



University of Minho
School of Engineering



Dados e Aprendizagem Automática

Quality Metrics and Model Validation

DAA @ MEI-1º/MiEI-4º – 1º Semestre

Bruno Fernandes, César Analide, Dalila Alves, Filipa Ferraz, Victor Alves

Contents

2

- A Machine Learning Algorithm (Decision Trees)
- Quality Metrics
- Model Validation
- Hands On

A Machine Learning Algorithm

3

Today we will be “playing” with **Decision Trees**!

You don’t know what that means? Don’t worry, we will see it soon! For now, you just need to know that **it is a model that predicts the value of a target feature**:

- A Decision Tree is a hierarchical graph (it’s a tree)!
- Each branch represents a selection among a set of alternatives;
- Each leaf node represents a class.



A Machine Learning Algorithm - Classifier

4

We can have a **Decision Tree classifier**, which is used for classification problems (the target feature is a class):

- Deciding if we should have lunch - binary classification - Yes/No
- Surviving the Titanic - again, binary classification - 1/0
- Classify a set of images - multiclass classification - oranges/apples/pears
- ...



A Machine Learning Algorithm - Regressor

5

But we can also have a **Decision Tree regressor**, which is used for regression problems (the target feature is real/continuous):

- Predict the traffic flow (km/h)
- Predict stock prices (€)
- ...



Implementing a Decision Tree Classifier - Data Loading

6

Let's use the **Titanic dataset** and start by loading the dataset:

```
'''  
Load CSV  
'''  
  
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\_csv.html  
df = pd.read_csv('aula_2_dataset.csv')
```

```
'''  
Inspect dataset  
'''  
  
df.columns  
  
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',  
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],  
      dtype='object')
```

Implementing a Decision Tree Classifier - X and y

We now need to define our **input** and our **target** features!

```
#Let's start by creating our X (input data) and our y (target feature - the Survived feature)
X = df.drop(['Survived'], axis=1)      #input features - everything except the Survived feature
y = df['Survived'].to_frame()          #target feature
```

PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...	891 rows x 11 columns										...

Implementing a Decision Tree Classifier - X and y

8

We now need to define our **input** and our **target** features!

X is typically used to **identify the input**.

y to **identify the target**.

```
#Let's start by creating our X (input data) and our y (target feature - the Survived feature)  
X = df.drop(['Survived'], axis=1)      #input features - everything except the Survived feature  
y = df['Survived'].to_frame()         #target feature
```

y

Survived	
0	0
1	1
2	1
3	1
4	0
...	...

891 rows × 1 columns

Implementing a Decision Tree Classifier - Train/Test split

9

Both the **X** and the **y** have 891 rows of data - that corresponds to the entire dataset!

Hence, our next step is to **leave aside a small set of data to test/validate the model** (25%), like this:

```
#Let's use the X and Y, which contain 891 rows of data  
#to create train and test sets of data.  
#Important -> Define the random_state for reproducibility
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=2021)
```

Amount of data for testing 25%

Random seed

```
print("The shape of X %s. X_train has shape %s while X_test has shape %s" %(X.shape, X_train.shape, X_test.shape))
```

The shape of X (891, 11). X_train has shape (668, 11) while X_test has shape (223, 11)

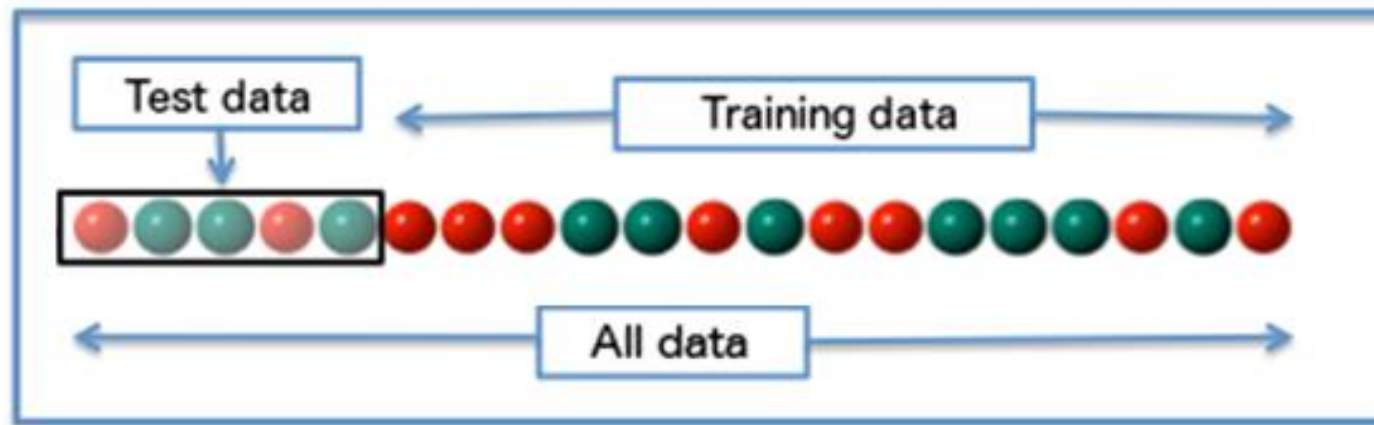
```
print("The shape of y %s. y_train has shape %s while y_test has shape %s" %(y.shape, y_train.shape, y_test.shape))
```

