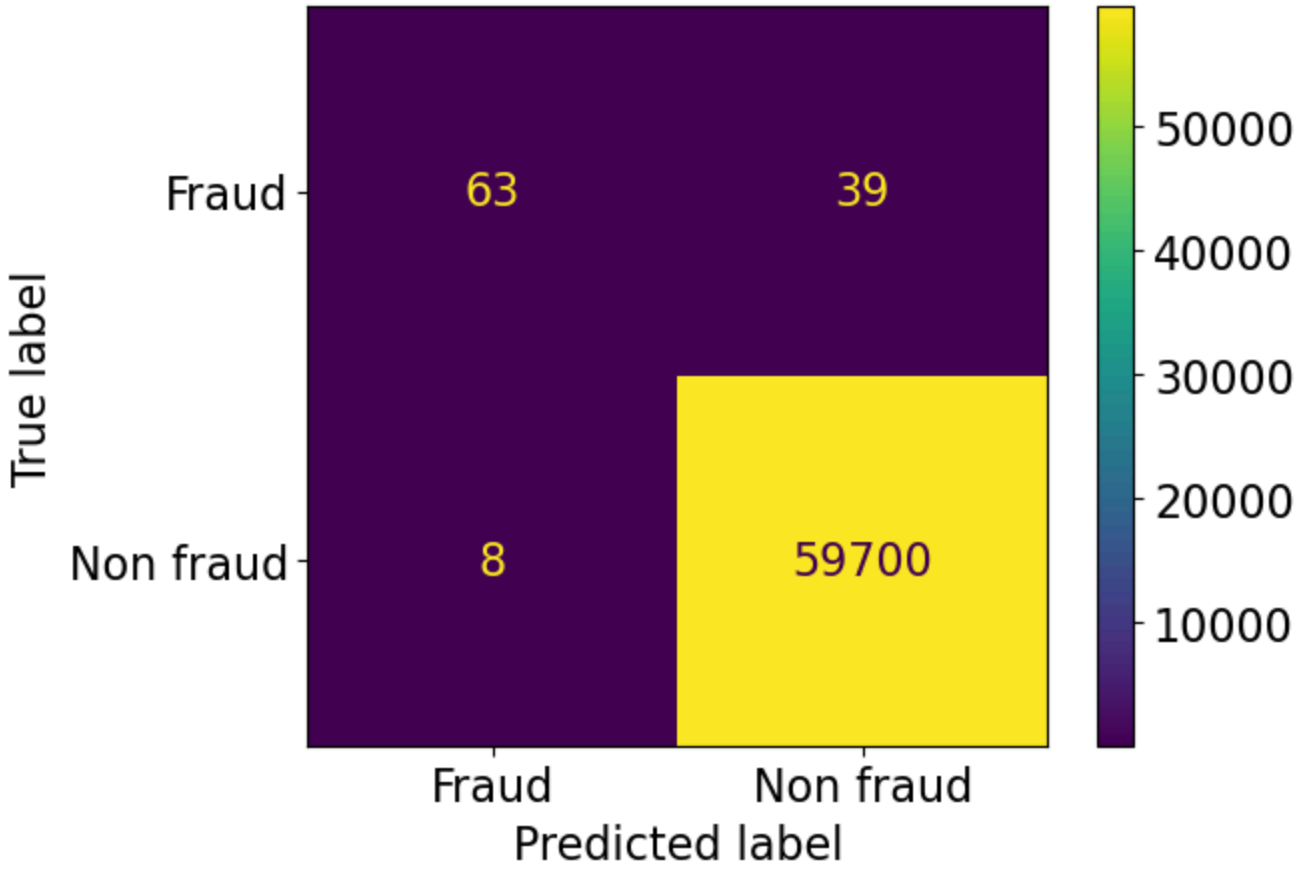
Let's dig a little deeper.

Confusion matrix

Instead of just reporting the accuracy, which tells us what proportion of the model predictions are correct, we can look into **how** the model predictions are correct and incorrect. A common

way of doing this is to create a **confusion matrix** which shows how each class' actual and predicted label.

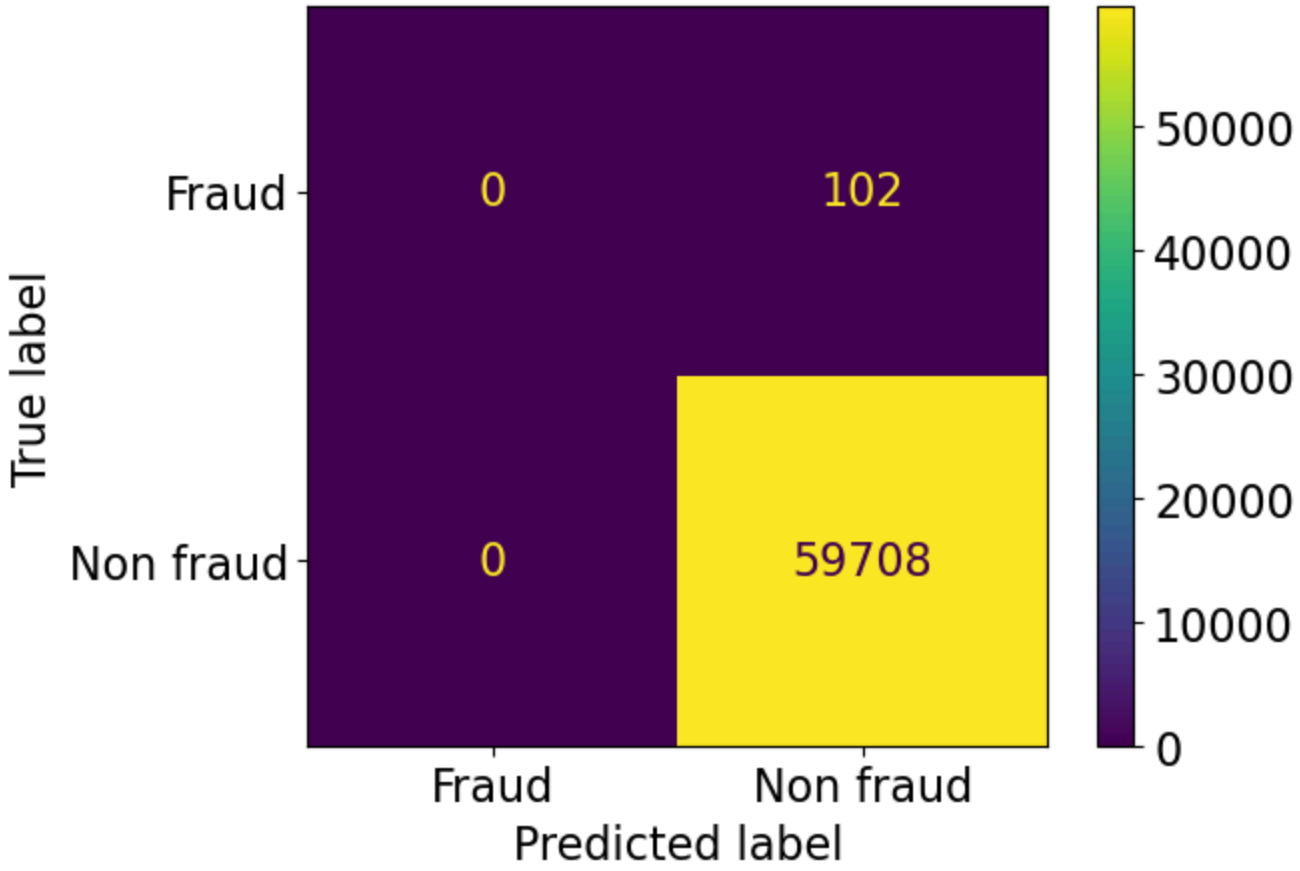
```
from sklearn.metrics import ConfusionMatrixDisplay

cm = ConfusionMatrixDisplay.from_estimator(
    pipe,
    X_valid,
    y_valid,
    values_format="d",  # Show the full number 59,700 instead of 6e+04,
    # normalize='all'  # Show the proportion in each square instead of the cou
)
```



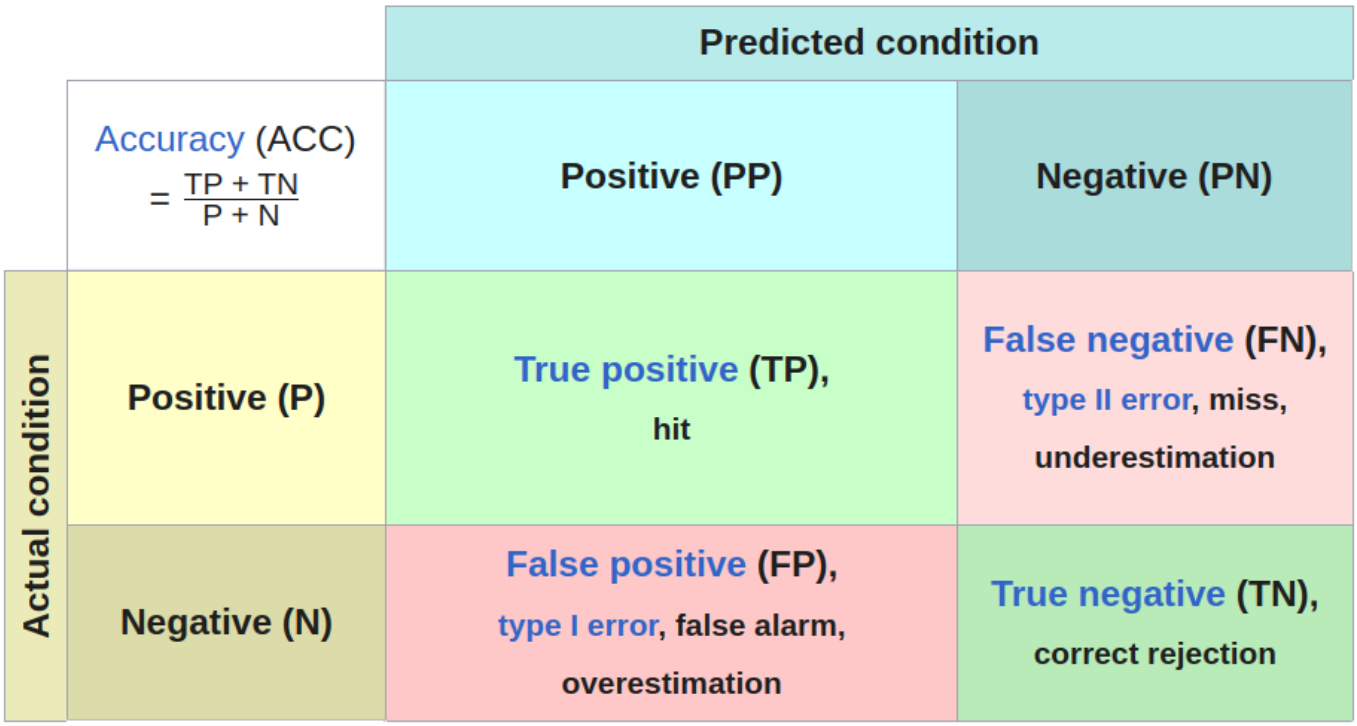
- Perfect prediction has all values down the diagonal
- Off diagonal entries can often tell us about what is being mis-predicted
- In this case it seems like we are doing really well for the non-fraud transactions, but for the fraudulent transactions, we're only predicting about 60% to be fradulent.
- Which quadrant/error do you think is more important here?
    - Probably the missed fradulent transactions since they would be very costly! There is still some cost associated with investigating the non-fradulent transactions so we need to consider that as well, but to a lesser extent.

As a comparison, our dummy matrix will always have all the predictions in the column for the most common class, in this case “Non fraud”.

```
ConfusionMatrixDisplay.from_estimator(
    dummy,
    X_valid,
    y_valid,
    values_format="d",
);
```



We can generalize the confusion matrix for a binary classification problem in a table like this:



How do we know which class is the “positive” one?

When we’re interested in spotting a class (spot fraud transaction, spot spam, spot disease, etc), the thing that we are interested in spotting is considered “positive”. Above we wanted to spot fraudulent transactions and so they are “positive”. If we’re not interested in one class more than the other, and just want to build a model that can distinguish between two classes, then we can big either as the positive one for the confusion matrix.

Using this more general notation, we could communicate additional insight into the errors of our model by saying that we had 8 false positives (type I error) in our confusion matrix above, and 38 false negatives. Since these numbers are only meaningful in the context of the total number of predictions, we could instead reporting them as proportions so that they are easier to interpret.

If you want access to the values without plotting them, you can use the `confusion_matrix` function.

With your newfound knowledge about confusion matrices, which of the following two statement do you agree with?

- In medical diagnosis, false positives are more damaging than false negatives (assuming “positive” means the person has a disease, “negative” means they don’t).
- In spam classification, false positives are more damaging than false negatives (assuming “positive” means the email is spam, “negative” means they it’s not).

