# Validation Metrics

This tutorial describes the validation metrics that you can use to evaluate the performance of regression and classification estimators.

Given a learning task the dataset for which we want to learn the model is split into two sets: the training set in which the model parameters are fitted, and a test set where the accuracy of the model is evaluated. A common validation technique is the so-called cross-validation.[1]

**K-fold Cross Validation**

Decide a value for **k** and split the data set into **k** subsets called **folds.**

- The model is trained using k-1 of the folds as the training data.
- The resulting model is tested on the remaining fold.
- The procedure is repeated k times. Each time with different k-1 folds to train and consequently a different test fold.
- The final accuracy is obtained by the average of the accuracies obtained on the different k iterations.

## Regression

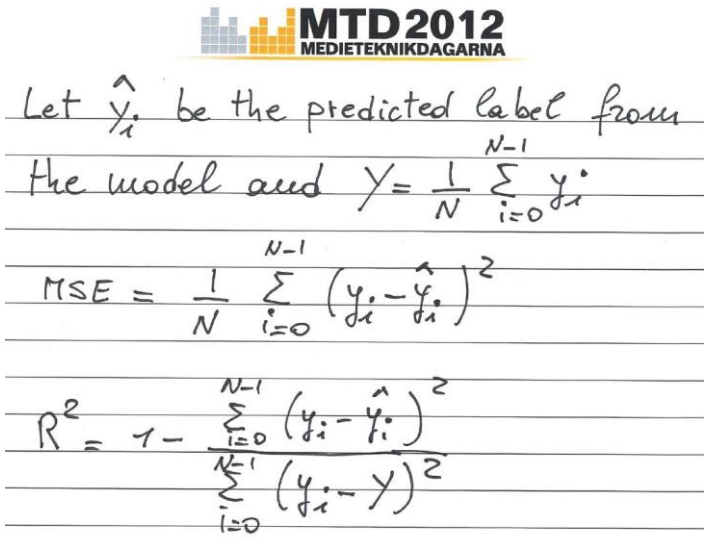
We will be using the housing dataset of Boston's suburbs available as data set at sklearn library.[2]

The dataset has 13 features:

[1] For a detailed presentation see https://scikit-learn.org/stable/modules/cross_validation.html
[2] The housing data set is archived at stored at http://archive.ics.uci.edu/ml/datasets/Housing

- **CRIM:** Per capita crime rate by town
- **ZN:** Proportion of residential land zoned for lots over 25,000 sqft
- **INDUS:** Proportion of non-retail business acres per town
- **CHAS:** Charles River variable (1 if tract bounds river; 0 otherwise)
- **NOX:** Nitric oxides concentration (parts per 10 million)
- **RM:** Average number of rooms per dwelling
- **AGE:** Proportion of owner-occupied units built prior to 1940
- **DIS:** Weighted distances from five Boston employment centers
- **RAD:** Index of accessibility to radial highways
- **TAX:** Full-value property tax rate per $10,000
- **PTRATIO:** Pupil-teacher ratio by town
- **B:** 1000(Bk - 0.63)^2, where Bk is the proportion of blacks by town
- **LSTAT:** The percentage of lower status of the population and the labels that we want to predict are MEDV, which represent the house value values (in $1000)

To evaluate the quality of the models, the mean squared error and the coefficient of determination $R^2$ are calculated.



The best result is $R^2=1$, which means the model perfectly fits the data, while $R^2=0$ is associated with a model with a constant line (negative values indicate an increasingly worse fit).

We start by importing the necessary libraries.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn import svm
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
```

The housing data is loaded using the sklearn library. We use 10 folds cross-validation to evaluate the performance of several estimators including linear regression, ridge regression, lasso regression, decision trees and random forests, and SVM regression.

```
from sklearn.datasets import load_boston
boston = load_boston()
X=boston.data
Y=boston.target
cv = 10

print('\nlinear regression')
lin = LinearRegression()
scores = cross_val_score(lin, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(lin, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('\nridge regression')
ridge = Ridge(alpha=1.0)
scores = cross_val_score(ridge, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(ridge, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('\nlasso regression')
```

