Facultad de Ciencias

Training a Large Language Model for Standard Name Generation in Climate and Forecast Metadata

Trabajo de Fin de Grado

para acceder al

GRADO EN FÍSICA

Autor: Mario Díez Fernández

Director: Antonio S. Cofiño González

Codirector: Sadie L. Bartholomew

Date June 2025

This project is dedicated to my parents for their unconditional support throughout my college years, and to my director, Antonio, for all his guidance and support during this project.

Resumen

El objetivo de este proyecto radica en la creación de un modelo de lenguaje (*Large Language Model*, LLM) para la generación automática de nombres estándar (*standard names*) empleados en las *Climate and Forecast Conventions* (CF), que son utilizados en el ámbito de las geociencias para estandarizar los metadatos y así mejorar la interoperabilidad y el intercambio de datos. Estos *standard names* son etiquetas que definen las variables dentro de un conjunto de datos.

El objetivo de estos *modelos del lenguaje*, es automatizar la creación de *standard names*, minimizando errores humanos, y permitiendo ampliar tanto la especificidad, como la cobertura del vocabulario disponible. Este proyecto estudia adaptar un *modelo de lenguaje* preentrenado para que sea capaz de generar *standard names* a partir de descripciones de parámetros o variables físicas, altamente especializadas, en las que diferencias sutiles en la terminología pueden implicar una semántica física, significativamente distintas.

Palabras clave: 'Large Language Models', 'Climate and Forecast Conventions', 'Standard Names', 'Ciencia de Datos', 'Inteligencia Artificial'

Abstract

The objective of this project is the development of a Large Language Model (LLM) for the automatic generation of standard names used in the Climate and Forecast Conventions (CF). These conventions are widely employed in geosciences to standardize metadata in order to improve interoperability and data exchange. Standard names serve as labels that precisely define the variables within a dataset.

This language model aims to minimize human error in the creation of standard names and to expand both the specificity and coverage of the existing vocabulary. The project explores the adaptation of a pretrained language model—originally designed for programming language tasks—to generate standard names from highly specialized physical descriptions, where subtle differences in terminology may correspond to significantly different physical meanings.

Key words: 'Large Language Models', 'Climate and Forecast Conventions', 'Standard names','Data Science', 'Artificial Intelligence'

Contents

1 Introduction

The Climate and Forecast Conventions (CF Conventions) are a standard for organizing scientific data in network common data form (netCDF) files to facilitate their reading, analysis, and automated processing, especially in climate and weather numerical modeling and observation. These CF Conventions allow for the integration of observational and simulated data, ensuring interoperability, long-term compatibility, and scientific reproducibility. The Chapter 2, introduces netCDF, and its data model, which is the standard data format used by the CF community to exchange data.

One of the most important aspects of these metadata is the clear differentiation between elements, which ensures that datasets are correctly interpreted and used in scientific research. To this purposes CF Conventions define standard names for describing physical parameters and its relationships between elements of a netCDF components to describe structural metadata of the data itself. The Chapter 3, introduces the CF Conventions and it's most important elements including standard names and its data model, related to netCDF data model.

In the current landscape, marked by the rise of fields like artificial intelligence (AI), and in particullary the Natural Language Processing tecnologies based on neural networks. A neural network is a machine learning model that mimics the human brain's function by using interconnected nodes to identify patterns, assess data, and make predictions[1].

Multiple layers of connected nodes, sometimes referred to as artificial neurons, make up a neural network. An input layer, one or more hidden layers, and an output layer are among them. Every node has a weight and threshold and is connected to other nodes. A node is activated and sends data to the next layer when its output exceeds its threshold. The signal is not transmitted if the output falls below the threshold.

As neural networks process more information, they gradually improve their accuracy by learning from training data. They can quickly classify and organize data after being taught, making them extremely useful tools in computer science and artificial intelligence (AI). They are particularly useful in fields like voice and picture recognition, where tasks that previously required hours of human expertise can be finished in minutes due to their speed and efficiency. The search algorithm used by Google [2] is a well-known illustration of a neural network in action.

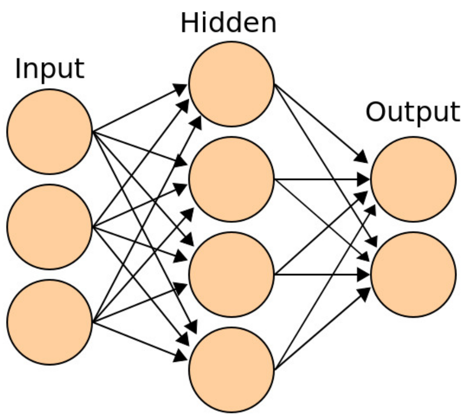


Figure 1.1: A schematic structure of a neural network. The circles represent the nodes and the arrows represent connection and data flow among processing units (Source [3])

John Hopfield created Hopfield networks, a kind of recurrent neural network, in the 1980s [4]. They are mostly used to simulate associative memory. They draw inspiration from how neurons behave collectively in the brain [5]. John Hopfield received the Nobel Prize in Physics in 2024, for which the prize motivation was “for foundational discoveries and inventions that enable machine learning with artificial neural networks”(Source [6]).

Hopfield networks function by using a network of connected neurons to store patterns as stable states. Every neuron is linked to every other neuron; the weights of these connections dictate how they affect each other. Associative recall is the process by which the network iteratively changes the neuronal states in response to a partial or noisy version of a remembered pattern until it converges to the closest stored pattern.

The idea of energy reduction is essential to their operation. In order to lower a global energy function, the network modifies the states of its neurons until it reaches a minimum that corresponds to a recorded memory. This enables Hopfield networks to carry out content-addressable memory tasks, which, like human memory, enable them to recall comprehensive information from incomplete input [5].

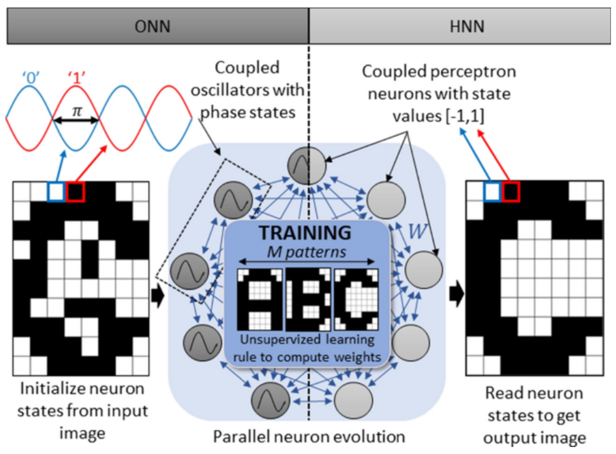

ONN
HNN
'0'
'1'
π
Coupled oscillators with phase states
Coupled perceptron neurons with state values [-1,1]
TRAINING
M patterns
W
Unsupervized learning rule to compute weights
Initialize neuron states from input image
Parallel neuron evolution
Read neuron states to get output image

Figure 1.2: How Hopfield's neural networks work (Source [7])

Natural Language Processing (NLP) is a field of computer science that enables computers to interpret language in a more human-like manner. A subtype of NLP is the large language model (LLM), which is trained on vast amounts of textual data using sophisticated neural networks. These models will be discussed in more detail in Chapter 4.

This project aims to develop an AI-based tool for the automatic generation of CF standard names, with a particular focus on enabling the model to understand the physical principles represented by these labels.

Finally, an experimental setup for modeling CF standard names using LLMs is described in Chapter 5, with conclusions presented in Chapter 6.

