Week 10: Classification 2

Dr Giuseppe Brandi

Northeastern University London

Outline

- K-Nearest Neighbors (KNN) algorithm
- Classification reports
- Model selection & hyper-parameter tuning
- Feature selection & engineering

What is K-Nearest Neighbors (KNN)?

- K-Nearest Neighbors (KNN) is a simple, intuitive, and non-parametric machine learning algorithm.
- It can be used for both **classification** and **regression** tasks.
- KNN classifies or predicts based on the **"closeness"** or **similarity** of data points, measured by a distance metric (e.g., Euclidean distance).

How KNN Works (Step-by-Step)

1. Choose the number of neighbors k.
2. Calculate the distance between the target point and all points in the training set.
3. Select the k closest points (neighbors).
4. For classification: Assign the class most common among neighbors.
   For regression: Take the average value of the neighbors.

Choosing the Value of k

- k is the number of neighbors to consider.
- **Low k values**: Makes the model more sensitive to noise, can lead to **overfitting**.
- **High k values**: Provides smoother predictions, but may lead to **underfitting**.
- **Finding Optimal k**: Commonly use cross-validation to determine the best k for the data.

Distance Metrics in KNN

- Different metrics can be used to calculate the "distance" or "similarity" between points.
- Common distance metrics include:
  - **Euclidean Distance**: For continuous data, calculates straight-line distance.
  - **Manhattan Distance**: For categorical or grid-like data, calculates the sum of absolute differences.
  - **Cosine Similarity**: For text and high-dimensional data, measures angle between vectors.

Advantages and Disadvantages of KNN

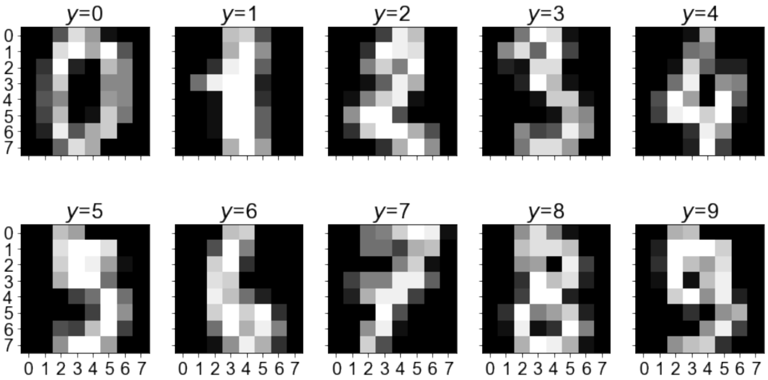
- Advantages
  - **Simple to understand** and easy to implement.
  - **No training phase**: All computation happens at prediction time.
  - **Flexible with distance metrics**, making it adaptable to various data types.
- Disadvantages
  - **Computationally expensive** for large datasets, as it requires calculating the distance to all points.
  - **Sensitive to irrelevant or noisy features** and scales of the data.
  - **Choice of k and distance metric** can significantly impact performance.

Applications of KNN

- **Recommendation Systems**: Finding similar items or users.
- **Medical Diagnosis**: Predicting disease based on patient symptoms and historical data.
- **Image and Text Classification**: Classifying objects in images or documents based on similarity.
- **Anomaly Detection**: Identifying outliers in data, such as fraud detection.

Example dataset

$1,797 \; 8 \times 8$ images of digits (0-9):



The goal is to generalise, not overfit

Tip

The **goal** of machine learning is **good generalisation**; that is, the ability to predict new data.

Splitting the data set in Python

Use 75% of data for *training* and 25% for *testing*; and **shuffle**!

