Model Improvement, Grid Search and Metrics

Daniel Hardt

Copenhagen Business School

Copenhagen
Business School
HANDELSHØJSKOLEN

Outline

Fitting Model to Data

- The point is to find models that **generalize** to data beyond training data
- That's why we *split* data into **training** and **test data**
- Test data score gives a better assessment of the model than training data score
- **Cross Validation:** do multiple splits between training and test data to get better assessment of model
- **Grid Search:** search for best parameter values, to get a better model

Cross Validation

- Instead of one train-test split, multiple splits
- For example with Five-fold CV, pick one fifth of data as test, and the other four fifths as training data
- Gives a better basis for assessing model – with one split, might be “lucky” or “unlucky” with test data

Five-fold Cross Validation

In[2]:

```
mglearn.plots.plot_cross_validation()
```

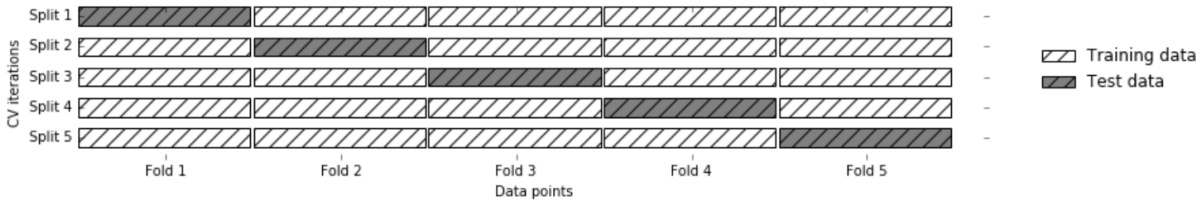


Figure 5-1. Data splitting in five-fold cross-validation

Copenhagen
Business School
HANDELSHØJSKOLEN

Cross Validation in SciKit-Learn

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()

scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```

Out[3]:

```
Cross-validation scores: [0.961 0.922 0.958]
```

Cross Validation in SciKit-Learn

cross_val_score performs

- split of train and test data
- fits model to train data
- scores model on test data

Assessing vs Improving Models

- Cross validation is simply a method to assess a model – does nothing to improve the model
- Grid search is a way to improve your model
- Combines well with cross validation

Example: Tuning an SVM

C controls regularization – higher values mean *less* regularization, just as with logistic regression

Gamma controls complexity of model in a different way

> A low value of **gamma** means that the decision boundary will vary slowly, which yields a model of low complexity, while a high value of **gamma** yields a more complex model.

(p 102 in text)

Example: Tuning an SVM

- Want to try these values for both gamma and C: .001, .01, 0.1, 1, 10, 100
- Nested **for loop** to try each of 36 combinations

Example: Tuning an SVM

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the test set
        score = svm.score(X_test, y_test)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters))
```

Out[19]:

```
Size of training set: 112   size of test set: 38
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}
```

Why We Need a Validation Set

- It's wrong to tune a model using scores from test set
- This won't give valid indication of how model generalizes
- Need to define a separate Validation Set which is used to tune model
- Scores on test data are only given once tuning is finished, and best model is selected

A Three-Way Split of Data

Train, Validation, and Test

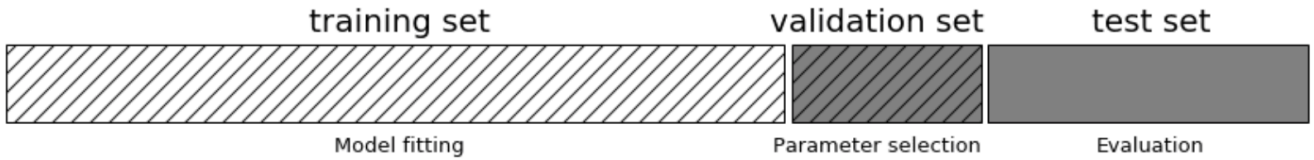


Figure 5-5. A threefold split of data into training set, validation set, and test set

Three-Way Split in Scikit

```
# split data into train+validation set and test set
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# split train+validation set into training and validation
sets
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
```

Three-Way Split in Scikit

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the validation set
        score = svm.score(X_valid, y_valid)
        # if we got a better score, store the score and
parameters
        if score > best_score:
            best_score = score
```

