Certificate 2 Session 3 — Script

[slide 1]
Today we are going to continue studying Natural Language Processing (NLP) and will prepare ourselves to play Predictive Text game.

[slide 2]
Last session we talked about an NLP task of language identification: given a text string, guess in which language it is written. In order to apply ML to the task, we needed to extract formal features from the text, which we did using ngrams. Finally, we chose to use Logistic Regression as the classification algorithm, and we investigated how to train it.

[slide 3]
Today we are going to study the problem in the Predictive Text game. 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*. The only constraint is: if your opponent is already assigned some word to a slot, you cannot do the same pick. You can see the detailed rule description here: https://www.aigaming.com/Help?url=PredictiveText

[slide 4]
One peculiarity of the game is that not only there can be several empty slots in a sentence, but also several empty slots in a row! This means that we cannot simply convert a sentence into a learning instance and guess all the words simultaneously. Instead, we need to take into account that slots are *interdependent*. Hence, we ned a learning algorithm that would take that into account.

[slide 5]
In supervised machine learning, there are two types of algorithms and models they produce (use, yet another dichotomy!). *Discriminative* models only care to differentiate between the classes. They seek to learn the boundary that separates the two classes, and in terms of probabilities, they try to learn a conditional probability of *x* being of class *y*: $p(y|x)$. On the other hand, *generative* models, instead of learning the separating line, learn the classes themselves. They *model* the data in the classes, and in principle, they can generate new potential instances of the classes, hence the name. In terms of probabilities, generative models approximate probabilities $p(x|y)$ (given the class *y*, how would its representative look like?) and $p(y)$ (how likely is it to be in class *y*?). From those two probabilities it is easy to derive the $p(y|x)$ which is then necessary for classification.

[slide 6]
Aaand one more classification of algorithms and models. Last one for this course! Many algorithms rely on the so called *i.i.d.* assumption, which stands for *independent and identically distributed* instances (or data points). Intuitively, this means that when we classify instances, classifying i_1 does not depend on the outcome of classifying i_2, but both classification decisions rely on the same data distribution. For instance, in language identification, all Swedish sentences follow the laws of the Swedish language, and deciding on the first sentence does not affect the other two.
But this is not the case for Predictive Text. in a phrase “until __ __ one inch below” it is crucial to assign the right word for the first slot in order to correctly classify the second one: “until it is one inch below”. The decisions depend on each other. Hence, this type of learning is called *structured*, or *sequence* learning.

[slide 7]
Let's put everything together. We have already seen simple discriminative model — Logistic Regression. Its generative counterpart would be, e.g., the Naive Bayes algorithm. As s structured discriminative algorithm one could pick CRFs, and the structured counterpart of Naive Bayes is Markov Models. This is want we need.

[slide 8]
How to model a sequence (of text)? The probability of a word sequence to appear in text is the product of probabilities, or simply frequencies, of its subsequences; and the probability of a subsequence is the conditional probability of its last word given the preceding words.
[Go carefully through the formula]
In practice, one seldom depends on all the preceding words, but rather on some of them, and this is defined by the size of the history window. For instance, if the window size is *2*, the probability of a sequence "I want to go to London" equals *freq*(I | `start`) * *freq*(want | `start`,I) * *freq*(to | I,want) * *freq*(go | want,to) * *freq*(to | to, go) * *freq*(London | go,to) * *freq*(`stop` | to,London).
Note that sometimes the start and the end of a sentence are included into the analysis as parts of the sequence. We are not going to do it today.

[slide 9]
Since this frequency-based model is generative, we can predict the missing words using it: for every slot we should try and fill it with all possible two-letter words, calculate probabilities of the resulting sequences, and pick the one with the highest probability. In other words, we are *maximising* the probability of a sequence being part of the language by filling in the missing words. In math notation it is captured by the *argmax* function.

