09_classification-metrics

June 23, 2022

CPSC 330
Applied Machine Learning

1 Lecture 9: Classification Metrics

UBC 2022 Summer

Instructor: Mehrdad Oveisi

1.1 Imports

[1]:
```
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_test_split
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler
from utils import *

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

```python
from IPython.display import Image
```

[2]:
```python
# Changing global matplotlib settings for confusion matrix.
plt.rcParams["xtick.labelsize"] = 18
plt.rcParams["ytick.labelsize"] = 18
```

1.2 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.

1.3 Evaluation metrics for binary classification: Motivation

1.3.1 Dataset for demonstration

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

[3]:
```python
cc_df = pd.read_csv("data/creditcard.csv", encoding="latin-1")
train_df, test_df = train_test_split(cc_df, test_size=0.3, random_state=111)
train_df.head()
```

[3]:
```
            Time        V1        V2        V3        V4        V5        V6  \
64454    51150.0 -3.538816  3.481893 -1.827130 -0.573050  2.644106 -0.340988
37906    39163.0 -0.363913  0.853399  1.648195  1.118934  0.100882  0.423852
79378    57994.0  1.193021 -0.136714  0.622612  0.780864 -0.823511 -0.706444
245686  152859.0  1.604032 -0.808208 -1.594982  0.200475  0.502985  0.832370
60943    49575.0 -2.669614 -2.734385  0.662450 -0.059077  3.346850 -2.549682

              V7        V8        V9  …       V21       V22       V23  \
64454   2.102135 -2.939006  2.578654  …  0.530978 -0.860677 -0.201810
37906   0.472790 -0.972440  0.033833  …  0.687055 -0.094586  0.121531
79378  -0.206073 -0.016918  0.781531  … -0.310405 -0.842028  0.085477
245686 -0.034071  0.234040  0.550616  …  0.519029  1.429217 -0.139322
60943  -1.430571 -0.118450  0.469383  … -0.228329 -0.370643 -0.211544

             V24       V25       V26       V27       V28  Amount  Class
64454  -1.719747  0.729143 -0.547993 -0.023636 -0.454966    1.00      0
37906   0.146830 -0.944092 -0.558564 -0.186814 -0.257103   18.49      0
```

```
79378    0.366005  0.254443  0.290002 -0.036764  0.015039   23.74      0
245686 -1.293663  0.037785  0.061206  0.005387 -0.057296  156.52      0
60943   -0.300837 -1.174590  0.573818  0.388023  0.161782   57.50      0

[5 rows x 31 columns]
```

[4]:
```
train_df.shape
```

[4]:
```
(199364, 31)
```

- Good size dataset
- For confidentially reasons, it only provides transformed features with PCA, which is a popular dimensionality reduction technique.

1.3.2 Exploratory Data Analysis (EDA)

[5]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 199364 entries, 64454 to 129900
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    199364 non-null  float64
 1   V1      199364 non-null  float64
 2   V2      199364 non-null  float64
 3   V3      199364 non-null  float64
 4   V4      199364 non-null  float64
 5   V5      199364 non-null  float64
 6   V6      199364 non-null  float64
 7   V7      199364 non-null  float64
 8   V8      199364 non-null  float64
 9   V9      199364 non-null  float64
 10  V10     199364 non-null  float64
 11  V11     199364 non-null  float64
 12  V12     199364 non-null  float64
 13  V13     199364 non-null  float64
 14  V14     199364 non-null  float64
 15  V15     199364 non-null  float64
 16  V16     199364 non-null  float64
 17  V17     199364 non-null  float64
 18  V18     199364 non-null  float64
 19  V19     199364 non-null  float64
 20  V20     199364 non-null  float64
 21  V21     199364 non-null  float64
 22  V22     199364 non-null  float64
 23  V23     199364 non-null  float64
 24  V24     199364 non-null  float64
```

```
 25  V25     199364 non-null  float64
 26  V26     199364 non-null  float64
 27  V27     199364 non-null  float64
 28  V28     199364 non-null  float64
 29  Amount  199364 non-null  float64
 30  Class   199364 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 48.7 MB
```

[6]:
```
train_df.describe(include="all")
```

[6]:
```
                Time             V1             V2             V3  \
count  199364.000000  199364.000000  199364.000000  199364.000000
mean    94888.815669       0.000492      -0.000726       0.000927
std     47491.435489       1.959870       1.645519       1.505335
min         0.000000     -56.407510     -72.715728     -31.813586
25%     54240.000000      -0.918124      -0.600193      -0.892476
50%     84772.500000       0.018854       0.065463       0.179080
75%    139349.250000       1.315630       0.803617       1.028023
max    172792.000000       2.451888      22.057729       9.382558

                  V4             V5             V6             V7  \
count  199364.000000  199364.000000  199364.000000  199364.000000
mean        0.000630       0.000036       0.000011      -0.001286
std         1.413958       1.361718       1.327188       1.210001
min        -5.683171     -42.147898     -26.160506     -43.557242
25%        -0.847178      -0.691241      -0.768512      -0.553979
50%        -0.019531      -0.056703      -0.275290       0.040497
75%         0.744201       0.610407       0.399827       0.570449
max        16.491217      34.801666      23.917837      44.054461

                  V8             V9  …            V21            V22  \
count  199364.000000  199364.000000  …  199364.000000  199364.000000
mean       -0.002889      -0.000891  …       0.001205       0.000155
std         1.214852       1.096927  …       0.748510       0.726634
min       -73.216718     -13.320155  …     -34.830382      -8.887017
25%        -0.209746      -0.642965  …      -0.227836      -0.541795
50%         0.022039      -0.052607  …      -0.029146       0.007666
75%         0.327408       0.597326  …       0.186899       0.529210
max        19.587773      15.594995  …      27.202839      10.503090

                 V23            V24            V25            V26  \
count  199364.000000  199364.000000  199364.000000  199364.000000
mean       -0.000198       0.000113       0.000235       0.000312
std         0.628139       0.605060       0.520857       0.481960
min       -44.807735      -2.824849     -10.295397      -2.241620
25%        -0.162330      -0.354604      -0.317761      -0.326730
```

```
50%        -0.011678       0.041031       0.016587      -0.052790
75%         0.146809       0.439209       0.351366       0.242169
max        22.083545       4.022866       6.070850       3.517346

                 V27            V28         Amount          Class
count  199364.000000  199364.000000  199364.000000  199364.000000
mean       -0.000366       0.000227      88.164679       0.001700
std         0.401541       0.333139     238.925768       0.041201
min       -22.565679     -11.710896       0.000000       0.000000
25%        -0.070929      -0.052819       5.640000       0.000000
50%         0.001239       0.011234      22.000000       0.000000
75%         0.090453       0.078052      77.150000       0.000000
max        12.152401      33.847808   11898.090000       1.000000

[8 rows x 31 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`.
- Do we want to scale time?
  - In this lecture we'll do it's probably not the best thing to do.
  - We'll learn about time series briefly later in the course.

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

[7]:
```
X_train_big, y_train_big = train_df.drop(columns=["Class"]), train_df["Class"]
X_test, y_test = test_df.drop(columns=["Class"]), 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 validation set.
- Our data is large enough so it shouldn't be a problem.

[8]:
```
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_big, y_train_big, test_size=0.3, random_state=123
)
```

1.3.3 Baseline

[9]:
```
dummy = DummyClassifier()
pd.DataFrame(cross_validate(dummy, X_train, y_train, return_train_score=True)).
 ↪mean()
```

[9]:
```
fit_time       0.026793
score_time     0.003569
test_score     0.998302
train_score    0.998302
dtype: float64
```

1.3.4 Observations

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

What's the class distribution?