2 The Network Common Data Form

The network common data form (netCDF) is a primary data management system for scientific information, which is vital in computer physics and climate and geophysical modeling and observations. The most valuable aspect of netCDF is its portability, meaning it can be reasonably, accessed and processed across a host of different computer architectures. This is necessary for fine-grained numerical simulations [8]. Then adding metadata of variables, physical units, and spatial-temporal coordinates. The netCDF data-model enables a way of data self-descriptiveness, which is necessary in scientific fields and in particular in physics for precise data interpretation datasets.

The climate and forecast conventions (CF or CF Conventions) [9] provides a standardized metadata framework to improve interoperability by guaranteeing that datasets include rich contextual information in addition to plain numerical values. In physics-based study fields including fluid dynamics, thermodynamics, atmospheric physics, and oceanography, provides precise spatial-temporal references, physical units, and physical variables definitions. By providing a standardized encoding method, CF conventions facilitate data consistency and comparability across different numerical models and observational datasets.

2.1 Classic Model

The classic netCDF data model is built around three main components: variables, dimensions, and attributes. This foundational approach to structuring data was established in the very firsts versions of netCDF and continues to serve as the backbone of all netCDF data products today. They consist of the following elements:

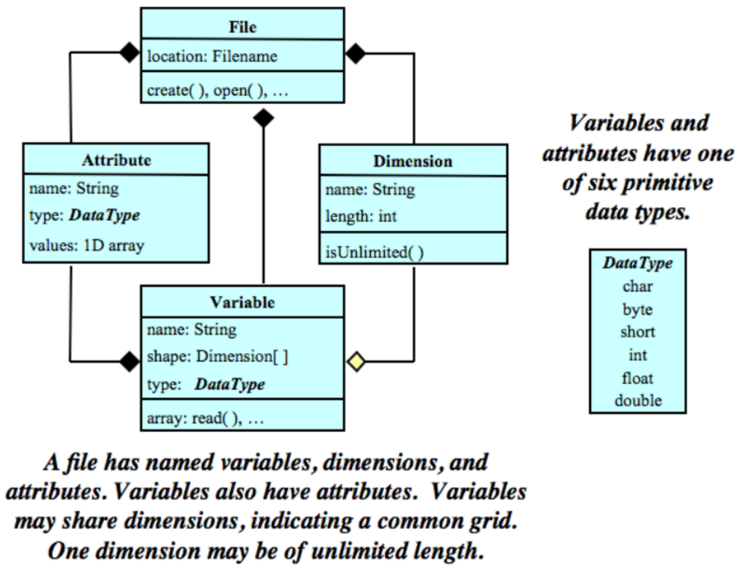

File
location: Filename
create( ), open( ), …
Attribute
name: String
type: DataType
values: 1D array
Dimension
name: String
length: int
isUnlimited( )
Variable
name: String
shape: Dimension[ ]
type: DataType
array: read( ), …
Variables and attributes have one of six primitive data types.
DataType
char
byte
short
int
float
double
A file has named variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One dimension may be of unlimited length.

Figure 2.1: netCDF's classic data model (Source [10])

- **Variable**: Multidimensional Array of data. The netCDF variable types can be one of six defined formats representing basic numerical types
- **Dimension**: In netCDF data multidimensional arrays, its sizes on each dimension are labeled as common dimension type and values, each with a name and length. There can be one unlimited dimension that grows as data is added to the file. Each netCDF file supports only a single unlimited dimension.
- **Attribute**: Attribute in netCDF add brief metadata to variables or the file itself (global attribute). They are scalar or 1D vector and typically kept small [10].

```
netcdf example {   // example of CDL notation
dimensions:
        lon = 3 ;
        lat = 8 ;
variables:
        float rh(lon, lat) ;
                rh:units = "percent" ;
                rh:long_name = "Relative humidity" ;
// global attributes
        :title = "Simple example, lacks some conventions" ;
data:
 rh =
  2, 3, 5, 7, 11, 13, 17, 19,
  23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89 ;
}
```

Figure 2.2: Example of common data language notation describing a simple netCDF dataset (source [11])

.

This common data language (CDL) describes a simple netCDF dataset called example, where we can see the three parts of the netCDF classic model.

As it's shown, the dimensions block defines two sized dimensions, 'lon' (longitude) with size 3 and 'lat' (latitude) with size 8, defining together a 3x8 grid. In the variables block declares a 2D float variable 'rh' (relative humidity), using the 'lon' and 'lat' dimensions, including two variable-level attributes, 'units=”percent”' that shows how the variables are measured and 'long_name = “Relative humidity”' which is a human-readable description.

After that, the attributes block which is a global attribute for documentation or metadata in this case ':title= Simple example, lacks some conventions”' which means that the file does not follow the standard conventions like climate and forecast conventions. And, finally, the data section, which provides 24 data values for rh (corresponding to the 24 values of the 3x8 grid).

2.2 The netCDF Enhanced Data Model

The netCDF data model represents our conceptual framework for organizing and interpreting data [10]. Originally, the classic model, based on just dimensions, variables and attributes, formed the foundation of this approach. With the release of netCDF version4.0 (netCDF4), the model was extended to create an enhanced data model (see Figure 2.2). This extended model maintains full backward compatibility with the classic model (i.e. netCDF3) while introducing powerful new features, including groups, support for multiple unlimited dimensions, and user-defined data types.

When the purpose of the data is to ensure the highest level of compatibility with existing software applications or tools, it is recommended, when possible, that new datasets be created using the simply classic model.

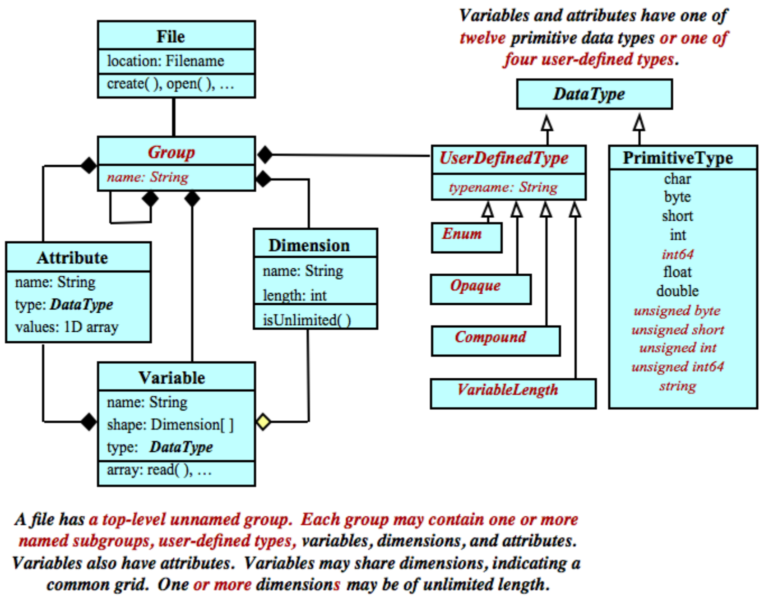


Figure 2.3: The netCDF Enhanced Data Model (Source [10])

3 The Climate and Forecast Conventions

The primary advantage of Climate and Forecast (CF, or CF Conventions) is the automation of data processing and model connection in CF [9]. Scientific computing software already knows how to read this structure when a dataset satisfies CF standards, allowing for rapid visualization, numerical methodology and large-scale simulations with little manual work required. They are also designed to accommodate a number of spatial and temporal representations compatibly with each other, in different modeling scenarios.

CF Conventions are especially useful in the context of climate modeling, atmospheric physics or numerical weather prediction, where structured, interpretable, and fast datasets need to be maintained. Aside from climate models, they have also been used for observational data now as well as station-based data, profile soundings, and satellite data, enabling researchers to cross-check theoretical models against empirical measurements. Such a combination of modeled and observed data is fundamental to understanding intricate physical systems, such as atmospheric circulation, ocean currents, and planetary boundary layers.