```
lasso = Lasso(alpha=0.1)
scores = cross_val_score(lasso, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(lasso, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('\ndecision tree regression')
tree = DecisionTreeRegressor(random_state=0)
scores = cross_val_score(tree, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(tree, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('\nrandom forest regression')
forest = RandomForestRegressor(n_estimators=50, max_depth=None, min_samples_split=2, rand
om_state=0)
scores = cross_val_score(forest, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(forest, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('\nlinear support vector machine')
svm_lin = svm.SVR(epsilon=0.2,kernel='linear',C=1)
scores = cross_val_score(svm_lin, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(svm_lin, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('\nsupport vector machine rbf')
clf = svm.SVR(epsilon=0.2,kernel='rbf',C=1.)
scores = cross_val_score(clf, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(clf, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('\nknn')
knn = KNeighborsRegressor()
scores = cross_val_score(knn, X, Y, cv=cv)
print("mean R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(knn, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))
```

Output:

```
linear regression
mean R2: 0.20 (+/- 1.20)
MSE: 34.60

ridge regression
mean R2: 0.25 (+/- 1.05)
MSE: 33.96

lasso regression
mean R2: 0.26 (+/- 1.00)
MSE: 34.06

decision tree regression
mean R2: -0.13 (+/- 1.95)
MSE: 36.23

random forest regression
mean R2: 0.43 (+/- 1.02)
MSE: 22.17

linear support vector machine
mean R2: 0.30 (+/- 1.11)
MSE: 32.84

support vector machine rbf
mean R2: -1.00 (+/- 2.17)
MSE: 91.15

knn
mean R2: -4.95 (+/- 25.12)
MSE: 107.66
```

The best model fit is obtained using a random forest (with 50 trees); it returns an average coefficient of determination of 0.43 and MSE=22.17. As expected, the decision tree regressor has a lower $R^2$ and higher MSE than the random forest; -0.13 and 36.23 respectively.

The support vector machine with the rbf kernel (C=1, $\varepsilon$ = 0.2 ) is one of the worst models with a huge MSE error 91.15 and at $R^2$ = -1.0, while SVM with the linear kernel (C=1, $\varepsilon$ = 0.2 ) returns a poor model ($R^2$ = 0.3 and MSE = 32.84 ).

The lasso and ridge regressors have comparable results, around $R^2$ =0.25 and MSE=34.

The k-neighbors regressor is the worst model with the lowest $R^2$ = -4.95 and highest MSE error 107.66.

**Recursive Feature Elimination (RFE)**

An important procedure to improve the model results is feature selection. It often happens that only a subset of the total features is relevant to perform the model training while the other features may not contribute at all to the model. Feature selection may improve $R^2$ because misleading data is disregarded (fewer features to consider).

There are several techniques for extracting the best features for a certain model. In this context we explore the so-called recursive feature elimination method (RFE) which essentially considers the attributes associated with the largest absolute weights until the desired number of features are selected. In the case of the SVM algorithm, the weights are just the values of w, while for regression they are the model parameters θ.

The RFE function returns a list of Booleans (the support_ attribute) to indicate which features are selected (true values) and which are not (false values). The selected features are then used to evaluate the model as we have done before.

The following script prints the 4 best features selected by RFE for linear regression.

```
best_features=4
rfe_lin = RFE(lin,best_features).fit(X,Y)
supported_features=rfe_lin.get_support(indices=True)
for i in range(0, 4):
  z=supported_features[i]
  print(i+1,boston.feature_names[z])
```

Output:

```
1 CHAS
2 NOX
3 RM
4 PTRATIO
```

We use the sklearn built-in function RFE specifying to select the 4 best features.

```
from sklearn.feature_selection import RFE

best_features=4

print('feature selection on linear regression')
rfe_lin = RFE(lin,best_features).fit(X,Y)
mask = np.array(rfe_lin.support_)
scores = cross_val_score(lin, X[:,mask], Y, cv=cv)
print("R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(lin, X[:,mask],Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('feature selection ridge regression')
rfe_ridge = RFE(ridge,best_features).fit(X,Y)
mask = np.array(rfe_ridge.support_)
scores = cross_val_score(ridge, X[:,mask], Y, cv=cv)
print("R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(ridge, X[:,mask],Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('feature selection on lasso regression')
rfe_lasso = RFE(lasso,best_features).fit(X,Y)
mask = np.array(rfe_lasso.support_)
scores = cross_val_score(lasso, X[:,mask], Y, cv=cv)
print("R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(lasso, X[:,mask],Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('feature selection on decision tree')
rfe_tree = RFE(tree,best_features).fit(X,Y)
mask = np.array(rfe_tree.support_)
scores = cross_val_score(tree, X[:,mask], Y, cv=cv)
print("R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(tree, X[:,mask],Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('feature selection on random forest')
rfe_forest = RFE(forest,best_features).fit(X,Y)
mask = np.array(rfe_forest.support_)
scores = cross_val_score(forest, X[:,mask], Y, cv=cv)
print("R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(forest, X[:,mask],Y, cv=cv)
```

