e2e-text: Automating Text Classification Pipeline Design Using Genetic Algorithms

Final Project for University of London BSc In Computer Science - By: Pablo Nicolas Marino
Github Repo: https://github.com/pablonm3/UoL_final

Template: CM3020 - Artificial Intelligence - Project idea 2 Automated design using evolutionary computation

Table of Contents

Introduction

Machine learning is one of the fastest growing fields in computer science. Today most high tech organizations rely on the development of high accuracy machine learning pipelines for the development of systems like chatbots, information retrieval, autonomous driving, hate speech detection, voice assistants, etc. The development of a machine learning pipeline involves the selection of a good combination of techniques, parameters and even neural architectures that in combination work well for modeling a specific problem. The number of combinations for designing a machine learning pipeline is infinite, it's a process that relies heavily on empirical evaluations and a lot of the STOTA methods take hours or even days to test only one possible solution.
Most AI developers don't have the resources or the time to test a lot of possible solutions and often end up using the designs that others have developed for similar(but not equal) tasks and are already known to work. These constraints on experimentation are a problem in a field so complex, dynamic, recent and so dependent on experimentation(since we still don't fully understand how it works) as machine learning because they limit our creativity and the quality of the solutions we deploy. Finding a good combination of parameters for a specific task in an infinite space of possibilities where every test takes time is a hard problem and in response the field of Auto Machine Learning(AutoML) was born.

AutoML are systems that automatically design Machine learning pipelines or neural network architectures or both from automatically testing different combinations of parameters. There are two categories of autoML systems based on the search space where they operate:

- ML pipeline design: find combination of components and their hyperparameters for designing an entire pipeline, eg: feature selection, preprocessing, vectorization, predictive model, training hyperparameters.
- Neural architecture search (NAS): design the architecture of a neural network eg: number and types of layers, dropout, non linear activation functions, etc.

In the literature review I'm going to review 5 open source AutoML projects and their limitations and will expand on what the field of AutoML looks like today. But the conclusion I have arrived is that there is no open source AutoML project that automates pipeline design of text classification tasks and uses the STOTA technologies(Transformer models). As an NLP engineer I've had the need for such a system several times since text classification is the most common task in the field, Also recent autoML articles talk about the need for more research in the full automation of ML, ie: both in the pipeline design and also in the model design(NAS)(p 36, white, et al 2023)[1] since most current system do one or the other but few do both. Additionally genetic algorithms are widely used in autoML research because they are simple to implement but very effective search techniques (Real et al 2019)[2] .
So my goal for this project is the following:
Develop a framework that given a dataset for multiclass text classification finds a good combination of preprocessing techniques, vectorization, vector combination operation, model design(NAS) and hyperparameters using a genetic algorithm and testing STOTA technologies in the field of NLP like Transformer models.

Domain and Users
The domain of my system is the NLP branch of artificial intelligence, and particularly the task of text classification using deep learning. The users of my systems would be AI, ML or NLP engineers who want to design a good pipeline for text classification problems using recent technologies but don't have the time to manually test several combinations and would benefit from a tool that automates such a process.

<u>Literature review</u>

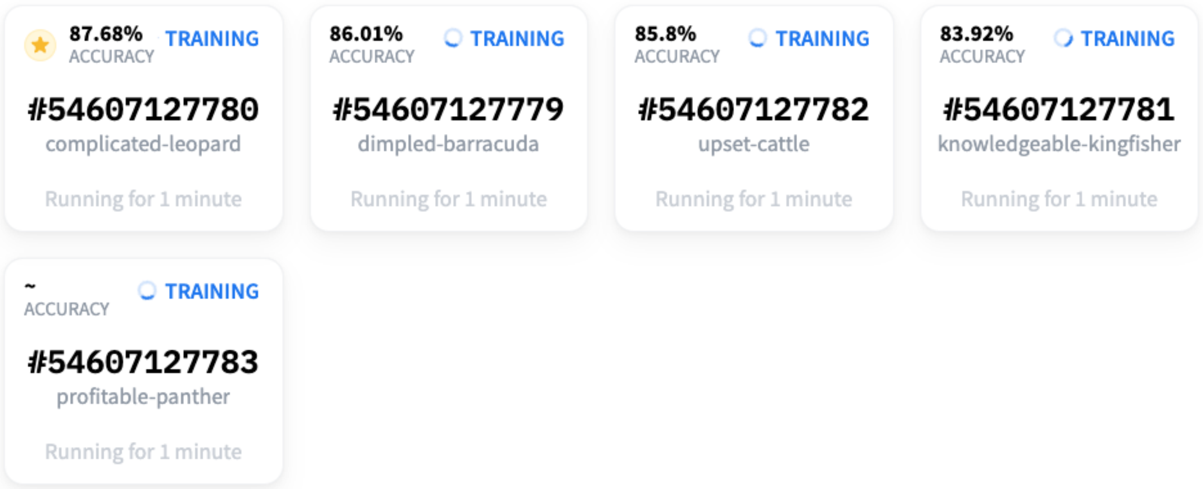
Neural Architecture Search: Insights from 1000 Papers[3]
This is the most relevant article I've read about the space of AutoML because it's very recent, it was published in January 2023 and written by 8 top notch AI researchers from different research labs including Microsoft Research. It's the most complete recent survey about the field of neural architecture search(NAS) because they analyze and contextualize the 1000 most recent research publications in the field in a magnificent way.
According to this article the field of neural architecture search(NAS) is rapidly growing in the AI communities, between the years 2020 and 2021 1000 research articles were published on the topic and they give an overview of the achievements and future directions of the field.
NAS search is the task of using an optimization algorithm(like a genetic algorithm) to find a good combination of parameters to design a neural network like: number of layers, dropout rate, activation function, batch size, etc. It was borned in the computer vision community where it was mostly focussed on image classification and after designing STOTA techniques it grew to other fields like protein folding, weather prediction and recently gained traction in the field of NLP.

From this article I took several leanings for my project:
- Most NAS systems use validation accuracy as the metric to optimize, this can be my fitness metric when I implement GA.
- Random search is a good baseline to comparing my optimization system against. It's commonly used in NAS field since interestingly it's not trivial to beat random search, from the article: "random search is highly recommended as a baseline comparison for new NAS algorithms (Lindauer and Hutter, 2020;Yang et al., 2020),".
- GAs are an efficient and still used technique for NAS due to their performance and simplicity of implementation. However SGD(stochastic gradient descent) must be used for training the weights of the network, GAs are not as efficient for optimizing NN weights.
- When implementing a GA for NAS techniques that are commonly used for selecting the initial population are: "using trivial architectures (Real et al., 2017), randomly sampling architectures from the search space (Real et al., 2019; Sun et al., 2019), or using hand-picked high-performing architectures (Fujino et al., 2017)."
- The article mentions many works that design new transformer architectures using NAS, some have successfully designed transformer language models using genetic algorithms which is an indication of the power and current relevance of the GAs in the field of machine learning.

- Another approach in NAS which is closer to pipeline design is the one of searching possible architectures and their hyperparameters without designing the actual NN topology, this means I could also choose to optimize the pipeline components but use common NN architectures in case I run into hardware restrictions for designing my own NN topology.
- Every experiment has a budget of time, if that budget expires the experiment terminates, this is a very important concept that I will have to implement in my project because training neural networks on transformer based vectors on my home computer can be time consuming and the GA requires to run on hundreds or thousands of experiments to work.
- There is a tradeoff between bias and efficiency of search: “if the size of the search space is small and includes many handpicked decisions, then NAS algorithms will have an easier time finding a high-performing architecture. On the other hand, if the search space is large with more primitive building blocks, a NAS algorithm will need to run longer, but there is the possibility of discovering truly novel architectures (Real et al., 2020).” What this means for my project is that in the beginning I will limit the amount of possible architectures to test the system and validate the GA works as expected and after will progressively add more parameters to the search space to increase the chances of designing a novel architecture.
- The future directions section of the article talks about the need for “Fully Automated Deep Learning” because recent research has shown that in order to design the best possible ML model, the NN architecture, hyperparameters, and preprocessing techniques must be evaluated together and not in isolation they state that future research should focus on automation of the entire deep learning pipeline. This is exactly what I’m doing in this project and it’s a great indication that there is a real need for my work in the AutoML community.

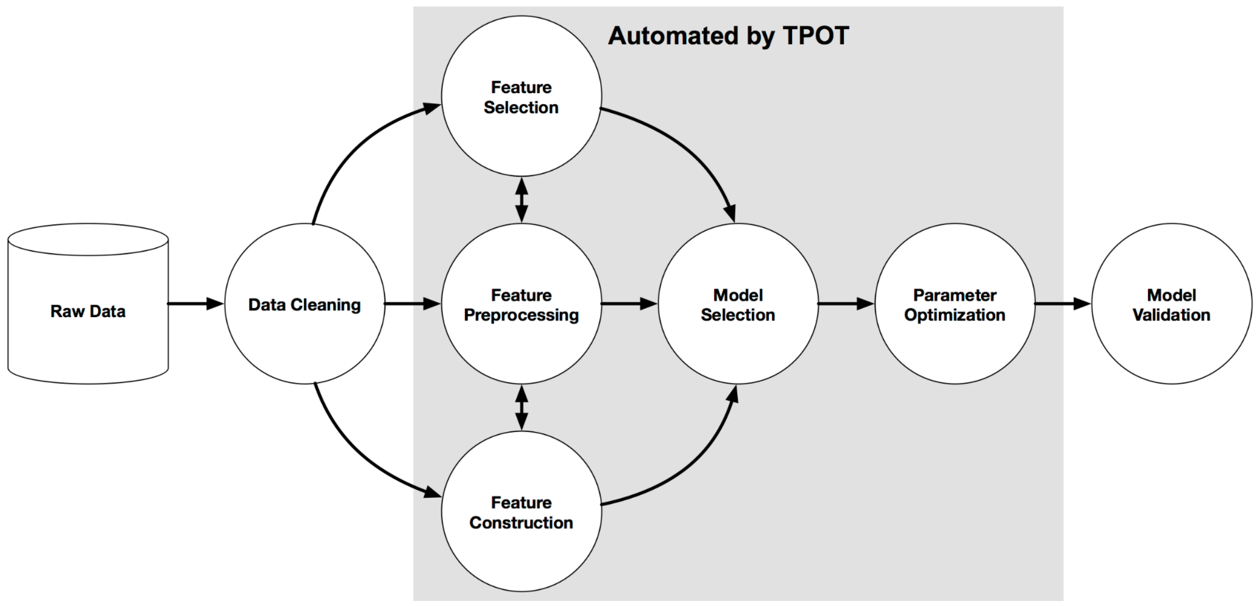
Hugging face Autotrain[4]
This is the AutoML system from HuggingFace, it’s a service where a user logs in a web dashboard, selects the task among many available like: between text classification, image classification, question answering,summarization, etc, uploads a dataset and the system starts training different models chosen automatically from Hugging face’s Model Hub and ranks them based on the performance. Out of all the systems I will compare in this work this is the easiest one to use, I was able to find a decent model for my dataset in a few clicks and for free(they give users some free compute and when exceeded you start paying) and very quickly because it evaluates models concurrently, also because it downloads models from their hub it ends up evaluating very recent models, some even state of the art.
This system still has many limitations the first one being that it’s closed source, also it only finds transformer models but doesn’t search all the other necessary components of a text classification pipeline like preprocessing techniques, design of the classification head and hyperparameter optimization. Most importantly a user doesn’t have control of the models evaluated, the system selects them automatically.

HF AutoTrain evaluating transformer models concurrently:



One takeaway from this system is the value of training and evaluating architectures concurrently, this greatly reduces the time for finding a quality solution. In the development phase I will consider concurrent training if my development machine is able to support it.

TPOT AutoML[5][6][7]
This is a very popular open source framework for AutoML, it uses a genetic algorithm to optimize an entire ML pipeline, it covers feature selection, feature preprocessing, feature construction, model selection and hyperparameter optimization. Out of the box it works with several shallow learning scikit-learn's components and also supports MLP(simple neural networks) implemented on Pytorch.
The system is able to build ensembles of models and according to their paper even though their base models are simple(shallow learning) when combined they can be able to map highly non linear problems similarly to the capabilities of deep neural networks, in their documentation they say: "TPOT will occasionally learn pipelines that stack several sklearn estimators. Mathematically, these can be nearly identical to some deep learning models. For example, by stacking several sklearn.linear_model.LogisticRegressions, you end up with a very close approximation of a Multilayer Perceptron; one of the simplest and most well known deep learning architectures.".
It supports customization so new models can be added but it contains a fundamental limitation which is that it only works with tabular data so can't be used for optimizing text classification.
Parts of the Machine learning pipeline optimized by TPOT:



The main takeaways for my project are:

- Their paper talks about GAs scale very well for optimizing complex pipelines with lots of different components and hyperparameters, this is useful to know for my project
- It utilizes the Python package DEAP(Fortin et al., 2012)[8] to implement the Genetic algorithm. I looked up this module and their github page claims they are utilized by several research optimization systems like this one, so I will consider it when implementing my solution.
- In their original paper they talk about a problem where the GA designs pipelines that overfit the testing data because the fitness metric was accuracy on testing set(rather than overfitting training data pipelines overfit testing set) and talk about ideas for modifying the fitness calculation to penalize models that are too complex. I will take this into account in case I see this issue happening in my project. Also the latest version of the TPOT by default uses average accuracy over a 5-fold cross validation(CV) for fitness score. They didn't mention why they chose that strategy but I think it could be to reduce the overfitting. Using CV in my project is probably not a good idea because of the long training time of the models it will use but I will take it into consideration.
- In their last paper they added a feature that allows the user to fill templates with the parameter space he wants to search, this reduces the search space greatly and thus the time to find a good pipeline. I will consider this idea in case my project takes too long to find a good pipeline.