Using these rules for ordering metadata provides data interoperability, process automation and long-term data compatibility with legacy systems that CF conventions enable. Crucially, in computational physics (long-lived simulations; geophysical models), this is important to keep the data structures usable over time. The interoperability between numerical models and observational datasets will allow scientists to step seamlessly into CF-compliant workflows, preserving interoperability of existing tools and applications.

In the end, the CF Conventions (extend netCDF just enough to be useful) give a firm standard, and so do CF conventions on top of them. CF lets you manage, share and analyse complex physical datasets with confidence. Which in turn ensures that numerical simulations, geophysical analysis and climate models can always be reproduced and as scientific rigorous as possible. These conventions help to ensure that the physical datasets are used to do physics and climate advancing atmospheric dynamics, oceanography, and Earth system modeling, using high precision coupled with the notional gap between theoretical research and observatory experiences from physicists to climatologists.

3.1 Principles for designing

The CF conventions aim to enable datasets to be self-descriptive, datasets that can be understood on their own without any external references. Rather than codes, they use controlled vocabulary with simple, self-documenting terms and fuller definitions in CF documentation.

The cost of changing the CF conventions mostly comes from how they are used in real-world situations, so changes are only made when truly necessary. To keep things easy to read and use, the rules for new standard names follow a clear structure, and the vocabulary is based on the CF data model.

Metadata is always written to be both human-readable and easy for software to process. For this reason, the conventions focus on being user-friendly for both data creators and users. They also aim to avoid repeating information to reduce errors and prevent inconsistencies.

Instead of telling people what data they must collect, CF standards give flexibility, most features can be used with or without CF. To make sure data stays usable over time, metadata created with older versions still works with newer versions.

New features are only added when existing methods can't do the job, so older datasets can continue to be supported [9].

3.2 Overview

All netCDF users, regardless of the software tools or libraries they use, can find a thorough description of their concepts in the NetCDF User's Guide (NUG, [12]). It contains a lot of high-level overviews, which is especially beneficial for data users, producers, managers or tool developers working with netCDF files. Also, the NUG contains voluminous specifications for developers and maintainers of netCDF data model implementations.

The 'Conventions' global attribute in a netCDF file specifies the standards or conventions that the dataset follows, usually within the context of the Unidata [13] ecosystem, Unidata is the organization that develops and mantains the CF conventions. This attribute contains a string of one or more convention names, separated by spaces or commas.

Unidata provides a page where many conventions are registered [14]. However, the list found there is neither complete nor regularly updated. Some of the conventions listed are obsolete or no longer in use, and there are others in active use that do not appear on the list. As a result, this page should not be viewed as a fully authoritative reference for netCDF conventions. At present, no single official or comprehensive source exists.

Conventions are typically accompanied by documentation and examples that define how netCDF files should be structured according to that specific standard [9].

The 'Conventions' global attribute in a netCDF file identifies the data standards or structural rules the file follows, generally from the Unidata community. These conventions guide how variables, dimensions, metadata, and units are described within the file to ensure consistency and interoperability.

The attribute itself is a string that may list one or more conventions, separated by spaces or commas. These conventions define expected practices for naming, organizing, and documenting data.

Unidata maintains a page listing many of these conventions. However, that list is not comprehensive or regularly maintained. Many of the conventions included there are outdated or no longer in common use, while some widely used modern conventions are missing altogether. For this reason, although the page is a helpful starting point, it cannot be considered a definitive or authoritative source. Unfortunately, no such complete reference currently exists. Each convention is typically associated with its own documentation and usage examples that demonstrate how a netCDF file should be structured under that standard.

UDUNITS-2 [15], used within some conventions, is a system and software library that defines and converts physical units of measurement in a standardized way.

Latitude, longitude and time are available standard names, specifying the location on the Earth's surface of data values at a moment in time. However, a single vertical coordinate is not always enough to locate a data value vertically. The 'standard_name' and 'formula_terms' attributes are used to establish a mapping from the values of the parametric vertical coordinate used in the dataset to the dimensional vertical coordinates for uniquely locating data on the surface of the Earth.

Occasionally, data observations are multidimensional cells, not single point values. The CF conventions provide a 'bounds' attribute to specify how big and exactly where these intervals or cells start and end, to describe the extent (in coordinate space) of extensive physical variables or fields (for example, accumulations or means). The NUG declares coordinate variables, although in practice, gridpoint data is often assumed to always be at the center of their cells. That is not how CF works. In the absence of bounds, the cell where this data point is located is not defined, so no assumptions can be made about its cell or extent.

Sometimes, the data is cell-based (which can be described with easily derived statistics such as mean or maximum), and this needs to be described with the 'cell_methods' attribute. This is especially helpful to document climatological and diurnal statistics.

Data can be packed to reduce size, which means reducing the data size by sacrificing precision. Values are stored with reduced precision by applying a chosen scaling and offset (see NUG[16] for more details), or it can be compressed while retaining full precision by omitting missing data.

3.3 The Climate and Forecast Conventions Data Model

Although the CF Conventions originally existed as a long descriptive document detailing the rules and structures of the convention and still serve that purpose, starting from version CF-1.11 a CF Data Model was introduced to provide a clearer and more formal representation of the underlying structure.

This data model helps standardize how datasets are understood conceptually, beyond just how they are encoded in netCDF. CF conventions are in place to facilitate the creation, interpretation, and interchange of climate and forecasting data as netCDF files. In order to assist in this goal, the CF Conventions specify an explicit data model that defines the conceptual structure of CF datasets in a way independent of netCDF's encoding.

The CF data model gives a standard and complete view of CF datasets. It leads the construction of future CF extensions, facilitates interoperability with other data models, and assists software developers in constructing CF-compliant tools and interfaces. By abstracting from encoding specifics, it also positions CF for possible use beyond netCDF.

A data model embodies the conceptual data structure; it assigns dataset elements, their scientific purpose, and their relationship to each other. It enforces rules and constraints that determine how metadata is associated with data so that it may be scientifically understood and processed meaningfully.

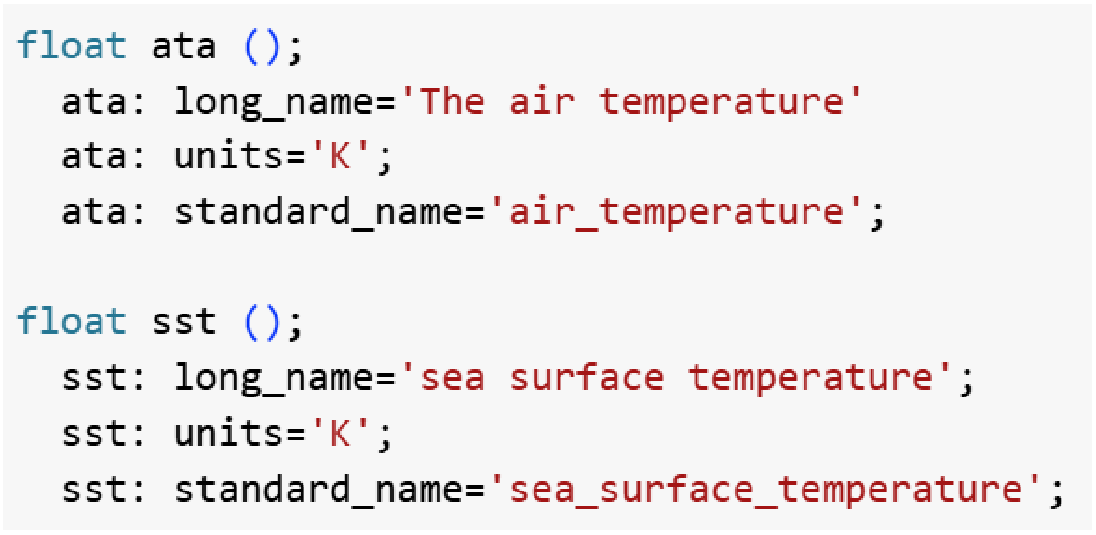
The data model was constructed based on several guiding principles:

- It should be able to describe any current or future CF-compliant dataset.
- It should only have the absolute minimum parts necessary to fully define CF concepts.
- It should not rely on netCDF-specific knowledge, so it is easier to use with other types of data in the future.