```
print("MSE: %0.2f" % mean_squared_error(Y,predicted))

print('feature selection on linear support vector machine')
rfe_svm = RFE(svm_lin,best_features).fit(X,Y)
mask = np.array(rfe_svm.support_)
scores = cross_val_score(svm_lin, X[:,mask], Y, cv=cv)
print("R2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
predicted = cross_val_predict(svm_lin, X,Y, cv=cv)
print("MSE: %0.2f" % mean_squared_error(Y,predicted))
```

Output:

```
feature selection on linear regression
R2: 0.59 (+/- 0.23)
MSE: 33.14

feature selection ridge regression
R2: 0.59 (+/- 0.23)
MSE: 33.22

feature selection on lasso regression
R2: 0.66 (+/- 0.16)
MSE: 27.34

feature selection on decision tree
R2: 0.74 (+/- 0.29)
MSE: 20.72

feature selection on random forest
R2: 0.82 (+/- 0.16)
MSE: 13.77

feature selection on linear support vector machine
R2: 0.58 (+/- 0.22)
MSE: 25.82
```

Even by using only four features we observe a significant increase of the value of $R^2$ for every model, and a decrese of the MSE error. The best model remains the random forest with 50 trees.

Note that the KNN algorithm does not provide weights on the features, so the RFE method cannot be applied.

# Classification

We present some popular metrics used to evaluate the performance of classification models. We start by introducing a useful matrix.

## Confusion Matrix

A confusion matrix is a table that is used to describe the performance of a classification model (or classifier) on a set of test data for which the true values are known.

A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. The confusion matrix shows the ways in which your classification model is confused when it makes predictions; one class is commonly mislabeled as the other. It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made.

Let us define an experiment from **P** positive instances and **N** negative instances for some condition. The four outcomes can be formulated in a 2×2 confusion matrix.

(*) ground True

| | | **Actual *** | |
|---|---|---|---|
| | | Positive | Negative |
| **Predicted** | Positive | True Positive | False Positive |
| | Negative | False Negative | True Negative |

**Definition of the Terms**

- Positive **P**: Observation is positive (*)
- Negative **N**: Observation is negative (*)
- True Positive **TP**: Observation is predicted positive and is positive
- False Positive **FP**: Observation is predicted positive but is negative
- True Negative **TN**: Observation is predicted negative and is negative
- False Negative **FN**: Observation is predicted negative but is positive

## Example 1

If a classification system has been trained to distinguish between cats, dogs and rabbits, a confusion matrix will summarize the results of testing the algorithm for further inspection.

Assume a sample of 27 animals — 8 cats, 6 dogs, and 13 rabbits.

| | | Actual class | | |
|---|---|---|---|---|
| | | **Cat** | **Dog** | **Rabbit** |
| **Predicted** | **Cat** | **5** | 2 | 0 |
| | **Dog** | 3 | **3** | 2 |
| | **Rabbit** | 0 | 1 | **11** |

Of the 8 actual cats, the system predicted that three were dogs, and of the six dogs, it predicted that one was a rabbit and two were cats. Of the 13 rabbits, the system predicted that two were dogs.

Clearly, the system in question has trouble distinguishing between cats and dogs, but can make the distinction between rabbits pretty well.

In the table above, all correct predictions are located in the diagonal of the table (highlighted in bold), so it is easy to visually inspect the table for prediction errors, as they will be represented by values outside the diagonal.

The confusion matrix for the class Cat is:

| | | Actual | |
|---|---|---|---|
| | | Cat | Non-cat |
| **Predicted** | Cat | T P<br>**5** | F P<br>**2** |
| | Non-cat | F N<br>**3** | T N<br>**17** |

## Standard Metrics

### Accuracy

Accuracy is the proportion of the total number of predictions that are correct.

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

The accuracy may not be an adequate performance measure when the number of negative cases is much greater than the number of positive or viceversa.

- Accuracy is not a reliable metric for the real performance of a classifier because it will yield misleading results if the data set is unbalanced (when the numbers of observations in different classes vary greatly).
- **Example** If there are 95 cats and only 5 dogs in the data, a particular classifier might classify all the observations as cats.

  The overall accuracy would be 95%, but in more detail the classifier would have a 100% recognition rate for the cat class
  but a 0% recognition rate for the dog class.

### Recall

Recall is the proportion of positive cases that are correctly classified.
High Recall indicates that the class is correctly recognized (small number of FN).

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

### Precision

Precision is the proportion of the predicted positive cases that are correct.
High Precision indicates that a sample labeled as positive is indeed positive (small number of FP).

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

- **High recall, low precision:**
  This means that most of the positive examples are correctly recognized (low FN) but there are a lot of false positives.

- **Low recall, high precision:**
  This shows that we miss a lot of positive examples (high FN) but those we predict as positive are indeed positive (low FP).

**F-Measure**

Since we have two measures (Precision and Recall) it helps to have a measurement that represents both of them. We calculate the F-measure which uses the harmonic mean in place of the arithmetic mean as it punishes the extreme values more.

The F-Measure will always be nearer to the smaller value of Precision or Recall.

$$\text{F-measure} = 2 * \frac{\text{Recall} * \text{Precision}}{\text{Recall} + \text{Precision}}$$

## Example 2

Consider again the classification problem outlined in Example 1. Assume that we want to evaluate the perfomance of the model with respect to the class Cat.

| | | Actual | |
|---|---|---|---|
| | | Cat | Non-cat |
| Predicted | Cat | T P **5** | F P **2** |
| | Non-cat | F N **3** | T N **17** |

Then, we can calculate the previous metrics.

- **Accuracy** = (TP+TN)/(TP+TN+FP+FN) = (5+17)/27 = 0.81
- **Recall** = TP/(TP+FN) = 5/(5+3) = 0.62
  Recall gives us an idea about when it's actually yes, how often does it predict yes
- **Precision** = TP/(TP+FP) = 5/(5+2) = 0.71
  Precision tells us about when it predicts yes, how often is it correct
- **F-measure** =
  2*(Recall*Precision)/(Recall+Precision) = 2*(0.62*0.71)/(0.62+0.71) = 0.66

## Car Evaluation Quality

Here we will use the car evaluation quality data set which can be downloaded from the course website. The data set has six features describing the main characteristics of a car; buying price, maintenance cost, number of doors, number of persons to carry, size of luggage boot and safety. The quality of a car can be: inaccurate, accurate, good, and very good.

To evaluate the accuracy of the classification, we will use the precision, recall, and f-measure. Note that in the case of multiple classes, these metrics are usually calculated as many times the number of labels, each time considering a class as the positive and all others as the negative. Different averages over the multiple classes' metrics are then used to estimate the total precision, recall, and f-measure.

We start by loading the necessary libraries.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn import svm
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
```

