8. Machine Learning: Part 2

Application of the week: the internet of things (IoT)

DataKirk

Overview of today's session

1. Summary of last week
2. The digits dataset
3. A new ML algorithm
4. K-fold cross validation

Last week: supervised learning

- Supervised learning is an important area in machine learning where one tries to make predictions given a **labelled** dataset.
- A labelled dataset means we have access to both the input variables (or *features*), and the true label or outcome. The task is then to make future predictions given only new input features.
- In supervised learning tasks, your data will always look something like this:

Feature 1	**Feature 2**	**Feature 3**	**Label**
feature 11	feature 21	feature 31	label 1
feature 12	feature 22	feature 32	label 2
feature 13	feature 23	feature 33	label 3
...	...	...	...

Last week: classification and regression

- Supervised learning breaks down into two sub-areas: **classification** and **regression**. The aim is often quite similar, but the tasks are subtly different.

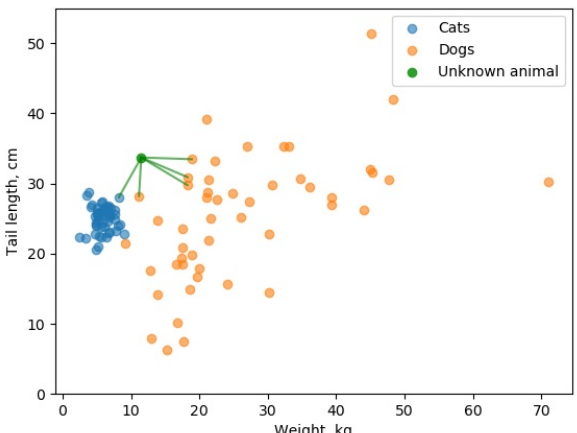
- In regression, one tries to predict some *number* given past data. For example, you could try to predict the life expectancy of a city given historical data about air pollution, access to health services, median wage, obesity rates etc.

- In classification, one tries true predict some *class* given past data. For example, you could try to predict whether a customer buys a product online given their age, their gender, what types of product they have bought in the past etc. In this case there are two distinct classes: buys product or does not buy product. However, there could be more than two classes.

Last week: the k-nearest neighbours algorithm

- K-nearest neighbours is a machine learning algorithm for classification.
- The idea is that predictions are made on new data points by analysing the "distance" between any new point, and all the other labelled points in data space.
- We then look at the k (say, 5) closest data points. Our prediction for the class of the new data point is the class that the majority of the neighbours has.

Last week: the k-nearest neighbours algorithm



Last week: train test split

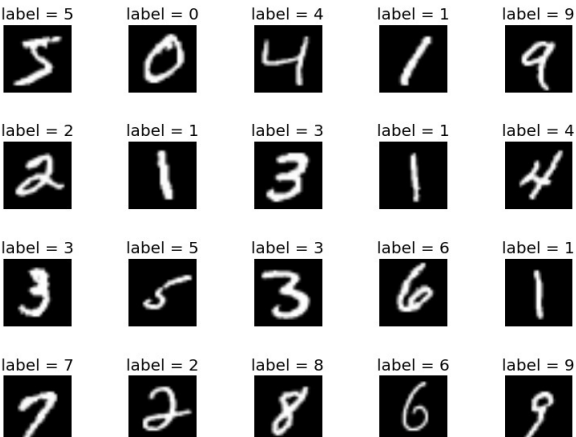
- In order to analyse how well our algorithm is performing, it's always a good idea to split data into a training set and a testing set.

- This means we randomly select say 60-80% of the rows to be training data, and the rest to be testing data.

- Using only the training data, we then try to make "predictions" for the true class label of the test set. Since we know what the true labels are for the test set, we can then analyse how well the algorithm has performed.

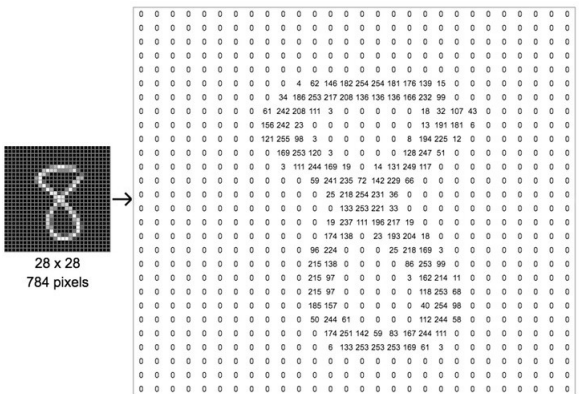
- If we don't do this, and instead analyse performance on our training set, we can be lulled into a false sense of security. It may seem like our algorithm is performing really well, but the truth is, when it is tried out on new data, it performs much worse than expected. This is called *overfitting* and is a very common problem in machine learning.