[10]:
```
train_df["Class"].value_counts()
```

[10]:
```
0    199025
1       339
Name: Class, dtype: int64
```

[11]:
```
train_df["Class"].value_counts(normalize=True)
```

[11]:
```
0    0.9983
1    0.0017
Name: Class, dtype: float64
```

- We have 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.

Let's scale the features and try `LogisticRegression`.

[12]:
```
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression())
pd.DataFrame(cross_validate(pipe_lr, X_train, y_train,␣
↪return_train_score=True)).mean()
```

[12]:
```
fit_time       0.970541
score_time     0.013835
test_score     0.999176
train_score    0.999249
dtype: float64
```

- We are getting a slightly better score with logistic regression.

- What score should be considered an **acceptable score** here?
- Are we actually spotting any "fraud" transactions?

- `.score` by default returns accuracy which is

$$accuracy = \frac{correct\ predictions}{total\ examples}$$

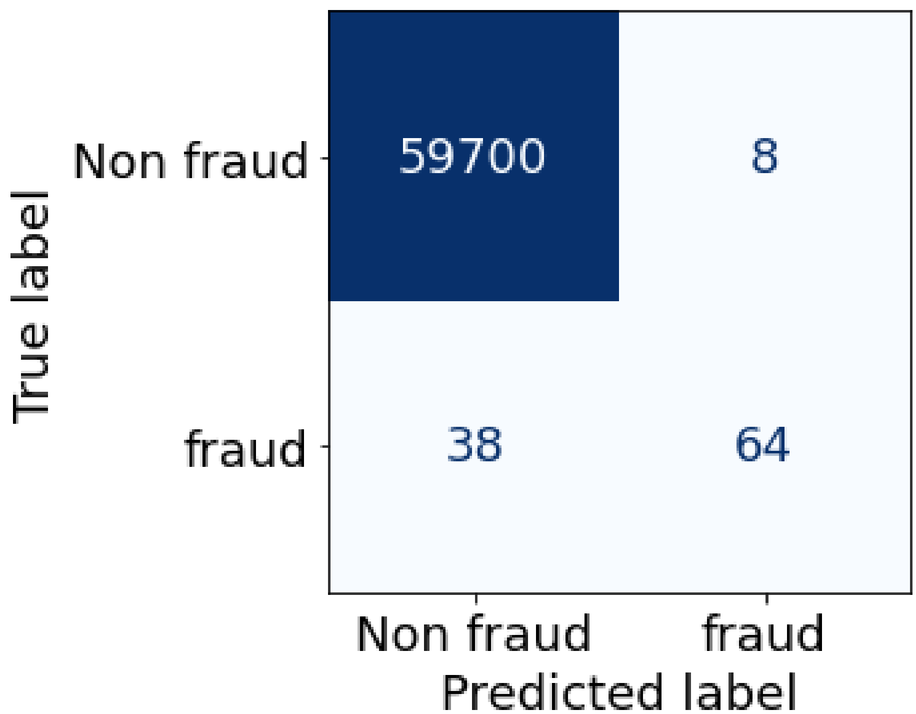
- Is accuracy a good metric here?
- Is there anything more informative than accuracy that we can use here?

Let's dig a little deeper.

1.4 Confusion matrix

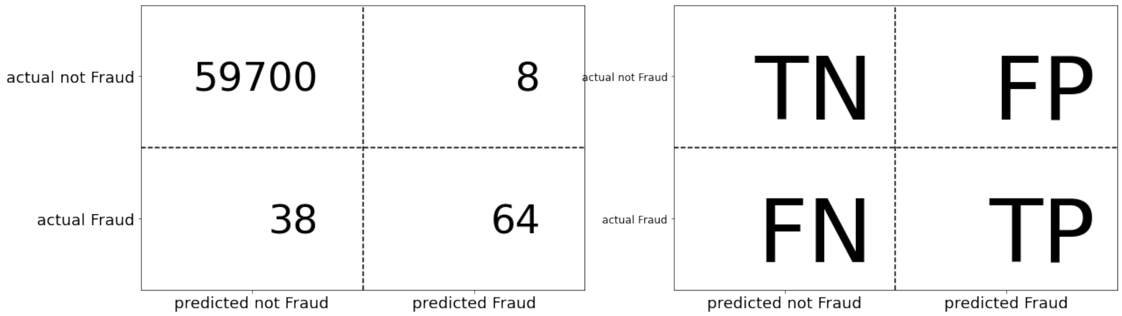
One way to get a better understanding of the errors is by looking at - false positives (type I errors), where the model incorrectly spots examples as fraud - false negatives (type II errors), where it's missing to spot fraud examples

[13]:
```
from sklearn.metrics import ConfusionMatrixDisplay
plt.rc('font', size=18)

pipe_lr.fit(X_train, y_train)
disp = ConfusionMatrixDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    display_labels=["Non fraud", "fraud"],
    values_format="d",
    cmap=plt.cm.Blues,
    colorbar=False,
)
```



[14]:
```
from sklearn.metrics import confusion_matrix

predictions = pipe_lr.predict(X_valid)
TN, FP, FN, TP = confusion_matrix(y_valid, predictions).ravel()
```

```
plot_confusion_matrix_example(TN, FP, FN, TP)
```



- **Perfect** prediction has all values **down the diagonal**
- **Off diagonal** entries can often tell us about what is being **mis-predicted**

1.4.1 What is "positive" and "negative"?

- Two kinds of binary classification problems
  - Distinguishing between two classes
  - **Spotting** a class (spot fraud transaction, spot spam, spot disease)
- In case of spotting problems, the thing that we are interested in spotting is considered "**positive**".
- Above we wanted to spot fraudulent transactions and so they are "positive".

You can get a numpy array of confusion matrix, and you can *unpack* it into its elements using numpy `ravel()` as follows:

[15]:
```
from sklearn.metrics import confusion_matrix

predictions = pipe_lr.predict(X_valid)  # note that pipe_lr must have already been fitted using train data
cm = confusion_matrix(y_valid, predictions)
TN, FP, FN, TP = cm.ravel()  # unpack cm elements
TN, FP, FN, TP
```

[15]:
```
(59700, 8, 38, 64)
```

[16]:
```
print("Confusion matrix for fraud dataset:\n", cm)
```

```
Confusion matrix for fraud dataset:
 [[59700     8]
 [   38    64]]
```