The shape of y (891, 1). y_train has shape (668, 1) while y_test has shape (223, 1)

Hold-out Validation

10

In essence, what we have done means we will **validate the model on unseen data**, i.e., we use a “partitioning method” to split the training and the testing data **once**. This means we **hold-out a subset of data** for testing (80/20; 75/25; 65/35...).



Implementing a Decision Tree Classifier - Model Fitting

11

How can we now use a DT to **predict whether a passenger would survive the disaster?**

We first **create an instance** of the classifier and then we use the **fit function**.

```
#Create an instance of a Decision Tree classifier  
#Again, defining the random_state for reproducibility  
clf = DecisionTreeClassifier(random_state=2021)
```

```
#Training, i.e., fitting the model (using the training data!!)  
clf.fit(X_train, y_train)
```

Implementing a Decision Tree Classifier - Model Fitting

12

Ups!!

```
#Create an instance of a Decision Tree classifier  
#Again, defining the random_state for reproducibility  
clf = DecisionTreeClassifier(random_state=2021)
```

```
#Training, i.e., fitting the model (using the training data!!)  
clf.fit(X_train, y_train)
```

```
-----  
ValueError                                Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel_1796\2168925757.py in <module>  
      1 #Training, i.e., fitting the model (using the training data!!)  
----> 2 clf.fit(X_train, y_train)  
  
ValueError: could not convert string to float: 'Ali, Mr. William'
```

Sklearn decision trees **do not handle categorical data** (see issue #12866)!!
(<https://github.com/scikit-learn/scikit-learn/pull/12866>)

Implementing a Decision Tree Classifier - Model Fitting

13

We could use techniques such as **Label** or **One-Hot encoding** to **handle categorical data!**
For now, let's just **drop** those features.

```
#dropping categorical features from the input data (X_train and X_test)  
X_train = X_train.drop(['Name', 'Sex', 'Age', 'Ticket', 'Cabin', 'Embarked'], axis=1)  
X_test = X_test.drop(['Name', 'Sex', 'Age', 'Ticket', 'Cabin', 'Embarked'], axis=1)
```

```
#Training, i.e., fitting the model (using the training data!!)  
clf.fit(X_train, y_train)
```

```
DecisionTreeClassifier(random_state=2021)
```


Implementing a Decision Tree Classifier - Prediction

14

With the model fitted, we can use the **predict function** to obtain the prediction of survival for each row/observation in the test set (0 - doesn't survive; 1 - survives).

```
predictions = clf.predict(X_test)
predictions
```

```
array([0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0,
       0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1,
       0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,
       0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1,
       0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1,
       1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
       1, 1, 0], dtype=int64)
```

We now have predictions for the test set (and we know the actual survival value as it is stored in the **y_test** variable). How do we evaluate our classification model? There are some options...

Quality Metrics and Model Evaluation

15

But first... Why **quality metrics**?

How else would we **quantify the model's performance**? Metrics are used to monitor and measure the performance of a model. Some metrics are the Mean Absolute Error, the Mean Squared Error, Accuracy, F1-Score, ... There are many!

However, **it depends on the problem in hands**. Is it a classification problem? A regression one? A time series?



Quality Metrics

16

- Confusion Matrix
- Accuracy
- Precision
- Recall
- ROC
- AUC
- F_1 Score
- F_β Score
- MAE
- MSE
- RMSE
- ...

Classification Models - Confusion Matrix

17

A **confusion matrix** is a table that is used to describe the performance of a **classification model** on a set of test data for which the true values are, again, known.

		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

Accuracy

18

Accuracy is simply calculated as the number of all correct predictions divided by the total number of observations.

