Chapter 2: Supervised Learning & Cross Validation

IMLP 2.1-2.2
APM 4.1-4.3
IMLP 5.1-5.2

Table of Contents

Supervised Learning

- Supervised machine learning is one of the most commonly used and successful types of machine learning today
- Supervised learning is used whenever we want to predict a certain outcome from a given input
- We have examples of input/output pairs

2.1 Types of Supervised Learning

- There are two types of supervised learning

  Classification (binary, multiclass)

  Regression

Classification

- the goal is to predict a class label, which is a choice from a predefined list of possibilities
- ***binary classification***, is a special case of distinguishing between exactly two classes
- ***multiclass classification***, which is classification between more than two classes

Classification

- You can think of binary classification as trying to answer a yes/no question
- Classifying emails as either spam or not spam is an example of a binary classification problem
- In this binary classification task, the yes/no question being asked would be “Is this email spam?
- The iris example, on the other hand, is an example of a multiclass classification problem

Regression

- the goal is to predict a continuous number, or a floating-point number in programming terms

- Predicting a person's annual income from their education, their age, and where they live is an example of a regression task

Generalization, Overfitting, and Underfitting

- If a model is able to make accurate predictions on unseen (test) data, we say it is able to generalize from the training set to the test set
- We want to build a model that is able to generalize as accurately as possible
- For example, if we allow ourselves to build very complex models, we can always be as accurate as we like on the training set but still it may perform badly on test set!
- Trying to recognize images using elephants as an example

Generalization- Buying a boat-overfitting

- Here we have a data set whose features (also called predictors) are

Age, Number of Cars Owned, Owns House, Number of Children, Marital Satus, Owns a Dog.

- The target (outcome) is if the person Bought a boat.

Table 2-1. Example data about customers

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

Generalization- Buying a boat-overfitting

No age appears twice in the data, so we could say people who are 66, 52, 53, or 58 years old want to buy a boat, while all others don't. Would this be correct generalization? NO.

This is an example of OVERFITTING

(works well on training, not on test cases)

Generalization- Buying a boat

- If the rule was “People older than 50 want to buy a boat,” and this would explain the behavior of all the customers, we would trust it more than the rule involving children and marital status in addition to age
- Note that there is a 64 age who bought the boat. But still this would generalize better
- Therefore, we always want to find the simplest model

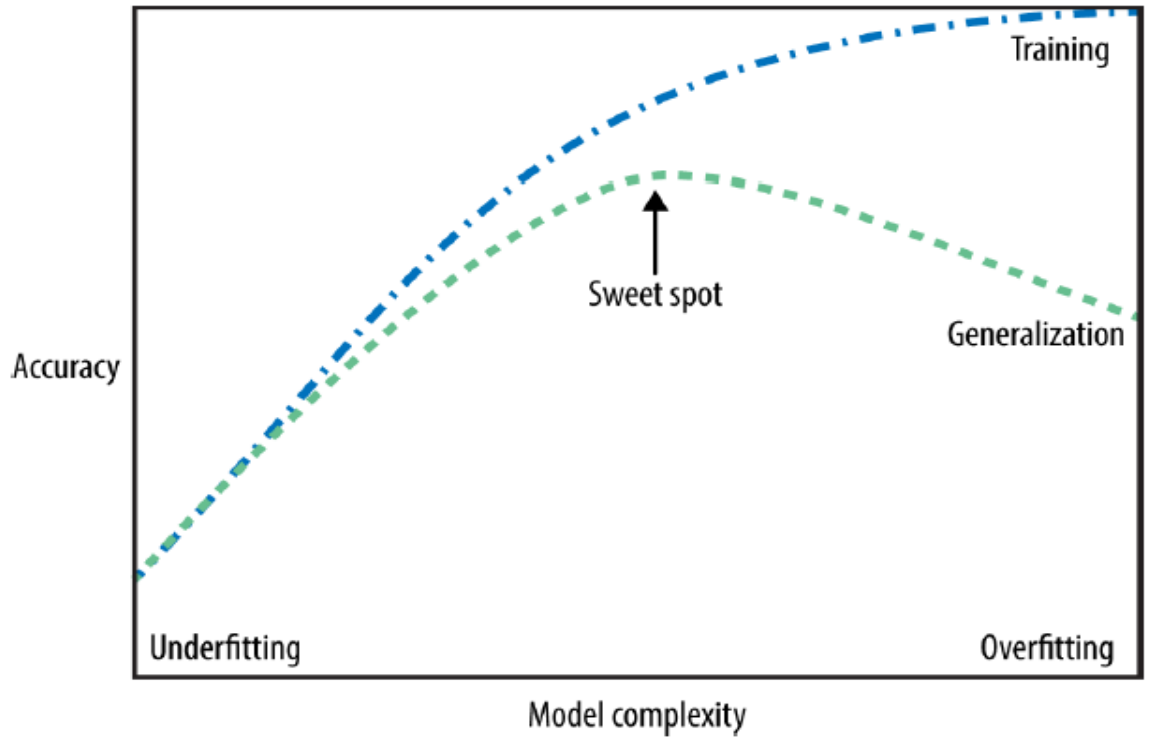
Generalization- Buying a boat -Underfitting

- On the other hand, if your model is too simple—say, “Everybody who owns a house buys a boat”—then you might not be able to capture all the aspects of and variability in the data, and your model will do badly even on the training set

- There is a sweet spot in between that will yield the best generalization performance. What is the sweet spot?

Generalization, Overfitting, and Underfitting

There is a sweet spot in between that will yield the best generalization performance. This is the model we want to find



ıre 2-1. Trade-off of model complexity against training and test accuracy

Generalization, Overfitting, and Underfitting

- It's important to note that ***model complexity is intimately tied to the variation of inputs*** contained in your training dataset: the larger variety of data points your dataset contains, the more complex a model you can use without overfitting

- Usually, collecting more data points will yield more variety, so larger datasets allow building more complex models

- In the real world, you often have the ability to decide how much data to collect, which might be more beneficial than tweaking and tuning your model. ***Never underestimate the power of more data***.

Supervised Learning Algorithms

- We use `P` for predictor (that does not include the target column) and `y` for the target column. Some books might use X for data.
- The main goal is to build the mathematical that when a row from `P` will predict corresponding `y` as closely as possible.
- If the model is developed assuming a linear relationship between `P` and `y`, then we call it ***Linear Model***. There are several variations of linear model.
- If the model is developed assuming non-linearity in the relationship between `P` and `y`, then we have a ***non-linear model***.
- If the model is based on tree structure is used to make predictions, we have a ***tree-based model***. ***(rule-based model is also included here)***

Linear models

Linear Models (ordinary least squares)

- Linear Regression
- Linear Classification
- Lasso Regression
- Ridge Regression
- Logistic Regression (Classification, misnomer)

Non-linear Models

- Neural Networks for Classification
- Neural Networks for Regression
- K-Nearest Neighbor for classification(k-NN)
- K-Nearest Neighbor for regression(k-NN)
- Multivariate Adaptive Regression Splines (MARS – NOT in scikit-learn)
- Support Vector Machines (SVMs)

Tree Based Models

- Single Trees (Regression and Classification)
- Rule Based Tree Models
- Bagged Trees
- Random Forests
- Gradient Boosted Trees

k-NN

It is based on the idea that the outcome for a data is best predicted by the outcome “nearest” to it.

(For example if data points that are near the given the sepal length, sepal width, petal width and petal length are all of type verginica, then possibly the given data point is virginica)

Three important decision to be made for k-NN

1. How do we define “nearest”?
2. How many nearest neighbors should be examined?
3. How do we find cumulative result?

K-NN Model

The k-NN algorithm is arguably the simplest machine learning algorithm, even though it is a non-linear model. The distance between two samples is written as:

$$\left(\sum_{j=1}^{P} |x_{aj} - x_{bj}|^q \right)^{\frac{1}{q}},$$

P is number of predictors/features, x_{aj} and x_{bj} are a^{th} and b^{th} row in data matrix of features. j is an index on the number of predictors. The || and q represent the type of distance measurement used.

K-NN Model

We can use the following for distance measurement

$$\left(\sum_{j=1}^{P} (x_{aj} - x_{bj})^2 \right)^{\frac{1}{2}},$$

When q = 2 we get Euclidean distance (Minkowski), when q=1 we get Manhattan distance; Other distances like Hamming distance etc. can be used.

K-NN Model predictions for classification.