▶ View answer

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_valid, pipe.predict(X_valid))
```

```
array([[   63,    39],
       [    8, 59700]])
```

Confusion matrix with cross-validation

- Above we used a confusion matrix on a single validation data set. This is just for educational purposes, since we will actually use multiple validation sets via CV to assess our model performance and tune hyperparameters.
- You can also calculate a confusion matrix with cross-validation, by using the `cross_val_predict` method.
- This adds up all the values of the TP, TN, FP, FN from the different CV folds, so that each part of the data is evaluated once (since we do crossvalidation with non-overlapping folds).

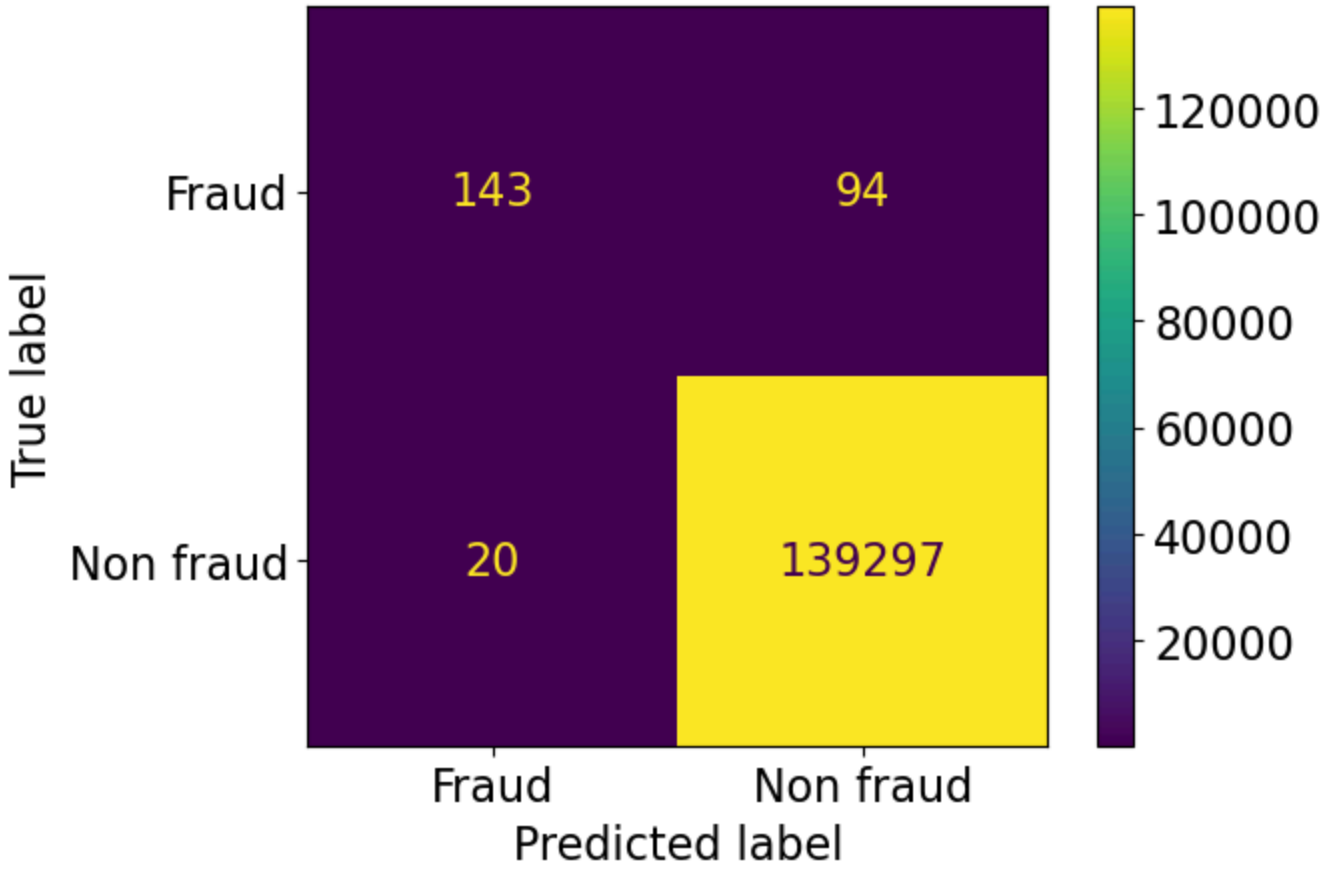
```
from sklearn.model_selection import cross_val_predict

confusion_matrix(y_train, cross_val_predict(pipe, X_train, y_train))
```

```
array([[   143,     94],
       [    20, 139297]])
```

- To plot it, we need to use the `from_predictions` method instead of the `from_estimator` method as we did above.

```
ConfusionMatrixDisplay.from_predictions(
    y_train,  # true class labels
    cross_val_predict(pipe, X_train, y_train),  # predicted class labels
    # Normalizing would be additionally helpful here since the overall number
    # to the number of rows in the data set (due to cross-validation) and is t
    normalize='all'
)
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x14dd18b60>
```



Precision and recall

While it is informative to inspect the confusion matrix to understand what our model is getting right and wrong, it can also useful to be able to report a single number of our model performance, e.g. for hyperparameter tuning or to compare many different models.

How can we distill the information from the confusion matrix into a single number? There are many metrics for this and they all focus on slightly different things. In this image, we can see eight metrics that are commonly calculated based on the TP, FP, TN, and FP from a confusion matrix. Note that the metrics are symmetric in the rows and columns (e.g. the False Positive Rate is the same as (1 - True Negative Rate), so there are really only four calculations to make:

		Predicted condition			
	Accuracy (ACC) $= \frac{TP + TN}{P + N}$	**Positive (PP)**	**Negative (PN)**		
Actual condition	**Positive (P)**	**True positive (TP),** hit	**False negative (FN),** type II error, miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{TP}{P} = 1 - FNR$	False negative rate (FNR), miss rate $= \frac{FN}{P} = 1 - TPR$
	Negative (N)	**False positive (FP),** type I error, false alarm, overestimation	**True negative (TN),** correct rejection	False positive rate (FPR), probability of false alarm, fall-out $= \frac{FP}{N} = 1 - TNR$	True negative rate (TNR), specificity (SPC), selectivity $= \frac{TN}{N} = 1 - FPR$
		Positive predictive value (PPV), precision $= \frac{TP}{PP} = 1 - FDR$	False omission rate (FOR) $= \frac{FN}{PN} = 1 - NPV$		
		False discovery rate (FDR) $= \frac{FP}{PP} = 1 - PPV$	Negative predictive value (NPV) $= \frac{TN}{PN}$ $= 1 - FOR$		

You don't have to memorize all the names of these metrics, but make sure you understand the general principal of how they are calculated. Two metrics that are particularly common to use in machine learning context (and important to memorize) are **precision** (also called PPV) and **recall** (also called TPR).

As we mentioned earlier, we're often more interested in the positive class and both these metrics are defined with that in mind as they both include the True Positives in the nominator. These metrics will help us assess our model, and summarize important aspects of the confusion matrix.

Precision

Among the positive predictions, what proportion was actually positive?

Precision is the ability of the classifier to putting a positive label on a positive observation. You can remember this as "How **p**recise are the model's **p**redictions?".

$$precision = \frac{TP}{TP + FP} = \frac{Correctly\ positive\ predictions}{All\ positive\ predictions}$$

Recall

Out of all actual positives, what proportion did the model identify?

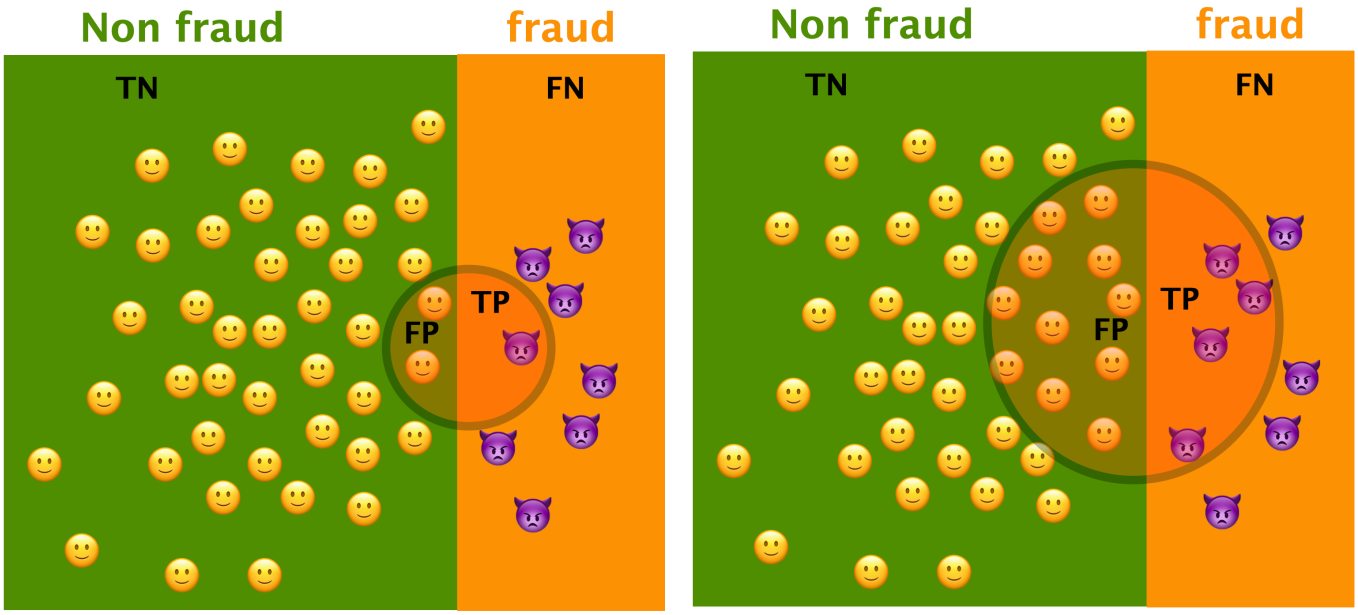
Recall is the ability of the classifier to find all the positive samples. You can remember it as "What proportion of the **r**eal positives the did the model **r**ecall.

$$recall = \frac{TP}{TP + FN} = \frac{Correctly\ positive\ predictions}{All\ actual\ positives}$$

Toy example

- Imagine that your model has identified everything outside the circle as non-fraud and everything inside the circle as fraud.

TN → True Negatives (non-fraud predicted as as non-fraud)
TP → True Positives (fraud predicted as fraud)
FN → False Negatives (fraud predicted as non-fraud)
FP → False Positives (non-fraud predicted as fraud)



$$Precision = \frac{1}{3} \qquad Precision = \frac{TP}{TP + FP} \qquad \downarrow Precision = \frac{4}{15}$$

$$Recall = \frac{1}{8} \qquad Recall = \frac{TP}{TP + FN} \qquad \uparrow Recall = \frac{4}{8}$$

Real example

Let's compute precision and recall on our data. Recall what the confusion matrix looks like:

```
ConfusionMatrixDisplay.from_estimator(
    pipe,
    X_valid,
    y_valid,
    values_format="d",
);
```

We access the numbers via the `confusion_matrix` function we saw earlier.

```
TP, FN, FP, TN = confusion_matrix(y_valid, pipe.predict(X_valid)).flatten()
print(TP, FN, FP, TN)
```

```
63 39 8 59700
```

```
precision = TP / (TP + FP)
precision
```

```
0.8873239436619719
```

```
recall = TP / (TP + FN)
recall
```

```
0.6176470588235294
```

We don't have to compute these by hand as there are built-in function in sklearn for them. We can see that our manual calculation is the same as the sklearn functions.

```
from sklearn.metrics import precision_score, recall_score

precision_score(y_valid, pipe.predict(X_valid), pos_label='Fraud')  # We need
```

```
0.8873239436619719
```

```
from sklearn.metrics import precision_score, recall_score

precision_score(y_valid, dummy.predict(X_valid), pos_label='Fraud')  # We need
```

```
/opt/miniconda3/envs/573/lib/python3.12/site-packages/sklearn/metrics/_classif
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
0.0
```

```
recall_score(y_valid, pipe.predict(X_valid), pos_label='Fraud')
```

```
0.6176470588235294
```

```
recall_score(y_valid, pipe.predict(X_valid), pos_label='Fraud')
```

```
0.6176470588235294
```

Drawbacks of precision and recall

While precision and recall are informative metrics to report. They often lead to subpar results when optimized on in isolation. To understand this, we can think of the extremes of the metrics.

Since `recall = TP / (TP + FN)` it will minimize FN to achieve a score closer to 1. An undesired way of minimizing FN would be to predict all samples to be positive. This would likely lead to high FP and low TN, but that is not part of the definition of recall.

Similarly, since `precision = TP / (TP + FP)` it will minimize FP to achieve a score closer to 1. An undesired way of minimizing FP would be to minimize the number of positive predictions made. For example, if we just make a single positive prediction and it is correct, we have maxed out precision, regardless of how many actual positives we have missed (so likely leading to high FN).

To avoid these pitfalls, we want to use a metric that looks at both the precision and recall. One such metric that is commonly used is the F1-score.

F1-score

F1-score combines precision and recall to give one score, which could be used in hyperparameter optimization. It is the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

```
# Using our definitions of `precision` and `recall` from above
f1_score = (2 * precision * recall) / (precision + recall)
f1_score
```

```
0.7283236994219653
```

```
from sklearn.metrics import f1_score

f1_score(y_valid, pipe.predict(X_valid), pos_label='Fraud')  # Recall and prec
```

```
0.7283236994219653
```

In the F1 score, both the precision and recall are regarded as equally important. However, in real life applications we might care more about one than the other but still want to hav a single number. For this we can use the more general Fbeta sore, where beta is the weight of recall vs precision. For example, beta = 2 means that recall is weighted twice as much as precision when computing the score; which we would refer to as the F2 score. Finding a suitable number for beta is context dependent and would require domain expertise in the specific field that the model would be deployed in.

```
from sklearn.metrics import fbeta_score

fbeta_score(y_valid, pipe.predict(X_valid), pos_label='Fraud', beta=1)  # Same
```

```
0.7283236994219653
```

```
fbeta_score(y_valid, pipe.predict(X_valid), pos_label='Fraud', beta=2)  # Reca
```

```
0.6576200417536534
```

Although F1 (and Fbeta more generally) are useful metrics that are commonly used to tune our models, it is important to understand that they are not the perfectly suitable for all scenarios. For example, imagine if we cared about the number of True Negatives in our problem, this is not taken into account at all in the F1 metric. There are other metrics that try to summarize the entire confusion matrix and all its basic rates (TPR, TNR, FPR, FNR). One of the most commonly used is Matthews Correlation Coefficient (MCC), which is available in sklearn and can be good to be aware of for problems where the TN are important:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP) \cdot (TP + FN) \cdot (TN + FP) \cdot (TN + FN)}}$$

Classification report

There is a convenient function called `classification_report` in `sklearn` which prints out the three metrics we have discussed all at once.