AutoPytorch[9][10]

AutoPytorch is an open source system that performs optimization of the entire machine learning pipeline: Data preprocessing, NAS and hyperparameter optimization, it only works with tabular data and I find this system relevant because it's one of the most popular AutoML Systems for NAS and their authors compared the performance of their system against other Popular AutoML

systems like AutoNet, AutoKeras, autoSklearn, hyperopt and AutoGluon and found that AutoPytorch performs the best.
Auto-Pytorch is a good system to learn from because it's one of the few that perform both NAS(design neural network topology), and what I call ML pipeline design which besides the model selection(or design) also entails data preprocessing and hyperparameter optimization.
I find this article very interesting because it proposes that since the training hyperparameters are highly tied to the NN architecture a good auto machine learning framework should optimize them both together.
For performing the optimization they used several techniques, the most important ones being multi fidelity optimization: Training an architecture in a proxy to the real problem but simpler to accelerate training(eg: training on a few epochs) and portfolio building(Warm starting search with a baseline of good enough parameters). The key takeaways I learnt from this research that I could apply to my project are:

- Training hyperparameters and NN topology together is promising since hyperparameters depend on the later one.
- Authors claim that performance greatly benefits when warm starting optimization with a portfolio of good enough parameters. I will consider this when thinking about an strategy for an initial population in my GA
- The parameters space is structured by a conditional hierarchy such as top-level parameters can activate or deactivate sub-levels of parameters and they achieve this with a Python Module named ConfigSpace that looks very useful for representing configuration spaces for optimization algorithms.
- For design of neural network topology they propose a small configuration space that contains only a few design decisions to simplify the problem, using an architecture they call shapedMLP they reduce the NAS and hyperparameter search spaces to the optimization of only these parameters: No. of layers, max number of neurons(in first layer and following layers reduce this number proportionally), batch size, learning rate, L2 regularization, momentum, max dropout rate. This is very useful for the NAS and hyperparameter search parts of my project and looks simple yet powerful so I will use this idea on my project, I may even simplify it a little more by getting rid of some hyperparameters that the authors claim don't have a big impact in the performance like momentum, batch size, dropout rate and max number of neurons(they found number of layers has a big impact on performance but number of neurons alone doesn't, in the Author's words: "can be explained by the fact that more layers also leads to overall more neurons by using our shaped networks; however more neurons (max. units) do not imply more layers.")

AutoGoal[11][12][13]
AutoGoal is an open source AutoML framework designed to optimize any machine learning pipeline including data preprocessing, feature engineering and extraction, model selection or NAS(model design), hyperparameter optimization and any other ML component a user could want to optimize because besides the 133 algorithms that the system includes by default a user can add any component, algorithm or model and show the system what are the parameters that

this component requires to optimize. It supports any machine learning problem and any type of data: text classification, image classification, tabular data classification and regression, NER(Named Entity Recognition) and anything the user may want to design a Machine learning pipeline for.
We could say that this framework is algorithm agnostic and this is its main advantage. It uses an evolutionary programming technique based on Probabilistic grammatical evolution(PGE) to search the space of parameters. PGE is a more efficient version of the traditional genetic algorithm(GA) which uses a probabilistic model to optimize the evolutionary process of the GA(PGE has a slightly better performance than GE according to its authors on Jessica Mégane et al, 2021[11]).
What I learnt from this work is that my idea for this project is highly needed in the industry mainly because AutoGoal is very similar to what I plan to do for this project,ie: use a genetic algorithm for designing Machine learning pipelines. Also this project includes several Machine learning components that I need in my project like lemmatizers, stemmers, stopwords removal functions, etc. So I may use them as a reference in the implementation phase.
Why do I still want to build a new tool for optimization rather than using AutoGoal?
AutoGoal is such a well designed and simple to use framework that I could do that. I would still need to add the code for implementing algorithms that are not included(for example it doesn't include Transformer models) but I could leverage their optimization module rather than building a new one.
I still consider it is a better idea to build a new optimization framework because the Optimization component of AutoGoal is a black box, it doesn’t use the GE algorithm I want to use for this project but a more complex version which we didn’t study in the course material(PGE) and I believe it’s out of the scope for this project to research alternatives to GE, additionally If I used AutoGoal I would like to understand fully how their optimization module works and although open source it’s not designed to be easily modified, I think it would be much more comprehensible for me to design my own optimizer from scratch using a GE.

Table product comparisons

System	Open source	Pipeline design	NAS	Includes transformer models	Supports text classification
HF AutoTrain		X		X	x
AutoPytorch	X	X	X		
AutoGoal	X	X	X		x
TPOT	X	X			
Mine	X	X	X	X	x

Design

Project AIMs

- Develop a Python framework that optimizes the design of a multi class text classification pipeline using a genetic algorithm
- Ensure the fitness function(classification macro f1) improves over generations to prove the GA is working.

Below I will describe the key components of my system.

Evolutionary algorithm
For implementing the evolutionary algorithm I'm going to utilize the Python package DEAP(Fortin et al., 2012)[8]. In my literature review I found TPOT AutoML uses it and also it is used by many other autoML frameworks.

Parameter space

Step: Text preprocessing
Options:
Remove stop words: yes, no
Word normalization: Lemmatization, stemming, none
Lowercasing: yes, no
Punctuation mark removal: yes, no
TF-IDF based word removal: yes, no
Justification:
Gathered the most used preprocessing techniques from the following articles [14] [15].
"TF-IDF based word removal" is like stopwords removal but the list of stopwords is calculated for the training set using TF-IDF.

Step: Vectorization
Options:
- BERT base uncased
- Roberta base
- E5-large-v2
- e5-base-v2
- e5-small-v2

Justification:
I selected BERT and Roberta base because they are common Transformer baselines today. I have used them in several projects and they are known to work well in general.
Also I wanted to add STOTA models so decided to include E5-large-v2 which at the time of writing is the top 1 on the Hugging face's text Embedding Benchmark Leaderboard[16]

additionally I included the base and small versions of the same model which are as well in the top 10 of the leaderboard.

Step: Vector combination
Options:
- Concat
- Mean
- Sum
- Max pooling
- First token, ie: use embedding of first token, [CLS] token for BERT based models

Justification:
In NLP practice these are the most common methods of combining word vectors to obtain a sentence or document embedding. More sophisticated techniques exist like using a Neural network to perform the combination of word vectors, for the sake of simplicity and for saving compute resources I will stick with the simple methods.

Step: Design NN architecture for classification(NAS)
Options:
- Feed forward hidden layers: range between 0 and 10
- Dropout_in_layers: none, input, all.
- Per_dropout: range between 0 and 0.2
- Max neurons: range between 50 and 300

Justification:
I took the idea of shapedMLP from AutoPytorch(analyzed in literature review), it consists of only optimizing a few parameters of the NN architecture which have a big impact on their performance. From their research I learnt that it's not necessary to optimize the number of neurons per layer, it's more important to optimize the number of layers.
For Dropout_in_layers I only give the option to add to None, input layer or all layers because if wanted to support dropout at specific layers would create a dependency between options(some values would only be available if the layer exists) and prefer to avoid that complexity, also dropout is recommended to be added in layers with highest number of parameters so input layer is the best candidate.
Dimension of the layers: The number of neurons per layer can be calculated automatically by starting with a number of neurons in the first hidden layer selected by the GA under "max neurons" parameter and then proportionally reducing it in the consecutive layers until reaching the dimension of the last layer which is going to be equal to the number of unique classes detected from the dataset.

Step: Training hyperparameters
Options:
- Epochs: 1,2,3,4,5,6,7,8,9,10
- Learning rate: range between 0 and 0.1
- Batch size: 1, 2, 4, 8, 16

Justification:
The 3 most important training hyperparameters. Maximum number of epochs is restricted to 10 to avoid running long experiments. If a good combination is found that works well with training for 10 epochs or less, engineers could experiment with more epochs later.

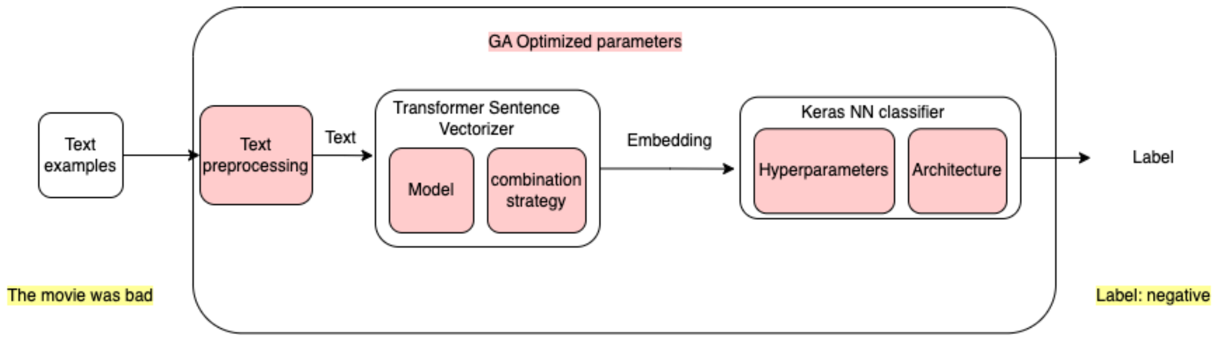
Cache
I will implement a cache to store the last hidden states of the Transformer model. Whenever my vectorizer component processes a list of sentences, it will first calculate the hash of these sentences. Then, it will check whether their corresponding word embeddings (i.e., the last hidden states) are already stored in the cache. If the embeddings are in the cache, it will save time by not running the model. If not, it will run the model to obtain the word embeddings, then add these to the cache before returning them.
Possible combinations for preprocessing techniques is 48 and for vectorization I have 5 possible models, that means for every row in the training dataset I will have 240 possible embeddings, if every word embedding requires at most 4 Kb of memory (biggest model E5-large-v2 has a dimension of 1024 per token, ie: 1024 floating point numbers of 4 bytes each = 1024*4 bytes = 4 Kb) and sentences have 512 tokens, that's 2048 Kb per sentence, 2Mb. For a small training set of 300 rows storing all embeddings in memory would require 600Mb and for storing all 240 combinations that's 144 Gb. Storing the cache in RAM memory on my 32Gb RAM macbook is not viable so I will Implement a Cache module that caches embeddings on the hard drive.

Fitness function
For calculating the fitness score when GA starts running it should be given a validation set of data. After designing and training a pipeline my fitness function will calculate the Macro average F1 score on the validation set and that will be my fitness metric. I chose this metric because for the problem of multiclass classification it's a single number that accounts for the precision and recall of all classes equally even if they are unbalanced.
From the literature review TPOT AutoML recommends performing cross validation for the fitness function since otherwise the GA could overfit the configuration for the validation set. However for the sake of simplicity and for saving compute resources I will rely on a single validation set.

Most important libraries and modules I will use

- DEAP, for evolutionary algorithm
- Hugging face Transformers, for generating embeddings from transformer models
- Keras for implementing the NN topology and training it

The following diagram shows the entire pipeline of my system where red boxes contain parameters optimized by the GA



Evaluation

The most important thing to evaluate in my system is whether the parameters selected by the GA algorithm yield a higher fitness score overtime, ie: whether the pipelines designed are getting better as the search progresses. So to evaluate my system after every fitness score is generated I will print to console the score to the terminal and add the entry to a CSV file that contains the parameters used, generation number and the fitness score obtained so that later I will plot the progress of fitness over time.
I will consider this project successful if the plot shows an increase in classification macro average F1 over time.
I will run my system on 2 datasets, an easy one and a more challenging one. I will use the easiest for the prototyping and development phases to validate that the system works but at the end of the project will run it on the second to see how it performs with less trivial data. The easy dataset is IMDB Dataset of 50K Movie Reviews[17], this is a sentiment analysis dataset with only 2 classes: positive and negative. The second dataset[22] has 3 classes: “Physics vs Chemistry vs Biology”.

Work Plan

I have divided the work into 4 sections:

Milestone 1/ prototype

Goal: Have a working GA that optimizes a pipeline for text classification. Pipeline will contain the most important components: vectorizer, vector combination and ML model(NAS). Each component will have only one possibility implemented but will print to screen the parameters received from the GA framework to show how GA selects the parameters. Also will implement a basic fitness metric that returns accuracy metric.
Ready by sprint: sprint 3

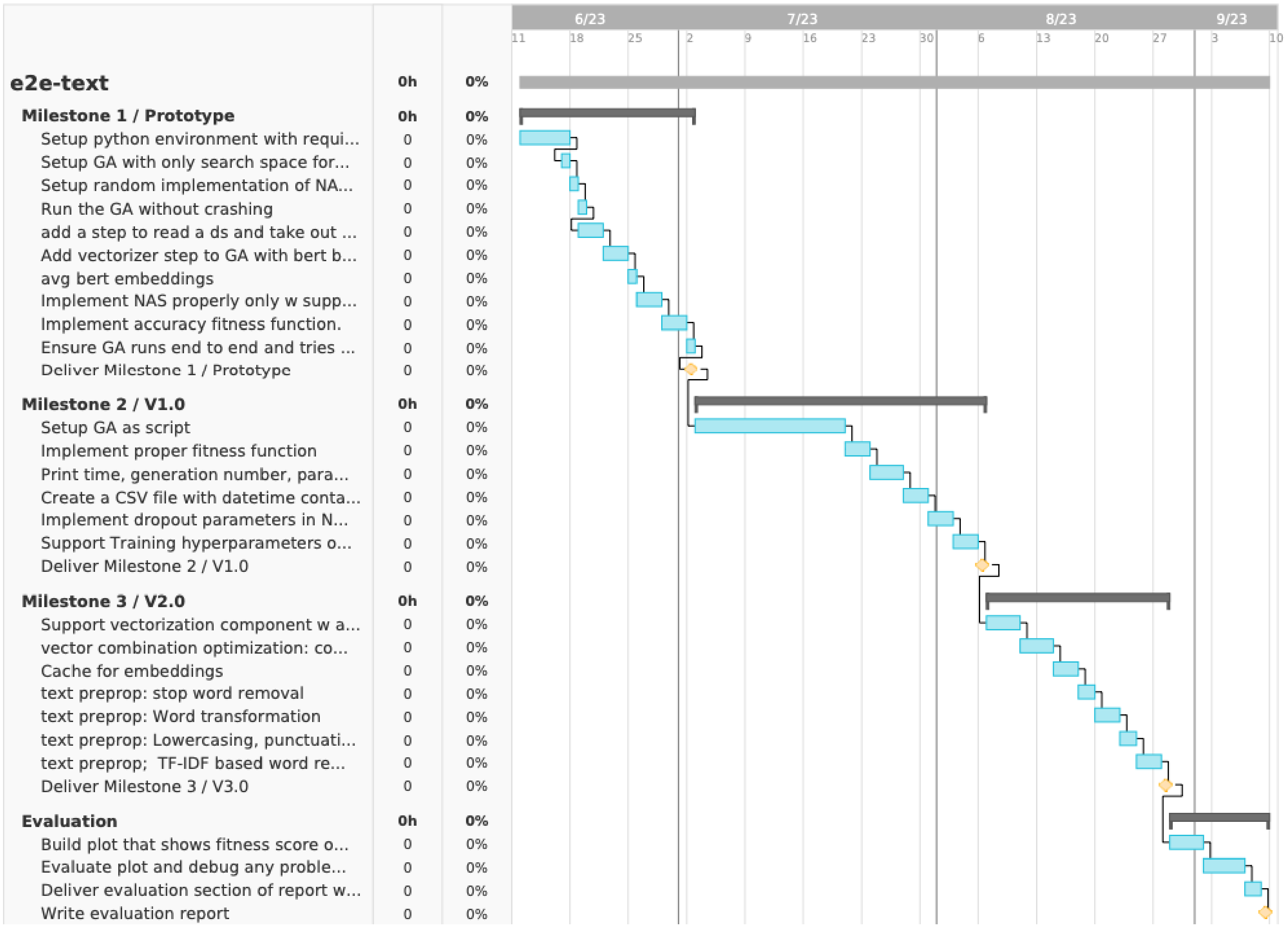
Milestone 2 / V1.0

Goal: Make prototype framework runnable as a script that reads parameters from a file, Support all parameters in the NAS component including hyperparameter optimization and evaluation metrics that track progress across generations and written to a CSV file.
ready by sprint: sprint 5

Milestone 3 / V2.0
Goal: Support all parameters specified in the design section and ensure when I run the GA as time progresses evaluation metrics improve. This is the final deliverable I plan on submitting for this module.
ready by sprint: TBC, need to review and plan after milestone 2 is reached

Evaluation
Goal: Show the improvement of validation macro F1 score as the GA runs for more generations. Ideally an improvement is seen in the plot otherwise this phase would become a debugging phase to find out why my GA is not working as expected.
I'm going to divide my work into sprints of one week duration. My goal is to have a working prototype by the end of sprint 3.
The details of the tasks and dates of every sprint can be seen in the Sprints section of the Appendix.

