PERFORMANCE
IN
CLASSIFICATION MODEL

Performance Evaluation

- Evaluation of the performance of your classifiers is extremely important.
- The SKlearn kit provides very good modules to address this issue.
- In particular, evaluation often involves measuring accuracy, precision, recall, and f-measure.
- The best way to understand these metrics is to think of a confusion matrix.
- Confusion matrices show how many elements from a class are correctly and incorrectly classified.

Cross-Validation

- The main drawback of Random subsampling is, it does not have control over the number of times each tuple is used for training and testing.

- Cross-validation is proposed to overcome this problem.

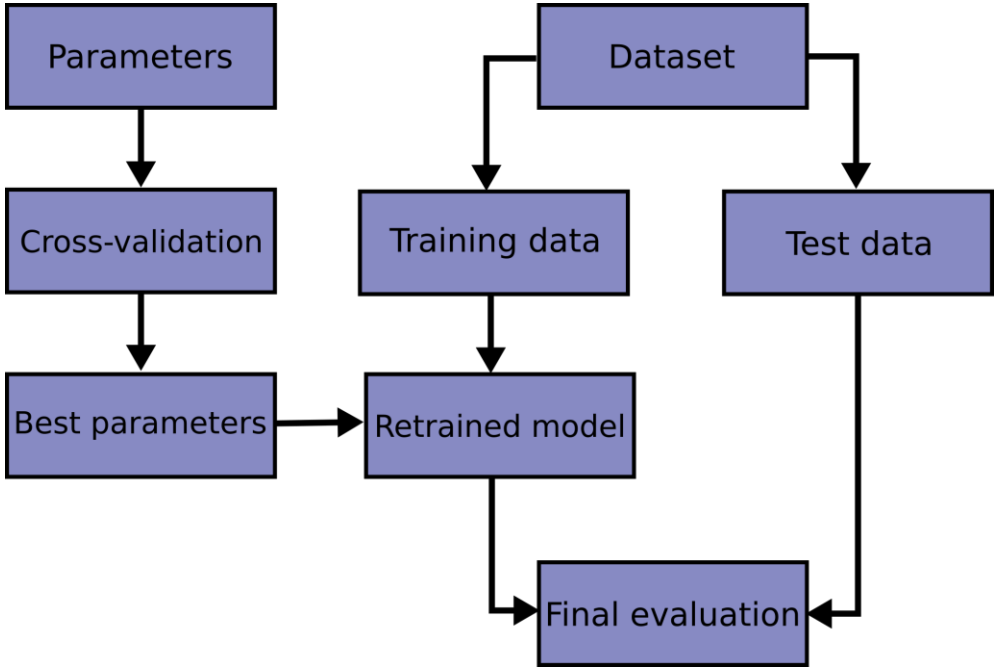
- Cross Validation is used to estimate test set prediction error rates associated with a given machine learning method to evaluate its performance, or to select the appropriate level of model flexibility.

Cross-Validation

Mempelajari parameter fungsi prediksi dan mengujinya pada data yang sama adalah kesalahan metodologis: model hanya akan mengulangi label sampel yang baru saja dilihatnya akan memiliki skor sempurna tetapi gagal memprediksi apa pun yang berguna pada- data yang tidak terlihat. Situasi ini disebut **overfitting**.

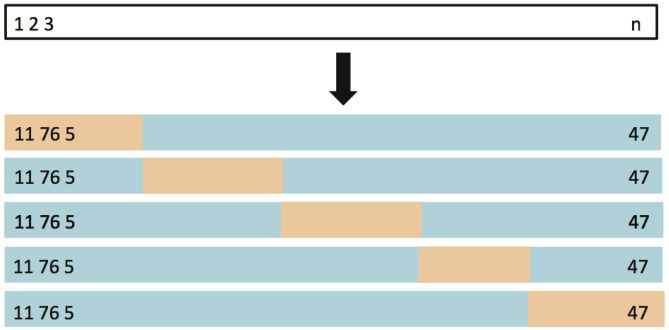
Untuk menghindarinya, saat melakukan eksperimen kita bisa menahan sebagian dari data yang tersedia sebagai **test set:** X_test, y_test.

