# COMP4650/6490 Document Analysis Assignment 2 – ML

This assignment has 3 questions you should attempt all questions. This is an individual assignment, so you should complete it by yourself.

In this assignment you will:

1. Develop a better understanding of how machine learning models are trained in practice, including partitioning of datasets and evaluation.
2. Become familiar with the sklearn package for machine learning with text
3. Become familiar with the Pytorch framework for implementing neural network-based machine learning models.

Throughout this assignment you will make changes to provided code to improve or complete existing models. In some cases, you will write your own code from scratch after reviewing an example. In addition, you will produce an answers file with your responses to each question. Your answers file must be a .pdf file named u1234567.pdf where u1234567 is your Uni ID. You should submit a .zip file containing all of the code files and your answers pdf file, **BUT NO DATA.**

Your answers to coding questions (or coding parts of each question) will be marked based on the quality of your code (is it efficient, is it readable, is it extendable, is it correct).

Your answers to discussion questions (or discussion parts of each question) will be marked based on how convincing your explanations are (are they sufficiently detailed, are they well-reasoned, are they backed by appropriate evidence, are they clear, do they use appropriate visual aids such as tables, charts, or diagrams).

## Question 1: Movie Review Sentiment Classification with Dense Word Vectors (25%)

For this question you have been provided with a movie review dataset. The dataset consists of 50,000 review articles written movies on IMBD, each labelled with the sentiment of the review – either positive or negative. Your task is to apply logistic regression with dense word vectors to the movie review dataset to predict the sentiment label from the review text.

A simple approach to building a sentiment classifier is to use train a logistic regression model that uses aggregated pre-trained word embeddings. While this approach, with simple aggregation, normally works best with short sequences you will try it out on the movie reviews.

You have been provided with a file *dense_linear_classifier.py* which reads in the dataset and splits it into training, testing, and validation sets; and then loads the pre-trained word embeddings. These embeddings were extracted from the *spacy 2.3.5* python package “en_core_web_md” model and, to save disk space, were filtered to only include words that occur in the movie reviews. Your task is to use a logistic regression classifier with aggregated word embedding features to determine the sentiment labels of documents from their text. First implement the *document_to_vector* function which converts a document into a vector by first tokenizing it (the TreebankWordTokenizer in the nltk package would be an excellent choice) and then aggregating the word embeddings of those words that exist in the dense word embedding dictionary. You will have to work out how to handle words that are missing from the dictionary. For aggregation, the *mean* is recommended but you could also try other functions such as *max.* Next, implement the *fit_model* and *test_model* functions using your *document_to_vector* function and LogisticRegression from the sklearn package. Using *fit_model, test_model,* and your training and validation sets you should then try several values for the regularization parameter C and select the best based on accuracy. To try regularization parameters, you should use an automatic hyperparameter search method. Next, re-train your classifier using the training set concatenated with the validation set and your best C value. Evaluate the performance of your model on the test set.

Answer the following questions in your **answer pdf**:

1. What range of values for C did you try? Explain, why this range is reasonable. Also explain what search technique you used and why it is appropriate here.
2. What was the best performing C value?
3. What was your final accuracy?

**Also make sure you submit your code.**

Hint: If you do the practical exercise in Lab 2 this question will be much easier.

Tip: If you use TreebankWordTokenizer then for efficiency you should instantiate the class as a global variable. The TreebankWordTokenizer compiles many regular expressions when it is initialized; doing this every time you want to tokenize a sentence is very inefficient.

Documentation for TreebankWordTokenizer
https://www.nltk.org/_modules/nltk/tokenize/treebank.html

## Question 2: Kaggle Task - Genre Classification (50% total: 30% competition, 20% writeup)

For this task you will design and implement a classification algorithm that identifies the genre of a piece of text. This task will be run as a competition on Kaggle. Your marks for this question will be partially based on your results in this competition, but your mark will not be affected by other students' scores, instead you will be graded against several solutions developed by the convener and tutor team. The other part of your mark will come from your code and write-up.

