Evaluation metrics for classification

Evaluation metrics for classification

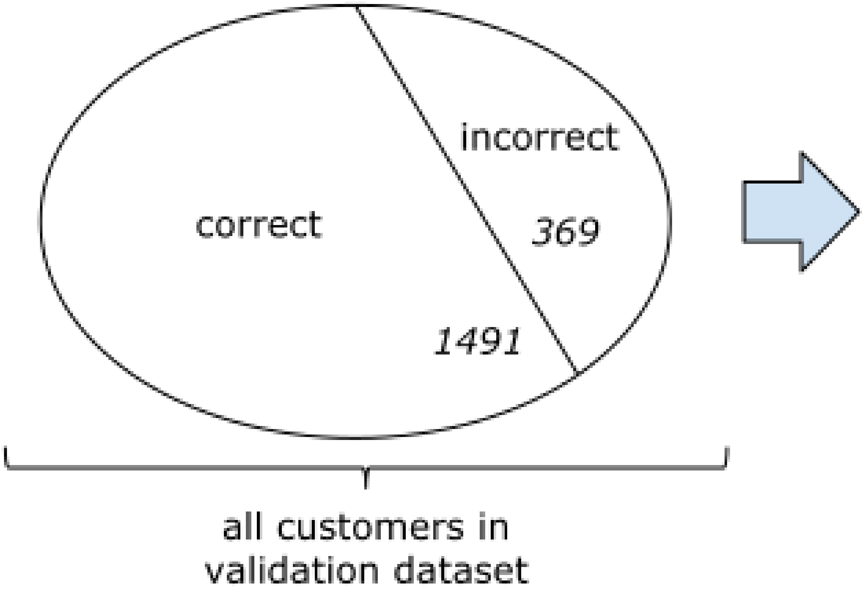
This lesson covers

- Accuracy as a way of evaluating binary classification models and its limitations
- Determining where our model makes mistakes using a confusion table
- Deriving other metrics like precision and recall from the confusion table
- Using ROC (receiver operating characteristics) and AUC (area under the ROC curve) to further understand the performance of a binary classification model

Evaluation metrics

- For this, we use a metric – a function that looks at the predictions the model makes and compares them with the actual values.
- Then, based on the comparison, it calculates how good the model is.
- This is quite useful: we can use it to compare different models and select the one with the best metric value.

Classification accuracy



$$\textbf{accuracy} = \frac{\text{correct}}{\text{total}} = \frac{1491}{1860} = 80\%$$

Classification accuracy

- Computing accuracy on the validation dataset is easy: we simply calculate the fraction of correct predictions:

```
y_pred = model.predict_proba(X_val)[:, 1] # A
churn = y_pred >= 0.5 # B
(churn == y_val).mean() # C
```

Classification accuracy

- Let's open it and add the import statement to import accuracy from Scikit-Learn's metrics package:

```
from sklearn.metrics import accuracy_score
```

Classification accuracy

- Now we can loop over different thresholds and check which one gives the best accuracy:

```
thresholds = np.linspace(0, 1, 11)

for t in thresholds:
    churn = y_pred >= t
    acc = accuracy_score(y_val, churn)
    print('%0.2f %0.3f' % (t, acc))
```

Classification accuracy

- When we execute the code, it prints the following:

```
0.00 0.261
0.10 0.595
0.20 0.690
0.30 0.755
0.40 0.782
0.50 0.802
0.60 0.790
0.70 0.774
0.80 0.742
0.90 0.739
1.00 0.739
```

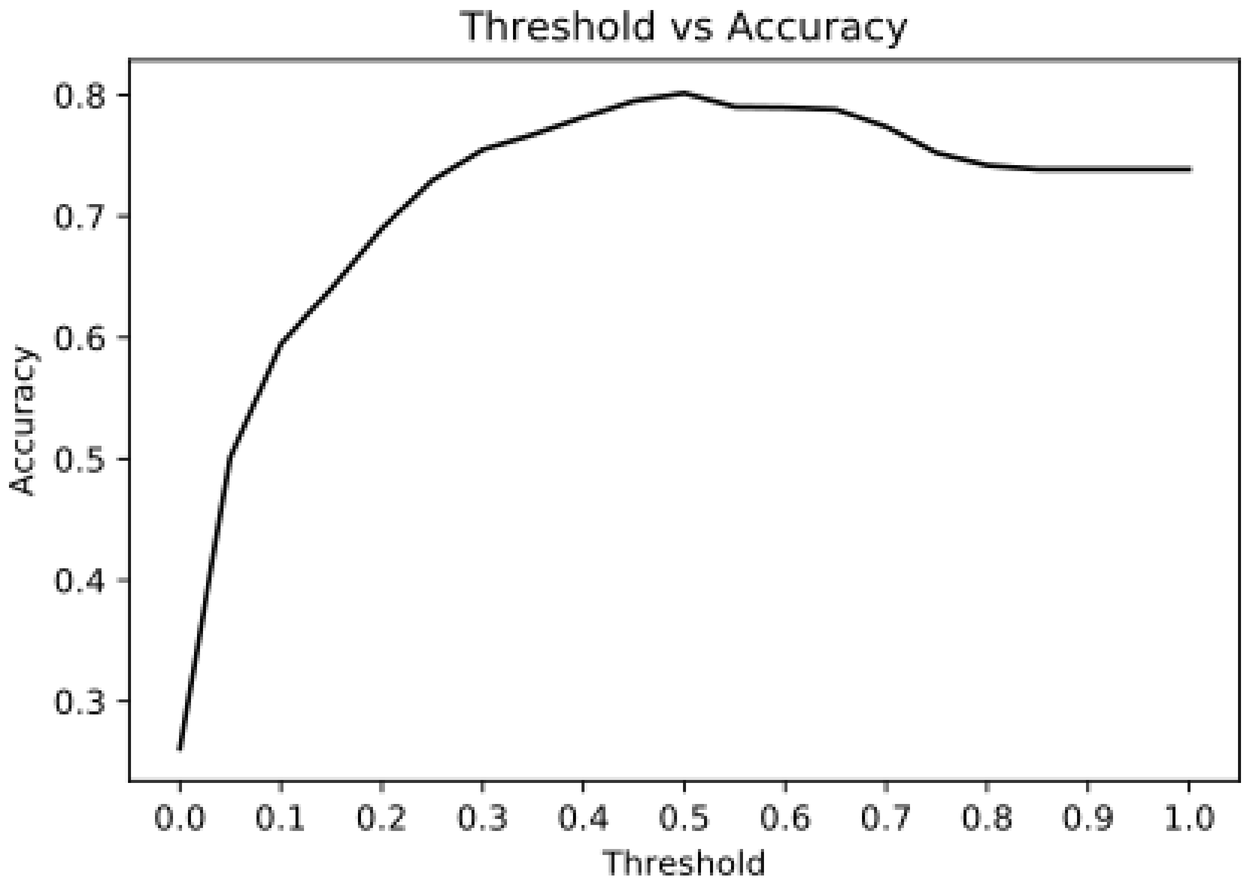
Classification accuracy

- We first put the values to a list:

```
thresholds = np.linspace(0, 1, 21)
accuracies = []
for t in thresholds:
    acc = accuracy_score(y_val, y_pred >= t)
    accuracies.append(acc)
```

- And then we plot these values using Matplotlib:

```
plt.plot(thresholds, accuracies)
```


Threshold vs Accuracy
Accuracy
Threshold
0.3
0.4
0.5
0.6
0.7
0.8
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Classification accuracy

- Let's also check its accuracy, For that, we first make predictions on the validation dataset and then compute the accuracy score:

```
val_dict_small = df_val[small_subset].to_dict(orient='rows')

X_small_val = dv_small.transform(val_dict_small)
y_pred_small = model_small.predict_proba(X_small_val)[:, 1]

churn_small = y_pred_small >= 0.5
accuracy_score(y_val, churn_small)
```

Dummy baseline

- Let's create this baseline prediction:

```
size_val = len(y_val)
baseline = np.repeat(False, size_val)
```

- To create an array with the baseline predictions we first need to determine how many elements are in the validation set.

Dummy baseline

- Now we can check the accuracy of this baseline prediction using the same code as previously:

```
accuracy_score(baseline, y_val)
```

Dummy baseline

- When we run the code, it shows 0.738.
- This means that the accuracy of the baseline model is around 74%

```
size_val = len(y_val)
baseline = np.repeat(False, size_val)
baseline
```

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

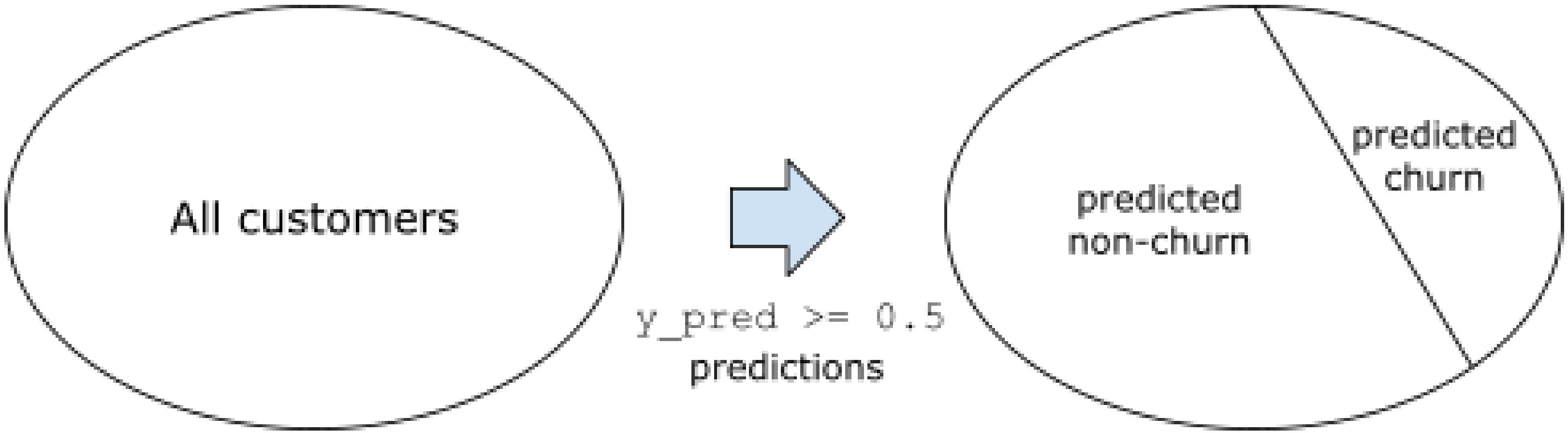
```
accuracy_score(baseline, y_val)
```

```
0.7387096774193549
```