```
from sklearn.metrics import classification_report

print(
    classification_report(
        y_valid, pipe.predict(X_valid)
    )
)
```

```
              precision    recall  f1-score   support

       Fraud       0.89      0.62      0.73       102
   Non fraud       1.00      1.00      1.00     59708

    accuracy                           1.00     59810
   macro avg       0.94      0.81      0.86     59810
weighted avg       1.00      1.00      1.00     59810
```

(Optional) Macro average and weighted average

Macro average

- Gives equal importance to all classes and average over all classes.
- For instance, in the example above, recall for non-fraud is 1.0 and fraud is 0.63, and so macro average is 0.81.
- More relevant in case of multi-class problems.

Weighted average

- Weighted by the number of samples in each class.
- Divide by the total number of samples.

Which one is relevant when depends upon whether you think each class should have the same weight or each sample should have the same weight.

Toy example

```
from sklearn.metrics import classification_report
y_true_toy = [0, 1, 0, 1, 0]
y_pred_toy = [0, 0, 0, 1, 0]
target_names_toy = ['class 0', 'class 1']
print(classification_report(y_true_toy, y_pred_toy, target_names=target_names_
```

```
              precision    recall  f1-score   support

     class 0       0.75      1.00      0.86         3
     class 1       1.00      0.50      0.67         2

    accuracy                           0.80         5
   macro avg       0.88      0.75      0.76         5
weighted avg       0.85      0.80      0.78         5
```

- weighted average is weighted by the proportion of examples in a particular class. So for the toy example above:
- weighted_average precision: 3/5 * 0.75 + 2/5 * 1.00 = 0.85
- weighted_average recall: 3/5 * 1.00 + 2/5 * 0.5 = 0.80
- weighted_average f1-score: 3/5 * 0.86 + 2/5 * 0.67 = 0.78

- macro average gives equal weight to both classes. So for the toy example above:
- macro average precision: 0.5 * 0.75 + 0.5 * 1.00 =0. 875
- macro average recall: 0.5 * 1.00 + 0.5 * 0.5 =0. 75
- macro average f1-score: 0.5 * 0.75 + 0.5 * 1.00 =0.765

Interim summary

- Accuracy is often not sufficient when scoring/comparing models and it is especially misleading when there is class imbalance.
- A confusion matrix provides a way to break down errors made by our model and understand how the model is wrong.
- We looked at three metrics based on confusion matrix:
  - precision, recall, f1-score.

- Note that what you consider "positive" (fraud in our case) is important when calculating precision, recall, and f1-score.
- If you flip what is considered positive or negative, we'll end up with different TP, FP, TN, FN, and hence different precision, recall, and f1-scores.

Cross validation with different metrics

- We can pass different evaluation metrics with `scoring` argument of `cross_validate`.

```
from sklearn.metrics import make_scorer

scoring = {
    "accuracy": 'accuracy',
    # We need to use `make_scorer` in order to set parameters such as pos_labe
    'f1': make_scorer(f1_score, pos_label='Fraud'),
    'f2': make_scorer(fbeta_score, pos_label='Fraud', beta=2),
    'recall': make_scorer(recall_score, pos_label='Fraud'),
    'precision': make_scorer(precision_score, pos_label='Fraud'),
}  # scoring can be a string, a list, or a dictionary
pipe = make_pipeline(StandardScaler(), LogisticRegression())
scores = cross_validate(
    pipe, X_train_big, y_train_big, return_train_score=True, scoring=scoring
)
pd.DataFrame(scores)
```

	fit_time	score_time	test_accuracy	train_accuracy	test_f1	train_f1	
0	0.227572	0.646450	0.999122	0.999335	0.700855	0.772532	0.
1	0.235183	0.635062	0.999223	0.999317	0.735043	0.765591	0.
2	0.239476	0.657359	0.999298	0.999210	0.754386	0.724891	0
3	0.240652	0.642117	0.999172	0.999254	0.697248	0.742981	0.
4	0.225068	0.632709	0.999147	0.999185	0.696429	0.716157	0.

- You read more about creating your own scoring function.
- Grid search also has a similar `scoring` parameter.

(Optional) Evaluation metrics for multi-class classification

Let's examine precision, recall, and f1-score of different classes in the HappyDB corpus.

```
df = pd.read_csv("data/cleaned_hm.csv", index_col=0)
sample_df = df.dropna()
sample_df.head()
sample_df = sample_df.rename(
    columns={"cleaned_hm": "moment", "ground_truth_category": "target"}
)
sample_df.head()
```

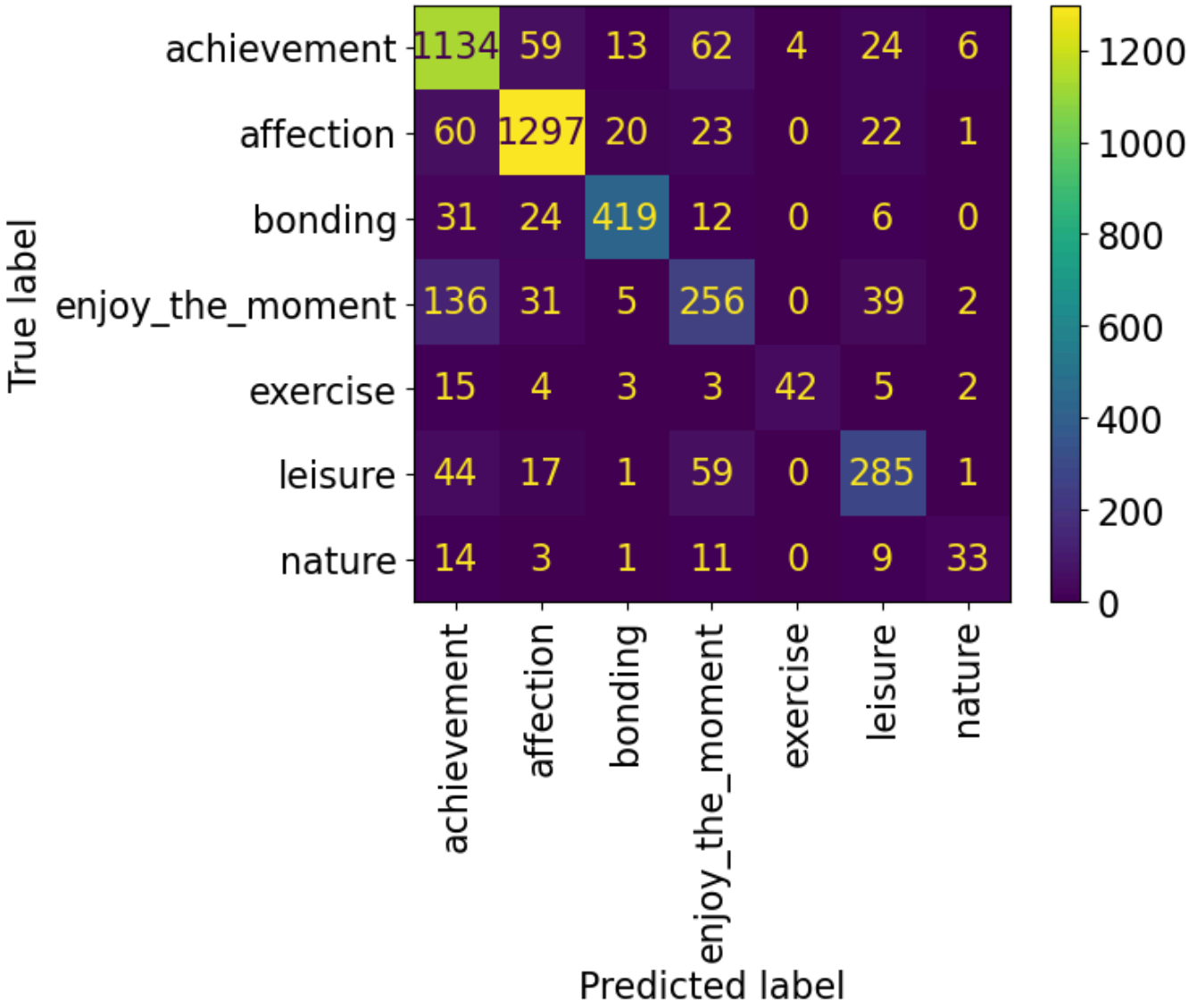
hmid	wid	reflection_period	original_hm	moment	modified	num_sente
27676	206	24h	We had a serious talk with some friends of ours who have been flaky lately. They understood and we had a good evening hanging out.	We had a serious talk with some friends of ours who have been flaky lately. They understood and we had a good evening hanging out.	True	
27678	45	24h	I meditated last night.	I meditated last night.	True	
27697	498	24h	My grandmother start to walk from the bed after a long time.	My grandmother start to walk from the bed after a long time.	True	
27705	5732	24h	I picked my daughter up from the airport and we have a fun and good conversation on the way home.	I picked my daughter up from the airport and we have a fun and good conversation on the way home.	True	
27715	2272	24h	when i received flowers from my best friend	when i received flowers from my best friend	True	

```
train_df, test_df = train_test_split(sample_df, test_size=0.3, random_state=12
X_train_happy, y_train_happy = train_df["moment"], train_df["target"]
X_test_happy, y_test_happy = test_df["moment"], test_df["target"]
```

```
from sklearn.feature_extraction.text import CountVectorizer

pipe_lr = make_pipeline(
    CountVectorizer(stop_words="english"), LogisticRegression(max_iter=2000)
)
```

```
pipe_lr.fit(X_train_happy, y_train_happy)
pred = pipe_lr.predict(X_test_happy)
```

```
ConfusionMatrixDisplay.from_estimator(
    pipe_lr, X_test_happy, y_test_happy, xticks_rotation="vertical"
);
```



The prediction and recall in a multi-class setting are defined as One-vs-the-rest (OvR), where the class of interested is picked as the positive and evertyhing else is negative.

```
print(classification_report(y_test_happy, pred))
```

```
                  precision    recall  f1-score   support

     achievement       0.79      0.87      0.83      1302
       affection       0.90      0.91      0.91      1423
         bonding       0.91      0.85      0.88       492
enjoy_the_moment       0.60      0.55      0.57       469
        exercise       0.91      0.57      0.70        74
         leisure       0.73      0.70      0.72       407
          nature       0.73      0.46      0.57        71

        accuracy                           0.82      4238
       macro avg       0.80      0.70      0.74      4238
    weighted avg       0.82      0.82      0.82      4238
```

- Seems like there is a lot of variation in the scores for different classes. The model is performing pretty well on *affection* class but not that well on *enjoy_the_moment* and *nature* classes.
- If each class is equally important for you, pick macro avg as your evaluation metric.
- If each example is equally important, pick weighted avg as your metric.

Precision-recall curve

- Confusion matrix provides a detailed break down of the errors made by the model.
- But when creating a confusion matrix, we are using "hard" predictions.
- Most classifiers in `scikit-learn` provide `predict_proba` method (or `decision_function`) which provides degree of certainty about predictions by the classifier.
- Can we explore the degree of uncertainty to understand and improve the model performance?

Let's revisit the classification report on our fraud detection example.

```
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression())
pipe_lr.fit(X_train, y_train);
```

```
y_pred = pipe_lr.predict(X_valid)
print(classification_report(y_valid, y_pred))
```

```
              precision    recall  f1-score   support

       Fraud       0.89      0.62      0.73       102
   Non fraud       1.00      1.00      1.00     59708

    accuracy                           1.00     59810
   macro avg       0.94      0.81      0.86     59810
weighted avg       1.00      1.00      1.00     59810
```

By default, predictions use the threshold of 0.5. If `predict_proba` > 0.5, predict "fraud" else predict "non-fraud". We can see this by manually typing in the 0.5 threshold and getting the same results as above.

```
fraud_column = np.where(pipe_lr.classes_ == 'Fraud')[0][0]

y_pred = np.array(['Non fraud'] * X_valid.shape[0])  # Base array with all "No
y_pred[pipe_lr.predict_proba(X_valid)[:, fraud_column] > 0.5] = 'Fraud'  # Ove
print(classification_report(y_valid, y_pred))
```

```
              precision    recall  f1-score   support

       Fraud       0.89      0.62      0.73       102
   Non fraud       1.00      1.00      1.00     59708

    accuracy                           1.00     59810
   macro avg       0.94      0.81      0.86     59810
weighted avg       1.00      1.00      1.00     59810
```

- Suppose for your business it is more costly to miss fraudulent transactions and suppose you want to achieve a recall of at least 75% for the "fraud" class.
- One way to do this is by changing the threshold of `predict_proba`.
  - `predict` returns 1 when `predict_proba`'s probabilities are above 0.5 for the "fraud" class.

Key idea: what if we threshold the probability at a smaller value so that we identify more examples as "fraud" examples?

Let's lower the threshold to 0.1. In other words, predict the examples as "fraud" if `predict_proba` > 0.1.

```
y_pred_lower_threshold = np.array(['Non fraud'] * X_valid.shape[0])  # Base ar
y_pred_lower_threshold[pipe_lr.predict_proba(X_valid)[:, fraud_column] > 0.1]
```

```
print(classification_report(y_valid, y_pred_lower_threshold))
```

```
              precision    recall  f1-score   support

       Fraud       0.77      0.75      0.76       102
   Non fraud       1.00      1.00      1.00     59708

    accuracy                           1.00     59810
   macro avg       0.88      0.88      0.88     59810
weighted avg       1.00      1.00      1.00     59810
```

Operating point

- Now our recall for "fraud" class is >= 0.75.
- Setting a requirement on a classifier (e.g., recall of >= 0.75) is called setting the **operating point**.
- It's usually driven by business goals and is useful to make performance guarantees to customers.

Precision/Recall tradeoff

- But there is a trade-off between precision and recall.
- If you identify more things as "fraud", recall is going to increase but there are likely to be more false positives.

Let's sweep through different thresholds.