The dataset consists of text sequences from English language books in the genres: horror (class id 0), science fiction (class id 1), humor (class id 2), and crime fiction (class id 3). Each text sequence is 10 contiguous sentences. Your task is to build the best classifier when evaluated with **macro averaged F1 score**.

Note: the training data and the test data come from different sets of books. You have been provided with bookids (examples with the same bookid come from the same book) for the training data but not the test data.

You have been provided with an example solution in *genre_classifier_0pc.py* this shows you how to read in the training data (*genre_train.json)*, test data (*genre_test.json*) and output a csv file that the judging system can read. This solution is provided only as an example (it is the 0% benchmark for this problem), you will want to build your own solution from scratch.

### *Setup:*

Please register for Kaggle **using your university email address** (https://www.kaggle.com/).

Submit solutions to: https://www.kaggle.com/t/042af1cd70dc4292b6d71d618cb5ef03
**Set your team name to your uid (e.g. u1234567).** This will allow us to match your submission to your assignment for marking purposes. ("team name" is a kaggle term, this is an individual assignment, and so you should not work with others to complete it)

If you are unable or unwilling to use Kaggle, you may submit a csv file along with your assignment. It should be named with your uid e.g. u1234567.csv . If you use Kaggle you should not submit this file with your assignment.

### *Rules:*

- Do not use additional supervised training data. That is, you are not allowed to collect a new genre classification dataset to use for training. Pre-training on other tasks is permitted.
- Do not use models pre-trained on genre classification. Models pre-trained on other tasks are permitted (eg you may use word vectors from *spacy* or *gensim*, or pre-trained transformers from *transformers*).
- You can use the following libraries (in addition to python standard libraries): *numpy*, *scipy, torch, transformers, tensorflow, nltk, sklearn, xgboost, pandas, gensim, spacy*. If you would like to use other libraries, please ask on the piazza forum.
- This is an individual task, do not collude with other individuals. **Copying code from other people models or models available on the internet is not permitted.**

### *Juding system:*

- You will upload your predictions for the test set as a csv file for judging.
- You are allowed to **submit up to 5 times per UTC Day.** Since you get immediate feedback from every submission it is best to start submitting early and plan ahead.
- The results shown on the public scoreboard prior to the conclusion of the contest only include 50% of the test data. Your solution will be judged on the **other** 50% of the test data when computing final rankings and marks.
- The judging system allows you to choose which of all your submissions you want to be your final one.

*Competition Marking:*

Marks will be assigned based on which judge baselines you beat on the hidden 50% of the test data. If, for example, you beat the 80% baseline but do not beat the 90% baseline you will be awarded a mark on a linear scale between these two based on your macro averaged f1 score. Exceeding the score of the 100% baseline will give you a mark of 100% for the **competition component** of this question. It will also award bragging rights. (The 100% baseline and all other baselines can be trained in less than 24 hours on a laptop pc without GPU support -- your solution may make use of any compute resources available to you).

*Write up:*

A fraction of your marks will be based on a write up that a minimum describes:

- How your final solution works
- How you trained and tested your model (e.g. validation split(s), hyperparameter search ...)
- What models you tested, which worked, and which didn't
- Why you think these other models didn't work

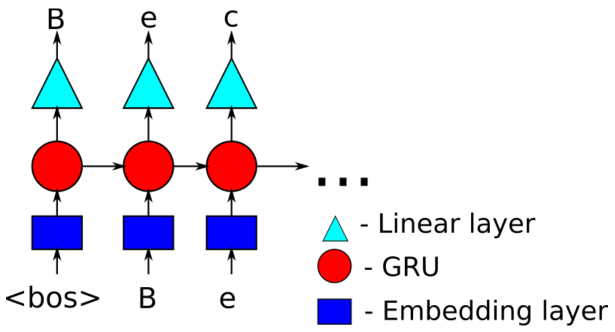
Aim for 1 page or slightly less. Only the first 2 pages will be marked. Bullet points are acceptable if they are understandable.