Then, we load the car data set into a pandas data frame.

```
df = pd.read_csv('data_cars.csv',header=None)
for i in range(len(df.columns)):
   df[i] = df[i].astype('category')
df.head()
```

Output:

```
       0      1  2  3      4     5      6
0  vhigh  vhigh  2  2  small   low  unacc
1  vhigh  vhigh  2  2  small   med  unacc
2  vhigh  vhigh  2  2  small  high  unacc
3  vhigh  vhigh  2  2    med   low  unacc
4  vhigh  vhigh  2  2    med   med  unacc
```

**Categorical feature values**

```
0. Buying:          v-high, high, med, low
1. Maintenance:     v-high, high, med, low
2. Doors:           2, 3, 4, 5-more
3. Persons:         2, 4, more
4. lug_boot:        small, med, big
5. safety:          low, med, high
6. car evaluation:  unacc, acc, good, vgood
```

These are mapped into numbers to be used in the classification algorithms.

```
#map catgories to values
map0 = dict( zip( df[0].cat.categories, range( len(df[0].cat.categories ))))
map1 = dict( zip( df[1].cat.categories, range( len(df[1].cat.categories ))))
map2 = dict( zip( df[2].cat.categories, range( len(df[2].cat.categories ))))
map3 = dict( zip( df[3].cat.categories, range( len(df[3].cat.categories ))))
map4 = dict( zip( df[4].cat.categories, range( len(df[4].cat.categories ))))
map5 = dict( zip( df[5].cat.categories, range( len(df[5].cat.categories ))))
map6 = dict( zip( df[6].cat.categories, range( len(df[6].cat.categories ))))

cat_cols = df.select_dtypes(['category']).columns
df[cat_cols] = df[cat_cols].apply(lambda x: x.cat.codes)
df = df.iloc[np.random.permutation(len(df))]

print(df.head())
```

