Certificate 2 Session 1 — Script

[slide 1]
Last session we tried for the first time to incorporate statistical, *probabilistic* decision making into our program: the program code remained deterministic, but the output relied heavily on an external set of data. The logic of the program stated what to do with the data (what statistics to collect, how to interpret it), but the actual choice of a move in the games of Battleships depended on what is in the data. Still the way we processed data was very simplistic.
In this session we are going to have an overview of a powerful technique of learning from data that gives phenomenal performance results in a vast range of areas — *machine learning*. Machine learning is one of the main components of the modern days' AI, although using the two terms interchangeably would be inaccurate. Still, recent advancements in AI are mostly based on machine learning.

[slide 2]
So what is machine learning? In order to answer this question we first need to formally define what learning is.

[slide 3]
A classical definition of machine learning given by Tom Mitchell in late 90s (go check his book, it is a classic one!) goes as follows: suppose you have some task *T*, for instance translating a sentence from one language to another, or predicting the stock price, or making the weather forecast, or even something very simple as picking up an object with a robotic arm. You can estimate how well the task was carried out by some *performance measure M*, e.g., was the translation correct or not, how accurate was the predicted price, was the robotic arm able to pick up an object or even hold it. You can repeat the task over and over again — this is called *experience*. Then learning means improving your performance at task *T* with experience, as measured by *M.* And while you can apply this definition to any kind of learning, including your own, its formal nature makes it perfect for defining machine-based, artificial, automated learning — ML.

[Please come up with your own examples of learning scenarios.]

The key ingredient of ML that makes it so useful in a multitude of areas is that one does not have to *directly program* a system to carry out a task; instead one needs to program on a meta-level so to say, i.e., to program *how to learn* to carry out a task. This means that instead of implementing the logic "if X do Y; if Z do A, B, C", the machine learning engineer implements some principle of how to *generalise, abstract,* learn from experience (or data). This may sound like a much more challenging task, and sometimes it is indeed, in which case you do not need ML and you stick to your usual way of programming (e.g., why learn how to search through a small number of neighbouring cells if you can do it declaratively). But surprisingly in many areas the declarative enumeration of if-this-then-that cases is prohibitively complex, and this is when ML comes in handy. For example, there are so many factors why a stock price may change, or there are so many pixels in a picture where one tries to automatically identify objects that it is easier, faster, cheaper and most importantly of higher accuracy to apply ML.

One last key feature of ML is that it is learning *from* something. In the definition above we call it experience, and in formalised, machine-readable way it will be simply *data*. Whenever you want apply ML to a problem, first you need to think about what data you are going to do the training (learning) on, and whether the data is of good quality. Having data is a necessary, although not sufficient condition for effectively applying ML models.

[slide 4]
Now let's look at an example of a problem where ML could help. Suppose you are a movie recommendation website, and you would like to train a model that given a user's profile predicts which movies she might like. First, we need the data, so we decide to explicitly ask the user which movies she likes — and very importantly which movies she does not like, so that we can differentiate. We ask about one movie per question, and we receive a you-no answer from the user. With every such question-answer pair (which becomes our data) we know a bit more about the user, so we can train the model a bit more efficiently — and with time the model starts suggesting relevant movies with fewer mistakes.

[slide 5]
What we just outlined is called *supervised learning*, i.e., learning that is guided by the correct answers — labels — to the data. To date, this is the most common type of ML, and the one that historically appeared first. So let's go through the standard *pipeline* of supervised ML along with some real-world examples.
First we need to collect data. Every data element is called a *data instance*. The instances depend on the task: if the task is to build a spam detecting software, the instances would be emails; if the task is to guess the user's age from her profile picture, the instances are people's photos; if the task is for the bank to be able to predict whether a client is solvent or not, a data instance would be the client's profile; if the task is to find tweets about similar topics, the instances would be tweets.
We cannot use the instances as is, in their raw representation, so the next step is to convert them to a machine-readable format. The most common way to represent data instances is via *feature vectors*. A feature vector is a vector (think of a Python one-dimensional list), where every element corresponds to some property of the instance: words used in a tweet or an email, client's age, picture size etc. The properties are encoded through some numeric values. in case of age of size the encoding is straightforward; in case of textual features it is not, but there are ways how to represent text as feature vector, and we will look into them later. It is important that the feature vectors of data instances in the same dataset are of the same length, and the features are encoded in the same order across the dataset.
Now comes the part that makes supervised learning supervised. For every data instance we provide the correct *label* — the correct answer to how it should be treated by the ML algorithm. For spam detection, every email should be labeled "spam" or "not spam"; for age prediction every picture is labeled with the user's age; every user profile in the bank system should be labeled "solvent" or "not solvent"; every tweet should be labeled with its topic (or topics) etc. The labeles can be assigned manually or generated automatically (i.e., age information can be taken from a person's profile).
At this point the dataset is prepared. We feed the dataset to a learning algorithm of choice, and it *trains a model* that seeks to correctly assign a label given an instance, whether seen or unseen. It is very important that the model performs well on unseen data, i.e., on data that was not used for its training. If it does, then the model *generalises* well.