Three-Way Split in Scikit

```
                best_parameters = {'C': C, 'gamma': gamma}

# rebuild a model on the combined training and validation
set,
# and evaluate it on the test set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)
```

Note on kwargs

- best_parameters = {'C': C, 'gamma' : gamma}
- Define *best_parameters* as a *dict*
- svm = SVC(best_parameters)
- Can pass a *dict* to a function expecting *keyword arguments*

Grid Search With Cross-Validation

- Model results can be very sensitive to how data is split
- Rather than use grid search with one split, can use it together with cross-validation for a better tuning process

Grid Search With Cross-Validation

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters,
        # train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

Grid Search With Cross-Validation

- Use *cross_val_score* instead of *score*
- Take *mean* of scores returned by *cross_val_score*
- Can use GridSearchCV class to implement this

Grid Search With Cross-Validation: Overview

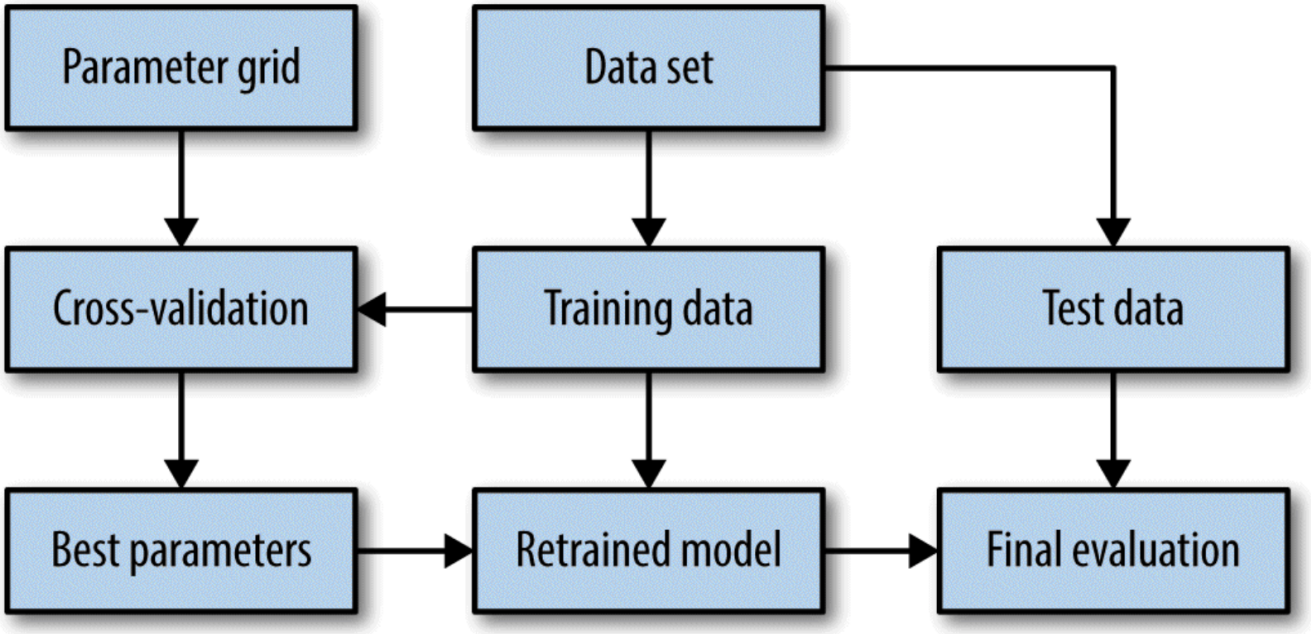


Figure 5-7. Overview of the process of parameter selection and model evaluation with GridSearchCV

Grid Search With Cross-Validation: Sci-kit

```
 Parameter grid:
 {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma':
 [0.001, 0.01, 0.1, 1, 10, 100]}
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5,
                           return_train_score=True)
  X_train, X_test, y_train, y_test = train_test_split(
      iris.data, iris.target, random_state=0)
```

Grid Search With Cross-Validation: Sci-kit

```
grid_search.fit(X_train, y_train)
  print("Test set score:
  {:.2f}".format(grid_search.score(X_test, y_test)))
```

