TME 286 (Interpretable artificial intelligence)

Assignments

General information

You will find all the assignments below. You can choose the level of ambition yourselves, as specified below.

Requirements for grade 3 (Chalmers) or grade G (GU):

The minimum requirement (for a passing grade) is to hand in satisfactory solutions to the (mandatory parts of the) problems marked as *mandatory*. There is no lower limit on the number of points (for a passing grade), but the solutions must be satisfactory (see the individual problems for a description of point deductions in case of required resubmissions). You have as many attempts as you need for the mandatory problems, but we only correct a resubmission if you have actually made the changes specified in the comments that you receive – if those changes have not been made, the solution is simply returned to you again (with further point loss).

Requirements for grades 4 and 5 (Chalmers) and grade VG (GU):

In order to obtain a higher grade (than 3 or G), you must solve some of the voluntary problems. The point requirements for the various grades are as follows:

Grade 4 (Chalmers): 60 p or more

Grade 5 (Chalmers): 80 p or more

Grade VG (GU): 72 p or more

Requirements (on submitted solutions)

Writing is the extension of thinking: You must write the text cells (describing your work) yourself. That is, you cannot hand in text written by *someone* else (e.g., another student) or *something* else (e.g., an LLM). Notebooks that do not fulfil this criterion will be returned uncorrected (with a point loss) and will need to be resubmitted. Note that even in code cells, it is good if you add some clarifying comments (in your code), and you may also include any number of clarifying text cells.

For the coding, you may ask an LLM for advice but may not hand in entirely LLM-written code. In any case, you are solely responsible for the code that you hand in (whether or not you use an LLM for assistance). You must be able to explain and defend every line of code that you hand in.

Furthermore, for each assignment *you must add a text cell at the top of your notebook, explaining how, and to what extent, you have used LLMs when solving the assignment*. This text cell should also contain *your full name and civic registration number*. *Solutions that lack this cell (or contain only partial information) will be returned unseen, with an accompanying point loss* (see below).

We will run a plagiarism check on all submitted solutions and also pass the text cells (of your hand-ins) through an LLM detector (which was written by Minerva and is very accurate).

Submissions:

- **Important:** You should hand in all assignments at the same time, collected in a single compressed file (in either .zip or .7z format). When decompressed, the file should generate folders Assignment1/, Assignment2/ … and so on.

Incomplete and erroneous submissions:

- Again, hand in all assignments as a unit, i.e., as a single compressed file. The submission time for the entire set (all assignments) will be taken as the time of the last hand-in (so, there should be only one hand-in, preferably on time! See also late hand-ins below.
- Submissions that do not fulfil the criteria described above (and on the assignment page in Canvas) will be returned unseen, with a loss of one (1) point.
- Individual assignments may also be returned (if so, with a loss of two (2) points for every time the assignment is returned), as specified in each assignment. This applies to the mandatory assignments. The voluntary assignments will only be corrected once – no resubmissions possible.
- Note that any assignment that you hand in must contain a complete, serious attempt at solving the problem in question – partial, incomplete solutions do not count. Exception: For those mandatory assignments that contain voluntary parts at the end, it is allowed to hand in only the mandatory parts (but one can then *not* hand in the remaining parts later – resubmissions will only concern the parts handed in the first time).

Late hand-ins:

- We correct solutions and resubmissions handed in during SP3, until 20260315, 23.59.59. Any resubmission received from 20260316 and onwards will be corrected either in the re-exam period in August or in the re-exam period in December (depending on when the resubmission is handed in).
- Voluntary assignments must be handed in no later than 20260315. After that date, we only accept mandatory assignments (including their voluntary parts, where applicable).
- Mandatory assignments that are handed in (for the first time) after 20260315 incur a loss of one (1) point (once and for all, for each assignment), though, so *strive to hand in the assignments on time*!
- During SP3 and until 20260315, 23.59.59, you should hand in solutions and resubmissions via Canvas. Resubmissions that are handed in on or after 20260316 should be mailed to Mattias. Note: Use your Chalmers' or GU e-mail address in the latter case. Solutions received from other e-mail addresses will not be considered. In the subject line of the e-mail specify *clearly* (i) which course the assignment pertains to (TME286, 2026) and (ii) which assignment you are resubmitting. E-mails that lack this information (in the subject line) will not be considered. If you are resubmitting multiple assignments, write one such e-mail per assignment. Remember also to always specify your full name and civic registration number in your Jupyter notebooks; see above.