Gantt chart:



e2e-text	**0h**	**0%**
Milestone 1 / Prototype	**0h**	**0%**
Setup python environment with requi...	0	0%
Setup GA with only search space for...	0	0%
Setup random implementation of NA...	0	0%
Run the GA without crashing	0	0%
add a step to read a ds and take out ...	0	0%
Add vectorizer step to GA with bert b...	0	0%
avg bert embeddings	0	0%
Implement NAS properly only w supp...	0	0%
Implement accuracy fitness function.	0	0%
Ensure GA runs end to end and tries ...	0	0%
Deliver Milestone 1 / Prototype	0	0%
Milestone 2 / V1.0	**0h**	**0%**
Setup GA as script	0	0%
Implement proper fitness function	0	0%
Print time, generation number, para...	0	0%
Create a CSV file with datetime conta...	0	0%
Implement dropout parameters in N...	0	0%
Support Training hyperparameters o...	0	0%
Deliver Milestone 2 / V1.0	0	0%
Milestone 3 / V2.0	**0h**	**0%**
Support vectorization component w a...	0	0%
vector combination optimization: co...	0	0%
Cache for embeddings	0	0%
text preprop: stop word removal	0	0%
text preprop: Word transformation	0	0%
text preprop: Lowercasing, punctuati...	0	0%
text preprop; TF-IDF based word re...	0	0%
Deliver Milestone 3 / V3.0	0	0%
Evaluation	**0h**	**0%**
Build plot that shows fitness score o...	0	0%
Evaluate plot and debug any proble...	0	0%
Deliver evaluation section of report w...	0	0%
Write evaluation report	0	0%

Implementation

Prototype

Features

I have implemented an initial version of my system, it has the following features:

- Evolutionary optimization using Deap module with mating, crossover and mutation.
- Genotype specification for optimizing 3 parameters: max no of neurons, number of layers and learning rate.
- Parsing function that converts genotype(float number optimized by the GA) into phenotype(actual parameters used by my classification pipeline)

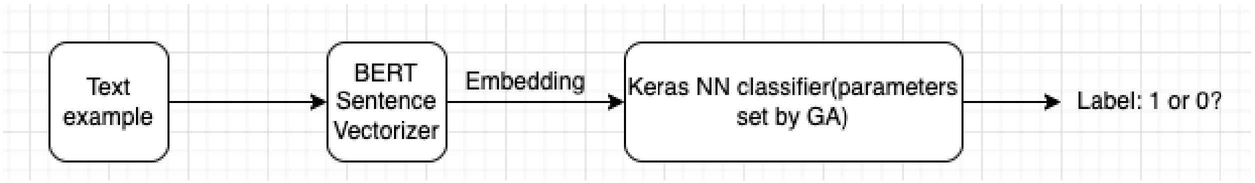
- NAS component that receives arguments: max no of neurons, number of layers and learning rate from the GA and builds a neural network model with such features
- BERT vectorizer component, generate text embeddings from BERT model
- Fitness function that returns the accuracy of the pipeline on a testing set.
- Hall of fame and statistics for tracking performance and best individual across generations.

Implementation

My prototype runs as a Python script that has no arguments(All parameters are hardcoded in the code for now). The most important parameters are stored at the top of the screen as constants:

```
DS_PATH = 'datasets/IMDB_Dataset.csv'
RANDOM_SEED = 42
TEXT_COLUMN = 'review'
LABEL_COLUMN = 'sentiment'
GENERATIONS = 15
N_POPULATION = 3
PROB_MUTATION = 0.2
MAX_SAMPLE_SIZE_DS = 150
NEURONS_CHANGE_FACTOR = 0.8 # reduce neurons by 20% each layer
```

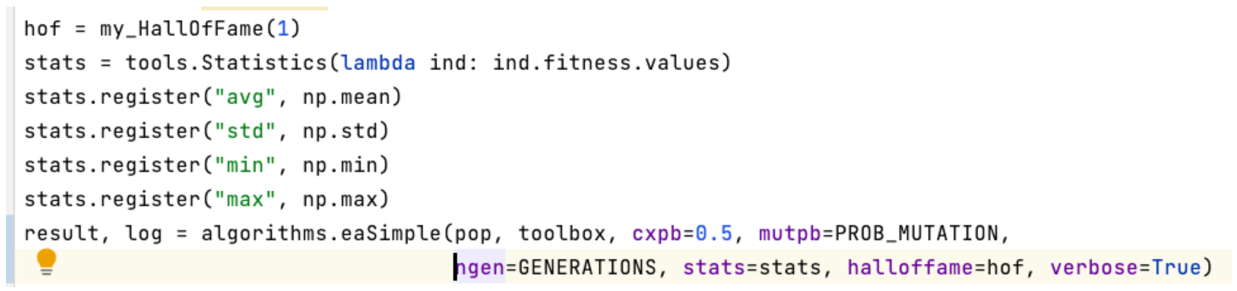
Every individual running in the GA loads the following text classification pipeline when evaluated by the fitness function:



On the above pipeline the following parameters of the Neural network classifier are optimized by the GA: Max number of neurons, learning rate and number of inner layers.
The specification for the GA parameters with the bounds and type of values supported is defined in the script as:

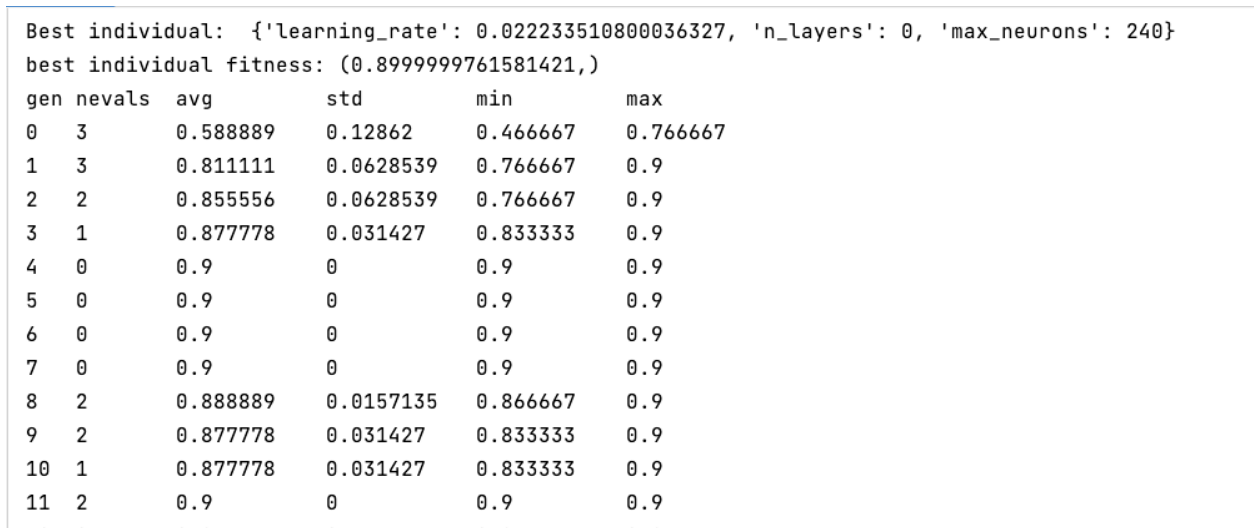
```
GENOTYPE_SPEC = [
    {"name": "learning_rate", "type": "float_range", "bounds": [0.001, 0.1]},
    {"name": "n_layers", "type": "int_range", "bounds": [0, 10]},
    {"name": "max_neurons", "type": "int_range", "bounds": [50, 300]}
]
```

For gathering metrics for evaluation the GA implements a hall of fame object which keeps track of the best individual and DEAP framework allows tracking statistics like: avg, min, max and standard deviation of fitness score at every generation. I have setup the hall of fame and statistics in the code below:



```
hof = my_HallOfFame(1)
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)
result, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=PROB_MUTATION,
                                  ngen=GENERATIONS, stats=stats, halloffame=hof, verbose=True)
```

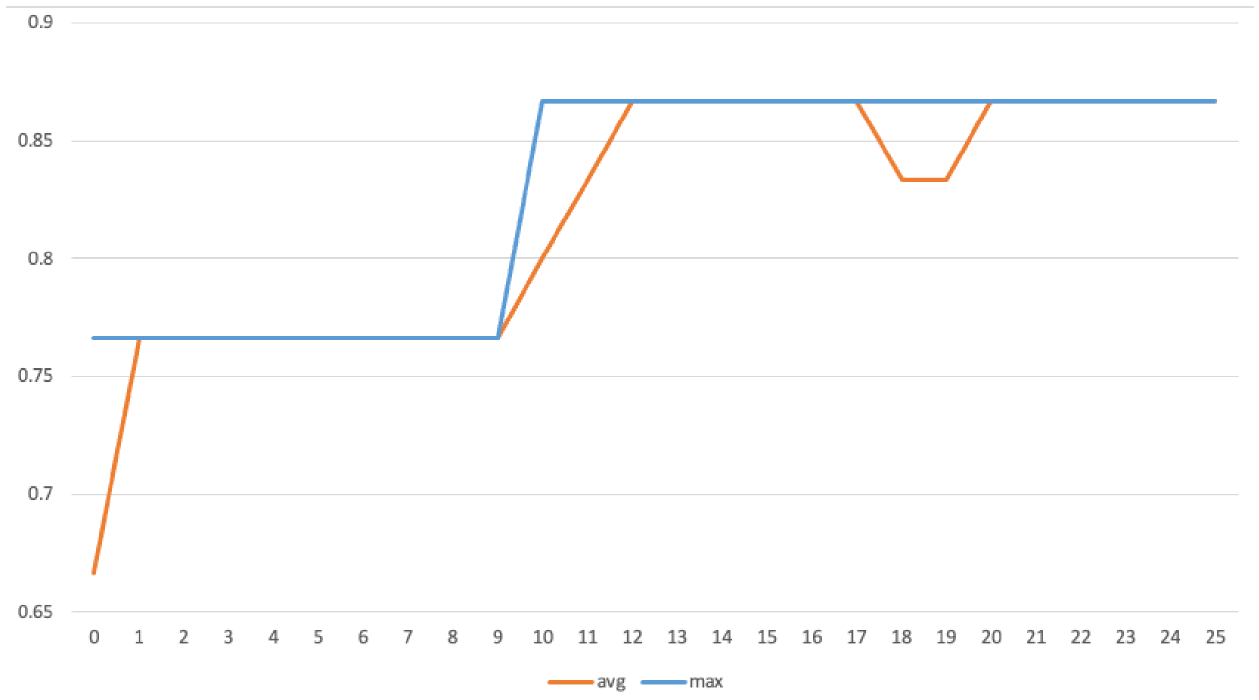
When script finishes running it prints to screen all the statistics gathered for evaluation:



```
Best individual:  {'learning_rate': 0.022233510800036327, 'n_layers': 0, 'max_neurons': 240}
best individual fitness: (0.8999999761581421,)
gen nevals  avg         std         min         max
0   3       0.588889    0.12862     0.466667    0.766667
1   3       0.811111    0.0628539   0.766667    0.9
2   2       0.855556    0.0628539   0.766667    0.9
3   1       0.877778    0.031427    0.833333    0.9
4   0       0.9         0           0.9         0.9
5   0       0.9         0           0.9         0.9
6   0       0.9         0           0.9         0.9
7   0       0.9         0           0.9         0.9
8   2       0.888889    0.0157135   0.866667    0.9
9   2       0.877778    0.031427    0.833333    0.9
10  1       0.877778    0.031427    0.833333    0.9
11  2       0.9         0           0.9         0.9
```

Evaluation of my prototype

I have run my system on a sample of 150 examples of the IMDB Dataset of 50K Movie Reviews[17], this is a sentiment analysis dataset with only 2 classes: positive and negative. My initial version seems to work well, as shown below the classification accuracy(Y axis) increases with the number of generations(X axis):



Refer to [fig 1] in appendix for the table that produced the above plot.

Limitations of my prototype

- The fitness score uses accuracy metric but such metric doesn't account for class imbalances, I will later implement macro average F1 as fitness
- GA was only run for 25 generations, it should run for hundreds of generations to get a reliable view of the performance over time, since between close generations randomness affects the fitness score and the improvement may not be very clear. The more generations we run the GA for, the more reliable the metrics will be and their plot would show clearer patterns.
- Evaluation was run with a tiny population of 3 individuals, this should be bigger to reduce randomness in my evaluation, also fitness should improve faster across generations for a bigger population.
- The pipeline only supports a few optimizable parameters
- Both training and testing sets are very small, only 150 examples in total, this can give a biased view of the problem. It's difficult to increase this number since my pipeline runs very slow due to high processing power required by the algorithms used.
- Embeddings are not cached so they are generated for every evaluation, this is the slowest part of the pipeline so I will later implement a cache for it.