1.4.2 Confusion matrix with cross-validation

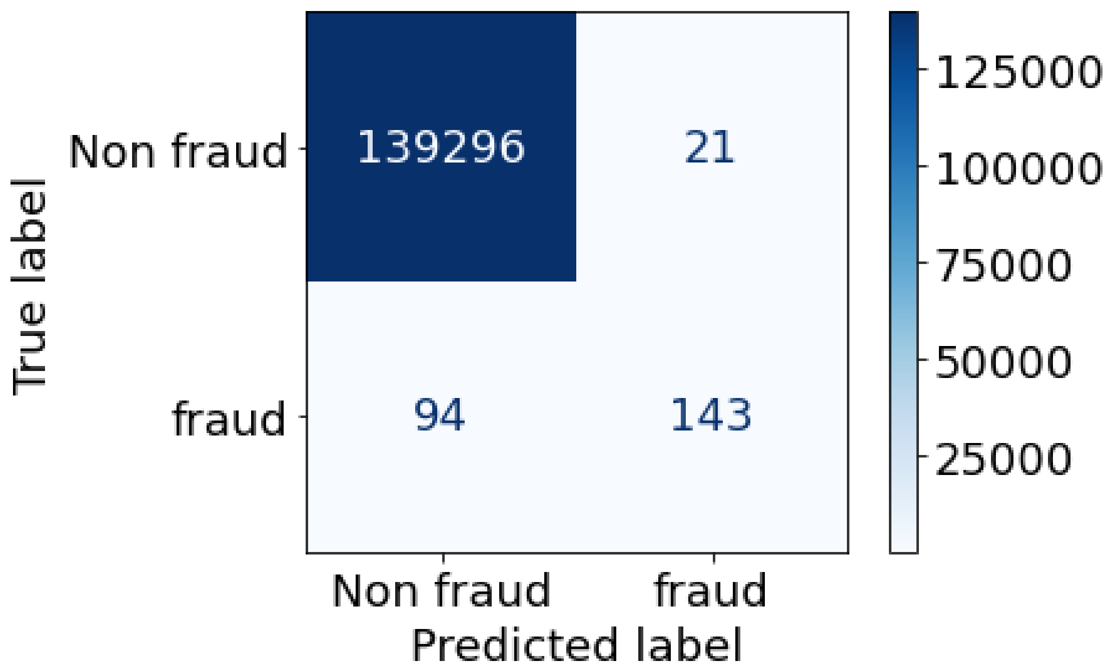
- You can also calculate confusion matrix with cross-validation using the `cross_val_predict` method.

- Then you need to use `ConfusionMatrixDisplay.from_predictions` to draw confusion matrix.

[17]:
```
from sklearn.model_selection import cross_val_predict

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

[17]:
```
array([[139296,     21],
       [    94,    143]])
```

[18]:
```
ConfusionMatrixDisplay.from_predictions(
    y_train,
    cross_val_predict(pipe_lr, X_train, y_train),
    display_labels=["Non fraud", "fraud"],
    values_format="d",
    cmap=plt.cm.Blues,
);
```



1.5 Precision, recall, f1 score

- We have been using `.score` to assess our models, which returns **accuracy by default**.
- Accuracy is misleading when we have class imbalance.
- We need other metrics to assess our models.

- We'll discuss three commonly used metrics which are **based on confusion matrix**:
  - *recall*

    - *precision*
    - *f1 score*
- Note that these metrics will only help us **assessing** our model.
- Later we'll talk about a few ways to address class imbalance problem.

[19]:
```
from sklearn.metrics import confusion_matrix

pipe_lr = make_pipeline(StandardScaler(), LogisticRegression())
pipe_lr.fit(X_train, y_train)
predictions = pipe_lr.predict(X_valid)
cm = confusion_matrix(y_valid, predictions)
TN, FP, FN, TP = cm.ravel()
print("TN, FP, FN, TP:", TN, FP, FN, TP, '\n')
cm
```

```
TN, FP, FN, TP: 59700 8 38 64
```

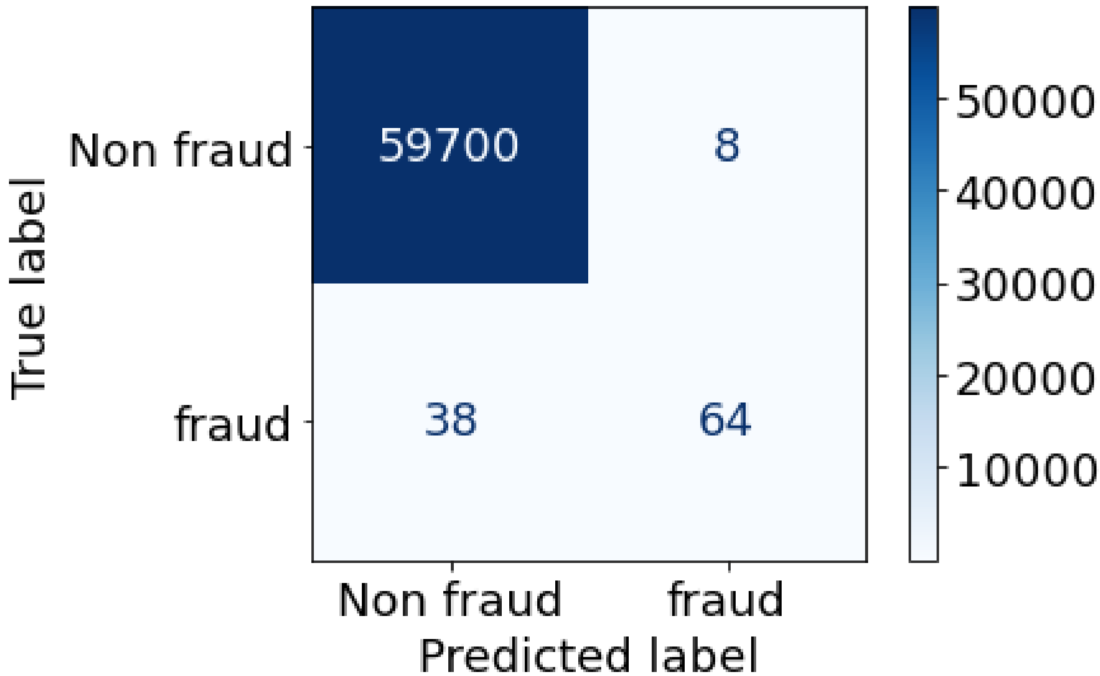
[19]:
```
array([[59700,     8],
       [   38,    64]])
```

1.5.1 Recall

Among all positive examples, how many did you identify?

$$recall = \frac{TP}{TP + FN} = \frac{TP}{\#positives}$$

[20]:
```
ConfusionMatrixDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    display_labels=["Non fraud", "fraud"],
    values_format="d",
    cmap=plt.cm.Blues,
);
```



[21]:
```
print("TP = %0.4f, FN = %0.4f" % (TP, FN))
recall = TP / (TP + FN)
print("Recall: %0.4f" % (recall))
```

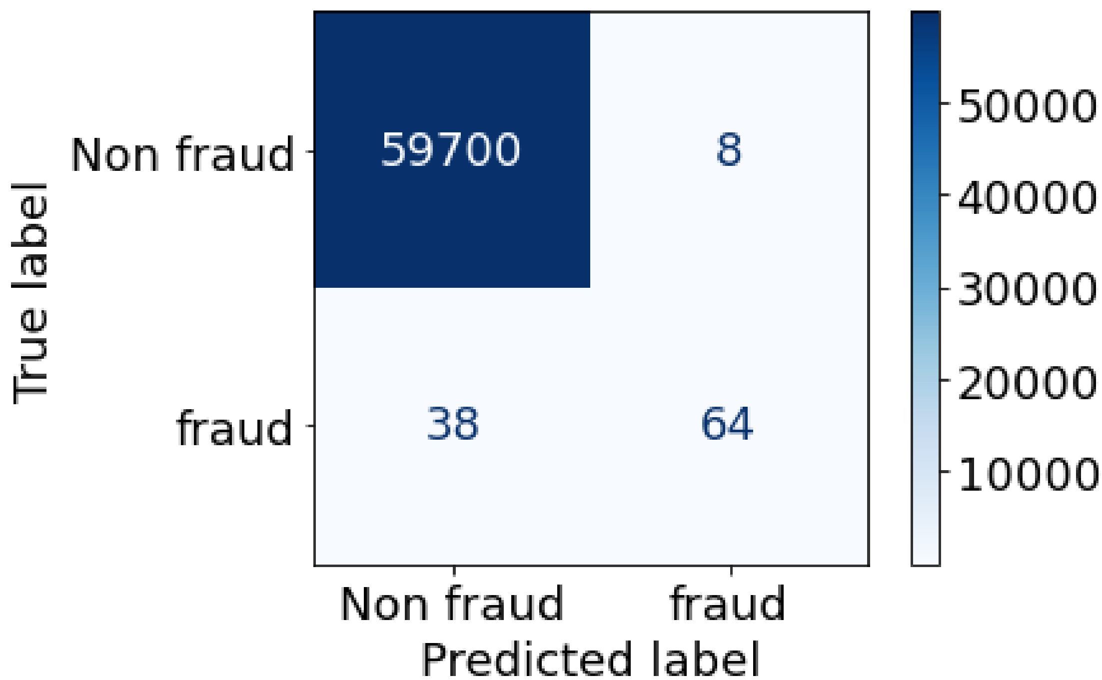
```
TP = 64.0000, FN = 38.0000
Recall: 0.6275
```

1.5.2 Precision

Among the positive examples you identified, how many were actually positive?

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

[22]:
```
ConfusionMatrixDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    display_labels=["Non fraud", "fraud"],
    values_format="d",
    cmap=plt.cm.Blues,
);
```