The CF data model's capacity to differentiate between seemingly identical data kinds is another especially remarkable characteristic. When working with values that have the same units and may appear to have the same physical meaning but actually represent completely different notions or measurements, this is particularly crucial.

For example, two quantities will have the same unit in meters, but one will be measuring the ocean bottom depth, and another will be measuring the surface height. In the practical world, they are inherently different quantities, although the units are the same and even the range of values is similar. To make these differences identified and maintained in the data set, the CF data model provides room for the required differences in metadata and structure.

Because CF promotes the use of descriptive metadata to supply context, not just standard names, units, axis definitions, and cell methods, but in ways that help make it even more understandable what a variable is referring to, this is particularly potent. In scientific data processing, where inference based on similarities at the surface level could result in false conclusions, this level of specificity is essential [9].



```
float ata ();
  ata: long_name='The air temperature'
  ata: units='K';
  ata: standard_name='air_temperature';

float sst ();
  sst: long_name='sea surface temperature';
  sst: units='K';
  sst: standard_name='sea_surface_temperature';
```

Figure 3.1: Example of two different temperature standard names

Looking at the figure, we notice two distinct attributes: one with the standard name 'air_temperature' and the other 'sea_surface_temperature'.

When we look for their definitions in the CF Standard Name Table [17]:

`standard_name:`	`air_temperature`
`canonical_units:`	`'K'`
`description:`	`Air temperature is the bulk temperature of the air, not the surface (skin) temperature. It is strongly recommended that a variable with this standard name should have a units_metadata attribute, with one of the values 'on-scale' or 'difference', whichever is appropriate for the data, because it is essential to know whether the temperature is on-scale (meaning relative to the origin of the scale indicated by the units) or refers to temperature differences (implying that the origin of the temperature scale is irrelevant), in order to convert the units correctly.`

While 'sea_surface_temperature':

standard_name:	sea_surface_temperature
canonical_units:	'K'
description:	Sea surface temperature is usually abbreviated as 'SST'. It is the temperature of sea water near the surface (including the part under sea-ice, if any). More specific terms, namely 'sea surface skin temperature', 'sea surface subskin temperature', and 'surface temperature' are available for the skin, subskin, and interface temperature respectively. For the temperature of sea water at a particular depth or layer, a data variable of sea water temperature with a vertical coordinate axis should be used. It is strongly recommended that a variable with this standard name should have a units metadata attribute, with one of the values 'on-scale' or 'difference'.

At first glance, both variables might appear to represent similar physical quantities. They both involve measurements of temperature, and they even share the same units, but they are not the same.

This example highlights the importance of relying on the standard name metadata, not just units or variable names, to accurately understand the role and meaning of each variable within a dataset. Careful attention to these distinctions is essential when working with CF-compliant data, ensuring that datasets are interpreted and used correctly in scientific research and applications.

3.3.1 Elements of CF-netCDF

In order to standardize the description of scientific data and facilitate uniform interpretation and usage in climatic and geospatial applications, CF-NetCDF [18] is a data format that combines NetCDF files with the CF Conventions.

The CF data model elements, which are intended to characterize climate and forecasting data, are the source of the CF data model. To reduce the number of pieces, remove netCDF specific aspects, and preserve the ability to fully define the CF conventions, the CF data model abstracts these elements (mentioned in Table I.1 from [7]) and their interrelationships (shown in Figure 3.1). This method guarantees that the model satisfies the design requirements of being both comprehensive and flexible.

Their elements are the following:

- **Domain Variable**: Locations in multi-dimensional space.
- **Data Variable**: Scientific data discretized within the domain.

- **Dimension**: Independent axis of the domain.
- **Coordinate Variable**: Unique coordinates for each axis.
- **Auxiliary Coordinate Variable**: Additional or alternative coordinates.
- **Scalar Coordinate Variable**: Coordinate for a size-one axis.
- **Grid Mapping Variable**: Horizontal coordinate system.
- **Boundary Variable**: Defines cell vertices.
- **Cell Measure Variable**: Describes cell areas or volumes.
- **Ancillary Data Variable**: Metadata related to the domain.
- **Mesh Topology Variable**: Describes related domains with cell connectivity.
- **Location Index Set Variable**: Defines domain with cell connectivity.
- **Formula Terms Attribute**: Vertical coordinate system.
- **Feature Type Attribute**: Describes the characteristics of sampling geometry.
- **Cell Methods Attribute**: Describes variation within cells.

Figure 3.1, it's been shown that the idea of an ”abstract generic coordinate variable” to refer to coordinates when the precise type (coordinate, auxiliary, or scalar) is irrelevant captures the relationships between these components. The CF data model maintains its flexibility by streamlining these relationships and eliminating netCDF-specific encoding, which allows it to enable possible future additions while properly describing CF conventions (for more details see [9]).

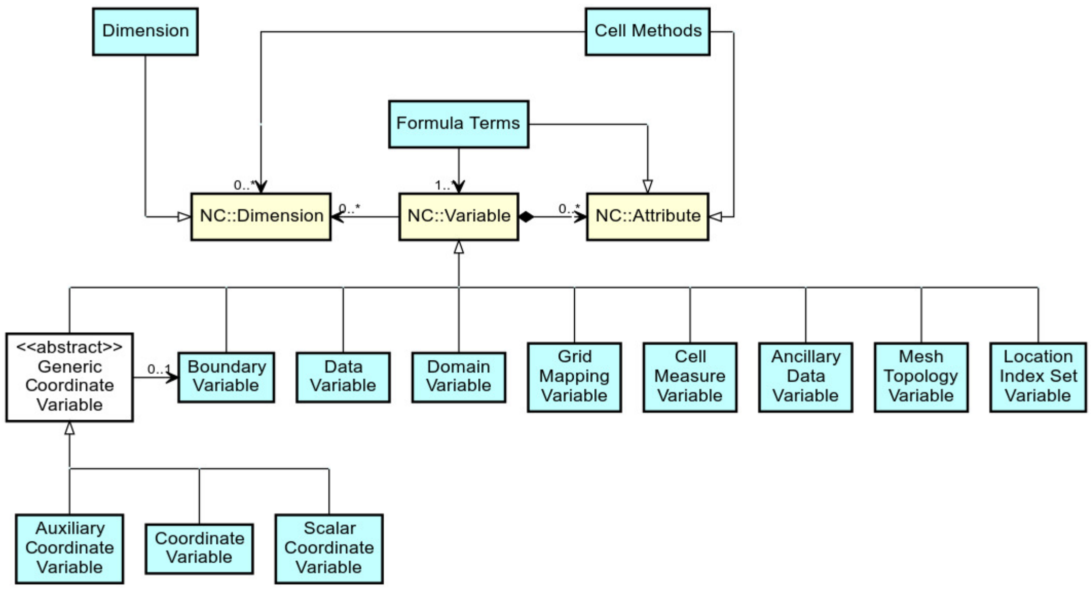


Figure 3.2: The relationships between CF-netCDF elements and netCDF variables are outlined, with an abstract generic coordinate variable introduced to simplify coordinate reference without specifying type (Source [9]).

