Certificate 2 Session 2 — Script

[slide 1]
Last session we built our first machine learning model. The bankruptcy dataset we were using was already given to us, and the features of the dataset were simple value encodings of *6* parameters: *1* means high value, *0* means average value, *-1* means low value.
Today we are going to work with *textual features*. How can we represent text in a machine-readable way, in the form of feature vectors? Which features can we use and how can we compute values for these features? In other words, we are going to enter the domain of Natural Language Processing, or NLP.
Secondly, we are going to study one of the most famous machine learning algorithms — logistic regression — and to run it on our textual data!

[slide 2]
And we will start with a concrete machine learning task called Language Identification: https://en.wikipedia.org/wiki/Language_identification
Suppose you have some text, a paragraph or just a phrase. The task is to name the language the text is written in. To make things a bit simpler, we are going to differentiate between three languages: Swedish, Norwegian and Dutch. All the three languages are from the same family (Norwegian and Swedish are North Germanic languages, and Dutch is West Germanic), so they are not too different from each other and not trivial to differentiate of you do not know them (and hopefully you don't =). We are going to process relatively short phrases, which makes the task a bit more challenging since the information about the instance is limited. Here you can see an example of the input data: three phrases meaning "happy birthday" in the three languages.

[slide 3]
We have collected our own dataset of phrases from Wikitravel phrasebooks. You can download the dataset from github: https://github.com/smartypanty/AIgaming-course/blob/master/C2S2/phrases.txt. In total there are *970* phrases: *333* for Swedish, *330* for Norwegian and *307* for Dutch. The file is structured as follows: there are two type of lines that follow one another. The lines commented out by the % symbol contain translations into English, and they are followed by phrase lines. The phrase lines are separated by the ||| symbol into the phrases themselves and the labels, i.e., the languages codes *SWE* (for Swedish), *NOR* (for Norwegian) and *DUT* (for Dutch).

[slide 4]
In order to represent our data instances formally, i.e., to extract features from the foreign phrases, we need to learn about *language modelling.* Language models are probability distributions of certain sequences of text in a language, or rather, over some *corpus* of texts (collection of texts, in simple words). Sequences could be characters, character sequences, words, phrases etc. You can read more about language modelling here: https://en.wikipedia.org/wiki/Language_model.
Using a language model, we can predict what is the probability of an item *i* to appear in text (think in Norwegian vs Swedish text), and also what is the probability of several items to appear in text in a certain order. This brings us to one of the central notions of NLP — *ngrams*. An ngram (sometimes spelled n-gram) is a continuous sequence of *n* items appearing in text. The two most commonly used types of ngrams are character ngrams and word ngrams. Think about a window of length *n* that you move along a piece of text, collecting the content of the window every time you move it one step further. For example, if you are given the sentence "how to make an AI gaming bot", and a window of size *3*, you can get the following *word trigrams*: 'how to make', 'to make an', 'make an AI', 'an AI gaming', 'AI gaming bot'. In the similar fashion you can also collect character trigrams from the same phrase: 'how', 'ow ', 'w t', ' to', 'to ', 'o m', ' ma', 'mak', 'ake', 'ke ', 'e a', ' an', 'an ', 'n A', ' AI', 'AI ', 'I g', ' ga', 'gam', 'ami', 'min', 'ing', 'ng '. A technical note: when harvesting character ngrams and creating a character-based model, one can either keep or discard the spaces. If the spaces are kept, the ngrams are able to capture patterns of word order which might be of great use.

[slide 5]
In practice, people usually do not go beyond ngrams of length *5*. Surprisingly, these simple units can capture linguistic patterns pretty efficiently and on several levels. Consider a Swedish phrase "Grattis på födelsedagen" which means happy birthday. First of all, character ngrams can capture a probability distribution of different letters in a language, which comes very much in handy in case a language with unique symbols, e.g., a letter *å*. Also if the task was to differentiate Chinese from Japanese from Korean, which use different sets of hieroglyphs, character unigrams would be enough (but you might not even need ML for this task). Secondly, ngrams can capture typical words and word parts, e.g., roots of verbs and nouns, prepositions, which also include suffixes, prefixes and endings of words that reflex the morphology of the language etc. Lastly, if we include spaces into the analysis, character ngrams can capture typical sequences of words. For instance, on a corpus of English texts an ngram "ed by" would have a high probability, and this is because in English a verb in passive voice (which usually ends with *-ed*) is often followed by a preposition *by* ("caused by", "divided by", "created" by etc.)

