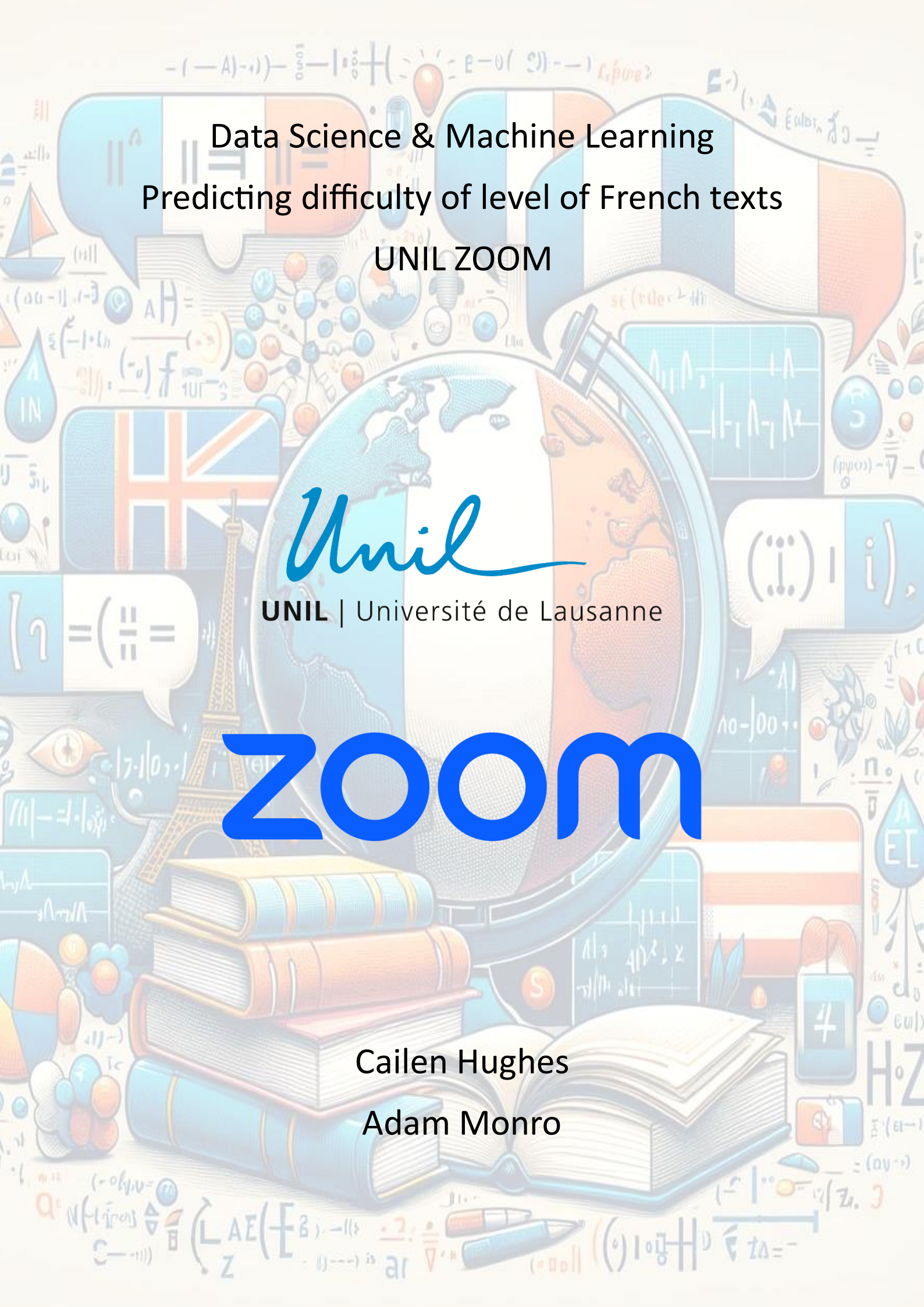
Data Science & Machine Learning

Predicting difficulty of level of French texts

UNIL ZOOM



UNIL | Université de Lausanne

ZOOM

Cailen Hughes

Adam Monro

Table of Contents

1. Introduction

In this report, we will go over the different models used for our project in Michalis Vlachos' class on Data Science & Machine Learning. The objective of this project was to develop a model tailored for English speakers learning French. The key challenge we addressed was finding French texts that match a learner's proficiency level, ranging from A1 to C2.