3.3.2 The field construct

A CF data variable and all of its related metadata are equivalent to a field construct in the CF data model. Field ancillary constructs with metadata defined over the same domain, a data array, a domain construct that specifies the measurement locations and cell properties for the data, and cell method constructs that explain how cell values represent physical variations within the domain are its constituent parts. It also contains attributes that characterize features of the data that are not domain-specific.

The data array is the only required component; all other components are optional. Certain netCDF features of variables, like 'units, 'long_name', and 'standard_name', as well as group attributes and global attributes like 'history' and 'institution in the root group, the root group is the structure of an netCDF4 file, corresponding to the qualities of the field construct. The word 'property' is used instead of 'attribute' because not all CF attributes are regarded as properties in this context; certain attributes serve structural purposes in the file or point to other netCDF variables.

Unless they are overridden by attributes of the same name in individual variables, netCDF group attributes in the data model apply to all data variables in the file. This is required because group attributes' metadata needs to be moved to the field construct because the data model does not take the group notion into account. It is regarded as a property of the field construct and applies to all data variables in the file with a discrete sampling geometry if the global file attribute 'featureType' is present.

Additionally, only specific units are suitable with each standard name since the unit's property is constrained by the 'standard_name' property. Moreover, it could place restrictions on the size that a data variable needs to possess. Similarly, the domain's axes are subject to criteria from the 'featureType' attribute. Nevertheless, 'standard_name' and 'featureType' are not handled as distinct constructs within the field, in keeping with the model's simplicity objective, because these constraints are not dependent on other constructs for interpretation (more details and explanations at [9]).

3.4 Description of the data

The following attributes qualify the material and establish the measurement units of each component. The 'long_name' and 'units' attributes follow the Cooperative Ocean/Atmosphere Research Data Service (COARDS) standard. COARDS is complemented by the 'standard_name' attribute, which provides unique identifiers for variables. This is particularly useful in data exchange, since variable names may differ within institutions, thus making them less readily identifiable.

The 'standard_name' attribute can be employed in order to name variables carrying data. The applications employing such attributes will also have to define coordinate types.

3.4.1 The units attribute

The 'units' attribute is required for all variables representing dimensional quantities, except for those related to boundaries and climatology. For dimensionless quantities, the 'units' attribute is optional.

The value of this attribute is a string recognized by the UDUNITS-2 package, with some exceptions. Unit strings are case-sensitive. CF uses UDUNITS only to define valid units but does not require the UDUNITS software for unit conversion. Most unit conversions involve multiplying by a scale factor, with special handling needed for temperature units and time coordinate variables.

While the COARDS convention disallows the degrees unit, CF allows it when appropriate, such as for the solar zenith angle or transformed grid coordinates like latitude and longitude. In these cases, the coordinate values are not true latitudes and longitudes and must be identified with more specific forms of degrees.

3.4.2 The long name attribute

The 'long_name' attribute, as defined by NUG, provides a detailed descriptive name that can be used, for example, to label plots. For backward compatibility with COARDS, this attribute is optional. However, it is strongly recommended to include either this attribute or the 'standard_name' attribute, described in the following section, for all data variables and coordinate variables to make the file self-describing. If a variable does not have the 'long_name' attribute, an application may default to using the 'standard_name' (if available) or the variable name itself.

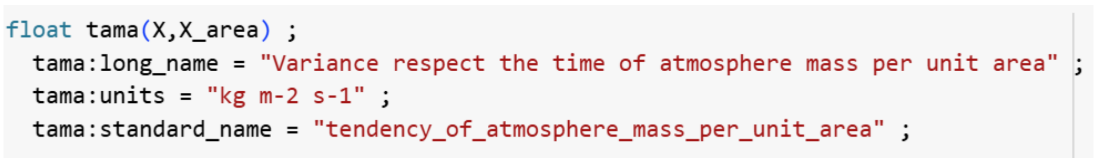
3.4.3 The standard name attributte

Accurately describing the physical quantities being represented is crucial for the exchange of scientific data. While the 'long_name' attribute contributes to this, it primarily serves as a means for reaching a goal. In many cases, providing a more precise description is far more beneficial, as it allows users from different sources to determine whether quantities are comparable. To ensure consistency, this convention assigns each variable a 'standard_name', providing a unified reference across datasets.

A standard name is associated with a variable via the 'standard_name' attribute, which contains a string value representing the standard name. This string may optionally be followed by one or more spaces and a standard name modifier, selected from a predefined list of modifiers.

Every standard name available is collected in the standard name table. Each entry must have the following attributes:

- **Standard name**: Name that qualifies the physical quantity. This standard name must not have any spaces between words.
- **Canonical units**: Units used to describe the physical quantity, unless it is a dimensionless magnitude. A variable with a standard name must have units that are equivalent to he canonical units
- **Description**: This defines the qualifiers for physical quantities, specifying aspects such as the surface on which a quantity is measured or the conventions for flux signs. While fundamental physical quantities have precise definitions in scientific literature, this description focuses on the rules governing variable types, attributes, and coordinates that must be adhered to by any variable assigned a standard name [9].



```
float tama(X,X_area) ;
  tama:long_name = "Variance respect the time of atmosphere mass per unit area" ;
  tama:units = "kg m-2 s-1" ;
  tama:standard_name = "tendency_of_atmosphere_mass_per_unit_area" ;
```

Figure 3.3: Example of a standard name usage

3.5 The standard name creation process

CF Standard Names Guidelines lay down some rules on how to name geophysical quantities used in science data [19]. The standards enable clarity and interoperability in the scientific data exchange, particularly in formats such as netCDF. CF standard names: a casing of lowercase words without special encoding. Text strings in quotes. The standards require CF standard names to be camelcase strings with underscores between words. Use American English (e.g., vapor instead of vapour). Wherever you see a name, it must be an unambiguous and unique physical quantity. Here, 'sea_surface_temperature' is the temperature on the sea surface and not 'air_temperature'.

The next is the general structure:

[Surface] [Component] standard_name [in medium] [due_to process] [assuming condition]

The measured quantity's geographic location is indicated by the surface. For instance, 'at_sea_level' denotes sea level, whereas 'at_surface' denotes the earth's surface. A vector's direction is specified by the component; for example, 'northward_wind' denotes the wind's northward component. The environment in which the quantity is measured, such as 'in_air' or 'in_sea_water', is described by the medium. The 'due_to_advection' term is an example of a quantity that may be broken down into its constituent parts using this approach. The condition, such as 'assuming_dry_adiabatic_lapse_rate', defines an assumption that is used to quantify the quantity.

Examples of standard names include 'air_temperature_at_2_m_above_ground_level', which refers to the air temperature at 2 meters above the ground level, and 'net_upward_longwave_radiation_flux_ in_sea_water_due_to_diffusion', which represents the net upward longwave radiation flux in sea water due to diffusion.

Additional rules include the use of 'net_' as a prefix to radiative fluxes to indicate the difference between incoming and outgoing radiation. The term 'content' is used for quantities that are vertically integrated, such as 'content_of_atmosphere_layer'. The term 'mass_fraction_of_X_in_Y' is used to describe the mass fraction of a component X in a medium Y. Units should not be included in the name; they are specified separately to avoid redundancy.

Any member of the scientific community can propose new standard names by submitting proposals to the discussion section on the Github. These proposals must include a clear justification and follow the established guidelines. If approved, they are incorporated into the list of standard names. These guidelines are essential for maintaining consistency and interoperability in the handling of scientific data, facilitating its use and understanding across disciplines and platforms.