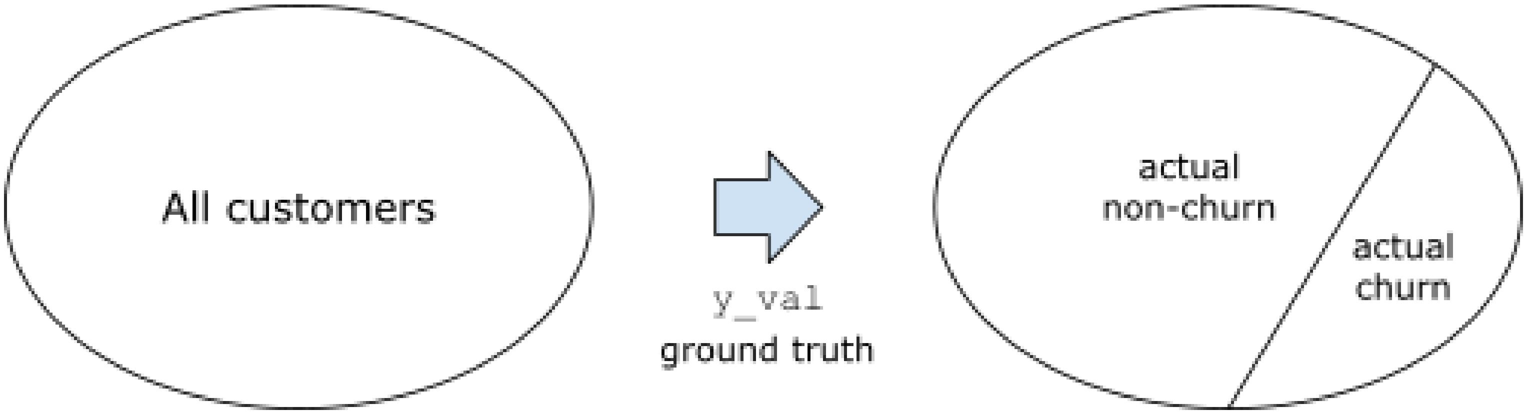
Confusion table

- Even though accuracy is easy to understand, it's not always the best metric.
- What is more, it sometimes can be quite misleading.
- We've already seen it: the accuracy of our model is 80%, and while that seems like a good number, it's just 6% better than the accuracy of a dummy model that always outputs the same prediction of "no churn."

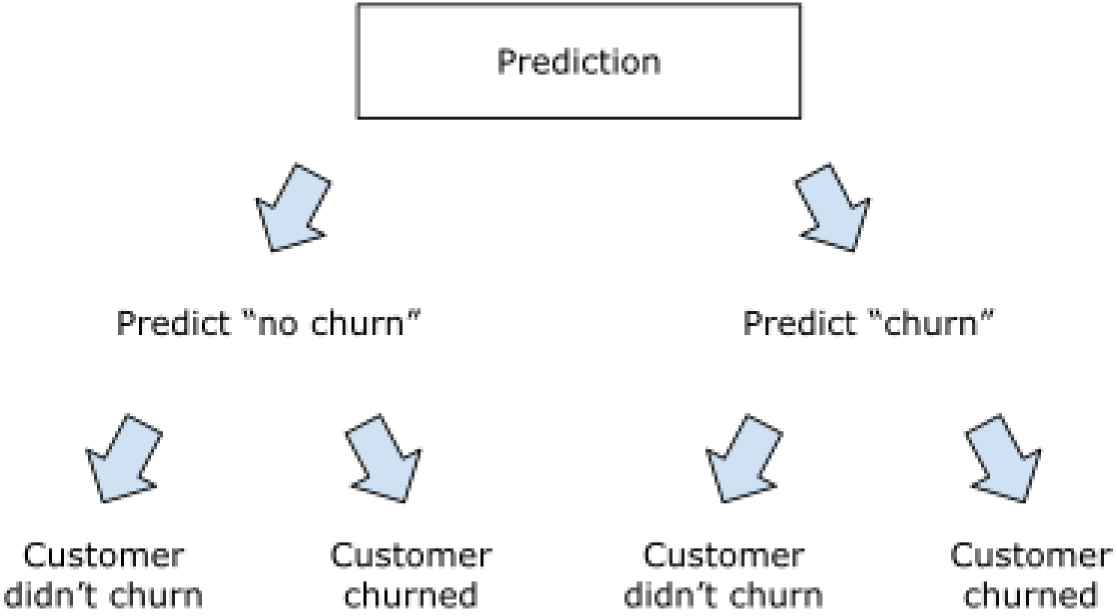
Introduction to confusion table



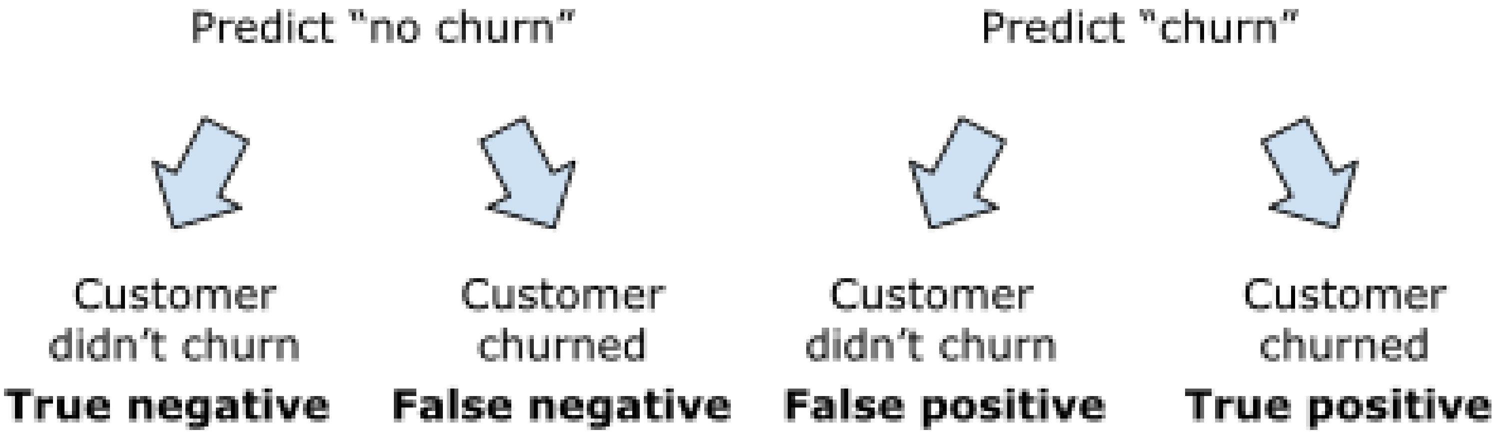
Introduction to confusion table



Introduction to confusion table



Introduction to confusion table



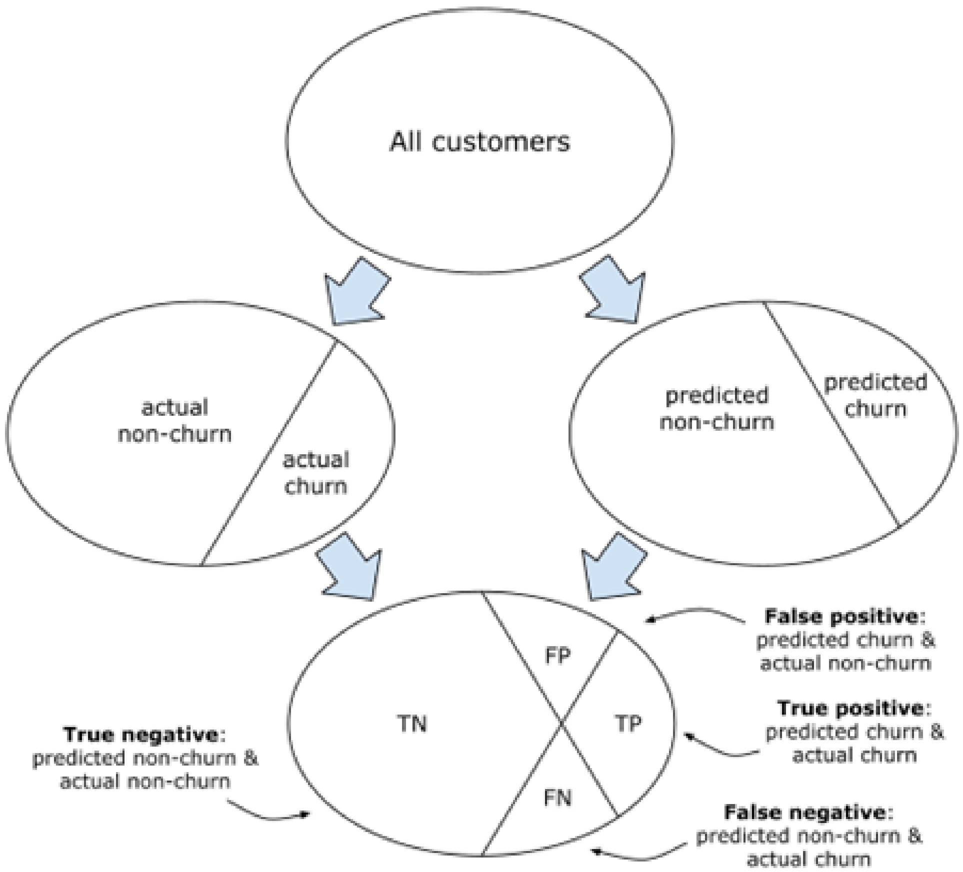
Introduction to confusion table

		Predictions	
		False ("no churn")	True ("churn")
Actual	False ("no churn")	TN	FP
	True ("churn")	FN	TP

