Note on use of AI

Although the assignment states the use of AI tools is “in general not allowed”, I think in this day and age such restrictions are impractical. I’m going to explain what I used, how I used it and why this use was legitimate.

All the chats used are linked in the **Resources** section. The only exception is the Gemini chat integrated in the Colab Notebook which, as far as I know, can’t be saved nor shared (it appears to be nuked at each new session).

I have used

- ChatGPT to ask specific questions on Python usage;
- Gemini (browser) to ask about attribution techniques and LSTM implementation;
- Gemini (colab) to explain and solve small compilation errors (e.g. appropriately reshaping arrays before passing to functions).

I did not use AI to write the report (besides maybe auto-completion).

The way I used these tools was for the most part as glorified search engines. Indeed this allowed me to use the time otherwise spent on scavenging stackexchange, to refine the analysis and do more testing. Since the focus of the course is not on programming or algorithmic details, I think this use is justified (and for the same reason the use of the aeon package, see later).

The Gemini chat is a bit different, but I think it resides within the limits of fairness nonetheless. I actually read the Deepmind article [2b] last year, from which I learned about the use of gradient saliency for neural networks, and more in general about attribution techniques. I tried googling attribution techniques but I only found non-pertinent results: as usual when dealing with multidisciplinary subjects, different

communities use different words to express the same concepts. Since it would be too difficult to craft a query to get what I wanted, I decided to pass the article to Gemini so that it could have enough context to pin down the exact concept I was looking for (the choice of Gemini is for its longer context window). I follow with some questions to make sure it did not hallucinate, by checking the consistency of the answers / presence of contradictions.

I also asked how to implement an LSTM with attention mechanism (though a generic example rather than for the project task). Unlike the other method proposed, I don't think I could have done it myself with my limited time and experience in Python (that is a couple of weeks). For this reason, you may exclude it from the valuation if you wish (on the other hand, there was no need to implement multiple methods). The reason I chose to do it anyway is to see how well could perform a model with an inductive bias for the data at hand, that is time series. I very much agree with the point in this tweet about AI tools giving the possibility to aim for more ambitious projects:

In other words, coding assistants are complements (especially for senior developers), not merely substitutes. They enable the creation of significant value that wouldn't have been possible otherwise, as many projects would have been too difficult, costly, or time-consuming to undertake without them.

Notebooks

The exploratory analysis is here. Although it is not as well documented as the final one, it could be helpful to understand some decisions made (for example analysis of time executions).

The final notebook is here: basically it's a cleaned up, slightly more polished version.

Dataset

When I started the project (end of July) the website from which we were supposed to download the data was blank. I tried to search for mirrors or other repositories, but all references were to the same site. Fortunately, ChatGPT suggested to use the aeon toolkit, a Python package for machine learning on time series. Consequently, I decided to use this package not only for data loading but also for the rest of the analysis.

The dataset contains very few samples (~250 total) which makes it difficult to train complex models. Complex models are more flexible (low bias) but they need a lot of data to balance their higher variance.

The samples are ordered by label, so it would be best to shuffle the training set before fitting a model sensitive to order. There is also an imbalance in the labels (the first classes are more prevalent), so we may need to weigh them differently (I haven't actually implemented this).

The time series are very long, that is they comprise a lot of timesteps (order ~10^5). We may need to downsample in order to run the

methods in a reasonable amount of time. The easiest way to do it is to divide in segments, keep one timestep (for example the first) and skip all the others. If we choose a step small enough, we can retain a faithful shape of the original while losing dense clusters of similar values or very short and localized spikes. Getting rid of these high frequencies may be even desirable, since they could be artifacts caused by noise in the experimental setup for the data collection.

The data is also multivariate (i.e. more than one feature/channel per sample), but many methods can't cope directly with it, so we may need to turn into univariate. Possible strategies are:

- apply the univariate method one channel at a time and then aggregate the results
- concatenate in a mega time series; the concatenation can be done on the channels (one after the other), or on the vectors of channels' values
- apply a dimensionality reduction technique (PCA for example).