Assignment 1 (Mandatory, 15p)

Time series prediction (black-box vs. interpretable)

In this assignment we will consider (single-step) time series prediction applied to a given time series based on a (known) generative function, comparing glass-box and black-box prediction methods.

Do the following:

1. Write Python code for reading and visualizing the time series data contained in the files A1_Training, A1_Validation.txt, and A1_Test.txt.
2. Fit an ARMA model to the training time series (A1_Training.txt). There are many solvers for ARMA models, but some of them have difficulties converging for this problem. Here, you should use the Nelder-Mead approach (which is derivative-free and is available in the Python package scipy.optimize: Set the method as 'Nelder-Mead'). The output from your program should be a plot showing the test data series (A1_Test.txt) as well as the fit that was obtained from the ARMA model. over that set. The root mean-square error (RMSE) should also be printed (as text).
3. An important aspect when fitting a linear (ARMA) model is to find suitable settings, i.e., the values of p (the order of the AR part) and q (the order of the MA part). One can do so by investigating the autocorrelation function (ACF) and the partial autocorrelation function (PACF) of the series. Do this, and try to identify suitable values of p and q (neither parameters should be larger than 4). Include the ACF and PACF plots in your report. However, in the end, one can use a simpler approach for these time series (especially since they are not really well-fitted by ARMA): Select the values of p and q for which the RMSE over the *validation* set is minimal (after fitting the ARMA model over the training set).
4. Next, train a single-layer LSTM network to predict the data series (for example using the LSTM subpackage of Keras). Here, use the Adam optimizer with the MSE loss function. Use the tanh activation function for the LSTM units. Use holdout validation, i.e., train over the training set and use the loss over the validation set to determine when to stop training. Store the network with the smallest validation loss and then run it over the test set. This program should also plot the test time series as well as the predicted data that is obtained from the LSTM model and should print the RMSE as text. As in the case of the ARMA model, the results depend on the structure of the network (and may also vary from run to run). Hence, experiment with a few different LSTM networks (varying, for instance, the number of LSTM units).
5. An advantage with the linear model is that one can easily study and understand its structure. Consider your best ARMA model (Step 3), and provide a brief analysis of its structure (what previous time steps are most important? Which parameters are positive and which are negative? Why? …and so on). Also, based both on the performance (RMSE error) and the interpretability, add a discussion comparing the two approaches (ARMA vs. LSTM) for this case.

What to hand in

You should hand in a Jupyter notebook (no other formats allowed) that contains *all* the required code for running through all the steps listed above (including any necessary installations with "pip install"), i.e., loading and visualizing the data sets (time series), generating, optimizing, and evaluating ARMA models (with different settings, i.e., several models – include all in the notebook), plotting the output (and the ground truth series) for the test set as well as giving the RMSE error (over the test set), and noting which ARMA settings work best, then generating, optimizing, and evaluating the LSTM networks, using different network structures (number of LSTM units), and plotting the output and providing the RMSE error (for the test set). Then, as the final step, include the comparative discussion (Step 5).

In general, you should include text fields (not just code) in your notebook that provide a *clear* description of the steps that you take (as well as any other relevant observations that you make along the way). Do *not* use LLMs when writing such text.

Note: You must make sure that your notebook can run (in its entirety) in Google Colab, without any tweaking or adjustment, and without using any other tools than Colab.

Grading

The entire problem (Parts 1 – 5) is mandatory.

If the problem is returned (for whatever reason), you lose 2p for each such iteration.

Assignment 2 (Mandatory, 15p)

Image classification (black-box vs. interpretable)

In this assignment you will compare black-box (CNN) image classifiers to interpretable classifiers for the task of image classification. You will use two standard data sets that have been used frequently in image classification, namely MNIST (containing hand-written digits, with 10 classes) and CIFAR-10 (containing color images of cars, dogs, etc., with a total of 10 classes). Image classification is a subfield where DNN-based classifiers dominate strongly, especially CNNs and (more recently) vision-transformers. However, while the performance of neural models is generally very good, they also suffer from a certain brittleness, such that even small amounts of noise can lead to completely incorrect classifications (as explored in Assignment 4 below).