Introduction to confusion table

		Predictions: False ("no churn")	Predictions: True ("churn")
Actual	False ("no churn")	1202	172
	True ("churn")	197	289

Calculating the confusion table with NumPy

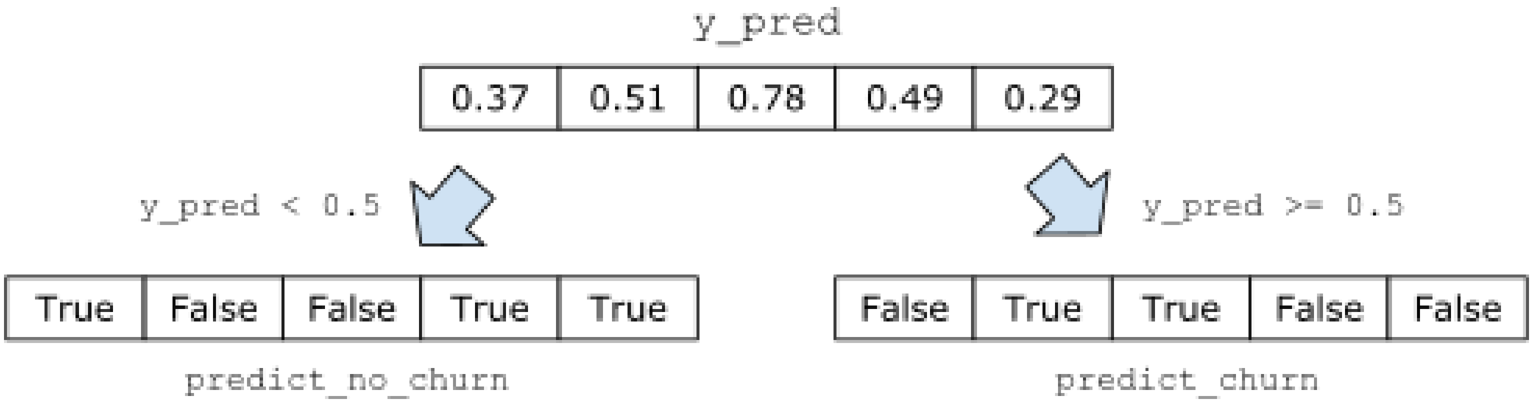


Calculating the confusion table with NumPy

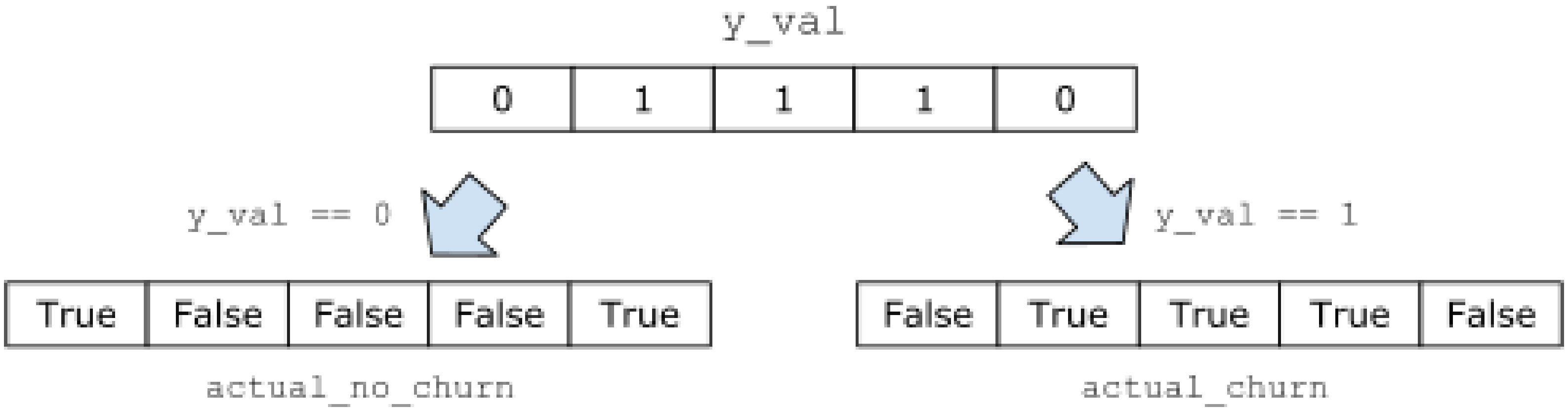
- Translating these steps to NumPy is straightforward:

```
t = 0.5
predict_churn = (y_pred >= t)
predict_no_churn = (y_pred < t)

actual_churn = (y_val == 1)
actual_no_churn = (y_val == 0)

true_positive = (predict_churn & actual_churn).sum()
false_positive = (predict_churn & actual_no_churn).sum()

false_negative = (predict_no_churn & actual_churn).sum()
true_negative = (predict_no_churn & actual_no_churn).sum()
```

Calculating the confusion table with NumPy



Calculating the confusion table with NumPy

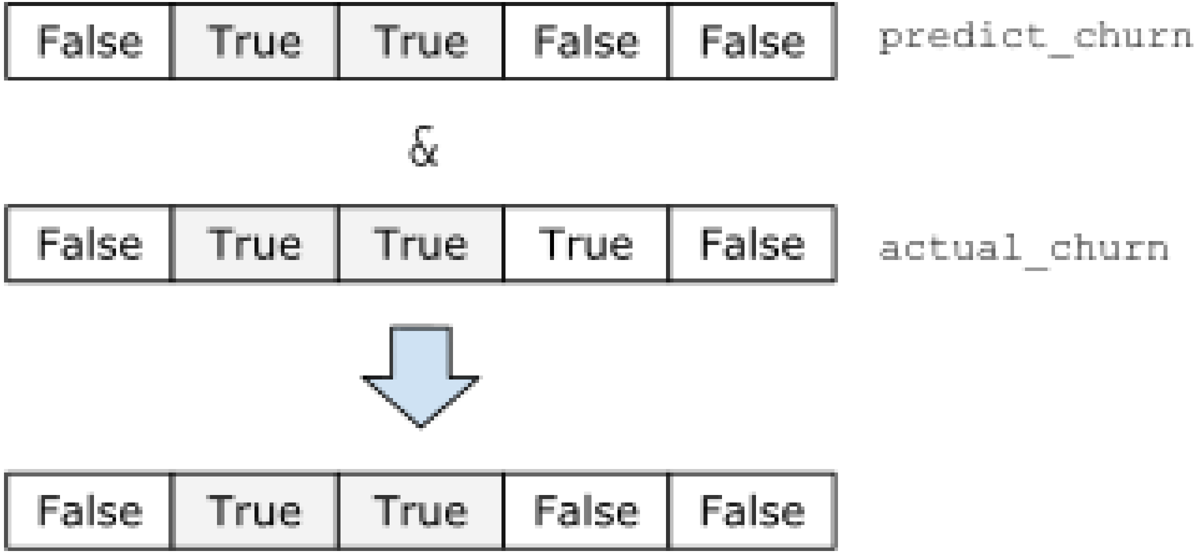


Calculating the confusion table with NumPy

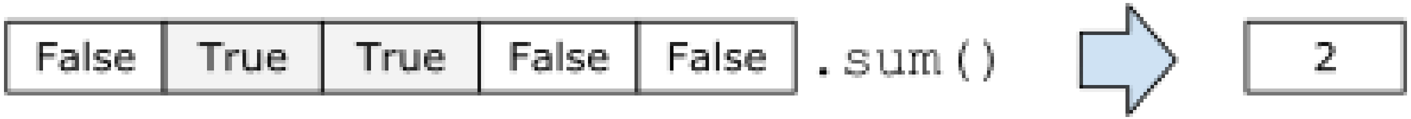
- To calculate the number of true positive outcomes in C, we use the logical “and” operator of NumPy (&) and the sum method:

```
true_positive = (predict_churn &
actual_churn).sum()
```

Calculating the confusion table with NumPy



Calculating the confusion table with NumPy



- As a result, we have the number of true positive cases.

- The other values are computed similarly in lines D, E, and F.

Calculating the confusion table with NumPy

- Now we just need to put all these values together in a NumPy array:

```
confusion_table = np.array(
 [[true_negative, false_positive],
 [false_negative, true_positive]])
```

- When we print it, we get the following numbers:

```
[[1202, 172],
 [ 197, 289]]
```

Calculating the confusion table with NumPy

- The absolute numbers sometimes may be difficult to understand, so we can turn them into fractions by dividing each value by the total number of items:

```
confusion_table / confusion_table.sum()
```

- This prints the following numbers:

```
[[0.646, 0.092],
[0.105, 0.155]]
```

Full model with all features

		Predicted	
		False	True
Actual	False	1202 (65%)	172 (9%)
	True	197 (11%)	289 (15%)

Small model with three features

		Predicted	
		False	True
Actual	False	1189 (63%)	185 (10%)
	True	248 (12%)	238 (13%)

"Complete Exercise "

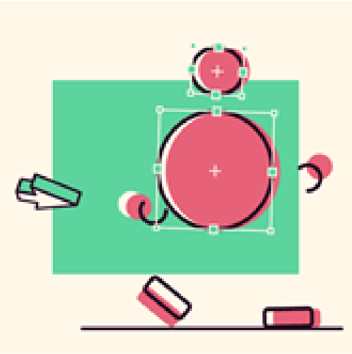
Precision and recall

- In our case it's the number of customers who actually churned (TP), out of all the customers we thought would churn (TP + FP):