- For a new sample (test set)- the k nearest neighbors are found. Then majority of class values is chosen for predicting the class for test set.
- For training set prediction, the distance between training sample and all other training rows are computed and top k are chosen and again majority vote is used for prediction of training data (to compute training set accuracy)
- The value of training score would always be 1 for classification if using 1-NN (why)

Computation

https://scikit-learn.org/stable/modules/classes.html#

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, *,
weights='uniform',  algorithm='auto', leaf_size=30, p=2,
metric='minkowski', metric_params=None,  n_jobs=None, **kwargs)
```

n_neighbors = *number of neighbors to consider*
weights = uniform means all points are considered equally
algorithm = auto (best algorithm used for finding the nearest neighbors among ballTree, kDTree, bruteForce etc)
leaf_size = 30 (leaf size for the algorithms used in searching for neighbors)
metric = "Minkowski"
p=2 power parameter for minkowski metric
metric_params = None
n_jobs: Number of parallel jobs to do when computing nearest neighbor tree

Computation

Methods: (Note scikit learn uses X for the predictor matrix P

fit(X, y)	Fit the k-nearest neighbors classifier from the training dataset.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(X)	Predict the class labels for the provided data.
predict_proba(X)	Return probability estimates for the test data X.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

Computation

```
clf = KNeighborsClassifier(n_neighbors=n_neighbors)
clf.fit(X_train,y_train)
print("Test set predictions:", clf.predict(X_test))
```

Output:

```
Test set predictions: [1 0 1 0 1 0 0]
```

```
print("Test set accuracy: {:.2f}".format(clf.score(X_test,
y_test)))
```

Output:

```
Test set accuracy: 0.86
```

Entire code is available by the author in the file:

02-supervised-learning.ipynb

Accuracy Measure for Classification

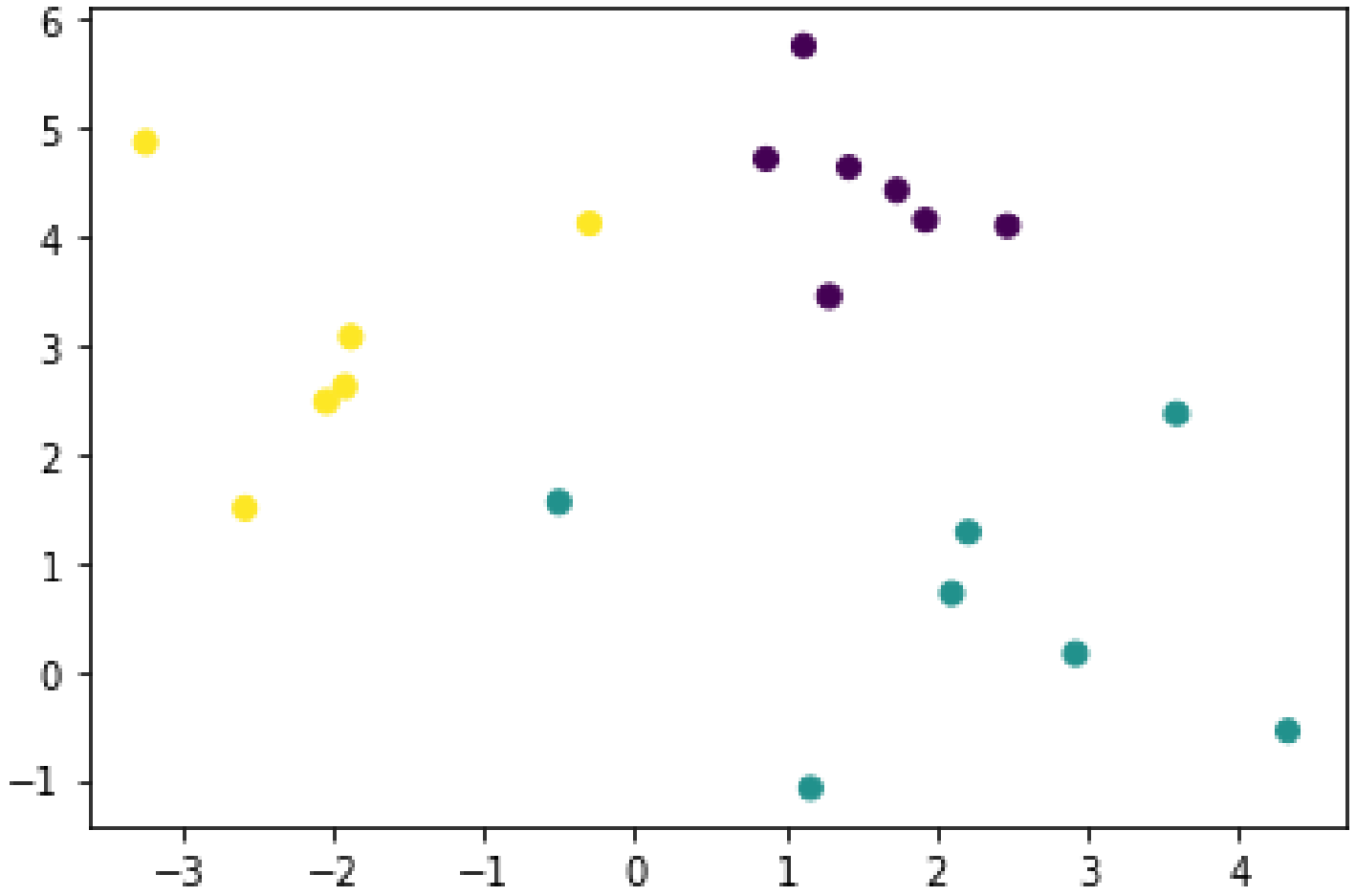
- Scikit learn will use score for classification and regression.
- In the classification case, the score refers to accuracy, i.e percentage of times the model predicted the correct class for the input.

make_blobs Dataset from scikit-learn

```
@author: skanchi
from sklearn.datasets import make_blobs
import numpy as np
import matplotlib.pyplot as plt

X, y = make_blobs(n_samples=20, centers=3,
n_features=2,random_state=0)
plt.scatter(X[:,0],X[:,1],c=y)
plt.show()
```

make_blobs Dataset from scikit-learn



mglearn make_forge Dataset

```
def make_forge():
    # a carefully hand-designed dataset lol
    X, y = make_blobs(centers=2, random_state=4, n_samples=30)
    y[np.array([7, 27])] = 0
    mask = np.ones(len(X), dtype=np.bool)
    mask[np.array([0, 1, 5, 26])] = 0
    X, y = X[mask], y[mask]
    return X, y
```