An alternative approach is to use an interpretable image classifier, for which there are many possible options. In this case, we will use a simple k-nearest neighbor (kNN) classifier, but with a tweak that can improve its performance.

Now, do the following:

1. Download the MNIST and CIFAR-10 data sets. Make sure to normalize the data as follows:
    a. For MNIST: Use min-max scaling, so that the values (for each pixel) are in the (floating-point) range [0,1] (in the raw data set, each pixel takes an integer grayscale value in the range [0,255]).
    b. For CIFAR-10 use Z-score normalization (mean = 0, standard deviation = 1).

2. For MNIST, set up and train (with the Adam optimizer) a CNN with the following structure:
    a. First layer: Convolution layer with 16-64 filters (3x3), ReLu activation, and batch normalization.
    b. Second layer: Max pooling (2x2)
    c. Third layer: Convolution layer with 32-128 filters (3x3), ReLu activation, and batch normalization.
    d. Fourth layer: Max pooling (2x2)
    e. Fifth layer: (after first flattening the output from the fourth layer): A fully connected (“dense”) layer with 128 neurons, ReLu activation (and dropout during training, with a suitable dropout range (somewhere in [0.2,0.5]).
    f. Sixth layer: A layer with 10 neurons (= the number of classes) and a softmax activation function.

    NOTE: For MNIST, the border pixels do not really matter, so you do not need to use padding.

3. For CIFAR-10, set up and train (again using the Adam optimizer) a CNN for which you select the structure yourself. Since the CIFAR-10 images are (much) more difficult to classify, the network must be bigger (more layers) than for MNIST and you *should* use padding (when doing convolutions) since, here, the border pixels do matter in many cases. You can experiment with a few different structures (and describe the structures that you tried, in your report). However, you need only give *results* for the best structure (= highest test accuracy) in your report.

   Note: For this case, the input shape (for the convolution kernels) must be 32 x 32 x 3, where the "3" accounts for the fact that the images are in color (i.e,, with three channels, R, G, B).

4. Next, implement kNN models, one for classifying MNIST and one for CIFAR-10 images (with the same normalization as above, for each data set). Use the standard Euclidean distance measure (applied pixel-by-pixel). Then run each model with different values of k (from 1 to 25, say), and pick the value of k (one for each data set) for which the *validation* accuracy is maximal. Then use that value to obtain the test accuracy. This is the simplest possible way to apply the kNN classifier, and it works reasonably well for MNIST (~0.97 accuracy) but much worse for CIFAR-10 (~0.35 accuracy).

5. Improve the kNN code such that, for each classified (test) image, the program shows the image (to be classified) along with the k nearest neighbors – with this interpretability feature one can easily see *why* an image is correctly (or incorrectly) classified.

6. **(Voluntary)** One of the reasons that the kNN classifier gets a rather poor result (for CIFAR-10) is that the individual pixels provide a poor representation of the content of an image (no matter how one computes the distance measure). In order to improve the kNN classifier, while maintaining interpretability to a great degree, one can use an approach that extracts so-called "bag of visual words" features (see also the notes from Lecture 6), i.e., considering (as in a CNN) more high-level features and patches over the images. Thus, in this step, you should implement the *bag of visual words* approach: Extract image patches, make a dictionary (codebook) using, for example, K-means clustering, then make a histogram, and compute the (Euclidean) distance of the histograms (for image pairs), then use the k-nearest histograms to classify the image (again, see also the slides from Lecture 6). With this approach, the accuracy over MNIST should go up to around 0.98 and around 0.65-0.70 for CIFAR-10.

Notes:

(i) For MNIST, the target (test) accuracy for the CNN model is in the range 0.990-0.995. This is usually achieved within a few epochs (< 10).

(ii) For CIFAR-10 the target (test) accuracy is around 0.80-0.85 or so, and you may have to run it for a while (longer than for MNIST) to get such levels of accuracy.

(iii) Even higher accuracy can be obtained (especially for CIFAR-10) by augmenting the data, but we will not do that here.

(iv) The performance of the bag-of-visual-words approach can be improved further by also considering so called spatial pyramids (essentially keeping track of the approximate *location* of a feature). However, you need not use that feature here.