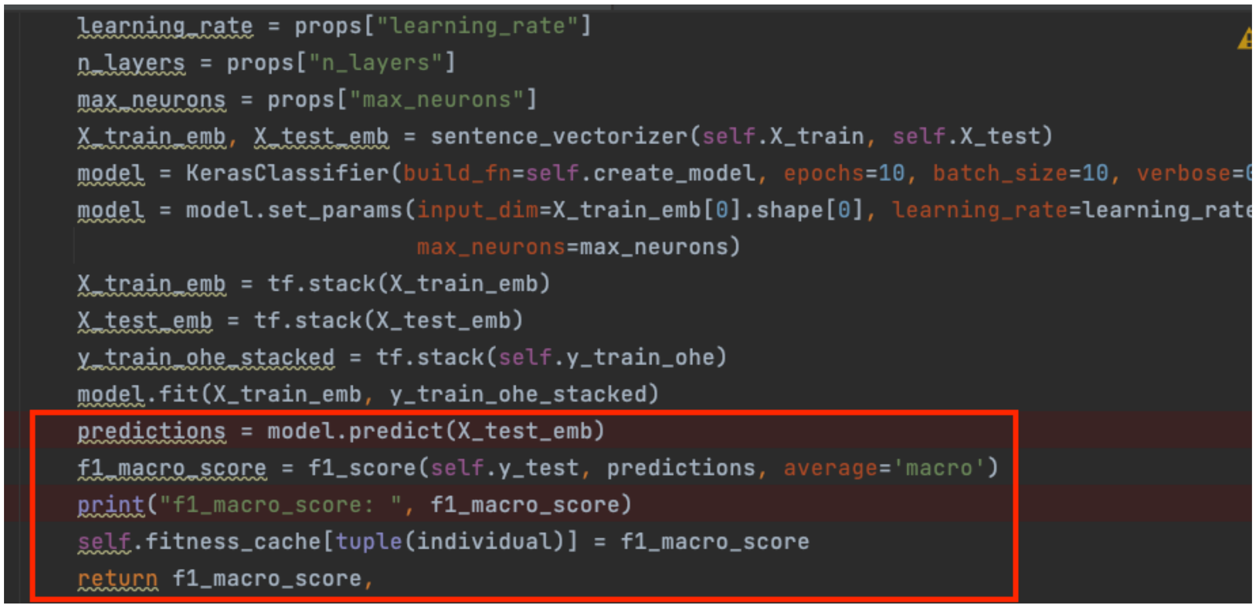
Testing Framework

In order to test new features in my system I'm going to implement the following features:

1- Proper fitness score

As shown in the design section the proper metric for fitness score is macro average F1, but the one implemented in the prototype is accuracy. I will mplement the proper fitness function to get accurate metrics of my system.

I use scikit-learn's metrics.f1_score function[18] for performing the macro average F1 calculation, in the code below I show where I added the f1_score call in the evaluation function of the GA, which return value is used as fitness score:

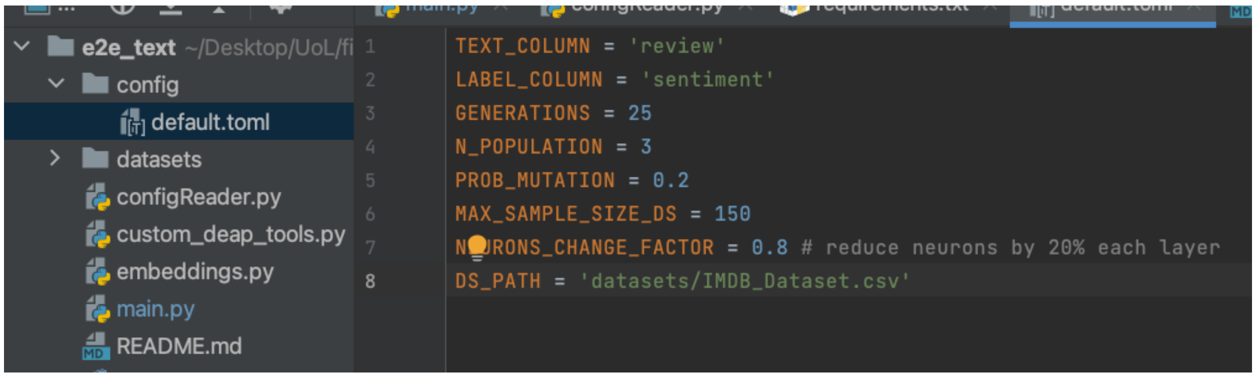


```
learning_rate = props["learning_rate"]
n_layers = props["n_layers"]
max_neurons = props["max_neurons"]
X_train_emb, X_test_emb = sentence_vectorizer(self.X_train, self.X_test)
model = KerasClassifier(build_fn=self.create_model, epochs=10, batch_size=10, verbose=
model = model.set_params(input_dim=X_train_emb[0].shape[0], learning_rate=learning_rate
                         max_neurons=max_neurons)
X_train_emb = tf.stack(X_train_emb)
X_test_emb = tf.stack(X_test_emb)
y_train_ohe_stacked = tf.stack(self.y_train_ohe)
model.fit(X_train_emb, y_train_ohe_stacked)
predictions = model.predict(X_test_emb)
f1_macro_score = f1_score(self.y_test, predictions, average='macro')
print("f1_macro_score: ", f1_macro_score)
self.fitness_cache[tuple(individual)] = f1_macro_score
return f1_macro_score,
```

2- Package GA as a script that reads parameters from configuration files

I'm going to package my prototype code as a script that receives only one parameter: "config" which specifies the name of the configuration file to use. In the directory config I will store configuration files, each has parameters for the GA, a default.toml file contains the default parameters and when the config argument is given to the script the toml file with that name is read and its parameters overwrite the ones in default.toml. In order to support this I will write a module that reads these config files.

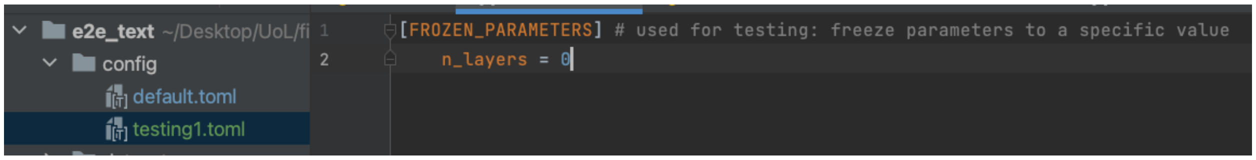
Default config file:



```
TEXT_COLUMN = 'review'
LABEL_COLUMN = 'sentiment'
GENERATIONS = 25
N_POPULATION = 3
PROB_MUTATION = 0.2
MAX_SAMPLE_SIZE_DS = 150
NEURONS_CHANGE_FACTOR = 0.8 # reduce neurons by 20% each layer
DS_PATH = 'datasets/IMDB_Dataset.csv'
```

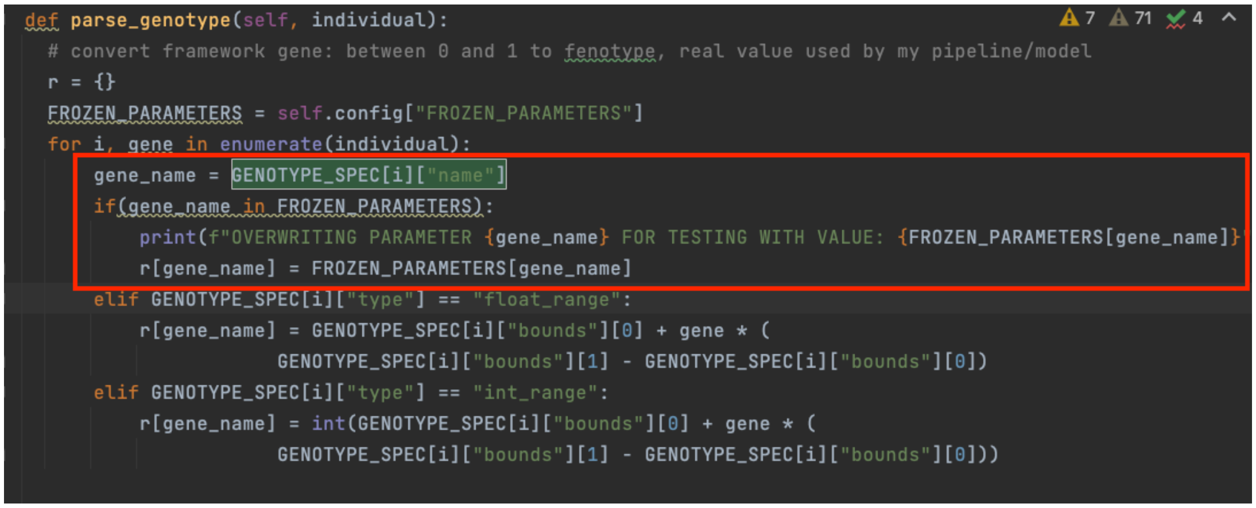
This allows me to set up several configurations, each stored in its own file and only with the change of a single argument in the script I can select which one runs. This is useful for testing because I could write a config file specific for testing that runs on a smaller population on smaller data and even have specific config files for testing specific features.

3-Ability to force the use of a GA parameter
When implementing new features I want the GA to change all other parameters except for the one I want to test. This feature allows specifying which module and setting I want to test so that the GA doesn't change it.
This is how to specify parameters to freeze in the configuration file:



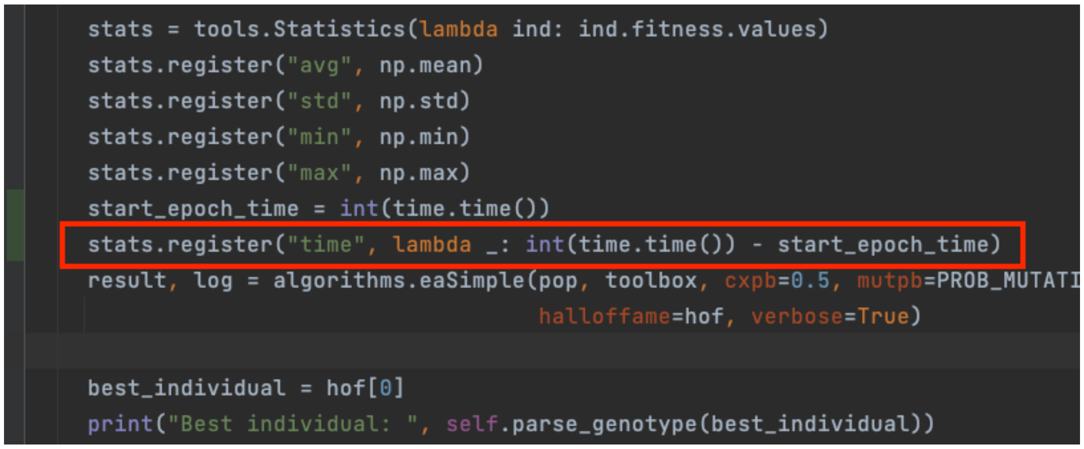
```
[FROZEN_PARAMETERS] # used for testing: freeze parameters to a specific value
    n_layers = 0
```

Frozen parameters overwrite the GA decision during parsing of the genotype to obtain the parameters for the model(phenotype).



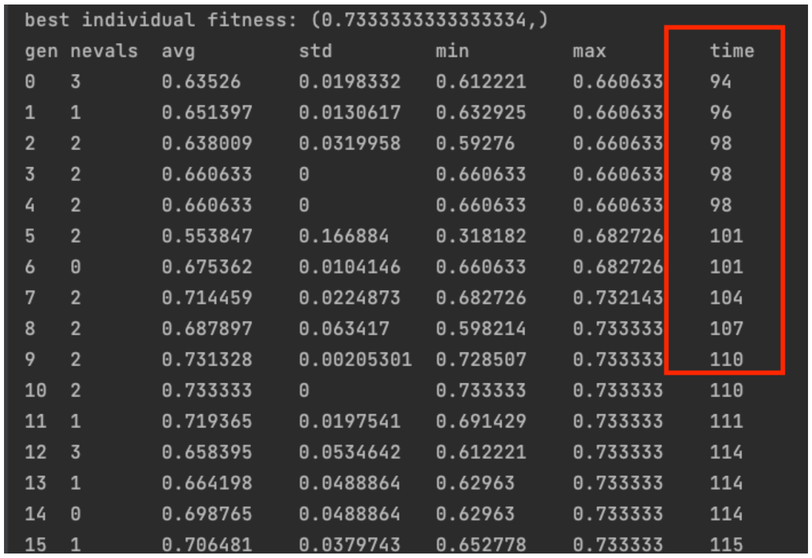
```
def parse_genotype(self, individual):
    # convert framework gene: between 0 and 1 to fenotype, real value used by my pipeline/model
    r = {}
    FROZEN_PARAMETERS = self.config["FROZEN_PARAMETERS"]
    for i, gene in enumerate(individual):
        gene_name = GENOTYPE_SPEC[i]["name"]
        if(gene_name in FROZEN_PARAMETERS):
            print(f"OVERWRITING PARAMETER {gene_name} FOR TESTING WITH VALUE: {FROZEN_PARAMETERS[gene_name]}
            r[gene_name] = FROZEN_PARAMETERS[gene_name]
        elif GENOTYPE_SPEC[i]["type"] == "float_range":
            r[gene_name] = GENOTYPE_SPEC[i]["bounds"][0] + gene * (
                        GENOTYPE_SPEC[i]["bounds"][1] - GENOTYPE_SPEC[i]["bounds"][0])
        elif GENOTYPE_SPEC[i]["type"] == "int_range":
            r[gene_name] = int(GENOTYPE_SPEC[i]["bounds"][0] + gene * (
                        GENOTYPE_SPEC[i]["bounds"][1] - GENOTYPE_SPEC[i]["bounds"][0]))
```

4- Add time to GA metrics
In order to optimize the performance of the GA I need to know how fast my algorithm runs, eg: how long does it take to run a generation or are there some configurations slower than others? In order to track this I will add a time delta to every statistic entry, the time delta will be the number of seconds elapsed between the time the GA started running and the generation finished running.
The code for it is added here:



```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)
start_epoch_time = int(time.time())
stats.register("time", lambda _: int(time.time()) - start_epoch_time)
result, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=PROB_MUTATI
                                  halloffame=hof, verbose=True)

best_individual = hof[0]
print("Best individual: ", self.parse_genotype(best_individual))
```

And this is how statistics look now after GA runs:



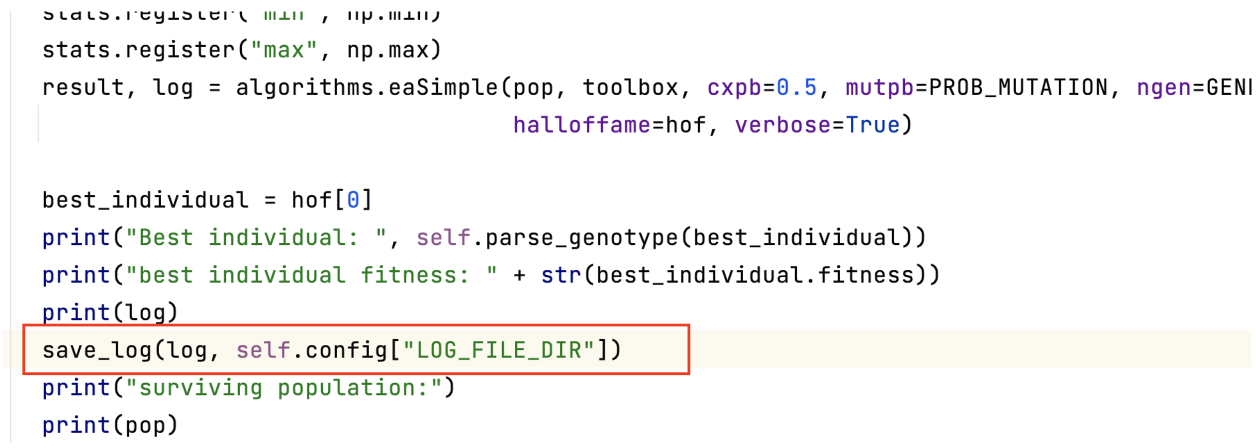
```
best individual fitness: (0.7333333333333334,)
gen nevals  avg         std         min         max         time
0   3       0.63526     0.0198332   0.612221    0.660633    94
1   1       0.651397    0.0130617   0.632925    0.660633    96
2   2       0.638009    0.0319958   0.59276     0.660633    98
3   2       0.660633    0           0.660633    0.660633    98
4   2       0.660633    0           0.660633    0.660633    98
5   2       0.553847    0.166884    0.318182    0.682726    101
6   0       0.675362    0.0104146   0.660633    0.682726    101
7   2       0.714459    0.0224873   0.682726    0.732143    104
8   2       0.687897    0.063417    0.598214    0.733333    107
9   2       0.731328    0.00205301  0.728507    0.733333    110
10  2       0.733333    0           0.733333    0.733333    110
11  1       0.719365    0.0197541   0.691429    0.733333    111
12  3       0.658395    0.0534642   0.612221    0.733333    114
13  1       0.664198    0.0488864   0.62963     0.733333    114
14  0       0.698765    0.0488864   0.62963     0.733333    114
15  1       0.706481    0.0379743   0.652778    0.733333    115
```

The first generation takes 94 seconds because the script takes 90 seconds to load the neural networks in memory and start working.

5- save GA metrics to a CSV file for analysis after running

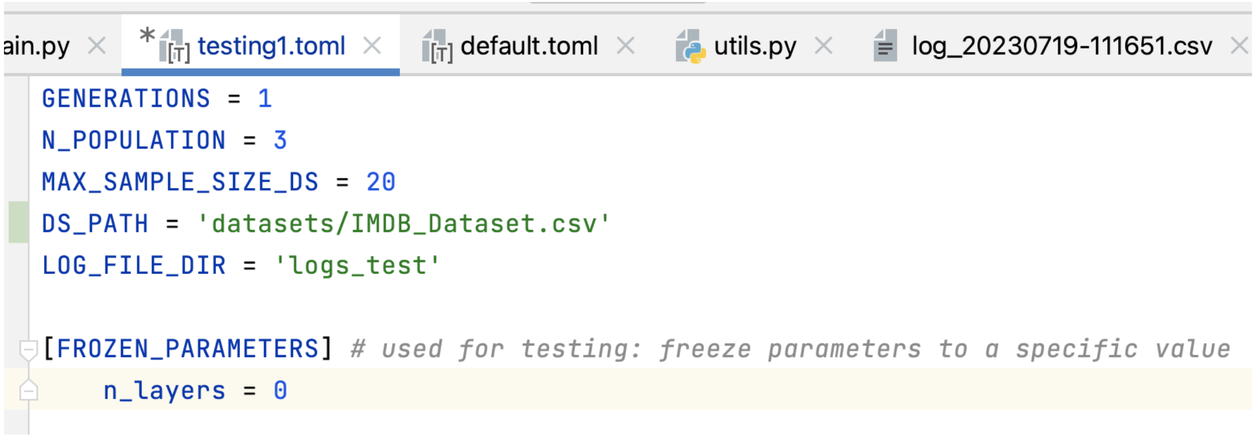
All metrics gathered during the parameter search will be saved to a CSV file so that I have them stored on disk and can later import them into Excel to plot the progress over time.This feature is mainly for evaluation but it can also help with testing since I may need to analyze the statistics of the system when testing a specific feature to see how it's affecting the performance of the system over time.

At the end of the script I added a call to a helper function that stores logger object in a CSV:


```
    stats.register("min", np.min)
    stats.register("max", np.max)
    result, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=PROB_MUTATION, ngen=GENI
                                      halloffame=hof, verbose=True)

    best_individual = hof[0]
    print("Best individual: ", self.parse_genotype(best_individual))
    print("best individual fitness: " + str(best_individual.fitness))
    print(log)
    save_log(log, self.config["LOG_FILE_DIR"])
    print("surviving population:")
    print(pop)
```

save_log takes the path to the logs directory from the configuration as argument, this allows me to have different folders for log files, one for important runs(evaluations) and another one for test runs.

6- Testing Process
After adding a new feature to my system I validate the feature works properly and integrates well with the rest of the system by running 2 tests:
1- run GA end to end on a small sample of the simple dataset IMDB(20 examples) on a small population and number of generations(1 to 3) to validate feature works, for testing new components like a new Transformer model or text preprocessing technique I set that component as frozen on FROZEN_PARAMETERS setting.
2- Run the GA with a bigger population, data samples and number of generations to validate fitness score still increases over time.
3-If necessary statistics for testing runs are stored as CSVs in /logs_test for a detailed analysis
Each type of test has its own configuration file where I set the dataset name, number of generations, number of samples, population, parameters to freeze, etc
For example for a quick validation(1) test I use the following:



```
GENERATIONS = 1
N_POPULATION = 3
MAX_SAMPLE_SIZE_DS = 20
DS_PATH = 'datasets/IMDB_Dataset.csv'
LOG_FILE_DIR = 'logs_test'

[FROZEN_PARAMETERS] # used for testing: freeze parameters to a specific value
    n_layers = 0
```

LOG_FILE_DIR has the suffix “_test” to store stats for test runs separately from the logs directory where only important evaluations are stored(logs).
If I were testing a specific feature like a new Transformer model I would specify it under FROZEN_PARAMETERS so that GA only uses the new model.

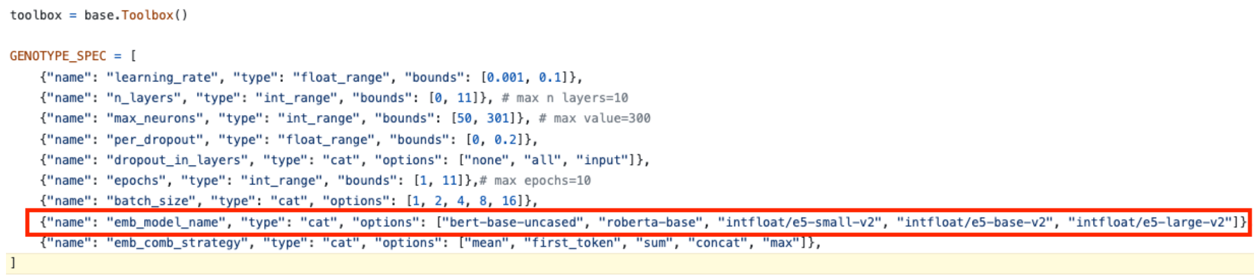
Development environment
E2e text is designed to run with Python V3.9 and requires the modules in [fig 2].
If you want to run the project on your machine follow these guidelines: [fig 3]

The development machine used for building and evaluating the system is a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 CPU and 32 Gb of memory Running Mac OS 13.4.

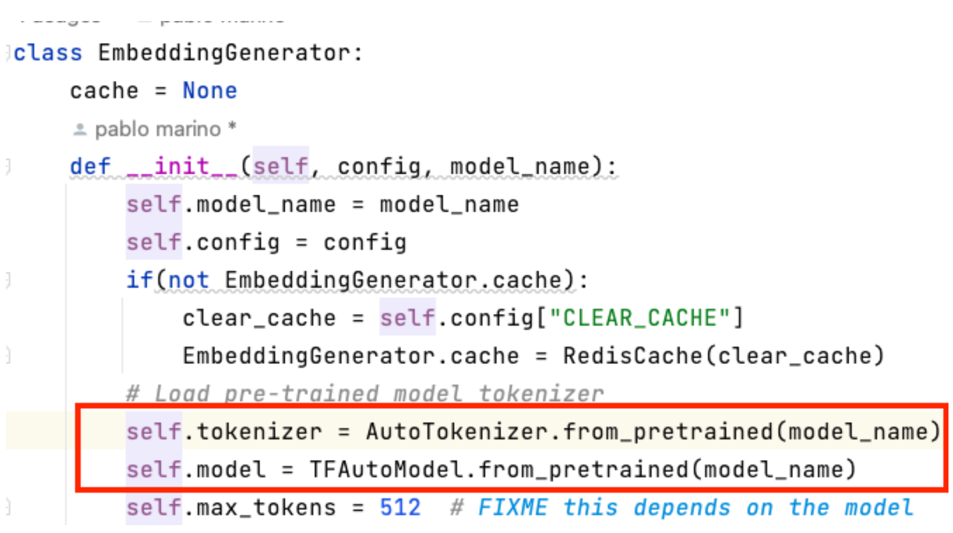
First Iteration (Milestone 2)
Features implemented in this iteration:
- Dropout parameters search in NAS
- Training hyperparameter optimization
- Optimization of vectorization models: bert, roberta, e5(small, base and large)
- Optimization of vector combination techniques: sum, avg, max pool, concat, first token
- Module for caching last hidden states of transformer models to hard drive.

Below are implementation details of some of the above features:

Optimization of vectorization models
Vectorization stage now supports 5 Transformer models defined in design section, this is what the new genome specification looks like:

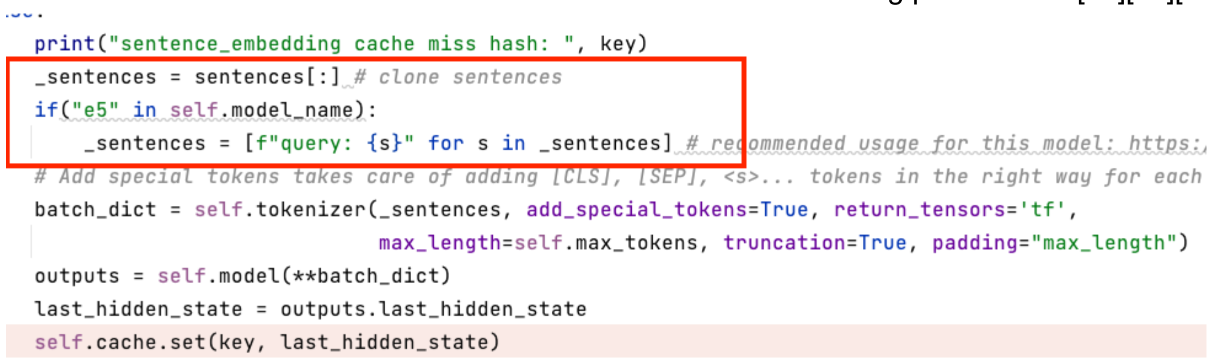

```
toolbox = base.Toolbox()

GENOTYPE_SPEC = [
    {"name": "learning_rate", "type": "float_range", "bounds": [0.001, 0.1]},
    {"name": "n_layers", "type": "int_range", "bounds": [0, 11]}, # max n layers=10
    {"name": "max_neurons", "type": "int_range", "bounds": [50, 301]}, # max value=300
    {"name": "per_dropout", "type": "float_range", "bounds": [0, 0.2]},
    {"name": "dropout_in_layers", "type": "cat", "options": ["none", "all", "input"]},
    {"name": "epochs", "type": "int_range", "bounds": [1, 11]},# max epochs=10
    {"name": "batch_size", "type": "cat", "options": [1, 2, 4, 8, 16]},
    {"name": "emb_model_name", "type": "cat", "options": ["bert-base-uncased", "roberta-base", "intfloat/e5-small-v2", "intfloat/e5-base-v2", "intfloat/e5-large-v2"]}
    {"name": "emb_comb_strategy", "type": "cat", "options": ["mean", "first_token", "sum", "concat", "max"]},
]
```

The vectorizer function is now a class that receives the model name in the init function and uses Transformer’s AutoTokenizer and TFAutoModel classes to load the appropriate architecture:



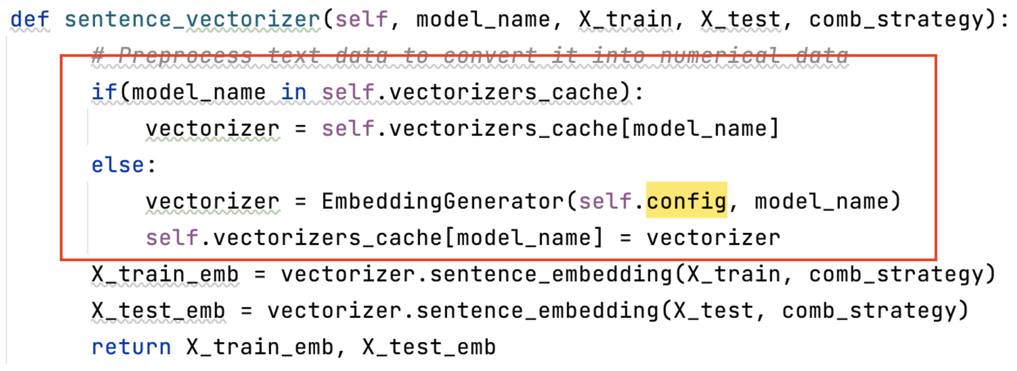
```
class EmbeddingGenerator:
    cache = None
    pablo marino *
    def __init__(self, config, model_name):
        self.model_name = model_name
        self.config = config
        if(not EmbeddingGenerator.cache):
            clear_cache = self.config["CLEAR_CACHE"]
            EmbeddingGenerator.cache = RedisCache(clear_cache)
        # Load pre-trained model tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = TFAutoModel.from_pretrained(model_name)
        self.max_tokens = 512  # FIXME this depends on the model
```