```
from collections import defaultdict

metrics = defaultdict(list)  # To be able to append to each entry in the dict

for threshold in np.arange(0.1, 1, 0.2):
    preds = np.array(['Non fraud'] * X_valid.shape[0])
    preds[pipe_lr.predict_proba(X_valid)[:, fraud_column] > threshold] = 'Frau
    metrics['threshold'].append(threshold)
    metrics['precision'].append(np.round(precision_score(y_valid, preds, pos_l
    metrics['recall'].append(np.round(recall_score(y_valid, preds, pos_label='

pd.DataFrame(metrics)
```

	threshold	precision	recall
0	0.1	0.7700	0.7549
1	0.3	0.7955	0.6863
2	0.5	0.8873	0.6176
3	0.7	0.8939	0.5784
4	0.9	0.8909	0.4804

Decreasing the threshold

- Decreasing the threshold means a lower bar for predicting fraud.
  - You are willing to risk more false positives in exchange of more true positives.
  - Recall would either stay the same or go up and precision is likely to go down
  - Occasionally, precision may increase if all the new examples after decreasing the threshold are TPs.

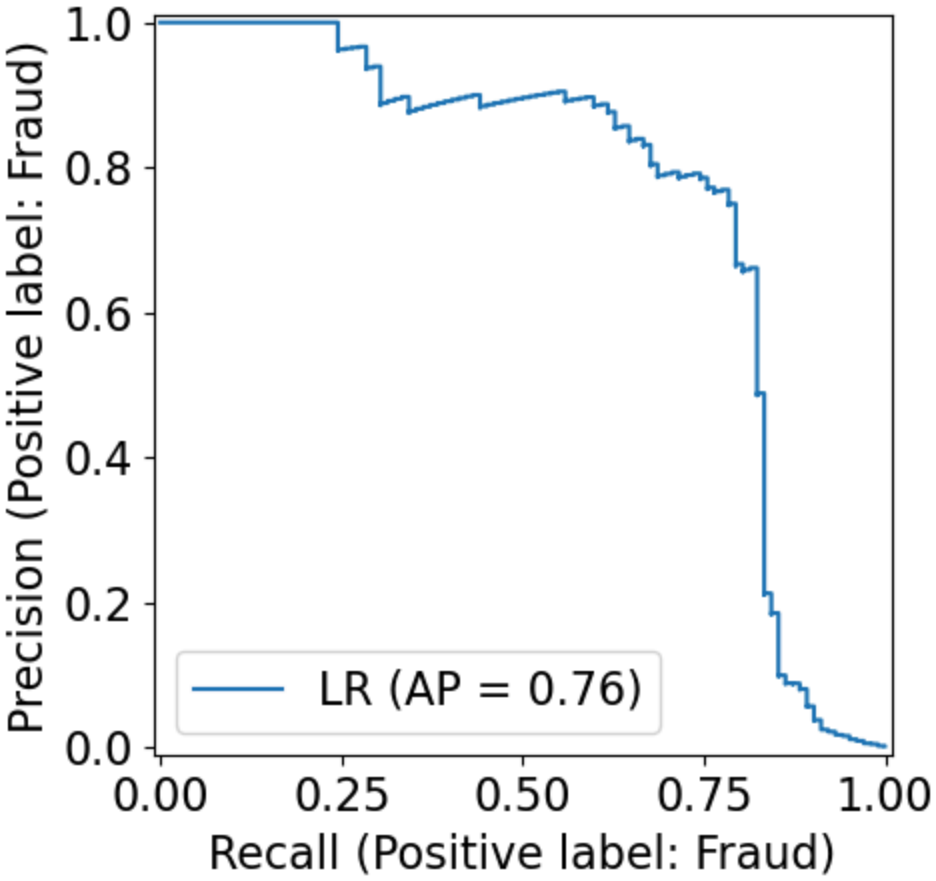
Increasing the threshold

- Increasing the threshold means a higher bar for predicting fraud.
  - Recall would go down or stay the same but precision is likely to go up
  - Occasionally, precision may go down if TP decrease but FP do not decrease.

Precision-recall curve

Often, when developing a model, it's not always clear what the operating point will be and to understand the the model better, it's informative to look at all possible thresholds and corresponding trade-offs of precision and recall in a plot. This type of plot is often called a PR curve, and the top-right would be a perfect classifier (precision = recall = 1).

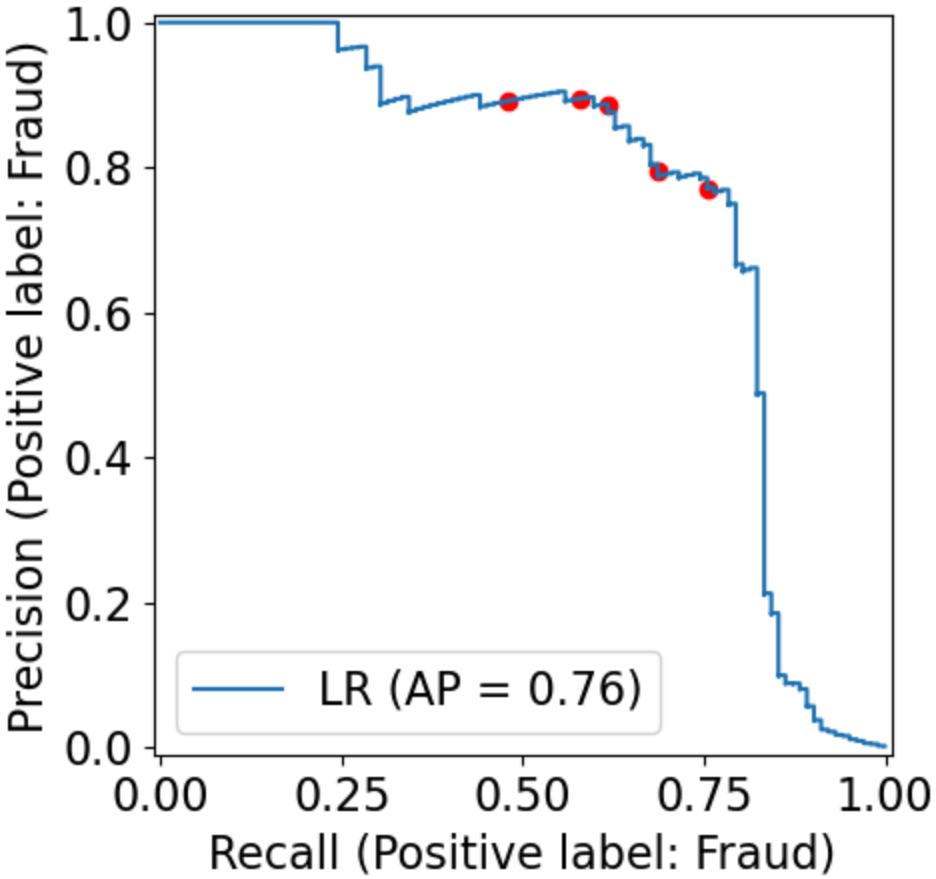
```
from sklearn.metrics import PrecisionRecallDisplay

PrecisionRecallDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    pos_label='Fraud',
    name='LR',  # For Logistic Regression
);
```



We can add the points for the thresholds we included in our `metrics` dataframe. As we can see below, the middle red point (the 0.5 default threshold), does seem to strike a good balance between precision and recall. However, the far right point (the 0.1 threshold) is a

good alternative, particularly if recall would be more highly valued than precision. (You don't need to remember the plotting code for adding the points, it is just for illustration).

```
PrecisionRecallDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    pos_label='Fraud',
    name='LR'
)
# Add a few threshold values as points
plt.scatter(
    x='recall',
    y='precision',
    data=pd.DataFrame(metrics),
    color='red',
);
```



- The probability threshold is not shown here, but it's going from 0 (lower-right) to 1 (upper left).
- At a threshold of 0 (lower right), we are classifying everything as "fraud", so there are no false negatives.
- Raising the threshold increases the precision but at the expense of lowering the recall.

- At the extreme left, where the threshold is 1, we get into the situation where all the examples classified as “fraud” are actually “fraud” (although they might be very few); we have no false positives.
- Usually the goal is to keep recall high as precision goes up.

If you want access to the underlying values directly as a dataframe, you can use the function `precision_recall_curve`.

```
from sklearn.metrics import precision_recall_curve

pd.DataFrame(
    precision_recall_curve(
        y_valid,
        pipe_lr.predict_proba(X_valid)[:, fraud_column],
        pos_label='Fraud',
    ),
    index=['precision', 'recall', 'threshold']
).T
```

	precision	recall	threshold
0	0.001705	1.000000	5.175855e-09
1	0.001705	1.000000	1.041297e-08
2	0.001705	1.000000	4.313910e-08
3	0.001705	1.000000	7.319365e-08
4	0.001706	1.000000	8.480516e-08
...	...	...	...
59045	1.000000	0.039216	1.000000e+00
59046	1.000000	0.029412	1.000000e+00
59047	1.000000	0.019608	1.000000e+00
59048	1.000000	0.009804	1.000000e+00
59049	1.000000	0.000000	NaN

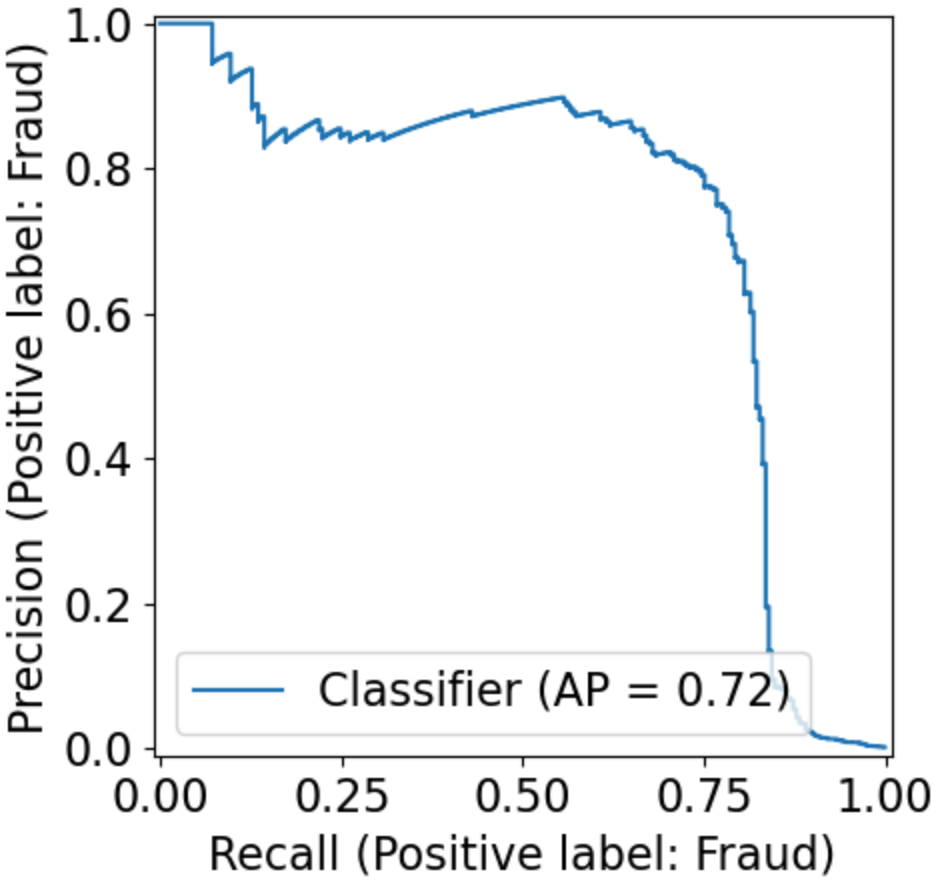
59050 rows × 3 columns

Precision-Recall curve via cross-validation

In our actual pipelines, we will not have a single validation set, but instead use cross-validation. Just as we did with the confusion matrix previously, we can create a PR curve from the `cross_val_predict` function.

```
PrecisionRecallDisplay.from_predictions(
    y_train,
    cross_val_predict(pipe_lr, X_train, y_train, method='predict_proba')[:, 0]
    pos_label='Fraud'
)
```

```
<sklearn.metrics._plot.precision_recall_curve.PrecisionRecallDisplay at 0x14fb
```



With the confusion matrix, we were interested in the hard predictions, which is the default from `cross_val_predict`. With the PR curve, we instead want to visualize the probabilities, so we need to use the `method` parameter to specify that the predictions we get back should be the soft probabilities and not the hard classes.

Also note the `[:, 0]` indexing above, when we use predict_proba, we get two columns back in our array, reflecting the probabilities to be assigned to either class. These two columns are orderd alphabetically by the name of the class. Therefore, the probability of `'Fraud'` is in index `0`, and the probability of `'Non-Fraud'` is in index `1`, but you will have to change this depending on your class names.

AP score

- Often it's useful to have one number summarizing the PR plot (e.g., in hyperparameter optimization)
- One way to do this is by computing the area under the PR curve.
- This is called **average precision** (AP score) and you can see that it is marked in the plots above.
- Average precision computes the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight
- AP score has a value between 0 (worst) and 1 (best).
- A random classifier has an AP score equal to the proportion of the positive class, so this is the baseline to compare against.
- The average_precision_score function supports multiclass and multilabel formats by computing each class score in a One-vs-the-rest (OvR) fashion and averaging

If we want to access the AP score number, we can use the `average_precision_score` function.