What to hand in

You should hand in a Jupyter notebook (no other formats allowed) that contains *all* the required code for running all the experiments above (including any necessary installations with "pip install"), i.e., loading and normalizing the two data sets, generating and optimizing the two CNNs (one for MNIST, one for CIFAR-10), implementing and running the kNN approach with the standard Euclidean distance measure (determining the optimal value of k), and implementing the visualization for the kNN approach. If you choose to do part 6 as well, you must include all relevant code for that part also.

In your notebook, you should also include text fields (not just code) that provide a *clear* description of the steps that you take (as well as any other relevant observations that you make along the way). Do *not* use LLMs when writing this text.

Note: You must make sure that your notebook can run (in its entirety) in Google Colab, without any tweaking or adjustment, and without using any other tools than Colab.

Grading

Parts 1-5 of the problem are mandatory and are worth a maximum of 11p (of which 8p for the code, and 3 for the descriptive text)

Part 6 is voluntary (and is worth a maximum of 4 p) (of which 3p for the code, and 1 for the descriptive text).

If the problem is returned (for whatever reason), you lose 2p for each such iteration.

Assignment 3 (Mandatory, 15p)

Text classification (black-box vs. interpretable)

In this assignment you will compare black-box models (specifically a fine-tuned version of BERT) to interpretable classifiers for the task of text (sentiment) classification. For this task, it is often claimed that transformer models (such as BERT and its versions) easily outperform classical, interpretable classifiers (e.g., linear perceptrons with n-gram features). However, one should bear in mind that BERT has been trained over massive amounts of text and thus has a great advantage in the form of a general knowledge of English (for example being familiar with synonym relationships, idiomatic expressions, and so on), whereas the classical classifiers are typically only trained over the specific data set under consideration. Thus, here after first comparing the black-box model's performance to that of a classical model, you will extend the classical model by giving it additional data, thus making the comparison (with the black-box model) less unfair.

Now, do the following:

1. Download the ReviewBase data set that, in turn, contains a training set, a validation set, and a test set (for details, see the assignment text on Canvas). The data sets have been preprocessed, so that you only need to split the data on the space character in order to tokenize it. Note, however, that the data set also contains a class label associated with each text.

2. Set up and fine-tune the DeBERTa-v3 model (a state-of-the-art version of BERT for text classification), using low-rank adaptation (LoRA) for fine-tuning with great efficiency (only modifying a small fraction of the parameters). Use a single fully connected layer (after the transformer blocks) as the final classification head (this is the default option - a single layer is sufficient here, adding more layers may lead to overfitting).

3. Next, set up and train a linear perceptron using n-gram features for the same task. Then run the training for $n_{\max}$ = 1 (unigrams only), $n_{\max}$ = 2 (unigrams and bigrams) and $n_{\max}$ = 3 (unigrams, bigrams, and trigrams). For each value of n, you should also try different minimum instance counts ($c_{\min}$) for feature inclusion, e.g. 1,3,10,100. Then (for each value of n) extract the model with the highest validation accuracy and compute its accuracy over the test set.

4. While the DeBERTa-v3 model outperforms the linear perceptron models, its decision-making is entirely opaque, whereas the decision-making of the linear perceptron models can easily be visualized in different ways, thereby potentially providing important information about the underlying reasons for a given class assignment. You should write code that produces a plot (a diagram), showing all the unigrams, bigrams, and trigrams, with a green bar for positive weights and a red bar for negative weights, such that the length of the bars is proportional to the

weight magnitude. Then apply the visualizer to a few (3 – 5, say) short reviews, with at most 15-25 words. (Longer reviews can also be visualized, but the plots become very large). Note that n-grams that do not appear in the text should still be shown in the plot (but without any green or red bars), making it possible to assess the coverage ratio (see the notes from Lecture 6). For each linear perceptron model, compute the coverage ratio over the test set.

Add a brief discussion comparing the DeBERTa-v3 model to the linear perceptron models both in terms of performance and in terms of interpretability. NOTE: If you only do the mandatory parts (Steps 1-4), add the discussion here. However, if you choose to do Steps 5-6, you should add this discussion at the very end, after Step 6 (including also the extended classifier, and its performance, in the discussion).

