Why is it not a good idea to train a classification algorithm with all the data points available? What do we do instead?

What three methods did we cover in the previous chapter that are common to all supervised algorithms in scikit-learn? Briefly explain what each does.

**Supervised Learning: Classification and Regression**

**Classification lingo:**
- **Rows -->** samples = data points = instances = observations
- **Columns -->** features = attributes = dimensions = measurements
- **Last column -->** label = target

<table>
<thead>
<tr>
<th>Sepal LengthCm</th>
<th>Sepal WidthCm</th>
<th>Petal LengthCm</th>
<th>Petal WidthCm</th>
<th>Species</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.9</td>
<td>3</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>5</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>setosa</td>
</tr>
<tr>
<td>4.6</td>
<td>3.4</td>
<td>1.4</td>
<td>0.3</td>
<td>setosa</td>
</tr>
<tr>
<td>5</td>
<td>3.4</td>
<td>1.5</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.4</td>
<td>2.9</td>
<td>1.4</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.9</td>
<td>3.1</td>
<td>1.5</td>
<td>0.1</td>
<td>setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.7</td>
<td>1.5</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.8</td>
<td>3.4</td>
<td>1.6</td>
<td>0.2</td>
<td>setosa</td>
</tr>
<tr>
<td>4.8</td>
<td>3</td>
<td>1.4</td>
<td>0.1</td>
<td>setosa</td>
</tr>
<tr>
<td>4.3</td>
<td>3</td>
<td>1.1</td>
<td>0.1</td>
<td>setosa</td>
</tr>
</tbody>
</table>

Classification --> binary vs. multiclass

Regression = Classification with continuous labels/targets, e.g. temperature, price, etc.
Generalization, Overfitting, Underfitting

When a new data point comes in, there are two possibilities:

1. It is the same as one in the training set (unlikely in large-scale problems), or
2. It is a new one, that the algorithm has not examined before --> We say that the algorithm generalizes.

What conditions ensure accurate generalization?

1. The algorithm (a.k.a. model) must be complex enough to capture (fit) all the relevant characteristics of the data.
2. The algorithm/model must not be overly complex.
3. If not, we say that the algorithm underfits the data
4. Examples: Train only on Setosa and Virginica, so the algorithm never “sees” a Versicolor during training, so it will classify incorrectly all instances of Versicolor in the testing set.
5. Use only one feature (say, Sepal_length) and ignore the rest.
6. If it is too complex, we say that the algorithm overfits the data; it models noise, rather than real properties of the data

7. Examples:\footnote{1}{https://en.wikipedia.org/wiki/Overfitting}
\footnote{2}{https://docs.aws.amazon.com/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html}
Model Complexity in Relation to the Size of the Dataset

Ideal case: Collect more data, which exhibits more characteristics, so the model can get more complex without overfitting.

When “more data” goes wrong:

- the “new” data is duplication of the old
- the “new” data is too closely related to old data

Rule of thumb: “Getting more data is better than improving the learning algorithm.”
The 4 Datasets Used in This Chapter

**forge** (synthetic, 26 data points with labels for classification)

```python
X, y = mlearn.datasets.make_forge()
print X.shape
print y.shape
mlearn.discrete_scatter(X[:,0], X[:,1], y)
```

**wave** (synthetic, *mlearn* can generate the requested nr. of data points, labels continuous, for regression)

```python
X, y = mlearn.datasets.make_wave(n_samples=100)
print X.shape
print y.shape
mlearn.discrete_scatter(X, y)
plt.plot(X, y)
```

**cancer** (real-world, has labels for classification)

```python
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print cancer.keys()
print cancer.data.shape
```


► Print the first 5 data points, and the first 5 labels (targets). Does 0 mean good or bad?
**housing** (real-world, labels continuous, regression)

```python
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.keys())
print(boston.data.shape)
```

Note that `target_names` is missing.

Feature engineering --> Creating “new” features, based on the old ones

Feature interactions --> features are not independent. One way to take it into account is by multiplying all pairs of features.

- An extended `boston` dataset is available in `mlearn`.

```python
import mlearn
X, y = mlearn.datasets.load_extended_boston()
print(X.shape)
```

☐ Explain 104.

► To do for next time: Read pp.37-39 **k-Nearest-Neighbors**, enter and run all commands shown. (It is a rehash of the `iris` KNN classification from Ch.1.)
fit(), predict(), and score() are common to all supervised algorithms in scikit-learn.