```
from sklearn.metrics import average_precision_score

ap_lr = average_precision_score(y_valid, pipe_lr.predict_proba(X_valid)[:, fra
print("Average precision of logistic regression: {:.3f}".format(ap_lr))
```

```
Average precision of logistic regression: 0.757
```

AP vs. F1-score

It is very important to note this distinction:

- F1 score is for a given threshold and measures the quality of `predict`.
- AP score is a summary across thresholds and measures the quality of `predict_proba`.
- Thus, optimizing towards a high F1 score means that you find the model that performs the best at the default decision threshold while considering both recall and precision (and using Fbeta score, you can easily tune if recall or precision should be considered more important). You can still use a PR curve to change the decision threshold after model optimization, but the model has not been optimized to perform well over many thresholds, so there might not be many attractive tradeoffs.
- Optimizing towards a high AP score means that you get the model that performs the best over all possible decision thresholds, which could give you more flexibility in your tradeoffs between precision and recall when deciding on a decision threshold. However, there is no guarantee that this includes the model that performs the best overall (at least theoretically this might have been a model that performed well only for a few thresholds, although that is unlikely).

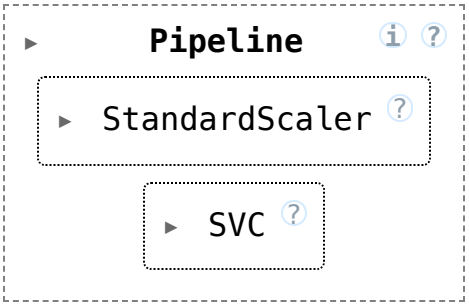
Important

Remember to pick the desired threshold based on the results on the validation set and **not** on the test set.

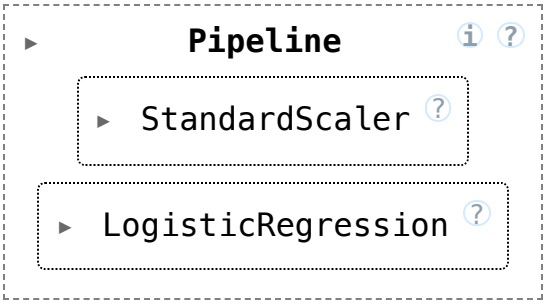
A few comments on PR curve

- Different classifiers might work well in different parts of the curve, i.e., at different operating points.
- We can compare PR curves of different classifiers to understand these differences.
- Let's create PR curves for SVC and Logistic Regression.

```
pipe_svc = make_pipeline(StandardScaler(), SVC())
pipe_svc.fit(X_train, y_train)
```

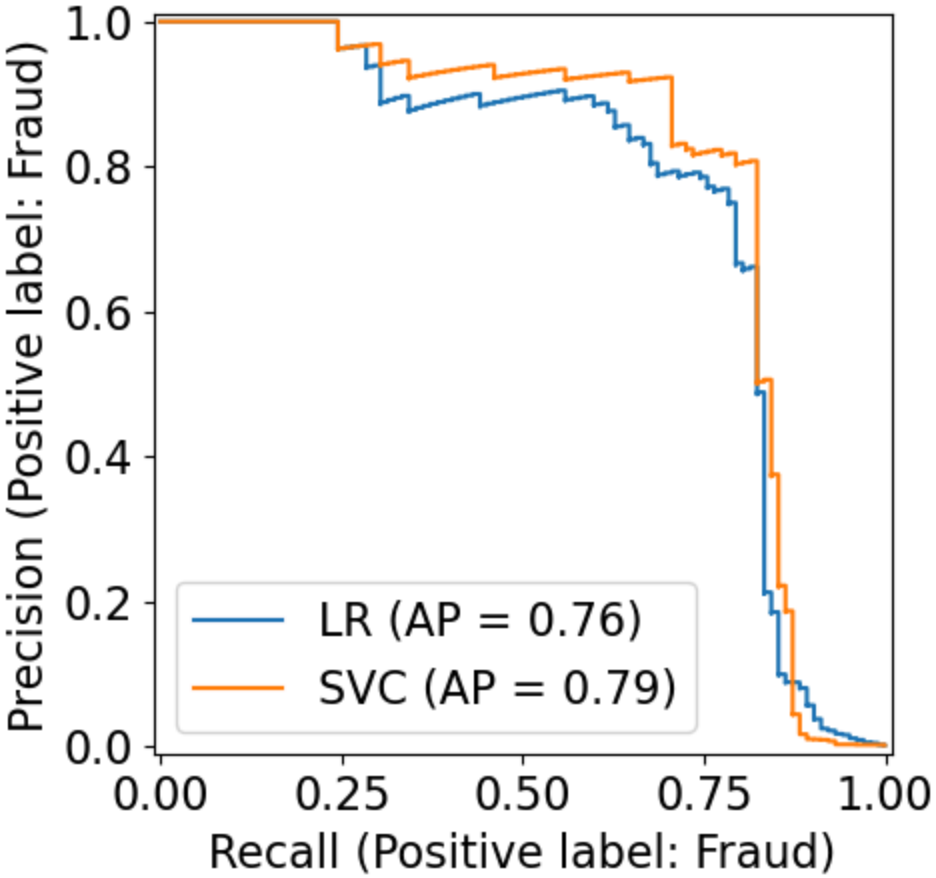


```
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression(max_iter=1000))
pipe_lr.fit(X_train, y_train)
```



How to compare the precision and recall curve for different models?

```
# Set up an empty figure
_, ax = plt.subplots()

# Instruct both curves to plot into the existing figure with the `ax` keyword
PrecisionRecallDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    pos_label='Fraud',
    name='LR',
    ax=ax
);
PrecisionRecallDisplay.from_estimator(
    pipe_svc,
    X_valid,
    y_valid,
    pos_label='Fraud',
    name='SVC',
    ax=ax
);
```



We can see that the SVC classifer performs slighlty better for most thresholds, which means that its AP score is slightly higher, as we can see in the chart.

If we compare to the f1 score (which only looks at a single threshold), we can see that the LR actually has a slightly higher score, so the two metrics don't always agree.

- Comparing the precision-recall curves provide us a detail insight compared to f1 score.
- For example, F1 scores for SVC and logistic regressions are pretty similar. In fact, f1 score of logistic regression is a tiny bit better.
- But when we look at the PR curve, we see that SVC is doing better than logistic regression for most of the other thresholds.

```
svc_preds = pipe_svc.predict(X_valid)
lr_preds = pipe_lr.predict(X_valid)
print("f1_score of logistic regression: {:.3f}".format(f1_score(y_valid, lr_pr
print("f1_score of svc: {:.3f}".format(f1_score(y_valid, svc_preds, pos_label=
```

```
f1_score of logistic regression: 0.728
f1_score of svc: 0.726
```

(Optional) Some more details

- How are the thresholds and the precision and recall at the default threshold are calculated?

How many thresholds?

- It uses `n_thresholds` where `n_thresholds` is the number of unique `predict_proba` scores in our dataset.

```
len(np.unique(pipe_lr.predict_proba(X_valid)[:, fraud_column]))
```

```
59049
```

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

```
precision_lr, recall_lr, thresholds_lr = precision_recall_curve(
    y_valid, pipe_lr.predict_proba(X_valid)[:, fraud_column], pos_label='Fraud
)
precision_svc, recall_svc, thresholds_svc = precision_recall_curve(
    y_valid, pipe_svc.decision_function(X_valid), pos_label='Fraud'
)
```

SVC doesn't have `predict_proba`. Instead it has something called `decision_function`. The index of the threshold that is closest to 0 of decision function is the default threshold in SVC.

For logistic regression, what's the index of the threshold that is closest to the default threshold of 0.5?

- We are subtracting 0.5 from the thresholds so that
  - the numbers close to 0 become -0.5
  - the numbers close to 1 become 0.5
  - the numbers close to 0.5 become 0
- After this transformation, we are interested in the threshold index where the number is close to 0. So we take absolute values and argmin.

```
close_default_lr = np.argmin(np.abs(thresholds_lr - 0.5))
```

```
close_zero_svm = np.argmin(np.abs(thresholds_svc))
```

```
ap_lr = average_precision_score(y_valid, pipe_lr.predict_proba(X_valid)[:, fra
ap_svc = average_precision_score(y_valid, pipe_svc.decision_function(X_valid),
```

```
print("Average precision of logistic regression: {:.3f}".format(ap_lr))
print("Average precision of SVC: {:.3f}".format(ap_svc))
```

```
Average precision of logistic regression: 0.757
Average precision of SVC: 0.001
```

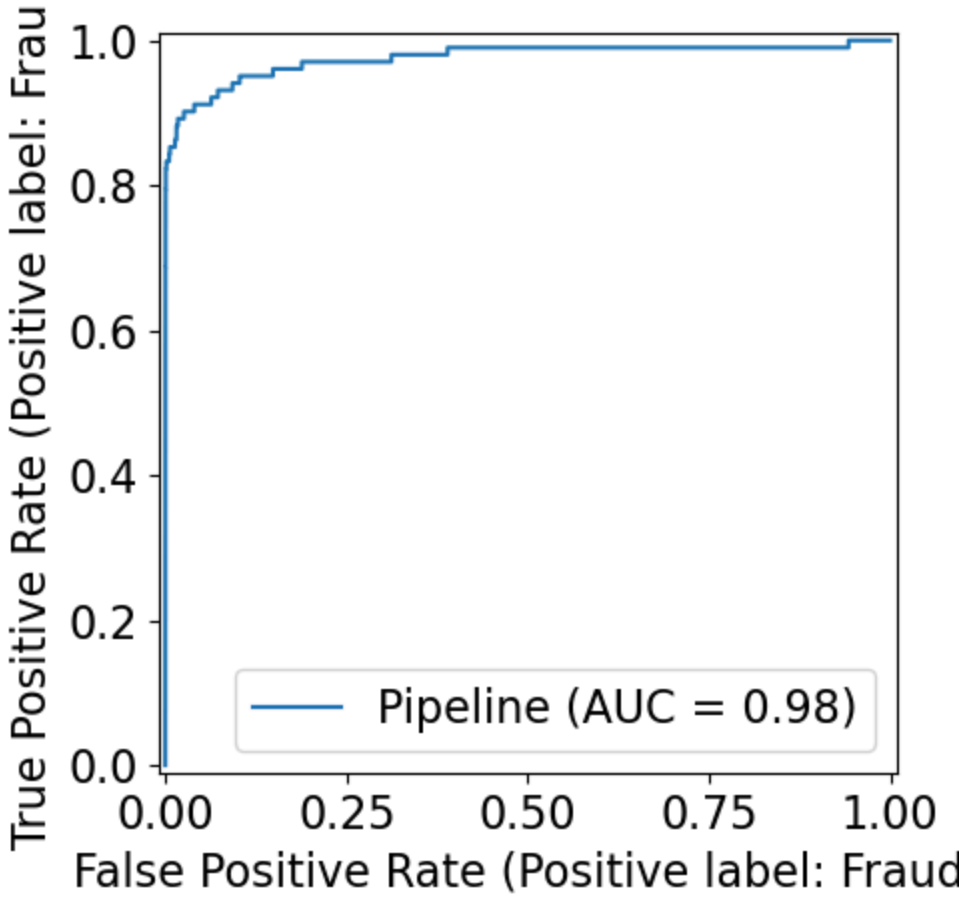
Receiver Operating Characteristic (ROC) curve

- Another commonly used tool to analyze the behavior of classifiers at different thresholds.
- Similar to PR curve, it considers all possible thresholds for a given classifier given by `predict_proba` but instead of precision and recall it plots false positive rate (FPR) and true positive rate (TPR or recall). $$TPR = \frac{TP}{TP+FN}$$

$$FPR = \frac{FP}{FP + TN}$$

- TPR $\longrightarrow$ Fraction of true positives out of all positive examples.
- FPR $\longrightarrow$ Fraction of false positives out of all negative examples.

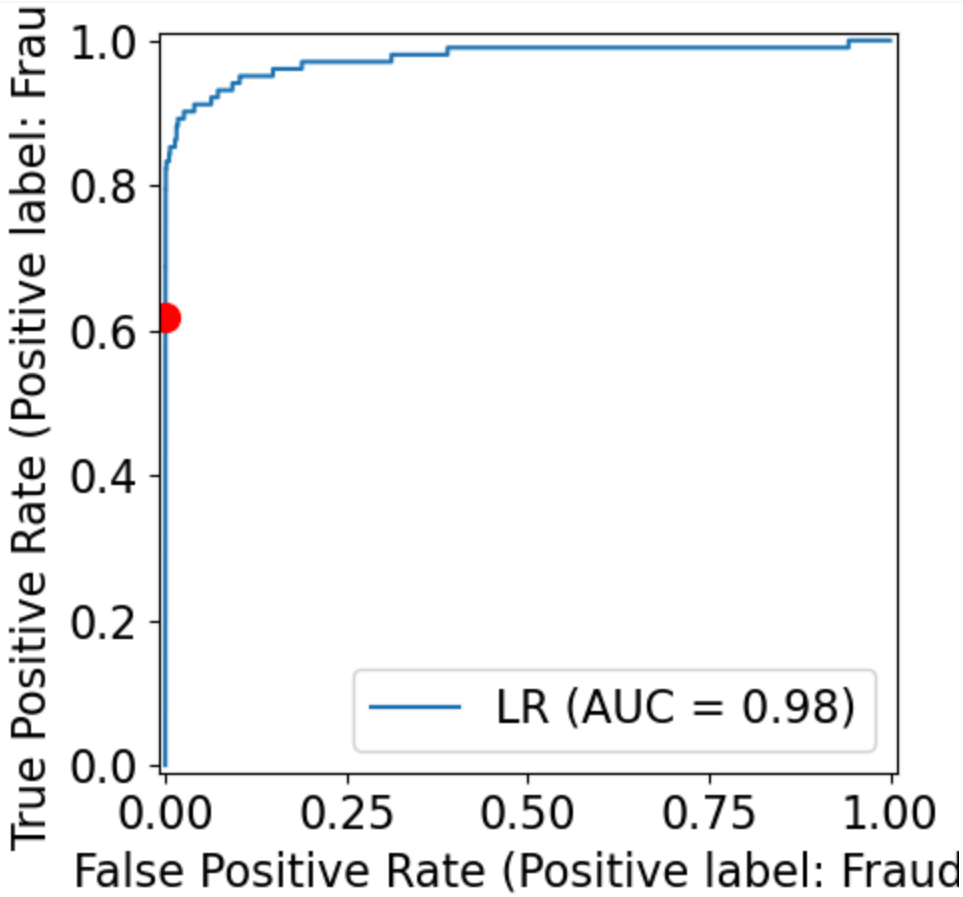
```
from sklearn.metrics import RocCurveDisplay

RocCurveDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    pos_label='Fraud'
);
```



We can add in a point for the default threshold of 0.5, and see that it is not optimal. We can increase or TPR, while keeping the FPR the same and also increate the FPR marginally over zero while still gaining some TPR. A good value depends on what is most important bettween TPR and FPR, but without any knowledge of that, we would try to get as close to the top left corner as possible, so a TPR around 0.9 and FPR around 0.05.