5. (**Voluntary**) In order to make a fairer comparison, one can extend the training set for the perceptron models, thereby increasing their coverage ratio and improving their performance. Thus, consider now ReviewExtended data set that is considerably larger than the base set considered above, and is composed of a training set and a validation set (no test set needed here). Next, train linear perceptron models with $n_{\max}$= 1 (i.e., with unigrams only), $n_{\max}$ (unigrams and bigrams), and $n_{\max} = 3$ (unigrams, bigrams, and trigrams), as above, over the extended set, again extracting the model with the highest validation accuracy (over the validation set for the extended set) for each n. Next, for each n, average the weights from the original base model (Step 3) with the weights obtained for the extended model (Step 5) as $w\_c = \alpha w\_b + (1 - \alpha)\, w\_e$, where w_b denotes a weight (for a given n-gram) from the base model and w_e denotes a weight (for the same n-gram) from the extended model. In cases where a given n-gram only exists in the extended model, use that weight without any averaging (i.e., set $w\_c = w\_e$) and in the (unlikely) case where a given n-gram only exists in the base model, use $w\_c = w\_b$. Then try a few different values of α, selecting (for each value of n) the value that maximizes the accuracy over the (note!) *validation set for from the base (not extended) data set*.

6. (**Voluntary**) Then, finally, using the models thus obtained (with optimized values of α) compute the accuracy (for each of the three combined linear perceptron models) over the test set (from the base data set). Make a table showing the accuracy of (i) the DeBERTa-v3 model, (ii) the base perceptron models (Step 3) and (iii) the extended perceptron models (Steps 5-6). Also, plot again the visualization (see Step 4) for the same text samples that were used in Step 4, noting the increase in the coverage ratio (relative to the base perceptrons). Then, compute the coverage ratio (for each of the three extended models) over the test set (from the base data set). Next, analyze and discuss the difference in coverage ratio between the extended perceptron models and the base perceptron models. Moreover, add a general discussion regarding the comparison of the three model types (DeBERTa-v3, base perceptron models, and extended perceptron models).

What to hand in

You should hand in a Jupyter notebook that contains all the required code for running all the experiments above (including any necessary installations with "pip install"), i.e., loading and tokenizing the data sets, loading and fine-tuning the DeBERTa-v3 model (and obtaining its test performance), implementing and running the linear perceptron models (for different values of n) and obtaining their performance scores, and making the visualization (interpretability) plots (for the selected data items. If you choose to do parts 5-6 as well, you must include all relevant code for that part also.

In your notebook, you should also include text fields (not just code) that provide a *clear* description of the steps that you take, as well as the comparative discussion(s) mentioned above. Do *not* use LLMs when writing this text.

Note: You must make sure that your notebook can run (in its entirety) in Google Colab, without any tweaking or adjustment, and without using any other tools than Colab.

Grading

Parts 1-4 of the problem are mandatory and are worth a maximum of 10p (of which 8p for the code, and 2 for the descriptive text)

Parts 5-6 are voluntary (and are worth a maximum of 5 p) (of which 4p for the code, and 1 for the descriptive text).

If the problem is returned (for whatever reason), you lose 2p for each such iteration.

Assignment 4 (Voluntary, 15p)

Adversarial attacks against black-box image classifiers

In this assignment, we will consider the brittleness of neural image classifiers (CNNs, in this case), applied to the case of the simple MNIST images (hand-written digits). Such networks can be subjected to *adversarial attacks*, i.e., deliberate attempts to make the CNN misclassify images. Adversarial attacks are of two kinds: White-box attacks, where the attacker has full knowledge of the CNN (its structure, weights, and loss function), and black-box attacks, in which the attacker does not have access to the internal structure of the model, but can present an input image and obtain an output (either a class label or a class probability). There is also the concept of adversarial *training*, in which one tries to make a neural network more robust to adversarial attacks (but we will not explore that topic here).

Now, do the following:

1. Generate Python code for reading the MNIST data set and setting up a CNN with a suitable structure (which you can choose yourself). Then train the CNN so that it achieves a high accuracy for the MNIST data set (~98% test accuracy, or better).