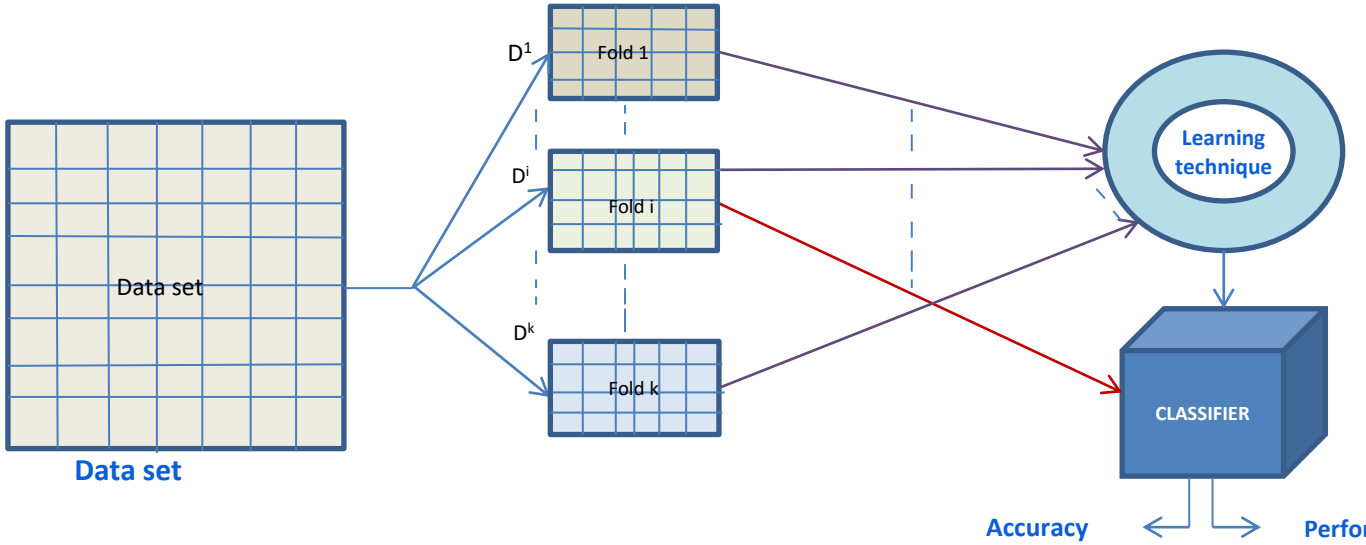
Diagram alur kerja validasi silang (cross validation) dalam training model sbb. Parameter terbaik dapat ditentukan dengan teknik pencarian grid.



k-fold Cross-Validation

- Dataset consisting of *N* tuples is divided randomly into *k* (usually, 5 or 10) equal.
- By averaging the *K* estimates of the test error, we get an estimated validation (test) error rate for new observations.
- The first fold is treated as a validation set, and the method is fit on the remaining *K* – 1 folds. The MSE is computed on the observations in the *held-out* fold. The process is repeated *K* times, taking out a different part each time.



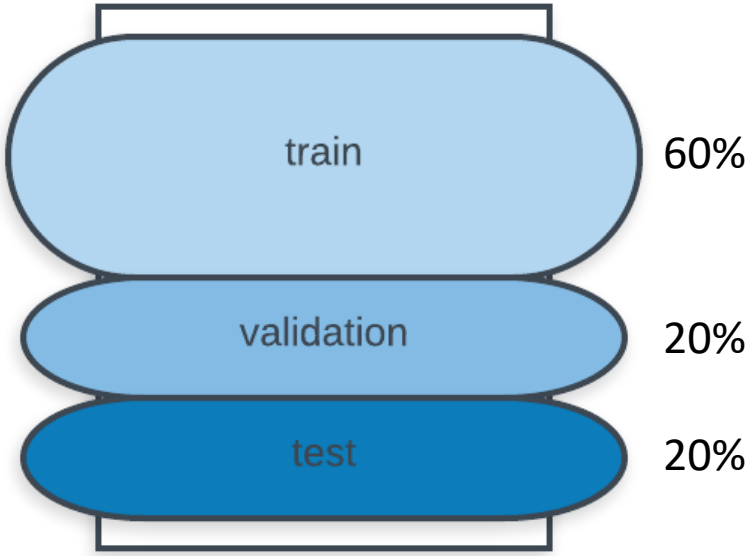


What are Performance Metrics?