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

		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

Precision and Recall

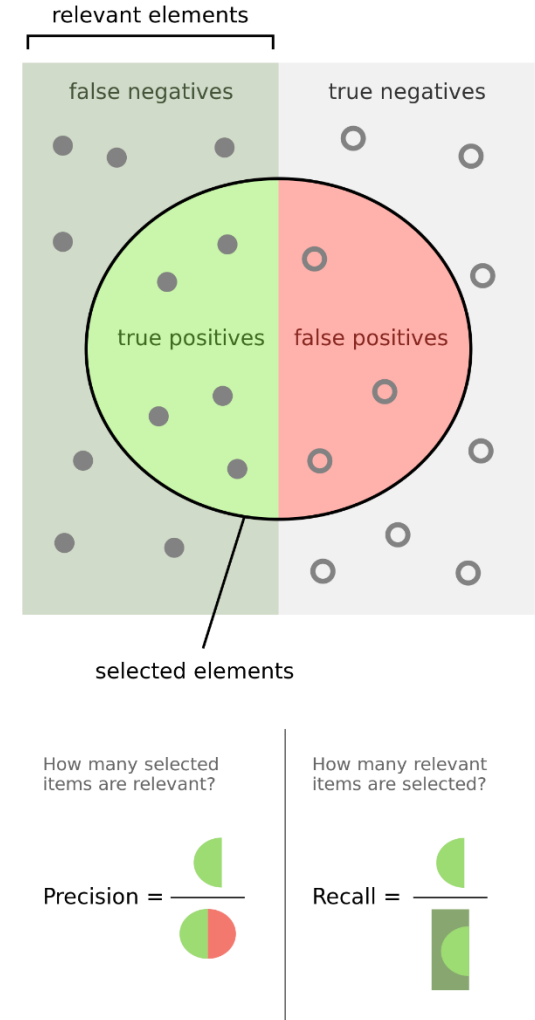
19

Precision (aka **Sensitivity**) is a **measure of exactness**, determines the fraction of relevant items among the retrieved items.

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

Recall (aka **Specificity**) is a **measure of completeness**, determines the fraction of relevant items that were obtained.

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



Precision and Recall

20

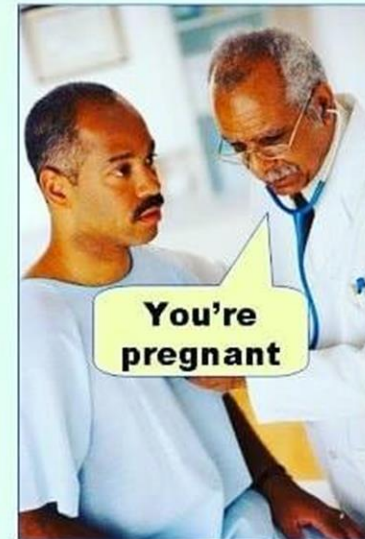
Precision (aka **Sensitivity**) is a **measure of exactness**, determines the fraction of relevant items among the retrieved items.

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

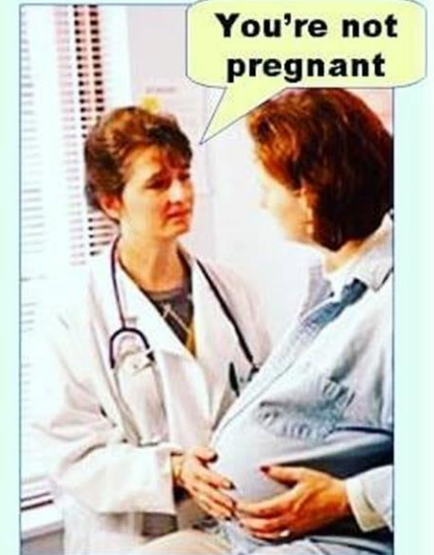
Recall (aka **Specificity**) is a **measure of completeness**, determines the fraction of relevant items that were obtained.

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

Type I error
(false positive)



Type II error
(false negative)



Confusion Matrix-based Metrics

21

Obtaining the **confusion matrix** is as simple as...

```
confusion_matrix(y_test, predictions)

array([[96, 39],
       [43, 45]], dtype=int64)
```

The same for the model's **accuracy**, **precision**, and **recall**!

```
accuracy_score(y_test, predictions)

0.6322869955156951
```

```
precision_score(y_test, predictions)

0.5357142857142857
```

```
recall_score(y_test, predictions)

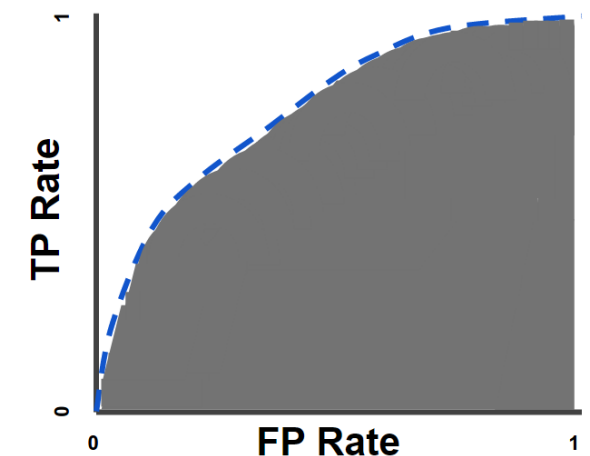
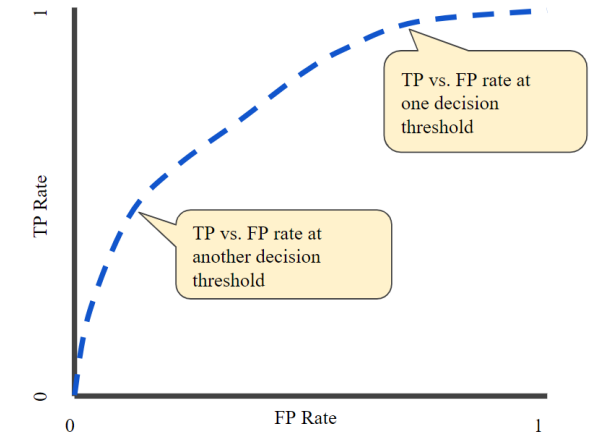
0.5113636363636364
```

ROC and AUC

22

The **Receiver Operating Characteristics** (ROC) curve finds the performance of a classification model at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives.