3.6 Modifiers for standard names

A key requirement for the exchange of scientific data is the ability to accurately describe the physical quantities being represented. While the 'long_name' attribute allows some description, it is used informally and inconsistently. For applications that require clear and consistent definitions, a more standardized approach is needed. To meet this need, variables can optionally be associated with a 'standard_name' that uniquely identifies the quantity they represent.

The 'standard_name' attribute links a variable to a predefined name that describes a specific physical quantity. This name may optionally be followed by one or more modifiers that clarify the nature of the data, for example, indicating uncertainty or a rate of change. All valid standard names are listed in a controlled vocabulary called the standard name table.

When relevant, the table also provides mappings to other naming systems like GRIB codes and AMIP identifiers.

The standard name table is published online in both XML and formatted text versions. The formatted version can be used to look up appropriate names for variables. Some standard names are tied to predefined sets of allowable values, for instance, for variables that describe specific regions or area types.

In some cases, a standard name alone is not enough to fully describe a variable. If the data reflects operations like averaging over time or represents a measure of uncertainty, additional qualifiers can be used. These may appear as part of the 'standard_name' attribute or in other metadata such as 'cell_methods' [9].

3.7 Standard name relevance

When sending climate data in formats like netCDF, the CF protocols, which are discussed in this section, ensure consistency and compatibility. The 'standard_name' property is essential for enabling data exchange and cross-institution comparison since it gives variables distinct IDs.

By ensuring that physical quantities are sufficiently described, the 'standard_name' property ensures data comparability and consistency. Additionally, the efficiency of research and visualization is improved by automated data processing enabled by uniform nomenclature.

A standard name table that provides a clear reference for every variable is part of the CF conventions. While the rules for developing standard names guarantee accuracy and clarity, this aids in data integration and analysis. To sum up, standard names are essential for the correctness, consistency, and interoperability of climate data; they facilitate data sharing and foster cooperation in scientific study.

The CF conventions might be improved even more by adding a Large Language Model (LLM) that can produce standard names from descriptions. As previously stated, the CF norms are crucial for guaranteeing the uniformity and compatibility of climate data, particularly when it is exchanged in netCDF-like formats.

The development of these standard names may be automated by an LLM, which would expedite the procedure and lower human error, thus saving scientists and researchers time. A standard name table that provides a clear reference for every variable is part of the CF conventions. While the rules for developing standard names guarantee accuracy and clarity, this aids in data integration and analysis. To sum up, 'standard_names' are essential for the correctness, consistency, and interoperability of climate data; they facilitate data sharing and foster cooperation in scientific study.

4 Large Language Model

A type of deep learning model called large language models (LLMs) is made to comprehend and produce human language on a large scale. Text generation, translation, summarization, question answering, and other natural language processing (NLP) activities can be accomplished by these models, which are trained on large text corpora and acquire statistical patterns and semantic structures.

Large datasets and increasing computer resources have sped up the development of LLMs in recent years, making them effective instruments for everyday applications, business, and research. Their intricacy, resource requirements, and possible biases, however, also bring up significant ethical and technical issues.

4.1 Natural language processing

The interdisciplinary study of Natural Language Processing (NLP) combines machine learning and linguistics to understand all aspects of human discourse [20]. NLP does not just read words; it also examines the context and the relationships between words in communication.

The following list of typical NLP tasks includes examples:

- Sentence-level classification: Determining the sentiment of a product review, identifying whether an email is spam, checking grammatical correctness, or assessing whether two sentences are logically connected.
- Word-level classification: Assigning roles to each word in a sentence, such as identifying parts of speech (e.g., noun, verb, adjective) or recognizing named entities like people,

places, or organizations.

- Text generation: Producing text based on a prompt, such as continuing a sentence or filling in missing words.
- Answer extraction: Finding specific answers within a passage of text when given a question and related context.
- Text-to-text generation: Transforming text into another form, such as translating between languages or summarizing content.

NLP can be used for tasks in speech and vision in addition to written language. For example, it can convert spoken audio into text or generate a description based on an image.

One particular kind of NLP system is a Large Language Model (LLM). NLP, which broadly focuses on understanding and interpreting human language, includes LLMs as powerful tools. Large amounts of text have been used to teach them to do a range of language-related activities, including answering queries, summarizing texts, translating languages, and generating human-like responses.

4.2 Large language model

A Large Language Model (LLM) is a type of text-to-text model that learns patterns, structures, and relationships in language, enabling it to generate and understand text based on vast amounts of training data. It uses resources such as tokenization and embedding.

Before modeling text data, tokenization is required because models do not understand text directly; they process numbers, known as tokens. Choosing an effective tokenizer is essential to accurately capture the meaning of the text, which ultimately improves the performance of the model. The tokenizer splits the text into predefined segments such as characters,words or sentences, assigning a number to each. The resulting array is then processed by the embedding.

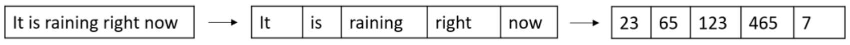


Figure 4.1: Example of tokenizer splitting the text in words

However, we cannot directly introduce raw numbers into the model, as static representations alone are not sufficient for capturing the true meaning of words. Embeddings are vectorial representations of words and characters in the text [21].

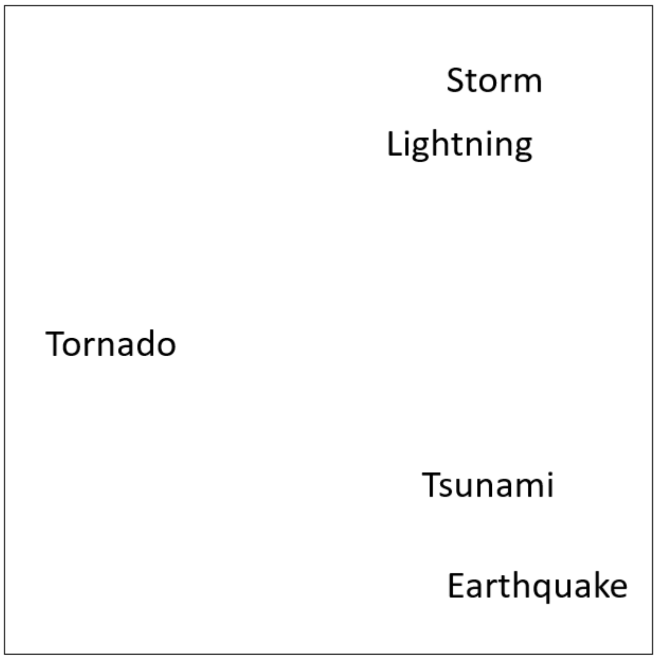


Figure 4.2: Example of embedding applied to weather forecast

As shown in Figure 4.2, we have a simple example of an embedding in a 2D space. Here, we can observe how embeddings work: words like *storm* and *lightning* are positioned close to each other, indicating a strong relationship. In contrast, their greater separation from the other words suggests weaker associations. Furthermore, we see that *tsunami* and *earthquake* are relatively close, though slightly farther apart than *storm* and *lightning*, implying that while they are related, their connection is not as strong as the first pair.

4.2.1 Tokenizer

Tokenization is a fundamental step in the operation of LLMs [21], transforming raw text into smaller units called tokens. These tokens can range from individual characters to subword fragments, depending on the tokenizer's design and the model's vocabulary. For example, the sentence “Tomorrow's weather looks unpredictable” might be tokenized into [”Tomorrow”, ” ’ ”, ”s”, ”weather”, ”looks”, ”un”, ”predictable”], allowing the model to process and interpret the text more effectively.