Determine how well a model performs on the data

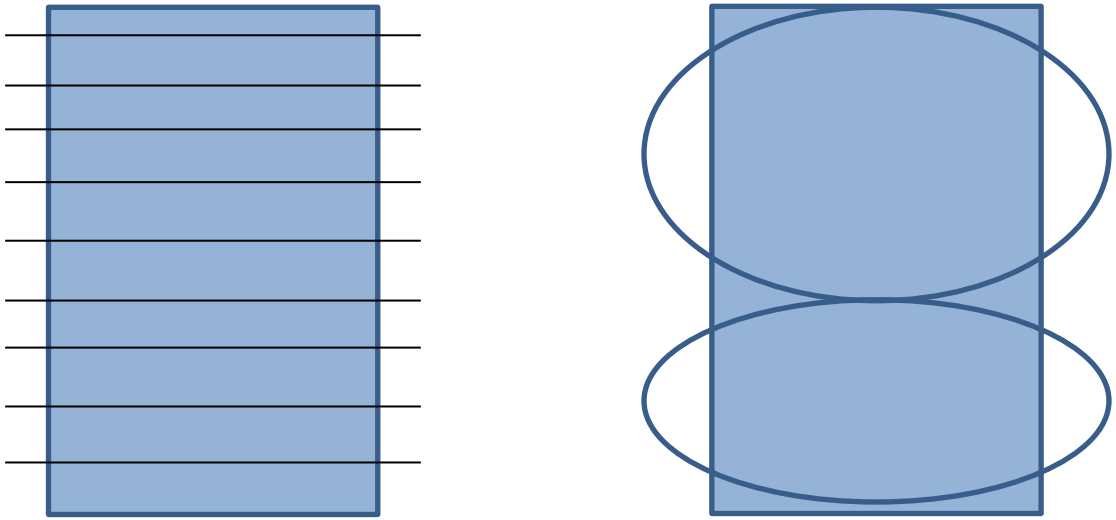
Data is divided

- train
- validation
- test



Splitting the Data

- 80%/20%
- 10-fold cross validation



Confusion Matrix

		PREDICTED VALUE: POSITIVE (True)	PREDICTED VALUE: NEGATIVE (False)
ACTUAL VALUE	POSITIVE (True)	TP	FN
	NEGATIVE (False)	FP	TN

- True Positive (TP) : Prediction = True, Actual = True
- False Positive (FP) : Prediction = True, Actual = False
- False Negative (FN) : Prediction = False, Actual = True
- True Negative (TN) : Prediction = False, Actual = False

True positives are data point classified as positive by the model that actually are positive (meaning they are correct).
False negatives are data points the model identifies as negative that actually are positive.
False positives are cases the model incorrectly labels as positive that are actually negative.

Confusion Matrix

Given a sample of 12 pictures, 8 of cats and 4 of dogs, where cats belong to class 1 and dogs belong to class 0.

actual = [1,1,1,1,1,1,1,1,0,0,0,0]

Assume that a classifier that distinguishes between cats and dogs is trained, and we take the 12 pictures and run them through the classifier, and the classifier makes 9 accurate predictions and misses 3: 2 cats wrongly predicted as dogs (first 2 predictions) and 1 dog wrongly predicted as a cat (last prediction).

prediction = [0,0,1,1,1,1,1,1,0,0,0,1]

		PREDICTED VALUE	
		CAT (1)	DOG (0)
ACTUAL VALUE	CAT (1)	6	2
	DOG (0)	1	3

Accuracy

Accuracy is how well the model performs

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

OR

$$\frac{CORRECT\_PREDICTION}{TOTAL}$$

This measure is dominated by the larger set (of positives or negatives) and favors trivial classifiers.

Example: if 5% of items are truly positive, then a classifier that always says “negative” is 95% accurate.

Precision

How often are we correct in our positive prediction?

Precision is defined as the number of true positives divided by the number of true positives plus the number of false positives.

Formula: (TP) / (TP + FP)

OR:

#CORRECT_POSITIVE_PREDICTIONS / #POSITIVE_SAMPLES

Recall / Sensitivity

How often did we wrongly classify something as not true (= false)?

Formula: (TP) / (TP + FN)

OR:

#CORRECT_POSITIVE_PREDICTIONS / #TRUE_TRUTH_VALUES

F1Score

Recall (r) and Precision (p) are two widely used metrics employed in analysis, where detection of one of the classes is considered more significant than the others.

$$F_1 = \frac{2r.p}{r+p} = \frac{2TP}{2TP + FP + FN}$$

Formula:

2 * ((Precision * Recall) / (Precision + Recall))

ROC Curve

This curve allows us to select the optimal model and discard sub-optimal ones.

Formula:

False Positive Rate (FPR) = X-Axis
True Positive Rate (TPR) = Y-Axis

- **FPR:** TP / (TP + FN)
- **TPR:** FP / (FP + TN)

ROC Curve (= Receiver Operating Characteristic) shows the performance.

