

Introduction to AI Scoring in Python

Christopher Ormerod
Cambium Assessment Inc.

Introduction to AI Scoring in Python

Course Outline

This hands-on workshop introduces participants to automated scoring using Python and machine learning techniques. Course materials include a code repository accessible through Google Colab notebooks.



By completing this workshop, participants will be able to:

Master Core Concepts: Grasp essential machine learning principles, particularly text classification methods and language models used in automated scoring applications.

Develop Technical Skills: Install and effectively use Python libraries essential for machine learning implementation.

Navigate the complete machine learning workflow: loading data, saving models, training algorithms, and calling text classifiers

Apply Standard AES methods: Implement a range of text classification approaches, from traditional frequency-based methods (such as TF-IDF and n-grams) to modern fine-tuned language models.

Apply Advanced AES methods: Implement Bayesian hyperparameter tuning, regression based modeling, and applications of generative models.

Colab Notebook

Use a web browser to navigate to

https://github.com/christopherormerod/AI-AES-Colab/blob/main/AI_Scoring_with_Python.ipynb



Sections

1. Introduction to Python

- Variables
- Data Types
- Control Structures
- Functions
- Libraries

2. Machine Learning

- Datasets
- Logistic Regression
- Random Forest
- Naive Bayes

Sections

3. Automated Essay Scoring

- Using Words as Features
- Singular Value Decomposition
- Hand-crafted Features
- Document Embeddings
- Fine-tuned Large Language Models

4. Advanced Techniques

- Hyperparameter Tuning
- Regression-based scoring
- Generative Models
- Scoring Using Generative Models

Introduction to Python

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language. Van Rossum's goal was to create a language that emphasized code readability and a simple, clean syntax, famously using indentation rather than braces to define code blocks. He named the project after the British comedy troupe Monty Python. The first public release, version 0.9.0, appeared in February 1991, introducing key features like exception handling, functions, and the core data types of list, dict, str, and others. It is now the most popular language for machine learning.



Python Basics

There are several basics to learn:



Variables: How to store information (like numbers, text, or true/false values) in named containers.

Data Types: Understanding the different kinds of information you can store, such as integers, floating-point numbers, strings (text), and booleans.

Operators: The symbols that perform actions, like + for addition, == for comparison, and && (or and) for logical operations.

Control Flow: This is how you direct the “flow” of your program.

Conditionals (if/else statements): Making decisions and running code only if a certain condition is true.

Loops (for/while loops): Repeating a block of code multiple times.

Functions (or Methods): Bundling lines of code into a reusable block that you can “call” by name to perform a specific task.


Data Structures: Simple ways to group data together, most commonly with Arrays/Lists (an ordered collection) and Hashmaps/Dictionaries (a collection of key-value pairs).

Variables, Types, and Control Structures

In all programming, variables, data types, and control structures are the fundamental building blocks you'll use to write any script or application. Together, they allow you to store information, understand what kind of information you have, and make decisions based on it.



Variables: The Data Containers

Think of a variable as a labeled box  where you can store a piece of information. It has two main parts: a name (the label on the box) and a value (the contents inside). You can change the value, which is why it's called a “variable.”

```
# 'name' is the variable, "Alice" is the value.  
name = "Alice"
```

```
# 'age' is the variable, 30 is the value.  
age = 30
```

```
# The value can be updated.  
age = 31
```

Python Basics

Data Types: The Kind of Information



A data type defines what kind of value a variable can hold. This is important because the type determines what operations you can perform on it (e.g., you can do math on numbers, but not on text). Python automatically detects the data type for you.

Here are the most common basic types:

```
# String
name = "Alice"
# Integer
age = 30
# Float
height = 1.99
```

If you are unsure, you can always ask for the type of a variable using the `type` function. You can also determine what functions and variables the variable contains by using `dir`.

```
print(type("string"))
dir("string")
```

Python Basics

Control Structures: The Flow of Logic

Control structures are like a recipe's instructions 📖 or a GPS giving directions; they direct the flow and order of your code's execution. They let your program make decisions and repeat actions.

The two main types are:

Conditionals (if/elif/else)

Conditionals let your code make choices. They execute a block of code if a certain condition is true.

Loops (for/while)

Loops are used to repeat a block of code multiple times. A for loop repeats an action for each item in a sequence (like a list). A while loop repeats an action as long as a certain condition remains true.

```
for x in range(100):  
    if x % 10 == 0:  
        print(x)
```

One of the advantages of Python is that the for loop iterates through objects like lists, so that the following is equivalent:

```
L = [2,3,5,7,11,13,17,19]  
for i in range(len(L)):  
    print(i, L[i])  
  
for i, val in enumerate(L):  
    print(i, val)  
  
# or without a index:  
for val in L:  
    print(val)
```

Functions, classes, and class functions

Functions, classes, and methods are core concepts in programming that help you organize code into logical, reusable, and efficient blocks. They allow you to move from writing simple scripts to building complex applications



Functions: The Reusable Tools

A function is a named, reusable block of code that performs a specific task. Think of it like a recipe 📖 or a coffee grinder ☕; you give it some inputs (arguments), it performs a series of steps, and it can give you an output (a return value).

Functions help you follow the DRY (Don't Repeat Yourself) principle. Instead of writing the same code multiple times, you write it once inside a function and then “call” that function whenever you need it.

```
# Defining a simple function
def add_numbers(a, b):
    result = a + b
    return result

# Calling the function and using its output
sum1 = add_numbers(5, 10) # sum1 is now 15
sum2 = add_numbers(100, 50) # sum2 is now 150
```

Installing Libraries with pip

You'll almost always use a command-line tool called pip, which is Python's standard package manager. It downloads and installs libraries from a central repository called the Python Package Index (PyPI).



To install a library in colab, use the !pip command, which executes pip from the command line:

```
!pip install datasets
```

Standard libraries

Python comes with a “battery-included” philosophy, meaning it has a rich Standard Library of modules that are pre-installed with every distribution. You can use these powerful tools just by importing them, no pip install needed.

Pandas: A library for handling panel data.

NumPy: A Library for numerical data manipulation.

SciPy: A library for Machine Learning

Sci-kit-learn: A library for traditional machine learning algorithms.

Given any library that has been installed (preinstalled or installed using pip), the command for importing the library is

```
import library
```

or, if we wish to give the library some alias, we can use the command

```
import library as alias
```

Standard libraries


For example, at the beginning of a python program, you might see a collection of imports, like so:

```
import pandas as pd
import numpy as np
import scipy as sp
import math
import sklearn
import transformers
import torch
import datasets
```

Once these libraries are imported, we can access any of the functions or variables.

```
print(math.cos(math.pi))
```


Classes: The Blueprints for Objects

A class is a blueprint  for creating objects. An object is a collection of related data and behaviors. The class defines what attributes (data) and methods (behaviors) all objects of that type will have.



For example, a Car class could define that all car objects must have attributes like color and model, and methods like start_engine() and drive().

```
# Defining a Car class (the blueprint)
class Car:
    # The __init__ method is a special method that runs when an object is created
    def __init__(self, color, model):
        self.color = color # Attribute
        self.model = model # Attribute
        self.is_running = False # Attribute

# Creating objects (instances) from the class
my_car = Car("Red", "Tesla Model S")
your_car = Car("Blue", "Ford Mustang")
```

Methods: Functions Inside a Class

A method is simply a function that is defined inside a class.  It defines a behavior or action that an object created from that class can perform.



The main difference between a function and a method is that a method is “aware” of the specific object it was called on. It can access and modify that object’s attributes (its internal data). This is done through the special first parameter, conventionally named `self`, which automatically refers to the object instance itself.

```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model
        self.is_running = False

    # This is a method
    def start_engine(self):
        print(f"The {self.model}'s engine is starting...")
        self.is_running = True # Modifies the object's attribute

    # This is another method
    def stop_engine(self):
        print("The engine is stopping.")
        self.is_running = False

# Create an object
my_car = Car("Red", "Tesla Model S")

# Call the method on the specific object instance
my_car.start_engine() # Output: The Tesla Model S's engine is starting...
print(f"Is the engine running? {my_car.is_running}") # Output: Is the engine running? True
```

Machine Learning

Machine Learning

Traditional machine learning classifiers are algorithms that learn to sort data into predefined categories. Think of it like a machine learning to sort mail into different bins (e.g., “Bills,” “Junk Mail,” “Personal Letters”). It learns from examples you’ve already sorted.



How They Work:

The process involves two main phases:

Training

The classifier is “trained” on a dataset where the correct categories, or labels, are already known. The algorithm analyzes the input features (e.g., for an email, features could be the sender, subject line, and specific words in the body) and learns the patterns associated with each label (e.g., “Spam” or “Not Spam”).

Prediction

Once trained, the classifier can take new, unlabeled data and predict which category it belongs to based on the patterns it learned.

This method is a core part of supervised learning, as it requires human-labeled data to supervise the learning process.

Common Examples

Some of the most well-known traditional classifiers include:



- **Logistic Regression:** Used for binary classification (two categories), it predicts the probability of an input belonging to a specific class.
- **Random Forest:** Creates a tree-like model of decisions. Each branch represents a feature, and each leaf node represents a class label. Creating many such trees creates a “Forest”.
- **Naive Bayes:** A probabilistic classifier based on Bayes’ theorem that works well for tasks like text classification.

These methods are often contrasted with modern deep learning models, which can automatically learn features from raw data (like images or text) without needing them to be manually defined first.

A Synthetic Dataset to get us started.