2. Brute-force black-box attack: One of the most straightforward ways to carry out a black-box attack is to use a *greedy random search*. Here, starting from a correctly classified image, one adds salt-and-pepper noise (randomly changing a few pixels) step-by-step, until the image is misclassified. More specifically, for a given step, after changing the image as just described, one keeps the modified image if the probability for the ground truth class C is reduced after the modification. If not, the previous image is retained, and noise is added again, and so on. Implement and run this approach (visualization will follow in Step 5 below) over a few MNIST images.

3. More sophisticated black-box attack: Another approach is the so-*called boundary attack*. In this method one starts with a target image T with ground truth class label C_T and another image I that is classified (label C_I) as something else than the target class (this image can either be another digit or a random pattern that just happens to trigger a class label different from the target class). One then tweaks the image (I) towards T, i.e., making it more resemble the target image. If the CNN still classifies the image as belonging to class C_I, the perturbation was successful. Otherwise, the step is rejected, and a smaller step is tried, and so on. Implement and run this approach over a few MNIST images.

4. White-box attacks: Here, too, there are many different approaches. We will consider only one, namely the *Fast gradient sign method* (FSGM). Since this is a white-box attack method, with full insight into the CNN model, one can simply reverse the training, in a manner of speaking: The weights of the CNN are not changed, but the image is modified so as to *maximize* the loss function, trying to make the CNN as wrong as possible. In equation form, one generates an image x_adv using

$$x_{adv} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

where x is the original image, ϵ is a small parameter, J is the loss function, and the gradient is taken with respect to the input pixels (determining which direction to follow in order to maximize the error). Implement and run this method over a few MNIST images.

5. Finally, write visualization code so that, starting from a given MNIST image, your Jupyter notebook runs through all the methods listed above, plotting (as final output) the original image, the three adversarial images (i.e., the final output from each of the three methods), as well as the starting image for the boundary attack (Step 3). Then also discuss your findings and write down some conclusions regarding adversarial attacks.

Note: You may re-use parts of your own code from Assignment 2 if you wish, but the notebook for this assignment (Assignment 4) should nevertheless be complete, i.e., it should contain *all* the necessary code for solving the entire assignment.

What to hand in

You should hand in a Jupyter notebook that contains all the required code for running all the experiments above (including any necessary installations with "pip install"), i.e., loading the MNIST data set sets, setting up, training, and testing the three different attack methods, visualizing the output, and discussing your findings.

In your notebook, you should also include text fields (not just code) that provide a *clear* description of the steps that you take, as well as the discussion(s) mentioned for Step 5. Do *not* use LLMs when writing this text.

Note: You must make sure that your notebook can run (in its entirety) in Google Colab, without any tweaking or adjustment, and without using any other tools than Colab.

Grading

In order for the problem to be graded (at all), you must complete all five steps and hand in the Jupyter notebook as described above.

Since the problem is voluntary, it will be graded *once* – no resubmissions allowed.

Assignment 5 (Voluntary, 20p)

Language models: LSTM models vs. n-gram models

In this assignment you will compare black-box language models (namely LSTM models) and interpretable (n-gram) language models. In general, large language models are built on the transformer architecture. However, those models require very large training sets. Here, we will consider a standard benchmark data set, namely the Wall Street Journal (WSJ) section of the Penn Treebank (PTB) data set. This set consists of a training set (~1 million tokens), a validation set (~73,000 tokens), and a test set (~82,000 tokens) and the vocabulary has been limited to 10,000 words (less frequent words are represented by an *unknown* (<unk>) token). Moreover, numbers have been replaced by a generic *N* token. Punctuation has been removed, but then an end-of-sentence (<eos>) token has been added at the end of each sentence. For this rather small data set, LSTM models perform better than transformers, reaching (test) perplexity values of roughly 75-100 (for standard LSTM models) and down to around 45-50 for various versions of LSTMs. n-gram models, by contrast, reach test perplexities of around 154-180. However, with some tweaking, the performance of the n-gram models can be improved a bit.

Do the following:

1. Implement (using existing code) an LSTM model, and then train it (using holdout validation) over the PTB data set. You may vary the exact structure of the LSTM model, i.e., the number of units and the number of stacked LSTM layers (the default value is 1), and you must obtain a test perplexity of 80 or below (lower is better).