Print the first 5 data points, and the first 5 labels (targets). Does 0 mean good or bad?

```python
import pandas as pd

cancer = pd.read_csv('cancer.data')
cancer.head()

cancer['target_names'] = ['malignant', 'benign']
cancer.head()
```

| 1.795000e+00 | 1.850000e+01 | 1.220000e+02 | 1.010000e+03 |
| 1.810000e+00 | 2.770000e+01 | 5.200000e+01 | 1.470000e+02 |
| 2.410000e+00 | 7.870000e+00 | 1.095000e+00 | 9.453000e+00 |
| 1.520000e+00 | 1.530000e+00 | 6.700000e+00 | 4.064000e+00 |
| 2.180000e+00 | 1.567000e+00 | 5.805000e+00 | 6.135000e+00 |
| 2.530000e+00 | 1.713000e+00 | 1.640000e+00 | 2.915000e+00 |
| 1.620000e+00 | 6.650000e+00 | 7.121000e+00 | 2.654000e+00 |
| 4.681000e+00 | 1.120000e+00 | 1.777000e+00 | 1.326000e+00 |
| 2.975000e+00 | 2.854000e+00 | 6.600000e+00 | 7.817000e+00 |
| 1.420000e+00 | 5.657000e+00 | 5.425000e+00 | 7.130000e+00 |
| 1.300000e+00 | 7.460000e+00 | 5.245000e+00 | 1.220000e+00 |
| 1.600000e+00 | 1.340000e+00 | 1.000000e+00 | 5.120000e+00 |
| 1.240000e+00 | 1.940000e+00 | 1.530000e+00 | 1.950000e+00 |
| 1.330000e+00 | 1.690000e+00 | 2.410000e+00 | 1.600000e+00 |
| 2.750000e+00 | 1.090000e+00 | 1.009000e+00 | 1.279000e+00 |
| 1.900000e+00 | 1.599000e+00 | 1.970000e+00 | 1.279000e+00 |
| 2.800000e+00 | 5.900000e+00 | 7.850000e+00 | 7.000000e+00 |
| 4.300000e+00 | 9.400000e+00 | 6.130000e+00 | 4.600000e+00 |
| 3.030000e+00 | 2.050000e+00 | 2.250000e+00 | 4.570000e+00 |
| 2.570000e+00 | 2.550000e+00 | 1.520000e+00 | 1.700000e+00 |
| 1.640000e+00 | 4.120000e+00 | 6.560000e+00 | 2.430000e+00 |
| 1.610000e+00 | 8.750000e+00 | 1.320000e+00 | 2.770000e+00 |
| 1.140000e+00 | 2.830000e+00 | 7.850000e+00 | 7.000000e+00 |
| 1.420000e+00 | 2.830000e+00 | 2.410000e+00 | 1.052000e+00 |
| 2.570000e+00 | 1.940000e+00 | 1.520000e+00 | 1.700000e+00 |
| 3.440000e+00 | 2.720000e+00 | 9.110000e+00 | 7.450000e+00 |
| 5.610000e+00 | 2.167000e+00 | 5.930000e+00 | 9.200000e+00 |
| 1.290000e+00 | 2.050000e+00 | 9.610000e+00 | 5.470000e+00 |
| 2.090000e+00 | 8.630000e+00 | 6.000000e+00 | 2.575000e+00 |
| 6.030000e+00 | 1.730000e+00 | 1.320000e+00 | 2.770000e+00 |
| 1.080000e+00 | 1.950000e+00 | 1.080000e+00 | 1.279000e+00 |
| 1.000000e+00 | 4.850000e+00 | 5.250000e+00 | 7.000000e+00 |
| 1.430000e+00 | 4.440000e+00 | 1.190000e+00 | 2.410000e+00 |
| 5.610000e+00 | 1.850000e+00 | 1.750000e+00 | 5.150000e+00 |
| 2.560000e+00 | 1.670000e+00 | 1.520000e+00 | 5.750000e+00 |
| 1.370000e+00 | 2.050000e+00 | 4.800000e+00 | 1.625000e+00 |
| 1.540000e+00 | 7.470000e+00 | 2.050000e+00 | 1.625000e+00 |
Lab 2 problems:

0] Display the versions of all the Python modules we have used so far.