```
X, y = mglearn.datasets.make_forge()
```

k-NN (k=1) on Forge Data Set

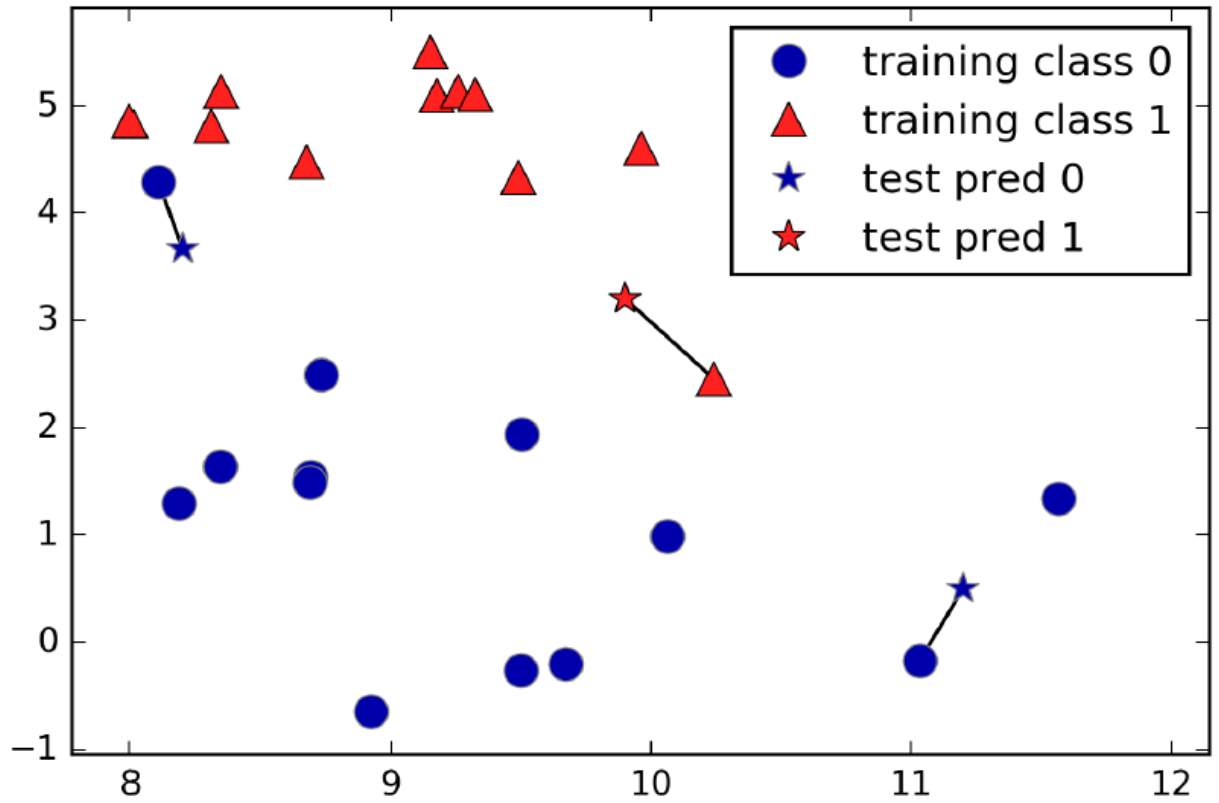


Figure 2-4. Predictions made by the one-nearest-neighbor model on the forge dataset

k-NN (k=3) on Forge Data Set

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

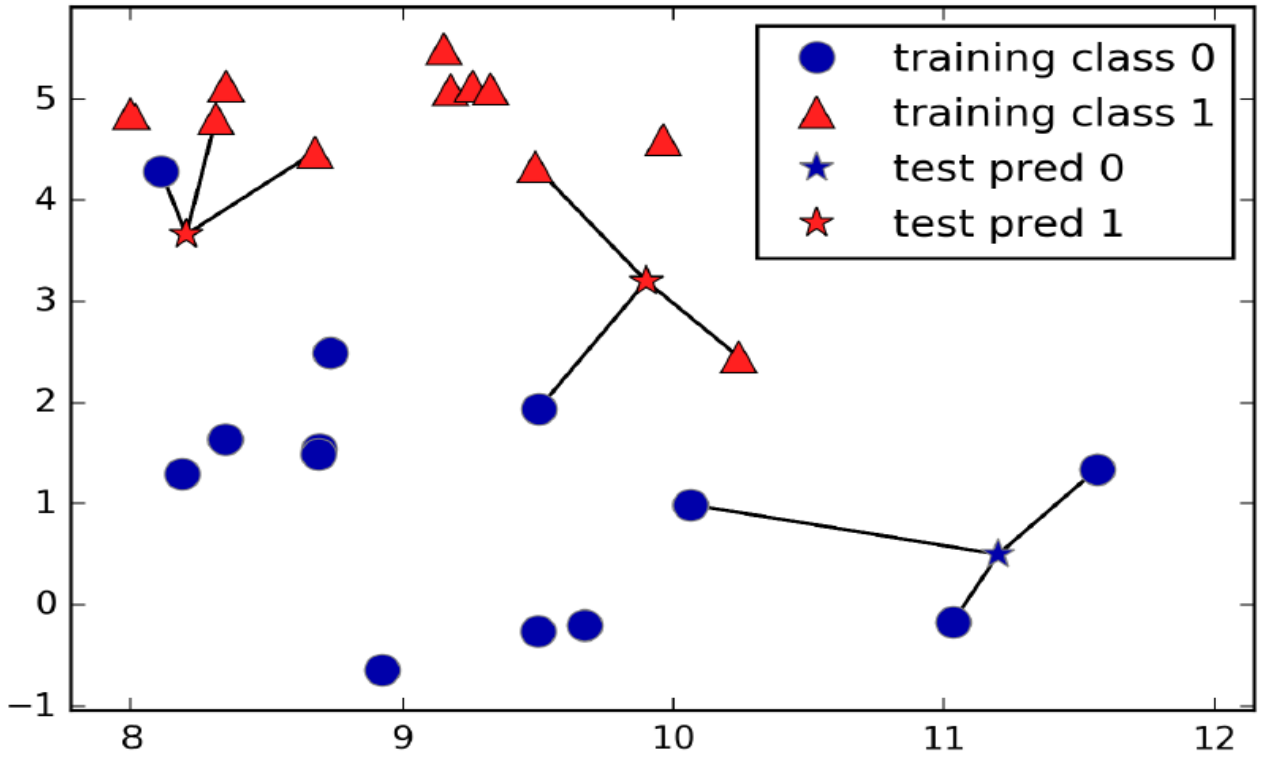


Figure 2-5. Predictions made by the three-nearest-neighbors model on the forge dataset

k-NN on Forge Data Set

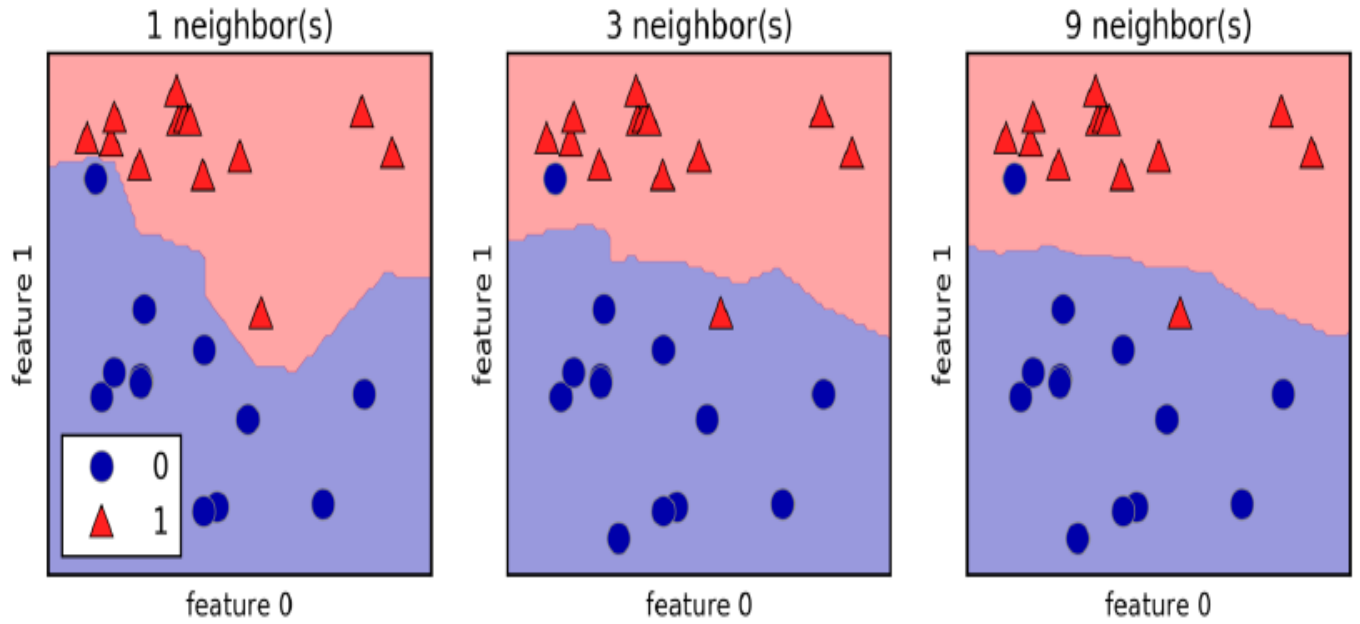


Figure 2-6. Decision boundaries created by the nearest neighbors model for different values of n_neighbors

Decision Boundary Drawing

To draw the decision boundary several points are chosen in the range of x and y and the model is used to predict the class. Then the following function is used to do the contour drawing.