2. Implement a smoothed trigram model (with fixed smoothing, i.e., with constant smoothing parameters, i.e., a model of the form

$$P_{\text{tri}}(w_i|w_{i-2},w_{i-1}) = c_1 P(w_i|w_{i-2},w_{i-1}) + c_2 P(w_i|w_{i-1}) + (1-c\_1-c\_2)P(w\_i)$$

   Then optimize the two smoothing parameters by trial and error in order to get the lowest *validation* perplexity. Next measure the test perplexity (should be around 170 or so).

3. Then improve the trigram model as follows: Make the smoothing parameters dependent on the number of instances (in the training set) of the preceding bigram (i.e., up to, but not including, the predicted token). For example, one can bin the counts (of the preceding bigrams) into k bins ($k = 5 - 25$ say) and use one set of parameters c_1, c_2 for each bigram count. Here, you should expect a perplexity drop of a few units (relative to the model in Step 2).

4. Then tweak the improved trigram model (from Step 3) to include unigram caching, i.e., a modification of the probability distribution such that words that appeared among the m last words (during evaluation) are given an increased probability. That is, the model should take the form

$$P(w_i|w_{i-2}, w_{i-1}, \text{cache}) = (1-\lambda)P_{tri}(w_i|w_{i-2}, w_{i-1}) + \lambda P_{cache}(w_i)$$

   for some (rather small) value of λ, where the P_cache distribution is thus built from a moving window of the m most recent preceding words. Here, you should find test perplexity values of around 135 or perhaps a bit lower.

5. Many studies of language models focus entirely on overall metrics (such as, for example, the perplexity score), but it is also of great interest to study what type of sentences a given model actually generates as output. Thus, to that end, write code for the LSTM model (from Step 1) and the final n-gram model (from Step 4). so that they can be used in generative manner. That is, given an input prompt that represents the start of a sentence, the model should produce a sequence of output tokens (where every new output token is then also added as context for the *next* step) until an end-of-sentence (EOS) token is produced. Then define a few (3-5) input prompts (using words and phrases that appear rather frequently in the PTB vocabulary) and generate some (5-10) examples of sentences (for each of the two models) with those prompts. Then compare the sentences from the two models. Can one spot any significant improvement in the sentences generated by the LSTM, relative to those generated by the best n-gram model? Discuss your findings (and include all generated sentences, for both models, in your notebook!)

What to hand in

You should hand in a Jupyter notebook that contains all the required code for running all the experiments above (including any necessary installations with "pip install"), i.e., loading and tokenizing the PTB data set sets, setting up, training, and testing the LSTM model, implementing and running the three n-gram models (Steps 2-4), and then finally generating the output sentences.

In your notebook, you should also include text fields (not just code) that provide a *clear* description of the steps that you take, as well as the discussion(s) mentioned for Step 5. Do *not* use LLMs when writing this text.

Note: You must make sure that your notebook can run (in its entirety) in Google Colab, without any tweaking or adjustment, and without using any other tools than Colab.

Grading

In order for the problem to be graded (at all), you must complete all five steps and hand in the Jupyter notebook as described above. Since the problem is voluntary, it will be graded *once* – no resubmissions allowed.

Assignment 6 (Voluntary, 20p)

Financial time series prediction (black-box vs. interpretable)

In this assignment we will again consider time series prediction, comparing black-box models (namely LSTM models) and interpretable models (in this case evolvable rule-based models). In standard time series prediction, using the sequence of preceding values, one tries to predict the next value (price, in this case) *for every time step*. Here, we will instead make predictions only for *certain* time steps, the rationale being that, in financial trading, it is neither realistic (given the noise levels) nor desirable to predict the next day's price for every stock, every day. Instead, the aim is to make predictions only in cases where the model is reasonably sure of a positive outcome. Note: we will consider only positive price changes in our predictions ("long-only" in financial terminology) – we will thus not make predictions of negative price changes (price drops). With this focus, we effectively turn a regression problem into a classification problem: *Which preceding sequences are indicative of a rapid rise (a breakout) in a stock price over the next period*?