[slide 10]
All the sentences from the Predictive Text games are test instances for us. In fact, they will definitely be not seen by our model since the game is made in such a way that one cannot google the sentences and hack the solution.
Hence, we need to train our language model on some external corpus of English language. There are numerous corpora available online and used in the area of NLP: https://en.wikipedia.org/wiki/List_of_text_corpora. In principle, the bigger the corpus, the better, since it can capture mode sequences, and in more detail. But bear in mind that even a big corpus may contain noise, may be unbalanced in terms of vocabulary, syntax, topics, genre etc.

[slide 11]
Today we are going to use a fragment of the NOW corpus of English language. You can read more about it here: www.corpusdata.org/now_corpus.asp For us what matters is that the corpus us general domain, consists of "proper" texts (as opposed to, say, tweets) and is of sufficient size, namely around *2* million words. You can download the corpus from our github.
We are not going to do sophisticated processing of the text and simply split it in words (*tokens*) using regular expressions: `corpus = re.split(" ", input)`. Feel free to do the necessary pre-processing yourselves.

[slide 12]
How are we going to compute probabilities and frequencies over the corpus? Very simply, a frequency of a word w_i appearing after a sequence $w_1,\ldots,w_{i-1}$ is the number of times the sequence $w_1,\ldots,w_i$ appears in the corpus divided by the number of times the subsequence $w_1,\ldots,w_{i-1}$ appears in the corpus. A simple proportion. In case of a single word it is the number of times it appears int eh corpus divided by the size of the corpus. These frequencies can then be multiplied to calculate the probability of a phrase or a sentence.
There is one caveat here. If a certain subsequence of words was not seen in the corpus, the whole product with amount to *0*, and this is not very informative (and in fact will happen in many cases). To avoid it, the simplest technique is called *smoothing*, and it amounts to increasing all counts, for all sequences, by one.

[slide 13]
In the Predictive Text game we need to be fast, and calculating probabilities can take some time. Hence, to speed up, we will precompute the counts for all subsequences (of fixed size, defined by

the history window) and save them into a file. At running time, we will access sequence counts from the file, and if the sequence is not seen before, set its count to *1* (remember the smoothing technique). We do not want the file to be too big and heavy either, so to save space, we will only save counts for sequences that appear more than a certain number of times. Unsurprisingly, setting such an threshold even to *1* (one appearance in the corpus) will significantly save the time and speed up the computation. Items with low occurrences are called *the long tail*, and it usually corresponds to the larger part of all sequences.

[slide 14]
You can compute counts yourself, setting your own threshold, or you can download counts for window size *2* from the github: *corpus-counts.txt*.
It is important to note here that we store sequences as keys and counts as their values in a dictionary while the counts are computed, and then we save the dictionary, or rather its string representation, into a file: `file1.write(str(top_frequencies))`. This way we can later load the dictionary as a file content and interpret it as a Python dictionary…

[slide 15]
… using the `eval()` function: `file2 = open('corpus-counts.txt','r'); dictionary = eval(file2.read())`. Once the counts are precomputed, it is only necessary to load them and not the corpus. However, we still need the information about the corpus size, so we save it explicitly into a variable: `corpus_size = 1959580`. We also store the history window and the two-letter words we consider.
Computing the sequence probability is defined in the `getSequenceProbability()` function: going through the sequence, we generate all subsequences of the necessary size, then we convert them into tuples (since tuples in Python are immutable, hence hashable) and query our dictionary with counts. Having the counts, we divide them by the corpus size and we iteratively update the product which eventually becomes the sequence probability.

[slide 16]
The last thing remaining is to predict values for the empty slot. First we generate all possible instantiations of an input sentence by generating all combinations of slot values. This is done by a *recursive* function `generatePotentialSequences()`. [go through the code thoroughly, as I am not sure whether they are familiar with recursion]
Then for every potential sequence we calculate its probability, and pick the sequence with the highest probability.

[slide 17]
For instance, for the sentence "I want to __ __ London" the highest ranking option using our corpus and minimal pre-processing is "I want to be in London". Not too bad!