```
[23]: print("TP = %0.4f, FP = %0.4f" % (TP, FP))
      precision = TP / (TP + FP)
      print("Precision: %0.4f" % (precision))
```

```
TP = 64.0000, FP = 8.0000
Precision: 0.8889
```

1.5.3 F1-score

- F1-score **combines precision and recall** to give one score, which could be used in hyper-parameter optimization, for instance.
- F1-score is a harmonic mean of precision and recall.

$$f1 = 2 \times \frac{precision \times recall}{precision + recall}$$

```
[24]: print("precision: %0.4f" % (precision))
      print("recall: %0.4f" % (recall))
      f1_score = (2 * precision * recall) / (precision + recall)
      print("f1: %0.4f" % (f1_score))
```

```
precision: 0.8889
recall: 0.6275
f1: 0.7356
```

Let's look at all metrics at once on our dataset.

```
[25]: ## Calculate evaluation metrics by ourselves
data = {
    "calculation": [],
    "accuracy": [],
    "error": [],
    "precision": [],
    "recall": [],
    "f1 score": [],
}
data["calculation"].append("manual")
data["accuracy"].append((TP + TN) / (TN + FP + FN + TP))
data["error"].append((FP + FN) / (TN + FP + FN + TP))
data["precision"].append(precision)  # TP / (TP + FP)
data["recall"].append(recall)  # TP / (TP + FN)
data["f1 score"].append(f1_score)  # (2 * precision * recall) / (precision +␣
 ↪recall)
df = pd.DataFrame(data)
df
```

```
[25]:   calculation  accuracy     error  precision    recall  f1 score
      0      manual  0.999231  0.000769   0.888889  0.627451  0.735632
```

- `scikit-learn` has functions for these metrics.

```
[26]: from sklearn.metrics import accuracy_score, f1_score, precision_score,␣
 ↪recall_score

data["accuracy"].append(accuracy_score(y_valid, pipe_lr.predict(X_valid)))
data["error"].append(1 - accuracy_score(y_valid, pipe_lr.predict(X_valid)))
data["precision"].append(
    precision_score(y_valid, pipe_lr.predict(X_valid), zero_division=1)
)
data["recall"].append(recall_score(y_valid, pipe_lr.predict(X_valid)))
data["f1 score"].append(f1_score(y_valid, pipe_lr.predict(X_valid)))
data["calculation"].append("sklearn")
df = pd.DataFrame(data)
df.set_index(["calculation"])
```

```
[26]:              accuracy     error  precision    recall  f1 score
      calculation
      manual       0.999231  0.000769   0.888889  0.627451  0.735632
      sklearn      0.999231  0.000769   0.888889  0.627451  0.735632
```

The scores match.

1.5.4 Classification report

- There is a convenient function called `classification_report` in `sklearn` which gives this info.

[27]:
```
pipe_lr.classes_
```

[27]:
```
array([0, 1])
```

[28]:
```
from sklearn.metrics import classification_report

print("y_valid, not fraud:", len(y_valid) - y_valid.sum())
print("y_valid, fraud:      ", y_valid.sum())
print("X_valid, total:    ", X_valid.shape[0], "\n\n")

print(classification_report(
        y_valid, pipe_lr.predict(X_valid), target_names=["non-fraud", "fraud"],␣
 ↪digits=4))
```

```
y_valid, not fraud: 59708
y_valid, fraud:       102
X_valid, total:     59810


              precision    recall  f1-score   support

   non-fraud     0.9994    0.9999    0.9996     59708
       fraud     0.8889    0.6275    0.7356       102

    accuracy                         0.9992     59810
   macro avg     0.9441    0.8137    0.8676     59810
weighted avg     0.9992    0.9992    0.9992     59810
```

[29]:
```
cr_dict = classification_report(
    y_valid, pipe_lr.predict(X_valid), target_names=["non-fraud", "fraud"],␣
 ↪output_dict=True)

cr = pd.DataFrame(cr_dict).T
cr
```

[29]:

	precision	recall	f1-score	support
non-fraud	0.999364	0.999866	0.999615	59708.000000
fraud	0.888889	0.627451	0.735632	102.000000
accuracy	0.999231	0.999231	0.999231	0.999231
macro avg	0.944126	0.813658	0.867624	59810.000000
weighted avg	0.999175	0.999231	0.999165	59810.000000

Let us manually calculate `macro avg` for `precision`, `recall`, and `f1-score`:

[30]:
```
cr.loc[['non-fraud','fraud']]  # the relevant subset of cr
```

[30]:

	precision	recall	f1-score	support
non-fraud	0.999364	0.999866	0.999615	59708.0
fraud	0.888889	0.627451	0.735632	102.0

[31]:
```
pd.concat([
    cr.loc[['non-fraud','fraud']].mean().rename('mean of "non-fraud" and␣
↪"fraud"'),
    cr.loc['macro avg']], axis=1).head(3).T
```

[31]:

	precision	recall	f1-score
mean of "non-fraud" and "fraud"	0.944126	0.813658	0.867624
macro avg	0.944126	0.813658	0.867624

So, our manual calculation matches `macro avg` calculated by `classification_report`.

Now, let us manually calculate `weighted_avg_of_precision`:

[32]:
```
cr_precision = cr.loc[['non-fraud','fraud'], ['precision','support']] #␣
↪relevant subset of cr
cr_precision
```

[32]:

	precision	support
non-fraud	0.999364	59708.0
fraud	0.888889	102.0

[33]:
```
weighted_avg_of_precision = cr_precision.product(axis=1).sum() /␣
↪cr_precision['support'].sum()
weighted_avg_of_precision
```

[33]: 0.9991754848687043

[34]:
```
weighted_avg_of_precision == cr.loc['weighted avg', 'precision']
```

[34]: True

Can you also manually calculate `weighted_avg_of_recall` and `weighted_avg_of_f1_score`?

1.5.5 Macro average

- You give **equal importance** to all classes and average over all classes.
- See our example above, calculating `macro avg` for `precision`, `recall`, and `f1-score`
- More relevant in case of **multi-class** problems.

1.5.6 Weighted average

- Weighted by the number of samples in each class.
- Divide by the total number of samples.
- See our example above, calculating `weighted_avg_of_precision`

1.5.7 Which one to use?

Which one of Weighted or Macro averages is relevant depends upon whether you think:

- each class should have the same weight or
- each sample should have the same weight.

That is, it will be domain/problem dependent.

1.5.8 Interim summary

- **Accuracy** is misleading when you have class **imbalance**.
- A **confusion matrix** provides a way to **break down errors** made by our model.
- 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.

1.5.9 Evalution metrics overview

There is a lot of terminology here.

1.5.10 Cross validation with different metrics

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

[35]:
```
scoring = [
    "accuracy",
    "f1",
    "recall",
    "precision",
]  # scoring can be a string, a list, or a dictionary
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression())
scores = cross_validate(
    pipe_lr, X_train_big, y_train_big, return_train_score=True, scoring=scoring
)
pd.DataFrame(scores)
```

[35]:
```
   fit_time  score_time  test_accuracy  train_accuracy   test_f1  train_f1  \
0  1.433211    0.073632       0.999147        0.999367  0.711864  0.783726
1  1.219011    0.072988       0.999298        0.999329  0.766667  0.770878
2  1.246399    0.093531       0.999273        0.999216  0.743363  0.726477
3  1.244750    0.070846       0.999172        0.999279  0.697248  0.753747
4  1.210791    0.088307       0.999172        0.999223  0.702703  0.731602