Do the following

1. Download (daily) financial data for a set of (around) 50-100 stocks, covering a time period of 10 years (or more) For each day, store the date (yyyymmdd), the adjusted close price (i.e., the last price of the day, adjusted for dividends and splits) and the volume (number of shares traded). (Free) data of this kind can be found in many places online, for example at finance.yahoo.com. It is also possible to download the data directly into Excel using its STOCKHISTORY command, but that requires a "Microsoft 365" subscription. The easiest option (recommended), however, is probably to use the Python library *yfinance*.

2. Divide your data into a training set, a validation set, and a test set. Each set should contain data from all the selected stocks. The test set should cover the period 20250101 until now, the validation set should cover the period 20240101 – 20241231, and the training set should contain all the remaining data, i.e., from 2015 (or so) until 20231231. This is not the only way to divide the data (i.e., with the test set containing the most recent data, and so on), and not necessarily the best, but it is sufficient for our purposes here.

3. Next, process the data, adding another column (for each time series, and in each set) where entries take the integer value 1 if the closing stock price on the next day is at least 3 % higher than the price on the current day, and 0 otherwise. Note **(Very important!)** the value 1 should be assigned to the row that corresponds to the current day (at the end of which the prediction is made) not for the predicted day (of which we have no knowledge when the prediction is made).

4. Then set up and train an LSTM network so as to minimize the difference between its output (a number in the range [0,1]) and the desired output, i.e., either 0 or 1, as specified in Step 3, given price and volume data as input. For any given stock, use the first 10 (say) days as a warmup period for the LSTM (i.e., do not use the output from those days), but for all the remaining days (excluding the last one, for which no prediction can be made, since no ground truth value is available) compute the error as just described. Then combine the list of errors for each stock to a total list of errors, which should be used (after normalization) in the loss function when training the LSTM network. Use holdout validation and select the network for which the error (loss) for the *validation* set is smallest. Then run the selected network over the test set as follows: Whenever the LSTM outputs a value above a certain threshold (in the range [0,1], here assuming that this is the range for the output neuron), output a 1 (i.e., make a prediction that the price will rise), if not, output a 0 (i.e., make no prediction). Then, for the days where predictions are made, *sum* the profits obtained (here defined as $(p\_(t+1)/p\_t) - 1$, where p_t is the price on day t (i.e., when the prediction is made), and normalize by $N \, x \, M$, where N is the number of stocks and M is the number of time steps considered for each time series (so that the value obtained represents the average profit per day). You can adjust the threshold using the validation set, but *not* the test set – that is used only for final evaluation, no tuning.

5. Next, set up and train an evolvable rule-based system, consisting of rules of the form *IF (condition)*, where the conditions involve ratios of prices (at different days) or ratios of volumes (at different days). The rules should output True in those cases where the condition is fulfilled, and False otherwise. Moreover, the rules should be organized into (evolvable) groups such that, if all rules in a given group are True, the group outputs a 1 (i.e., predicting a price breakout). At every time step, all rule groups should be evaluated and if at least one of them outputs 1, the system (as a whole) outputs a 1 as well (i.e., predicting a price breakout), otherwise the output from the system should be 0. Apply an evolutionary algorithm that can (i) change both the parameters (conditions) of individual rules, as well as the number of rules in each group, and also (ii) add or remove entire groups of rules. Then, again using holdout validation, run the evolutionary algorithm, and select the system with the best validation performance. You should experiment with different parameter settings (for the algorithm), making several runs before selecting the very best system. Once you have made that selection, evaluate the best system over the test set, computing the total profit as follows: Whenever the system outputs a 1, add $(p\_(t+1)/p\_t) - 1$ to the total profit (do nothing when it outputs a 0), and then normalize the total profit as in Step 4.

6. Compare the LSTM network and the evolvable rule-based system, in terms of performance and interpretability. Discuss your findings.

What to hand in

You should hand in a Jupyter notebook that contains all the required code for running all the experiments above (including any necessary installations with "pip install"), Steps 1-5 above and the analysis and discussion (Step 6) (in a text cell in the notebook). Also (note!) hand in your data sets (training, validation, and test).

Note: You must make sure that your notebook can run (in its entirety) in Google Colab, without any tweaking or adjustment, and without using any other tools than Colab.

Grading

In order for the problem to be graded (at all), you must complete all six steps and hand in the Jupyter notebook as described above.

Since the problem is voluntary, it will be graded *once* – no resubmissions allowed.