Our solution was to build an algorithm capable of assessing the difficulty level of French written texts. This model proved especially useful in educational contexts, such as in a recommendation system to suggest news articles appropriate to a learner's current stage of French proficiency.

For instance, presenting a B2 level text to a learner at the A1 level would have been too complex, potentially hindering their learning. Our model aimed to recommend texts that were mostly within the learner's comprehension but included some new vocabulary or grammar structures for gradual learning.

Our goal was to streamline the language learning process, providing resources that were both engaging and suitable for each learner's specific stage. This personalized approach to language learning is what we believed would make our project impactful in the field of language education.

We were provided with data sets such as a training data set, a test data set and a dataset that was an example of a submission data set so that the data could be read correctly.

All our results were submitted to the Kaggle competition to see the accuracy of the submission data set. There was also a leaderboard with all the teams so that we could compare our progress to the rest of the participants.

We will now move on to the different models we used and how they worked.

2. Models

Throughout this project we used multiple techniques and models to try and improve our accuracy. We started off by using **Logistic Regression** to train our model, then we tried with **kNN** (k-Nearest Neighbors Algorithm). Following this we used **Random forests** to try and boost our accuracy before moving on to the **Decision Tree** method. Finally, we used different **HuggingFace** pre-existing models to up our accuracy.

Here are the explanations of the codes for each method.

2.1 Logistic Regression

The Logistic Regression code can be found on our GitHub in the code folder: 0.456-Logistic_Regression.py

(https://raw.githubusercontent.com/AdamMonroUnil/DSML/main/Code/0.456-Logistic_Regression.py)

This code uses Logistic Regression. It is designed to classify sentences into different 'difficulty' levels. The achieved accuracy of 0.456 suggests that while the model has some predictive power, it is not highly accurate. Let's examine the code and explore potential reasons for this moderate level of accuracy.

Code Overview:

1. Data Loading and Splitting: The dataset is loaded into a DataFrame and split into features (X - sentences) and the target variable (y - difficulty levels). This is further split into training and validation sets.
2. Feature Extraction: TfidfVectorizer is used to convert text data into TF-IDF features, which is a common approach to handling text data in machine learning.
3. Model Setup: A Logistic Regression model with increased maximum iterations (max_iter=1000) is employed. This model is a popular choice for binary and multiclass classification problems.
4. Pipeline Creation: A pipeline combining the TF-IDF vectorizer and Logistic Regression model is created, simplifying the process of data transformation and model application.
5. Model Training and Prediction: The model is trained on the training set, and predictions are made on the validation set.
6. Evaluation: Various performance metrics (accuracy, precision, recall, F1-score) are calculated, and a confusion matrix is generated to assess the model's performance.
7. Prediction on New Data: Finally, the model is used to predict the difficulty of new, unlabelled data, which is then saved for submission.

Potential Reasons for Moderate Accuracy (0.456):

1. Model Simplicity: Logistic Regression, while effective in many scenarios, might be too simplistic for complex text classification tasks, especially when dealing with nuanced language data.
2. Feature Representation Limitations: TF-IDF is a linear method that might not capture all the intricacies of natural language, such as context, word order, and semantic meanings.
3. Imbalanced Dataset: If the dataset is imbalanced with uneven distribution of 'difficulty' levels, the model may not perform well, particularly for underrepresented classes.
4. Hyperparameter Tuning: The default parameters of Logistic Regression (except for max_iter) are used. Further tuning of hyperparameters might improve the model's accuracy.
5. Data Preprocessing: The effectiveness of the model also heavily depends on the quality of data preprocessing. Essential steps like text normalization, handling of stopwords, and stemming could enhance model performance.
6. Class Overlap: If there is a significant overlap between classes (i.e., different difficulty levels are not distinctly separable based on the text), the model may struggle to accurately classify the sentences.