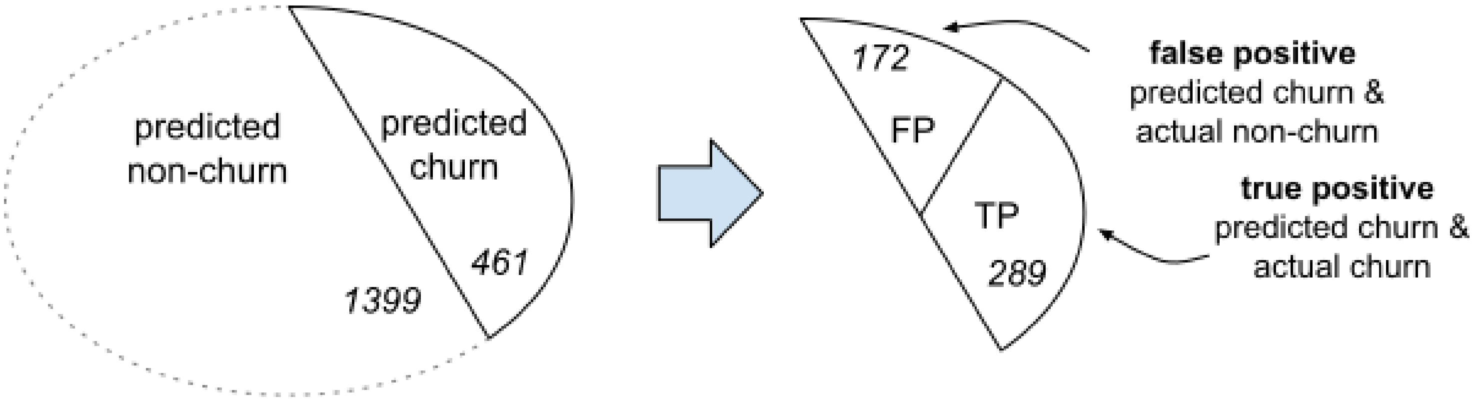
$$P = TP / (TP + FP)$$

- For our model the precision is 62%:

$$P = 289 / (289 + 172) = 172 / 461 = 0.62$$



Precision and recall

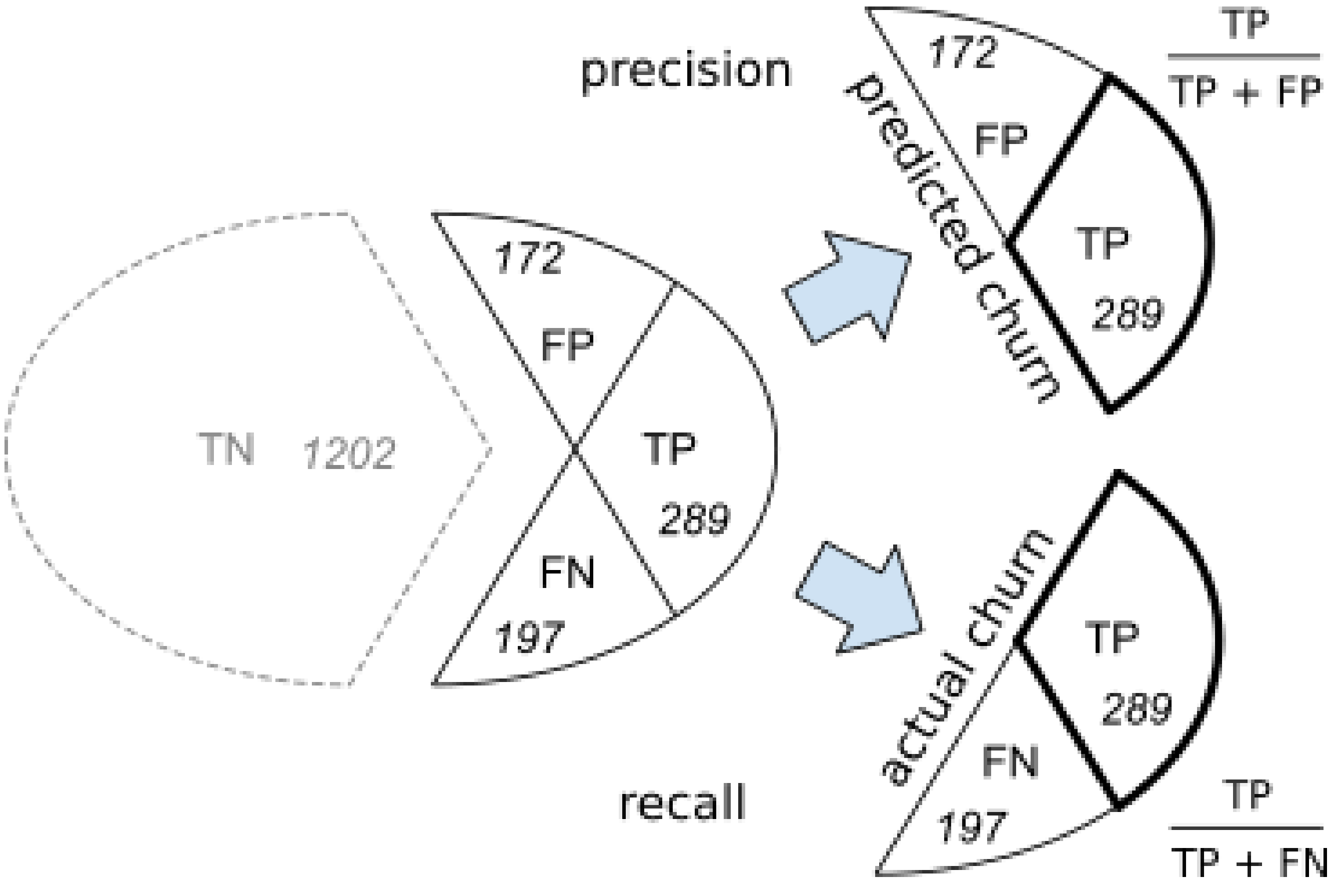


$$\text{precision} = \frac{\text{correct predictions}}{\text{predicted churn}} = \frac{TP}{TP + FP} = \frac{289}{461} = 62\%$$

Precision and recall

- Now we can check the accuracy of this baseline prediction using the same code as previously:

```
accuracy_score(baseline, y_val)
```


precision
TP
TP + FP
172
FP
predicted churn
TP
289
TN 1202
172
FP
TP
289
FN
197
actual churn
TP
289
FN
197
recall
TP
TP + FN

"Complete Exercises "

ROC curve and AUC score

- ROC stands for “receiver operating characteristic,” and it was initially designed for evaluating the strength of radar detectors during World War II.

- It was used to assess how well a detector could separate two signals: whether an airplane was there or not.

True positive rate and false positive rate

The ROC curve is based on two quantities, FPR and TPR:

- **False positive rate (FPR)** – The fraction of false positives among all negative examples
- **True positive rate (TPR)** – The fraction of true positives among all positive examples

True positive rate and false positive rate

		Predictions	
		False ("no churn")	True ("churn")
Actual	False ("no churn")	TN	**FP**
	True ("churn")	FN	**TP**

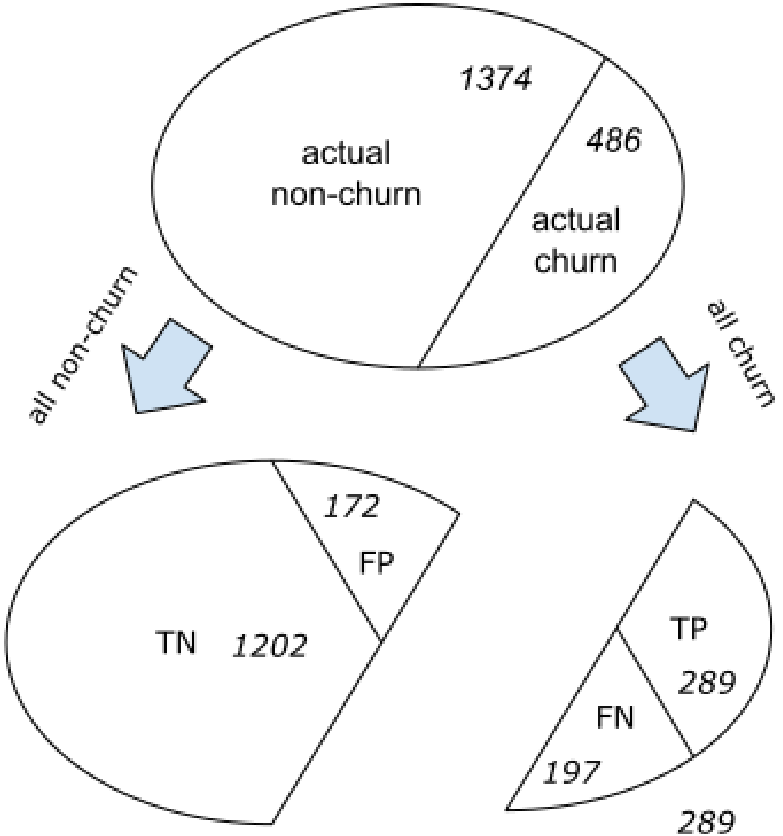
$$FPR = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{TP + FN}$$

FPR

$$\frac{\text{false positive}}{\text{actual non-churn}}$$

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



$$\frac{172}{1374} = 12.5\%$$

$$\frac{289}{486} = 59\%$$

TPR

$$\frac{\text{true positive}}{\text{actual churn}}$$

$$\frac{TP}{TP + FN}$$

Evaluating a model at multiple thresholds

- For that, we first iterate over different threshold values and compute the values of the confusion table for each.

```
scores = []

thresholds = np.linspace(0, 1, 101)

for t in thresholds:
    tp = ((y_pred >= t) & (y_val == 1)).sum()
    fp = ((y_pred >= t) & (y_val == 0)).sum()
    fn = ((y_pred < t) & (y_val == 1)).sum()
    tn = ((y_pred < t) & (y_val == 0)).sum()
    scores.append((t, tp, fp, fn, tn))
```