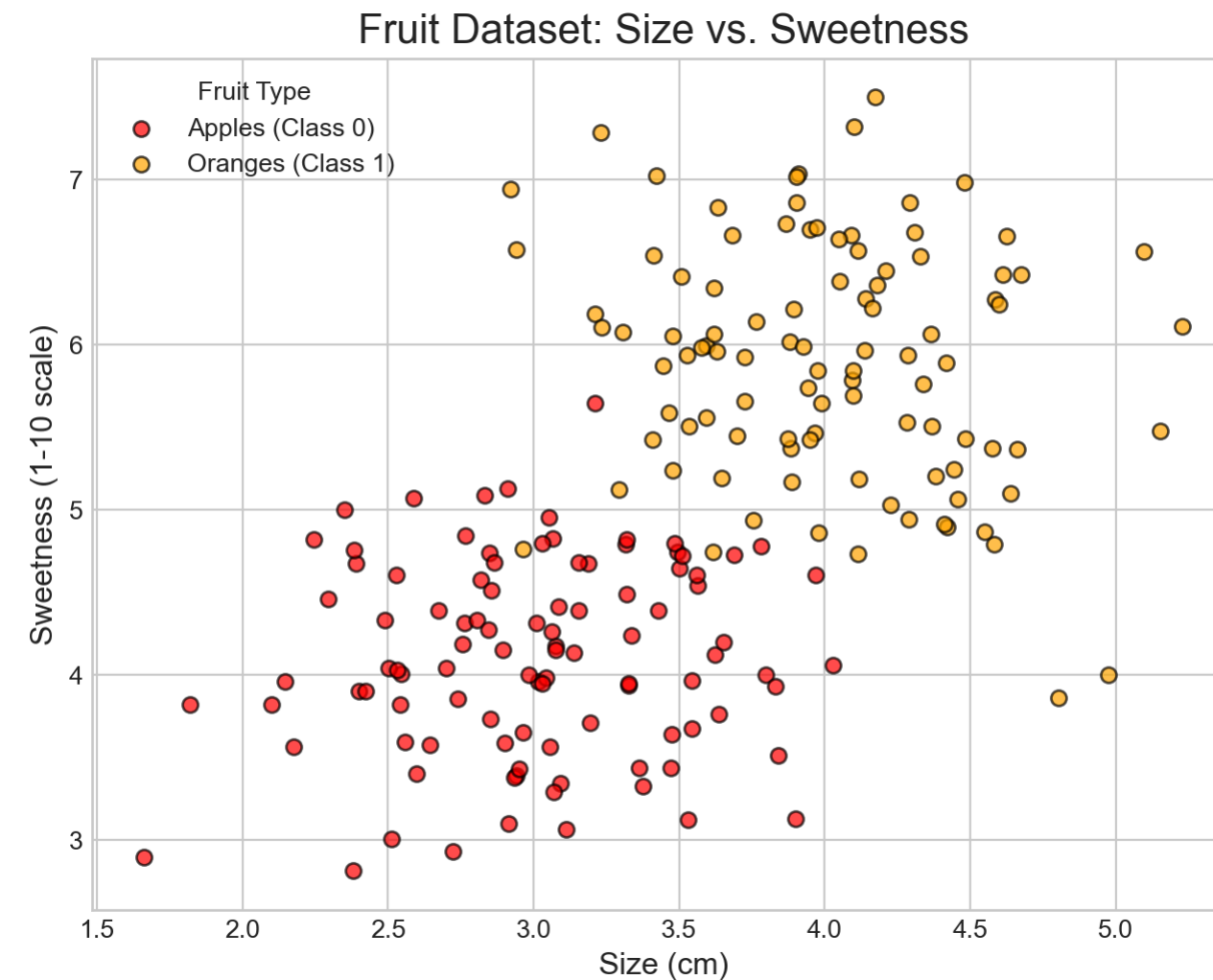
We start with a dataset of apples and oranges with two features; size and sweetness. In this dataset the size and sweetness are normally distributed with different means and standard deviations.

```
import matplotlib.pyplot as plt
import numpy as np

n_apples = 100
n_oranges = 100
apples_size = np.random.normal(3, 0.5, n_apples)
apples_sweetness = np.random.normal(4, 0.6, n_apples)

# Oranges (Class 1): Generally larger and sweeter
oranges_size = np.random.normal(4, 0.5, n_oranges)
oranges_sweetness = np.random.normal(6, 0.7, n_oranges)

plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(apples_size, apples_sweetness, c='red', label='Apples (Class 0)', alpha=0.7, edgecolors='k')
ax.scatter(oranges_size, oranges_sweetness, c='orange', label='Oranges (Class 1)', alpha=0.7, edgecolors='k')
ax.set_title('Fruit Dataset: Size vs. Sweetness', fontsize=16)
ax.set_xlabel('Size (cm)', fontsize=12)
ax.set_ylabel('Sweetness (1-10 scale)', fontsize=12)
ax.legend(title='Fruit Type')
ax.grid(True)
```



Data Management : Train, Test, and Dev

In machine learning, splitting your data into training, development (or validation), and test sets is a crucial practice for building robust and reliable models. Each set serves a distinct and vital purpose throughout the model development lifecycle.



Training Set: The Learning Ground 🧠

The **training set** is the largest portion of your data and is used to teach the machine learning model. Think of it as the textbook and practice problems you give to a student. The model observes this data, learns the underlying patterns and relationships between the input features and the output, and adjusts its internal parameters accordingly.

The primary need for a training set is to **fit the model**. Without it, the model would have no information to learn from and would be unable to make any predictions.

Test Set: The Final Exam 🏆

The **test set** is a final, completely unseen portion of the data that is used only once, after all the training and tuning is complete. This is the equivalent of a final exam. Its sole purpose is to provide an **unbiased evaluation of the final, tuned model's performance**.

The necessity of a test set lies in its ability to give you a true measure of how your model will perform in the real world on brand-new data. Because you used the development set to make decisions about the model (choosing the best architecture and hyperparameters), the model has been indirectly influenced by that data. Therefore, the performance on the development set might be slightly optimistic. The test set provides a final, objective assessment of the model's generalization capabilities.

Data Management : Train, Test, and Dev

Development (Validation) Set: The Practice Exam



The **development set**, often called the **validation set**, is a separate portion of the data that the model hasn't seen during training. This set acts like a practice exam for the student. Its main purpose is to provide an unbiased evaluation of the model's performance during the development phase and to help in **hyperparameter tuning**.

Here's why you need a development set:

- **Model Selection:** You'll often train several different models (e.g., with different architectures or algorithms). The development set helps you compare their performance and choose the best one.
- **Hyperparameter Tuning:** Machine learning models have various settings called hyperparameters (e.g., learning rate, number of layers in a neural network) that are not learned from the training data. You use the development set to experiment with different hyperparameter values and find the combination that yields the best performance.
- **Preventing Overfitting:** Overfitting occurs when a model learns the training data too well, including its noise and idiosyncrasies, and as a result, performs poorly on new, unseen data. By evaluating the model on the development set, you can get a more realistic estimate of its ability to generalize. If the model performs exceptionally well on the training set but poorly on the development set, it's a clear sign of overfitting.

Sklearns train-test split

We arrange this in terms X , which contains features and y which is our labels. 0 for apples and 1 for oranges.



```
X = np.vstack((
    np.column_stack((apples_size, apples_sweetness)),
    np.column_stack((oranges_size, oranges_sweetness))
))
y = np.hstack(([0] * n_apples, [1] * n_oranges))
```

Now we use this twice

```
X_pretrain, X_test, y_pretrain, y_test = sklearn.model_selection.train_test_split(X, y, test_size=0.2, random_state=42)
X_train, X_dev, y_train, y_dev = sklearn.model_selection.train_test_split(X_pretrain, y_pretrain, test_size=0.15, random_state=42)
```

Our goal is to distinguish apples and oranges based on the features.

Sklearn Classifiers



Logistic Regression: The Linear Approach

Logistic Regression is a simple yet powerful classifier that works by finding a straight line (or a flat plane in higher dimensions) that best separates the different classes. It learns the optimal position and angle for this line from the training data.

Once this line, known as the decision boundary, is established, the classifier can decide the class of a new, unseen fruit. If the new fruit’s data point falls on one side of the line, it’s classified as an Apple; if it falls on the other, it’s an Orange. It’s best suited for problems where the data is mostly linearly separable.

```
clf = sklearn.linear_model.LogisticRegression()  
clf.fit(X_train, y_train)  
y_pred = clf.predict(X_test)  
print(sklearn.metrics.classification_report(y_test, y_pred))
```

The results are as follows:

	precision	recall	f1-score	support
0	0.95	0.95	0.95	21
1	0.95	0.95	0.95	19
accuracy			0.95	40
macro avg	0.95	0.95	0.95	40
weighted avg	0.95	0.95	0.95	40

Sklearn Classifiers

Random Forest: The Flexible Approach



A Random Forest is a more complex and flexible model. It works by building a multitude of individual decision trees and then combining their outputs. Each tree asks a series of simple questions about the features (e.g., “Is the size > 4.5cm?”).