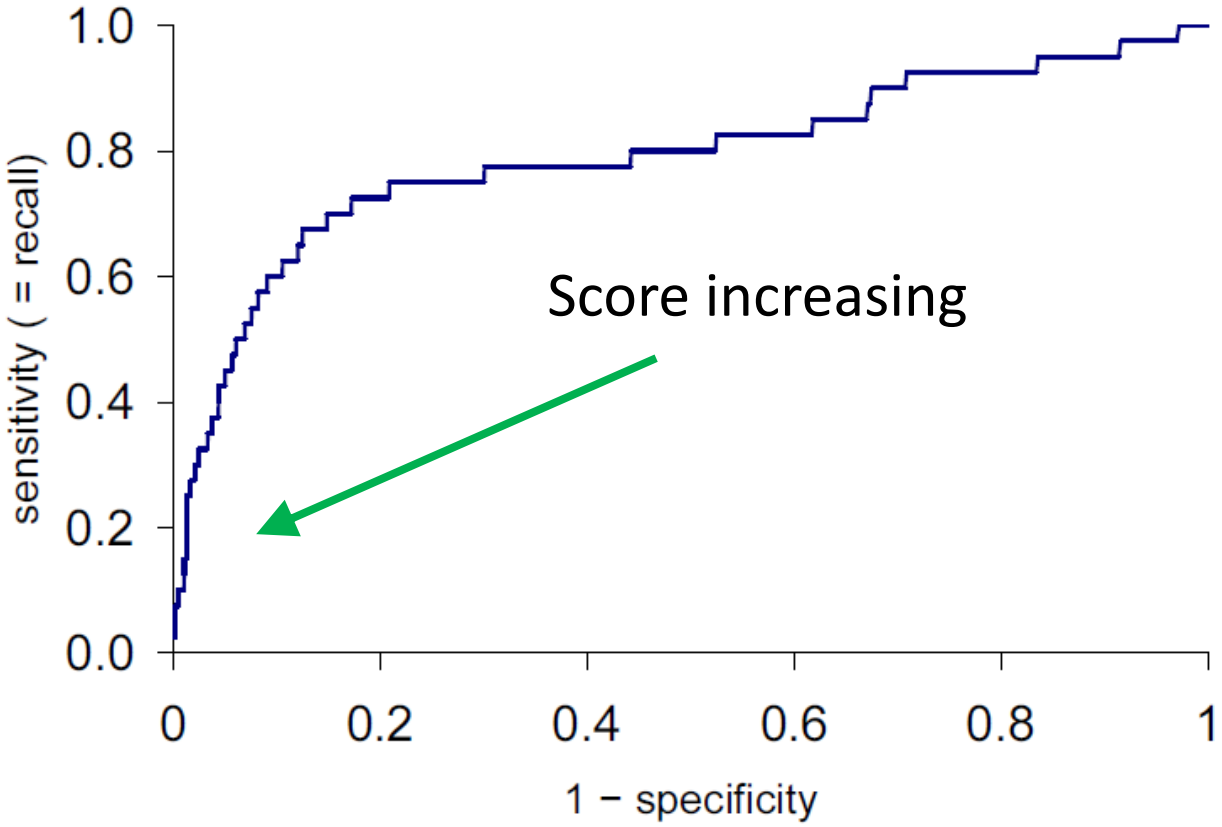
AUC (= Area Under the Curve) performance metric allows us to describe this as a value to measure the performance of classification models.

ROC plots

ROC is Receiver-Operating Characteristic. ROC plots

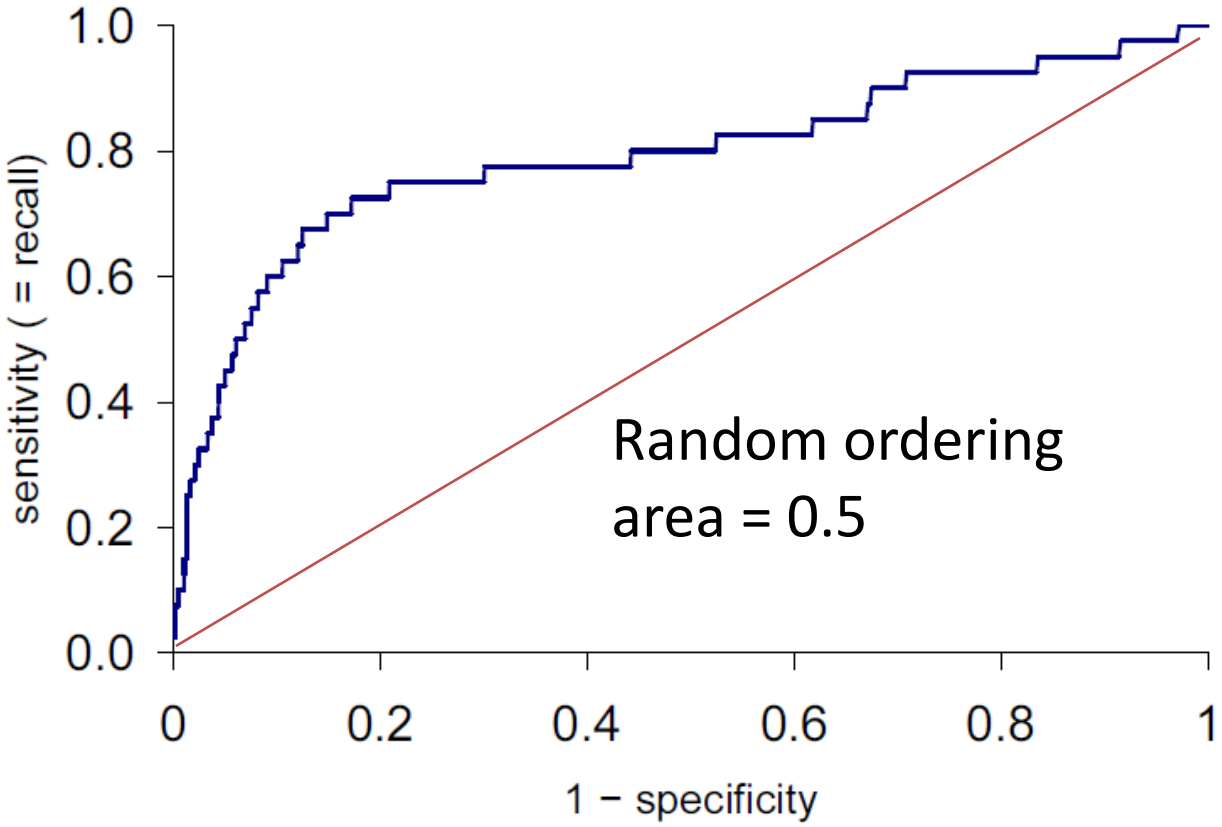
Y-axis: true positive rate = tp/(tp + fn), same as recall