Output:

```
      0  1  2  3  4  5  6
272   3  2  2  0  2  0  2
1257  2  1  2  1  0  1  2
1120  2  2  1  1  1  2  0
1152  2  2  2  2  2  1  2
411   3  1  3  0  0  1  2
```

Note that the numbers in the first column are the sample numbers of the data set (1728 samples)

Since we need to calculate and save the measures for all the methods, we write a standard function CalcMeasures and divide the label vector Y from the features X.

```
df_f1 = pd.DataFrame(columns=['method']+sorted(map6, key=map6.get))
df_precision = pd.DataFrame(columns=['method']+sorted(map6, key=map6.get))
df_recall = pd.DataFrame(columns=['method']+sorted(map6, key=map6.get))

def CalcMeasures(method,y_pred,y_true,df_f1=df_f1,df_precision=df_precision,
        df_recall=df_recall):

    df_f1.loc[len(df_f1)] = [method]+list(f1_score(y_true,y_pred,average=None))

    df_precision.loc[len(df_precision)] =
                [method]+list(precision_score(y_pred,y_true,average=None))

    df_recall.loc[len(df_recall)] = [method]+list(recall_score(y_pred,
                y_true,average=None))

X = df[df.columns[:-1]].values

Y = df[df.columns[-1]].values
```

A 10 folds cross validation has been used.[3]

```
cv = 10
method = 'linear support vector machine'
clf = svm.SVC(kernel='linear',C=50)
y_pred = cross_val_predict(clf, X,Y, cv=cv)
CalcMeasures(method,y_pred,Y)

method = 'naive bayes'
clf = MultinomialNB()
y_pred = cross_val_predict(clf, X,Y, cv=cv)
CalcMeasures(method,y_pred,Y)

method = 'logistic regression'
clf = LogisticRegression()
y_pred = cross_val_predict(clf, X,Y, cv=cv)
CalcMeasures(method,y_pred,Y)

method = 'k nearest neighbours'
clf = KNeighborsClassifier(weights='distance',n_neighbors=5)
y_pred = cross_val_predict(clf, X,Y, cv=cv)
CalcMeasures(method,y_pred,Y)
```

[3] Executing this script may give some warning when the metric values are 0.0

The measures' values are stored in the data frames.

Output: **df_f1**

```
                            method       acc  good     unacc     vgood
  0  linear support vector machine  0.266667   0.0  0.847084  0.000000
  1                    naive bayes  0.040404   0.0  0.825581  0.000000
  2            logistic regression  0.270903   0.0  0.822967  0.027778
  3           k nearest neighbours  0.793872   0.6  0.948201  0.686869
```

Output: **df_precision**

```
                           method       acc      good     unacc     vgood
0  linear support vector machine  0.177083  0.000000  0.984298  0.000000
1                    naive bayes  0.020833  0.000000  0.997521  0.000000
2            logistic regression  0.210938  0.000000  0.923967  0.015385
3           k nearest neighbours  0.742188  0.478261  0.990909  0.523077
```

Output: **df_recall**

```
                           method       acc      good     unacc     vgood
0  linear support vector machine  0.539683  0.000000  0.743446  0.000000
1                    naive bayes  0.666667  0.000000  0.704201  0.000000
2            logistic regression  0.378505  0.000000  0.741871  0.142857
3           k nearest neighbours  0.853293  0.804878  0.909022  1.000000
```

- The algorithm that performs best is k nearest neighbours. It has a good recall for all the classes, but a bad precision and F1 valu for the classes good and vgood.
- Naive Bayes, logistic regression, and SVM with linear kernel (C=50) return poor models, especially for the good and very good classes because there are few points with those labels.

We can calculate the number of samples in each class.

```
labels_counts=df[6].value_counts()
pd.Series(map6).map(labels_counts)
```

Output:

```
acc       384
good       69
unacc    1210
vgood      65
```

- In percentage the very good (vgood) and good are 3.7% and 3.8% respectively, compared to 70% of inaccurate and 22% of accurate.
- We can conclude that naïve bayes, logistic regression and SVM with linear kernel are not suitable for predicting classes that are scarcely represented in a dataset.