```
RocCurveDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    pos_label='Fraud',
    name='LR'
);

from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_valid, pipe_lr.predict_proba(X_valid)[:, fr
default_threshold = np.argmin(np.abs(thresholds - 0.5))

plt.plot(
    fpr[default_threshold],
    tpr[default_threshold],
    "or",
    markersize=10,
    label="threshold 0.5",
)
```

```
[<matplotlib.lines.Line2D at 0x31cfa6c30>]
```



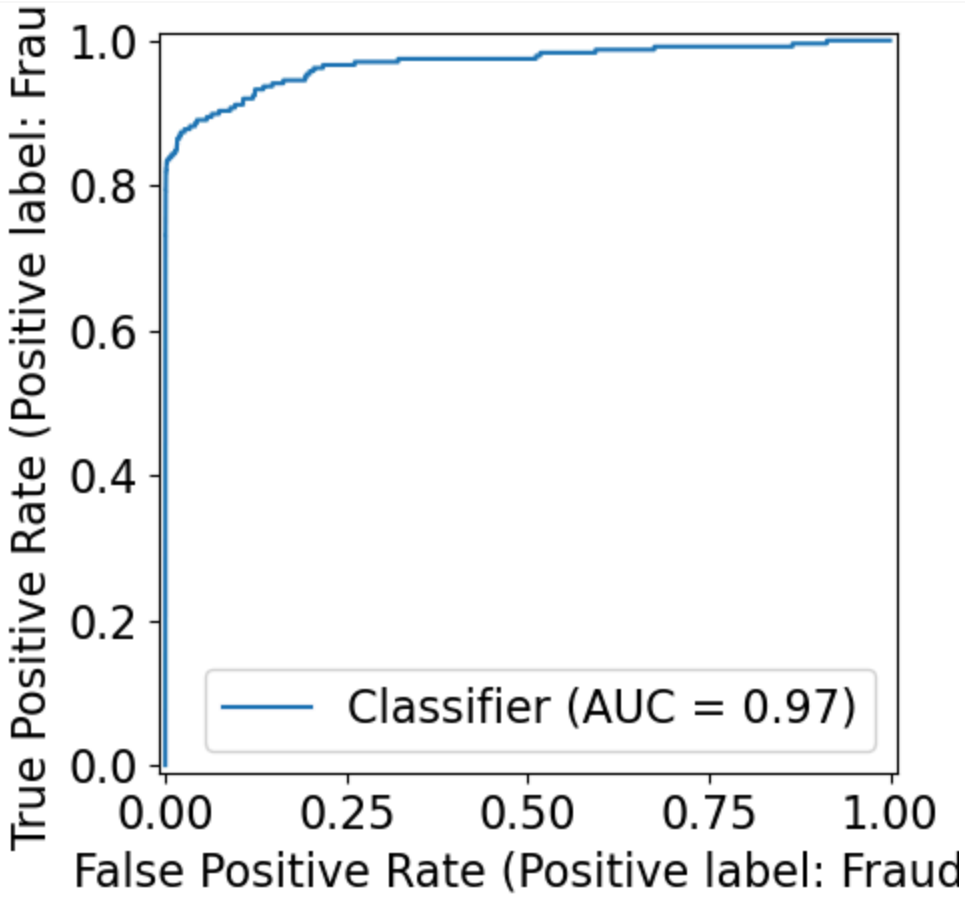
- The ideal curve is close to the top left
    - Ideally, you want a classifier with high recall while keeping low false positive rate.

- The red dot corresponds to the threshold of 0.5, which is used by predict by default.
- We see that compared to the default threshold, we can achieve a better recall of around 0.8 without increasing FPR.
- We could also compare different models on the ROC curve, just as we did for the PR curve above.

Just as for the confusion matrix and PR curve, we normally create the ROC curve from the predictions during cross-validation.

```
RocCurveDisplay.from_predictions(
    y_train,
    cross_val_predict(pipe_lr, X_train, y_train, method='predict_proba')[:, 0]
    pos_label='Fraud'
)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x31cf4c440>
```



> (Optional) Note that concatenating the cross-validation results and then creating a single ROC curve can in some instances lead to an overly conservative performance

evaluation. This is due to that the individual models have not had their probabilities calibrated with each other so the ranking that the ROC performs might be thrown off when the probabilities are brought togeher in a single ranking. Instead, it can be beneficial to create one curve per CV fold and an average as in this example, which also allows us to report on the spread.

Area under the curve (AUC)

- AUC provides a single number summarizing the ROC curve.

```
from sklearn.metrics import roc_auc_score

roc_lr = roc_auc_score(y_valid, pipe_lr.predict_proba(X_valid)[:, 1])
roc_svc = roc_auc_score(y_valid, pipe_svc.decision_function(X_valid))
print("AUC for LR: {:.3f}".format(roc_lr))
print("AUC for SVC: {:.3f}".format(roc_svc))
```

```
AUC for LR: 0.976
AUC for SVC: 0.938
```

- AUC of 0.5 means random chance, which is the baseline to compare against.
- AUC can be interpreted as evaluating the **ranking** of positive examples.
- What's the probability that a randomly picked positive point has a higher score according to the classifier than a randomly picked point from the negative class.
- AUC of 1.0 means all positive points have a higher score than all negative points.

Important

For classification problems with imbalanced classes, using AP score or AUC is often much more meaningful than using accuracy.

See also

Check out these visualization on ROC and AUC in more detail.

See also

Check out how to plot ROC with cross-validation here.

Reporting performance from the test data set

After selecting the best hyperparameters and decision threshold, for our model during cross-validation, it is time to assess how well our model fairs on unseen data. Although we used a specific metric when we tuned our model, it is helpful to report multiple metrics when communicating the test data performance. It is useful to focus on metrics that are important for the problem at hand, but also has a meaningingful interpretation, such as precision and recall. This process could looks something like this:

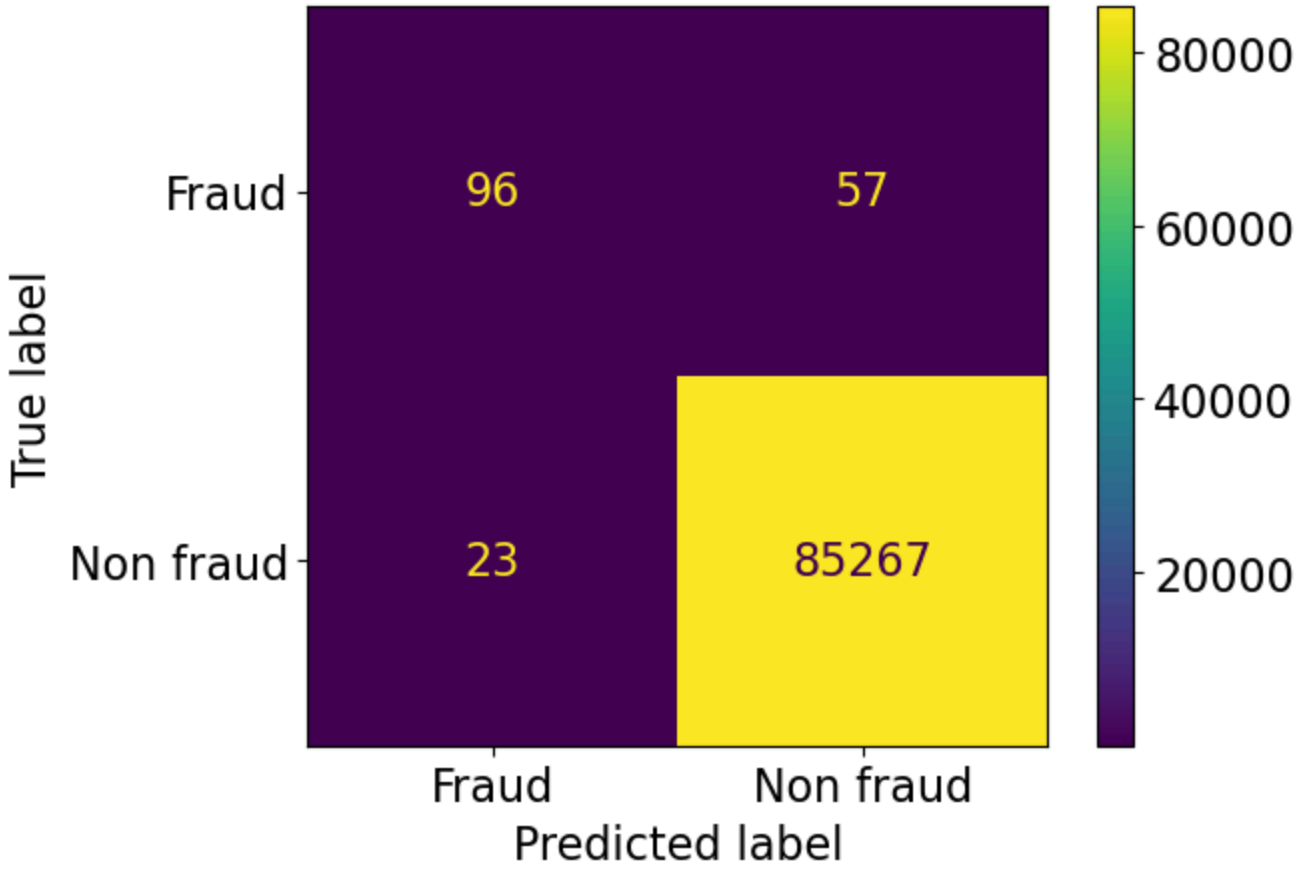
```
pd.DataFrame({
    'F1': [f1_score(y_test, pipe_lr.predict(X_test), pos_label='Fraud')],
    'Precision': [precision_score(y_test, pipe_lr.predict(X_test), pos_label='
    'Recall': [recall_score(y_test, pipe_lr.predict(X_test), pos_label='Fraud'
})
```

	F1	Precision	Recall
0	0.705882	0.806723	0.627451

It's often effective to communicate with a confusion matrix as well, especially to less technical audiences, but even in more technical context, it's helpful to see the actual numbers (or proportions) of TP, FP, TN, FN.

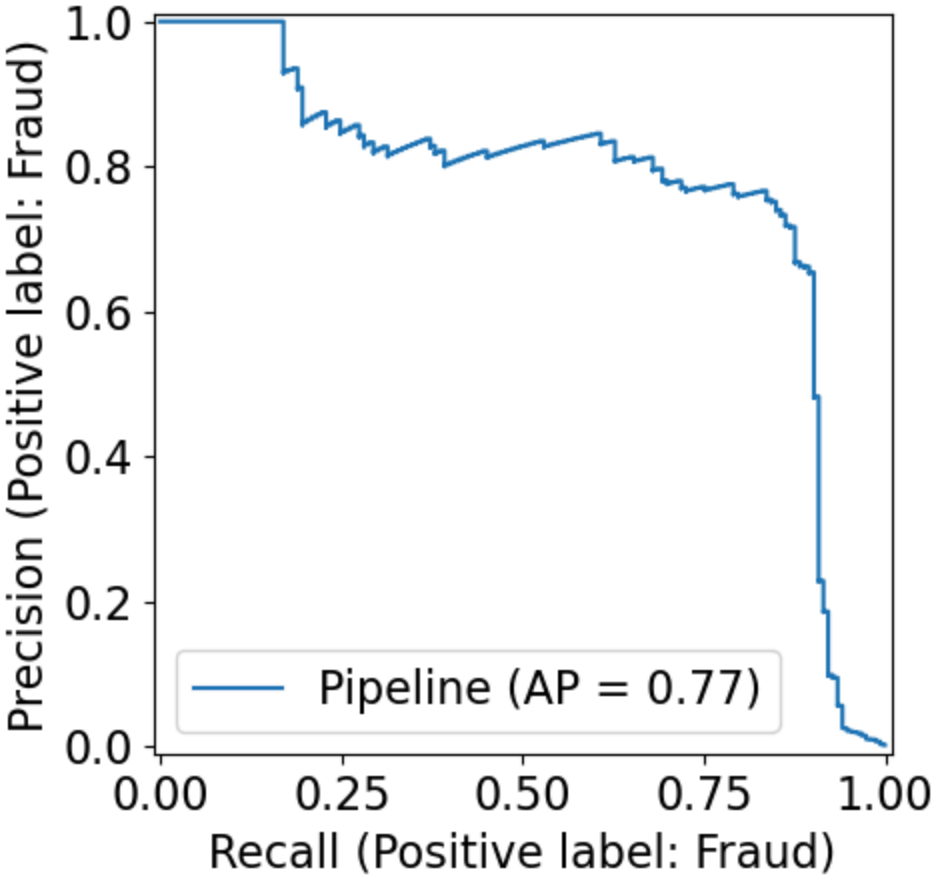
```
ConfusionMatrixDisplay.from_estimator(
    pipe_lr,
    X_test,
    y_test
)
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x31cf6cbc0>
```



And finally, seeing a PR or ROC curve helps to understand the classifiers performance over multiple thresholds.