Tuning – Summing Up

Tuning on Validation Set

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the validation set
        score = svm.score(X_valid, y_valid)
        # if we got a better score, store the score and
parameters
        if score > best_score:
            best_score = score
```

Tuning on Validation Set using Cross-Validation

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters,
        # train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

Tuning using Grid Search with Cross-Validation

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma':
[0.001, 0.01, 0.1, 1, 10, 100]}
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5,
                           return_train_score=True)
```

Takeaways

- Split train and test data to assess a model
- **Cross validation** is a better way to assess a model
- **Grid Search** improves model by finding best hyperparameter values
- Need three-way split of data: train, validation and test
- Can combine Grid Search and Cross Validation

Basic Metrics

Classification: Accuracy

$$\frac{\textit{Correctly classified samples}}{\textit{Total number of samples}}$$

Basic Metrics

Regression: R^2

Coefficient of Determination

$$\frac{ExplainedVariation}{TotalVariation}$$

Business Impact

What is the goal? What is the business impact of using the model?

Business Impact

Many different metrics for models

Confusion Matrix

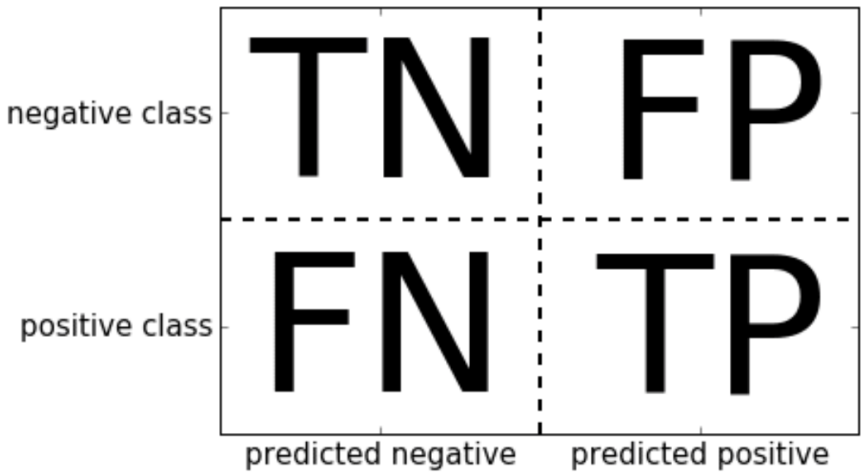


Figure 5-11. Confusion matrix for binary classification

Accuracy

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

Precision

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

Recall

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

F Score

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Digit Classification

Convert **digits** into unbalanced binary classification task

```
y = digits.target == 9
```

Target value *y* is **true** if digits.target is 9, **false** otherwise

Unbalanced: 10% of target values are **true**, 90% **false**

Digit Classification – DummyClassifier

```
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train
, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
print("Unique predicted labels:
{}".format(np.unique(pred_most_frequent)))
print("Test score:
{:.2f}".format(dummy_majority.score(X_test, y_test)))
```

Out[42]:

```
Unique predicted labels: [False]
Test score: 0.90
```

Confusion Matrices

```
Most frequent class:
[[403   0]
 [ 47   0]]

Dummy model:
[[361  42]
 [ 43   4]]

Decision tree:
[[390  13]
 [ 24  23]]

Logistic Regression
[[401   2]
 [  8  39]]
```

Classification Report: Dummy

Dummy Classifier: Most Frequent

	precision	recall	f1-score	support
not nine	0.90	1.00	0.94	403
nine	0.00	0.00	0.00	47
micro avg	0.90	0.90	0.90	450
macro avg	0.45	0.50	0.47	450
weighted avg	0.80	0.90	0.85	450

Classification Report: Logreg

Logistic Regression Classifier

	precision	recall	f1-score	support
not nine	0.98	1.00	0.99	403
nine	0.95	0.83	0.89	47
micro avg	0.98	0.98	0.98	450
macro avg	0.97	0.91	0.94	450
weighted avg	0.98	0.98	0.98	450

Classification and Thresholds

Linear Classifier

- $\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$

Classifiers compute a value that is compared against threshold – for linear classifiers, default is 0

Lower Default Threshold

```
y_pred_lower_threshold = svc.decision_function(X_test) > -.8

print(classification_report(y_test, y_pred_lower_threshold))
```

Default Threshold (0)

	precision	recall	f1-score	support
0	0.97	0.89	0.93	104
1	0.35	0.67	0.46	9
micro avg	0.88	0.88	0.88	113
macro avg	0.66	0.78	0.70	113
weighted avg	0.92	0.88	0.89	113