Splitting a dataset

```
from sklearn.model_selection
      import train_test_split as split

train.X, test.X, train.y, test.y =
                         split(images,
                               labels,
                               test_size=0.25,
                               random_state=123456789)
```

Training a classifier

As an example, let's use a *k*-**nearest neighbours** classifier:

Training a *k*-nearest neighbours classifier

```python
from sklearn.neighbors import KNeighborsClassifier

# Create the classifier
classifier = KNeighborsClassifier()

# Fit the training data
classifier.fit(X=train.X, y=train.y);
```

Test *vs* training accuracy

Computing test accuracy

```python
accuracy = classifier.score(test.X, test.y)
accuracy = accuracy * 100.0
print('Test accuracy is {:5.2f}%'.format(accuracy))
```

```
Test accuracy is 97.33%
```

This is to be **contrasted** with a training accuracy of 99.33%.

Confusion matrix

A confusion matrix C is such that $C(i,j)$ equals the number of observations known to have label i and assigned label j.

- Correct predictions are on the **diagonal**
- Non-zeros not on the diagonal are *incorrect* predictions
- Each **row** represents a class (in our case, 0-9)

Confusion matrix in Python

Calculating a confusion matrix

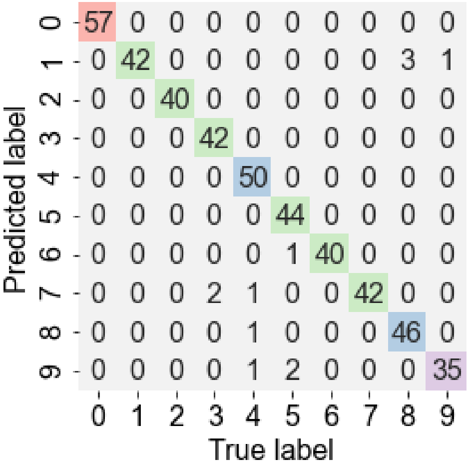
```python
from sklearn.metrics import confusion_matrix

# test.y are the known labels for each test image.
# But what does our model predict?
predictions = classifier.predict(test.X)

C = confusion_matrix(test.y, predictions)
```

Let's plot the confusion matrix.

Visualising a confusion matrix



Classification report

Generating a classification report

```python
from sklearn.metrics import classification_report
report = classification_report(test.y, predictions)
print(report)
```

```
    precision    recall  f1-score    support

0        1.00      1.00      1.00         57
1        0.91      1.00      0.95         42
2        1.00      1.00      1.00         40
3        1.00      0.95      0.98         44
4        1.00      0.94      0.97         53
...
```

Understanding a classification report

- **Precision** measures the fraction of correct predictions, incl. errors, for a label
- **Recall** measures a classifier's ability to predict correctly all instances of a label
- For example, for digit "1"
  - Precision is 91% – not all predictions were correct
  - Recall is 100% – all "1"s were classified correctly
  - So, our classifier mistaken other digits for a "1"

Model selection problem

In practical machine learning applications, we need to set a number of *hyper-parameters*

Besides tuning a particular model, we also want to consider a range of different types of model to find the best one for a particular application

Model selection problem

The typical approach is to split our data into three sets:

- Training set
- Validation set
- Test set

But what if the supply of data is limited? One solution is to use **cross-validation**

Cross validation

The main idea is to use **all data** for training and testing by splitting the data set into **K equally-sized folds** (i.e., data partitions)

Suppose that $K = 4$, resulting in 4 folds, S_1, S_2, S_3 and S_4

Then, for we perform $K = 4$ experiments:

- Train the model with S_2, S_3, S_4 and test with S_1
- Train the model with S_1, S_3, S_4 and test with S_2
- Train the model with S_1, S_2, S_4 and test with S_3
- Train the model with S_1, S_2, S_3 and test with S_4

and average the classification scores from each run.

Cross validation in Python

Creating a K-fold

```python
from sklearn.model_selection import KFold,
                                    cross_val_score

folds = KFold(n_splits=10, shuffle=True)
classifier = KNeighborsClassifier()
scores = cross_val_score(estimator=classifier,
                         X=images, y=labels,
                         cv=folds)
print(f'{len(scores)} scores: {scores.mean():.2%}
        +- {scores.std():.2%}')
```

Model comparison

It is difficult to know in advance which machine learning models will perform best for a given dataset.