[slide 6]
Now we are all equipped to perform feature extraction! Let' start with extracting character ngrams. Our example string is "Kunt u mij dat tonen op de kaart" which means "Can you show me this on the map" in Dutch. We set the length of the ngrams in a separate variable *n*, which we set for *3* for now. Then by means of a *for*-loop you go through the sentence, from its first symbol until the symbol at position that is *n* steps away from the end (`for i in range(len(s) – n + 1)`), and builds an ngram of length 3 by means of slicing (`ngram = s[i:i+n]`, where `n` is *3*). Do not forget to save ngrams into the list of ngrams (`ngrams.append(ngram)`)!
You task is now to modify the code such that it generates all character ngrams *up to size n*, i.e., *1,2,…,n*. You can find the code in the *feature_extraction.py* file on github.

[slide 7]
What else can we do in language modelling? Ngrams are solid enough foundation for doing machine learning, but there is room for improvement:
(a) try running your classification model on a combined set of features, using both character and word ngrams;
(b) try playing with the size of the ngrams; does the performance of your model change when you go from *n=2* to *n=5*?
(c) do not forget to handle upper-case letters; do you want to differentiate "an" from "An", or do you want it to be the same ngram?
(d) you might not need extra spaces that accidentally appeared in the text; for that you might want to use the `strip()` function over strings in Python;
(e) as in the case of upper case, you need to decide what to do punctuation signs, whether to keep them (or some of them) or not; on the one hand, full stops are signs of the end of a sentence, which might be a useful feature for some languages; on the other hand, commas might be pretty useless and might only create unnecessary noise in the data.

[slide 8]
Now we are ready to prepare the dataset for language identification. It is important to remember that we not only train models on seen (train) data, but we also generate features only using the seen data. In other words, ngrams that appear in test but not in train data are excluded from the analysis. Let's go through the code.
1) We load the content of the phrases.txt file into the `file` variable and read it into `input`. As we did last session, we split the input file into lines (strings) and then shuffle them. Shuffling is extremely important, since in the original file the instances are ordered by language. Then we parse `instance_strings` line by line, filter out strings that are commented out (`if not s.startwith("%"):`), and split the non-comment strings into the actual phrases and language codes using the "|||" delimiter (`instance = re.split("\|\|\|", s)`). Phrases and language labels are saved into separate lists.
2) We define the ratio according to which we will split the data into train and test. Given the ratio and the dataset size, we identify the id of the instance at which we slice the dataset. DO not forget to round the number so that it is an integer: `train_size = round(len(data) *`

`train_test_ratio`). After we sliced the data into `train` and `test`, we go through the train phrases and generate ngrams (here we generate character trigrams). Note that we store the ngrams in a set since all features should be unique. Every unique ngram is one feature in out modelling.

[slide 9]
3) and 4) Now we generate feature vectors for all train and test instances. For that for every instance we go through the feature set (the ngrams) and check whether a given ngram appears in the instance. If it does, the corresponding slot in the feature vector is filled with *1*, otherwise *0*. This way we create *binary* feature vectors based on ngrams. In NLP there are numerous methods how to take ngrams into account: counting ngram occurrences, normalising these counts by instance length etc.

[slide 10]
As we discussed last session, there is a plethora of classification algorithms, even families of algorithms. Last time we were using k Nearest Neighbour, and today we are going to look closely at another classical supervised ML algorithm — *logistic regression*.

[slide 11]
Logistic regression is a standard, well-known classification algorithm. In its basic modification it is a binary classifier, i.e., it learns to differentiate between two classes. The important assumption that underlines the algorithm is that the data it classifies should be *linearly separable*, i.e., it should be possible to cut the feature space with a hypersurface (hyperplane, surface, plane, line) so that elements of the two classes were located on the opposite sides of the surface.
The goal of the Logistic Regression algorithm is to find such surface, such separator, which is formally called a *discriminant* (since it discriminate instances of the two classes). The shape of the discriminant is a polynomial of *n* dimensions, where *n* is the number of features in the dataset. In other words, among the set of infinitely many polynomial functions of a given shape we need to find the one that best separates the data, and since the shape of the function is fixed, we simply need to find the right coefficients of the polynomial.
Let us look at the example. We have a small dataset about student performance at the exam. Given the information about how many hours a student have studied before the exam, how many hours she slept and whether she passed the exam or not, we need to find such values for w_1, w_2 and w_3 that the polynomial w_1 * Studied + w_2 * Slept + w_3 produced scores that correspond to class values of the instances.

[slide 12]
The name of Logistic Regression is a bit misleading: it is called regression, yet it is a classification algorithm that handles categorical labels. The reason for that is that Logistic Regression stems from another algorithm Linear Regression, which is a true regression in a sense that it predicts numeric values. Linear Regression is also based on computing a multidimensional polynomial that models the data, but instead of separating instances of different classes, it *fits* the instances. While the behaviour and the purpose of the polynomials in the two algorithms are different, the way the coefficients of the polynomial are computed is similar, hence the name regression.