The **Area Under the Curve** (AUC) measures the two-dimensional area underneath the ROC curve (think integral calculus). It measures how well predictions are ranked, rather than their absolute values, and ranges from 0 to 1. A model whose predictions are 100% wrong has an AUC of 0; one whose predictions are 100% correct has an AUC of 1.



ROC and AUC

23

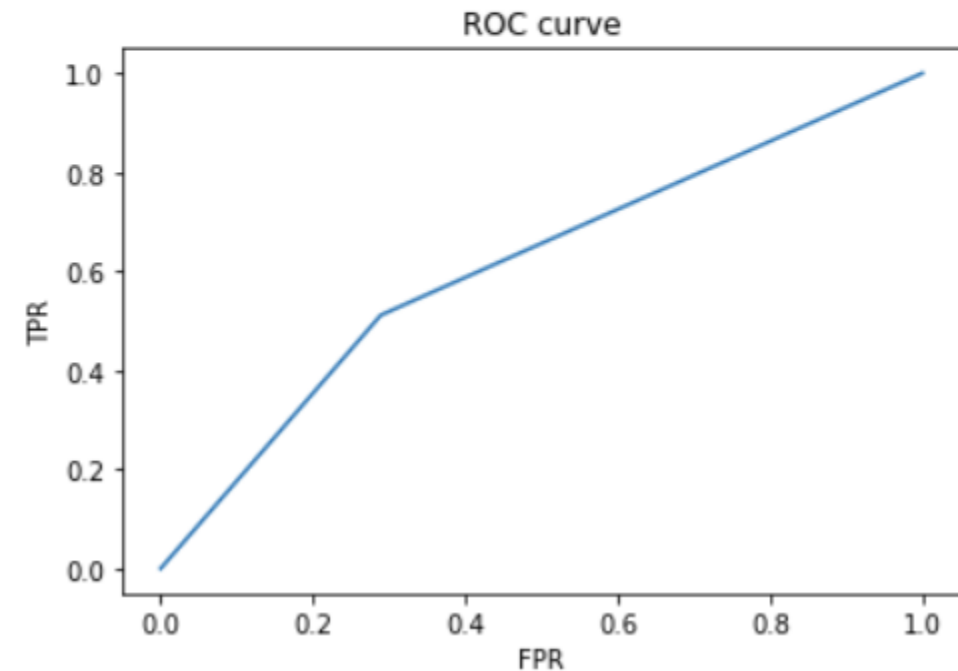
As for the **ROC** and the **AUC**...

```
roc_auc_score(y_test, predictions)
```

0.6112373737373737

```
fpr, tpr, _ = roc_curve(y_test, predictions)
```

```
plt.clf()  
plt.plot(fpr, tpr)  
plt.xlabel('FPR')  
plt.ylabel('TPR')  
plt.title('ROC curve')  
plt.show()
```



F_1 and F_β Score

24

The F_1 Score combines precision and recall into a single value for comparison purposes. Can be used to obtain a more balanced view of performance.

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F_1 Score gives equal weight to precision and recall. Use F_β Score to weight recall by a factor of β . With $\beta=1$, F_1 and F_β are equivalent.

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

F_1 and F_β Score

25

As for the F_1 and F_β Score ...

```
f1_score(y_test, predictions)
```

```
0.5232558139534884
```

```
fbeta_score(y_test, predictions, beta=0.5)
```

```
0.5306603773584905
```

A Decision Tree Regressor

26

But let's say that we wanted to **predict the FARE** paid by those that went to the Titanic (maybe not a very good problem, but it serves its purpose)!

For that, we would need a **Decision Tree regressor**!