The process for obtaining the word embeddings is the same regardless of the model name, however for E5 models I must add the text “query: ” to the beginning of each sentence since that's how model was trained and authors recommend it for not losing performance[19][20][21]



```
print("sentence_embedding cache miss hash: ", key)
_sentences = sentences[:] # clone sentences
if("e5" in self.model_name):
    _sentences = [f"query: {s}" for s in _sentences] # recommended usage for this model: https:/
# Add special tokens takes care of adding [CLS], [SEP], <s>... tokens in the right way for each
batch_dict = self.tokenizer(_sentences, add_special_tokens=True, return_tensors='tf',
                            max_length=self.max_tokens, truncation=True, padding="max_length")
outputs = self.model(**batch_dict)
last_hidden_state = outputs.last_hidden_state
self.cache.set(key, last_hidden_state)
```

Every model architecture is instantiated in an EmbeddingGenerator object, and the GA caches these objects in order to avoid having to reload the entire Transformer model for every evaluation since this process can take up to 2 minutes. This is how I cache them:

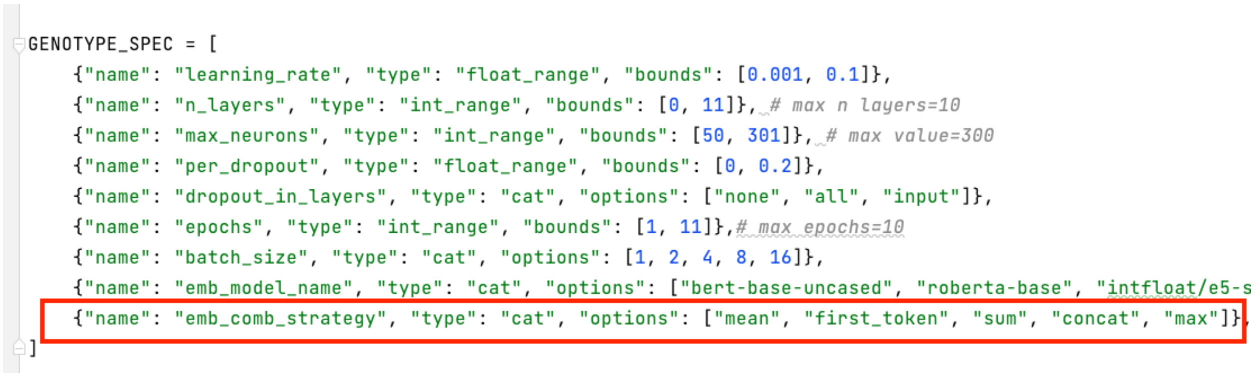


```
def sentence_vectorizer(self, model_name, X_train, X_test, comb_strategy):
    # Preprocess text data to convert it into numerical data
    if(model_name in self.vectorizers_cache):
        vectorizer = self.vectorizers_cache[model_name]
    else:
        vectorizer = EmbeddingGenerator(self.config, model_name)
        self.vectorizers_cache[model_name] = vectorizer
    X_train_emb = vectorizer.sentence_embedding(X_train, comb_strategy)
    X_test_emb = vectorizer.sentence_embedding(X_test, comb_strategy)
    return X_train_emb, X_test_emb
```

Because my GA only evaluates 5 models I can store them all in RAM memory and implement a simple cache using a Python dictionary.

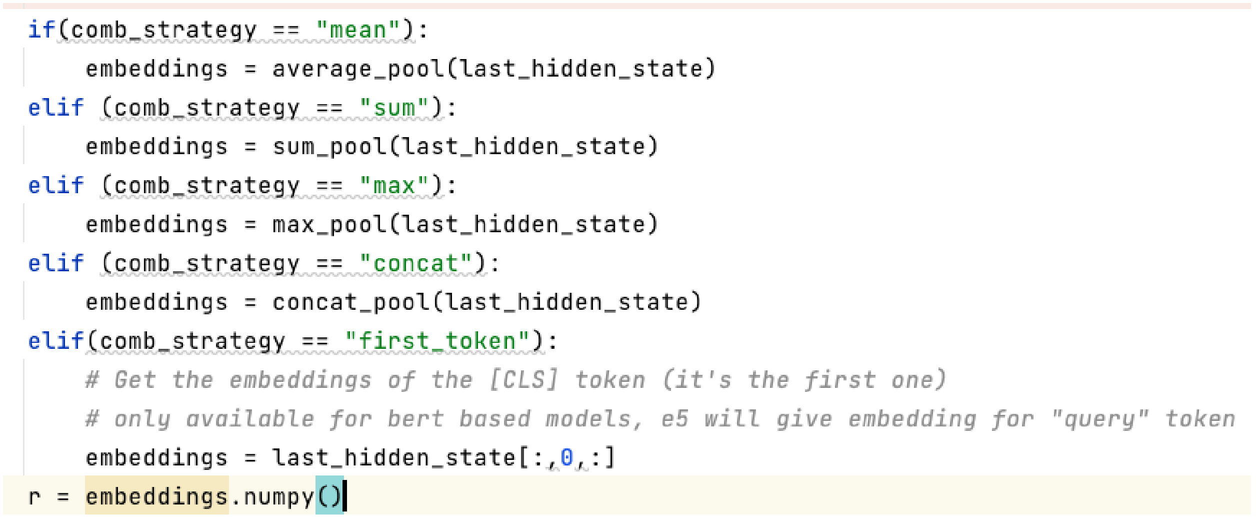
Optimization of vector combination techniques

Added options for word embedding combination operations to the genome specification:



```
GENOTYPE_SPEC = [
    {"name": "learning_rate", "type": "float_range", "bounds": [0.001, 0.1]},
    {"name": "n_layers", "type": "int_range", "bounds": [0, 11]}, # max n layers=10
    {"name": "max_neurons", "type": "int_range", "bounds": [50, 301]}, # max value=300
    {"name": "per_dropout", "type": "float_range", "bounds": [0, 0.2]},
    {"name": "dropout_in_layers", "type": "cat", "options": ["none", "all", "input"]},
    {"name": "epochs", "type": "int_range", "bounds": [1, 11]},# max epochs=10
    {"name": "batch_size", "type": "cat", "options": [1, 2, 4, 8, 16]},
    {"name": "emb_model_name", "type": "cat", "options": ["bert-base-uncased", "roberta-base", "intfloat/e5-s
    {"name": "emb_comb_strategy", "type": "cat", "options": ["mean", "first_token", "sum", "concat", "max"]},
]
```

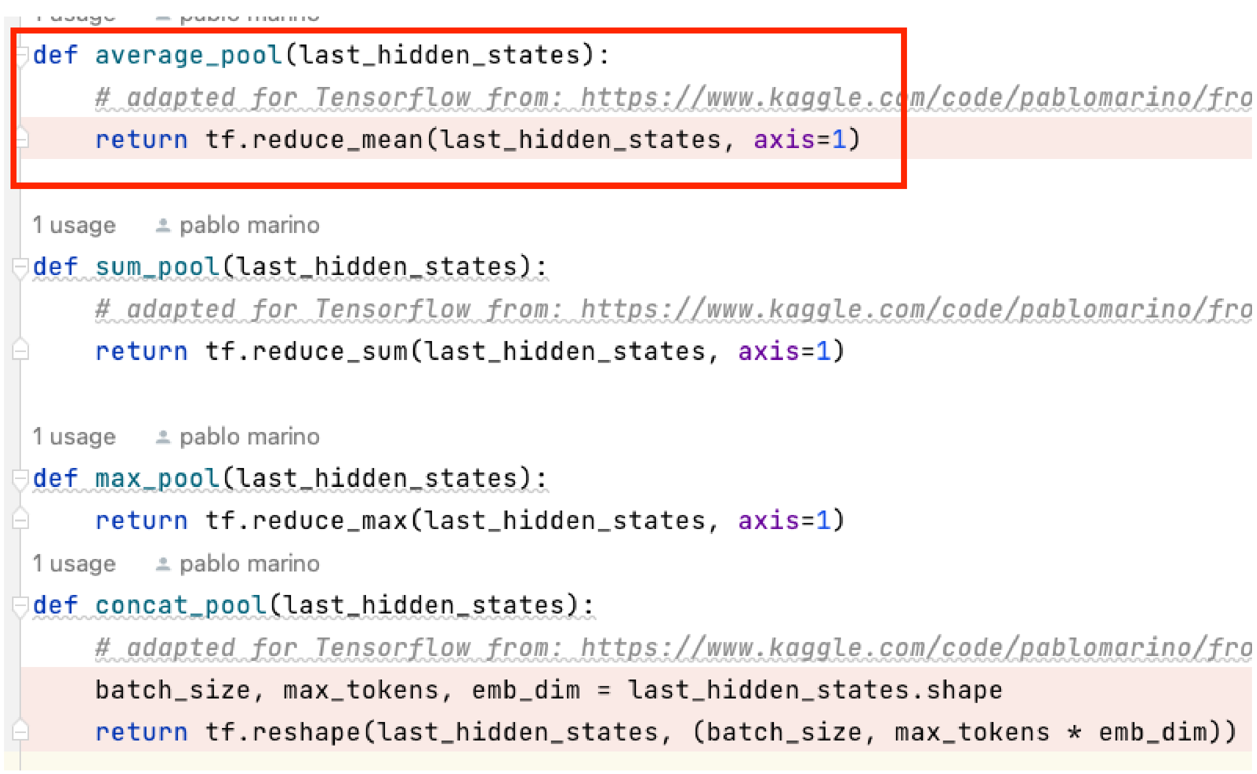
Every operation receives the last hidden states of the Transformer models and perform the defined operation on that:



```
if(comb_strategy == "mean"):
    embeddings = average_pool(last_hidden_state)
elif (comb_strategy == "sum"):
    embeddings = sum_pool(last_hidden_state)
elif (comb_strategy == "max"):
    embeddings = max_pool(last_hidden_state)
elif (comb_strategy == "concat"):
    embeddings = concat_pool(last_hidden_state)
elif(comb_strategy == "first_token"):
    # Get the embeddings of the [CLS] token (it's the first one)
    # only available for bert based models, e5 will give embedding for "query" token
    embeddings = last_hidden_state[:,0,:]
r = embeddings.numpy()
```

For example comb_strategy=first_token converts last_hidden_state from dimension: (batch_size, tokens_per_sentence, embedding_dimension) to (batch_size, embedding_dimension) because it only keeps the embedding of the first token for each sentence, for BERT and Roberta this is the [CLS] token and for E5 models this is “query” token added to the beginning of each sentence.

For other combination operations I have defined functions, for example for mean I perform the average of all embeddings in a sentence:

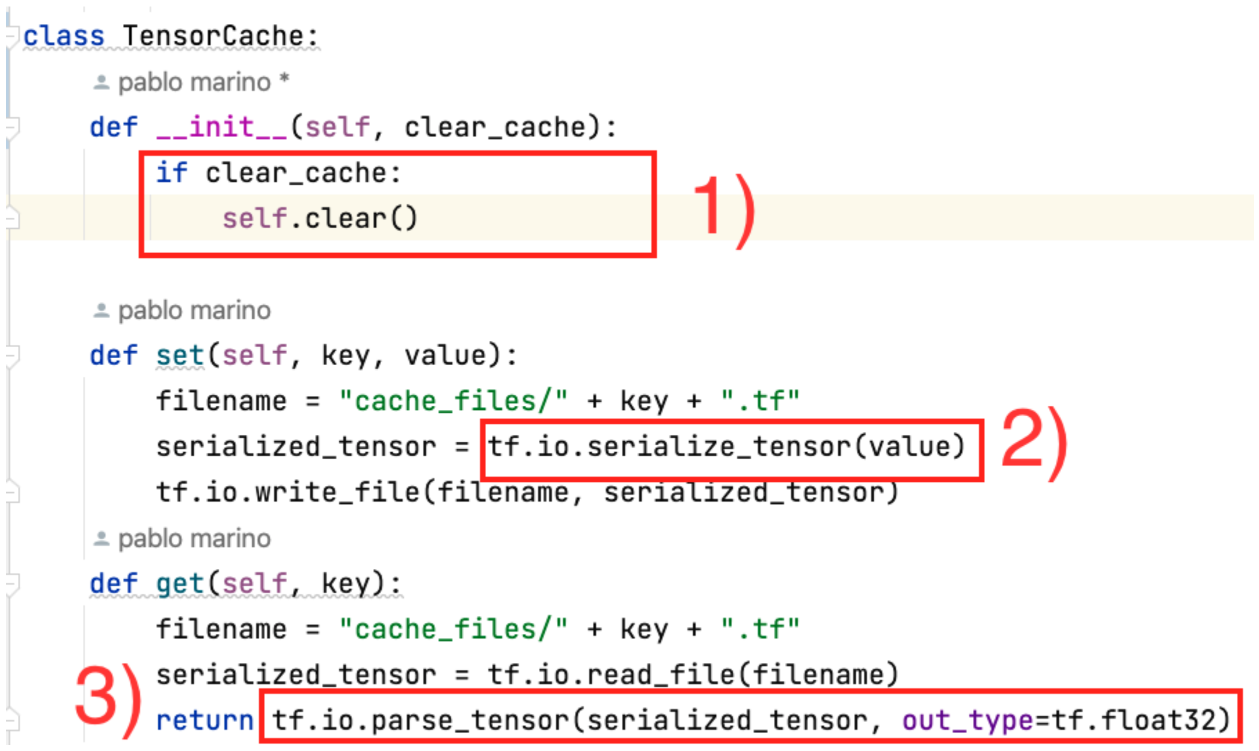


```
1 usage   pablo marino
def average_pool(last_hidden_states):
    # adapted for Tensorflow from: https://www.kaggle.com/code/pablomarino/fro
    return tf.reduce_mean(last_hidden_states, axis=1)

1 usage   pablo marino
def sum_pool(last_hidden_states):
    # adapted for Tensorflow from: https://www.kaggle.com/code/pablomarino/fro
    return tf.reduce_sum(last_hidden_states, axis=1)

1 usage   pablo marino
def max_pool(last_hidden_states):
    return tf.reduce_max(last_hidden_states, axis=1)
1 usage   pablo marino
def concat_pool(last_hidden_states):
    # adapted for Tensorflow from: https://www.kaggle.com/code/pablomarino/fro
    batch_size, max_tokens, emb_dim = last_hidden_states.shape
    return tf.reshape(last_hidden_states, (batch_size, max_tokens * emb_dim))
```

axis=1 refers to performing the average across the axis of tokens_per_sentence to average embeddings for all tokens in the sentence.