By combining hundreds of these simple trees, a Random Forest can create a highly complex, non-linear decision boundary that can curve and adapt to the data’s structure. This makes it very powerful for problems where a simple straight line isn’t enough to separate the classes effectively. It essentially takes a “wisdom of the crowd” approach to classification.

```
clf = sklearn.ensemble.RandomForestClassifier()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(sklearn.metrics.classification_report(y_test, y_pred))
```

The results:

	precision	recall	f1-score	support
0	1.00	0.86	0.92	21
1	0.86	1.00	0.93	19
accuracy			0.93	40
macro avg	0.93	0.93	0.92	40
weighted avg	0.94	0.93	0.92	40

Sklearn Classifiers



Naive Bayes: Simple Priors

A simple yet powerful probabilistic classifier based on Bayes’ Theorem. Its core principle is to calculate the probability that a given data point belongs to a particular class based on its features. The “naive” part of its name comes from a key, simplifying assumption: it treats all features as being independent of one another. For example, when classifying an email as spam, the algorithm assumes the presence of the word “free” has no bearing on the presence of the word “winner,” even though in reality, they might often appear together. Despite this “naive” assumption, the classifier is remarkably effective, especially for text classification and spam filtering, because of its speed, simplicity, and low data requirements.

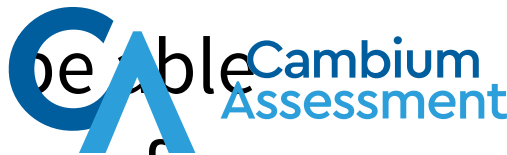
```
clf = sklearn.naive_bayes.GaussianNB()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(sklearn.metrics.classification_report(y_test, y_pred))
```

The results:

	precision	recall	f1-score	support
0	0.95	0.95	0.95	21
1	0.95	0.95	0.95	19
accuracy			0.95	40
macro avg	0.95	0.95	0.95	40
weighted avg	0.95	0.95	0.95	40

Saving and Loading Models

Sometimes the training process takes minutes, hours, days, or even months. In most cases, it is useful to be able to save and load a trained model. For the classifiers above, the joblib library facilitates the saving and loading of a model.



```
import joblib

joblib.dump(clf, 'classifier.joblib')
clf_copy = joblib.load('classifier.joblib')

y_pred = clf.predict(X_test)
print(sklearn.metrics.classification_report(y_test, y_pred))

y_pred = clf_copy.predict(X_test)
print(sklearn.metrics.classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.95	0.95	0.95	199
1	0.96	0.96	0.96	201
accuracy			0.95	400
macro avg	0.95	0.95	0.95	400
weighted avg	0.95	0.95	0.95	400

	precision	recall	f1-score	support
0	0.95	0.95	0.95	199
1	0.96	0.96	0.96	201
accuracy			0.95	400
macro avg	0.95	0.95	0.95	400
weighted avg	0.95	0.95	0.95	400

Automated Essay Scoring

Automated Essay Scoring

Generally, Automated Essay Scoring (AES) works by using statistical models to assign grades that approximate human scoring. The methods have evolved over time:



Traditional Methods: Initial AES systems used Bag-of-Words (BoW) models. These models relied on a combination of frequency-based data and hand-crafted “global features” such as word count, sentence length, and readability metrics. These features were not based on the essay’s detailed semantics or organizational structure.

Modern Methods: Many current AES engines use transformer-based Large Language Models (LLMs). This approach offers improved accuracy compared to older bag-of-words models. However, the improvement in accuracy comes at the cost of reduced interpretability because the features are defined implicitly by the model rather than being explicitly engineered.

Hybrid Approaches: Some researchers combine features derived from LLMs with traditional hand-crafted features in an effort to enhance both accuracy and interpretability

The primary idea is that we take a piece of text and we transform this into a set of features. Those features become the basis for a traditional method.

Dataset

For this little course, we will be taking the



```
data_id = "llm-aes/asap-7-original"
data = datasets.load_dataset(data_id)
```

The first few entries are here:

	essay_id	essay_set	essay	rater1_domain1	rater2_domain1	domain1_score	rater1_trait1	r
0	17834	7	Patience is when your waiting .I was patience ...	8	7	15	1.0	2
1	17836	7	I am not a patience person, like I can't sit i...	6	7	13	1.0	1
2	17837	7	One day I was at	7	8	15	1.0	2

essay_id	essay_set	essay	rater1_domain1	rater2_domain1	domain1_score	rater1_trait1	r
		basketball practice and I was...					

Metrics for automated scoring

The standards for automated scoring evaluation as specified by Williamson et al. (2012) include the following metrics:



- Cohen's Quadratic Weighted Kappa (QWK): This is the primary statistic used to measure agreement. QWK values range from -1 (perfect disagreement) to 1 (perfect agreement) and are often interpreted as the probability of agreement beyond what would be expected by random chance.
- Exact Agreement: This is a secondary statistic that is considered less reliable because it can be skewed by uneven score distributions.
- Standardized Mean Difference (SMD): This statistic is used to measure standardized relative bias. A positive or negative SMD value indicates that the model is introducing a positive or negative bias, respectively. The SMD can also be calculated for specific demographic subgroups to assess the model's bias for that group.

```
def aes_metrics(y1, y2):  
    qwk = sklearn.metrics.cohen_kappa_score(y1, y2, weights="quadratic")  
    acc = sklearn.metrics.accuracy_score(y1, y2)  
    smd_numerator = np.mean(y1) - np.mean(y2)  
    smd_denominator = np.sqrt((np.std(y1)**2 + np.std(y2)**2)/2)  
    smd = smd_numerator / smd_denominator  
    return {"QWK": qwk, "Acc": acc, "SMD": smd}
```

We need to know the inter-rater statistics by comparing rater 1 to rater 2:

```
{'QWK': np.float64(0.7299561819547309), 'Acc': 0.321656050955414, 'SMD': np.float64(-0.1046846948907742)}
```

Bag of Words Classifier

The most basic features are word counts. The system creates a vocabulary of thousands of words, and the vector for an essay contains the frequency of each of those words.



```
class EssayScorer:

    def __init__(self):
        self.tfidf = TfidfVectorizer()
        self.classifier = LogisticRegression(class_weight="balanced")

    def train(self, X, y):
        X_tfidf = self.tfidf.fit_transform(X)
        self.classifier.fit(X_tfidf, y)

    def score(self, X):
        X_tfidf = self.tfidf.transform(X)
        return self.classifier.predict(X_tfidf)
```

If we are going to implement this:

```
Pete = EssayScorer()
Pete.train(train['essay'], train['rater1_domain1'])
pred = Pete.score(test['essay'])
print(aes_metrics(test['rater1_domain1'],pred))
```

Result:

```
{'QWK': np.float64(0.5665739821251241), 'Acc': 0.2229299363057325, 'SMD': np.float64(-0.20059043467181287)}
```

Writing Our Second Classifier

One thing we can do is group like terms. Given the large vocabulary, we want to group terms that appear similarly with each other. One way to do this is to use dimension reduction techniques. Given a matrix, X , we can form a Singular Value Decomposition (SVD) of X , so that

$$X = UDV^t$$

where D is a rectangular diagonal matrix and V^t is **orthogonal**. If we assume it is square, we can write it like this:

$$D = \begin{pmatrix} d_0 & 0 & \dots & 0 \\ 0 & d_1 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & d_{n-1} \end{pmatrix}$$

Where $|d_0| \geq |d_1| \geq \dots \geq |d_{n-1}|$. Let D_k be the matrix that only contains the first k values, U_k contain the first k rows of U and V_k^t contain the first k columns of V .

The

$$X = UDV \approx U_k D_k V_k^t$$

so we obtain $X \rightarrow \tilde{X} = U_k D_k$ by multiplication on the left by V_k (orthogonality).

SVD: Fewer features, better features

Singular Value Decomposition (SVD) helps control overfitting by acting as a powerful regularization and dimensionality reduction technique. It separates the important, underlying patterns (signal) in your data from the less important, random fluctuations (noise).



By using Truncated SVD, you can create a lower-rank approximation of your data. This involves two key steps:

1. Decomposition: You decompose your original data matrix, A , into its singular values and vectors.
2. Truncation: You keep only the top 'k' singular values and their corresponding vectors, discarding the rest. These 'k' components represent the strongest signals.

The result is a simplified, “denoised” version of your data that occupies a lower-dimensional space.

Implementation

To implement this idea, we can import the right libraries.



```
class TruncEssayScorer:

    def __init__(self, dim):
        self.tfidf = TfidfVectorizer()
        self.svd = TruncatedSVD(n_components=dim)
        self.classifier = LogisticRegression(class_weight="balanced")

    def train(self, X, y):
        X_tfidf = self.tfidf.fit_transform(X)
        X_svd = self.svd.fit_transform(X_tfidf)
        self.classifier.fit(X_svd, y)

    def score(self, X):
        X_tfidf = self.tfidf.transform(X)
        X_svd = self.svd.transform(X_tfidf)
        return self.classifier.predict(X_svd)
```

To use it, we have

```
dim = 300
Pete = TruncEssayScorer(dim)
Pete.train(train['essay'], train['rater1_domain1'])
pred = Pete.score(test['essay'])
print(aes_metrics(test['rater1_domain1'], pred))
```

Result:

```
{'QWK': np.float64(0.5735467165498909), 'Acc': 0.24203821656050956, 'SMD': np.float64(-0.15478184912459145)}
```

Writing Our Third Classifier

What if the words aren't sufficient. We need a measure of quality. Some simple text statistics could reveal a little more about the essay. For example, the average letters per word may indicate that the student is using sophisticated vocabulary. The number of words per sentence may give us sentence complexity. Readability measures give an indication of relative grade level and so on. So let us import a library that provides us with some of these statistics:



```
!pip install textstat
import textstat
```

This exposes lots of functions

```
'automated_readability_index',
'avg_character_per_word',
'avg_letter_per_word',
'avg_sentence_length',
'avg_sentence_per_word',
'avg_syllables_per_word',
'backend',
'char_count',
'coleman_liau_index',
'count_arabic_long_words',
'count_arabic_syllables',
'count_complex_arabic_words',
'count_faseeh',
'crawford',
```

Feature-based engine

Let us start with a function that derives a collection of supposedly useful features:

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier

def get_features(texts):
    features = [[textstat.letter_count(x),
                  textstat.sentence_count(x),
                  textstat.avg_letter_per_word(x),
                  textstat.avg_sentence_per_word(x),
                  textstat.flesch_reading_ease(x),
                  textstat.automated_readability_index(x),
                  textstat.dale_chall_readability_score(x),
                  textstat.avg_syllables_per_word(x),
                  textstat.reading_time(x)] for x in texts]
    return np.array(features)
```

Lets apply a classifier to the output:

```
X_train = get_features(train['essay'])
X_test = get_features(test['essay'])
clf = RandomForestClassifier(class_weight="balanced")
clf.fit(X_train, train['rater1_domain1'])
pred = clf.predict(X_test)
print(aes_metrics(test['rater1_domain1'],pred))
```

So with just a few features, we get

```
{'QWK': np.float64(0.6329539878810928), 'Acc': 0.2898089171974522, 'SMD': np.float64(-0.15438930588658045)}
```


Combining the Features for a Classifier


The power of this method comes from combining both feature sets into a single, comprehensive vector for each document. The process is a straightforward concatenation. 

Generate Vectors: For a single document, first, generate its TF-IDF vector (e.g., a vector of 10 dimensions).

Calculate Features: Next, calculate its hand-crafted features (e.g., a vector of 10 dimensions).

Concatenate: Append the hand-crafted feature vector to the end of the TF-IDF vector. This creates a new, combined feature vector (in this example, 10,005 dimensions).

Train the Model: This final, combined vector is then used as the input to train any standard classification algorithm, such as a Support Vector Machine (SVM), Logistic Regression, or Random Forest.

By combining these two types of features, the model gets a more holistic view of the data. It learns from both the nuanced word choices captured by TF-IDF and the explicit structural properties captured by the hand-crafted features, often leading to a significant boost in classification performance. 

In code

It is fairly easy to implement this.



```
class BoWScorer:

    def __init__(self, dim):
        self.tfidf = TfidfVectorizer()
        self.svd = TruncatedSVD(n_components=dim)
        self.classifier = RandomForestClassifier(class_weight="balanced")

    def train(self, X, y):
        X_tfidf = self.tfidf.fit_transform(X)
        X_svd = self.svd.fit_transform(X_tfidf)
        X_features = get_features(X)
        X_combined = np.concatenate((X_svd, X_features), axis=1)
        self.classifier.fit(X_combined, y)

    def score(self, X):
        X_tfidf = self.tfidf.transform(X)
        X_svd = self.svd.transform(X_tfidf)
        X_features = get_features(X)
        X_combined = np.concatenate((X_svd, X_features), axis=1)
        return self.classifier.predict(X_combined)
```

The results aren't bad

```
dim = 10
Pete = BoWScorer(dim)
Pete.train(train['essay'], train['rater1_domain1'])
pred = Pete.score(test['essay'])
print(dim, aes_metrics(test['rater1_domain1'], pred))
{'QWK': np.float64(0.6776741192311676), 'Acc': 0.3821656050955414, 'SMD': np.float64(-0.16812655240837246)}
```

Turning on a GPU

Instructions

1. Click on the **Runtime** tab in the top menu bar.
2. From the dropdown menu, select **Change runtime type**.
3. In the pop-up window, click the dropdown menu for **Hardware accelerator**.
4. Select **GPU** from the list.
5. Click **Save**.

Verification

To confirm the GPU is active, you can run the following code cell in your notebook. It uses the `nvidia-smi` command to display information about the connected NVIDIA GPU.



```
1 !nvidia-smi
```

If the command executes and shows a table with GPU details (like driver version, CUDA version, and GPU name, e.g., Tesla T4), you have successfully enabled the GPU. If it returns an error, something went wrong, and you should try the steps again.

Pretrained Embeddings

Creating a document embedding involves condensing the meaning of all the words in a document into one fixed size numerical vector. There are a few common ways to do this:



- **Pooling Word Embeddings:** A simple approach is to take the pretrained word embeddings (like GloVe or Word2Vec) for every word in the document and then combine them, for instance, by averaging them. This creates a single vector that represents a “mean” of the document’s meaning.
- **Specialized Models:** More advanced models are trained specifically to create document vectors. These models learn not just word meanings but also how word order and grammar contribute to the overall meaning of a sentence or paragraph.
- **Transformer-Based Models:** Modern models like BERT are excellent at this. They can generate a single vector (often from a special [CLS] token) that represents the entire input text, taking the full context of the sequence into account.

Implementation



```
class EmbeddingEssayScorer:

    def __init__(self, model_id):
        self.emb = SentenceTransformer(model_id)
        self.classifier = LogisticRegression(class_weight="balanced")

    def train(self, X, y):
        self.X_train = self.emb.encode(list(X))
        self.classifier.fit(self.X_train, list(y))

    def score(self, X):
        self.X_test = self.emb.encode(list(X))
        return self.classifier.predict(self.X_test)
```

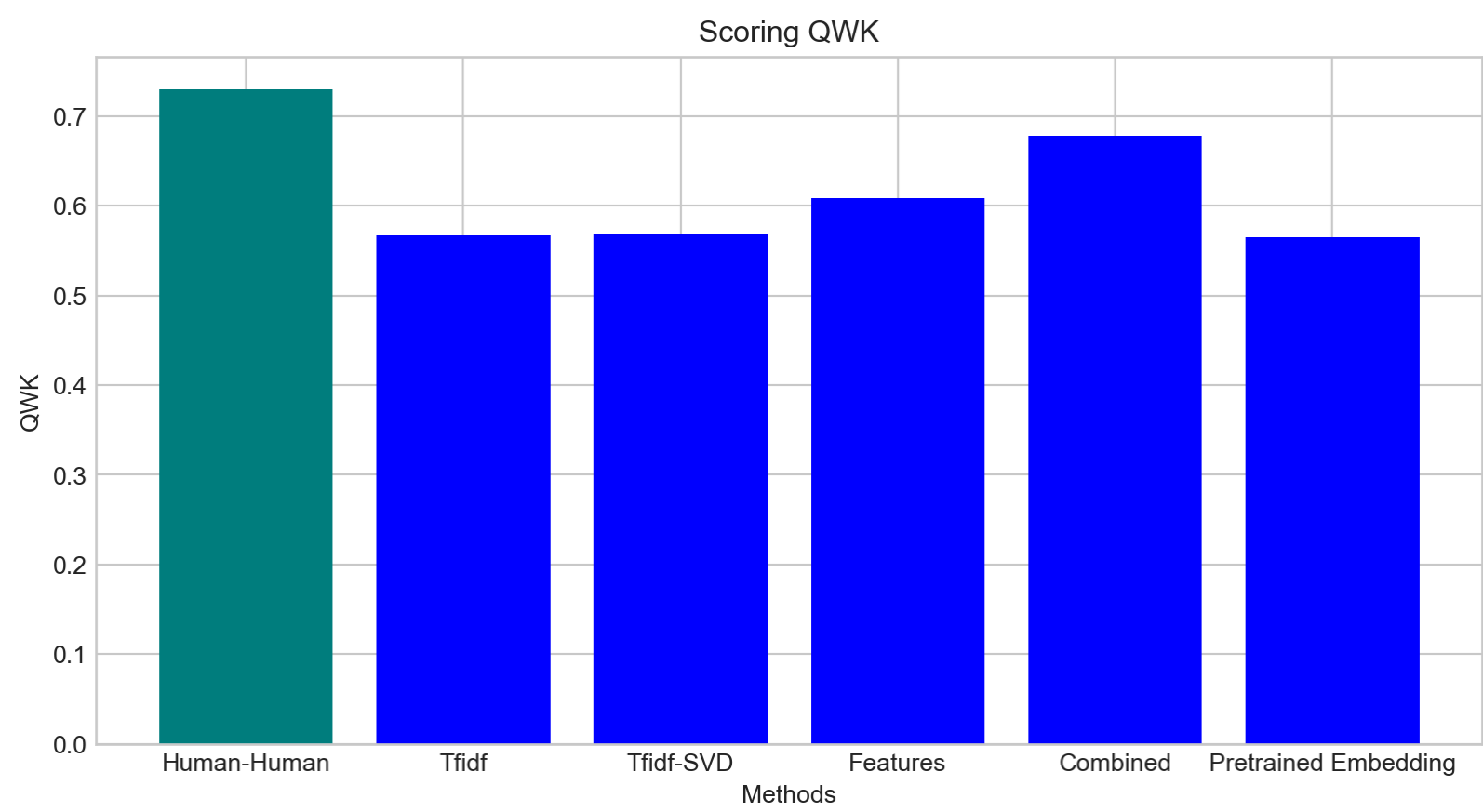
Then implementing this method gives us pretty disappointing results:

```
model_id = "all-MiniLM-L6-v2"
Pete = EmbEssayScorer(model_id)
Pete.train(list(train['essay']), list(train['rater1_domain1']))
pred = Pete.score(test['essay'])
print(aes_metrics(test['rater1_domain1'],pred))

{'QWK': np.float64(0.5647803912501483), 'Acc': 0.2070063694267516, 'SMD': np.float64(0.02212369076606493)}
```

Performance so far

The performance of our models so far is a little underwhelming.