Evaluating a model at multiple thresholds

- It's not easy to deal with a list of tuples, so let's convert it to a Pandas dataframe:

```
df_scores = pd.DataFrame(scores)
df_scores.columns = ['threshold', 'tp', 'fp',
'fn', 'tn']
```

```
df_scores[::10]
```

	threshold	tp	fp	fn	tn
0	0.0	486	1374	0	0
10	0.1	458	726	28	648
20	0.2	421	512	65	862
30	0.3	380	350	106	1024
40	0.4	337	257	149	1117
50	0.5	289	172	197	1202
60	0.6	200	105	286	1269
70	0.7	99	34	387	1340
80	0.8	7	1	479	1373
90	0.9	0	0	486	1374
100	1.0	0	0	486	1374

Evaluating a model at multiple thresholds

- Now we can compute the TPR and FPR scores.
- Because the data is now in a dataframe, we can do it for all the values at once:

```
df_scores['tpr'] = df_scores.tp / (df_scores.tp + 
df_scores.fn)
df_scores['fpr'] = df_scores.fp / (df_scores.fp + 
df_scores.tn)
```

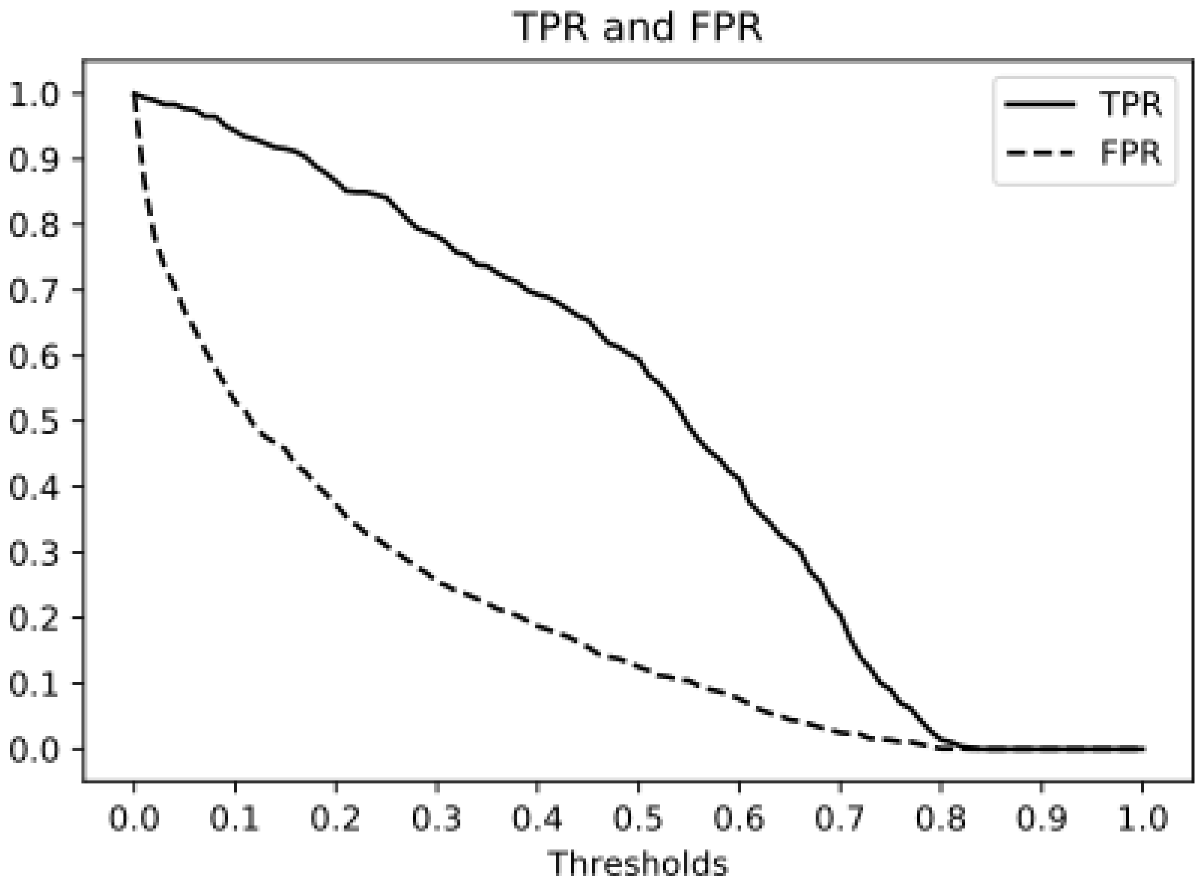
```
df_scores[::10]
```

	threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	458	726	28	648	0.942387	0.528384
20	0.2	421	512	65	862	0.866255	0.372635
30	0.3	380	350	106	1024	0.781893	0.254731
40	0.4	337	257	149	1117	0.693416	0.187045
50	0.5	289	172	197	1202	0.594650	0.125182
60	0.6	200	105	286	1269	0.411523	0.076419
70	0.7	99	34	387	1340	0.203704	0.024745
80	0.8	7	1	479	1373	0.014403	0.000728
90	0.9	0	0	486	1374	0.000000	0.000000
100	1.0	0	0	486	1374	0.000000	0.000000

Evaluating a model at multiple thresholds

- Let’s plot them (next figure):

```
plt.plot(df_scores.threshold, df_scores.tpr,
label='TPR')
plt.plot(df_scores.threshold, df_scores.fpr,
label='FPR')
plt.legend()
```


TPR and FPR
TPR
FPR
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Thresholds

Random baseline model

- A random model outputs a random score between 0 and 1 regardless of the input.
- It's easy to implement: we simply generate an array with uniform random numbers:

```
np.random.seed(1)
y_rand = np.random.uniform(0, 1, size=len(y_val))
```

```python
def tpr_fpr_dataframe(y_val, y_pred):
    scores = []

    thresholds = np.linspace(0, 1, 101)

    for t in thresholds:
        tp = ((y_pred >= t) & (y_val == 1)).sum()
        fp = ((y_pred >= t) & (y_val == 0)).sum()
        fn = ((y_pred < t) & (y_val == 1)).sum()
        tn = ((y_pred < t) & (y_val == 0)).sum()
        scores.append((t, tp, fp, fn, tn))

    df_scores = pd.DataFrame(scores)
    df_scores.columns = ['threshold', 'tp', 'fp', 'fn', 'tn']

    df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
    df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)

    return df_scores
```

Random baseline model

- Now let's use this function to calculate the TPR and FPR for the random model:
df_rand = tpr_fpr_dataframe(y_val, y_rand)
- This creates a dataframe with TPR and FPR values at different thresholds

```
np.random.seed(1)
y_rand = np.random.uniform(0, 1, size=len(y_val))
df_rand = tpr_fpr_dataframe(y_val, y_rand)
df_rand[::10]
```

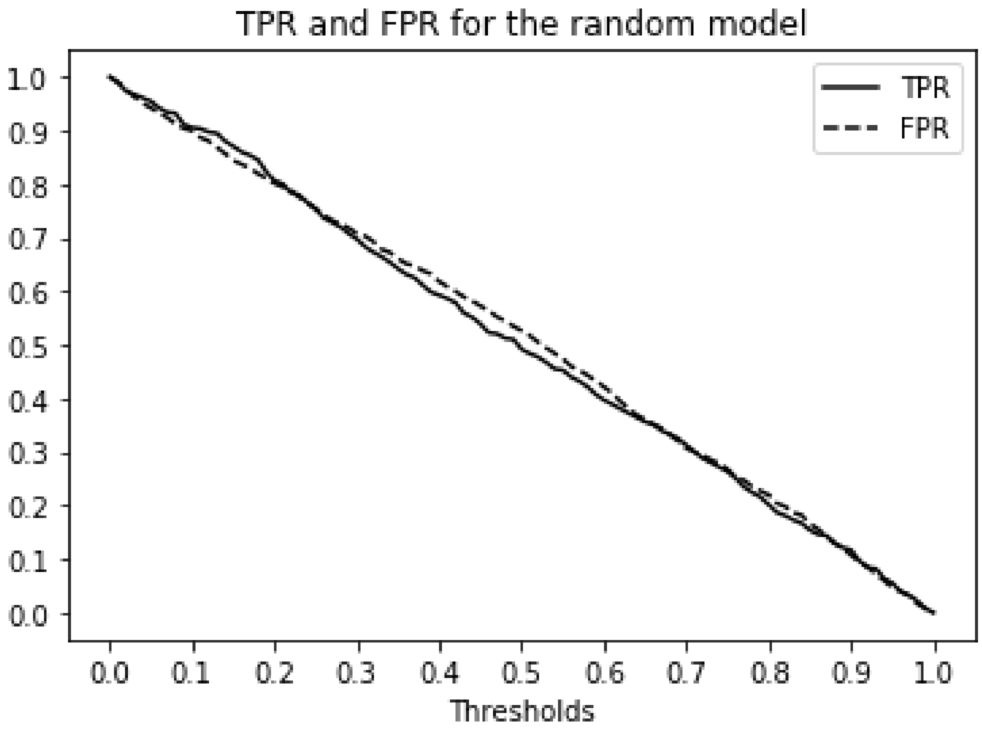
	threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	440	1236	46	138	0.905350	0.899563
20	0.2	392	1101	94	273	0.806584	0.801310
30	0.3	339	972	147	402	0.697531	0.707424
40	0.4	288	849	198	525	0.592593	0.617904
50	0.5	239	723	247	651	0.491770	0.526201
60	0.6	193	579	293	795	0.397119	0.421397
70	0.7	152	422	334	952	0.312757	0.307132
80	0.8	98	302	388	1072	0.201646	0.219796
90	0.9	57	147	429	1227	0.117284	0.106987
100	1.0	0	0	486	1374	0.000000	0.000000