1] Display a table with the numbers N from 1 to 20 as the first column, their factorials as the second column, and their tangent (trig. function, found in the math module – make sure you convert from angles to radians first!) in the third column. Use the format method, not %.
   - There should be three column headings: \( N \), \( N! \), and \( \tan(N) \), all centered
   - The headings should be separated from the data with dash symbols –
   - The columns should be separated from each other with the pipe symbol |
   - Where dashes meet pipes, use the plus symbol +

Pandas
2] Create a Pandas Series \( ds1 \) containing all the even integers between 1 and 20. Print the Series.
   Now create a second Series \( ds2 \) containing all the odd integers between 1 and 20. Print it.
   Find the average, minimum and maximum numbers in each Series.

3] Add, subtract, multiply, divide, and exponentiate all elements of \( ds1 \) by the corresponding elements of \( ds2 \). (Hint: Don’t write loops, use vectorized operations!)

4] We know that a DataFrame is a dictionary of columns. Use this to create a DataFrame \( df \) having the Series \( ds1 \) \( ds2 \) above as columns. Name the columns \( s1 \) and \( s2 \). Print \( df \). What do you notice about its index?

5] Change the first index of \( ds1 \) from the default value of 0 to 42. (Hint: Convert the index in to a list, modify the list, then assign the new list back to the index.)

6] Repeat #4 above with the new Series \( ds1 \). What do you notice?
   A: 

7] Train the KNN classifier with 7 neighbors, on 90% of the iris data. Use 0 for the random seed, so we can compare results.
   Classify only the first 5 data points in the test set, print their predicted labels and their actual labels.

Email the lab report containing screenshots of the code and output to the instructor: agapie@tarleton.edu.
Please write 4086 Lab 2 in the subject line.
K-Nearest-Neighbors

Modify the code to display the decision boundaries for four numbers of neighbors: 2, 5, 13 and 26. Include titles for the plots, as shown in the text code (p.40).

--Does the plot for n=2 make sense? (Hint: overfitting!)
--Does the plot for n=26 make sense? (Hint: underfitting!)
Breaking ties: the parameter **weights** (Not in text)

**weights**: str or callable, optional (default = 'uniform')

weight function used in prediction. Possible values:
- 'uniform': uniform weights. All points in each neighborhood are weighted equally.
- 'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable]: a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

```python
from matplotlib import pyplot as plt
fig, axes = plt.subplots(1, 2, figsize=(13, 6))
clf = KNeighborsClassifier(n_neighbors=26).fit(X, y)
mlearn.plots.plot_2d_separator(clf, X, fill=True,
                           eps=0.5, ax=axes[0], alpha=0.4)
mlearn.discrete_scatter(X[:,0], X[:,1], y, ax=axes[0])

clf = KNeighborsClassifier(n_neighbors=26, weights='distance').fit(X, y)
mlearn.plots.plot_2d_separator(clf, X, fill=True,
                           eps=0.5, ax=axes[1], alpha=0.4)
mlearn.discrete_scatter(X[:,0], X[:,1], y, ax=axes[1])
```

Training accuracy vs. testing accuracy (generalization)

```python
import matplotlib.pyplot as plt
import numpy as np
import sklearn.datasets as load_breast_cancer
import sklearn.neighbors as KNeighborsClassifier
import sklearn.model_selection as train_test_split

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target,
    stratify=cancer.target, random_state = 66)
training_accuracy = []
testing_accuracy = []

for n in range(1, 11):
    clf = KNeighborsClassifier(n_neighbors=n).fit(X_train, y_train)
    training_accuracy.append(clf.score(X_train, y_train))
    testing_accuracy.append(clf.score(X_test, y_test))

plt.plot(range(1, 11), training_accuracy, label='training')
plt.plot(range(1, 11), testing_accuracy, label='testing')
plt.legend()
```

Note: **Stratification** means that, while selecting points randomly for the train and test sets, the proportion of the classes in those sets is kept the same as in the total dataset. (More details on p.256 ff.)

□ Which do you think takes more time, `clf.fit()` or `clf.predict()`?
The complexity of prediction in KNN (not in text)

Notation: \( N \) is the nr. of points \( p \) in the training set (rows), \( d \) is the nr. of attributes (columns), and \( k \) is the nr. of neighbors.

Consider the following algorithm for predicting a new data point \( p_{\text{new}} \) (pseudocode):

For each of \( N \) points \( p \):
   - Calculate distance from \( p_{\text{new}} \) to \( p \)
   - Store distance
   - Mark \( p \) as \textit{not\_selected}

For \( i \) in range(\( k \)):
   - Find the minimum distance to all points \( p \) which are \textit{not\_selected}
   - Store that point (its label)
   - Mark the point as \textit{selected}

\( \square \) What is the time complexity (Big Oh) of the algorithm above in terms of \( N \), \( d \), and \( k \)?