Lower Threshold (-.8)

	precision	recall	f1-score	support
0	1.00	0.82	0.90	104
1	0.32	1.00	0.49	9
micro avg	0.83	0.83	0.83	113
macro avg	0.66	0.91	0.69	113
weighted avg	0.95	0.83	0.87	113

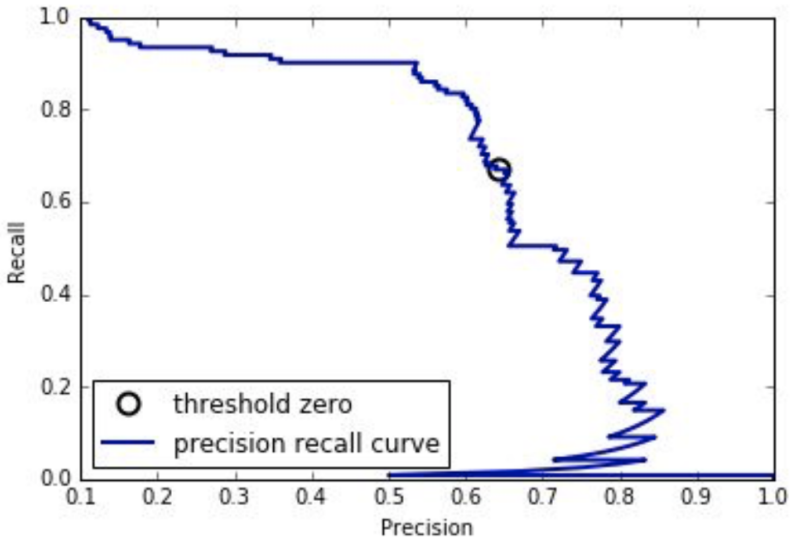
Precision-Recall Curve

- Tradeoff between precision and recall
- Can explore this with different thresholds
- Precision_recall_curve gives precision and recall values for different threshold values

Precision-Recall Curve

```
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
```

Precision-Recall Curve



ROC Curve

- Plots True Positive Rate (recall) against False Positive Rate
- Best values are higher (more true positives) and to the left (fewer false positives)

False Positive Rate

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

True Positive Rate (Recall)