   test_recall  train_recall  test_precision  train_precision
0     0.617647      0.675277        0.840000         0.933673
1     0.676471      0.664207        0.884615         0.918367
```

2	0.617647	0.612546	0.933333	0.892473
3	0.558824	0.649446	0.926829	0.897959
4	0.582090	0.621324	0.886364	0.889474

- You can also create your own scoring function and pass it to `cross_validate`.

1.5.11 Questions for you

1.5.12 True/False questions: decision theory, evaluation metrics

1. In medical diagnosis, false positives are more damaging than false negatives (assume "positive" means the person has a disease, "negative" means they don't). **FALSE**
2. In spam classification, false positives are more damaging than false negatives (assume "positive" means the email is spam, "negative" means they it's not). **FALSE**
3. In the medical diagnosis, high recall is more important than high precision. **TRUE**
4. If method A gets a higher accuracy than method B, that means its precision is also higher. **FALSE**
5. If method A gets a higher accuracy than method B, that means its recall is also higher. **FALSE** Both cases can be caused by very imbalanced dataset.

Method A - higher accuracy but lower precision

Negative	Positive
90	5
5	0

Method B - lower accuracy but higher precision

Negative	Positive
80	15
0	5

1.6 Precision-recall curve and ROC 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.

[36]:
```
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression())
pipe_lr.fit(X_train, y_train);
```

[37]:
```
y_pred = pipe_lr.predict(X_valid)
```

```
print(classification_report(y_valid, y_pred, target_names=["non-fraud",␣
↪"fraud"], digits=4))
```

```
              precision    recall  f1-score   support

   non-fraud     0.9994    0.9999    0.9996     59708
       fraud     0.8889    0.6275    0.7356       102

    accuracy                         0.9992     59810
   macro avg     0.9441    0.8137    0.8676     59810
weighted avg     0.9992    0.9992    0.9992     59810
```

By default, predictions use the **threshold of 0.5**. If `predict_proba` > 0.5, predict "fraud" (positive) else predict "non-fraud" (negative).

In the above code, the function `predict` returns a boolean array, `y_pred`:

[38]:
```
y_pred
```

[38]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[39]:
```
np.unique(y_pred)
```

[39]:
```
array([0, 1])
```

We can create the same boolean array `y_pred` directly using `predict_proba` > 0.5:

[40]:
```
# negative class column is 0, and positive class column is 1, so we want [:, 1]
y_pred = pipe_lr.predict_proba(X_valid)[:, 1] > 0.50
print(classification_report(y_valid, y_pred, target_names=["non-fraud",␣
↪"fraud"], digits=4))
```

```
              precision    recall  f1-score   support

   non-fraud     0.9994    0.9999    0.9996     59708
       fraud     0.8889    0.6275    0.7356       102

    accuracy                         0.9992     59810
   macro avg     0.9441    0.8137    0.8676     59810
weighted avg     0.9992    0.9992    0.9992     59810
```

Now, - Suppose for your business it is more costly to miss fraudulent transactions and 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.

[41]:
```
y_pred_lower_threshold = pipe_lr.predict_proba(X_valid)[:, 1] > 0.1
```

[42]:
```
print(classification_report(y_valid, y_pred_lower_threshold, digits=4))
```

```
              precision    recall  f1-score   support

           0     0.9996    0.9996    0.9996     59708
           1     0.7800    0.7647    0.7723       102

    accuracy                         0.9992     59810
   macro avg     0.8898    0.8822    0.8859     59810
weighted avg     0.9992    0.9992    0.9992     59810
```

1.6.1 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.

1.6.2 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.

[43]:
```
thresholds = np.arange(0, 1, 0.1)
thresholds
```

[43]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

[44]:
```
pr_dict = {"threshold": [], "precision": [], "recall": [], "f1 score": []}
for threshold in thresholds:
    preds = pipe_lr.predict_proba(X_valid)[:, 1] > threshold
    pr_dict["threshold"].append(threshold)
    pr_dict["precision"].append(precision_score(y_valid, preds))
    pr_dict["recall"].append(recall_score(y_valid, preds))
    pr_dict["f1 score"].append(f1_score(y_valid, preds))
```

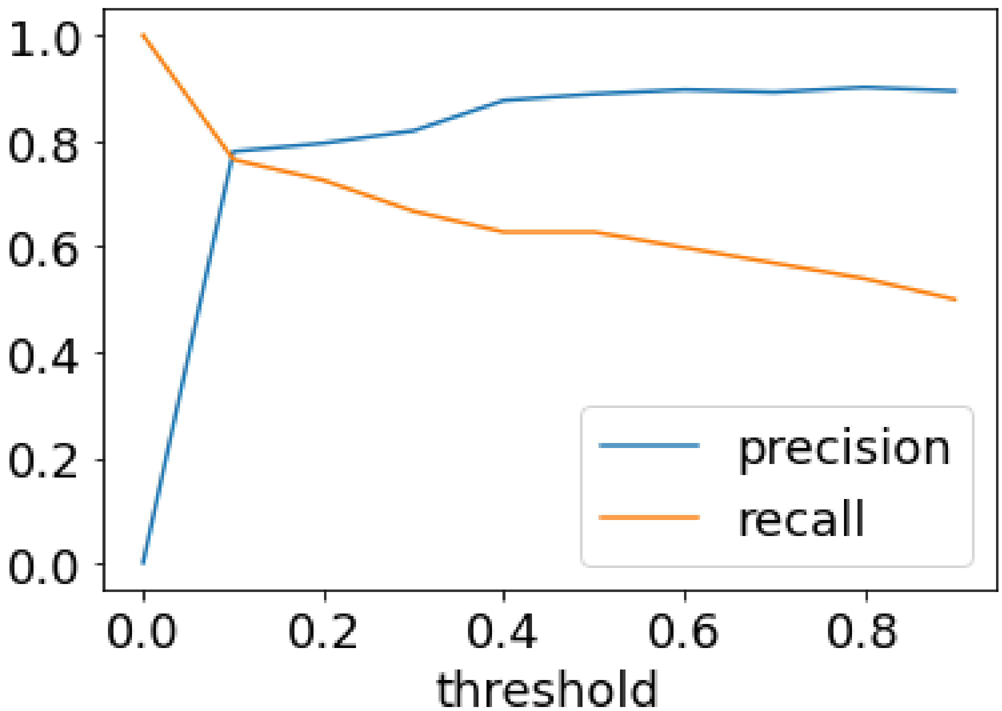
[45]:
```
pr_df = pd.DataFrame(pr_dict).set_index('threshold')
pr_df
```

[45]:

threshold	precision	recall	f1 score
0.0	0.001705	1.000000	0.003405
0.1	0.780000	0.764706	0.772277
0.2	0.795699	0.725490	0.758974
0.3	0.819277	0.666667	0.735135
0.4	0.876712	0.627451	0.731429
0.5	0.888889	0.627451	0.735632
0.6	0.897059	0.598039	0.717647
0.7	0.892308	0.568627	0.694611
0.8	0.901639	0.539216	0.674847
0.9	0.894737	0.500000	0.641509

[46]:

```
pr_df[['precision', 'recall']].plot();
```



1.6.3 Decreasing the threshold

- ***Decreasing the threshold*** means a lower bar for predicting fraud.
  - You are willing to risk more false positives FP in exchange of more true positives TP .
    - In general, predicted positives (TP + FP) go up or stay the same
    - In general, predicted negatives (TN + FN) go down or stay the same
  - recall is likely to go up or stay the same
    - TP / (TP +FN) so generally recall
  - precision is likely to go down or stay the same

* TP / (TP +FP) so generally precision

1.6.4 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.

[47]:
```
from sklearn.metrics import precision_recall_curve

precision, recall, thresholds = precision_recall_curve(
    y_valid, pipe_lr.predict_proba(X_valid)[:, 1]
)
print(precision.shape, recall.shape, thresholds.shape)
thresholds = np.append(thresholds, 1)  # make thresholds same length as␣
↪precision and recall
```

```
(50990,) (50990,) (50989,)
```

[48]:
```
df_PR = pd.DataFrame(
    {"precision": precision, "recall":recall, "thresholds":thresholds}).
↪set_index("thresholds")
df_PR.head()
```

[48]:

thresholds	precision	recall
0.00005	0.001998	1.000000
0.00005	0.001979	0.990196
0.00005	0.001979	0.990196
0.00005	0.001979	0.990196
0.00005	0.001979	0.990196

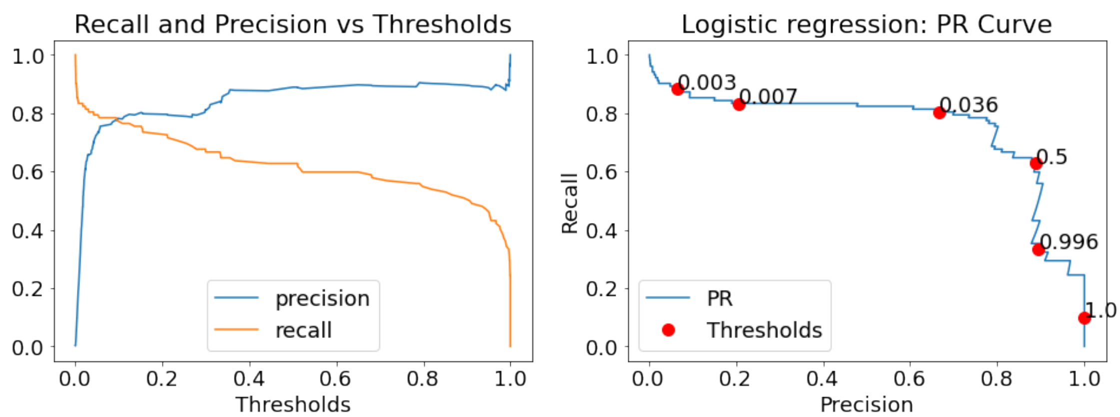
[49]:
```
# ensure that precision and recall exist for threshold 0.5
df_PR.loc[0.5] = [
    precision_score(y_valid, pipe_lr.predict(X_valid)),
    recall_score(y_valid, pipe_lr.predict(X_valid))]
df_PR = df_PR.sort_index()
# df_PR.query("0.4 < thresholds < 0.6")
```

[50]:
```
# select some table rows (including threshold 0.5) to draw below
rows = np.geomspace(1, len(df_PR), num=10)
rows = np.unique(rows.astype(int))  # truncate to integers and drop duplicates
rows = max(rows) - rows  # get reverse geomspace: going from larger to shorter␣
↪distances
rows = np.append(rows, df_PR.index.get_loc(0.5))  # add the index for threshold␣
↪0.5
rows = np.sort(rows)
```

```
rows = rows[3:-2] # the end points on graph will be too close, so drop some␣
 ↪points at both ends
rows # the indices to some thresholds
```

[50]:
```
array([49616, 50579, 50868, 50919, 50954, 50980])
```

[51]:
```
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

df_PR.plot(ax=axes[0])
axes[0].set_title("Recall and Precision vs Thresholds")
axes[0].set_xlabel("Thresholds")
axes[0].legend(loc="best")

df_PR.plot(x="precision", y="recall", ax=axes[1], label="PR")
df_PR.iloc[rows].plot(
    x="precision", y="recall", style="or", markersize=10, ax=axes[1],␣
 ↪label="Thresholds")
axes[1].set_title("Logistic regression: PR Curve")
axes[1].set_xlabel("Precision")
axes[1].set_ylabel("Recall")

for thres, pr in df_PR.iloc[rows].iterrows():
    axes[1].annotate(round(thres, 3), pr)
```



- Each point in the curve corresponds to a possible threshold of the `predict_proba` output.
- We can achieve a recall of 0.8 at a precision of 0.4.
- The red dots mark the point corresponding to some thresholds, including 0.5.
- The top-right would be a perfect classifier (precision = recall = 1).

- The threshold is going from 0 (upper-left) to 1 (lower right).
- At a threshold of 0 (upper left), we are classifying everything as "fraud".
- Raising the threshold increases the precision but at the expense of lowering the recall.
- At the extreme right, where the threshold is 1, we get into the situation where all the examples

classified as “fraud” are actually “fraud”; we have no false positives.
- Here we have a high precision but lower recall.
- Usually the goal is to keep recall high as precision goes up.

1.6.5 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.

1.6.6 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)
- AP score has a value between 0 (worst) and 1 (best).

[52]:
```
from sklearn.metrics import average_precision_score