Improving Model Performance:

a. Advanced Models: Experiment with more complex models like Support Vector Machines, Random Forests, or neural networks, which might capture the complexity of the task better.

b. Enhanced Feature Engineering: Use more sophisticated text representation methods, like word embeddings or contextual embeddings from models like BERT.
c. Hyperparameter Optimization: Employ techniques like GridSearchCV or RandomizedSearchCV to find the best set of hyperparameters for the Logistic Regression model.
d. Balanced Dataset: Ensure that the dataset is balanced or use techniques like class weighting or oversampling to mitigate the effects of class imbalance.
e. In-depth Preprocessing: Improve text preprocessing steps, such as advanced tokenization, lemmatization, and exploration of different n-gram ranges.

2.2 kNN ((k-Nearest Neighbors Algorithm)

The kNN code can be found on our GitHub in the code folder: 0.32-kNN.py (https://raw.githubusercontent.com/AdamMonroUnil/DSML/main/Code/0.32-kNN.py)

This code uses a k-Nearest Neighbors (kNN) classifier in a machine learning pipeline. The aim is to classify sentences into different 'difficulty' levels. Despite following standard machine learning practices, the model achieved a relatively low accuracy of 0.32. Let's analyze the code and identify potential reasons for this performance.

Code Overview:

1. Data Loading: The dataset is loaded into a Pandas DataFrame from a CSV file. It contains features (X - sentences) and the target variable (y - difficulty levels).
2. Data Splitting: The dataset is split into training and validation sets. This allows for evaluating the model's performance on unseen data.
3. Feature Extraction: The TfidfVectorizer with ngram_range=(1, 2) is used to convert text data into TF-IDF features. This represents text data numerically for the KNN model.
4. Model Definition: A K-Nearest Neighbors classifier with n_neighbors=5 is used. KNN is a simple yet effective algorithm for classification tasks.
5. Pipeline Creation: A pipeline is created combining the TF-IDF vectorizer and KNN classifier, streamlining the process of transforming data and making predictions.
6. Model Training: The pipeline is trained using the training dataset.
7. Prediction and Evaluation: Predictions are made on the validation set, and performance metrics (accuracy, precision, recall, F1-score) are calculated. A confusion matrix is also generated.
8. Application to New Data: The model is applied to a new dataset, and the predictions are prepared for submission.

Potential Reasons for Low Accuracy (0.32):

1. Model Choice: KNN is a basic algorithm that can struggle with high-dimensional data, like text data transformed into TF-IDF features. It relies on distance metrics that may not be very effective in high-dimensional spaces.
2. Feature Representation: TF-IDF is a simple linear method for text representation. It might not capture the complexities and semantics of the text adequately, especially for classification tasks.
3. Imbalanced Data: If the dataset is imbalanced, with some 'difficulty' levels having much more data than others, it can lead to poor classification performance, particularly for the minority classes.
4. Parameter Settings: The choice of n_neighbors=5 and the default settings of TfidfVectorizer might not be optimal. Parameter tuning could improve model performance.

5. Data Quality and Preprocessing: The quality of data and the preprocessing steps (like normalization, handling of stopwords, stemming, etc.) can significantly impact the model's effectiveness.
6. Algorithm Limitations: KNN's performance heavily depends on the choice of distance measure and the number of neighbors. It might not be the best-suited algorithm for complex text classification tasks.

Improving the Model:

a. Experiment with Different Models: Trying different algorithms like Random Forest, Support Vector Machines, or neural networks might yield better results.
b. Advanced Text Processing: Use more sophisticated text processing techniques, such as word embeddings (e.g., Word2Vec, GloVe).
c. Hyperparameter Tuning: Optimize the parameters of both the KNN classifier and the TF-IDF vectorizer.
d. Handling Imbalanced Data: Techniques like oversampling minority classes, undersampling majority classes, or using class weights can help.
e. Feature Engineering: Experimenting with different features extracted from the text might provide more insight for the classification task.