Implementing a Decision Tree Regressor

27

We first need to re-define our **input** (X) and our **target** (y) features!

```
#Let's assume a REGRESSION problem! Let's predict the FARE paid by a person  
 #(maybe not a very good problem but it serves its purpose)!  
#Let's start by creating our X (input data) and our y (target feature - the Survived feature)  
X = df.drop(['Fare'], axis=1)      #input features - everything except the Survived feature  
y = df['Fare'].to_frame()         #target feature
```

And to **hold-out some data for testing** (and again **drop the categorical features**)!

```
#Let's use the X and Y, which contain 891 rows of data  
#to create train and test sets of data.  
#Important -> Define the random_state for reproducibility  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=2021)  
  
#dropping categorical features from the input data (X_train and X_test)  
X_train = X_train.drop(['Name', 'Sex', 'Age', 'Ticket', 'Cabin', 'Embarked'], axis=1)  
X_test = X_test.drop(['Name', 'Sex', 'Age', 'Ticket', 'Cabin', 'Embarked'], axis=1)
```

Implementing a Decision Tree Regressor

28

Now, just **fit** and **predict** the fare paid by the people at the test set.

```
#Training, i.e., fitting the model (using the training data!!)  
clf.fit(X_train, y_train)
```

```
DecisionTreeRegressor(random_state=2021)
```

```
predictions = clf.predict(X_test)  
predictions
```

```
array([ 18.7875,   7.8958,  10.5   ,  27.9   ,  34.375 ,  24.15   ,  
       211.5   ,   7.7417,  12.35   ,   7.2292,   7.2292,  52.    ,  
       15.5   ,  18.7875,  13.    ,   8.6625,   8.05   ,  52.    ,  
        8.6625,  35.5   ,  76.2917,  15.5   ,   7.6292,   8.4042,  
        8.6625,   7.125 ,   7.775 ,   7.65   ,  93.5   ,  26.    ,  
        7.8792,   8.6625,  39.6875,  16.1   , 113.275 ,  12.35   ,
```

We now have fare predictions for the test set (and we know the actual fare value as it is stored in the **y_test** variable). How do we evaluate our **regression model**? There are, again, some options...

Mean Absolute Error

29

MAE

Mean Absolute Error measures the average magnitude of the errors in a set of predictions, without considering their direction.

$$MAE = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j|$$

Where n is the number of observations, and y_j and \hat{y}_j are the actual observation and the predicted value, respectively.

Mean Squared Error

30

MSE

Mean Squared Error consists of the average of squared differences between the prediction and the actual observation, without considering their direction.

$$MSE = \frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2$$

Where n is the number of observations, and y_j and \hat{y}_j are the actual observation and the predicted value, respectively.

Root Mean Squared Error

31

RMSE

Root Mean Squared Error consists of the square root of the average of squared differences between the prediction and the actual observation, without considering their direction.

$$RMSE = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

Where n is the number of observations, and y_j and \hat{y}_j are the actual observation and the predicted value, respectively.

Regression Quality Metrics

32

$$MAE = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j| \quad MSE = \frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2 \quad RMSE = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

Important notes:

- Three of the most common metrics used to **measure accuracy for continuous variables**
- All express **average model prediction error** (lower values are better)
- All range from 0 to ∞ and are **indifferent to the direction of errors**
- **MAE** and **RMSE** express the prediction error **in units of the variable of interest**
- **MSE** and **RMSE**, by squaring the error, gives a relatively **high weight to large errors**
- Hence, **MSE** and **RMSE** are more **useful when large errors** are particularly **undesirable**

Regression Quality Metrics

33

$$MAE = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j|$$

$$MSE = \frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2$$

$$RMSE = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

#	Error	Error	Error ²
1	1	1	1
2	-1	1	1
3	3	3	9
4	3	3	9

MAE	MSE	RMSE
2	5	2.24

#	Error	Error	Error ²
1	0	0	0
2	0	0	0
3	0	0	0
4	10	10	100

MAE	MSE	RMSE
2.5	25	5

Regression Quality Metrics

34

Obtaining the model's **MSE**, **MAE**, and **RMSE** is as simple as:

```
mean_absolute_error(y_test, predictions)
```

```
14.68592556053812
```

```
#squared parameter as TRUE for MSE
```

```
mean_squared_error(y_test, predictions, squared=True)
```

```
1399.7722041242152
```

```
#squared parameter as FALSE for RMSE
```

```
mean_squared_error(y_test, predictions, squared=False)
```

```
37.41352969347072
```

For example, the RMSE tells us that our mean error is of 37.41€ ...

Quality Metrics

35

Here are some basic functions/classes you'll need (somewhen) in the future:

- `sklearn.metrics.accuracy_score`
- `sklearn.metrics.auc`
- `sklearn.metrics.mean_absolute_error`
- `sklearn.metrics.mean_squared_error`
- `sklearn.metrics.f1_score`
- `sklearn.metrics.make_scorer`
- ...