We can improve the performance with very specific features, such as those derived specifically for the prompt, spelling errors, organization-based measures. But we are here to talk about AI.

Fine-Tuning Language Models for Classification

This is where the model is specialized for a task like sentiment analysis, topic categorization, or spam detection.



The process involves a few key steps:

Add a Classification Head: The pre-trained model is an expert at understanding text but doesn't know how to output a simple category label. To fix this, a small, new neural network layer, called a “classification head,” is attached to the end of the pre-trained model. The job of this head is to take the rich, contextual understanding from the base model and map it to the desired output classes (e.g., “Positive,” “Negative,” “Neutral”).

Prepare Labeled Data: You need a custom dataset that's specific to your task. This dataset consists of examples paired with the correct labels (e.g., an email labeled as “Spam” or a news headline labeled as “Sports”). This dataset is typically thousands of times smaller than the data used for pre-training.

Train the Model: The labeled data is fed through the combined model (base + head). The model makes a prediction, which is compared against the true label to calculate an error. This error is then used to make small adjustments to the parameters of both the new classification head and, crucially, the original pre-trained model. This step doesn't change the model drastically; it just “nudges” its vast existing knowledge to become more focused and accurate for your specific task.

Training Regimes

Lets start with a basic definition



```
class FineTunedEssayScorer:

    def __init__(self, model_id, max_score, min_score):
        self.max_score = max_score
        self.min_score = min_score
        self.classifier = AutoModelForSequenceClassification.from_pretrained(model_id, num_labels = max_score - min_score + 1)
        self.tokenizer = AutoTokenizer.from_pretrained(model_id)

    def score(self, essays):
        self.classifier.eval()
        scores = []
        with torch.no_grad():
            for X_batch in tqdm(essay):
                X_batch = self.tokenizer(X_batch,
                                         return_tensors='pt',
                                         padding="max_length",
                                         truncation=True,
                                         max_length=512)

                outputs = self.classifier(**X_batch)
                scores.append(int(outputs.logits.cpu().argmax(dim=1)) + self.min_score)
        return scores
```

Using an Optimizer

The optimizer we use, Adam with a weight decay, is an advanced variant of the SGD method. The training loop consists of



- Obtaining a model prediction
- Calculating a loss function
- Calculate the gradients of the loss function
- Iterate the optimizer algorithm

```
def train(self, X, y, epochs = 10):
    self.classifier.train()
    optimizer = torch.optim.AdamW(self.classifier.parameters(), lr=5e-5)
    N = len(X)
    for e in range(epochs):
        for i in tqdm(range(N)):
            optimizer.zero_grad()
            X_batch = self.tokenizer(X[i],
                                     return_tensors='pt',
                                     padding="max_length",
                                     truncation=True,
                                     max_length=512).to(self.classifier.device)
            y_batch = y[i] - self.min_score
            outputs = self.classifier(**X_batch, labels=torch.tensor([y_batch]).to(self.classifier.device))
            loss = outputs.loss
            loss.backward()
            optimizer.step()
```

Implementing the fine-tuned model

Once we have trained the model, we implement it by



```
Pete = FineTunedEssayScorer("google/electra-small-discriminator", max_score = max(train['rater1_domain1']), min_score = min(train['rater1_domain1']))
Pete.train(list(train['essay']), list(train['rater1_domain1']), epochs=4)
pred = Pete.score(test['essay'])
print(aes_metrics(test['rater1_domain1'],pred))
```

Using the code above for 20 epochs provided our best result so far:

```
{'QWK': 0.698031295937573, 'Acc': 0.23885350318471338, 'SMD': -0.0015097224978592957}
```

Things not done

The above constitutes how we actually trained the model back in 2019 when we first trained BERT for scoring in our publication “Language models and Automated Essay Scoring”. We had no other way.



There are many things this code does not do that are now standard practice:

- Separate a dev set: As explained above, it is good practice to have a development set. If something has gone wrong, the dev set will let you know.
- Early Stopping: At the end of each epoch, we test the model performance on the dev set, saving the state space of the best model over the epochs.
- Learning Rate Scheduler: The best practices are to decrease the learning rate as the optimization progresses. This means the SGD makes smaller and smaller steps as we get closer to converging.
- Gradient Clipping: Sometimes things go wrong (think Newton's method when the gradient is 0). Gradient clipping allows the gradient to be a maximum of 1 over the model parameters to prevent wildly moving in unwanted directions.
- Saving/Loading: We also want to save the weights on the local drive. This can be done using the `save_pretrained` function.
- Comments: Make sure you say what you are going to do. This allows someone else to read the code.

Fully Fleshed Out Code

A fully fleshed out piece of code that does this is



```
# Import necessary libraries
from transformers import AutoModelForSequenceClassification, AutoTokenizer # For loading pre-trained models and tokenizers
import torch # PyTorch library for tensors and neural networks
from tqdm.notebook import tqdm, trange # For displaying progress bars

class FineTunedEssayScorer:
    """
    A class to fine-tune and use a transformer model for automated essay scoring.

    This class wraps the process of loading a pre-trained model, fine-tuning it
    on a specific essay scoring task (as a classification problem), and using it
    for inference (scoring new essays).
    """
```

A Fully functional version implementing all of these is here:

<https://github.com/christopherormerod/AI-AES-Colab/blob/main/FullExample.ipynb>

However, many of these issues are handled directly by the next section.

```
{'QWK': 0.7234159302258418, 'Acc': 0.30254777070063693, 'SMD': -0.07607090838657149}
```

Huggingface Trainer

The Hugging Face Trainer class is a high-level API that significantly simplifies the process of training Transformer models. While a manual loop offers maximum flexibility, the Trainer provides numerous advantages for most common use cases:



- **Reduced Boilerplate:** It abstracts away the repetitive and often complex code required for a standard training loop. You don't need to manually write loops for epochs and batches, handle device placement (`.to(device)`), manage optimizer steps, or calculate gradients.
- **Integrated Features:** The Trainer comes with many powerful, built-in features that you would otherwise have to implement yourself:
- **Distributed Training:** Easily scales your training across multiple GPUs or TPUs with minimal code changes.
- **Mixed-Precision Training:** Enables faster training and reduced memory usage with a simple flag (`fp16=True`).
- **Checkpoint Management:** Automatically saves and manages model checkpoints during training, making it easy to resume or use the best-performing model.

Continued Trainer



- Logging & Progress Bars: Provides clean and informative progress bars and integrates seamlessly with logging libraries like TensorBoard and Weights & Biases.
- Evaluation: Includes a built-in evaluation loop that runs validation and computes metrics at specified intervals.
- Gradient Accumulation: Allows for training with larger effective batch sizes than what can fit into memory.
- Simplicity and Readability: The training configuration is neatly organized in the TrainingArguments class, making your code cleaner, easier to read, and less prone to bugs compared to a long, manual training script.
- Optimization: The Trainer is highly optimized and maintained to work efficiently with the entire Hugging Face ecosystem, including the datasets and evaluate libraries.

Datasets and formalizing the code

When something has sufficient properties, it should be its own object. Datasets are a good example:



```
train_encodings = tokenizer(list(train['essay']), truncation=True, padding=True)
test_encodings = tokenizer(list(test['essay']), truncation=True, padding=True)

class EssayDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

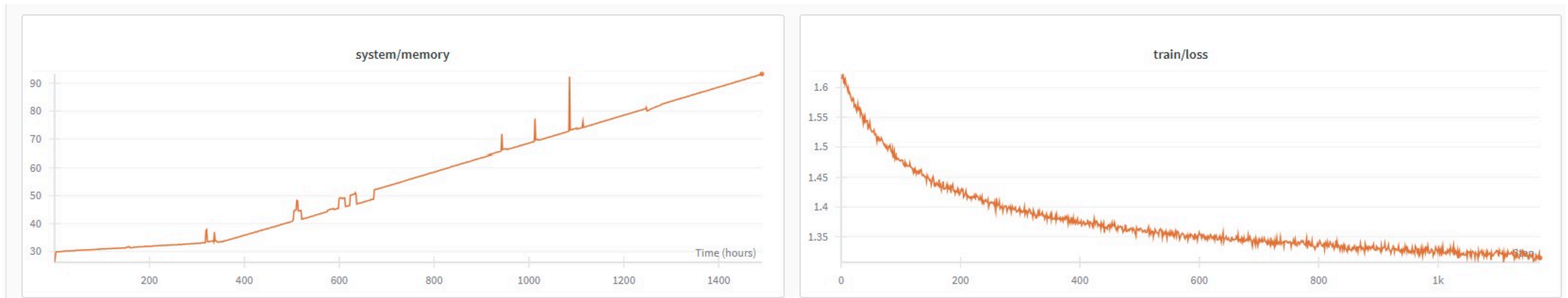
train_dataset = EssayDataset(train_encodings, list(train['rater1_domain1']))
test_dataset = EssayDataset(test_encodings, list(test['rater1_domain1']))
```

Callbacks and Other functions

On each epoch, we can build in which metrics to compute

```
def compute_metrics(eval_pred):  
    logits, labels = eval_pred  
    predictions = np.argmax(logits, axis=-1)  
    return {"QWK":cohen_kappa_score(labels, predictions,weights="quadratic"),  
            "SMD":SMD(labels, predictions),  
            "ACC":accuracy_score(labels, predictions)}
```

while Callbacks can handle progress bars and integration with Weights and Biases.