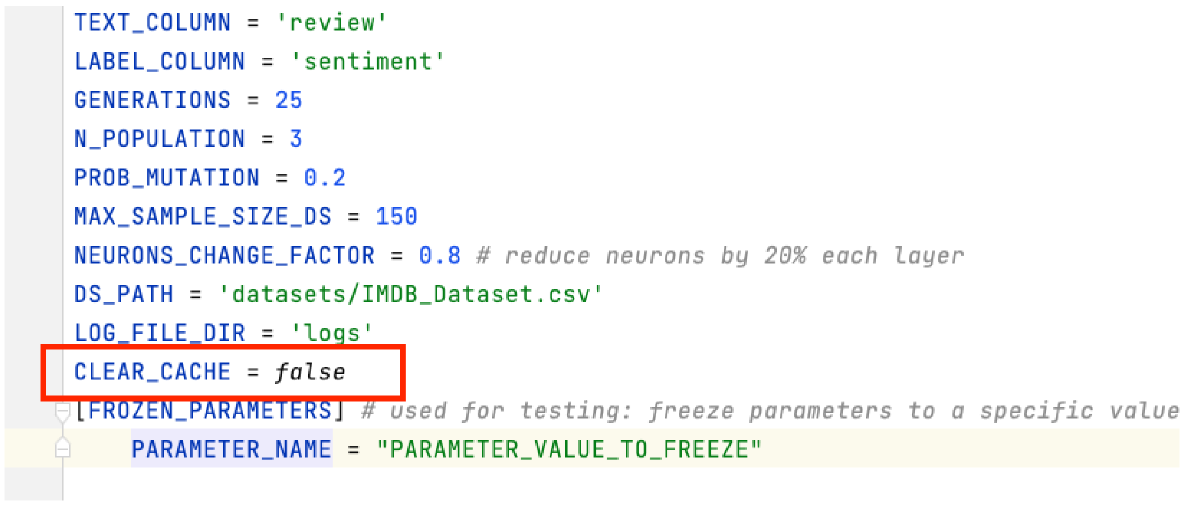
Tensor cache

Because I found no reliable database or module for caching Tensors to hard drive I have implemented a class called TensorCache that offers an interface for caching and retrieving Tensorflow tensors from the hard drive:



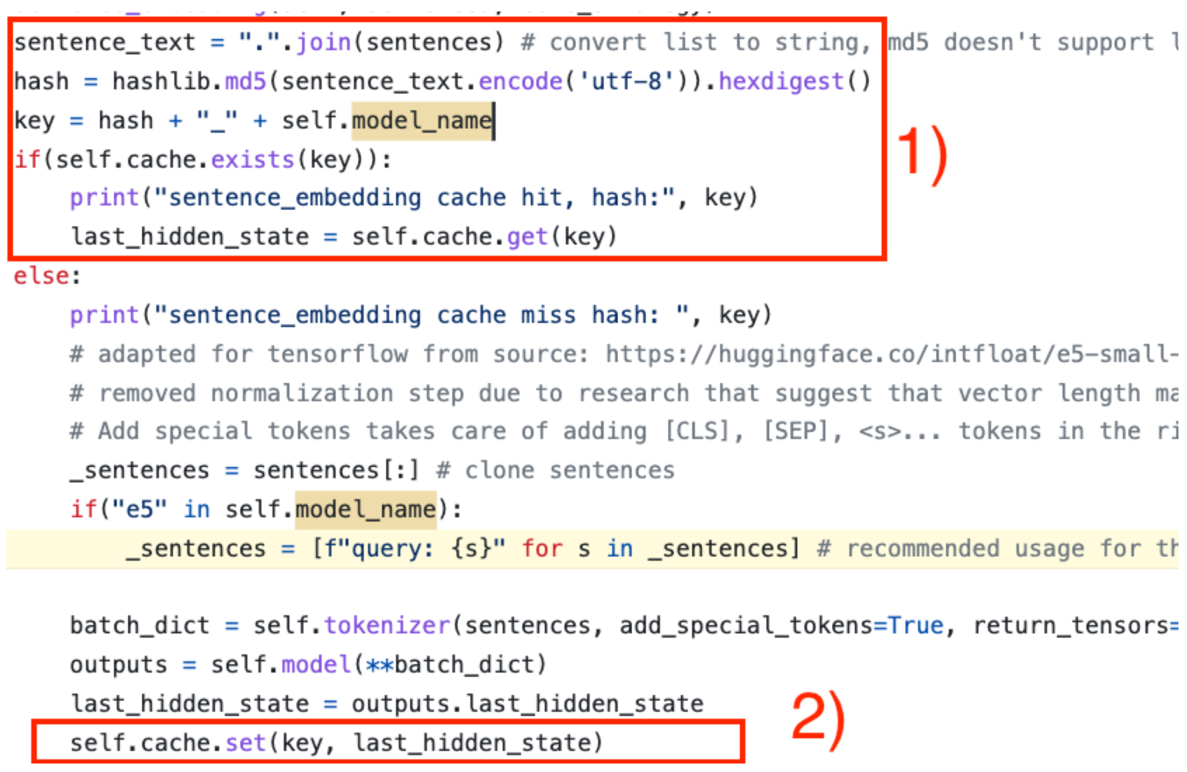
```
class TensorCache:
    pablo marino *
    def __init__(self, clear_cache):
        if clear_cache:
            self.clear()

    pablo marino
    def set(self, key, value):
        filename = "cache_files/" + key + ".tf"
        serialized_tensor = tf.io.serialize_tensor(value)
        tf.io.write_file(filename, serialized_tensor)
    pablo marino
    def get(self, key):
        filename = "cache_files/" + key + ".tf"
        serialized_tensor = tf.io.read_file(filename)
        return tf.io.parse_tensor(serialized_tensor, out_type=tf.float32)
```

In 1) my class has a parameter for cleaning the entire database, this is very necessary because data stored in the hard drive can take 140gb of space, which is useful for maintaining the cache between runs but sometimes I need to run my GA with a clean cache for testing or if something changes in the data(eg: a new dataset) I may run out of disk space if I don't get rid of old entries. I can turn on off this feature from the config files:



```
TEXT_COLUMN = 'review'
LABEL_COLUMN = 'sentiment'
GENERATIONS = 25
N_POPULATION = 3
PROB_MUTATION = 0.2
MAX_SAMPLE_SIZE_DS = 150
NEURONS_CHANGE_FACTOR = 0.8 # reduce neurons by 20% each layer
DS_PATH = 'datasets/IMDB_Dataset.csv'
LOG_FILE_DIR = 'logs'
CLEAR_CACHE = false
[FROZEN_PARAMETERS] # used for testing: freeze parameters to a specific value
    PARAMETER_NAME = "PARAMETER_VALUE_TO_FREEZE"
```

In 2) I serialize the tensor before storing it as a file and in 3) I convert the bytes read from the file back to a Tensor.
The TensorCache class gets consumed by the EmbeddingGenerator in the following way:



```
sentence_text = ".".join(sentences) # convert list to string, md5 doesn't support l
hash = hashlib.md5(sentence_text.encode('utf-8')).hexdigest()
key = hash + "_" + self.model_name
if(self.cache.exists(key)):
    print("sentence_embedding cache hit, hash:", key)
    last_hidden_state = self.cache.get(key)
else:
    print("sentence_embedding cache miss hash: ", key)
    # adapted for tensorflow from source: https://huggingface.co/intfloat/e5-small-
    # removed normalization step due to research that suggest that vector length ma
    # Add special tokens takes care of adding [CLS], [SEP], <s>... tokens in the ri
    _sentences = sentences[:] # clone sentences
    if("e5" in self.model_name):
        _sentences = [f"query: {s}" for s in _sentences] # recommended usage for th

    batch_dict = self.tokenizer(sentences, add_special_tokens=True, return_tensors=
    outputs = self.model(**batch_dict)
    last_hidden_state = outputs.last_hidden_state
    self.cache.set(key, last_hidden_state)
```

In 1) given a list of sentences I obtain the MD5 hash of the string with all the sentences and use that and the model name as the key for the cache. This is because every model has its own embeddings for a given set of sentences. Then I use the key to check whether that is stored in the cache and use the value from the cache if so. If the key is not in cache I calculate the last_hidden_state tensor from the model and add it to the cache(2).

Second Iteration (Milestone 3)
In this milestone I added support for all the text preprocessing techniques explained in the design section.

Evaluation

GA Parameter selection

According to [23] the most important parameter to optimize in a genetic algorithm is the mutation rate. So in this section I will test 3 different values to select the best mutation rate, which in the next section will be used for fully evaluating each dataset for 500 generations.

For each mutation rate I will run the GA 10 times and consider the average of all runs, this is important to reduce the noise that can arise from the stochastic nature of the genetic algorithm.
The base parameters for each experiment are:
Number of generations: 30 (the more the better, but this is as high I can go to perform 10 runs in a reasonable amount of time for each dataset)

Population: 100

I have defined the following metrics which I will compare for every mutation rate:

ACR: Average Convergence Rate, this is a measure of how fast the population's average fitness improves over generations. If it improves quickly, the GA is said to have a fast convergence rate. To calculate this I will average the differences in mean fitness scores between consecutive generations. If later generations have higher mean fitness then ACR is positive and the higher the better.

GTC: generations to converge, number of generations until GA reaches maximum fitness score of 1 or until reaching maximum number of generations, less is better.

AFL: Average Fitness on Last generation

Max fitness: highest fitness score reached in a full GA run(30 generations)

Easy dataset (binary classification):
Number of examples for training: 160, for testing: 40
*for this experiment all embeddings were precalculated and stored in the cache, ie: time doesn't include embedding generation

Mutation rate	Avg ACR	Avg GTC	Avg max fitness	Avg AFL	Avg time (sec)
0.1	**0.02047**	**2.4**	1	**0.98232**	**12175.4**
0.2	0.02007	2.8	1	0.97199	15624.7
0.3	0.0198	3	1	0.95637	15411.4

For details on how to replicate above experiment check [ER1]

Harder dataset (3 classes):
Number of examples for training: 320, for testing: 80
*for this experiment all embeddings were precalculated and stored in the cache, ie: time doesn't include embedding generation

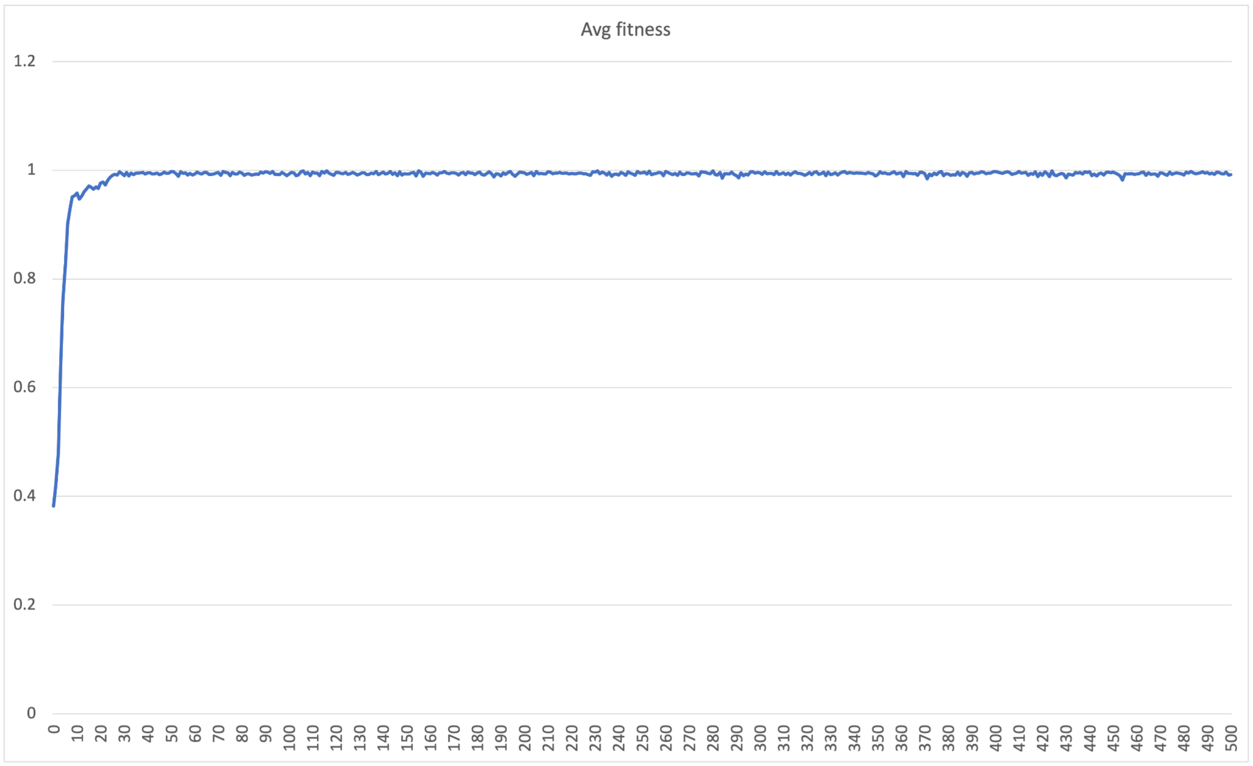
Mutation rate	Avg ACR	Avg GTC	Avg max fitness	Avg AFL	Avg time (sec)
0.1	**0.01646**	-	0.71392	**0.70163**	11676.2
0.2	0.01562	-	0.71355	0.67876	**9517.4**
0.3	0.01534	-	**0.74450**	0.67017	14644.7

For details on how to replicate above experiment check [ER2]

Evaluation using best parameters

Evaluation on easy dataset
Previous section showed that the best mutation rate is 0.1.
Using that information I ran the GA using the following parameters:
Mutation rate: 0.1
Generations: 500
Population: 100
Size of training set: 240
Size of testing set: 60

Average fitness across generations:

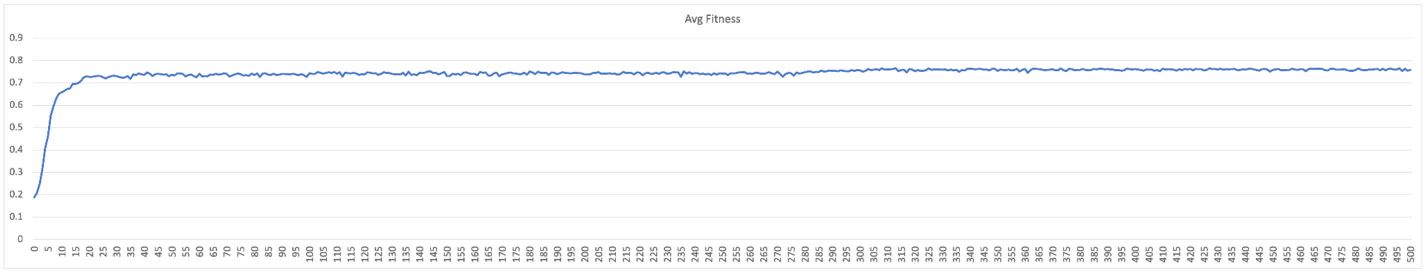


Total time taken to run this experiment: 7 Hours 17 Min. Note that for this experiment the embeddings were precalculated and stored in the cache, this reduced the duration considerably.