\( \square \) Advanced algorithm: What is the time complexity (Big Oh) of the algorithm above if, instead of the second for loop, we use \textit{quickselect} (O(N)) to find the \( k \)th smallest distance, and then simply retrieve all points whose distances are smaller than that?
KNN regression

When using \( k > 1 \) neighbors, the predicted value is the average of the kNN, e.g.

\[
\text{mlearn.plots.plot_knn_regression(n_neighbors=3)}
\]

- Modify the code on p.39 (forge - classification) into the code on p.44 (wave - regression)
X, y = mlearn.datasets.make_wave(n_samples=40)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state = 0
)
reg = KNeighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
print reg.predict(X_test)
print 'R^2 =', reg.score(X_test, y_test)

```
[-0.05396539  0.35686046  1.13671923 -1.89415682 -1.13881398 -1.63113382  
  0.35686046  0.91241374 -0.44680446 -1.13881398]
R^2 = 0.834417244625
```

### Training accuracy vs. testing accuracy (generalization)

Since \( X_{\text{train}} \) is only 1D in this example, we can visualize the prediction curve directly (code on p.45):

![Graphs showing training and testing accuracy for different numbers of neighbors in KNN classification and regression.](image)

### Conclusions for KNN classification and regression

- Simple, easy to understand, gives reasonable results w/o much tinkering --> often used as baseline, against which more complex algorithms are evaluated.

- Most important parameter: the nr. of neighbors. \( 2^{nd} \) most important parameter(s): the metric (parameters \texttt{metric} and \texttt{p} in the constructor), i.e. how the algorithm calculates distances --> outside the scope of our course and text. Use the default (Euclidean).

- KNN is a “lazy” classifier, so building the model is fast, little more than simply storing the data set in memory. But this means that all data needs to be in memory for prediction!

- The classification time for one point is \( O(N(d+k)) \), which can become large if \( N \) or \( d \) (or both!) are large.

- There is no known way for the KNN computations to be efficiently performed on sparse matrices (e.g. CSR), so the \( O(Nd) \) complexity becomes prohibitive for very large \( d \) (of which most are zeroes).
**KNN usually requires data to be normalized (brought to the same range of values, or the same average), otherwise the small features will have almost no say in the classification. This is especially true if Euclidian metric is used, as opposed to Manhattan.**

**Gives only one answer, rather than multiple answers with probabilities (see other classification algorithms). When two classifications are almost equal, the classifier may apply other algorithms, or even ask for human input!**

For more “theoretical” practice, see the link 30 Q&A on KNN on our webpage!

Solutions

- Modify the code to display the decision boundaries for four numbers of neighbors: 2, 5, 13 and 26. Include titles for the plots, as shown in the text code (p.40).

--- Does the plot for n=2 make sense? (Hint: overfitting!)
--- Does the plot for n=26 make sense? (Hint: underfitting!)

```python
from matplotlib import pyplot as plt
fig, axes = plt.subplots(1, 4, figsize=(15,5))
for n, ax in zip([2, 5, 13, 26], axes):
    clf = KNeighborsClassifier(n_neighbors=n).fit(X, y)
    mlearn.plot2d_separator(clf, X, fill=True,
                           eps=0.5, ax=ax, alpha=0.4)
    mlearn.discrete_scatter(X[:,0], X[:,1], y, ax=ax)
    ax.set_title('k neighbors'.format(n))
```

--- For n=2, the algorithm is still overfitting, less than for n=1, but more than for n=3.
--- There are 26 data points total, 13 in each class; the algorithm broke the tie in favor of the circles.

□ Which do you think takes more time, clf.fit() or clf.predict()?

A: KNN is a “lazy classifier”, i.e. it doesn’t actually compute any model parameters. clf.fit() simply stores X_train and y_train in memory. The complexity is $O(Nd)$, where $N$ is the nr. of points in the training set, and $d$ the number of features. The real work takes place during prediction, as explained in the notes.

□ What is the time complexity (Big Oh) of the algorithm above in terms of N, d, and k?

A: $O(Nd + Nk) = O(N(d+k))$.

□ Advanced algorithm: What is the time complexity (Big Oh) of the algorithm above if, instead of the second for loop, we use quickselect ($O(N)$) to find the kth smallest distance, and then simply retrieve all points whose distances are smaller than that?

A: $O(Nd)$. 