```
PrecisionRecallDisplay.from_estimator(
    pipe_lr,
    X_test,
    y_test,
    pos_label='Fraud',
);
```



Note

Note that we are not allowed to make any modifications to our model, based on the scores or visualizations of the test data! That would break the golden rule and lead to an overly optimistic model since we are using info from the the test data to improve the model. Any information in the test data must be completely ignored for it to be an valid approximation of the model's generalization performance on unseen data.

Dealing with class imbalance [video]

Class imbalance in training sets

- This typically refers to having many more examples of one class than another in one's training set.
- Real world data is often imbalanced.
  - Our Credit Card Fraud dataset is imbalanced.
  - Ad clicking data is usually drastically imbalanced. (Only around ~0.01% ads are clicked.)
  - Spam classification datasets are also usually imbalanced.

Addressing class imbalance

A very important question to ask yourself: "Why do I have a class imbalance?"

- Is it because one class is much more rare than the other?
  - If it's just because one is more rare than the other, you need to ask whether you care about one type of error more than the other.
- Is it because of my data collection methods?
  - If it's the data collection, then that means *your test and training data come from different distributions*!

In some cases, it may be fine to just ignore the class imbalance.

Which type of error is more important?

- False positives (FPs) and false negatives (FNs) have quite different real-world consequences.
- In PR curve and ROC curve, we saw how changing the prediction threshold can change FPs and FNs.
- We can then pick the threshold that's appropriate for our problem.
- Example: if we want high recall, we may use a lower threshold (e.g., a threshold of 0.1). We'll then catch more fraudulent transactions. Let's first see the report with the standard

0.5 threshold.

```
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression())
pipe_lr.fit(X_train, y_train)
y_pred = pipe_lr.predict(X_valid)
print(classification_report(y_valid, y_pred))
```

```
              precision    recall  f1-score   support

       Fraud       0.89      0.62      0.73       102
   Non fraud       1.00      1.00      1.00     59708

    accuracy                           1.00     59810
   macro avg       0.94      0.81      0.86     59810
weighted avg       1.00      1.00      1.00     59810
```

Then we change the threshold to 0.1, which increases the recall and decreases the precision.

```
y_pred = np.array(['Non fraud'] * X_valid.shape[0])  # Base array with all "No
y_pred[pipe_lr.predict_proba(X_valid)[:, fraud_column] > 0.1] = 'Fraud'  # Ove
print(classification_report(y_valid, y_pred))
```

```
              precision    recall  f1-score   support

       Fraud       0.77      0.75      0.76       102
   Non fraud       1.00      1.00      1.00     59708

    accuracy                           1.00     59810
   macro avg       0.88      0.88      0.88     59810
weighted avg       1.00      1.00      1.00     59810
```

Handling imbalance

Can we change the model itself rather than changing the threshold so that it takes into account the errors that are important to us?

There are two common approaches for this:

- **Changing the data (optional)** (not covered in this course)
  - Undersampling
  - Oversampling

    - Random oversampling
    - SMOTE
- **Changing the training procedure**
    - `class_weight`

Changing the training procedure

- All `sklearn` classifiers have a parameter called `class_weight`.
- This allows you to specify that one class is more important than another.
- For example, maybe a false negative is 10x more problematic than a false positive.

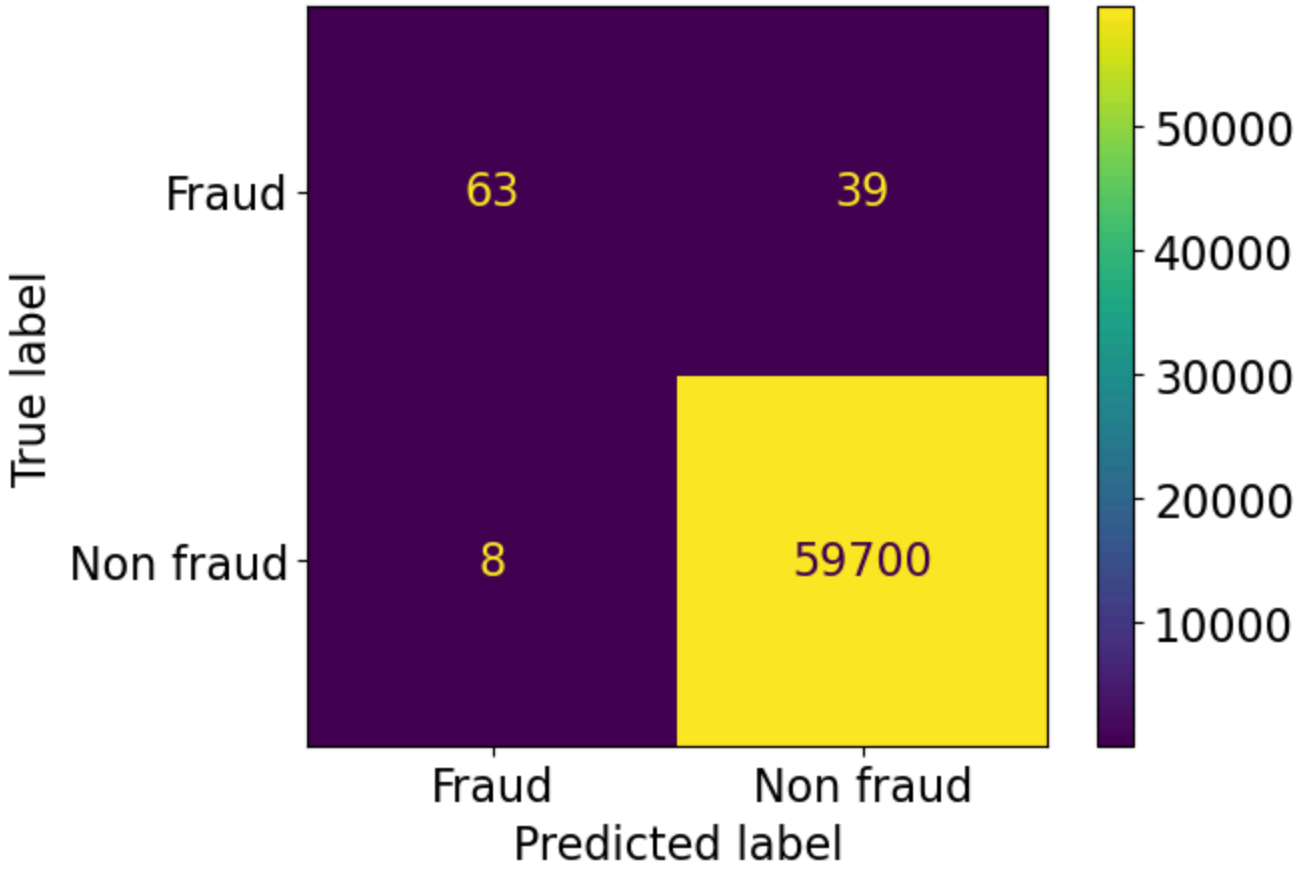
Example: `class_weight` parameter of `sklearn LogisticRegression`

> class sklearn.linear_model.LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, **class_weight=None**, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)

> class_weight: dict or 'balanced', default=None

> Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one.

```
ConfusionMatrixDisplay.from_estimator(
    pipe_lr, X_valid, y_valid, values_format="d"
);
```

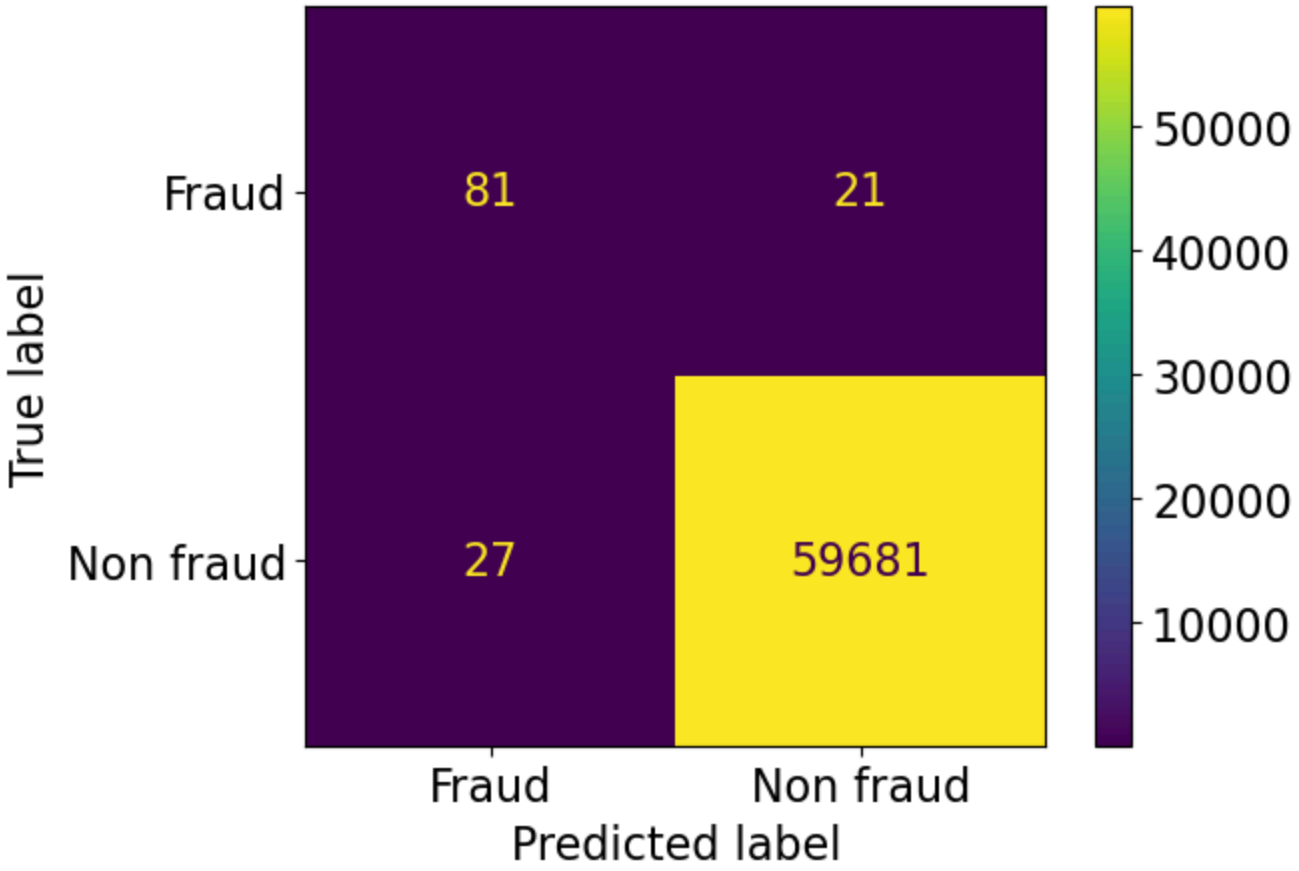



```
pipe_lr.named_steps["logisticregression"].classes_
```

```
array(['Fraud', 'Non fraud'], dtype=object)
```

Let's set "fraud" class a weight of 10 to see how that leads to more of the Fraudulent examples being predicted correclty.

```
pipe_lr_weight = make_pipeline(
    StandardScaler(), LogisticRegression(max_iter=500, class_weight={'Non frau
)
pipe_lr_weight.fit(X_train, y_train)
ConfusionMatrixDisplay.from_estimator(
    pipe_lr_weight,
    X_valid,
    y_valid,
    values_format="d",
);
```



- Notice we've reduced false negatives and predicted more Fraud this time.
- This was equivalent to saying give 10x more "importance" to fraud class.
- Note that as a consequence we are also increasing false positives.

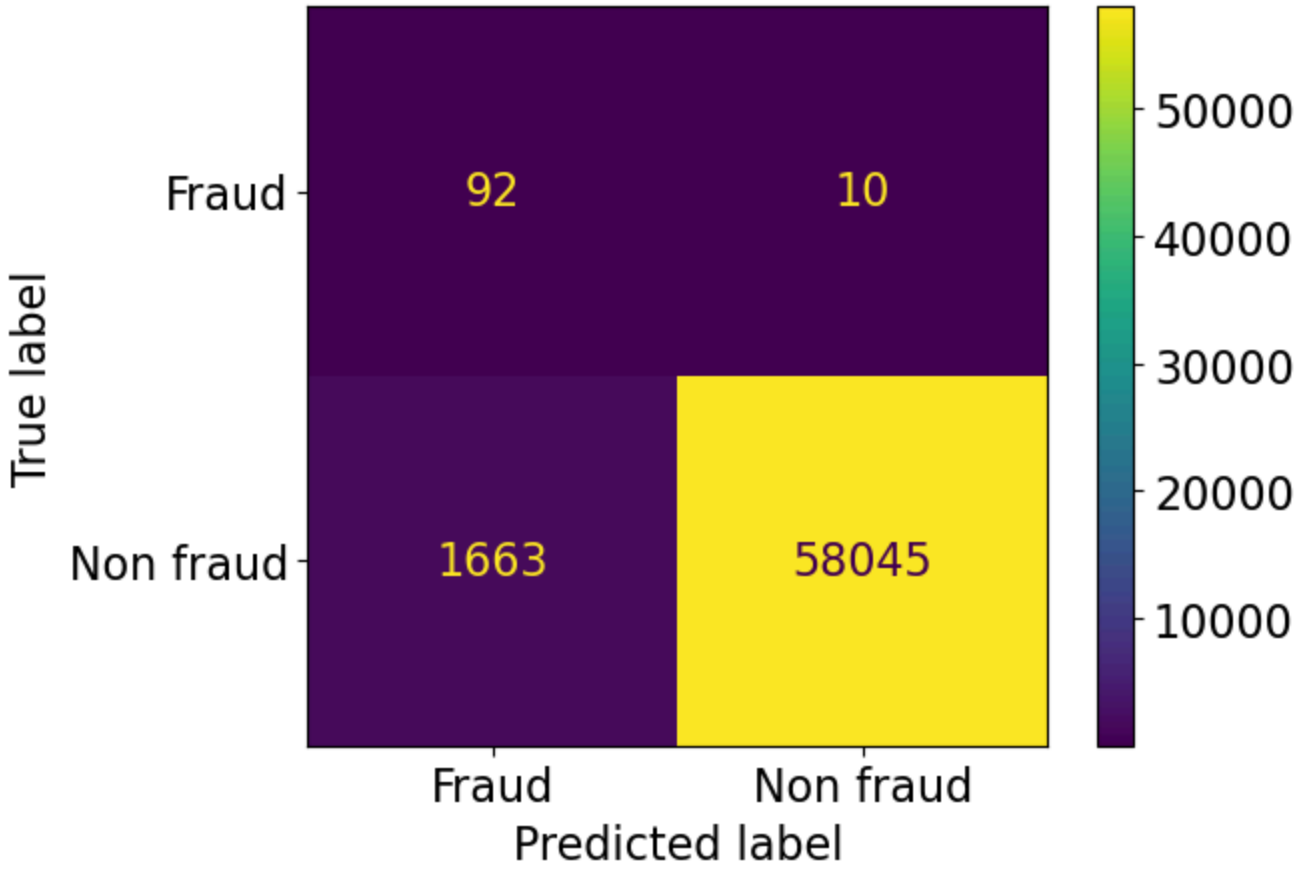
`class_weight="balanced"`

- A useful setting is `class_weight="balanced"`.
- This sets the weights so that the classes are "equal".

> class_weight: dict, 'balanced' or None If 'balanced', class weights will be given by n_samples / (n_classes * np.bincount(y)). If a dictionary is given, keys are classes and values are corresponding class weights. If None is given, the class weights will be uniform.

> sklearn.utils.class_weight.compute_class_weight(class_weight, classes, y)

```
pipe_lr_balanced = make_pipeline(
    StandardScaler(), LogisticRegression(max_iter=500, class_weight="balanced"
)
pipe_lr_balanced.fit(X_train, y_train)
ConfusionMatrixDisplay.from_estimator(
    pipe_lr_balanced,
    X_valid,
    y_valid,
    values_format="d",
);
```



We have reduced false negatives but we have many more false positives now ...

Are we doing better with `class_weight="balanced"`?