New topic: The digits dataset



The digits dataset

- The digits dataset, also known as MNIST, is a classic dataset in machine learning. The task is to predict a number, from 0 to 9, given a 28x28 grid of pixels.



The digits dataset

- The data has been produced by digitising thousands of hand-written digits.
- Each pixel gives an "intensity", which is a number ranging from 0 to 256. This grid can be flattened to give a data table that looks something like this:

Pixel 1	Pixel 2	Pixel 3	...	Pixel 784	Label
0	15	125	...	3	4
11	0	34	...	35	6
65	93	4	...	150	1
...	...	...	...	...	...

- This is where the idea of "artificial intelligence" begins to become more apparent. Recognising handwritten digits is an eerily human ability!

The digits dataset

- Today we will be using a slightly simplified version of the full MNIST dataset, where the digits are on an 8x8 grid, rather than 28x28. This makes things a little easier for the algorithm, but is essentially the exact same task. This digits dataset can be loaded in directly using scikit-learn.

```
from sklearn.datasets import load_digits
data = load_digits(as_frame=True).frame
data
```

pixel_0_0	pixel_0_1	pixel_0_2	...	target
0	0	5	...	0
0	0	0	...	1
0	0	0	...	2

The digits dataset

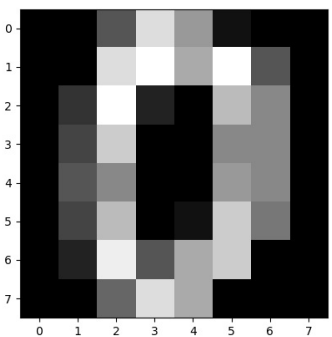
```
# use .loc[] to get the first row
# then use .values to get this row as a pure array
# then take [:-1], meaning everything up to the last value
# becuase the last value is the label
row = data.loc[0].values[:-1]
# then reshape into a grid
grid = row.reshape(8, 8)
print(grid)
```

```
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

The digits dataset

- Once we have reshaped this row into a grid, we can use matplotlib's `imshow()` function to view the image.

```
plt.figure()
plt.imshow(grid, cmap='gray')
```



The digits dataset

- Let's turn this into a function to plot any row.

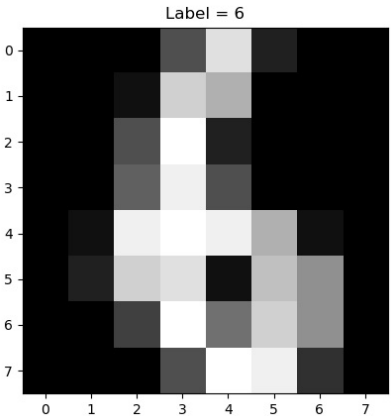
```
def show_digit(row_number):

    row = data.loc[row_number].values[:-1]
    label = data.loc[row_number].values[-1]
    grid = row.reshape(8, 8)

    plt.figure()
    plt.imshow(grid, cmap='gray')
    plt.title('Label = ' + str(int(label)))
```

The digits dataset