The first option is not really leveraging the possible interactions between the features.

For the third option it may be difficult to do the attribution, since we can't clearly separate the channels anymore.

The time series span very different ranges of values both across channels and samples. For methods that are sensitive to scaling (like neural networks), it would be best to normalize the data. It is not clear to me what is the best way to do it for multivariate data: normalize channel-wise or across all channels, like here.

Methods

Our goal is to develop one simple model as a baseline and then one complex model with inductive bias for the data, namely a RNN.

We refer to the classification here.

The simplest kind of model for time series we can use is a distance based method. We compute pairwise (dis)similarities between samples and feed them to a simple classifier, like KNN or a decision tree. We could use Euclidean distance, but this is sensitive to shifts: two time series with similar shape could result very far apart if they are shifted so that they don't match. So for time series we prefer to use so called elastic distances, of which the most common is DTW.
For more details see here.

Due to the aforementioned length of the time series, it is imperative to constrain the computation of the warping path with a bounding window. Downsampling is essential (otherwise it just takes too long) and it also has the nice effect of keeping the distance values low.

There are many other methods we could use, but for one reason or another I ended up discarding.

Feature based methods extract descriptive statistics from the time series (like mean, std, slope, spectral etc.) and feed them to a classifier. The most complex part is of course establishing which features to extract. This can be done essentially in three ways:

- manually, needs domain knowledge to understand which characteristics are more relevant for the task
- automatic, needs a potent model (see later)
- ready made collections, not ideal for this kind of project

Interval based methods select some intervals of the time series from which extract features to use in a classifier (like a decision tree / random forest). Same problem as above, plus now we need to devise a way to select the intervals.

More complex methods like shapelet, dictionary, convolution or deep learning based methods require a deeper understanding of the technique and a lot of design (for example engineer the kernels or the architectures). This is not only a problem for the training/testing but also for the attribution.

The RNN of course suffers from all the problems of the deep learning models (and even more than usual), in particular the selection of hyperparameters. For this reason we use the simplest architecture possible: a single LSTM layer. LSTM is a standard improvement on bare RNN trying to mitigate the vanishing gradient problem, necessary for such long sequences. We also take it to be bidirectional to get most information possible from the sequences.

We try both univariate and multivariate versions. Unfortunately the univariate version doesn't perform very well, despite all the hyperparameters fiddling: I tried feeding just one channel instead of the mega time series; remove data normalization / shuffling; changing downsampling parameter; changing number of neurons (both LSTM and attention layer), number of epochs, batch size; stack another LSTM layer; add dropout; change optimizer, learning parameter; add gradient clipping. Unfortunately I'm not proficient enough in Python to do this model selection in a more structured way, but from my trial and error resulted no significant differences.

The multivariate version seems to perform quite well, but it is not clear how much robust it is since I have not tested it nearly enough.

Attribution

In [2b] it is showcased an example of gradient saliency applied to neural network interpretability. The idea is that if by changing a little one feature the output changes a lot, then that feature is important to the classification. Since neural networks are just differentiable functions, we can get a concrete estimate of this concept by computing the gradients.