The above experiment shows how trivial this dataset is, it reaches 0.99 average fitness score at generation 26 and after that it keeps circling around that value for all future generations.

For details on how to replicate above evaluation check [ER3]

Evaluation on hard dataset
Previous section showed that the best mutation rate is 0.1.
Using that information I ran the GA using the following parameters:
Mutation rate: 0.1
Generations: 500
Population: 100
Size of training set: 400
Size of testing set: 100
Average fitness across generations:



Total time taken to run this experiment: 23 hours. For this experiment embeddings were not cached, GA had to calculate all embeddings from scratch, that's why it took considerably longer than the previous experiment.

An interesting fact about this dataset is that the most upvoted Kaggle notebook[24] that tries to solve this challenge reached 0.68 macro Average f1 score by training on 6956 examples while the top configuration found by my GA has 0.77 macro average F1, while only training on 400 examples(Note that the testing datasets differ).

For details on how to replicate above evaluation check [ER4]

Evaluation Takeaways
As can be seen in the above plots I have reached my goal of designing a GA that optimizes text classification pipelines, for both datasets the fitness score improves over time until reaching a maximum point where it stops improving and keeps circling around the best designs.

Limitations and considerations of my evaluation

- I have evaluated my system in trivial problems: binary classification and 3-class and have observed the GA reached the max performance very quickly, it would have been useful to evaluate the system in a hard classification problem that takes my project to the limit and where ideally I can see a longer curve of improvement where the GA performance doesn't plateau after 35 iterations.
- In the mutation rate search I only tried 3 values, it would be useful to test values between 0 and 0.1 for the mutation rate.

- GA has to run on small data otherwise it's too slow and my computer runs out of disk space. That means both training and validation sets are small and are not 100% representative of the real population.
- I'm not considering whether my GA is overfitting the pipeline design for the testing set, would be informative to run the best configuration on a hold out set to test if this is the case, TOPTP paper[6] talks about this issue and how they used cross validation to solve it
- GA optimization was ran for only 30 generations, unless GA converges fast like for the binary dataset to analyze impact of mutation rate properly I should get metrics for a greater number of generations since some mutation rates could work great in the beginning when fitness is low but once it reaches a certain point the GA could overshoot the global maximum because it needs to mutate in smaller steps. I may even need a dynamic mutation rate for some problems where mutation rates starts high and becomes smaller over time

Conclusion

I have designed and implemented a genetic algorithm that optimizes both the design of multiclass text classification pipelines and of the neural network architecture and considers state of the art pretrained transformer models for vectorization. In the evaluation section I tried different mutation rates and evaluated the system in 2 datasets: one with 2 classes and another one with 3, for both cases I have shown an improvement of the fitness across generations and both are able to run for 500 generations and evaluate thousands of solutions in a reasonable amount of time where the slowest experiment lasted 23 hours running locally in a home computer without the aid of a GPU.
My tool is a good starting point for every engineer who wants to design a text classifier in an automated fashion, it currently has one limitation which is that it requires a lot of free disk space to cache embeddings so in home computers is only able to run for small data, however this can be enough for finding candidate designs in small data and then manually testing them on the entire dataset.
Below is a list of ideas I consider would be worth exploring in the future for improving this project

Future work

- Be able to consume embedding models running in a GPU either locally or in the cloud to speed up embedding generation which is the bottleneck of the current system.
- Research ways to reduce the size of the embeddings cache, it currently takes above 100gb of disk space of cache to run an experiment on 400 examples. Some ideas are to compress the embeddings or to store only the final sentence embeddings rather than the embeddings for every single word and get rid of the concat technique for generating sentence embeddings since it creates massive sentence embeddings which don't perform well.
- Compare accuracy against random search and other auto ML frameworks to see where this framework stands in comparison with others in the field.

- Modify framework to be able to run GA in multi thread or multiprocess. One idea I would like to explore is to separate the process in 2 phases: first precalculate all the embeddings and store them in cache using a single thread and after run GA in multiple threads without loading the Transformer models to save RAM since embeddings are retrieved from disk.
- add a compute budget to penalize expensive/slow pipelines. Once GA finds solutions that reach top accuracy, slow solutions get ranked the same as fast solutions. fast and accurate solutions should be prioritized in the fitness function, one idea could be to take into account the number of parameters of the transformer model, ie: less parameters and good accuracy has higher fitness that same accuracy but bigger model.
- Make framework more easily extensible so that adding support for new Transformer models and architectures is trivial and build a docker image to run the GA on it to allow for easy deployment and replication of experiments
- Like TOPTP paper[6] perform cross validation to calculate fitness to prevent GA from overfitting the pipeline design to a single testing set.

Appendix

Tables and figures

[fig 1]

gen	nevals	avg	std	min	max
0	3	0.666667	0.0981307	0.533333	0.766667
1	0	0.766667	0	0.766667	0.766667
2	2	0.766667	0	0.766667	0.766667
3	0	0.766667	0	0.766667	0.766667
4	1	0.766667	0	0.766667	0.766667
5	0	0.766667	0	0.766667	0.766667
6	2	0.766667	0	0.766667	0.766667
7	2	0.766667	0	0.766667	0.766667
8	2	0.766667	0	0.766667	0.766667
9	2	0.766667	0	0.766667	0.766667
10	2	0.8	0.0471405	0.766667	0.866667
11	2	0.833333	0.0471405	0.766667	0.866667
12	3	0.866667	0	0.866667	0.866667
13	2	0.866667	0	0.866667	0.866667

14	2	0.866667	0	0.866667	0.866667
15	2	0.866667	0	0.866667	0.866667
16	2	0.866667	0	0.866667	0.866667
17	2	0.866667	0	0.866667	0.866667
18	1	0.833333	0.0471405	0.766667	0.866667
19	0	0.833333	0.0471405	0.766667	0.866667
20	3	0.866667	0	0.866667	0.866667
21	2	0.866667	0	0.866667	0.866667
22	2	0.866667	0	0.866667	0.866667
23	3	0.866667	0	0.866667	0.866667
24	0	0.866667	0	0.866667	0.866667
25	2	0.866667	0	0.866667	0.866667

[fig 2]

```
toml==0.10.2
transformers==4.30.2
scikit-learn==1.2.2
pandas==2.0.2
setuptools==57.5.0
deap==1.3.3
tensorflow==2.12.0
spacy==3.6.0
nltk==3.8.1
```

[fig 3]

0- Ensure you have Anaconda installed, if not either install it or adapt the commands below to the package management system of your choice.

1- Create a conda environment with python V3.9 and pip using the following command:

```
conda create -n YOUR_ENV_NAME_HERE python=3.9 pip
```

2-With the environment active and inside the project directory install the required modules by running:

```
pip install -r requirements.txt --use-deprecated=legacy-resolver
```

3- Download Spacy model with command:

```
python -m spacy download en_core_web_sm
```

Sprints

Sprint 1
Date: June 12-18
Goal: Setup an environment for my work with all required modules and implement a mock GA using DEAP module with only one step: NAS(only number of layers parameter) that has different options for topology but returns random predictions regardless of parameters, a fitness score that returns a random number and be able to run the GA for a text classification dataset without crashing.
Tasks:

- Setup python environment with required modules
- Setup GA with only search space for NAS
- Setup random implementation of NAS and fitness score(doesn't need validation set)
- Run the GA without crashing

Sprint 2
Date: June 19-25
Goal: GA can read a (hardcoded) binary classification ds, vectorize with bert and average vectors to generate sentence embedding
Tasks:

- add a step to read a ds and take out a validation set
- Add vectorizer step to GA with bert base as only option
- Add averaging of BERT vectors step

Sprint 3(milestone 1 deliverable)
Date: June 26- July 2
Deliverable: Milestone 1
Goal: Have working prototype: implement proper NAS component to support only no of layers prop.
Tasks:

- Implement NAS properly only with support for no. of layers.
- Implement simple fitness function which uses validation set and measures: accuracy of predictions
- Ensure GA runs end to end and tries different no. of layers parameters

Break
Date: July 3 - 17

Goal: Pause development of this project to focus on preliminary report and midterm projects for other modules

Sprint 4
Date: July 17 - 23
Goal: Implement an interface to the GA so I can run the tool as a script and pass different arguments, Setup a proper fitness function
Tasks:

- Setup GA as script
- Implement proper fitness function, use macro avg f1 score.

Sprint 5
Date: July 24 - 30
Goal: Add evaluation metrics that track progress across generations and print them to terminal and also store them in a CSV.
Tasks:

- Print time, generation number, parameters and fitness score to console.
- Create a CSV file with datetime contained in the filename upon start of the GA and add a new row to it with date, generation number, parameters and fitness score to it.

Review and planification of next springs: when reaching this point should already have a working prototype, assess the current work, previous sprints and plan the next springs accordingly.

Experiment replication

[ER1] Dataset 1 parameter optimization

Run with Python commands:
```
python GA_optimizer.py --config optimization_ds1_exp1 --times 10
python GA_optimizer.py --config optimization_ds1_exp2 --times 10
python GA_optimizer.py --config optimization_ds1_exp3 --times 10
```
Location of stats files generated:
logs/optimization_ds1_exp1, logs/optimization_ds1_exp2, logs/optimization_ds1_exp3
Notebook used to generate statistics table from log files:
analysis.ipynb

[ER2] Dataset2 parameter optimization

Run with Python commands:
```
python GA_optimizer.py --config optimization_ds2_exp1 --times 10
python GA_optimizer.py --config optimization_ds2_exp2 --times 10
python GA_optimizer.py --config optimization_ds2_exp3 --times 10
```
Location of stats files generated:
logs/optimization_ds2_exp1, logs/optimization_ds2_exp2, logs/optimization_ds2_exp3
Notebook used to generate statistics table from log files:
analysis.ipynb

[ER3] Dataset1 evaluation

Run with Python command:
```
python main.py --config final_evaluation_ds1_V3
```
Location of stats file generated:
logs/log_20230810-153127.csv
How to generate plot:
Load the CSV stats file in to Excel and plot column gen(generation) VS avg(average fitness)

[ER4] Dataset2 evaluation

Run with Python command:
```
python main.py --config final_evaluation_ds2_V3
```
Location of stats file generated:
logs/log_20230823-190726.csv
How to generate plot:
Load the CSV stats file in to Excel and plot column gen(generation) VS avg(average fitness)

References

[1] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, Frank Hutter. 2023. Neural Architecture Search: Insights from 1000 Papers. https://arxiv.org/abs/2301.08727

[2] Esteban Real, Alok Aggarwal, Yanping Huang, Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. https://arxiv.org/abs/1802.01548

[3] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, Frank Hutter. 2023. Neural Architecture Search: Insights from 1000 Papers. https://arxiv.org/abs/2301.08727

[4] Hugging Face. 2023. AutoTrain. https://huggingface.co/autotrain

[5] Github. 2023. TPOT. https://github.com/epistasislab/tpot

[6] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd & Jason H. Moore. 2016. Automating Biomedical Data Science Through Tree-Based Pipeline Optimization. in Lecture Notes in Computer Science book series (LNTCS,volume 9597), Springer, pages 123–137.

[7] Trang T Le, Weixuan Fu, Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector, in Bioinformatics, Volume 36, Issue 1, January 2020, Pages 250–256

[8] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, Christian Gagné. 2012. DEAP: evolutionary algorithms made easy. https://dl.acm.org/doi/10.5555/2503308.2503311

[9] Github. 2023. AutoPytorch. https://automl.github.io/Auto-PyTorch

[10] Lucas Zimmer, Marius Lindauer, Frank Hutter. 2020. Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL. https://arxiv.org/abs/2006.13799

[11] Jessica Mégane, Nuno Lourenço, Penousal Machado. 2021. Probabilistic Grammatical Evolution. https://arxiv.org/abs/2103.08389

[12] Github. 2023. AutoGoal. https://github.com/autogoal/autogoal

[13] Suilan Estevez-Velarde, Yoan Gutiérrez, Andres Montoyo, and Yudivián Almeida Cruz. 2020. Automatic Discovery of Heterogeneous Machine Learning Pipelines: An Application to Natural Language Processing. In Proceedings of the 28th International Conference on Computational Linguistics, Barcelona, Spain (Online). International Committee on Computational Linguistics, pages 3558–3568

[14] Tithi Sreemany. September 3, 2021. Analytics Vidhya. Essential Text Pre-processing Techniques for NLP!
https://www.analyticsvidhya.com/blog/2021/09/essential-text-pre-processing-techniques-for-nlp/,

[15] Dr. S. Vijayarani, Ms. J. Ilamathi, Ms. Nithya. 2015. Preprocessing Techniques for Text Mining - An Overview.
https://www.researchgate.net/profile/Vijayarani-Mohan/publication/339529230_Preprocessing_Techniques_for_Text_Mining_-_An_Overview/links/5e57a0f7299bf1bdb83e7505/Preprocessing-Techniques-for-Text-Mining-An-Overview.pdf

[16] Hugging Face. 2023. Mteb Leaderboard. https://huggingface.co/spaces/mteb/leaderboard

[17] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In The 49th Annual

Meeting of the Association for Computational Linguistics. (ACL) Portland, Oregon, USA. June 2011. Pages 142–150

[18] Scikit Learn. 2023. Sklearn.metrics.f1_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

[19] Hugging Face. 2023. E5-small-v2. https://huggingface.co/intfloat/e5-small-v2#usage

[20] Hugging Face. 2023. E5-base-v2. https://huggingface.co/intfloat/e5-base-v2#usage

[21] Hugging Face. 2023. E5-large-v2. https://huggingface.co/intfloat/e5-large-v2#usage

[22] Kaggle. 2021. Physics vs Chemistry vs Biology. https://www.kaggle.com/datasets/vivmankar/physics-vs-chemistry-vs-biology

[23] K. L. Mills, J. J. Filliben and A. L. Haines. 2015. "Determining Relative Importance and Effective Settings for Genetic Algorithm Control Parameters," in Evolutionary Computation, vol. 23, no. 2, pp. 309-342, June 2015.

[24] Kaggle. 2022. Basic-NLP||Text classification. https://www.kaggle.com/code/aminaromdhani/basic-nlp-text-classification