```
h = .02 # step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cmap_light)
```

k-NN on Forge Data Set

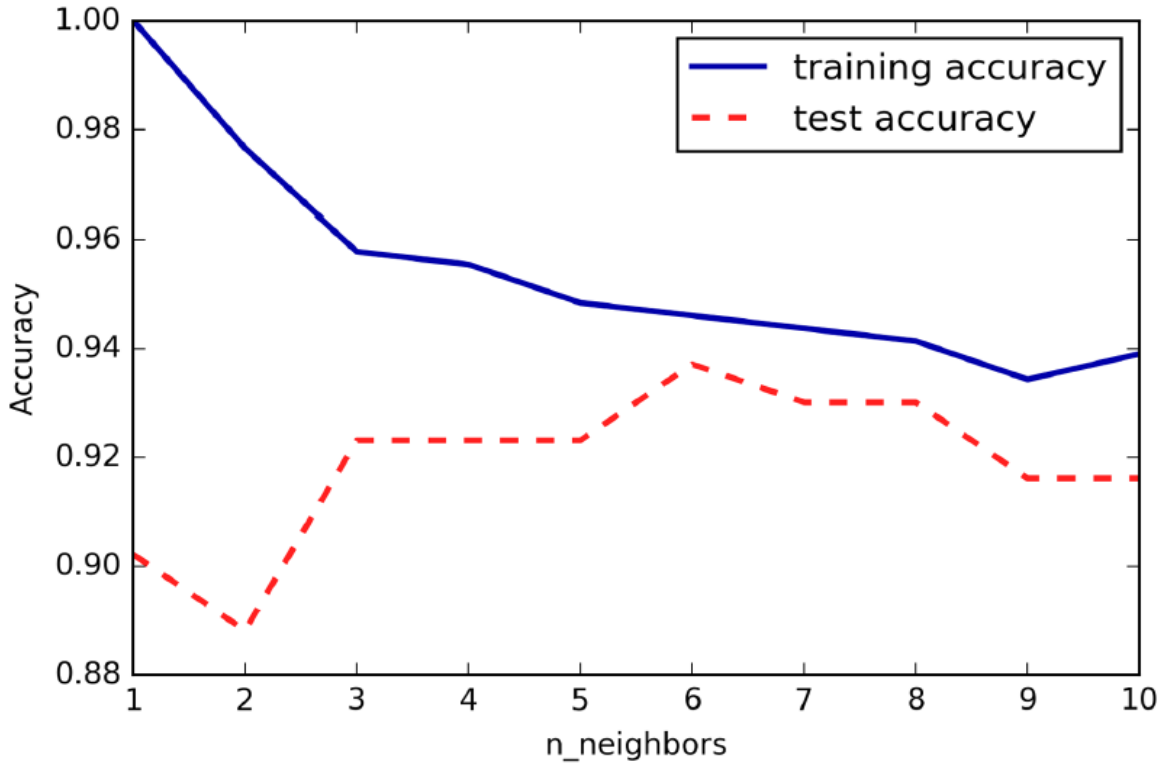


Figure 2-7. Comparison of training and test accuracy as a function of n_neighbors

K-NN:Strengths and Weaknesses

- In practice, using a small number of neighbors like three or five often works well, but you should certainly adjust this parameter
- One of the strengths of k-NN is that the model is very easy to understand, and often gives reasonable performance without a lot of adjustments
- Building the nearest neighbors model is usually very fast, but when your training set is very large prediction can be slow
- When using the k-NN algorithm, it’s important to preprocess your data

K-NN: Strengths and Weaknesses

- This approach often does not perform well on datasets with many features , and it does particularly badly with datasets where most features are 0 most of the time
- So, while the k-nearest neighbors algorithm is easy to understand, it is not often used in practice, due to prediction being slow and its inability to handle many features

make_wave Dataset from scikit-learn

```
def make_wave(n_samples=100):
    rnd = np.random.RandomState(42)
    x = rnd.uniform(-3, 3, size=n_samples)
    y_no_noise = (np.sin(4 * x) + x)
    y = (y_no_noise + rnd.normal(size=len(x))) / 2
    return x.reshape(-1, 1), y
```