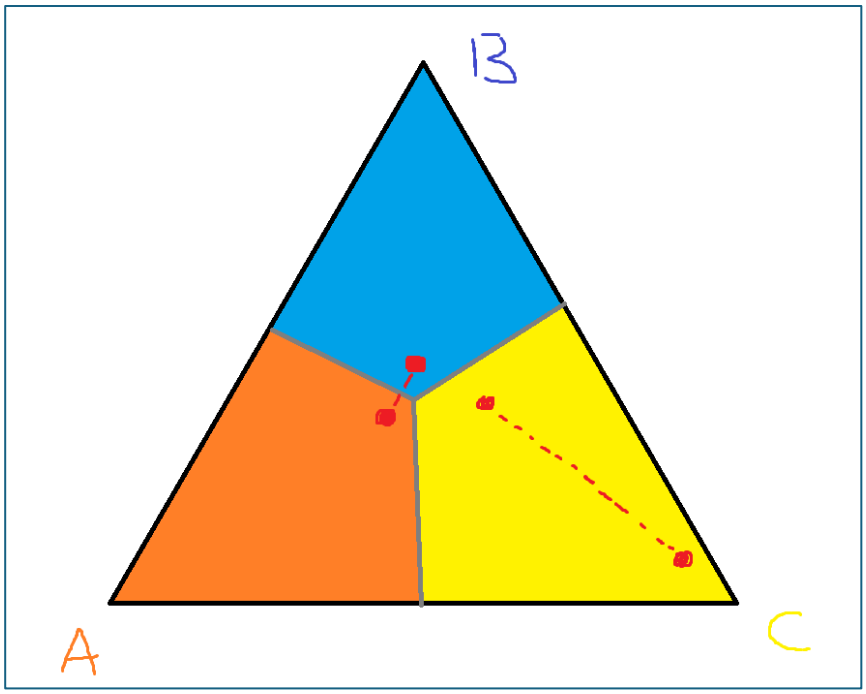
Now for a RNN is actually more attractive the idea of adding an attention mechanism to learn which parts of the sequence are taken more in consideration by the model, and thus are more important.

I implemented it for the univariate version and it seems to work. By looking at which part of the mega time series are more “light up”, we can also infer the channels that contribute more to the prediction.

For the multivariate version, probably we need to adapt the attention layer in order to distribute the attention along the feature vectors in a way that we can still plot it afterwards (I did not attempt to do this).

For the distance + KNN model we can’t use gradients nor attention, but we need a model agnostic method. The easiest is called feature occlusion: we feed the model the same sample with one of the feature masked out (for example by putting to 0 or the mean) and see how the prediction changes; as before the idea is that bigger changes mean the feature is important.

Since we are dealing with a classification task, it is actually better to compare the probability vectors instead of the class predicted. Indeed we can picture such probabilities as points of a simplex with as many vertices as the number of classes.



If the probability vectors are very close to the separation regions (the grey segments in the picture), small perturbations can easily change the predicted class (think going from 49% to 51% majority).
On the other hand, two vectors could be very far apart even if the predicted class is the same (think going from 51% to 99%).
In other words, the predicted class is not a reliable indicator for sensitivity analysis.

In the end we mask one channel at a time, compute the Euclidean distance between the probability vectors and the greater the distance, the more important the feature is.

Notice that both attribution methods provide quantitative estimates of (absolute) feature importance.

Resources used

[1] books

a. G. James, D. Witten, T. Hastie, R. Tibshirani, J. Taylor. *An Introduction to Statistical Learning with Applications in Python*. https://www.statlearning.com/

[2] articles

a. A.E.X. Brown, E.I. Yemini, L.J. Grundy, T. Jucikas, & W.R. Schafer, A dictionary of behavioral motifs reveals clusters of genes affecting *Caenorhabditis elegans* locomotion, *Proc. Natl. Acad. Sci. U.S.A.* 110 (2) 791-796, https://doi.org/10.1073/pnas.1211447110 (2013).

b. Davies, A., Veličković, P., Buesing, L. et al. *Advancing mathematics by guiding human intuition with AI*. Nature 600, 70–74 (2021). https://doi.org/10.1038/s41586-021-04086-x

[3] AI chat

a. ChatGPT

b. Gemini

[4] libraries

a. https://www.aeon-toolkit.org/

b. https://numpy.org/doc/stable/

c. https://scikit-learn.org/stable/user_guide.html

d. https://www.tensorflow.org/api_docs/python/tf/keras/

[5] tutorials

a. https://www.aeon-toolkit.org/en/stable/examples.html

b. https://www.tensorflow.org/tutorials/structured_data/time_series

c. https://www.tensorflow.org/guide/keras/working_with_rnns

[6] answers

a. https://stackoverflow.com/

b. https://stats.stackexchange.com/

c. https://stats.stackexchange.com/questions/352036/what-should-i-do-when-my-neural-network-doesnt-learn