Imports

36

And here are (some of) the imports you may need:

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import confusion_matrix
from sklearn.metrics import recall_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.metrics import f1_score
from sklearn.metrics import fbeta_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
#USING CROSS VALIDATION
from sklearn.model_selection import cross_val_score
```


Model Validation

37

What about **model validation**?

Well, we must **confirm that the model actually achieves its intended purpose**! The goal is to check the accuracy/performance of the model based on data the model doesn't know.

Until now, we have been using **hold-out validation**! But that's a very basic way to address the problem, right?

Do you see any problems with it?



Model Validation

38

- Hold-out Validation
- Cross Validation
- k-fold Cross Validation
- Leave-one-out Cross Validation
- ...

Cross Validation

40

Cross validation is another model validation technique!

The goal is to have an accurate metric of how the model will perform in practice.

In essence, it consists in dividing the dataset into k folds. In each run of the model, $k-1$ folds are used for training and 1 fold (the remaining) is used as test. Keep repeating the process until all folds have been used for testing.

The final error metric is based on the mean value of all error metrics:

$$E = \frac{1}{k} \sum_{i=1}^k E_i$$

Cross Validation - How many folds?

41

Well, ...

A **greater number of folds** will lead to a **better error estimate** of the model, a **lower bias** and **less overfitting**! However, it comes with a **higher computational cost**!

If we have a **large dataset**, a **smaller k** may be enough since we will have a larger amount of data for training. If we have a **small dataset**, we may want to **use leave-one-out cross validation** to maximize the amount of data for training...

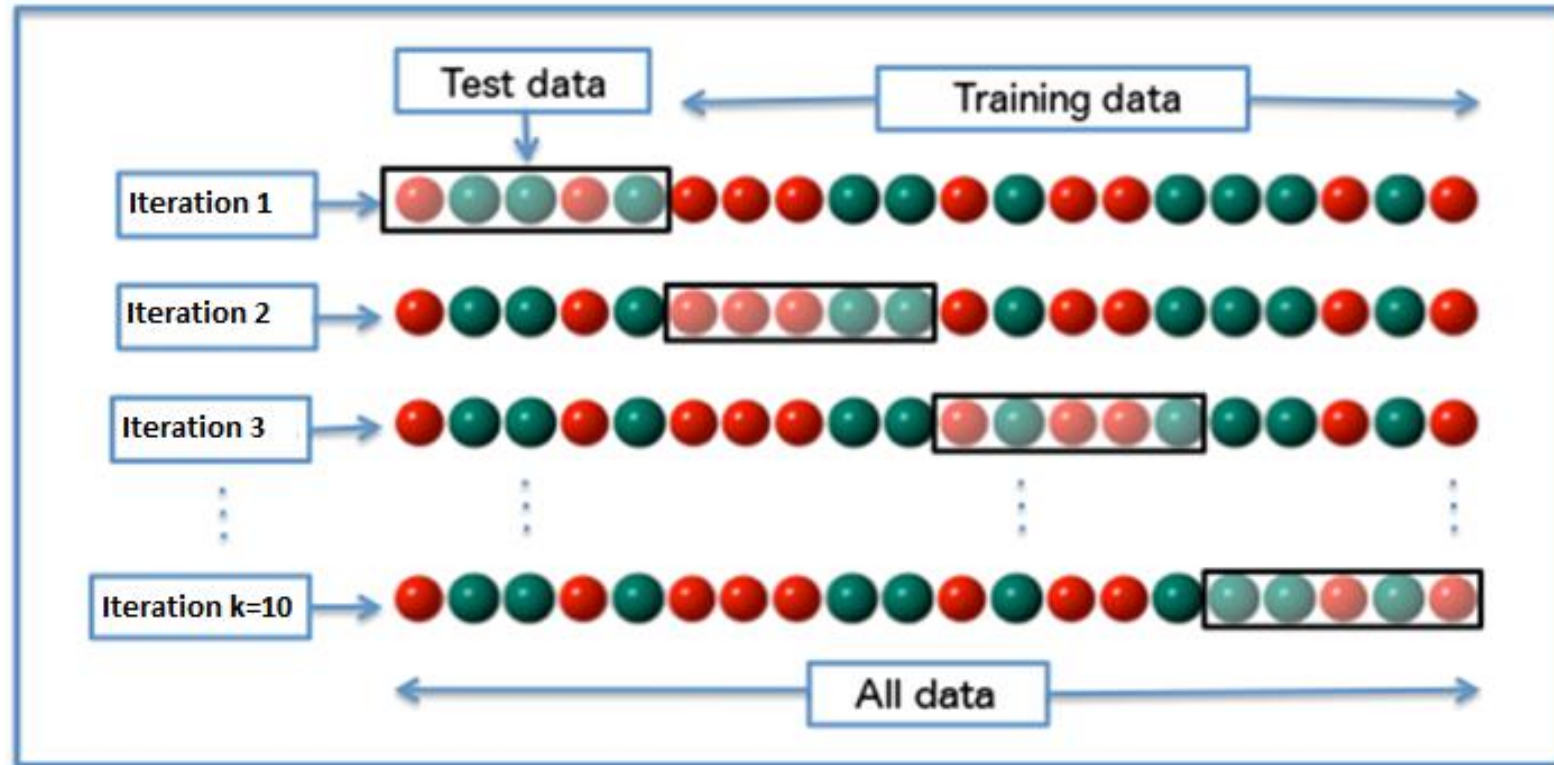
In reality, **k depends on N!!**

Rule of thumb → **k=10!**

Row ID	D Error in %	I Size of Test Set	I Error Count
fold 0	38.281	128	49
fold 1	35.156	128	45
fold 2	44.531	128	57
fold 3	42.969	128	55
fold 4	42.969	128	55
fold 5	41.406	128	53
fold 6	41.406	128	53
fold 7	40.625	128	52
fold 8	41.406	128	53
fold 9	42.52	127	54

k-fold Cross Validation

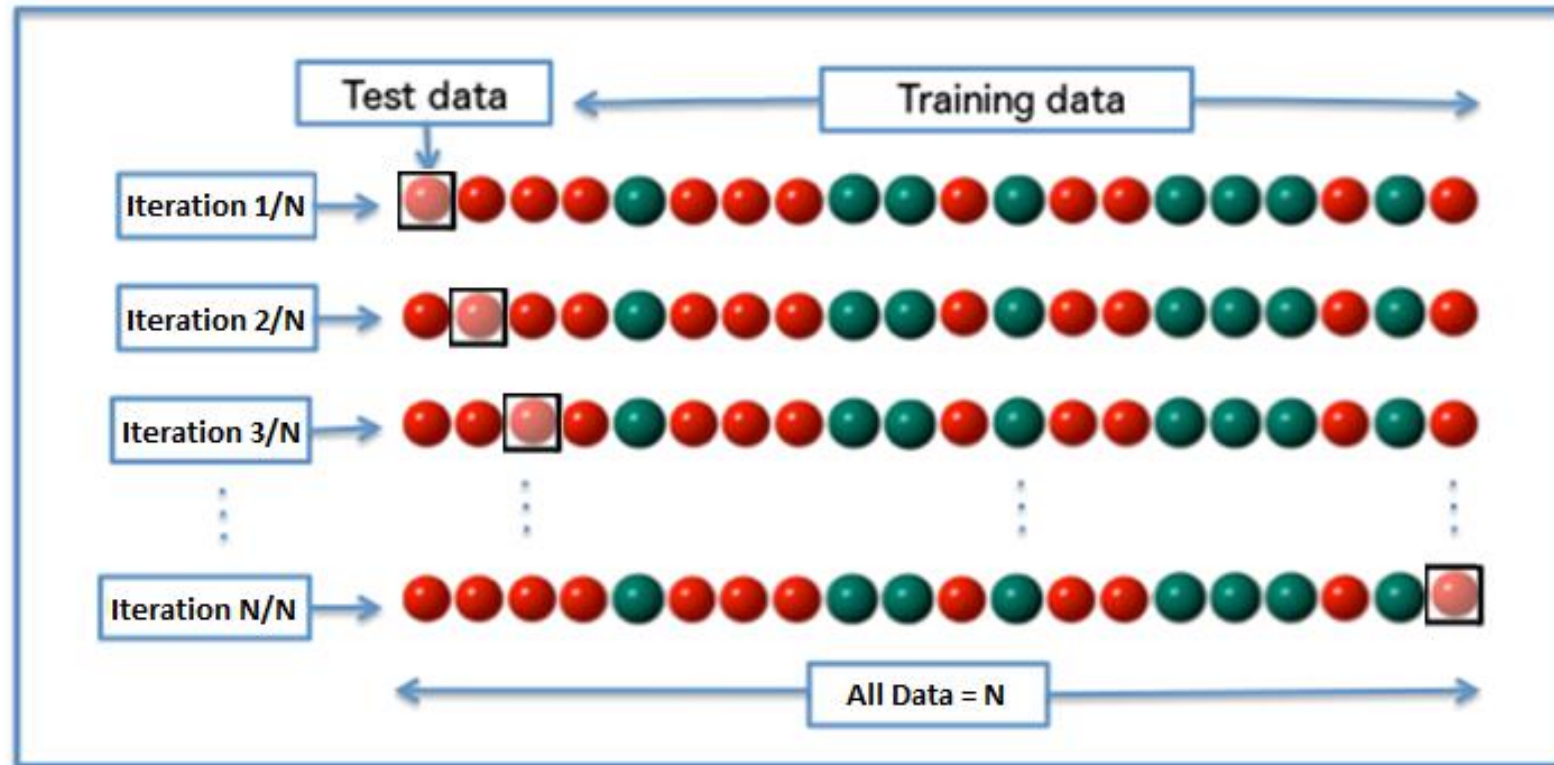
42



Usually, $k=10$.

Leave-one-out Cross Validation ($k=N$)

43



The special case of having $k=N$. Expensive...
But a good approach when we have a **small dataset**.

Model Validation

44

Here are some basic functions/classes you'll need (somewhen) in the future:

- `sklearn.model_selection.train_test_split`
- `sklearn.model_selection.Kfold`
- `sklearn.model_selection.LeaveOneOut`
- `sklearn.model_selection.StratifiedKFold`
- `sklearn.model_selection.GridSearchCV`
- `sklearn.model_selection.RandomizedSearchCV`
- ...

Model Validation

45

Using the `cross_val_score` API.