```
X, y = mglearn.datasets.make_wave(20)
```

k-NN for Regression- Wave Data Set

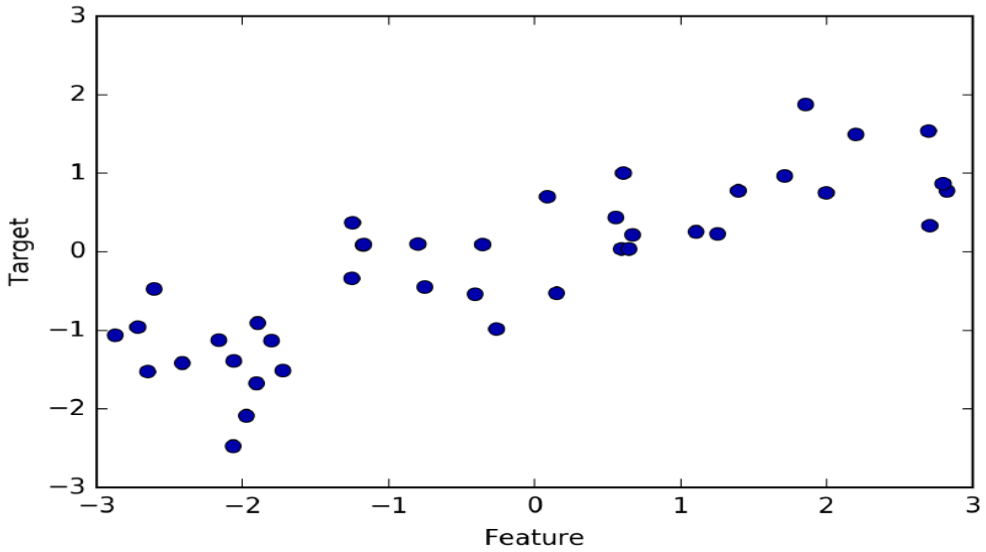


Figure 2-3. Plot of the wave dataset, with the x-axis showing the feature and the y-axis showing the regression target

k-NN (k=1)for Regression- Wave Data Set

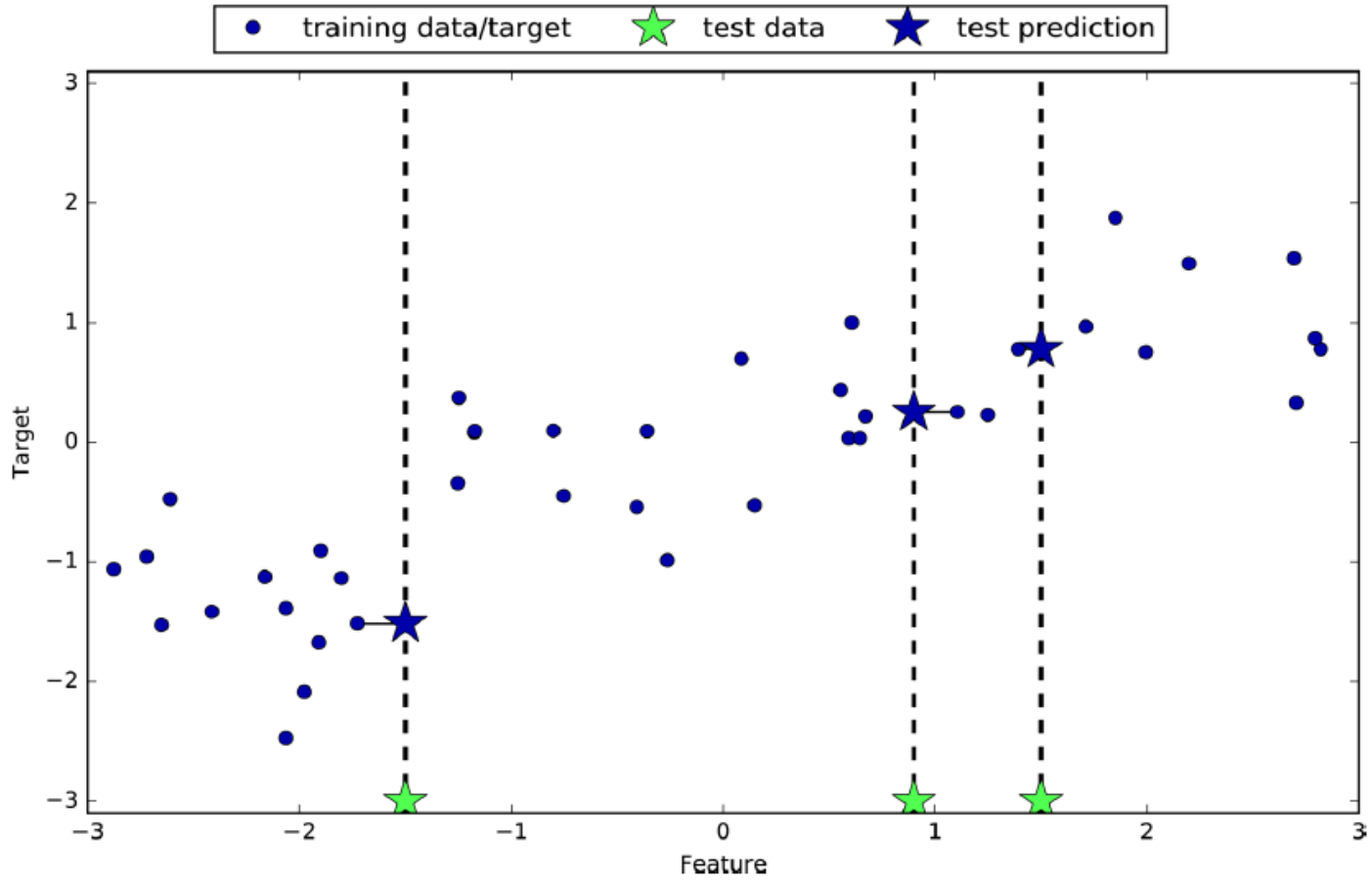


Figure 2-8. Predictions made by one-nearest-neighbor regression on the wave dataset

k-NN (k=3) for Regression- Wave Data Set

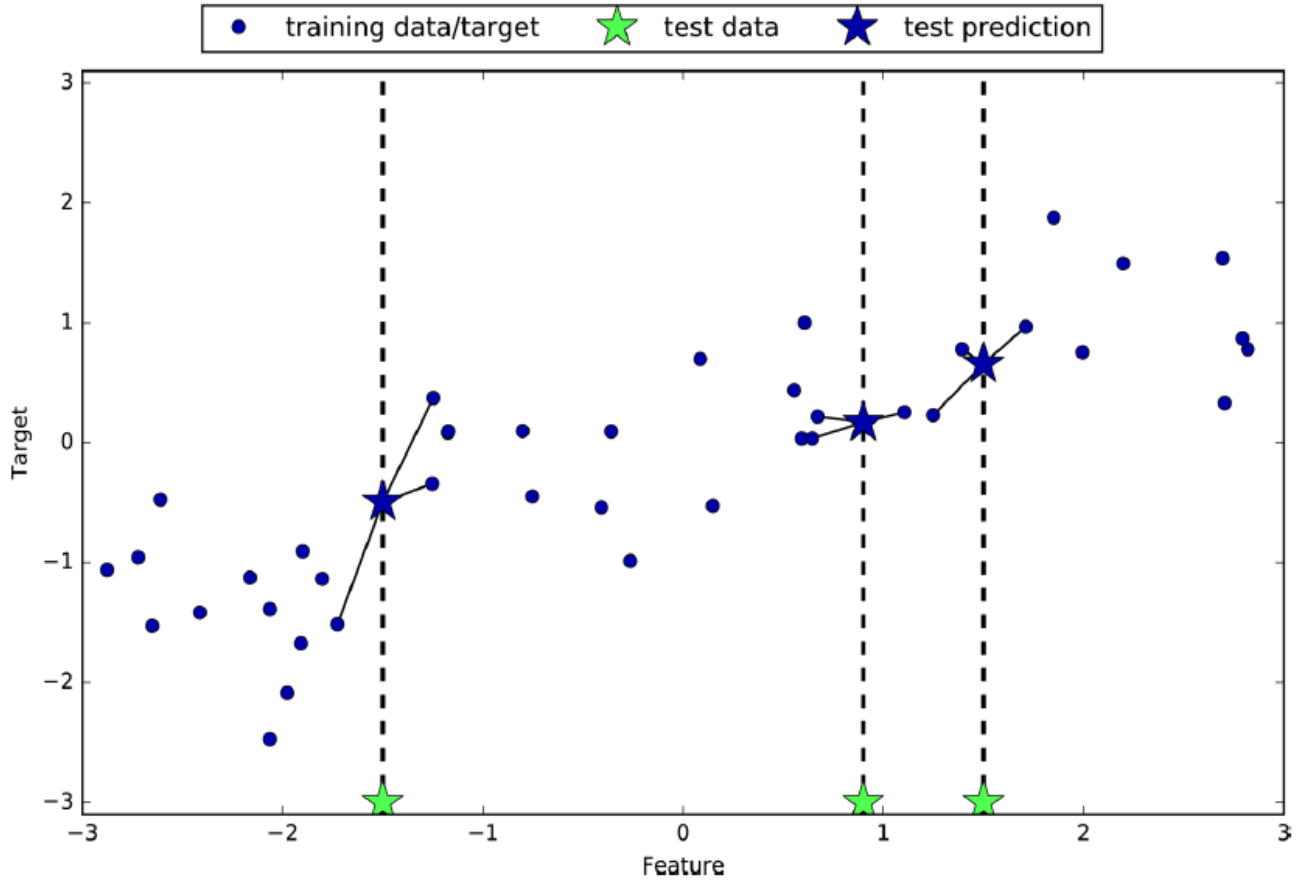


Figure 2-9. Predictions made by three-nearest-neighbors regression on the wave dataset

k-NN Comparison Regression- Wave Data Set

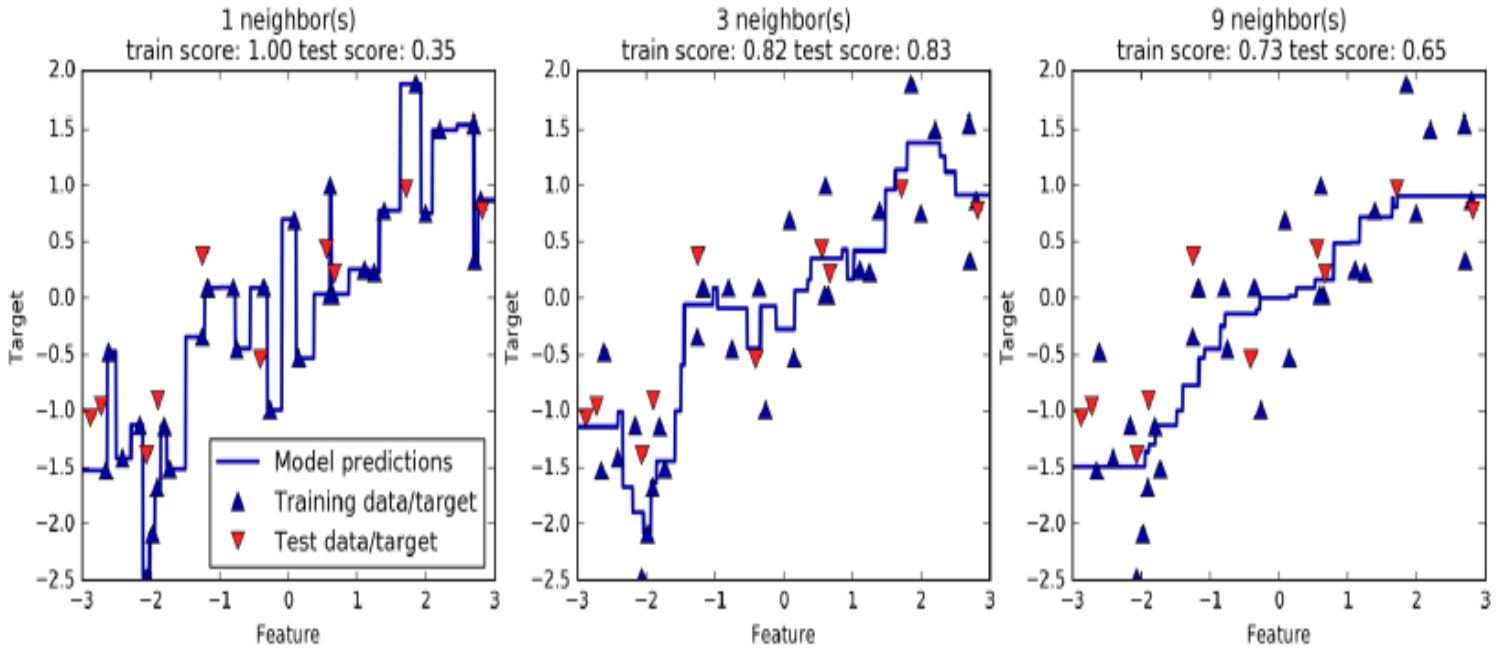


Figure 2-10. Comparing predictions made by nearest neighbors regression for different values of `n_neighbors`

Computation

```
from sklearn.neighbors import KNeighborsRegressor
X, y = mglearn.datasets.make_wave(n_samples=40)
# split the wave dataset into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=0)
# instantiate the model and set the number of neighbors to consider to 3
reg = KNeighborsRegressor(n_neighbors=3)
# fit the model using the training data and training targets
reg.fit(X_train, y_train)
print("Test set predictions:\n", reg.predict(X_test))
```

```
Test set predictions:
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]
```

```
print("Test set R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

```
Output:
Test set R^2: 0.83
```

Score of regression

Return the coefficient of determination of the prediction. The coefficient of determination
is defined as the residual sum R^2 given by

$$R^2 = 1 - \frac{\Sigma(y_{true} - y_{pred})^2}{\Sigma(y_{true} - y_{true.mean})^2}$$

- The best possible score is 1.0
- The score can be negative (because the model can be arbitrarily worse goes against the trend of data.).
- A (constant) model that always predicts the mean value of y, disregarding the input features, would get a score of 0.0.