2.3 Random Forests

The Random Forests code can be found on our GitHub in the code folder: 0.363-RandomForest.py (https://raw.githubusercontent.com/AdamMonroUnil/DSML/main/Code/0.363-RandomForest.py)

This code uses a RandomForestClassifier within a GridSearchCV framework. The aim is to classify sentences into various 'difficulty' levels. Despite the comprehensive approach, the model achieved a relatively low accuracy of 0.363. Let's dissect the code to understand its components and discuss potential reasons for the low accuracy.

Code Overview:

1. Data Loading and Splitting: The dataset is loaded from a CSV file into a DataFrame, with features (X - sentences) and the target variable (y - difficulty levels). It's split into training and validation sets.
2. Feature Extraction: The TfidfVectorizer converts text data into TF-IDF features, which are numerical representations suitable for the RandomForestClassifier.
3. Model and Pipeline Setup: A RandomForestClassifier is set up, and a pipeline is created combining the TF-IDF vectorizer and the classifier.
4. Hyperparameter Tuning: GridSearchCV is used for hyperparameter tuning, searching over a predefined grid of hyperparameters (n_estimators and max_depth for RandomForestClassifier).
5. Model Training and Evaluation: The model is trained and evaluated on the validation set, and various performance metrics are calculated.
6. Prediction on New Data: The trained model is then used to predict the 'difficulty' of new, unlabelled data, which is saved for submission.

Potential Reasons for Low Accuracy (0.363):

1. Model Complexity: RandomForestClassifier, while powerful, may struggle with the high-dimensional, sparse data typically output by TF-IDF vectorization, especially if the right hyperparameters are not chosen.

2. Feature Representation: TF-IDF is a linear and simple representation of text. It may not capture all the complexities and semantic relationships in the text, which are crucial for accurate classification.
3. Imbalanced Dataset: If the 'difficulty' classes in the dataset are imbalanced, the model may be biased towards the majority class, reducing overall accuracy.
4. Hyperparameter Selection: The grid of hyperparameters in GridSearchCV may not include the optimal settings for the RandomForestClassifier. The choice of hyperparameters significantly impacts model performance.
5. Data Quality and Preprocessing: If the dataset contains noise or is not preprocessed adequately (e.g., removing stopwords, stemming), the model's performance can be affected.
6. Scoring Metric in GridSearchCV: The GridSearchCV is configured to use 'precision_weighted' as the scoring metric. While precision is important, optimizing solely for it might not always yield the highest accuracy.

Improving Model Performance:

a. Experiment with Different Models: Trying different algorithms, including more advanced ones like Gradient Boosting Machines or neural networks, might yield better results.
b. Enhanced Text Processing: Employ more sophisticated text processing techniques, such as word embeddings (e.g., Word2Vec, GloVe), which might capture textual information more effectively.
c. Hyperparameter Optimization: Expanding the grid search or using randomized search with a wider range of hyperparameters might help find a better configuration.
d. Handling Imbalanced Data: Techniques such as SMOTE for oversampling, class weights in the classifier, or focused evaluation metrics for imbalanced data can be considered.
e. Feature Engineering: Experimenting with different features extracted from the text, or including additional relevant features, could improve the model's predictive power.

2.4 Decision Tree

The Random Forests code can be found on our GitHub in the code folder: 0.276-DecisionTree.py (https://raw.githubusercontent.com/AdamMonroUnil/DSML/main/Code/0.276-DecisionTree.py)

This code uses a machine learning pipeline. The goal is to classify text sentences into different 'difficulty' levels. The pipeline includes data loading, preprocessing, feature extraction, model training, prediction, and evaluation. Let's break down the key components and discuss potential reasons for the observed low accuracy (0.276).

Code Breakdown:

1. Data Loading: The data is loaded from a CSV file into a Pandas DataFrame. The dataset is split into features (X) which are the sentences, and the target (y) which is the 'difficulty' level.
2. Data Splitting: The data is split into training and validation sets. This is important for evaluating the model on unseen data.
3. Feature Extraction: The TfidfVectorizer is used to transform the text data into a more manageable form for the machine learning model. It converts text to a matrix of TF-IDF features.
4. Model Definition: A Decision Tree Classifier is used for classification. Decision trees are simple but can be prone to overfitting.
5. Pipeline Creation: A pipeline combining the TF-IDF transformer, and the Decision Tree model is created. This simplifies the process of transforming and predicting.