```
show_digit(34)
```



New topic: A new ML algorithm

- Now we will consider using a new ML algorithm called **support vector machines**.

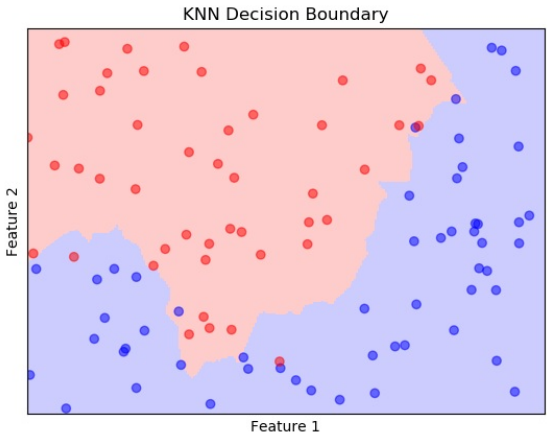
- The mathematical details of how this algorithm works are not important for the purpose of this session, however, here is a high-level overview of what is going on.

- Classification can be thought of as drawing a *decision boundary* in data space. This means we want to divide up all of the input space into regions where we say:

> "If a new data point is in a certain region, I'm going to guess it should be class 1. If it's outside of that region, I'll guess it's class 2. "

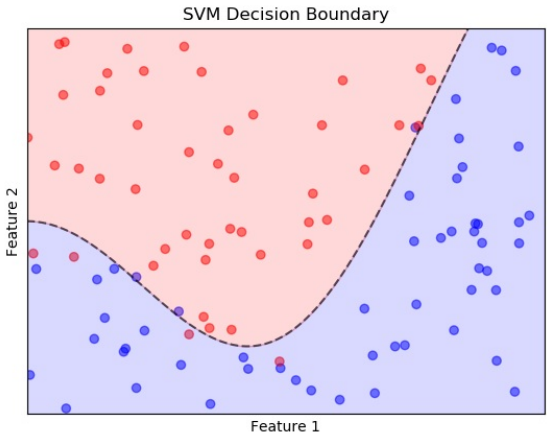
Decision boundaries

- K-nearest neighbours has an implied decision boundary.



Decision boundaries

- A support vector machine tries to draw a smooth decision boundary.



Support vector machines

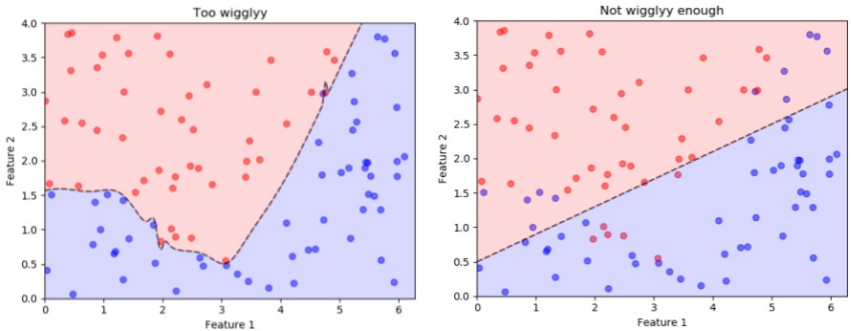
- The only hyperparameter we need to worry about for support vector machines is the "wigglyness" of this line.

- A super-wiggly line might fit the data very well, curving around all the points perfectly, so that it fits the observed data with 100% accuracy. However, this is unlikely to generalise well to new data. This would be considered overfitting.

- On the other hand, a very smooth or straight line may not fit the data well enough. This would be considered underfitting.

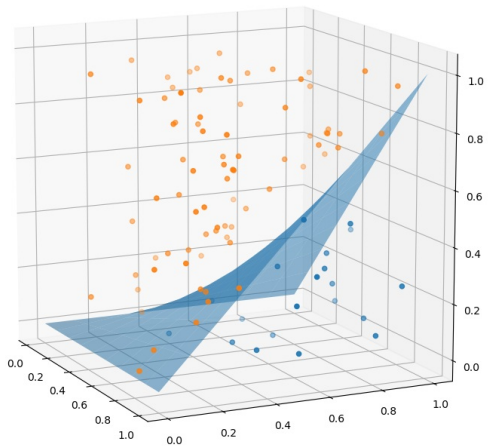
- The best balance will be somewhere in between.

Support vector machines



Support vector machines

- So far we've only looked at two dimensions, however, as with K-nearest neighbours, this idea can be extended to higher dimensions. In this case, we're not finding a *line* that separates different regions, but a *hyperplane*.



Support vector machines

- Back to the digits dataset. Since each image is a set of 64 pixels, each one can be thought of as a "point" in 64 dimensional space. Our algorithm is going to find the best set of dividing lines in this 64-dimensional space to split it up into 8 different regions - one for each digit.

- Although this sounds like a lot, the code for implementing a support vector machine is very similar to that for K-nearest neighbours from last time.

Support vector machines

```
# import the svm algorithm from scikit-learn
from sklearn.svm import SVC

# import train test split
from sklearn.model_selection import train_test_split

# perform train test split
train_data, test_data = train_test_split(data, test_size=0.2,
random_state=37)

# create a new classifier. C=1 here sets the "wigglyness"
classifier = SVC(C=1)

# fit the classifier on the training data
classifier.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

# make a prediction on the test data
prediction = classifier.predict(test_data.iloc[:, :-1])
```

Support vector machines

- As before, let's test the model's accuracy