Random baseline model

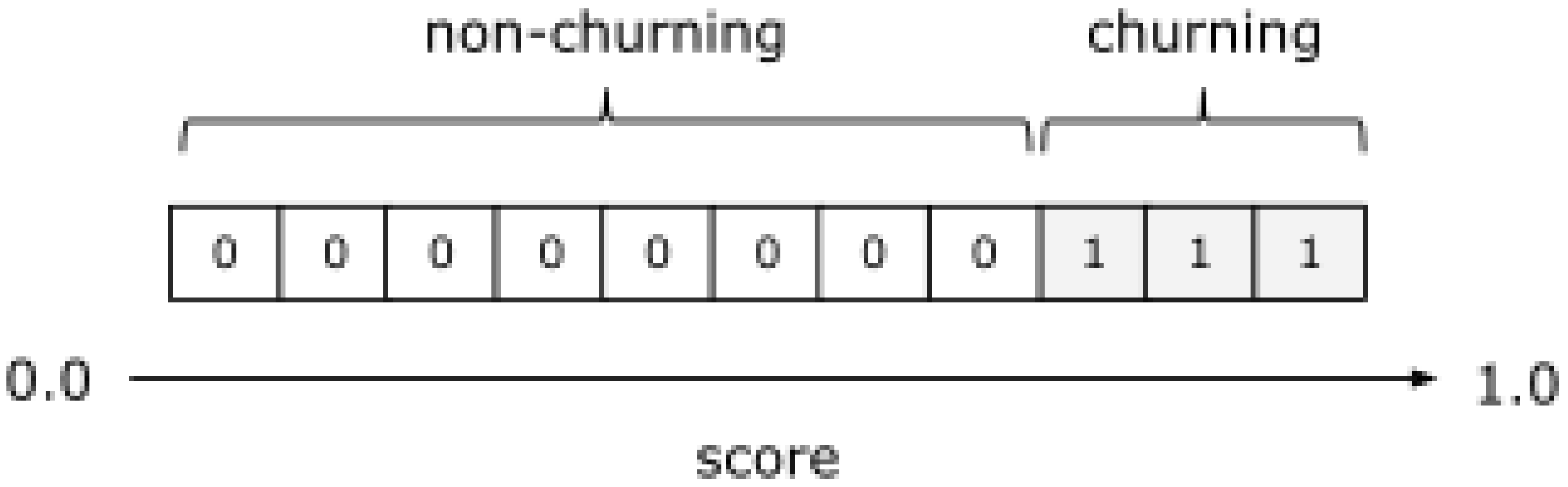
- Let’s plot them:

```
plt.plot(df_rand.threshold, df_rand.tpr,
label='TPR')
plt.plot(df_rand.threshold, df_rand.fpr,
label='FPR')
plt.legend()
```

Random baseline model



The ideal model



The ideal model

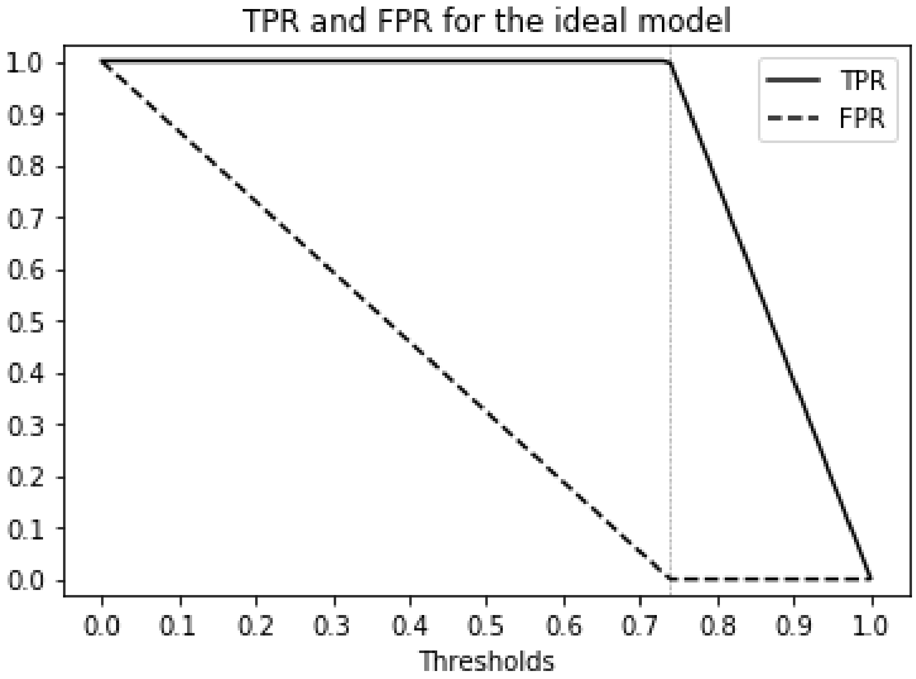
- Let’s do it:

```
num_neg = (y_val == 0).sum()
num_pos = (y_val == 1).sum()

y_ideal = np.repeat([0, 1], [num_neg, num_pos])
y_pred_ideal = np.linspace(0, 1, num_neg +
num_pos)

df_ideal = tpr_fpr_dataframe(y_ideal,
y_pred_ideal)
```

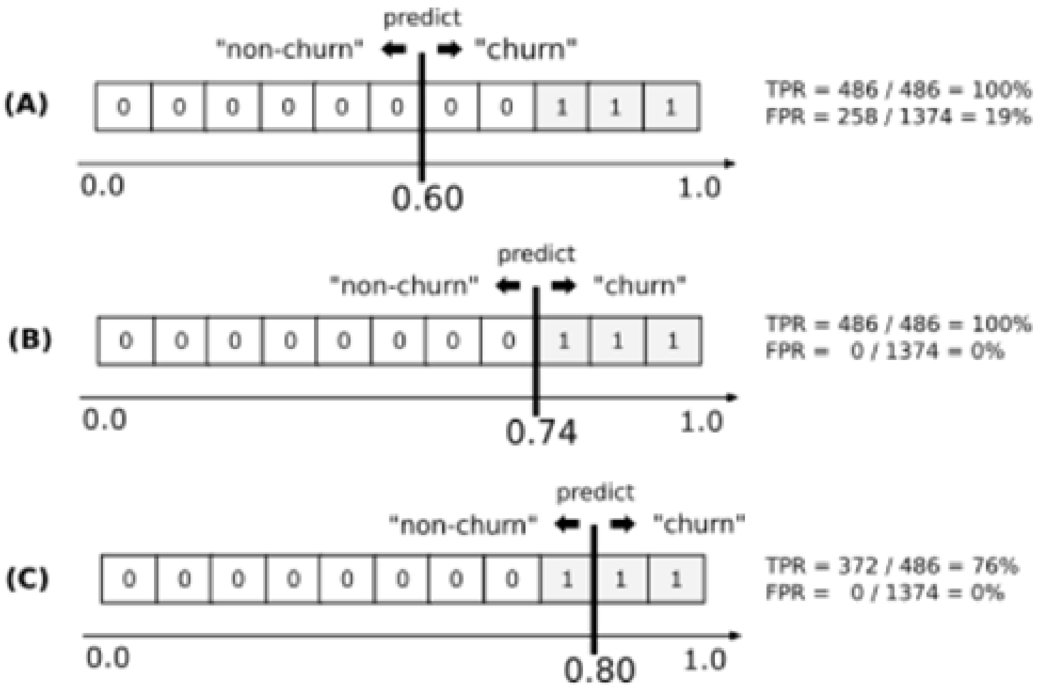
	threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	486	1188	0	186	1.000000	0.864629
20	0.2	486	1002	0	372	1.000000	0.729258
30	0.3	486	816	0	558	1.000000	0.593886
40	0.4	486	630	0	744	1.000000	0.458515
50	0.5	486	444	0	930	1.000000	0.323144
60	0.6	486	258	0	1116	1.000000	0.187773
70	0.7	486	72	0	1302	1.000000	0.052402
80	0.8	372	0	114	1374	0.765432	0.000000
90	0.9	186	0	300	1374	0.382716	0.000000
100	1.0	1	0	485	1374	0.002058	0.000000

- Now we can plot it (figure):

```
plt.plot(df_ideal.threshold, df_ideal.tpr, label='TPR')
plt.plot(df_ideal.threshold, df_ideal.fpr, label='FPR')
plt.legend()
```



The ideal model



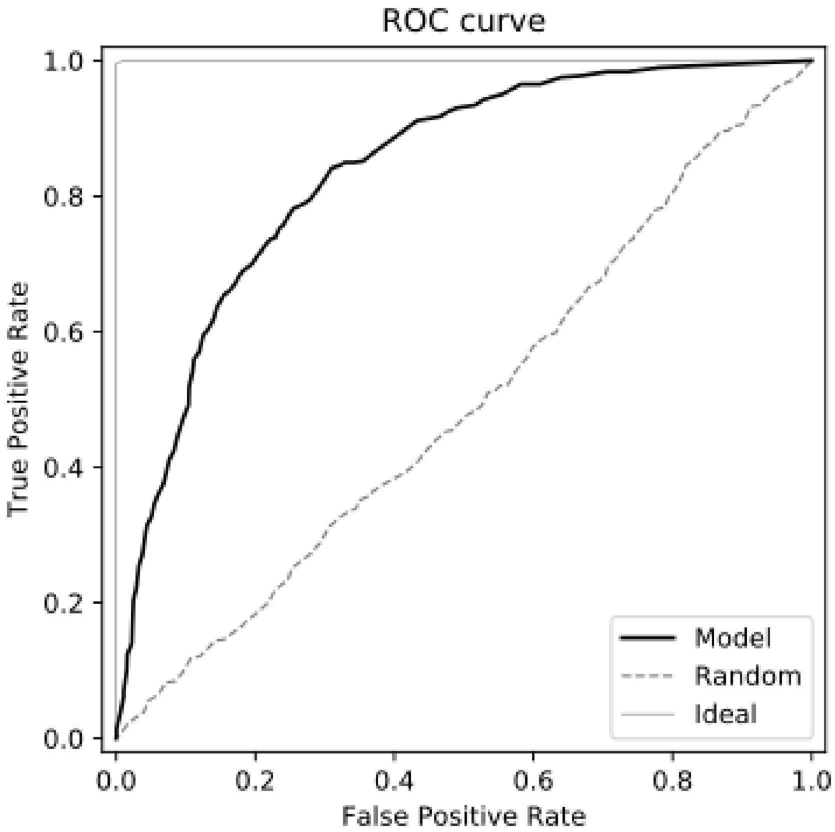
"Complete Exercise "