*What to submit:*

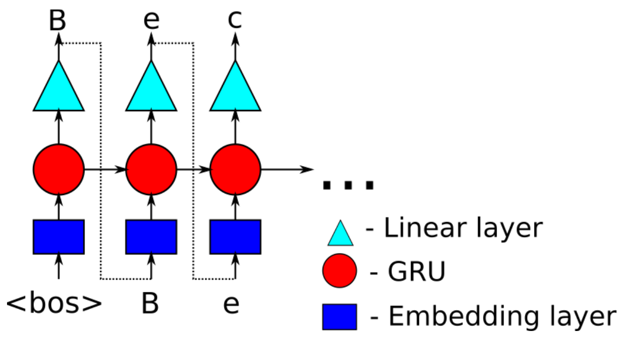
- The code of your best solution, including training pipeline. **Also make sure your team name is set to your uid on kaggle. Do not submit stored parameters or data.**
- Your write-up in your answer pdf file
- If you could not use Kaggle (**and only if you could not use Kaggle**) a <your_uid>.csv file with your models output in the correct judging format.

## Question 3: RNN Name Generator (25%)

Your task is to develop an autoregressive RNN model which can generate people's names. The RNN will generate each character of a person's name given all previous characters. Your model should look like the following:



Note that the input is shown here as a sequence of characters but in practice the input will be a sequence of character ids. There is also a softmax non-linearity after the linear layer but this is not shown in the diagram. The output (after the softmax) is a categorical probability distribution over the vocabulary, what is shown as the output here is the ground truth label. Notice that the input to the model is just the expected output shifted to the right one step with the <bos> (beginning of sentence token) prepended. The three dots to the right of the diagram indicate that the RNN is to be rolled out to some maximum length. When generating sequences, rather than training, the model should look like the following:



Specifically, we choose a character from the probability distribution output by the network and feed it as input to the next step. Choosing a character can be done by sampling from the probability distribution or by choosing the most likely character (otherwise known as argmax decoding).

The character vocabulary consists of the following:

“” : The null token padding string
<bos> : The beginning of sequence token
. : The end of sequence token
a-z : All lowercase characters
A-Z : All uppercase characters
0-9 : All digits
“ “ : The space character

Starter code is provided in *rnn_name_generator.py,* and the list of names to use as training and validation sets are provided in *names_small.json.* To complete this question you will need to complete three functions and one class method: the function *seqs_to_ids*, the *forward* method of *RNNLM,* the function *train_model*, and the function *gen_string.* In each case you should read the description provided in the starter code.

*seqs_to_ids* - Takes as input a list of names. Returns a 2d numpy matrix containing the names represented using token ids. All output rows (each row corresponds to a name) should have the same length of max_length. Achieved by either truncating the name or padding it with zeros. For example, an input of:

[“Bec.”, “Hannah.”, “Siqi.”] with a max_length set to 6 should return (normally we will use max_len = 20 but for this example we use 6)

```
[[30  7  5  2  0  0]
[36  3 16 16  3 10]
[47 11 19 11  2  0]]
```
Where the first row represents “Bec.” and two padding characters, the second row represents “Hannah”, the third row represents “Siqi.” with one padding character.

*forward* – A method of RNNLM. In this function you need to implement the GRU model shown in the diagram above. The layers have all been provided for you in the class initializer.

*train_model* – In this method you need to train the model by batch gradient decent. The optimizer and loss function are provided to you. Note that the loss function takes logits (out put of the linear layer before softmax is applied) as input. At the end of every epoch you should print the validation loss using the provided *calc_val_loss* function.

*gen_string* – In this method you will need to generate a new name, one character at a time. You will also need to implement both sampling and argmax decoding.

For this question, please include in your **answers pdf** the most likely name your code generates using argmax decoding as well as 10 different names generated using sampling. Also remember to **submit your code**. Your code should all be in the original *rnn_name_generator.py* file, other files will not be marked. For this question **you should not import additional libraries,** use only those provided in the starter code (you may uncomment the import tqdm statement if you want to use it as a progress bar).