6. Model Training: The model is trained on the training dataset.
7. Prediction and Evaluation: The model makes predictions on the validation set, and various metrics (accuracy, precision, recall, F1-score) are calculated. A confusion matrix is also printed.
8. Application to New Data: The trained model is then used to predict 'difficulty' for a new, unlabelled dataset, and the results are saved to a CSV file.

Potential Reasons for Low Accuracy (0.276):

1. Model Complexity: Decision Trees are quite basic and can struggle with complex text classification tasks. They are prone to overfitting, especially if the tree is very deep.
2. Feature Extraction: TF-IDF is a simple way to convert text to features, but it might not capture all the nuances needed for accurate classification. It treats each word or phrase separately and doesn't account for the context or the order of words.
3. Imbalanced Dataset: If the dataset is imbalanced (some classes have many more examples than others), it can lead to poor performance, especially for minority classes.
4. Parameter Tuning: The Decision Tree classifier is used with default parameters. Tuning these parameters (like max_depth, min_samples_split) could potentially improve performance.
5. Data Quality and Preprocessing: The quality of the data and how it's preprocessed can significantly affect the model's performance. If the dataset contains a lot of noise, or if important preprocessing steps (like stemming, lemmatization, removing stop words) are skipped, the model may not perform well.
6. Model Choice: Decision Trees might not be the best model for this task. Other models, like Random Forests, Gradient Boosted Machines, or even neural network-based models, might be more effective for text classification.

To improve the model's accuracy, we might consider experimenting with different models, tuning the hyperparameters, improving the feature extraction process, and ensuring the dataset is well-preprocessed and balanced.

2.5 Camembert (HuggigFace)

The best camembert code can be found on our GitHub in the code folder: 0.606_ZOOM.py (https://raw.githubusercontent.com/AdamMonroUnil/DSML/main/Code/0.606_ZOOM.py)

This code is a more advanced text classification pipeline utilizing the Camembert model, which is a transformer-based model trained specifically for French language tasks. The accuracy of this model (0.606) is notably higher than previous models you mentioned. Let's explore the key components of this pipeline and why it likely performed better.

Key Components:

1. Installation of Necessary Libraries: accelerate, nlpaug, and sentencepiece are installed. These libraries are essential for efficient computation and text augmentation.
2. Environment Setup and Data Preparation: The script utilizes GPU for computation (if available), enhancing processing speed and efficiency. The training data is loaded, and labels are encoded using LabelEncoder.
3. Tokenizer: The CamembertTokenizer is used for tokenizing the text data. Tokenization is crucial in preparing text data for transformer models.
4. Dataset Preparation: A custom dataset class is created to handle the tokenized data.

5. Dataset Split: The dataset is split into training and validation sets to train the model and evaluate its performance.
6. Model Loading: CamembertForSequenceClassification is loaded with the number of labels corresponding to the classification task.
7. Training Setup: TrainingArguments are defined for training, which includes parameters like batch size, number of epochs, warmup steps, etc.
8. Model Training: The model is trained using the Trainer class from the Hugging Face's Transformers library.
9. Test Data Preparation and Prediction: After training, the model is used to predict the difficulty level on a new set of unlabelled data.

Reasons for Improved Performance (Accuracy: 0.606):

1. Advanced Model Architecture: Camembert is a transformer-based model which has shown superior performance in various NLP tasks compared to traditional machine learning models. It captures contextual information better due to its attention mechanism.
2. Specific to French Language: Given that Camembert is specifically trained on French data, it is more adept at understanding the nuances and context of the French language, which likely contributes to higher accuracy.
3. Effective Tokenization: The use of a tokenizer that's aligned with the model (CamembertTokenizer) ensures that the text data is properly prepared and tokenized, which is crucial for transformer models.
4. GPU Utilization: The use of GPU for training and prediction significantly speeds up the process, allowing for more efficient handling of complex models and large datasets.
5. Fine-Tuning: The model is fine-tuned on the specific task of classifying text into different 'difficulty' levels, which allows it to adapt better to the nuances of the specific dataset.
6. Hyperparameter Tuning: The use of specific training arguments and hyperparameters that are suitable for the dataset and task at hand could have contributed to the model's improved performance.