ap_lr = average_precision_score(y_valid, pipe_lr.predict_proba(X_valid)[:, 1])
print(f"Average precision of logistic regression: {ap_lr:.3f}")
```

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

1.6.7 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`.

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

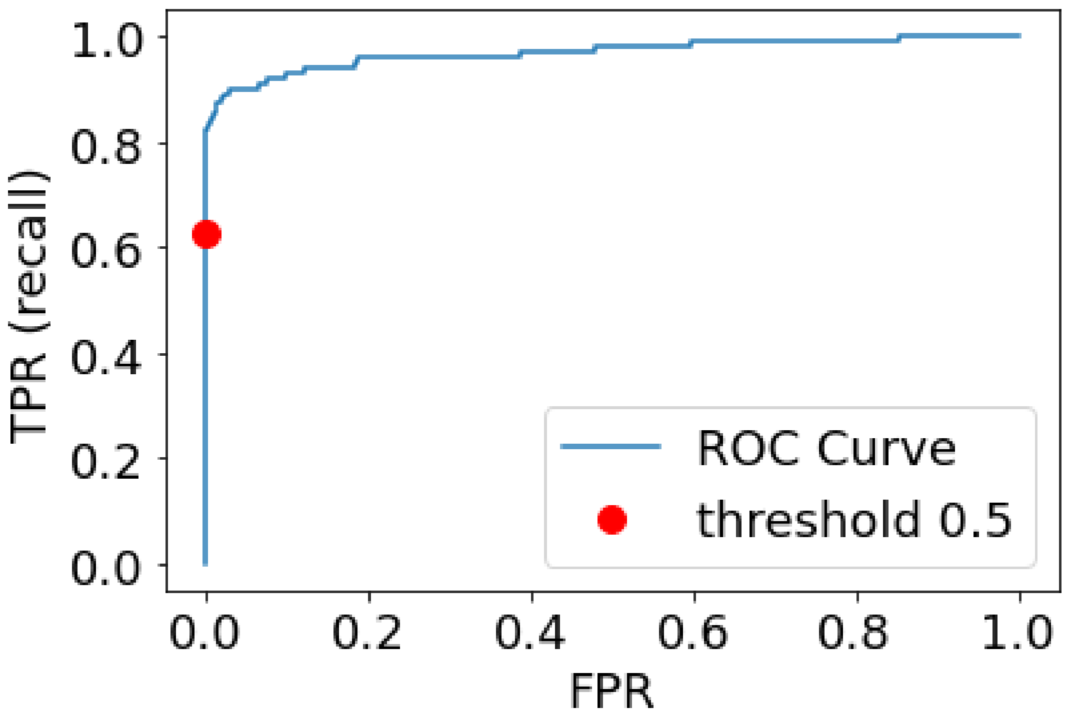
1.6.8 Receiver Operating Characteristic (ROC) curve

- 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 it plots false positive rate (FPR) and true positive rate (TPR or recall).

$$FPR = \frac{FP}{FP + TN}, TPR = \frac{TP}{TP + FN}$$

[53]:
```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_valid, pipe_lr.predict_proba(X_valid)[:, 1])
plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
```

```
plt.ylabel("TPR (recall)")

default_threshold = np.argmin(np.abs(thresholds - 0.5))

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



- 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.

1.6.9 Area under the curve (AUC)

- AUC provides a single meaningful number for the model performance.

[54]:
```
from sklearn.metrics import roc_auc_score
```

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

```
AUC for SVC: 0.969
```

- AUC of 0.5 means random chance.
- 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.

1.6.10 Let's look at all the scores at once

[55]:
```
scoring = ["accuracy", "f1", "recall", "precision", "roc_auc",␣
 ↪"average_precision"]
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression())
scores = cross_validate(pipe_lr, X_train_big, y_train_big, scoring=scoring)
pd.DataFrame(scores).mean()
```

[55]:
```
fit_time                  1.236158
score_time                0.130924
test_accuracy             0.999212
test_f1                   0.724369
test_recall               0.610536
test_precision            0.894228
test_roc_auc              0.967438
test_average_precision    0.744030
dtype: float64
```

See Also Check out these visualization on ROC and AUC.

1.7 Dealing with class imbalance

1.7.1 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.

1.7.2 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.

1.7.3 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.

[56]:
```
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, target_names=["non-fraud",␣
 ↪"fraud"], digits=4))
```

```
              precision    recall  f1-score   support

   non-fraud     0.9994    0.9999    0.9996     59708
       fraud     0.8889    0.6275    0.7356       102

    accuracy                         0.9992     59810
   macro avg     0.9441    0.8137    0.8676     59810
weighted avg     0.9992    0.9992    0.9992     59810
```

[57]:
```
y_pred = pipe_lr.predict_proba(X_valid)[:, 1] > 0.10
print(classification_report(y_valid, y_pred, target_names=["non-fraud",␣
 ↪"fraud"], digits=4))
```

```
              precision    recall  f1-score   support

   non-fraud     0.9996    0.9996    0.9996     59708
       fraud     0.7800    0.7647    0.7723       102

    accuracy                         0.9992     59810
   macro avg     0.8898    0.8822    0.8859     59810
weighted avg     0.9992    0.9992    0.9992     59810
```

1.7.4 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`

1.7.5 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.

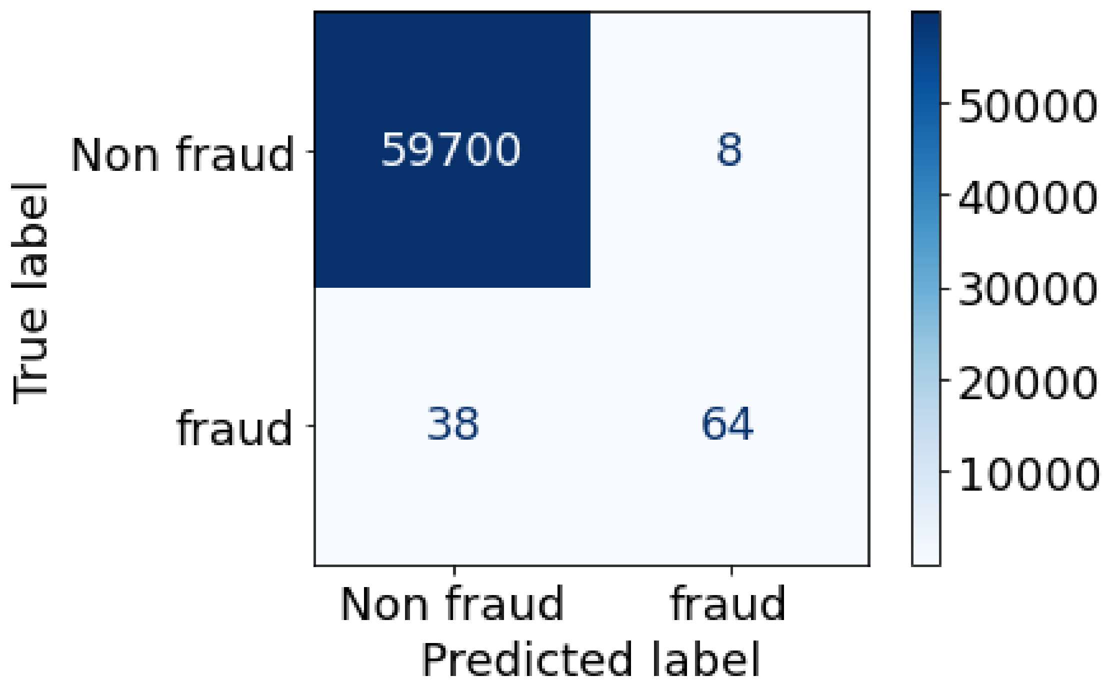
1.7.6 Example: `class_weight` parameter of `sklearn LogisticRegression`

> 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.