For example, one of the names sampled from the model solution was: Dasbie Miohmazie

# COMP4650/6490 Document Analysis Assignment 3 – NLP

For this assignment, you will implement several common NLP algorithms: probabilistic context free grammar learning, dependency parsing, and relation extraction in Python.

Throughout this assignment you will make changes to provided code to improve or complete existing models. In addition, you will produce an answers file with your responses to each question. Your answers file must be a .pdf file named u1234567.pdf where u1234567 is your Uni ID. Some questions also require you to submit an output file in a format specified by the question. You should submit a .zip file containing **all the code files**, your **answers pdf** file, and these **requested output files** but **BUT NO OTHER DATA.**

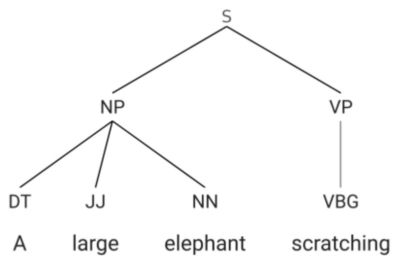
Your answers to coding questions (or coding parts of each question) will be marked based on the quality of your code (is it efficient, is it readable, is it extendable, is it correct). The output files requested will be used as part of the marking for the coding questions.

Your answers to discussion questions (or discussion parts of each question) will be marked based on how convincing your explanations are (are they sufficiently detailed, are they well-reasoned, are they backed by appropriate evidence, are they clear).

This is an individual assignment. Group work is not permitted. Assignments will be checked for similarities.

## Question 1: Probabilistic Context Free Grammars (20%)

For this question you will be modifying the starter code in *syntax_parser.py.* You have been provided with a training dataset containing 73638 English language sentences*,* which have been parsed into constituency trees *(parsed_sents_list.json)*. For, example given the sentence: *"A large elephant scratching".* The parse tree is:



For this structure the parsed sentence in *parsed_sents_list.json* is *['S', ['NP', ['DT', 'a'], ['JJ', 'large'], ['NN', 'elephant']], ['VP', ['VBG', 'scratching']]]*. Note that *parsed_sents_list.json* contains a list of parses, each for a different sentence.

Your task is to estimate a probabilistic context free grammar PCFG given this dataset. You should estimate the conditional probabilities of the rules in this grammar (including all rules that express facts about the lexicon) using the maximum-likelihood approach given in lectures. Some examples of rules in this grammar are:

*S -> NP*
*VBN -> parked*
*ADVP -> IN NP*
*INTJ -> , NP VP .*

Note that some terminals and non-terminals are punctuation characters. There are many more rules than those listed here and you will first have to identify all the rules. All terminals are lower case or punctuation.

When your *syntax_parser.py* code is run, write a list of all the transition rules and their **conditional probabilities** to a file called "q1.json". There is a class called *RuleWriter* in *syntax_parser.py* which will write the rules in the correct format for the automated judging system (do not modify this class). The expected format is (example of format only not actual answers):

[["NP", ["PP", "NP"], 0.8731919455731331], ["NNS", ["locations"], 0.062349802638746935], ….
<many more rules>]

Make sure to submit "q1.json" for marking. Also make sure to **submit your code.**

Hint: if you create a test input file containing only [["A", ["B", ["C", "blue"]], ["B", "cat"]]] your program should output (note: the order the rules are listed in the output does not matter):
[["A", ["B", "B"], 1.0], ["B", ["C"], 0.5], ["B", ["cat"], 0.5], ["C", ["blue"], 1.0]

## Question 2: Dependency Parsing (40%)

Complete both part A and part B of this question.