X-axis: false positive rate = fp/(fp + tn) = 1 - specificity



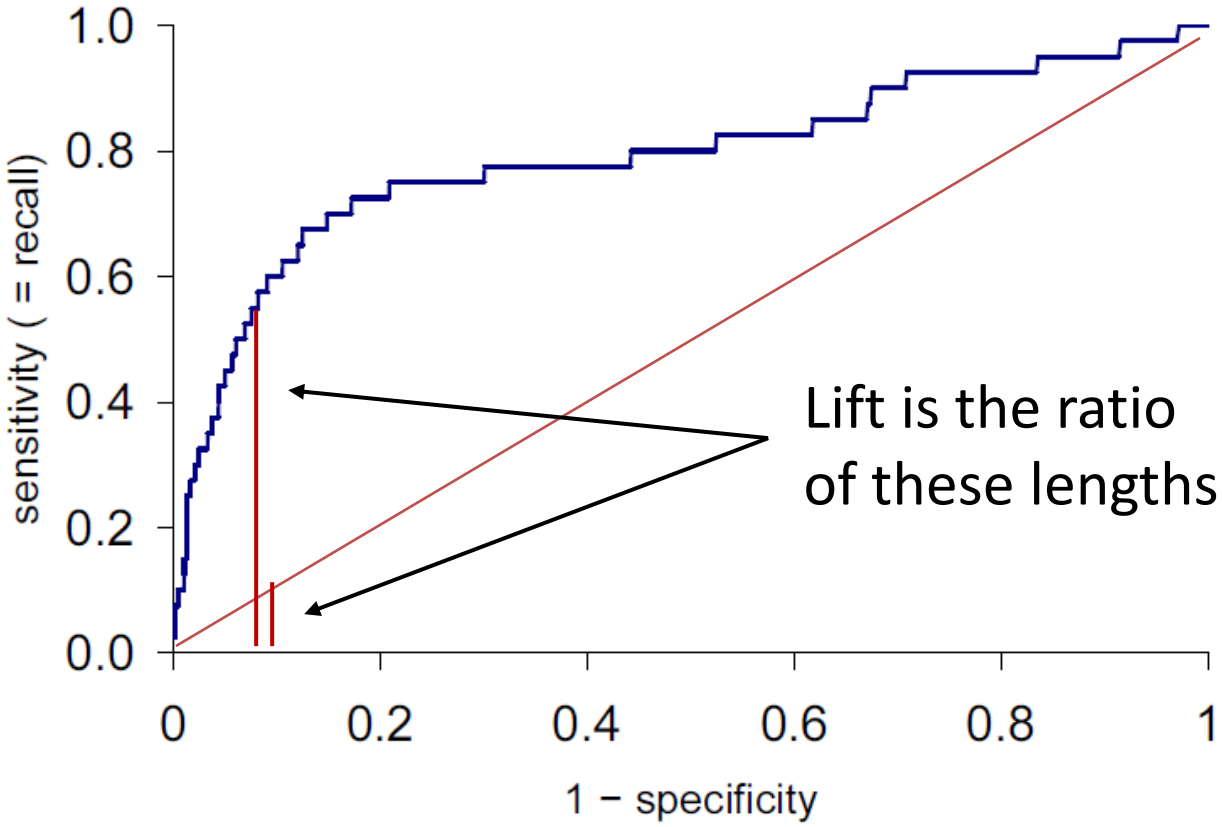
ROC AUC

ROC AUC is the “Area Under the Curve” – a single number that captures the overall quality of the classifier. It should be between 0.5 (random classifier) and 1.0 (perfect).