```
```

```
from sklearn.metrics import roc_auc_score, average_precision_score

scoring = {
    "accuracy": 'accuracy',
    # We need to use `make_scorer` in order to set parameters such as pos_labe
    'f1': make_scorer(f1_score, pos_label='Fraud'),
    'recall': make_scorer(recall_score, pos_label='Fraud'),
    'precision': make_scorer(precision_score, pos_label='Fraud'),
    # We need to set the response method to 'predict_proba' when not using a 0
    'average_precision': make_scorer(average_precision_score, pos_label='Fraud
    'roc_auc': make_scorer(roc_auc_score, response_method='predict_proba')
}

pipe_lr = make_pipeline(StandardScaler(), LogisticRegression(max_iter=500))
scores = cross_validate(
    pipe_lr, X_train_big, y_train_big, scoring=scoring
)
pd.DataFrame(scores)
```

	fit_time	score_time	test_accuracy	test_f1	test_recall	test_precision	t
0	0.255253	0.559520	0.999122	0.700855	0.602941	0.836735	
1	0.234963	0.552853	0.999223	0.735043	0.632353	0.877551	
2	0.233854	0.564115	0.999298	0.754386	0.632353	0.934783	
3	0.243386	0.558791	0.999172	0.697248	0.558824	0.926829	
4	0.229784	0.555100	0.999147	0.696429	0.582090	0.866667	

- Recall is much better but precision has dropped a lot; we have many false positives.
- You could also optimize `class_weight` using hyperparameter optimization for your specific problem.

- Changing the class weight will **generally reduce accuracy**.
- The original model was trying to maximize accuracy.
- Now you're telling it to do something different.
- But that can be fine, accuracy isn't the only metric that matters.

Stratified Splits

- A similar idea of "balancing" classes can be applied to data splits.
- We have the same option in `train_test_split` with the `stratify` argument.

- By default it splits the data so that if we have 10% negative examples in total, then each split will have 10% negative examples.
- If you are carrying out cross validation using `cross_validate`, by default it uses `StratifiedKFold`. From the documentation:

> This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

- In other words, if we have 10% negative examples in total, then each fold will have 10% negative examples.

Is stratifying a good idea?

- Well, it's no longer a random sample, which is probably theoretically bad, but not that big of a deal.
- If you have many examples, it shouldn't matter as much.
- It can be especially useful in multi-class, say if you have one class with very few cases.
- In general, these are difficult questions.

(Optional) Changing the data

- Undersampling
- Oversampling
  - Random oversampling
  - SMOTE

We cannot use sklearn pipelines because of some API related problems. But there is something called `imbalance learn`, which is an extension of the `scikit-learn` API that

allows us to resample. It's already in our course environment. If you don't have the course environment installed, you can install it in your environment with this command:

```
conda install -c conda-forge imbalanced-learn
```

Undersampling

```
import imblearn
from imblearn.pipeline import make_pipeline as make_imb_pipeline
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler()
X_train_subsample, y_train_subsample = rus.fit_resample(X_train, y_train)
print(X_train.shape)
print(X_train_subsample.shape)
y_train_subsample.value_counts()
```

```
(139554, 29)
(474, 29)
```

```
Class
Fraud        237
Non fraud    237
Name: count, dtype: int64
```

```
from collections import Counter

from imblearn.under_sampling import RandomUnderSampler
from sklearn.datasets import make_classification

X, y = make_classification(
    n_classes=2,
    class_sep=2,
    weights=[0.1, 0.9],
    n_informative=3,
    n_redundant=1,
    flip_y=0,
    n_features=20,
    n_clusters_per_class=1,
    n_samples=1000,
    random_state=10,
)
print("Original dataset shape %s" % Counter(y))
rus = RandomUnderSampler(random_state=42)
X_res, y_res = rus.fit_resample(X, y)
print("Resampled dataset shape %s" % Counter(y_res))
```

```
Original dataset shape Counter({1: 900, 0: 100})
Resampled dataset shape Counter({0: 100, 1: 100})
```

```
undersample_pipe = make_imb_pipeline(
    RandomUnderSampler(), StandardScaler(), LogisticRegression()
)

scoring = {
    'average_precision': make_scorer(average_precision_score, pos_label='Fraud
    'roc_auc': make_scorer(roc_auc_score, response_method='predict_proba')
}
scores = cross_validate(
    undersample_pipe, X_train, y_train, scoring=scoring
)
pd.DataFrame(scores).mean()
```

```
fit_time                  0.137661
score_time                0.045798
test_average_precision    0.482441
test_roc_auc              0.028933
dtype: float64
```

Oversampling

- Random oversampling with replacement
- SMOTE: Synthetic Minority Over-sampling Technique

```
from imblearn.over_sampling import RandomOverSampler

ros = RandomOverSampler()
X_train_oversample, y_train_oversample = ros.fit_resample(X_train, y_train)
print(X_train.shape)
print(X_train_oversample.shape)
y_train_oversample.value_counts()
```

```
(139554, 29)
(278634, 29)
```

```
Class
Non fraud    139317
Fraud        139317
Name: count, dtype: int64
```

```
oversample_pipe = make_imb_pipeline(
    RandomOverSampler(), StandardScaler(), LogisticRegression(max_iter=1000)
)
scores = cross_validate(
    oversample_pipe, X_train, y_train, scoring=scoring
)
pd.DataFrame(scores).mean()
```

```
fit_time                 0.972984
score_time               0.046790
test_average_precision   0.712874
test_roc_auc             0.038268
dtype: float64
```

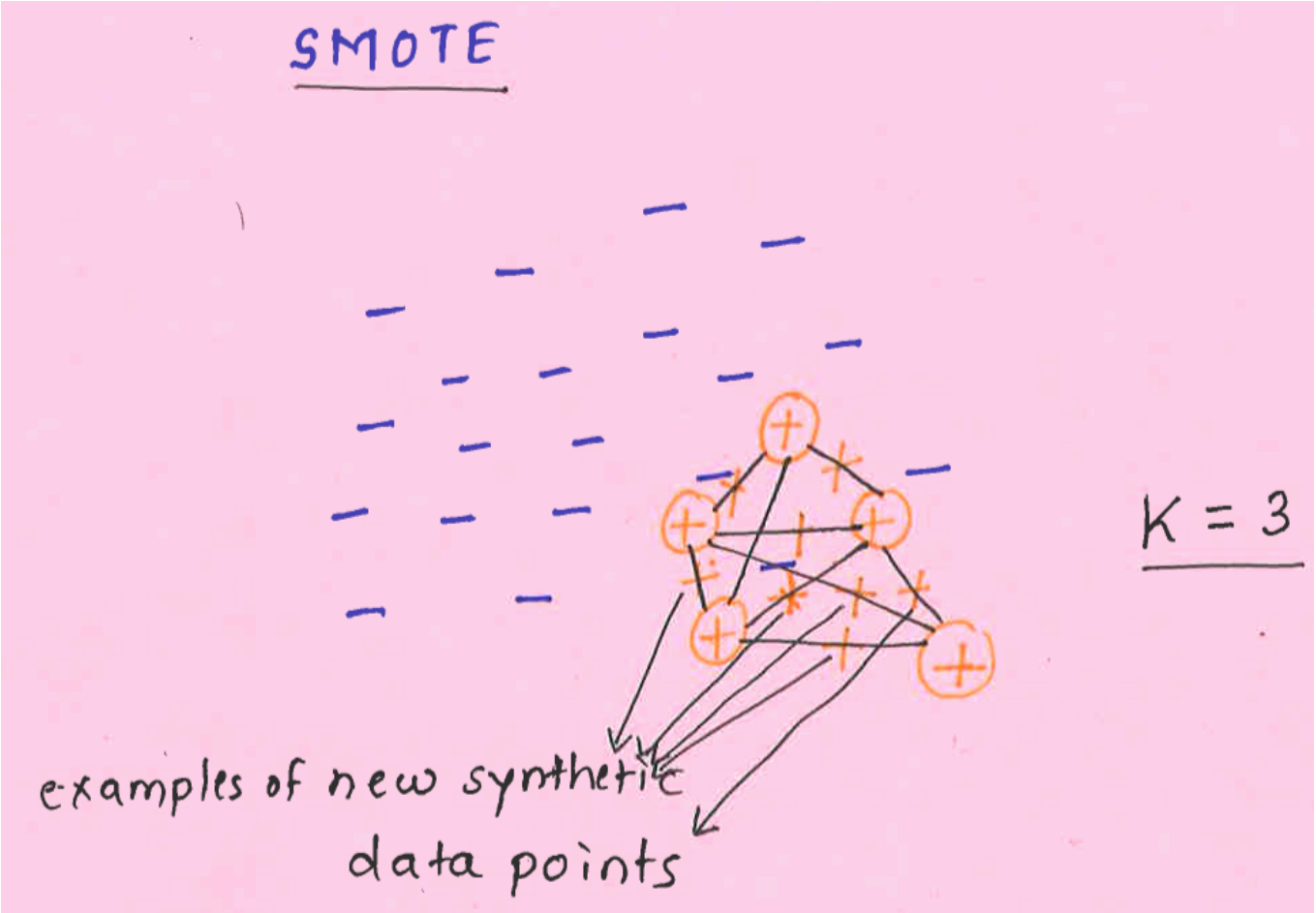
SMOTE: Synthetic Minority Over-sampling Technique

sklearn SMOTE

- Create “synthetic” examples rather than by over-sampling with replacement.
- Inspired by a technique of data augmentation that proved successful in handwritten character recognition.
- The minority class is over-sampled by taking each minority class sample and introducing synthetic examples along the line segments joining any/all of the k minority class nearest neighbors.
- k is chosen depending upon the amount of over-sampling required.

SMOTE idea

- Take the difference between the feature vector (sample) under consideration and its nearest neighbor.
- Multiply this difference by a random number between 0 and 1, and add it to the feature vector under consideration.
- This causes the selection of a random point along the line segment between two specific features.
- This approach effectively forces the decision region of the minority class to become more general.



Using SMOTE

- You need to `imbalanced-learn`

> class imblearn.over_sampling.SMOTE(sampling_strategy='auto', random_state=None, k_neighbors=5, m_neighbors='deprecated', out_step='deprecated', kind='deprecated', svm_estimator='deprecated', n_jobs=1, ratio=None)

> Class to perform over-sampling using SMOTE.

> This object is an implementation of SMOTE - Synthetic Minority Over-sampling Technique as presented in this paper.

```
from imblearn.over_sampling import SMOTE

smote_pipe = make_imb_pipeline(
    SMOTE(), StandardScaler(), LogisticRegression(max_iter=1000)
)
scoring = {
    'average_precision': make_scorer(average_precision_score, pos_label='Fraud
    'roc_auc': make_scorer(roc_auc_score, response_method='predict_proba')
}
scores = cross_validate(
    smote_pipe, X_train, y_train, cv=10, scoring=scoring
)
pd.DataFrame(scores).mean()
```

```
fit_time                 1.203977
score_time               0.026035
test_average_precision   0.732590
test_roc_auc             0.037043
dtype: float64
```

- We got a slightly higher average precision score with SMOTE in this case.
- The above are rather simple approaches to tackle class imbalance.
- If you have a problem such as fraud detection problem where you want to spot rare events, you can also think of this problem as anomaly detection problem and use different kinds of algorithms such as isolation forests.
- If you are interested in this area, it might be worth checking out this book on this topic. (I've not read it.)
    - Imbalanced Learning: Foundations, Algorithms, and Applications (available via UBC's library).

What did we learn today?

- A number of possible ways to evaluate machine learning models

    - Choose the evaluation metric that makes most sense in your context or which is most common in your discipline
- Two kinds of binary classification problems
    - Distinguishing between two classes (e.g., dogs vs. cats)
    - Spotting a class (e.g., spot fraud transaction, spot spam)
- Precision, recall, f1-score are useful when dealing with spotting problems.
- The thing that we are interested in spotting is considered "positive".
- Do you need to deal with class imbalance in the given problem?
- Methods to deal with class imbalance
    - Changing the training procedure
        - `class_weight`
    - Changing the data
        - undersampling, oversampling, SMOTE
- Do not blindly make decisions solely based on ML model predictions.
- Try to carefully analyze the errors made by the model on certain groups.

Relevant papers and resources

- The Relationship Between Precision-Recall and ROC Curves
- Article claiming that PR curve are better than ROC for imbalanced datasets
- Precision-Recall-Gain Curves: PR Analysis Done Right
- ROC animation
- Generalization in Adaptive Data Analysis and Holdout Reuse