These are made available online at
<https://wandb.ai/>

Implementation

There are 3 steps; defining the arguments, instantiating the trainer, running the trainer.

```
def train(self, X, y, epochs = 20):
    args = TrainingArguments(
        output_dir="output",
        evaluation_strategy="epoch",
        save_strategy="epoch",
        learning_rate=2e-5,
        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        num_train_epochs=epochs,
    )
    trainer = Trainer(
        model=self.classifier,
        args=args,
        train_dataset=train_dataset,
        eval_dataset=test_dataset,
        compute_metrics=compute_metrics,
    )
    trainer.train()
```

Advanced topics

Hyperparameter Tuning

Grid Search: The Brute-Force Method



Grid Search is the most straightforward method for hyperparameter tuning. You define a specific list of values for each hyperparameter you want to tune, and the algorithm exhaustively trains and evaluates a model for every possible combination of these values.

Imagine you're tuning two hyperparameters:

Learning Rate: [0.1, 0.01, 0.001]

Batch Size: [32, 64]

Grid Search will create a “grid” of all combinations and test each one:

Learning Rate = 0.1, Batch Size = 32

Learning Rate = 0.1, Batch Size = 64

Learning Rate = 0.01, Batch Size = 32

Learning Rate = 0.01, Batch Size = 64

Learning Rate = 0.001, Batch Size = 32

Learning Rate = 0.001, Batch Size = 64

Hyperparameter Tuning

Grid Search is the most straightforward method for hyperparameter tuning. You define a specific list of values for each hyperparameter you want to tune, and the algorithm exhaustively trains and evaluates a model for every possible combination of these values.



Pros:

Simple: Easy to understand and implement.

Exhaustive: It's guaranteed to find the best combination within the specific grid you've defined.

Cons:

Inefficient: It's computationally very expensive and suffers from the curse of dimensionality. If you have many hyperparameters or many values to test, the number of combinations explodes, making it impractical.

Blind: It doesn't learn from past evaluations. A run with terrible performance provides no information to guide the search away from that region of the hyperparameter space.

Implementation

```
for dim in dim_values:
    print(f"Training model with dim = {dim}...")

    # Initialize and train the model
    scorer = TruncEssayScorer(dim=dim)
    scorer.train(train['essay'], train['domain1_score'])

    # Make predictions on the development set
    dev_predictions = scorer.score(dev['essay'])

    # Calculate the Quadratic Weighted Kappa (QWK)
    qwk = cohen_kappa_score(dev['domain1_score'], dev_predictions, weights='quadratic')

    # Store the result
    results[dim] = qwk
    print(f"    -> QWK on dev set: {qwk:.4f}\n")

best_dim = max(results, key=results.get)
best_qwk = results[best_dim]

print(f"Best 'dim' found: {best_dim}")
print(f"Corresponding QWK on dev set: {best_qwk:.4f}")

test_predictions = scorer.score(test['essay'])
test_qwk = cohen_kappa_score(test['domain1_score'], test_predictions, weights='quadratic')
print(f"QWK on test set: {test_qwk:.4f}")
```

Bayesian Optimization: The Smart Search Method

Bayesian Optimization is a more intelligent and efficient approach. Instead of blindly trying all combinations, it uses the results from previous trials to make an informed decision about which set of hyperparameters to try next. It builds a probabilistic model of the objective function (e.g., model accuracy) and uses this model to select the most promising hyperparameters.

The process works in a loop:

Pick a set of hyperparameters and train the model.

Record the performance (e.g., validation accuracy).

Update a probabilistic “surrogate model” that maps hyperparameters to performance. This model represents the algorithm’s “belief” about how good any given set of hyperparameters might be.

Use an acquisition function to decide the next set of hyperparameters to test. This function balances exploitation (choosing hyperparameters that the surrogate model predicts will perform well) and exploration (choosing hyperparameters in an uncertain region to gather more information).

Repeat until a stopping criterion (like a set number of iterations) is met.

Tree-structured Parzen Estimator (TPE)

TPE is a popular Bayesian optimization algorithm. Unlike traditional methods that model $P(\text{score} \mid \text{hyperparameters})$, TPE cleverly models $P(\text{hyperparameters} \mid \text{score})$.



Here's the core idea:

It gathers a history of hyperparameter-score pairs.

It divides those observations into two groups: a “good” group (e.g., the top 25% of scores) and a “bad” group (the rest).

It creates two probability distributions: one for the hyperparameters in the “good” group and one for those in the “bad” group.

To choose the next hyperparameters to test, it looks for values that are highly likely in the “good” distribution but unlikely in the “bad” one. This effectively asks, “Which hyperparameters look more like the ones that have performed well in the past?”

Tree-structured Parzen Estimator (TPE)



Pros:

- Efficient: It finds better hyperparameters in far fewer iterations than Grid Search or Random Search.
- Intelligent: It learns from its mistakes and progressively focuses on more promising regions of the search space.

Cons:

- Complex: More difficult to understand and implement from scratch.
- Sequential: It is harder to parallelize because the choice for the next trial depends on the results of the previous ones.

Implementation

For this, we use Optuna, which has implemented a TPE method:

```
import optuna

def objective(trial):
    dim = trial.suggest_int('dim', 10, 300)
    scorer = TruncEssayScorer(dim=dim)
    scorer.train(train['essay'], train['rater1_domain1'])
    dev_predictions = scorer.score(dev['essay'])
    qwk = cohen_kappa_score(dev_predictions, dev['rater1_domain1'], weights="quadratic")
    return qwk

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=20)

best_dim = study.best_params['dim']
```

Once the best dimension is found, we need to

```
best_dim = study.best_params['dim']
scorer = TruncEssayScorer(dim=best_dim)
scorer.train(train['essay'], train['rater1_domain1'])
test_predictions = scorer.score(test['essay'])
test_metrics = aes_metrics(test['rater1_domain1'], test_predictions)
print(test_metrics)
```

Regression-Based Scoring

When you have a large number of targets, and few training examples to work from per target, sometimes it is worth working with regression based scoring instead of cross-entropy. The first goal is to map the scores from min_score to max_score to the interval $I = [0, 1]$. The best way to do this is by dividing the interval I into equal pieces, then mapping each score point to the midpoint of those intervals. Some simple algebra tells us that this linear function is

$$\mu(n) = \frac{n - \min + \frac{1}{2}}{\max - \min + 1}$$

The inverse of this mapping, which sends the each interval back to the range $\{\min, \dots, \max\}$ is given by

$$\mu^{-1}(x) = [y(\max - \min + 1) + \min - 0.5]$$

Now, as the output layer of the LLM is a linear function, we use σ to map the output to I .

Implementation

The main trick is that we use I instead of integers. We use the functions:

```
def sigmoid(x):  
    return 1 / (1 + np.exp(-x))  
  
def mu(n, min_score, max_score):  
    return (n - min_score + 0.5) / (max_score - min_score + 1)  
  
def mu_inv(x, min_score, max_score):  
    return int(np.round(x * (max_score - min_score + 1) + min_score - 0.5))
```

so that the dataset returns $x \in I$ instead of integers.

```
class EssayDataset(Dataset):  
    def __init__(self, encodings, labels):  
        self.encodings = encodings  
        self.labels = labels  
  
    def __len__(self):  
        return len(self.labels)  
  
    def __getitem__(self, idx):  
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}  
        item['labels'] = torch.tensor([mu(self.labels[idx], min_score, max_score)])  
        return item
```

Implementation - Personalized Trainer

We can write our own trainer that inherits from the trainer. We just need to change the loss function:



```
def compute_loss(self, model, inputs, return_outputs=False, num_items_in_batch=1):
    labels = inputs.pop("labels")
    outputs = model(**inputs)
    logits = outputs.logits
    loss = torch.nn.functional.mse_loss(logits.sigmoid(), labels)
    return (loss, outputs) if return_outputs else loss
```

We can also write our own callback to track the metrics by mapping back to the integers then calculating QWK:

```
class MyCallback(TrainerCallback):
    def on_epoch_end(self, args, state, control, **kwargs):
        preds = control.trainer.predict(control.dev)
        pred_ids = [mu_inv(x,min_score, max_score) for x in sigmoid(preds.predictions)]
        true_ids = [mu_inv(x,min_score, max_score) for x in preds.label_ids]
        qwk = sklearn.metrics.cohen_kappa_score(pred_ids, true_ids, weights="quadratic")
        print(f"QWK: {qwk}")
        if qwk > control.qwk:
            control.qwk = qwk
            control.state_dict = kwargs['model'].state_dict()
```