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

ROC Curve

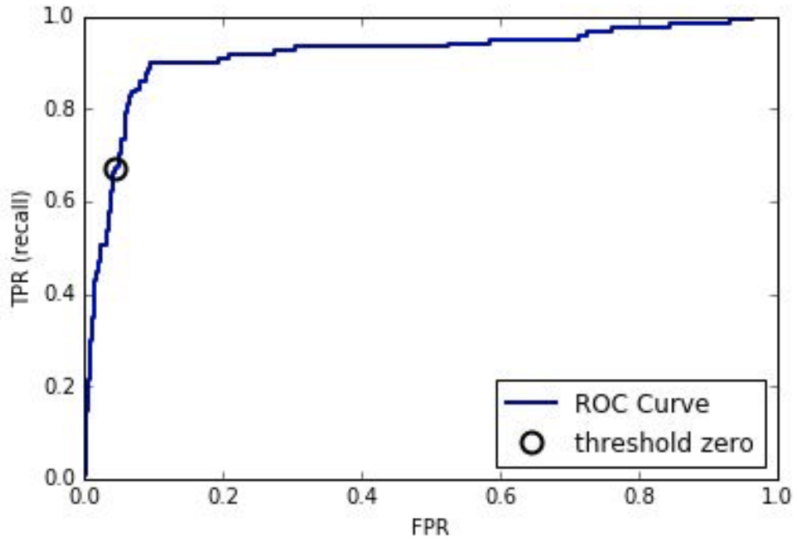


Figure 5-15. ROC curve for SVM

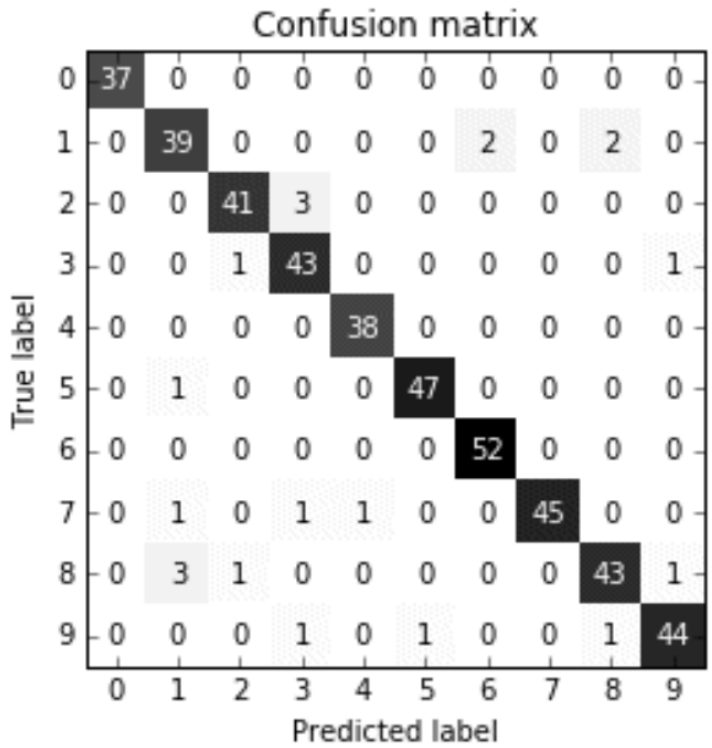
AUC

- AUC is Area Under (ROC) Curve
- Single number to summarize ROC curve
- Ranges from 0 to 1
- Random guessing always gives 0.5, even with unbalanced datasets
- Can be more revealing with unbalanced data

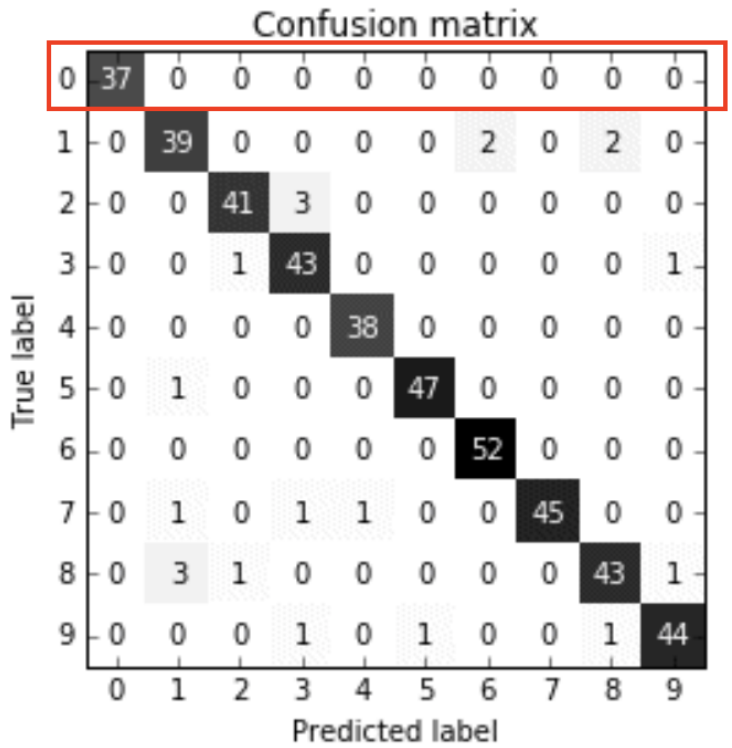
Metrics for Multiclass Classification

```
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
```