[slide 13]
Let's discuss the mathematical meaning of the output the discriminant produces. The predicted scores reflect how probable it is for a given instance to be of one class and not the other one. However, we cannot use the scores per se, since they can take any value between (-∞, +∞), while probabilities operate in the interval from *0* to *1*. Hence, we need to "squash" the original scores into [*0,1*]. The solution is the *sigmoid function* (https://en.wikipedia.org/wiki/Sigmoid_function) that has the form $f(x) = 1 / (1 + e^{-x})$. It is a bounded, monotonic real-valued function whose range is *(0,1)*. Now that the predicted scores are "rescaled" and fit between *0* and *1*, they can be interpreted as probabilities. In case of binary classification, a probability *P(i,a)* of an instance *i* being of class *a* equals to *1 - P(i,b)*, where P(i,b) is the probability of the same instance being of class *b*. Hence, using a hand-picked threshold, e.g., *0.5*, we can interpret the rescaled predicted score as either the label for class *a*, in case it is large enough, or of class *b*, if it is smaller than the threshold.

[slide 14]
So far so good. But what happens if we have more than two classes, just like in the language identification example? The solution is quite simple: instead of one discriminant, we need to build *k* of them, by the number of classes. Each discriminant will build a surface between the given class and the rest of the data. Each discriminant will produce a probability of an instance belonging to its respective class, and then the probabilities can be compared, and the class with the highest probability can be chosen.

[slide 15]
OK, now we know how to use the discriminant(s). But how to compute it? Here comes the learning part of machine learning.
Suppose you have *some* discriminant for a given problem. How well does it perform on the data? We can measure it in terms of errors, or *cost*. We define a *cost function* that takes as input both the true labels and the predictions for train instances, and returns a numeric value that reflects how many and how significant are the errors, or the discrepancies between the labels and the predictions. We will discuss the details of the cost function in a bit.
Given the cost function, we can randomly instantiate the weights, or the coefficients, of the discriminant, and iteratively update the weights so that the cost is minimised. The update can be done using the *gradient descent* optimisation method.

[slide 16]
If we plot the possible values of the discriminant's coefficients against the cost the discriminant with these values produces on train data, the function that we get is known to be continuous and convex. We can use this property as follows: at each point on the plot, i.e., at any given coefficient values, it is possible to move up and down the plot. The idea then is to move in the direction of smaller costs, i.e., to descend down the plot a bit. In order to find this direction, one needs to compute the *gradient* (multi-variate derivative) of the function at the given point, and take the step proportional to the negative of the gradient. We can iteratively compute gradients and taking steps until a minimum, the lowest point in some "valley" is reached. The gradient descent method guarantees to reach the local minimum on any function, and the global minimum in case the function is convex. When the minimum is reached, we know the coefficient values and how exactly the discriminant looks like.

[slide 17]
One last thing that remains to be discussed is the cost function. As we mentioned, the purpose of the cost function is to give us an idea about how well the current configuration of the discriminant performs, and to guide the gradient descent. The cost function that is used in Logistic Regression is called *cross entropy*, or *log loss* (http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#loss-cross-entropy).

[slide 18]
The cross entropy function compares two probability distributions over the same set of events, in our case the predictions and the true labels. The cross entropy of one instance measures how far the prediction diverges from the actual label and amounts to *-1* times the sum of two components: *y * log(p)* or predicting one class and *(1-y) * log(1-p)* for predicting the other class. This way cross-entropy penalises both types of errors (misclassifying *a* and *b*).
Look at the shape of the function on the graphic. This is the cross entropy (the loss) plotted against a predicted probability for an instance being of class *1* when the true label of the instance is indeed *1*. The higher the predicted probability of the instance being of that class, the smaller the loss, and visa versa. In order to calculate the cross entropy over the whole dataset, an average over all instances is taken.

[slide 19]
As you have seen, even one of the basic ML algorithms proves quite tedious to implement, let alone implement correctly and efficiently. Luckily, we have specialised libraries that contain implementations of all major classification, regression and clustering algorithms. Nevertheless, it is very important to know how each algorithm is working and which type of data it is best suited for.

We are going to use the `skicit-learn` implementation of Logistic Regression. The code for training and testing the model is similar to the one we used last session, when the algorithm of choice was k Nearest Neighbours. Do not forget to import the model: `from sklearn.linear_model import LogisticRegression`. Initialise and train the model on training data `dataset_X_train` and `dataset_Y_train`. Test it in `dataset_X_test`. Compare the predictions with true labels. How well does the model perform? The task is more complex than bankruptcy prediction, the data is noisier, hence the classification might be imperfect. In order to improve the classification, you should play with the feature sets, train-test ratios etc.
The full code can be found on github in file *LI-classification.py*.

[slide 20]
Now you are fully equipped to play the Predictive Text game on the platform! The rules of the game are the following: given a set of English sentences where the words of length *2* are left our as blank spaces, complete the sentences by guessing the correct words. There are only so many two-letter English words that are commonly used: *of, to, on, it, is, if, in, he, we, be, up, at, as, by*.
Hence, the problem is an instance of multi-class classification! Good luck!