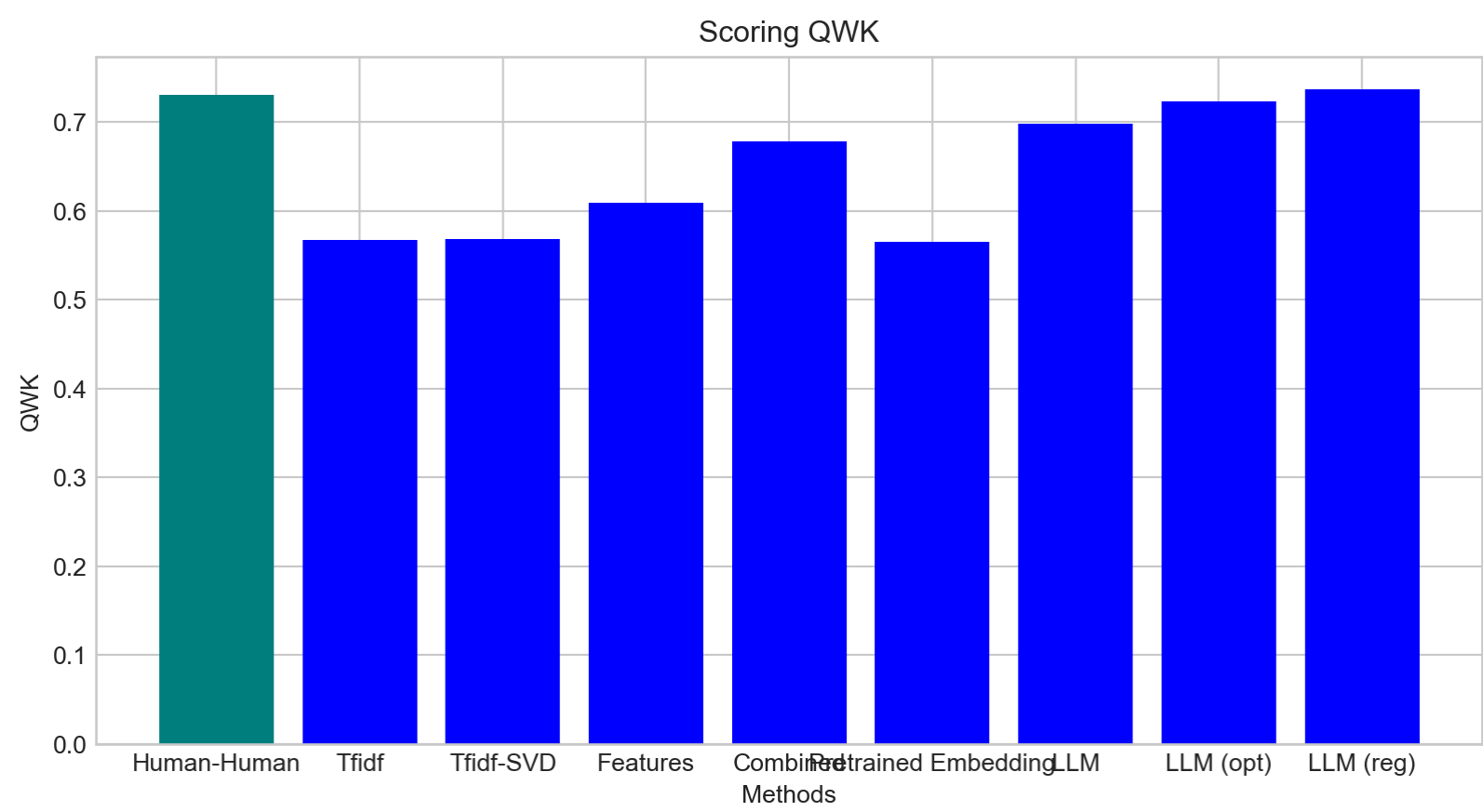
This may be complicated but using this code

```
{'QWK': 0.7367987733941068, 'Acc': 0.21019108280254778, 'SMD': np.float64(0.2515459013194259)}
```

while still using a very small model.

Results so far

So we have a model that performs above the human-human agreement.



Extra Optimizations

The following could be used to improve the results further



- **Cutoff optimization:** In our regression, we assumed equal pieces. We can optimize where the cutoff points for each score are with respect to QWK.
- **Ensemble:** Most models benefit from being combined with other models. The BoW and Transformer based models usually ensemble well.
- **Hyperparameters:** We have chosen $5e-5$ as our learning rate, but this is by no means optimal. Other variables like batch size and dropout can be applied.
- **Scaling:** We used a 53M parameter language model, but this is one of the smallest models available. We can always increase the model size to improve performance.

Generative Models

Calling generative models



There are two ways we will be calling generative models:

- API-based:
- Local Models:

Because this workshop is meant to be completely free, I will be using two free (limited) tools to use these models.

API-based: Cerebras and Groq

Cerebras and Groq are two prominent companies in the AI hardware space, each with a radically different architectural philosophy aimed at solving distinct problems in artificial intelligence workloads. While both create specialized processors that challenge traditional GPUs, they are optimized for different tasks.

Core Philosophy and Architecture

Cerebras: “Go Big” with Wafer-Scale Integration Cerebras’s core innovation is the **Wafer-Scale Engine (WSE)**, the largest chip ever built. Instead of dicing a silicon wafer into hundreds of individual chips, Cerebras uses the entire wafer as a single, massive processor.

- **Architecture:** The latest WSE-3 contains 4 trillion transistors and 900,000 AI-optimized compute cores on a single piece of silicon.
- **Problem Solved:** This design eliminates the primary bottleneck in large-scale AI: communication latency between individual chips. By keeping all compute cores and memory on one substrate with a high-speed interconnect fabric, data transfer is orders of magnitude faster than in a cluster of interconnected GPUs.
- **Analogy:** If a traditional GPU cluster is like a city with thousands of cars (data packets) stuck in traffic moving between buildings (chips), the Cerebras WSE is like having the entire city inside one massive, interconnected skyscraper.

Core Philosophy and Architecture

Groq: “Go Fast” with Deterministic Execution Groq’s innovation is the **Language Processing Unit (LPU)**, a chip designed from the ground up for speed and predictability, particularly for AI inference.



- **Architecture:** The LPU is a single-core, tensor-streaming processor. Its design is intentionally simple and relies on a software-first approach. The compiler schedules every instruction and data movement in advance.
- **Problem Solved:** This results in **deterministic execution**, meaning a computation takes the exact same amount of time every time it runs. It eliminates the unpredictability and overhead of traditional hardware (like caches and schedulers), leading to extremely low latency.
- **Analogy:** If a GPU is like a chaotic workshop with many workers grabbing tools as needed (dynamic scheduling), the Groq LPU is a perfectly synchronized assembly line where every movement is pre-planned for maximum speed and efficiency.

Primary Use Case

This is the most significant point of differentiation between the two companies.



- **Cerebras: Excels at Training Massive AI Models.** The WSE's vast compute power and unparalleled memory bandwidth make it ideal for training foundational models with trillions of parameters from scratch. It dramatically reduces the time required for training runs that would otherwise take weeks or months on large GPU clusters. While capable of inference, its primary strength is in large-scale training and research.
- **Groq: Excels at Ultra-Low Latency Inference.** The LPU is purpose-built to run already-trained models and deliver responses as quickly as possible. This is critical for real-time applications like conversational AI, live translation, and other services where speed is paramount. Its architecture is specifically tuned to maximize tokens-per-second output for large language models (LLMs).

Calling the models

```
from google.colab import userdata
import groq

groq_client = groq.Client(api_key = userdata.get('groq_key'))

def get_llm_response(user_prompt):
    chat_completion = groq_client.chat.completions.create(
        messages=[
            {
                "role": "user",
                "content": user_prompt,
            }
        ],
        model="openai/gpt-oss-20b",
```



We start with the following question:

```
question = """A water tank is being filled by two pipes.
```

```
Pipe A can fill the tank in 4 hours, and Pipe B can fill it in 6 hours.
```

```
There is also a drain, Pipe C, that can empty the full tank in 3 hours.
```

```
If all three pipes are opened simultaneously when the tank is empty, how long will it take for the tank to be filled?
```

```
Please show your step-by-step reasoning."""
```


Calling the LLM

```
print(get_llm_response(question))
```



Results in

Step 1 – Convert each pipe’s rate to “tanks per hour.”

Pipe	Time to fill (or empty)	Rate (tanks/hour)
A	4 h	(1/4)
B	6 h	(1/6)
C (drain)	3 h to empty a full tank	(1/3) (negative sign for draining)

Step 2 – Add the fill rates, then subtract the drain rate.

$$\text{Net rate} = \frac{1}{4} + \frac{1}{6} - \frac{1}{3}$$

Find a common denominator (12):

$$\frac{1}{4} = \frac{3}{12}, \quad \frac{1}{6} = \frac{2}{12}, \quad \frac{1}{3} = \frac{4}{12}$$

Continued

$$\text{Net rate} = \frac{3}{12} + \frac{2}{12} - \frac{4}{12} = \frac{1}{12} \text{ tank per hour}$$

So together they move water in at a net pace of one-twelfth of the tank per hour.

Step 3 – Find the time needed to fill one tank.

If the net rate is $(1/12)$ tank/hour, the time (t) to fill one full tank satisfies

$$\frac{1}{12} t = 1 \quad \implies \quad t = 12 \text{ hours.}$$

Result

All three pipes open simultaneously from an empty tank will fill the tank in **12 hours**.

This is the working of “openai/gpt-oss-20b”

Cerebras

Using a similar API

```
from cerebras.cloud.sdk import Cerebras
from google.colab import userdata

cerebras_client = Cerebras(api_key=userdata.get("cerebras_key"))

def get_llm_response(user_prompt):
    chat_completion = cerebras_client.chat.completions.create(
        messages=[
            {"role": "user", "content": user_prompt,}
        ],
        model="llama-4-scout-17b-16e-instruct",
    )
    return chat_completion.choices[0].message.content
```

we call the same function

```
print(get_llm_response(question))
```



Llama 4 Scout Solution

Step 1: Determine the rate of filling for each pipe and the drain.

The rate of filling for Pipe A is ($\frac{1}{4}$) of the tank per hour, since it can fill the tank in 4 hours. The rate of filling for Pipe B is ($\frac{1}{6}$) of the tank per hour, as it can fill the tank in 6 hours. The rate of emptying for Pipe C (the drain) is ($\frac{1}{3}$) of the tank per hour, since it can empty the tank in 3 hours.