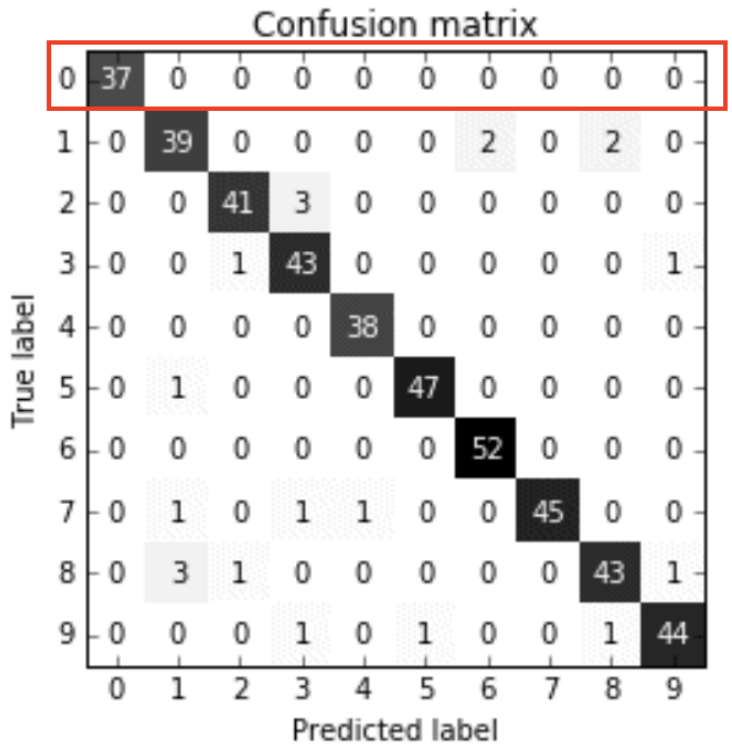
Metrics for Multiclass Classification



Metrics for Multiclass Classification



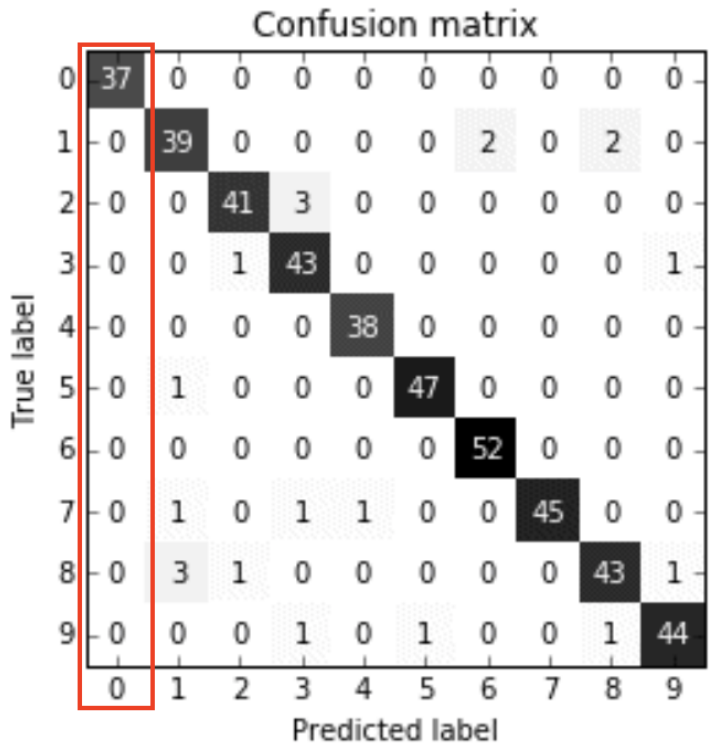
Metrics for Multiclass Classification



Recall: 37 / 37 = 1.00

Copenhagen Business School
HANDELSHØJSKOLEN

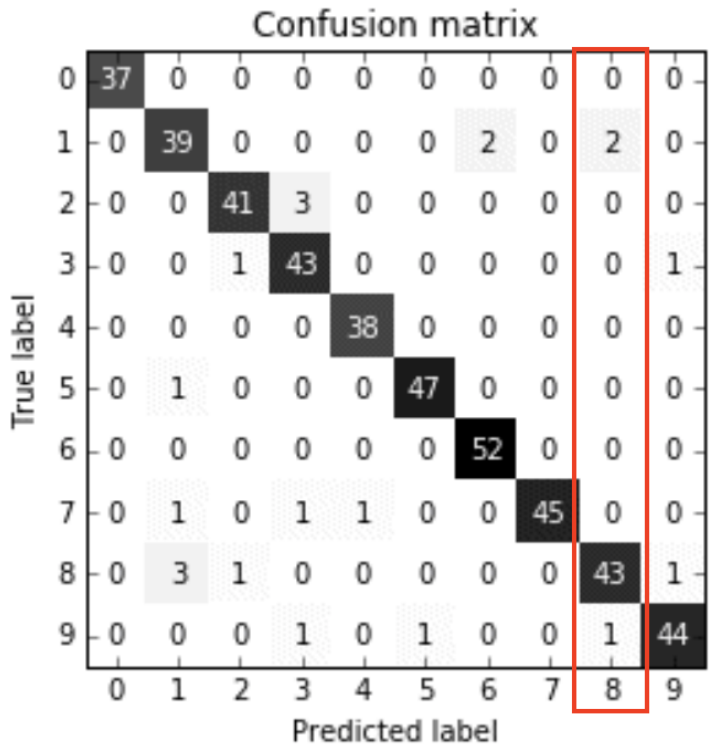
Metrics for Multiclass Classification



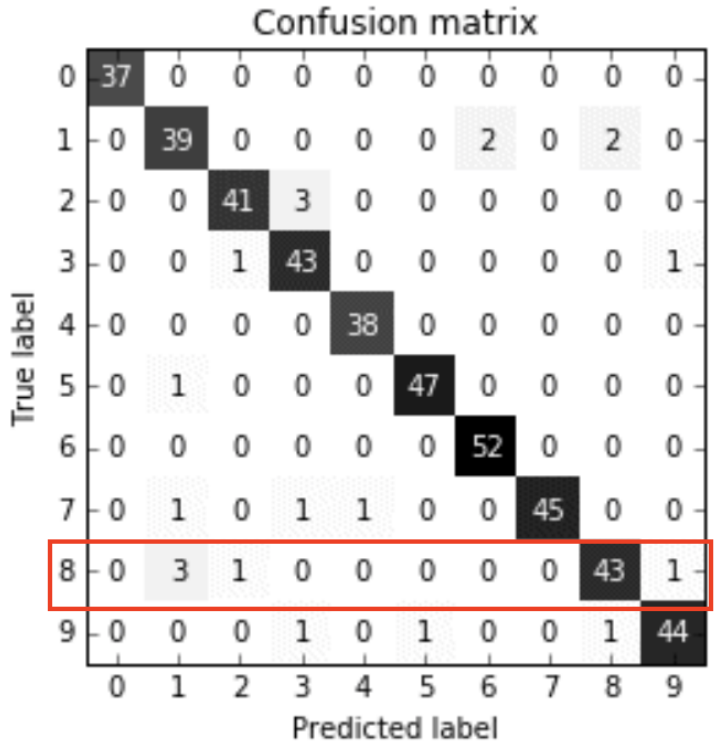
Precision: 37 / 37 = 1.00

Copenhagen Business School
HANDELSHØJSKOLEN

Metrics for Multiclass Classification



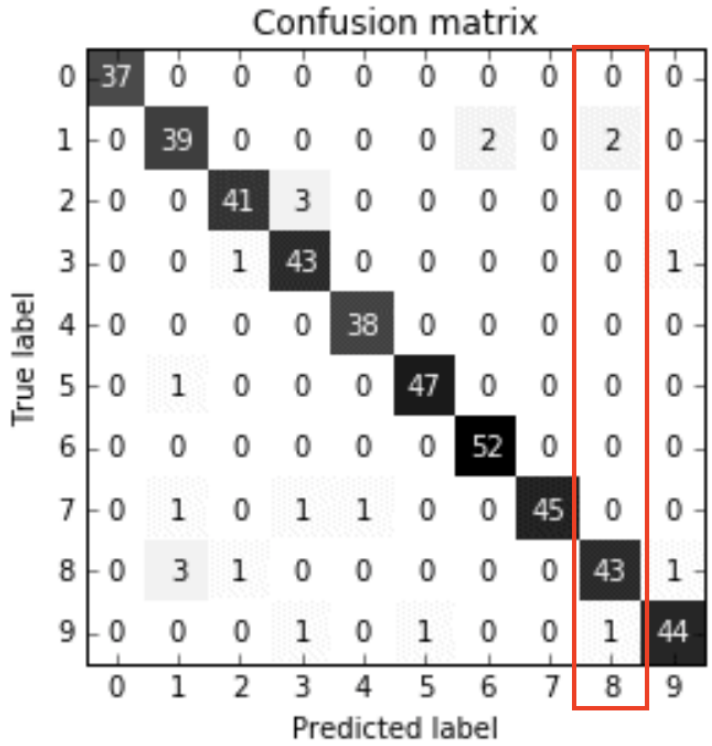
Metrics for Multiclass Classification



Recall: 43 / 48 = .90

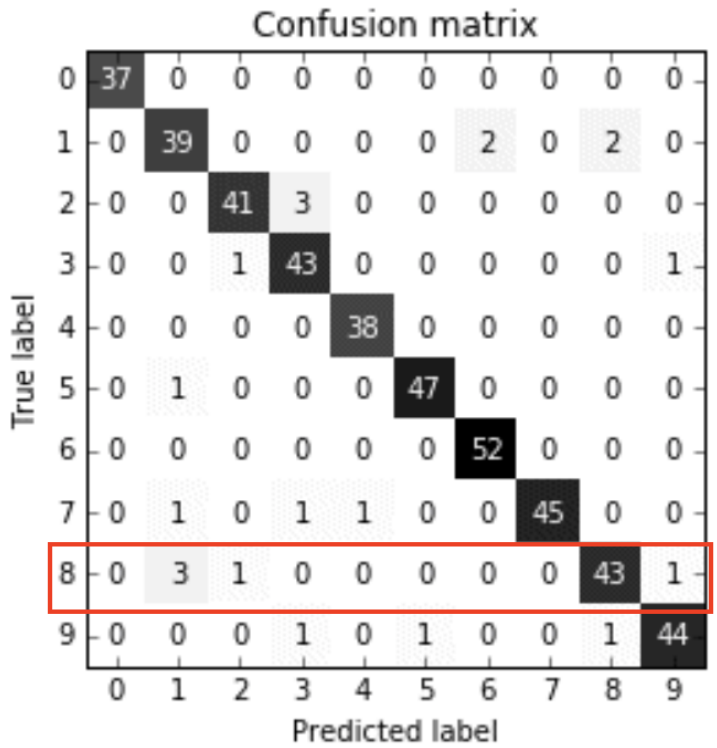
Copenhagen Business School
HANDELSHØJSKOLEN

Metrics for Multiclass Classification



Precision: 43 / 46 = .93

Metrics for Multiclass Classification



Metrics for Multiclass Classification

	precision	recall	f1-score	support
0	1.00	1.00	1.00	37
1	0.89	0.91	0.90	43
2	0.95	0.93	0.94	44
3	0.90	0.96	0.92	45
4	0.97	1.00	0.99	38
5	0.98	0.98	0.98	48
6	0.96	1.00	0.98	52
7	1.00	0.94	0.97	48
8	0.93	0.90	0.91	48
9	0.96	0.94	0.95	47

Metrics for Multiclass Classification