In class work

1. Using the boat purchase data set, compute Euclidean distance between first two rows ignoring the non-numeric data.

Table 2-1. Example data about customers

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

In class work

1. Using the boat purchase data set, compute Euclidean distance between first two rows ignoring the non-numeric data.

Answer:

$$\sqrt{(66-52)^2+(1-2)^2+(2-3)^2}$$

$$= \sqrt{14 * 14 + 1 + 1}$$

$$= \sqrt{196 + 1 + 1 =}$$

$$= \sqrt{198}$$

$$= 14.07$$

END SESSION

Table of Contents

Model Evaluation

To evaluate our supervised models, so far we have split our dataset into a training set and a test set using the train_test_split function, built a model on the training set by calling the fit method, and evaluated it on the test set using the score method, which for classification computes the fraction of correctly classified samples and in case of regression computes R^2

```
X_train, X_test, y_train,
y_test=train_test_split(X,y,random_state=0)
logreg = LogisticRegression().fit(X_train, y_train)
print("Test set score: {:.2f}".format(logreg.score(X_test,
y_test)))
```

```
Test set score: 0.88
```

What if we changed the training set and test set within same data? Would we get a different model and different score?

Cross Validation

- *Cross-validation* is a statistical method of evaluating generalization performance
- In cross validation, the ***data*** is split repeatedly and multiple models are trained.
- Most commonly used version of cross-validation is *k-fold cross-validation*, where *k* is a user-specified number, usually 5 or 10

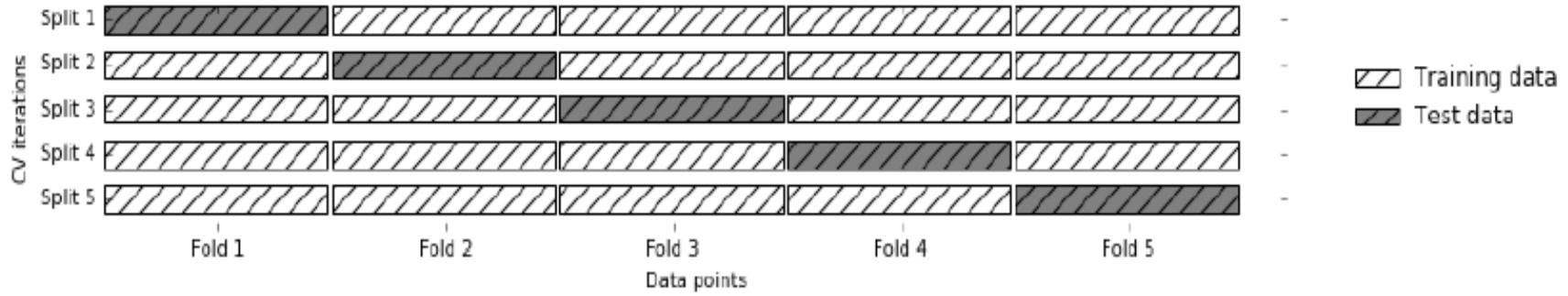


Figure 5-1. Data splitting in five-fold cross-validation

Computation Using cross_val_score

```
sklearn.model_selection.cross_val_score(estimator, X, y=None,
*, groups=None, scoring=None, cv=None, n_jobs=None, verbose=0,
fit_params=None, pre_dispatch='2*n_jobs', error_score=nan)
```

estimator = The object used to fit the data
X,y = Data and target
cv = Number of folds (None means 5!)
Rest we will ignore for now.

Computation using cross_val_score

```
logreg = LogisticRegression();
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)// default is used
print("Cross-validation scores: {}".format(scores))
// data is split into training and test and training is split into 5 folds


Cross-validation scores: [1.  0.967 0.933 0.9 1. ]
```

- Note that scores are the 5 scores for the 5-fold cross validation.
- `cross_val_score` can work on any model and you need to specify the model as a parameter!
- The function `cross_val_score` does the split of the data so we do not need to call
- `test_train_split`. It does it `cv` times and runs the `fit` method.
- The entire `data X,y` is provided to `cross_val_score`

- QUESTION: Why use `cross_val_score` instead of `fit and score`?
- ANSWER: To Justify the use of the specific estimator (model) stating that it performs no matter how you split the data.

Computation using cross_validate

Similar to `cross_val_score` but also returns training score if needed

```
res = cross_validate(logreg, iris.data, iris.target, cv=5,
return_train_score=True)
display(res)
```

```
{'fit_time':  array([0.002, 0.002, 0.002, 0.001, 0.002]),
'score_time': array([0.,    0. ,   0.001, 0.001, 0.001]),
'test_score': array([1. ,   0.967, 0.933, 0.9 ,  1.   ]),
'train_score':array([0.95 , 0.967, 0.967, 0.975, 0.958])}
```

QUESTION: Why use `cross_validate`?

ANSWER: To check if the model is overfitting/under-fitting the data, in addition to justifying the estimator.

Computation using cross_validate

```
sklearn.model_selection.cross_validate(estimator, X, y=None, *,
groups=None, scoring=None, cv=None, n_jobs=None, verbose=0,
fit_params=None, pre_dispatch='2*n_jobs',
return_train_score=False,
return_estimator=False, error_score=nan)
```

Splitting Using kFold

- `KFold` cross-validator is basically a “splitter” which can be used as the value of `cv` instead of `cv=5`, in previous methods `cross_validate` and `cross_val_score`.
- We can provide `train/test` indices to split data into training and test set.
- Split dataset into k consecutive folds (without shuffling by default).
- Each fold is then used once as a validation while the `k -1` remaining folds form the training set.
- You must still `fit` the data to the model

Splitting with kFold and Shuffle

```
logreg = LogisticRegression();
kfold = KFold(n_splits=3,shuffle=True, random_state=0)
scores=cross_val_score(logreg, iris.data, iris.target, cv=kfold)
print("Cross-validation scores:\n{}".format(scores)
```

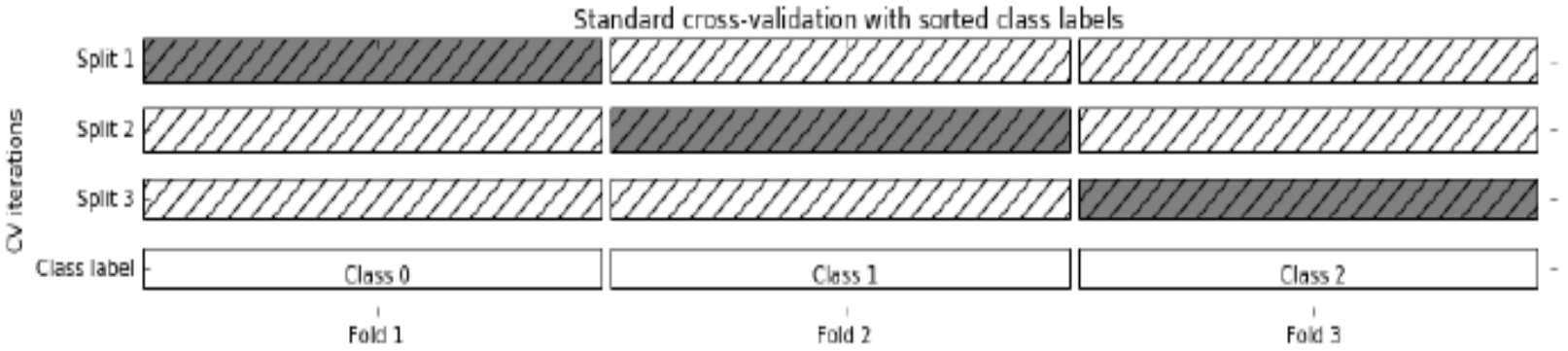
`shuffle`: Whether to shuffle the data before splitting into batches.
Note that the samples within each split will not be shuffled.