Rather than relying on traditional methods that split text based on spaces or punctuation, LLMs typically employ subword tokenization techniques. These approaches are especially effective at balancing vocabulary size and model generalization. They allow the model to handle uncommon or complex words by breaking them into known subcomponents, turning a word like “thunderstorm” into [”thunder”, ”storm”], for instance, which reduces the likelihood of encountering out-of-vocabulary tokens.

Tokenization is critical to model performance and efficiency [22]. Since LLMs do not interpret text at the word level but rather as sequences of tokens mapped to numerical vectors, the structure of these tokens influences how much context a model can handle. Most models are limited to a fixed number of tokens (e.g., 2048 or 4096), so poor tokenization can lead to hitting the limit early and losing relevant content. Efficient tokenization maximizes the meaningful information passed into the model and ensures coherent output generation.

Tokenization consistency is also essential for pretraining and fine-tuning. Even minor differences in token handling can have an impact on downstream performance because LLMs are trained on enormous datasets with billions of tokens. When reusing pretrained tokenizers for transfer learning or modifying models for other domains, this is particularly crucial.

4.2.2 Embedding

A key method in natural language processing, particularly when it comes to LLMs, is text embedding [23]. Fundamentally, they convert text into a high-dimensional vector of real numbers, whether it be a word, sentence, or paragraph. These numerical representations capture the input text's semantic content, therefore they are not arbitrary. Two weather-related phrases, like "Chance of showers tomorrow" and "Expect rain in the morning," would have embeddings that place them near to one other in the vector space to demonstrate their similar meaning.

LLMs excel at tasks like search, categorization, and intent recognition because of this spatial relationship between vectors. Instead of comparing texts through surface-level pattern matching, the model measures how "close" two pieces of language are in this multidimensional space. Closeness is typically computed using distance metrics such as Euclidean or cosine distance. The former is based on geometric distance, while the latter evaluates how similar the directions of two vectors are useful when magnitude is less important than semantic alignment.

Embeddings power a wide range of practical applications. To retrieve the most semantically equivalent content, for example, a query such as "Will it snow this weekend?" can be embedded and compared to a collection of pre-embedded papers or reports. Similar to this, embeddings can be used in sentiment analysis or moderation to assess how closely a new message matches examples of content that have been tagged as suitable, unfavorable, or positive. Since embeddings are model outputs, they can be precomputed and stored efficiently, making them ideal for real-time systems that rely on fast similarity comparisons.

OpenAI's 'text-embedding-ada-002' model, derived from the GPT family, is currently one of the most effective tools for generating embeddings [24]. It generates 1536-dimensional vectors that demonstrate a thorough comprehension of the text, allowing for superior semantic analysis. Once created, these embeddings can be utilized for more than simply comparisons; they can also be used to visualize trends in the data using dimensionality reduction techniques like UMAP or to cluster similar articles together using algorithms like k-means.

The 'openai' package can be used to access OpenAI's API and create these embeddings in Python. With a single call to the 'Embedding.create()' method, a piece of text can be transformed into a dense vector. These vectors can then be used in downstream tasks, whether it's grouping similar reviews, powering a recommendation engine, or analyzing sentiment trends in real-time weather feedback.

By reducing human language to structured vectors, embeddings bridge the gap between raw text and mathematical reasoning. In the world of LLMs, they are what allow models to generalize, compare, and make sense of language, far beyond surface-level syntax.

4.2.3 Transformer

A Transformer is a type of neural network architecture specifically designed to handle sequences of data, like text. Its main function is to understand the meaning and relationships between words in a sentence, regardless of how far apart they are. Unlike older models that read text one word at a time, Transformers can look at an entire sentence or paragraph all at once. This allows them to capture context more effectively and generate more accurate, meaningful responses. Transformers are the foundation of modern language models like GPT because they are fast, flexible, and exceptionally good at understanding language through a mechanism called attention, which helps the model decide which words to focus on while processing or generating text.

Here's a simplified breakdown of the key components of a Transformer:

1. **Input and Input Embeddings**
   Text is broken into tokens (words or subwords), then converted into numerical vectors called embeddings, which represent the meaning of words in a mathematical space. The model learns these representations during training. These embeddings are crucial because they allow the model to capture the relationships between words in a mathematical form. These embeddings are learned during the training process, so the model gradually builds a deeper understanding of how different words relate to one another, based on the context in which they appear.

2. **Positional Encoding**
   Since word order matters in language, Transformers add positional information to embeddings so the model understands the sequence of words. Positional encodings are added to the input embeddings to inject information about the relative positions of tokens within the sequence. This ensures that the model understands the importance of word order in language. These encodings are mathematically designed to allow the model to distinguish between statements where the positions of the words alter the meaning of the sentence. Positional encodings are crucial for the transformer's ability to capture the syntactic structure of language.

3. **Encoder**
   The encoder is the first core component of the transformer. It processes the input sequence

by passing it through several self-attention layers, which enable the model to capture the relationships and dependencies between all tokens in the sequence. Unlike RNNs, which process the input one token at a time and have a limited capacity to capture long-range dependencies, the transformer can attend to every token in the sequence simultaneously. This is done through the self-attention mechanism, which computes attention scores between every pair of tokens. For each token, the model decides which other tokens it should focus on, based on their relevance to the token in question.

The self-attention mechanism computes three vectors for each word: the Query, the Key, and the Value. These vectors are used to calculate a weight (or ”attention score”) for each pair of tokens. The attention scores determine how much focus each word should place on every other word in the sequence. The weighted values are then combined to generate the output for each token. This approach allows the model to capture both local and global dependencies across the entire sequence of words, which is essential for understanding complex linguistic structures.

4. **Multi-Head Attention**
The multi-head attention technique is a major improvement over conventional attention processes. Transformers use various sets of weights to apply attention repeatedly in parallel rather than just once. This enables the model to concentrate on various facets of word relationships. For instance, one attention head might concentrate on semantic linkages (such word meanings or co-reference) and another on syntactic dependencies (like subject-verb agreement). The model may develop a deeper representation of the input sequence by combining these several attention heads, which is essential for intricate tasks like text synthesis, machine translation, and question answering.

5. **Shifted Outputs (for Training)**
When training, the model is taught to predict the next word by seeing only the previous ones. This is done by shifting the output sequence to the right, so the model never ”cheats” by looking ahead.

6. **Output Embeddings**
Just like input embeddings, outputs are also converted into vectors. These go through positional encoding and are used to calculate the loss function, which helps the model improve its predictions during training.

7. **Decoder**
The decoder is the second key component of the transformer. Its role is to generate the output sequence, based on the processed input from the encoder. Like the encoder, the decoder consists of several layers of self-attention and feed-forward neural networks. However, the decoder has an additional feature: it attends not only to its own previous output (as in the self-attention layers) but also to the encoder’s output. This allows the decoder to generate a coherent and contextually appropriate sequence of tokens. In

tasks like text generation or translation, the decoder is essential for producing fluent, contextually relevant responses. The decoder works in a step-by-step manner, predicting one word at a time until the entire sequence is generated.

8. **Feedforward Networks and Residual Connections**
   Following the self-attention mechanism in both the encoder and the decoder, a feedforward neural network (FFN) processes the output. A non-linear activation function sits between the two completely connected layers that make up this network. The feedforward network facilitates the model's learning of increasingly intricate input-output mappings and linkages. Furthermore, residual connections are employed in the vicinity of the feedforward and self-attention layers. By adding each layer's input to its output, these residual connections assist solve the vanishing gradient issue and make it possible to train very deep models more effectively.