This is particularly true when using a library like `sklearn`, where implementation details are hidden from us.

Let's compare the *k*-nearest neighbour classifier with another method:

- `DecisionTreeClassifier()`

Model comparison

Comparing three classifiers

```
estimators = {
    'K-nearest neighbours': KNeighborsClassifier(),
    'Decision Tree': DecisionTreeClassifier()
}

folds = KFold(n_splits=10,
              random_state=123456789,
              shuffle=True)
```

Comparing three classifiers

```python
for name, classifier in estimators.items():
    # Cross-validate classifier and report result
    scores = cross_val_score(estimator=classifier,
                             X=images,
                             y=labels,
                             cv=folds)
    print(f'{name:20s}: {scores.mean():.2%}
                        +- {scores.std():.2%}')
```

```
K-nearest neighbours: 98.72% +- 0.79%
Decision Tree       : 86.09% +- 2.37%
```

Hyper-parameter tuning

Each model has a number of (hyper-)parameters that can be tuned for a particular application and data set.

Examples include:

- Leaf size & num. neighbours for `KNeighborsClassifier()`
- Max tree depth for `DecisionTreeClassifier()`

Grid search

It is difficult to find the values of the important parameters of a model that provide the best generalisation performance.

Grid search is a method to try all possible combinations of the parameters of interest.

- Use the `sklearn.model_selection.GridSearchCV` method
- Uses K-fold cross-validation behind the scenes for finding the best combination

Grid search for k-nearest neighbours

```python
from sklearn.model_selection import GridSearchCV
# Parameters to search, and possible values
grid = {
    'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'leaf_size':   [10, 20, 30, 40, 50]
}
g = GridSearchCV(classifier, grid, cv=10)
g.fit(X=train.X, y=train.y)
g.best_params_
```

```
{'leaf_size': 10, 'n_neighbors': 5}
```

Feature selection

We can use a **univariate feature selection** method:

- Checks for a *statistically significant relationship* between each feature and target
- Each feature is individually considered
- Features that are related with the highest confidence are selected

Feature selection in Python

Let's try to select the 42 most important features (out of 64) in our `digits` dataset

Select 42 out of 64 pixels per image

```python
# Define a two-part selection method
extractor = make_pipeline(
                VarianceThreshold(threshold=0),
                SelectKBest(k=42))

extractor.fit(train.X, train.y)
extracted = extractor.transform(train.X)
```

Classifier re-trained with `extracted` data has same accuracy!

Feature selection – outlook

Besides univariate feature selection, we can use:

- Model-based feature selection
  - Train another model, this time for feature selection
- Iterative feature selection
  - Try all combinations

Feature engineering

Consider the following data set. Feature *Group* cannot be used as an input to a machine learning algorithm directly.

Name	*Group*	*Grade*
Alex	A	70.1
Jane	A	95.0
John	B	65.8
Lucy	C	100.0

But we can engineer it.

Feature engineering

Creating a one-hot encoding of a column

```python
from sklearn.preprocessing import OneHotEncoder

# Select column 'group'
groups = df.group.values.reshape(-1, 1)
# One-hot encoding
encoder = OneHotEncoder(sparse=False)
g = encoder.fit_transform(groups)
```

```
1 0 0   # was A
1 0 0   # was A
0 1 0   # was B
0 0 1   # was C
```

Feature engineering

Similar to one-hot encodings, we can **discretise continuous variables** by splitting them into a fixed number of bins

Creating a one-hot encoding of a column

```
from sklearn.preprocessing import KBinsDiscretizer

binning = KBinsDiscretizer(n_bins=5,
                           strategy='uniform',
                           encode='onehot-dense')
```

```
1 0 0 0 0  # was 70.1
...
```

Summary

- Classification reports – test accuracy & confusion matrix
- Model selection & hyper-parameter tuning
  - Cross-validation
- Feature selection & engineering