### *Part A*

The file *dependency_parser.py* contains a partial implementation of Nivre's arc-eager dependency parsing algorithm. Your first task is to implement the missing *left_arc* method, *reduce* method*,* *right_arc* method, and *parse* function. Your implementation should follow the transitions as given by the lecture slides (also some additional instructions are provided as comments in *dependency_parser.py)*. Make sure that your class methods and parse function return True if the operation(s) was/were successfully applied, or else False. The *parse* function should return True if it generates a dependency forest. Your implementation **must** make use of the *DepContext* class which stores the current state of the parser. **Do not modify the *DepContext* class or the name and parameters of the methods and functions you have been asked to implement** because part A of this question will be automatically marked. The auto-marker will inspect the internal state of DepContext to confirm that operations have been applied correctly. You should not use any additional imports for this question (except from python standard libraries). The judging system will use python 3.8.10 in a sandboxed Linux environment. In your **answers pdf** file include the output of *dependency_parser.py* (it runs a small number of test cases and prints them to the terminal). Make sure to submit your **code**.

Hint: As mentioned in lectures we need to make sure that words are uniquely represented (in case there are multiple copies of the same word), to do this we use the position of the word in the sequence, with the [ROOT] token prepended to the sequence. This is handled by DepContext.

### *Part B*

Your second task is to implement the standard oracle for arc-eager dependency parsing in *oracle_parser.py* (specifically you should implement the *parse_gold* function). Make use of the parsing class you developed in Part A of this question -- do not modify your parsing class from part A, all code for this second part should be put in the *oracle_parser.py*. The parsing class from part A has been imported. Additional imports are not permitted (except from python standard libraries). You may add helper functions in *oracle_parser.py.* The oracle you need to implement determines which action to be performed in each state to reconstruct the ground truth dependency parse tree. It returns the list of actions, and the states in which they occur – though in this case you should return only features of the state (using the provided extract_features function -- do not modify the feature extraction code). The details of this algorithm were not covered in lectures, instead you should implement **Algorithm 1** from this paper: https://aclanthology.org/C12-1059.pdf (while this paper covers different oracle methods including a dynamic oracle, you should only implement **Algorithm 1** on page 963). You should implement un-labelled dependency parsing so you will need to modify this algorithm to work without labels. To implement Algorithm 1 you will need to make use of the parsing class you developed in Part A of this question.

Once you have implemented the *parse_gold* function, run *oracle_parser.py* and it will determine the actions that could generate the ground truth dependency parse trees in *gold_parse.json* under the rule set in *rule_set.json*. It will print the first 3 cases where the oracle could reproduce the ground truth, the first 3 cases where it could not reproduce the ground truth, and after a short wait the number of total failure and total success cases. In your **answers pdf** file include the output of *oracle_parser.py* and **explain why** it failed in the three printed cases.

Make sure to submit all your **code** for both parts of this question.

## Question 3: Relation Extraction (40%)

Your task is to design and implement a relation extraction model along with its training and testing pipeline. Specifically, your model should extract the relation 'nationality' (denoted /people/person/nationality in our dataset) which holds between a PERSON and a GEOPOLITICAL ENTITY. For example /people/person/nationality(Yann LeCun, France) , /people/person/nationality(Julie Bishop, Australia). The training data is provided in (*sents_parsed_train.json*) the testing data is provided in (*sents_parsed_test.json*). The test data does not have the ground truth set of relations; you program must extract them. Some starter code has been provided (*relextract.py*) which shows you how to read in the data, access its various fields, and write a valid output file.

The final goal is to **extract all relations** of the type *nationality* that exist in the **test data**, not to simply identify if a relation type is in each sentence. Your output should be a list of all the relations in the test data, in CSV format (you should name it q3.csv) -- the starter code provides a function to write this file. The order of lines in this csv does not matter, and duplicate lines will be removed before scoring. You will be scored based of F1. An example judging script (eval_output.py) is provided for you. Since you have not been given the ground truth for the test data you can only run this script using the training data. To do this run (python eval_output.py --train-set). Note that your final output will be evaluated against the test set by the examiner. Thus, make sure that you submit your **output file** named **q3.csv and make sure it contains the output from the test data**.