```
print(sum(prediction == test_data.iloc[:, -1]) / len(prediction))
```

```
0.9944444444444445
```

- Over 99% accuracy! Not bad.

Analysing the model

- It might be interesting to take a look at some of the examples where the algorithm is failing. How "intelligent" is it? Do we feel sorry for some of the mistakes?

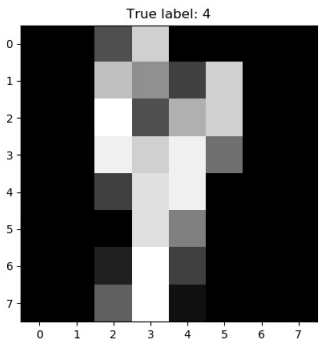
```
fails = test_data[prediction != test_data.iloc[:, -1]]
fails
```

	pixel_0_0	pixel_0_1	pixel_0_2	...	target
1628	0	0	5	...	4
5	0	0	12	...	5

- Looks like we failed to predict two numbers: 4 and 5. But what did they look like?

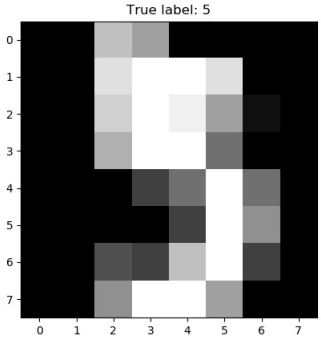
Analysing the model

```
plt.figure()
plt.imshow(fails.iloc[0, :-1].values.reshape(8, 8), cmap='gray')
plt.title('True label: 4')
```



Analysing the model

```
plt.figure()
plt.imshow(fails.iloc[1, :-1].values.reshape(8, 8), cmap='gray')
plt.title('True label: 5')
```



Analysing the model

- Both pretty wonky if you ask me! But what did we predict?

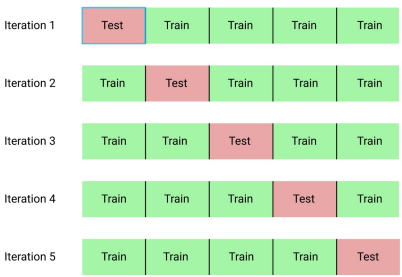
```
predicted = prediction[prediction != test_data.iloc[:, -1]]
print(predicted)
```

```
[9, 9]
```

- 9 on both counts - I think that's a reasonable guess to be honest!!

K-fold cross validation

- K-fold validation is a way of getting a better understanding of how accurate your model is.
- The idea is this: we will split the data into 5 different sections, with 20% of all the examples randomly put into each section.
- Then, we will assess the model accuracy five different times. Each time, we will train on 8-0% of the data, and test on 20% - using a different test 20% each time.



K-fold cross validation

- Scikit-learn has a nice function that allows us to do this relatively easily.

```
# import the KFold function
from sklearn.model_selection import KFold

# make a k-fold iterator with 5 splits, shuffling the data
kf = KFold(n_splits=5, shuffle=True)

# perform 5 loops, getting random rows
for train_index, test_index in kf.split(data):

    X_train = data.iloc[train_index, :-1]
    X_test = data.iloc[test_index, :-1]

    y_train = data.iloc[train_index, -1]
    y_test = data.iloc[test_index, -1]
```

K-fold cross validation

- On each iteration of the loop, let's fit a new classifier and print its accuracy.

```
for train_index, test_index in kf.split(data):

    X_train = data.iloc[train_index, :-1]
    X_test = data.iloc[test_index, :-1]

    y_train = data.iloc[train_index, -1]
    y_test = data.iloc[test_index, -1]

    classifier = SVC(C=1)
    classifier.fit(X_train, y_train)
    prediction = classifier.predict(X_test)

    print(sum(prediction == y_test) / len(prediction))
```

K-fold cross validation

- This prints something like this:

```
0.9916666666666667
0.9944444444444445
0.9805013927576601
0.9832869080779945
0.9860724233983287
```

- Looks like our initial estimate for the accuracy was a little high!
- A good way of reporting your accuracy is to average over the accuracy of your different cross validation slices.
- So here, our reported accuracy would be

```
sum([0.99166, 0.99444, 0.98050, 0.98328, 0.98607]) / 5
```

```
0.98719
```

Thanks!