[slide 6]
Let us schematically represent the pipeline of supervised learning. We collect data instances, convert them into symbolic feature vectors of the same length, assign labels to each data instance. We then feed them into an algorithm which generates a trained model. Once trained, the model can be reused, so that when a new data instance arrives, it can be labeled (hopefully correctly).

[slide 7]
The process depicted horizontally is called *training*,…

[slide 8]
…and the process depicted vertically is called *testing*. To be mode precise, if the unseen data instance still has a label that we can verify and compare with the label the mode outputs, it is called testing. Testing is a crucial process in ML as it gives us an estimate of how well the model is trained and how well it generalises. Through the analysis of mistakes the model does at training phase it is sometimes possible to understand what should be improved, e.g., the model needs more data to learn from, or some important features are missing, or the dataset had noisy data

that are hard to learn from etc. It is very important that training and testing are done on different subsets of the data.

[slide 9]
Last bit of theory so that you can navigate through basic ML examples. So far we have looked at supervised learning, in which case the correct labels are presented at the training and testing phases. We know which emails are spam and which come from your landlord. However, in real case scenarios labeled data is not always available. Manual labelling is usually slow and costly, and cannot provide the amount of data instances needed for decent training (we are talking hundreds of thousands if the task is complex). Hence, people came up with other approaches.
In *unsupervised learning*, we do not know the correct labels. Suppose you found an archive of unclassified articles, you do not know what is in the archive and you cannot manually read every article and label it with its topic. In this case you can group, or *cluster*, similar articles together based on which words appear in them, in hope that articles from the same cluster would be on the same, or similar topic (and articles from different clusters would be on different topics).
The marriage of the two approaches is called *semi-supervised learning.* It is a very common type of learning, since in many real cases you have a certain relatively small amount of labeled data that you managed to acquire, and a much bigger set o unlabeled data. E.g., you managed to read some articles from the archive and determine their topics. You then bootstrap the learning process with the labeled data and carry on with the rest.

There are algorithms for each type of learning, and each type of learning comes with a price: labelling requires resources, but provides better guidance to the algorithm, while clustering is easier to do, but there is no information how many clusters there should be and what they should contain.

[slide 10]
Learning techniques can also be differentiated by the type of labels. As we mentioned, when there are no known data labels, the learning process is called clustering. If the learning is supervised, and the labels are categories, it is called *classification*. If, on the other hand, the labels are numeric values, like age, size, score etc., it is called *regression*. Think about two similar examples: a bank wants to predict your credit rating to issue you a loan. If it predicts a rating *category* (excellent, good, average, poor), this is a classification task. If it predicts a numeric *score* as rating, this is a regression task.

[slide 11]
Now let's play with machine learning! Today we are going to look at classification, the most traditional ML task.

[slide 12]
We have the following task: we have information about 250 different companies: what they do, what is their market, how well they grow, their credit history etc. I have taken this dataset from a research paper, and have put it on github: https://github.com/smartypanty/AIgaming-course/tree/master/C2S1
Based on the data about each company, we would like to predict whether it will go bankrupt or not. For those 250 companies we already know their business destiny, hence we can train a classification model!

[slide 13]
Let's look at the features in the dataset. For every instance there are 6 non-label features: industrial risk, management risk, financial flexibility, credibility, competitiveness and operating risk.
[You can talk through the features in a bit more detail.]
As you can see, these features correspond to complex parameters, but in order for them to be machine-readable, every feature can be encoded as a value of either *1* (positive performance), *0* (average performance) or *-1* (negative performance). Plus every data instance is labeled as bankrupt or not.

[slide 14]
Let's look into the data. All the complex intuition about the companies' performance is now modelled and captured in a concise vector representation. Every line corresponds to one

company, every position in the vector corresponds to a specific feature. you can directly read through the data, i.e., the company in line *4* has high values for all its parameters, and unsurprisingly it did not go bankrupt. On the other hand, the company in line *8* seemed to have an average to low performance and eventually went bankrupt. C'est la vie!