```
print("USING A DECISION TREE WITH cross_val_score (MEAN ACCURACY)...")
X = X.drop(['Name', 'Sex', 'Age', 'Ticket', 'Cabin', 'Embarked'], axis=1)
clf = DecisionTreeClassifier(criterion = 'gini', max_depth = 10, random_state=2021)
scores = cross_val_score(clf, X, y, cv=10)
print(scores)
print("RESULT: %0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
USING A DECISION TREE WITH cross_val_score (MEAN ACCURACY)...
[0.58888889 0.61797753 0.52808989 0.50561798 0.61797753 0.70786517
 0.70786517 0.70786517 0.59550562 0.74157303]
RESULT: 0.63 accuracy with a standard deviation of 0.08
```

10 folds!



Model Validation

46

Or iterating manually with K-fold...

```
...
Iterating manually (with K-fold, Repeated K-fold, Leave One Out, Shuffle Split, Stratified k-fold, TimeSeriesSplit, ...)
...

print("USING A DECISION TREE WITH MANUAL ITERATION (KFold) and obtaining confusion matrix...")
from sklearn.model_selection import KFold
scores = []
kf = KFold(n_splits=10)
for train, test in kf.split(X):
    clf.fit(X.loc[train,:], y.loc[train,:])
    score = clf.score(X.loc[test,:], y.loc[test,:])
    scores.append(score)
    y_predicted = clf.predict(X.loc[test,:])
    print("Confusion Matrix:")
    print(confusion_matrix(y.loc[test,:], y_predicted))
    print(score)
print("RESULT: %.2f accuracy with a standard deviation of %.2f" % (np.mean(scores), np.std(scores)))
```

USING A DECISION TREE WITH MANUAL ITERATION (KFold) and obtaining confusion matrix...

Confusion Matrix:

```
[[45  6]
 [27 12]]
```

0.6333333333333333

Confusion Matrix:

```
[[33 36]
 [ 8 12]]
```

0.5056179775280899

Confusion Matrix:

```
[[38 17]
 [19 15]]
```

0.5955056179775281

RESULT: 0.65 accuracy with a standard deviation of 0.06

Hands On

47

Spyder (Python 3.6)

File Edit Search Source Run Debug Consoles Projects Tools View Help

Editor - C:\data\PythonWorkspace\dev\meanshift_algorithm.py

```
37 class Mean_Shift:
38     def __init__(self, radius=None, radius_normalize_step = 150):
39         self.radius = radius
40         self.radius_normalize_step = radius_normalize_step
41
42     def fit(self, data):
43
44         if self.radius == None:
45             all_data_centroid = np.average(data, axis=0)
46             all_data_norm = np.linalg.norm(all_data_centroid)
47             self.radius = all_data_norm/self.radius_normalize_step
48
49         centroids = {}
50
51         #initialize centroids
52         for i in range(len(data)):
53             centroids[i] = data[i]
54
55         weights = [1 for i in range(self.radius_normalize_step)]
56
57         while True:
58             new_centroids = []
59             for i in centroids:
60                 in_range = []
61                 centroid = centroids[i]
62
63                 for featureset in data:
64                     distance = np.linalg.norm(featureset-centroid)
65                     if distance == 0:
66                         distance = 0.0000000001
67                     weight_index = int(distance/self.radius)
68                     if weight_index > self.radius_normalize_step-1:
69                         weight_index = self.radius_normalize_step-1
70                     to_add = (weights[weight_index]**2)*[featureset]
71                     in_range += to_add
72
73             new_centroid = np.average(in_range, axis=0)
```

Variable explorer

Name	Type	Size	Value
batch_size	int	1	100
mnist	contrib.learn.python.learn.datasets.base.Datasets	3	Datasets object of...
n_classes	int	1	10
n_nodes_h1	int	1	500
n_nodes_h2	int	1	500
n_nodes_h3	int	1	500

Variable explorer | File explorer | Help

IPython console

Console 1/A

See 'tf.nn.softmax_cross_entropy_with_logits_v2'.

Epoch 0 completed out of 10 loss: 1666037.4677734375
Epoch 1 completed out of 10 loss: 377809.3128890991
Epoch 2 completed out of 10 loss: 201302.4857263565
Epoch 3 completed out of 10 loss: 119427.91378033161
Epoch 4 completed out of 10 loss: 72651.25679710507
Epoch 5 completed out of 10 loss: 45327.621502393486
Epoch 6 completed out of 10 loss: 31955.17812934518
Epoch 7 completed out of 10 loss: 23664.35610633137
Epoch 8 completed out of 10 loss: 18248.740643078025
Epoch 9 completed out of 10 loss: 19962.00065876091
Accuracy: 0.9511

In [2]:

IPython console | History log

HANDS ON