It is very important that your relation extraction algorithm work for entities **not seen during training**. To clarify, this means that your model should be designed so that it can identify relations such as /people/person/nationality(Julie Bishop, Australia) even if "Julie Bishop" and "Australia" entities are not in the training data. None of the PERSON, GPE pairs that constitute a nationality relation occur in both the training dataset and the test dataset.

The focus of this question is to develop a good training/validation pipeline and to explore different lexical, syntactic or semantic features. You should use the 'Feature-based supervised relation classifiers' technique introduced in the Speech and Language Processing textbook (3rd edition draft 30th December 2020) Section 17.2.2 -- and figure 17.6. You do not have to implement all the suggested features, but an attempt to implement a variety of them is expected. You will likely need to do substantial pre-processing of the input data as well as some post processing of the output list to get a good result. You should use LogisticRegression from sklearn as the classifier. **Neural networks and vector features generated by neural networks are not to be used for this question.** You may use SpaCy to extract additional features from your text; however, several features are already provided for you. It is possible to get full marks using only the features provided; however, these features will likely need some processing before they can be used appropriately as part of a classification pipeline.

**Permitted Libraries:** sklearn, numpy, pandas, python standard libraries, nltk, spacy, json.
You should **not** use other libraries, datasets, or ontological resources. Except for the provided country_names.txt.

Answer the following questions in your **answers pdf file** (please answer in plain English, dot points are preferred, do not include code snippets or screenshots of code as these will not be marked):

1. What pre-processing did you do?

2. How did you split the data into training and validation sets?
3. How did you choose hyperparameters?
4. What features did you choose and why did you choose these features?
5. What post-processing did you do?

Submit your **code**. Submit your **output file** named **q3.csv** (**make sure this is the output from the test data**). Submit **your answers** to the questions.

## Marking:

Part of the mark for this question will be based on the f1 score of your q3.csv file on the held-out test data. The rest will be based on an examination of your code and answers to the questions in your answers pdf file.

## Description of the data:

The data you have been given is described in detail below. Briefly, it is a set of sentences, each of which contain a relation between two entities. You may use the relations that are not the target relation “/people/person/nationality” to generate negative examples on which to train your model.

*sents_parsed_train.json* and *sents_parsed_test.json* each contain a list of dictionaries. Each dictionary corresponds to a different example.

**Each example dictionary has the following fields:**

“tokens”: a tokenised sentence that contains a relation

“entities”: a list of the named entities in the sentence

“relation”: a description of the relation in the sentence

**Each *entity* is a dictionary with the following fields:**

“start”: the token index where the entity starts

“end”: the token index where the entity ends

“label”: the entity type. PERSON = a persons, GPE = a geopolitical entity, and several others

**Each *relation* is a dictionary with the following fields:**
Note: the relation field has been removed from *sents_parsed_test.json*.

“a”: first entity that is part of the relationship (tokenized into a list)

“b”: the second entity that is part of the relationship (tokenized into a list)

“a_start”: the token index where the first entity starts

“b_start”: the token index where the second entity starts

“relation”: the type of relation, note we are interested in the “/people/person/nationality” relation for this assignment.

There are also some **additional features** that may or may not be helpful (each of the features below is a list, the array indices match to those of the tokenized sentence):

“lemma”: the tokenized text that has been lemmatized.

“pos”: part of speech tags for each word

“dep”: the dependency relationships

“dep_head”: the head of each dependency relationship (allowing you to reconstruct the full dependency tree)

“shape”: shape features for each word. This preserves things such as word length and capitalization while removing much of the content of the word.

“isalpha”: is each word only alphabetic.

“isstop”: is each word in the spacy stop word dictionary (and thus likely to be common)

A file containing a list of country names (country_names.txt) is also provided for you to use.

All features were extracted using SpaCy 2.3.5 en_core_web_md models. The data you have been provided was constructed automatically by matching ground truth relations to text. For the training data any sentence that had both the named entities that participated in a ground truth relation was marked as containing that relation. No manual checks were done to ensure accuracy. You might like to consider how this will affect the data quality.