ROC Curve

```
plt.figure(figsize=(5, 5))

plt.plot(df_scores.fpr, df_scores.tpr,
label='Model')
plt.plot(df_rand.fpr, df_rand.tpr,
label='Random')
plt.plot(df_ideal.fpr, df_ideal.tpr,
label='Ideal')

plt.legend()
```

ROC Curve

- As a result, we get a ROC curve



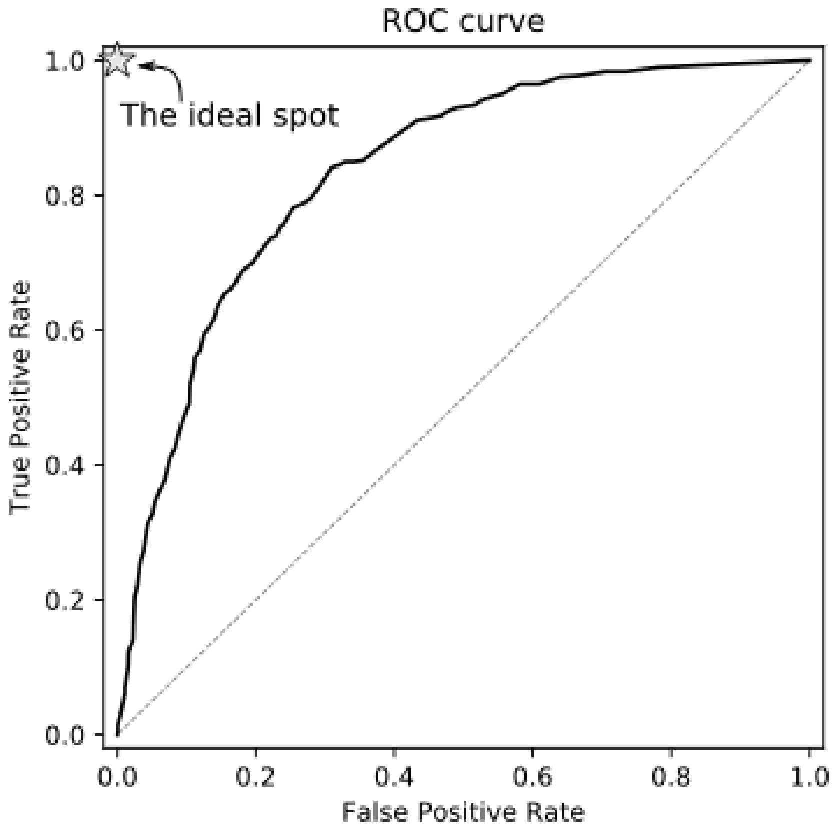
ROC Curve

- With this information, we can reduce the code for plotting the curve to the following:

```
plt.figure(figsize=(5, 5))
plt.plot(df_scores.fpr, df_scores.tpr)
plt.plot([0, 1], [0, 1])
```

ROC Curve

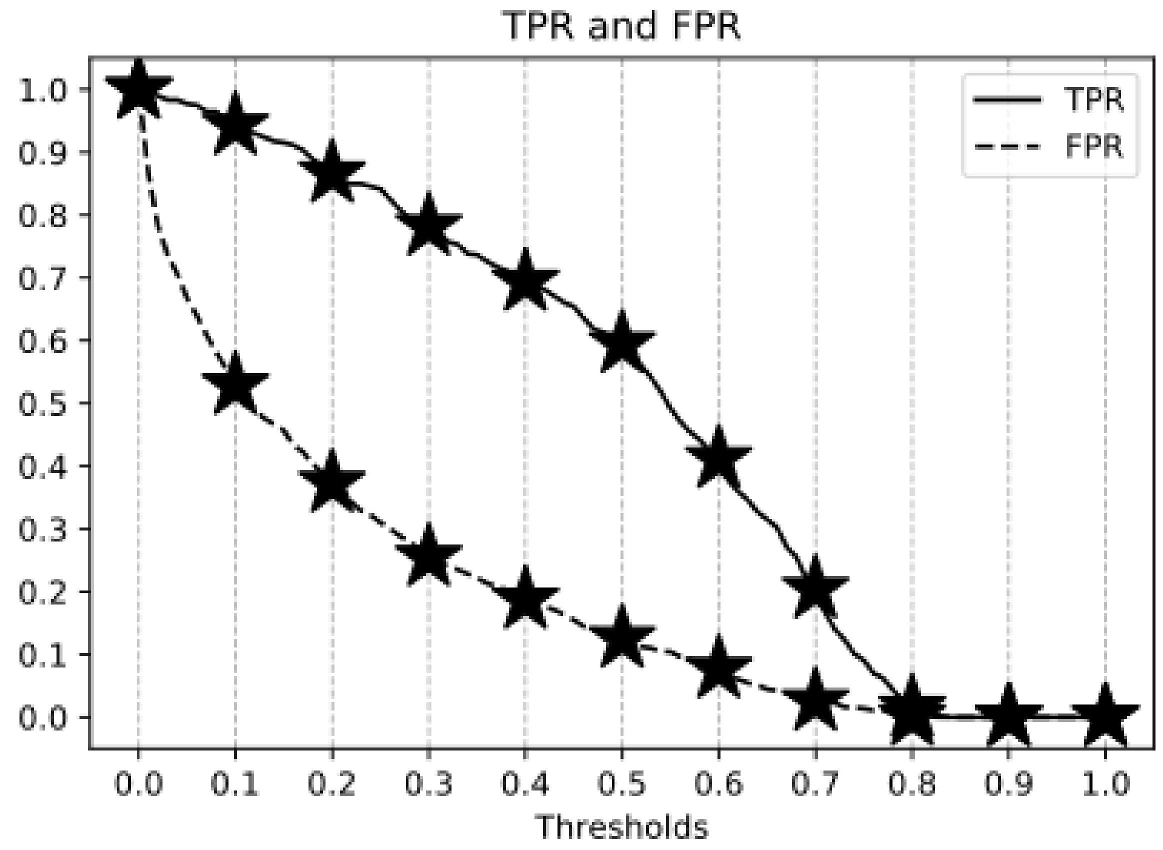
- Produces the result in figure



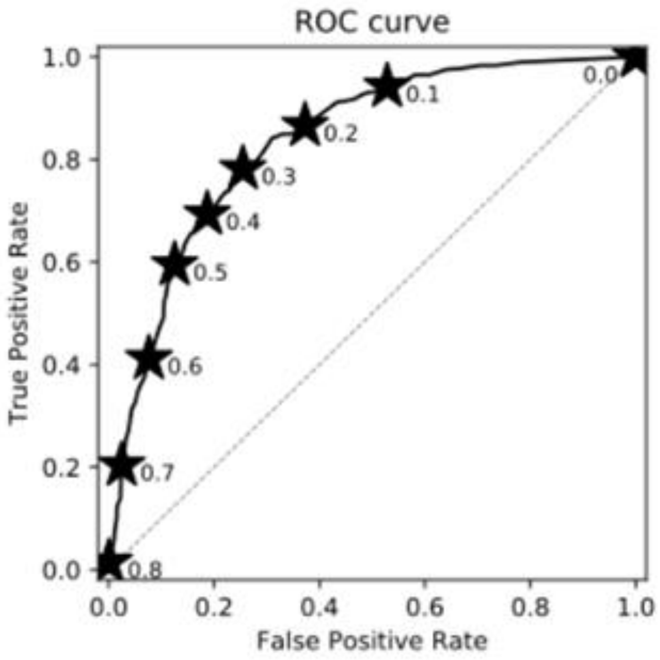
ROC Curve

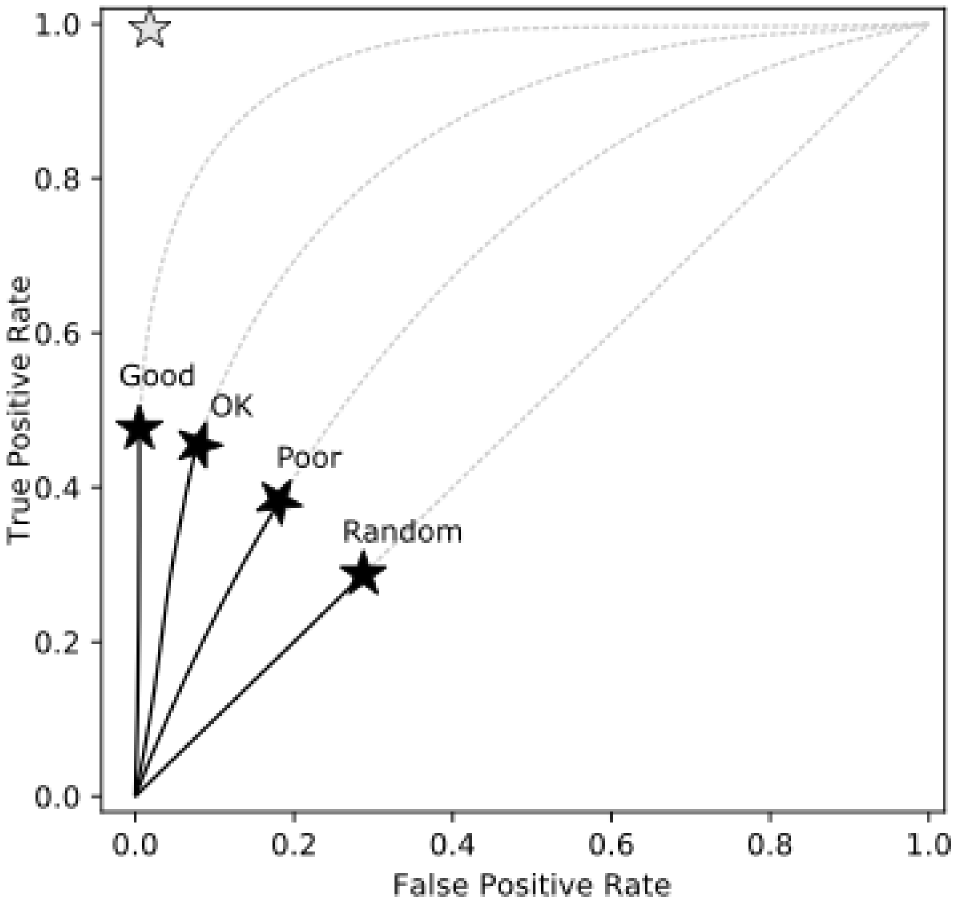
- We simply can use the roc_curve function from the metrics package of Scikit-Learn:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_val, y_pred)