Lift Plot

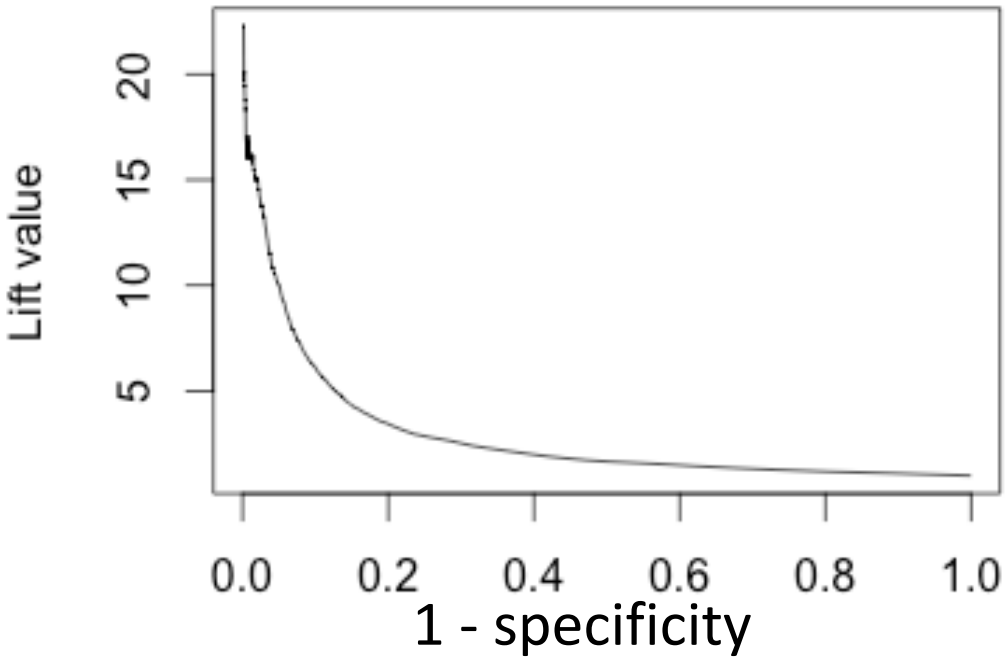
A derivative of the ROC plot is the lift plot, which compares the performance of the actual classifier/search engine against random ordering, or sometimes against another classifier.



Lift Plot

Lift plots emphasize initial precision (typically what you care about), and performance in a problem-independent way.

Note: The lift plot points should be computed at regular spacing, e.g. 1/00 or 1/1000. Otherwise the initial lift value can be excessively high, and unstable.

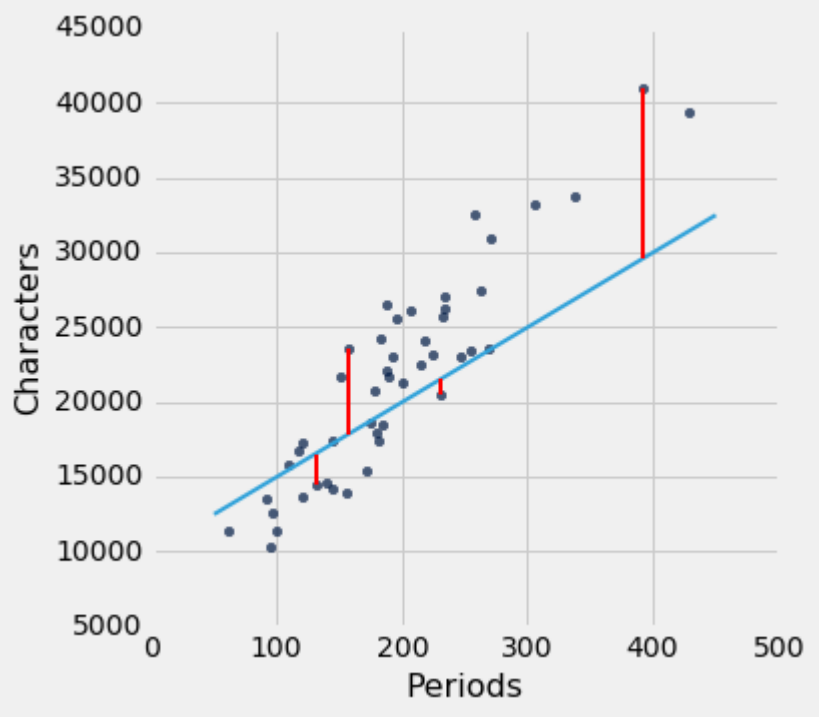


PERFORMANCE
IN
REGRESSION MODEL

Root Mean Square Error (RMSE)

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

The distance between the actual point and the predicted point, squaring this and then dividing by the amount of points we have for the mean.



R-Squared

R-Squared describes how well a model fits for a linear regression model. The higher R, the better the fit.

R-Squared (or also called the “Coefficient of Determination”) will show how close the data is to the fitted regression line.

In other words, It indicates the percentage of the variance in the dependent variable that the independent variables explain collectively.

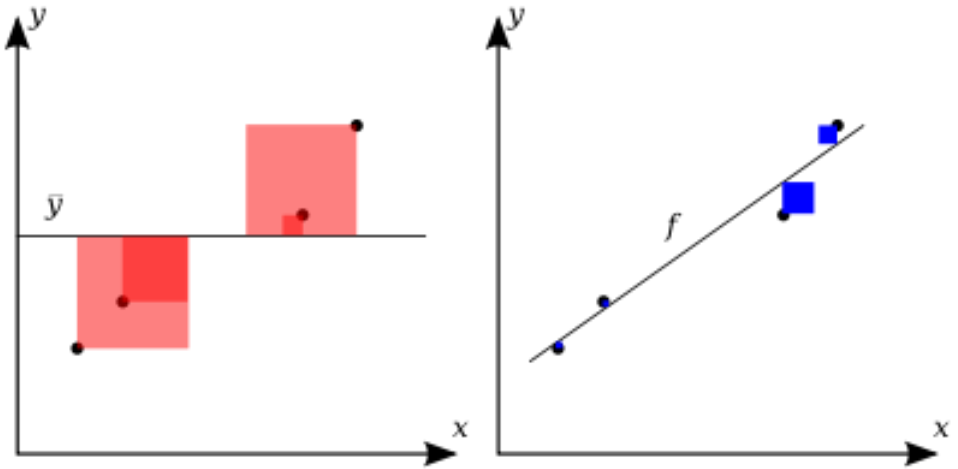
$$R_2 = 1 - \frac{ExplainedVariation}{TotalVariation} = 1 - \frac{SS_{res}}{SS_{tot}}$$