[slide 15]
Today we are not going to play games on the platform, but will train our first classification model. You can work in any editor of your choice — or in terminal. For today's session we are going to need *3* libraries. For the first steps of data preprocessing we will need a classical Python library *numpy* that focuses on handling arrays and matrices — a useful thing since we use feature vectors. We are also going to use the *re* library for handling regular expressions and reading the dataset from a text file.
We open the dataset file and read it into a variable `input`:

```
file = open('bankruptcy_dataset.txt', 'r'),
input = file.read()
```

We then convert symbolic feature values into numeric one, creating a one-to-one correspondence between letters and integers.
You then split the input into lines, and you can check that you have indeed *250* instances in the dataset: `print(len(instance_strings))`.
In the dataset file, the instances are ordered: first non-bankruptcy instances, then bankruptcy instances. We shuffle the instances using the `shuffle()` method from the *numpy* library.
Finally, we convert instance strings to lists using a simple regular expression (you can think of other ways of parsing the file). The resulting `instances` is a list of lists, each list being one data instance.

[slide 16]
No we need to bring our dataset to the format required by the learning algorithm we are going to use. The `dataset` should be a dictionary with *3* key-value pairs: '`data`' corresponds to a list of instances, '`targets`' corresponds to a list of labels, and '`target_names`' corresponds to the text labels 'bankrupt' and 'not-bankrupt':

```
dataset = {
    u'data': data,
    u'targets':targets,
    u'target_names':target_names
}
```

In order to create the lists, we go through `instances`, *slice* every instance `i` into first 6 elements (features) and the label and append the resulting values to the corresponding lists:

```
for i in instances:
    instance_data = i[:6]
    data.append(instance_data)
    target = i[6]
    targets.append(target)
```

As for the target_names list, we simply create a list of length *2*: `target_names = ['bankrupt', 'not-bankrupt']`

[slide 17]
We create two separate lists for ourselves: the list of data that is usually denoted by a vector **X**, and a list of labels (targets) that is usually denoted by a vector **Y**:

```
dataset_X = dataset.get('data')
dataset_Y = dataset.get('targets')
```

Now we split our dataset into training and testing parts. In practice, you would like to use as much training data as possible, especially in cases when data is scarce. Hence, we leave *90%* of our dataset for training and allocate *10%* for testing. It is of utmost importance that the distribution of classes ("bankrupt" and "not-bankrupt") is the same in train and test data (which is why we shuffled the data at the beginning). If you are going to do a more serious ML project, you should pay more attention to the data distribution; for now we simply shuffled the dataset, and we are going to slice it into *9/10* and *1/10*:

```
dataset_X_train = dataset_X[:225] # 90% for training
dataset_Y_train = dataset_Y[:225]
dataset_X_test  = dataset_X[225:] # 10% for testing
dataset_Y_test  = dataset_Y[225:]
```

Hoorah! We are ready to train our machine learning model! Our algorithm of choice is the famous k-nearest neighbour algorithm. It is a simple classification algorithm that takes a new data instance, depending on its features detects *k* training instances that are most similar to the given instance, looks at their classes and assigns the majority class to the new instance. You can read mode about kNN here: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

We are going to use the ML library *scikit-learn*. From the library we import the kNN algorithm:

```
from sklearn.neighbors import KNeighborsClassifier
```

We then instantiate a model of type kNN: `model = KNeighborsClassifier()`

And we train it (in their parlance, we make the model fit training instances `dataset_X_train` to their labels `dataset_Y_train`): `model.fit(dataset_X_train, dataset_Y_train)`
The model for predicting bankruptcy is trained! Now let's test it.

[slide 18]
In order to test the model on the test data, we call the function `predict()` of the model over the test data. Note that in the test phase, we (or rather the model) do not know the correct labels, so we run `predict()` only on the instances, not the targets:

```
predictions = model.predict(dataset_X_test).
```

One remaining thing in the pipeline is to evaluate the predictions. For that we can simply output `predictions` along with the true test instance labels `dataset_y_test`.

[slide 19]
Usually what happens is that there is a certain error rate, and in order to minimize it, an ML engineer goes back to the training phase to modify its components and parameters. After the modification the model is retrained and evaluated anew. This creates the train-test loop of developing an ML system. Try modifying the following parameters and see what changes in the behaviour of the model.