Summary: The higher accuracy of this model is primarily due to the advanced capabilities of the Camembert transformer model, its suitability for the French language, effective data preprocessing and tokenization, and efficient training procedures facilitated using GPU. The combination of these factors makes transformer models like Camembert particularly effective for complex NLP tasks.

3. Summary of Models

Having started off with the Logistic Regression model, we managed to up its accuracy to 0.456 (45.6%) by fine tuning and testing the different parameters. It started off at around 0.433 of accuracy. We then tried using **kNN**, **Random Forest** and **Decision Tree**.

It was at this moment that we found out about huggingface.co in class. Huggingface allowed us to have access to many pre-trained models for language difficulty prediction.

We started off by using the RoBERTta model and through trial and error we arrived at around 50% accuracy. We then moved on to the Bert-multilingual (bert-base-multilingual-cased) model and

achieved, through fine tuning the hyperparameters, also around 50-55% accuracy. We later discovered the 'classic' Bert model (bert-base-uncased) that gave us slightly less precise data as it is not a French trained model. Following this, we tried to find a model that was solely based off French texts and found the Bert-base-french-europeana-cased model. This model didn't achieve the accuracy we were expecting.

The best model we found was CamemBert (camembert-base). This model is based on the RoBERTa model and achieved the best accuracy for us.

4. Optimizing our model

We managed to improve the accuracy by fine tuning some parameters. In the training arguments for epochs, we found that 3 epochs were best for our model and data. Regarding warmup steps and weight decay, around 10% of the steps suited our model the best, in this case 1000 and for weight decay we used 0.01. For the batch sizes, for the training side we found that a batch size of 2 was best and for the evaluation side we found that the optimal was 64.

Another important factor in improving our accuracy was to trial and error with the data set. We started of by generating some new data and sentences in the training data set. This didn't get us any better results, so we went back to the original data set. We then found that duplicating the training data was useful to improve results. However, if we duplicated it too much, it doesn't work as well, and the accuracy goes down.

To save time on this project it was vital that we used a GPU to run the algorithms. To do so we used Google Colab and ran our assignments on the T4 GPU that is free. To do so it is important to write a small script at the beginning of the code to connect to it correctly.

5. Streamlit

To finish this project we were asked to create an interface using Streamlit. To do so we downloaded our best model in a .pth file and our label encoder in a .joblib file. To use these du to the size of them we stored them on a Google Drive as they could not be uploaded to GitHub.

Streamlit allows us to create an app and an interface easily. Our app works in the following manner: the code uses the model and label encoder to determine the difficulty level of a French text that is typed into the application by the user. The model will then predict the difficulty and return the predicted difficulty in the form of a message. The app also has the possibility to upload a YouTube URL to a video. Our algorithm then downloads the captions and predicts the difficulty of them. It then returns a message with the predicted difficulty based off the captions. If the captions are not found or are not in French, another message informing the user of this appears.

To create this application, we imagined a scenario for the user. Let's imagine the user want to learn French, they can upload video and find out if they have the appropriate difficulty level before watching it. Also, they can type in a text to check whether the difficulty is appropriate to their level.

6. Conclusion

Throughout this report, we went through all our models used in detail. We explained the different methods and how they could have gotten better accuracies and be improved. We also explained how we got our best results by fine tuning the hyper parameters. To do better we could have used OpenAI to predict the difficulty of French texts but opted out of it as it is not a free service.

Doing this project was an interesting challenge for us, having never approached machine learning to predict any results by training models, we learnt a lot of useful information.

You can find all our models on our GitHub, as well as our data used and documents explaining how to use our app. There is also a file that documents the results of each model used throughout the project.