$\hat{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ (the mean of the observed data)

SS_{tot}: $\sum_i (y_i - \hat{y})^2$ (total sum of squares)

SS_{res}: $\sum_i (y_i - f_i)^2$ (sum of squares of residuals)

- SStot: red
- SSres: blue



EXAMPLE CV
in KNN Classifier

Reading Dataset

First step is to read in the data we will use as input.
Example: we are using the diabetes dataset

```
import pandas as pd

#read in the data using pandas
df = pd.read_csv('data/diabetes_data.csv')

#check data has been read in properly
df.head()
```

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age	diabetes
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Reading Dataset

How much data we have?
We will call the **'shape'** function on our dataframe to see how many rows and columns there are in our data.

Rows indicate the number of patients
Columns indicate the number of features (age, weight, etc.)

```
#check number of rows and columns in dataset
df.shape
```

```
(768, 9)
```

768 rows of data (potential diabetes patients)
9 columns (8 input features and 1 target output)

Split Dataset into Input & Target

Split up our dataset into inputs (X) and our target (y)
Our input will be every column **except 'diabetes'** because 'diabetes' is what we will be attempting to predict (our target).
We will use pandas **'drop'** function to drop the column **'diabetes'** from our dataframe and **store it in the variable 'X'**

```
#create a dataframe with all training data except the target column
X = df.drop(columns=['diabetes'])

#check that the target variable has been removed
X.head()
```

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

Split Dataset into Input & Target

Insert the ‘diabetes’ column of our dataset into our target variable (y)

```
#separate target values
y = df['diabetes'].values

#view target values
y[0:5]
```

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

Split Dataset into Train & Test Data

Training data is the data that the model will learn from.
Testing data is the data we will use to see how well the model performs on unseen data.
Scikit-learn has a function we can use called **'train_test_split'** that makes it easy for us to split our dataset into training and testing data.

'train_test_split' has 5 parameters
The first two parameters are the input and target data we split up earlier (**X, y**).
Next, we will set **'test_size' to 0.2**. This means that 20% of all the data will be used for testing, which leaves 80% of the data as training data for the model to learn from.
Setting **'random_state' to 1** ensures that we get the same split each time so we can reproduce our results.
Setting **'stratify' to y** makes our training split represent the proportion of each value in the y variable. For example, in our dataset, if 25% of patients have diabetes and 75% don't have diabetes, setting 'stratify' to y will ensure that the random split has 25% of patients with diabetes and 75% of patients without diabetes.

```
from sklearn.model_selection import train_test_split


#split dataset into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1, stratify=y)
```

Building & Training Model

Create a new **k-NN classifier** and set **'n_neighbors' to 3**.
This means that if at least 2 out of the 3 nearest points to an new data point are patients without diabetes, then the new data point will be labeled as 'no diabetes', and vice versa. In other words, a new data point is labeled with by majority from the 3 nearest points.

We have set 'n_neighbors' to 3 as a starting point.
Next, we need to train the model. In order to train our new model, we will use the **'fit'** function and pass in our training data as parameters to fit our model to the training data.

```
from sklearn.neighbors import KNeighborsClassifier

# Create KNN classifier
knn = KNeighborsClassifier(n_neighbors = 3)

# Fit the classifier to the data
knn.fit(X_train,y_train)
```

Testing Model

Once the model is trained, we can use the **'predict'** function on our model to make predictions on our test data.

As seen when inspecting 'y' earlier, 0 indicates that the patient does not have diabetes and 1 indicates that the patient has diabetes.

To save space, we will only show print the first 5 predictions of our test set.

```
#show first 5 model predictions on the test data
knn.predict(X_test)[0:5]
```

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

We can see that the model predicted **'no diabetes' for the first 4 patients** in the test set and **'has diabetes' for the 5th patient**.

Check Accuracy of Our Model

How our accurate our model is on the full test set?