[58]:
```python
from IPython.display import IFrame
url = "https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.
↪LogisticRegression.html"
IFrame(src=url, width=1200, height=600)
```

[58]: `<IPython.lib.display.IFrame at 0x7f2c446bd180>`

[59]:
```python
ConfusionMatrixDisplay.from_estimator(
    pipe_lr,
    X_valid,
    y_valid,
    display_labels=["Non fraud", "fraud"],
    values_format="d",
    cmap=plt.cm.Blues,
);
```

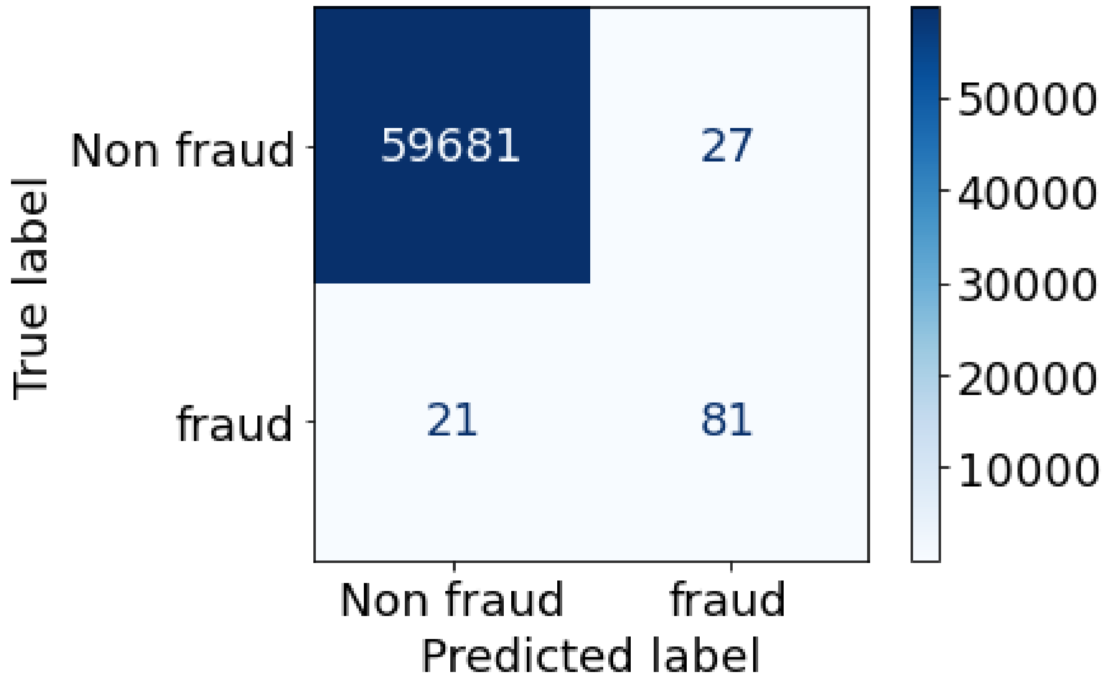


[60]:
```
pipe_lr.named_steps["logisticregression"].classes_
```

[60]:
```
array([0, 1])
```

Let's set "fraud" class a weight of 10.

[61]:
```
pipe_lr_weight = make_pipeline(
    StandardScaler(), LogisticRegression(max_iter=500, class_weight={0:1, 1:10})
)
pipe_lr_weight.fit(X_train, y_train)
ConfusionMatrixDisplay.from_estimator(
    pipe_lr_weight,
    X_valid,
    y_valid,
    display_labels=["Non fraud", "fraud"],
    values_format="d",
    cmap=plt.cm.Blues,
);
```



- 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**.

1.7.7 `class_weight="balanced"`

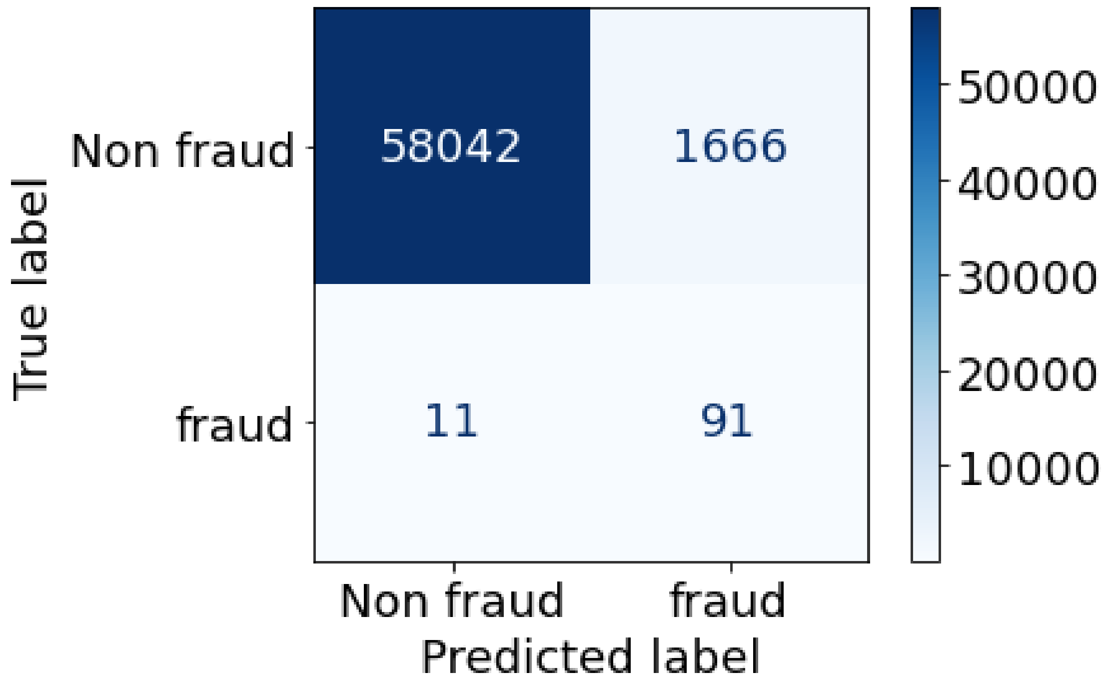
- A useful setting is `class_weight="balanced"`.
- This sets the weights so that the classes are "equal".

> 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)

[62]:
```
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,
    display_labels=["Non fraud", "fraud"],
    values_format="d",
```

```
    cmap=plt.cm.Blues,
);
```



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

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

[63]:
```
comp_dict = {}
pipe_lr = make_pipeline(StandardScaler(), LogisticRegression(max_iter=500))
scoring = ["accuracy", "f1", "recall", "precision", "roc_auc",␣
 ↪"average_precision"]
orig_scores = cross_validate(pipe_lr, X_train_big, y_train_big, scoring=scoring)
```

[64]:
```
pipe_lr_balanced = make_pipeline(
    StandardScaler(), LogisticRegression(max_iter=500, class_weight="balanced")
)
scoring = ["accuracy", "f1", "recall", "precision", "roc_auc",␣
 ↪"average_precision"]
bal_scores = cross_validate(pipe_lr_balanced, X_train_big, y_train_big,␣
 ↪scoring=scoring)
comp_dict = {
    "Original": pd.DataFrame(orig_scores).mean().tolist(),
    "class_weight='balanced'": pd.DataFrame(bal_scores).mean().tolist(),
}
```

[65]:
```
pd.DataFrame(comp_dict, index=bal_scores.keys())
```

[65]:

	Original	class_weight='balanced'
fit_time	1.226477	1.365792
score_time	0.127844	0.135241
test_accuracy	0.999212	0.973626
test_f1	0.724369	0.103831
test_recall	0.610536	0.896883
test_precision	0.894228	0.055119
test_roc_auc	0.967438	0.970881
test_average_precision	0.744030	0.730627

- 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.

1.7.9 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% positive examples in total, then each fold will have 10% positive examples.

1.7.10 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.

1.8 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`

1.8.1 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