plt.figure(figsize=(5, 5))
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1])
```


TPR and FPR
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
TPR
FPR
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Thresholds

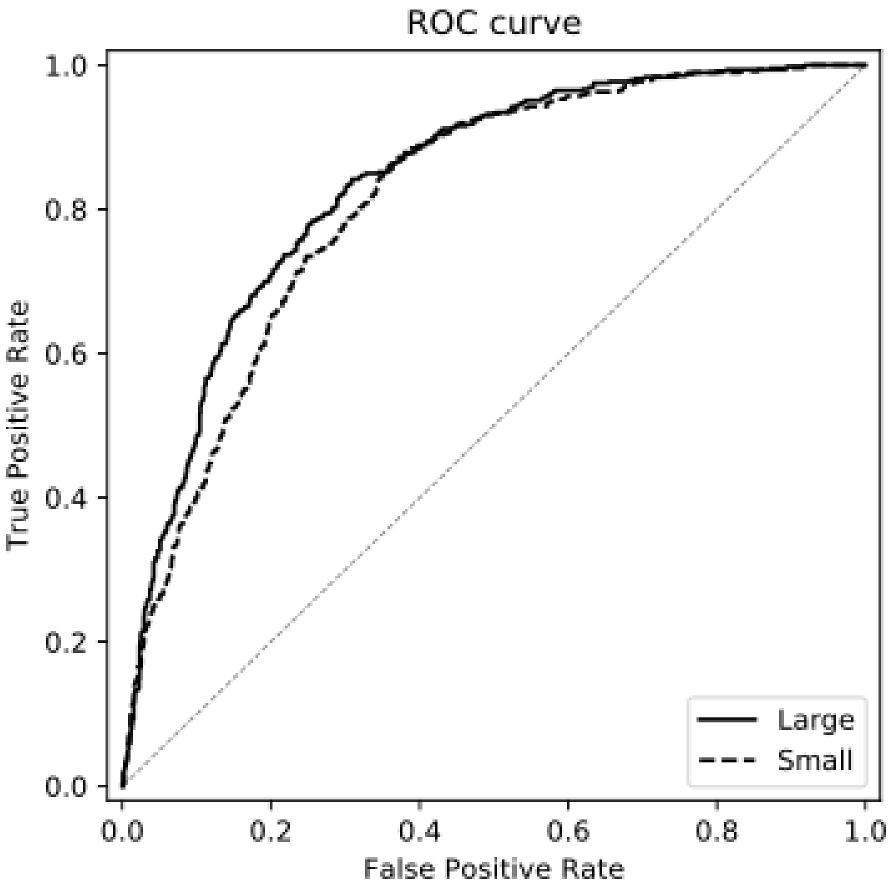
	threshold	fpr	tpr
100	1.0	0.000000	0.000000
90	0.9	0.000000	0.000000
80	0.8	0.000728	0.014403
70	0.7	0.024745	0.203704
60	0.6	0.076419	0.411523
50	0.5	0.125182	0.594650
40	0.4	0.187045	0.693416
30	0.3	0.254731	0.781893
20	0.2	0.372635	0.866255
10	0.1	0.528384	0.942387
0	0.0	1.000000	1.000000




True Positive Rate
False Positive Rate
Good
OK
Poor
Random
0.0
0.2
0.4
0.6
0.8
1.0

ROC Curve

```
fpr_large, tpr_large, _ = roc_curve(y_val, y_pred)
fpr_small, tpr_small, _ = roc_curve(y_val,
y_pred_small)

plt.figure(figsize=(5, 5))

plt.plot(fpr_large, tpr_large, color='black',
label='Large')
plt.plot(fpr_small, tpr_small, color='black',
label='Small')
plt.plot([0, 1], [0, 1])
plt.legend()
```


ROC curve
1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
Large
Small

Area under the ROC curve (AUC)

- To calculate the AUC for our models we can use auc, a function from the metrics package of Scikit-Learn:

```
from sklearn.metrics import auc
auc(df_scores.fpr, df_scores.tpr)
```

Area under the ROC curve (AUC)

- For the large model, the result is 0.84; for the small model, it's 0.81 (figure).
- Churn prediction is a complex problem, so an AUC of 80% is quite good.

```
from sklearn.metrics import auc
auc(df_scores.fpr, df_scores.tpr)
```

```
0.8359001084215382
```

```
auc(df_scores_small.fpr, df_scores_small.tpr)
```

```
0.8125475467380692
```

Area under the ROC curve (AUC)

- If all we need is the AUC, we don't need to compute the ROC curve first.
- We can take a shortcut and use a function from Scikit-Learn that takes care of everything and simply returns the AUC of our model:

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_val, y_pred)
```

Area under the ROC curve (AUC)

- We get approximately the same results as previously

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_val, y_pred)
```

```
0.8363396349608545
```

```
roc_auc_score(y_val, y_pred_small)
```

```
0.8129354083179088
```

Area under the ROC curve (AUC)

```
neg = y_pred[y_val == 0]
pos = y_pred[y_val == 1]

np.random.seed(1)
neg_choice = np.random.randint(low=0,
high=len(neg), size=10000)
pos_choice = np.random.randint(low=0,
high=len(pos), size=10000)
(pos[pos_choice] > neg[neg_choice]).mean()
```

Parameter tuning

- It tells us how well the model will perform on these specific data points.
- However, it doesn’t necessarily mean it will perform equally well on other data points. So how do we check if the model indeed works well in a consistent and predictable manner?

K-fold cross-validation

K-FOLD CROSS-VALIDATION

```
def train(df, y):
    cat = df[categorical + numerical].to_dict(orient='rows')

    dv = DictVectorizer(sparse=False)
    dv.fit(cat)

    X = dv.transform(cat)

    model = LogisticRegression(solver='liblinear')
    model.fit(X, y)

    return dv, model
```

K-FOLD CROSS-VALIDATION

- We apply the vectorizer to the dataframe, get a matrix and finally apply the model to the matrix to get predictions:

```
def predict(df, dv, model):
    cat = df[categorical + 
numerical].to_dict(orient='rows')

    X = dv.transform(cat)
    y_pred = model.predict_proba(X)[:, 1]

    return y_pred
```

```python
from sklearn.model_selection import KFold

kfold = KFold(n_splits=10, shuffle=True, random_state=1)

aucs = []

for train_idx, val_idx in kfold.split(df_train_full):
    df_train = df_train_full.iloc[train_idx]
    df_val = df_train_full.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    aucs.append(auc)
```

K-fold cross-validation

- We used K-fold cross-validation with K=10.
- Thus, when we run it, at the end we get 10 different numbers – 10 AUC scores evaluated on 10 different validation folds:

0.849, 0.841, 0.859, 0.833, 0.824, 0.841, 0.844, 0.822, 0.845, 0.861

K-FOLD CROSS-VALIDATION

- It’s not a single number anymore, and we can think of it as a distribution of AUC scores for our model.
- So we can get some statistics from this distribution, such as the mean and standard deviation:

```
print('auc = %0.3f ± %0.3f' % (np.mean(aucs),
np.std(aucs)))
```

Finding best parameters

- We first adjust the train function to take in an additional parameter:

```
def train(df, y, C):
    cat = df[categorical + numerical].to_dict(orient='rows')

    dv = DictVectorizer(sparse=False)
    dv.fit(cat)

    X = dv.transform(cat)

    model = LogisticRegression(solver='liblinear', C=C)
    model.fit(X, y)

    return dv, model
```

```python
nfolds = 5
kfold = KFold(n_splits=nfolds, shuffle=True, random_state=1)

for C in [0.001, 0.01, 0.1, 0.5, 1, 10]:
    aucs = []

    for train_idx, val_idx in kfold.split(df_train_full):
        df_train = df_train_full.iloc[train_idx]
        df_val = df_train_full.iloc[val_idx]

        y_train = df_train.churn.values
        y_val = df_val.churn.values

        dv, model = train(df_train, y_train, C=C)
        y_pred = predict(df_val, dv, model)

        auc = roc_auc_score(y_val, y_pred)
        aucs.append(auc)

    print('C=%s, auc = %0.3f ± %0.3f' % (C, np.mean(aucs), np.std(aucs)))
```

Finding best parameters

- When we run it, it prints:

```
C=0.001, auc = 0.825 ± 0.013
C=0.01, auc = 0.839 ± 0.009
C=0.1, auc = 0.841 ± 0.008
C=0.5, auc = 0.841 ± 0.007
C=1, auc = 0.841 ± 0.007
C=10, auc = 0.841 ± 0.007
```

Finding best parameters

- Let's use our train and predict functions for that:

```
y_train = df_train_full.churn.values
y_test = df_test.churn.values

dv, model = train(df_train_full, y_train, C=0.5)
y_pred = predict(df_test, dv, model)

auc = roc_auc_score(y_test, y_pred)
print('auc = %.3f' % auc)
```

"Complete Exercises & Lab"

Summary

- A metric is a single number that can be used for evaluating the performance of a machine learning model.
- Once we choose a metric, we can use it to compare multiple machine learning models with each other and select the best one.
- Accuracy is the simplest binary classification metric: it tells us the percentage of correctly classified observations in the validation set.