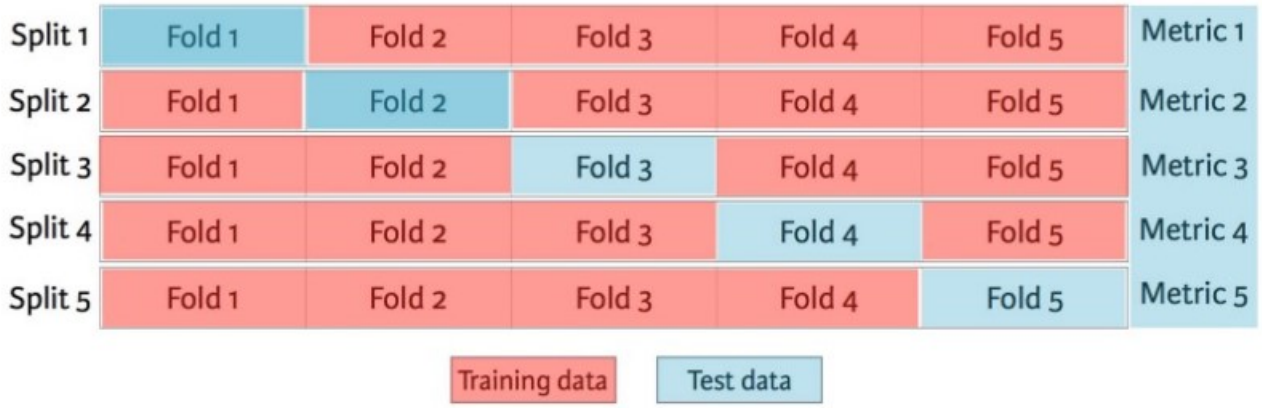
To do this, we will use the **'score'** function and pass in our test input and target data to see how well our model predictions match up to the actual results.

```
#check accuracy of our model on the test data
knn.score(X_test, y_test)
```

```
0.66883116883116878
```

K-Fold Cross Validation

- Cross-validation is when the dataset is randomly split up into ‘k’ groups.
- One of the groups is used as the test set and the rest are used as the training set.
- The model is trained on the training set and scored on the test set.
- Then the process is repeated until each unique group as been used as the test set.
- Example: for 5-fold cross validation, the dataset would be split into 5 groups, and the model would be trained and tested 5 separate times so each group would get a chance to be the test set. This can be seen in the graph below.



K-Fold Cross Validation

train-test-split method we used in earlier is called **'holdout'**
Cross-validation is better than using the holdout method because the holdout method score is dependent on how the data is split into train and test sets.
Cross-validation gives the model an opportunity to test on multiple splits so we can get a better idea on how the model will perform on unseen data.
In order to train and test our model using cross-validation, we will use the **'cross_val_score'** function with a cross-validation value of 5.
'cross_val_score' takes in our k-NN model and our data as **parameters**. Then it splits our data into 5 groups and fits and scores our data 5 separate times, recording the accuracy score in an array each time. We will save the accuracy scores in the **'cv_scores'** variable.
To find the average of the 5 scores, we will use numpy's mean function, passing in 'cv_score'.

```
from sklearn.model_selection import cross_val_score
import numpy as np


#create a new KNN model
knn_cv = KNeighborsClassifier(n_neighbors=3)


#train model with cv of 5
cv_scores = cross_val_score(knn_cv, X, y, cv=5)


#print each cv score (accuracy) and average them
print(cv_scores)
print('cv_scores mean:{}'.format(np.mean(cv_scores)))
```

```
[ 0.68181818  0.69480519  0.75324675  0.75163399  0.68627451]
cv_scores mean:0.7135557253204311
```

Hypertuning Parameter using GridSearch CV

Hypertuning parameters is to **find the optimal parameters for your model to improve accuracy**.
We use **GridSearchCV** to find the optimal value for 'n_neighbors'.
GridSearchCV works by **training our model multiple times on a range of parameters that we specify**.
So, we can test our model with each parameter and figure out the optimal values to get the best accuracy results.
We will specify a range of values for 'n_neighbors' to see which value works best for our model.
We will create a dictionary, setting 'n_neighbors' as the key and using numpy to create an array of values from 1 to 24.
Our new model using grid search will take in a new k-NN classifier, our param_grid and a cross-validation value of 5 in order to find the optimal value for 'n_neighbors'.

```
from sklearn.model_selection import GridSearchCV

#create new a knn model
knn2 = KNeighborsClassifier()

#create a dictionary of all values we want to test for n_neighbors
param_grid = {'n_neighbors': np.arange(1, 25)}

#use gridsearch to test all values for n_neighbors
knn_gscv = GridSearchCV(knn2, param_grid, cv=5)

#fit model to data
knn_gscv.fit(X, y)
```

Hypertuning Parameter using GridSearch CV

After training, we can check which of our values for 'n_neighbors' that we tested performed the best.

We will call **'best_params_'** on our model.

```
#check top performing n_neighbors value
knn_gscv.best_params_
```

```
{'n_neighbors': 14}
```

We can see that **14 is the optimal value for 'n_neighbors'**.
We can use the **'best_score_'** function to check the accuracy of our model when 'n_neighbors' is 14.
'best_score_' outputs the mean accuracy of the scores obtained through cross-validation.

```
#check mean score for the top performing value of n_neighbors
knn_gscv.best_score_
```

```
0.7578125
```