- Macro Average: take average of each class value (classes are treated equally, even if unbalanced)
- Micro Average: take average of each class value (values for frequent classes get more emphasis)
- Use **Macro** if you care about each class equally – if you are focused on number of instances, use **Micro**

Using Metrics in Model Selection

- We select models by tuning hyperparameters
- Use GridSearchCV, cross_val_score
- By default, *accuracy* is optimized
- Can change this to other metrics, such as *average precision*

Using Metrics in Model Selection

```
# using the default scoring of accuracy:
grid = GridSearchCV(SVC(), param_grid=param_grid)
grid.fit(X_train, y_train)
print("Grid-Search with accuracy")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (accuracy)):
{:.3f}".format(grid.best_score_))
```

Using Metrics in Model Selection

```
# using AUC scoring instead:
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="average_precisi
on")
grid.fit(X_train, y_train)
print("Grid-Search with average precision")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (average precision):
{:.3f}".format(grid.best_score_))
```

Using Metrics in Model Selection

```
Available scorers:
['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score',
'average_precision',
 'balanced_accuracy', 'brier_score_loss', 'completeness_score',
'explained_variance',
 'f1', 'f1_macro', 'f1_micro', 'f1_samples', 'f1_weighted',
'fowlkes_mallows_score',
 'homogeneity_score', 'mutual_info_score', 'neg_log_loss',
'neg_mean_absolute_error',
 'neg_mean_squared_error', 'neg_mean_squared_log_error',
'neg_median_absolute_error',
 'normalized_mutual_info_score', 'precision', 'precision_macro',
'precision_micro',
 'precision_samples', 'precision_weighted', 'r2', 'recall',
'recall_macro',
 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc',
 'v_measure_score']
```

Takeaways

Metrics for Evaluation

- There are lots of ways to evaluate models
- We should select a metric that corresponds to the **goals** and **business impact** of building the model
- Can explore thresholds for classifier models
- We select models by **tuning hyperparameters** – can do this wrt. the metric best suited to your case