2: Calculate the net rate of filling the tank when all pipes are open.

To find the net rate of filling the tank when all three pipes are open, we add the rates of Pipe A and Pipe B, and then subtract the rate of Pipe C (since it's emptying the tank). The net rate (R) is given by:

$$R = \frac{1}{4} + \frac{1}{6} - \frac{1}{3}$$

Continued

3: Perform the calculation for the net rate.



First, find a common denominator for (1/4), (1/6), and (1/3). The least common denominator is 12. Convert each fraction:

$$\frac{1}{4} = \frac{3}{12}$$

$$\frac{1}{6} = \frac{2}{12}$$

$$\frac{1}{3} = \frac{4}{12}$$

Then, calculate the net rate:

$$R = \frac{3}{12} + \frac{2}{12} - \frac{4}{12}$$

$$R = \frac{3 + 2 - 4}{12}$$

$$R = \frac{1}{12}$$

Local Models

Running language models (LLMs) locally, rather than relying on cloud-based APIs, is fundamentally important for ensuring robust security and privacy. When you use an external service, you are sending your data to a third party, creating inherent risks that are eliminated with a local approach.

Key Security Advantages

- **Complete Data Control:** The most significant benefit is that your data never leaves your machine or private network. When you send a prompt containing proprietary code, a sensitive legal document, or a confidential business strategy to a cloud API, that information is transmitted over the internet and processed on someone else's servers. A local model keeps all data within your trusted environment.
- **Reduced Attack Surface:** By processing data locally, you eliminate the risk of data interception during transit (a “man-in-the-middle” attack). You also remove the third-party's infrastructure as a potential point of failure or attack. The security of your data is entirely dependent on your own network's security, which you control.
- **Immunity to External Breaches:** If a cloud AI provider suffers a data breach, any data you have sent them could be compromised. With a local model, you are completely insulated from the security failures of external companies.

Huggingface Pipelines

Pipelines also simplify the running of local models:



```
from transformers import pipeline
pipe = pipeline("text-generation", model="LiquidAI/LFM2-350M-Math")
chat = [{"role": "user", "content": question}]
pipe_out = pipe(chat, max_new_tokens=2048)
from print(pipe_out[0]['generated_text'][1]['content'])
```

Output

Okay, let's see. So there's this water tank being filled by three pipes: Pipe A, Pipe B, and Pipe C. Pipe A can fill it in 4 hours, Pipe B in 6 hours, and Pipe C can empty it in 3 hours. We need to figure out how long it will take to fill the tank when all three are opened at the same time. Hmm, right.

So, let's start with Pipe A. It takes 4 hours to fill the tank. Therefore, its rate is 1 tank per 4 hours, which is $\frac{1}{4}$ per hour. Similarly, Pipe B takes 6 hours, so its rate is $\frac{1}{6}$ per hour. Pipe C takes 3 hours to drain, so its rate is $-\frac{1}{3}$ per hour (negative because it's emptying).

Now, adding up their rates: $\frac{1}{4}$ (from A) + $\frac{1}{6}$ (from B) + $(-\frac{1}{3})$ (from C). Let me calculate that. To add these fractions, I need a common denominator. The denominators are 4, 6, and 3. The least common denominator here is 12.

Converting each fraction:

$$\frac{1}{4} = \frac{3}{12},$$

$$\frac{1}{6} = \frac{2}{12},$$

$$-\frac{1}{3} = -\frac{4}{12}.$$

Adding them together: $\frac{3}{12} + \frac{2}{12} - \frac{4}{12} = \frac{(3 + 2 - 4)}{12} = \frac{1}{12}$.

****Time to fill the tank**:**

$$\begin{aligned} & \left[\right. \\ & \frac{1}{\text{Combined rate}} = \frac{1}{\frac{1}{12}} = 12 \text{ hours} \\ & \left. \right] \end{aligned}$$

****Answer**:** $\boxed{12}$ hours.

Scoring Using Generative Models

The Challenge: Fine-Tuning Giant Models

Fully fine-tuning a large language model (LLM) like GPT-3 or Llama involves updating every single one of its billions of parameters for a new task. This process is incredibly expensive and inefficient.

- **High Computational Cost:** It requires a massive amount of GPU memory and processing power.
- **Large Storage Needs:** For each new task, you have to store a complete, multi-gigabyte copy of the fine-tuned model. This becomes impractical if you need to adapt the model for dozens of different tasks.

The Solution: Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) methods solve this problem by freezing the vast majority of the pre-trained model's parameters and only training a very small number of new or existing parameters. This makes fine-tuning much more accessible and manageable.

Low-Rank Adaptation (LoRA) is one of the most popular and effective PEFT techniques.

Enter LoRA: Low-Rank Adaptation

The core insight behind LoRA is that the change in the model's weights during fine-tuning (the “update matrix”) can be effectively approximated using a much lower-rank representation.

How It Works

Instead of directly updating the original weight matrix W of a layer, LoRA introduces two small, trainable “adapter” matrices, A and B .



1. **Freeze Original Weights:** The pre-trained model weights (W) are kept frozen and are not updated during training.
2. **Inject Adapters:** For a specific layer (like the attention layers in a Transformer), LoRA injects the two smaller matrices, A and B , alongside the original weight matrix W . The product of these matrices, BA , creates a low-rank approximation of the weight update that full fine-tuning would have learned.
3. **Train Only Adapters:** During training, only the parameters of matrices A and B are updated.

The mathematical representation for a layer’s forward pass is modified from $h = Wx$ to:

$$h = Wx + \Delta Wx = Wx + BAx$$

Where: * $W \in \mathbb{R}^{d \times k}$ is the frozen, pre-trained weight matrix. * $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ are the small, trainable LoRA matrices. * The **rank**, r , is a hyperparameter that is typically very small (e.g., 4, 8, 16), so $r \ll \min(d, k)$.

Inference

A major advantage of LoRA is that it introduces **zero additional inference latency**. Once training is complete, the learned adapter weights can be merged directly into the original weights:

$$W_{new} = W + BA$$

You can then deploy this merged model just like the original, with no extra calculations needed during inference.

Key Advantages of LoRA

- **Drastically Reduced Trainable Parameters:** Reduces the number of trainable parameters by up to 10,000 times, significantly lowering memory and computational requirements.
- **No Inference Latency:** The adapter weights merge with the base model, so it's just as fast as the original model for inference.
- **Efficient Task Switching:** Instead of storing a full model copy for each task, you only need to save the tiny LoRA adapter weights. To switch tasks, you can simply load the base model and apply the desired adapter.

Coded Example

Obtaining a LoRA model is simple

```
from transformers import (AutoModelForCausalLM,
                          AutoTokenizer,
                          Trainer,
                          TrainingArguments)

from peft import (LoraConfig,
                  get_peft_model)

model = AutoModelForCausalLM.from_pretrained("LiquidAI/LFM2-350M")
tokenizer = AutoTokenizer.from_pretrained("LiquidAI/LFM2-350M")

config = LoraConfig(
    r=16,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj", "k_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM")

model = get_peft_model(model, config)
```

The data is the tricky part

Prompt Template



```
def chat_template(essay, score, min_score, max_score):
    chat = [{"role": "system", "content": "You are an essay grader."},
            {"role": "user", "content": f""""Provide a score between {min_score} and {max_score}. Your answer should be of the form

**Score**: [score]

**Essay**:
{essay}"""}],
            {"role": "assistant", "content": f""**Score**: {score}""}]
    return chat
```

Once this template is applied, everything flows naturally:

```
from trl import (SFTTrainer,
                 SFTConfig)

train_texts = [{'text': tokenizer.apply_chat_template(chat_template(essay, score, min(train['rater1_domain1']), max(train['rater1_domain1'])),
                                                       tokenize=False)} for essay, score in zip(test['essay'], test['rater1_domain1'])]

train_dataset = datasets.Dataset.from_list(train_texts)

args = SFTConfig(
    output_dir="output",
    save_strategy="no",
    learning_rate=2e-3,
    report_to="none",
    per_device_train_batch_size=1)

trainer = SFTTrainer(
    model=model,
    args=args,
    train_dataset=train_dataset)
trainer.train()
```

Scoring using GLMs

We simply loop through and score:

```
scores = []

from tqdm import tqdm

for x in tqdm(test['essay']):
    scores.append(score(x, 0, 12))

from sklearn.metrics import cohen_kappa_score

qwk = cohen_kappa_score(test['rater1_domain1'], scores, weights="quadratic")
print(qwk)
```

QWK = 0.6926505755917152

This is just the beginning.

This is not the end, this is just the beginning. With generative models, this is the minimum performance. We can adjust for the following: - Add Rubric information. - Add features. - Add examples to the text. - Chain of Thought prompting. - Prompt optimizations.



If I got to this point, I went too fast!!

References

- Rodriguez, P. U., Jafari, A., & Ormerod, C. M. (2019, September). Language models and automated essay scoring [arXiv preprint arXiv:1909.09482]. <https://arxiv.org/abs/1908.09079>
- Ormerod, C. M., Malhotra, A., & Jafari, A. (2021, February). Automated essay scoring using efficient transformer-based language models arXiv preprint arXiv:2102.13136
- Ormerod, C. M. (2022). Mapping between hidden states and features to validate automated essay scoring using deberta models. *Psychological Test and Assessment Modeling*, 64(4), 495-526.
- Ormerod, C., & Kwako, A. (2024) Automated Text Scoring in the Age of Generative AI for the GPU-poor arXiv preprint arXiv:2408.01873