`random_state`: Is affected when shuffle is True.

```
Cross-validation scores:
[0.9 0.96 0.96]
```

Standard cross validation

If shuffle is on with KFold then data is shuffled before splitting into classes



Splitting Using kFold

```
class sklearn.model_selection.KFold(n_splits=5, *,
shuffle=False, random_state=None)
```

- `n_splits:` number of splits of training data
- `shuffle:` Whether to shuffle the data before splitting into batches. Note that the ***samples within each split will not be shuffled***.
- `random_state:` if shuffle = True, it provides a seed to the randomness

Splitting Using KFold

```
logreg = LogisticRegression();
kfold = KFold(n_splits=3)
scores = cross_val_score(logreg, iris.data, iris.target, cv
= kfold)
print("Cross-validation scores:\n{}".format(scores))
```

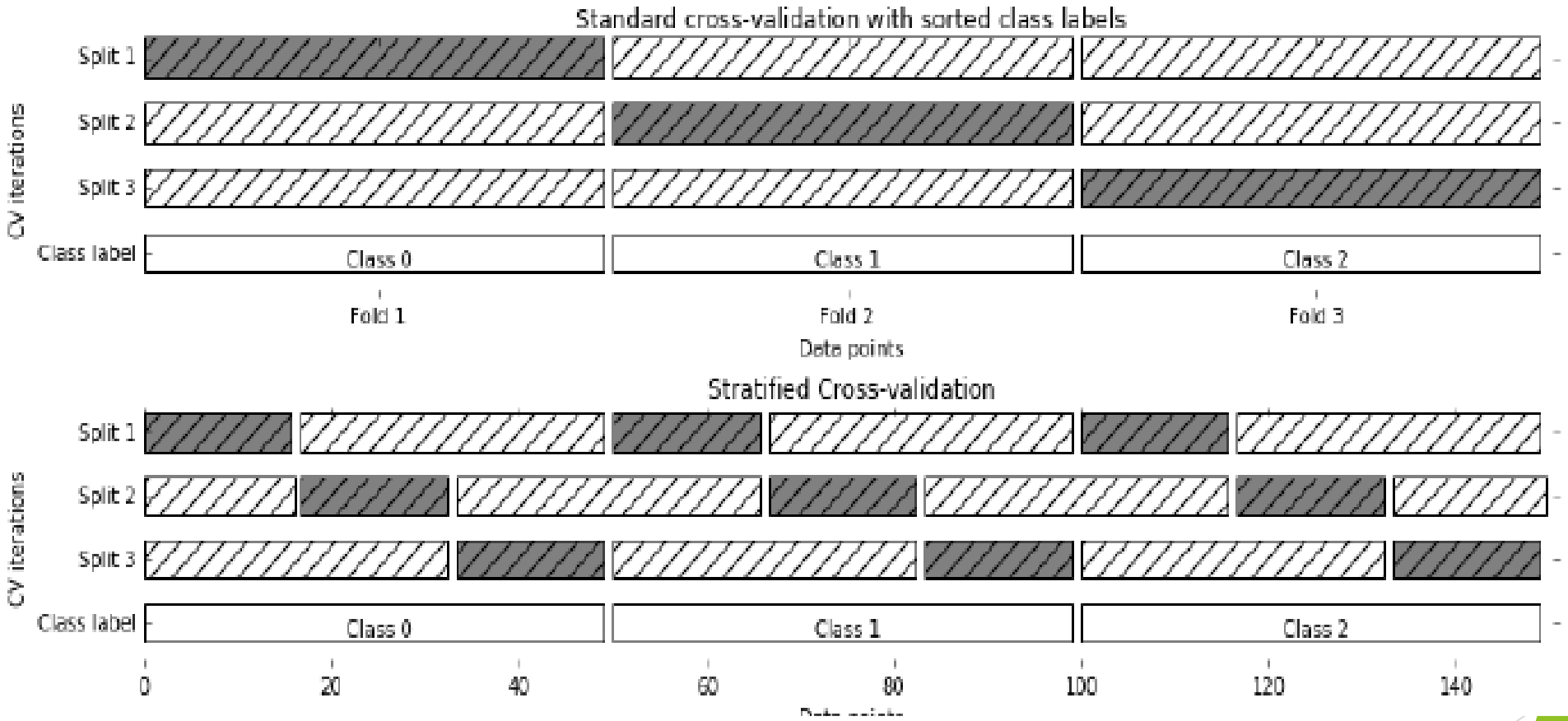
```
Cross-validation scores:
[0. 0. 0.]
```

Why they are all 0s?

Stratified KFold Split

- Both ***cross_validate*** and ***cross_val_score***, do NOT shuffle the data at all.
- If the data is organized by target values, the results would be wrong.
- Data has to be shuffled using KFold and setting shuffle=True
- Stratified cross validation avoids that.

Stratified Kfold



Splitting using Stratified kFold

```
class sklearn.model_selection.StratifiedKFold(n_splits=5,
*,shuffle=False, random_state=None)
```

`shuffle`: Whether to shuffle each class's samples before splitting into batches. Note that the samples within each split will not be shuffled.
`random_state`: Is affected when shuffle is true.

```
logreg = LogisticRegression();
skfold = StratifiedKFold(n_splits=3)
print("Cross-validation scores:\n{}".format(
cross_val_score(logreg, iris.data, iris.target, cv=skfold)))
```

```
Cross-validation scores:
[0.9  0.96  0.96]
```

Splitting using Leave One Out

- For each split, you pick a single data point to be the test set.
- This can be very time consuming, particularly for large datasets, but sometimes provides better estimates on small datasets:

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut();
logreg = LogisticRegression();
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo);
print("Number of cv iterations: ", len(scores));
print("Mean accuracy: {:.2f}".format(scores.mean()));
```

```
Number of cv iterations: 150
Mean accuracy: 0.95
```

Splitting Using Shuffle-Split

- Leave one out is too expensive.
- We use shuffle-split to leave “some” out
- In `shuffle-split`, in each split, we use `train_size` number of data points for the training set and `test_size` number of data points for the test set.
- This splitting is repeated `n_splits` times.

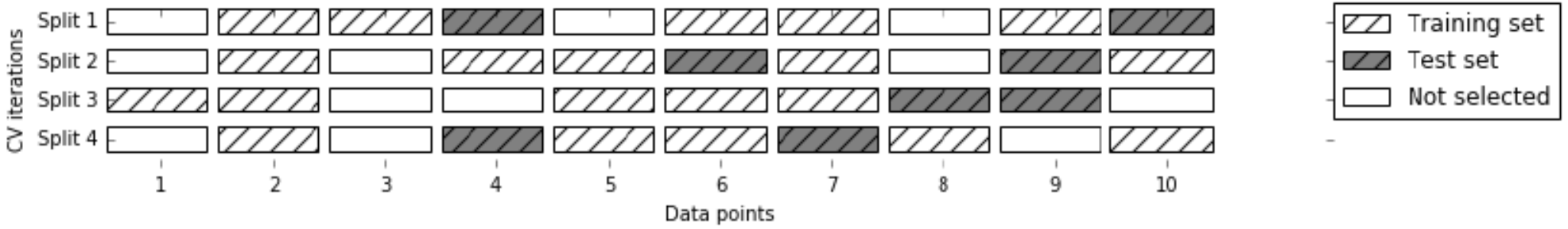


Figure 5-3. ShuffleSplit with 10 points, train_size=5, test_size=2, and n_splits=4

Computation for ShuffleSplit

```
from sklearn.model_selection import ShuffleSplit
logreg = LogisticRegression();
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

```
Cross-validation scores:
[0.973 0.92 0.96 0.96 0.893 0.947 0.88 0.893 0.947 0.947]
```

Grid Search

- Grid Search is used to tune the parameters of the model. For k-NN for instance, k is a parameter.
- It is important to understand what the parameters mean before trying to adjust them
- The range of possible parameter values should come from understanding the model parameters.
- Parameters of Support Vector Machine (SVM) not covered yet. There are two of them C and Gamma.
- We would like to know the performance of SVM for each of these parameter combination.

	C = 0.001	C = 0.01	...	C = 10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	...	SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	...	SVC(C=10, gamma=0.01)
...	...	...	...	...
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	...	SVC(C=10, gamma=100)

Computation- Naïve Cross Validation with loops

```
best_score = 0;
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
      for C in [0.001, 0.01, 0.1, 1, 10, 100]:
            svm = SVC(gamma=gamma, C=C)
            svm.fit(X_train, y_train)
            score = svm.score(X_valid, y_valid)
            if score > best_score:
                  best_score = score;
                  best_parameters = {'C': C, 'gamma': gamma}
svm = SVC(**best_parameters)
svm.fit(X_train, y_train)
test_score = svm.score(X_test, y_test)
print("Best parameters: ", best_parameters)
print("Test set score with best parameters: {:.2f}".format(test_score))
```

```
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}meters))
```

Grid Search with Cross Validation

- Note that the grid search of parameters was for a specific train/test split. This accuracy is not reliable for a general model.
- For each parameter combination we get only one score in the code above.
- We also need to cross validate across various train/test splits for each each parameter combination.

Computation – Naïve Grid Search with CV

```
best_score = 0;
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
      for C in [0.001, 0.01, 0.1, 1, 10, 100]:
      svm = SVC(gamma=gamma, C=C)
      scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
      score = np.mean(scores)
      if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters)
```

Naïve Grid Search with CV

```
mglearn.plots.plot_cross_val_selection()
```

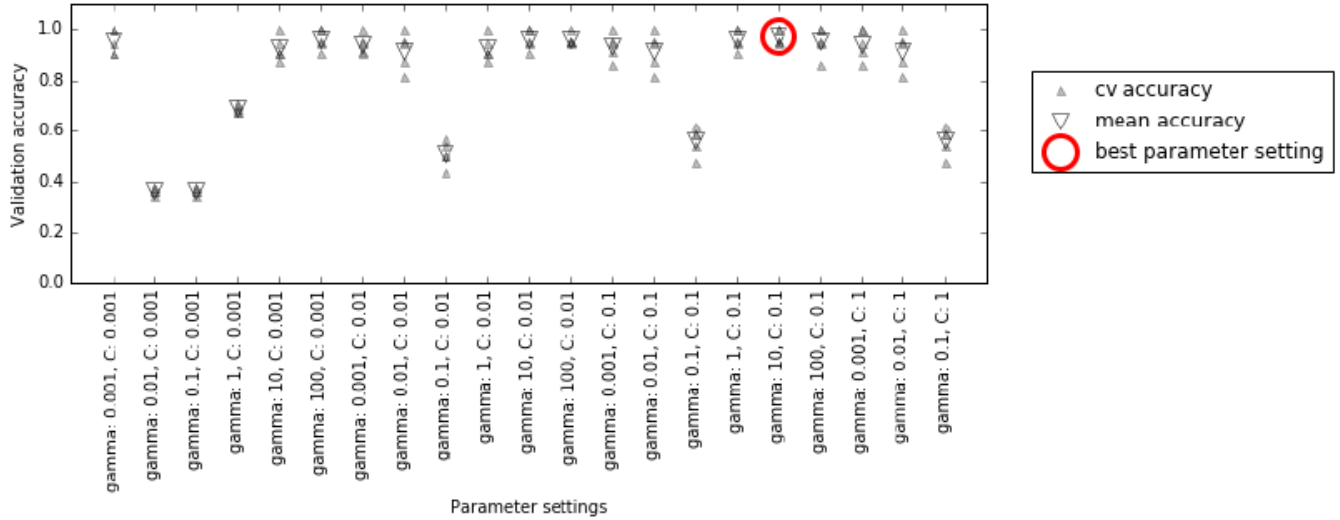


Figure 5-6. Results of grid search with cross-validation

GridSearchCV: Avoiding the Loop

```
from sklearn.model_selection import GridSearchCV

from sklearn.svm import SVC

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10,
100]}

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.fit(X_train, y_train) // fit will use cross validation

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

Test set score: 0.97
```

- GridSearchCV does NOT fit the data. It only sets grid search and cross validation parameters.
- We still need to call fit to fit the data to the model.
- Important note here is that we chose not to use test data when creating the model. Score is based on test data.

Datasets

Some popular datasets for Machine Learning include

1. UCI
http://archive.ics.uci.edu/ml

The University of California (Irvine) is a well-known location for classification and regression data sets.

2. Kaggle
Kaggle.com Competition Data Sets

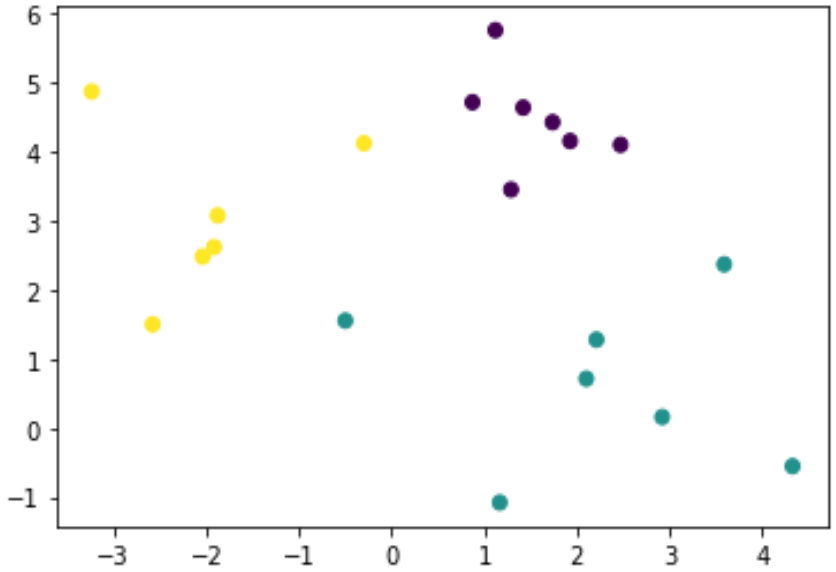
Data sets from a variety of competitions. Also a good source for class project ideas.

3. KDNUGGETS
http://www.kdnuggets.com/datasets

The Association For Computing Machinery (ACM) has a special interest group on Knowledge Discovery in Data (KDD).
The KDD group organizes annual machine learning competitions

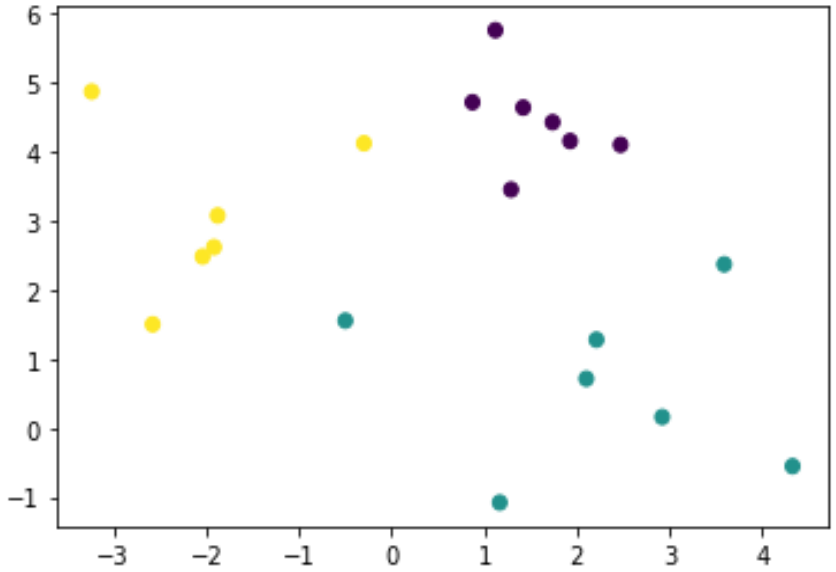
In class Work

1. Given the following data set and test set compute the accuracy (score) of 1-NN and 3-NN using the prediction for the test points as A(0,4.2) (yellow being correct answer),B (-0.5, 1.5) (green being correct answer), and C(1.75, 4.2) (blue being the correct answer)



In class work

1. A (0,4.2) – 1-NN Yellow 3-NN Blue Correct Answer: Yellow
2. B(-0.5, 1.5) – 1-NN green, 3-NN Yellow, Correct Answer: Green
3. C(1.75, 4.2) 1-NN Blue, 3-NN Blue, Correct Answer: Blue



1. Accuracy : 1-NN 100%
2. Accuracy:3-NN : 1 out of 3 correct: 33.33%

In class work

2. Split the following data into training and test sets using the following

a) cross_val_score using 3 fold
b) Kfold with Shuffle = True, 3-fold
c) StratifiedKFold with Shuffle = False, and cv= 3-fold
d) Leave one out
e) ShuffleSplit with train_size = 4, test_size = 3, n_splits = 2

In class work

Table 2-1. Example data about customers

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

In class work

2. There are 12 rows (samples) in the data.

a) cross_val_score using 3 fold

Fold 1: Training: Rows 1-8 and Test: Rows 9-12

Fold 2: Training: Rows 1-4, 9-12 and Test: Rows 5-8

Fold 3: Training: Rows 5-12 and Test:Rows 1-4

b) with Shuffle = True, 3-fold

Shuffle the rows and then pick

Fold 1: Training: Rows 1-8 and Test: Rows 9-12

Fold 2: Training: Rows 1-4, 9-12 and Test: Rows 5-8

Fold 3: Training: Rows 5-12 and Test:Rows 1-4

c) StratifiedKFold with Shuffle = False, and cv= 3-fold (30% of 4 numbers 1.3 but we pick 1 for testing)

Fold 1: Training: Rows 2,3,4,6,7,8,10,11,12 and Test: Rows 1,5,9

Fold 2: Training: Rows 1,3,4,5,,8,9,11,12 and Test: Rows 2,6,10

Fold 3: Training:Rows 1,2,3,5,6,8,9,10,12 and Test: 3,7,11

a) Leave one out

Fold 1: Training Rows 2-12 Test Row 1, Fold 2: Training Rows 1,3-12 Test Row 2, Fold 3: Training Rows 1,2,4-12 Test Row 3, ….

Fold 12: Training Rows 1-11, Test Row 12

a) ShuffleSplit with train_size = 4, test_size = 3, n_splits = 2

Training: Rows 1-4, Test Rows 5,7,10 Left Out Rows 6,8,9, 11,12

Training Rows 4,8,9,10 Test Rows 3,11,12 Left Out Rows : 1,2, 5, 6,7