9. **Linear Layer and Softmax**
   After decoding, a linear layer transforms the data into a format that can be compared with the vocabulary. Then, softmax assigns probabilities to each possible next word, allowing the model to choose the most likely one(For more information read [20] [25]).

Transformers' capacity to manage long-range relationships and execute parallel computation has fundamentally altered the field of NLP [26]. Transformers employ the self-attention mechanism to capture dependencies between all tokens simultaneously, in contrast to Recurrent Neural Networks(RNN) or Long Short-Term Memories(LSTM), which evaluate input sequences sequentially and struggle to catch long-distance links in text. Because of this, transformers can accept longer input sequences and scale effectively without being constrained by the computational limits of older architectures.

Because of their adaptability, transformers can be used for a number of activities outside text production, including summarization, machine translation, and text classification. They are very good at comprehending complex language patterns because they can focus on multiple portions of the sequence at once and grasp both local and global interdependence.

To sum up, transformers mark a substantial advancement in the way neural networks interpret and produce language. They are able to create rich, contextualized representations of text through mechanisms including positional encoding, multi-head attention, and self-attention, which help them do well on a variety of NLP tasks. Transformers are now an essential component of contemporary AI systems since their architecture serves as the basis for models like GPT, which have raised the bar for language creation.

4.2.4 Trainer

The function of the data trainer has become crucial in developing and optimizing sophisticated language processing systems in a world that is becoming more and more AI-driven [27]. These experts are the designers of quality and accuracy in AI solutions, turning vast amounts of data into strong bases for producing responses that are logical, pertinent, and appropriate for the given context. They are far from being merely technical contributors.

The meticulous selection and curation of data is the first step in the Data Trainer's job. Large volumes of textual content are insufficient; sources that are representative and pertinent to the particular field must be found and converted into datasets that accurately reflect the richness and complexity of human language. This selection process is not at all random; it calls for discernment, subject-matter expertise, and a thorough comprehension of the project's goals.

Following the collection of the data, the Data Trainer meticulously and precisely organizes and annotates the data. AI systems can now identify relationships, entities, and patterns in unprocessed data that would otherwise be invisible thanks to this annotation process. To ensure that the system can produce precise, understandable, and pertinent responses, the Data Trainer, for instance, identifies standard names, canonical units and descriptions while creating a dataset for a virtual assistant in the CF conventions.

The Data Trainer serves as a guarantee of data quality in addition to annotation. One of their duties is to carefully verify data using metrics like error rates, coherence, and completeness in order to spot and fix any discrepancies or errors that can jeopardize results. Reliability depends on the Data Trainer's ability to maintain high standards because bad data can compromise performance regardless of how complex the underlying architecture or algorithms are.

Advanced sampling strategies are often used by data trainers to optimize training process performance and efficiency. They can prioritize the most representative and instructive instances using techniques like Ask-LLM sampling and density sampling, which speeds up convergence and maximizes resource utilization. But in the end, these procedures are guided by the Data Trainer's expertise and judgment; they choose which data to include and how to strike a balance between relevance and diversity in order to prevent bias and guarantee a successful learning process.

A special combination of language and technical skills is required for this position. In addition to being fluent in linguistic nuances, data trainers must comprehend the goals of the project and modify data so that the final system can comprehend context and intent in addition to words. The result is a solution that comprehends and addresses real-world needs in addition to processing text.

In addition, the Data Trainer is essential in combating prejudice and advancing equity. They take measures to guarantee diversity and neutrality in the datasets they create because they understand that data represents human viewpoints and possible biases. Their work goes beyond technical prowess to include the proper and moral application of AI.

Numerous contemporary language processing applications, such as automatic code production, content creation, and multilingual customer support, clearly demonstrate the influence of Data Trainers. The foundation of every interaction that these technologies provide is the painstaking labor of the Data Trainer, who has transformed disparate data into a dependable and cohesive knowledge base that can be tailored to various situations and use cases.

In the end, the Data Trainer is the vital connection between artificial intelligence's potential and human competence. Their meticulous judgment, subject-matter knowledge, and dedication to excellence serve as the foundation for solutions that offer companies and users real value that goes beyond the technology itself.

5 Experimental setup

The programming ecosystem used in this project is based in Python [28] and Google Colab [29]. Data was taken from the CF conventions tables (in XML format) which is where the process started. This data was then converted in a ‘DataFrame’ from Pandas [30], which is a way to structure the data in Pyhon, so that the data was more usable.

5.1 Initial set-up

The code used in this section can be found in Appendix A1.

This stage was to create an LLM that can emit standard names in CF conformance. This phase converted the ’DataFrame’ into a clean dataset, good for data handling (data is already in the correct form to be consumed by the model) for efficient data transformation.

A tokenizer was prepared and its function (using ’BertTokenizerFast’) was defined. Finally, a well-formed dataset is needed to be able to train the model and be able to get more accurate estimations.

Next, the tokenization of our dataset (using ’BertTokenizerFast’) was done. In the context of this task, the pre-trained BERT model ’bert-base-uncased’ is often used because it has a superior understanding of context in a text sequence of words. Input text was converted into tokens by performing tokenization.

After the tokenizer was configured and the dataset was tokenized, the next step was feeding this data into the transformer model. The transformer is meant to enable the model to understand the meaning of words or phrases in the provided input. In this case, the aim was for the model to correctly generate standard names corresponding to the descriptions in the dataset.

Subsequently, the model was loaded, and the transformer was built to make predictions based on the provided descriptions. Random elements from the dataset were selected and used as input for the model, with the generated predictions compared to the real standard names. However, the results were not satisfactory. The predictions generated by the model were incorrect and did not reflect the real standard names associated with the descriptions provided. As a result, the decision was made to adjust the code to improve the model's performance.

5.2 Dataset adaptation

The code used in this section can be found in Appendix A.2.

After analysing the results from the initial setup, some issues appeared related to the way the standard names were structured. In the format used by CF conventions, standard names contain an underscore ("_") to separate words within the same name. However, the tokenizer may have been interpreting these underscores as part of a single word, causing the model to fail in capturing the correct meaning of each individual word within the standard name.

This issue is particularly critical because, as mentioned in the CF conventions, the standard names are very specific and must be treated accurately. If the tokenizer joins those words together into one token, then the model may not get all the context and meaning from an entity name context.

A line of code was then added to change the underscores in standard names into spaces (basically the same names but with a different spacing between words). The change, as small as this, is to enable the tokenizer to operate on words separately and, therefore, hopefully make model predictions more fluent.

Despite this adjustment, the predictions continued to be inaccurate. After further reflection, it was concluded that the pre-trained BERT model might be too generic for this specific task. Instead of working well with the domain-specific standard names in the CF conventions, the model appeared to be making predictions based on a broader and more general context. This led to the decision to train a more specialized model that could better fit the characteristics of the data and improve prediction accuracy.

5.3 Trainer creation

The code used in this section can be found in Appendix A.3.

For the third trial, a trainer was used. Trainer tool is one offered by Hugging Face Transformers library for training and evaluating a language model. When using a trainer, it was hoped that cleaner results would be returned, inherently more precise and with less control over the aspect of training that the tool optimizes.

The initial trainer load was estimated to take about 10 hours for the training process. Upon doing more research, I found that actually running the code on a GPU helped a hell of a lot versus doing it on just a CPU and substantially cut down the training time. The training time reduced from 35 hours to 35 minutes with the GPU, which was a big speed boost and efficiency.

During GPU execution of the code, the model showed those two things it felt were most relevant: one that had a step (number), and another training loss. Training loss is a crucial metric telling how accurate the output of our model is for the given inputs; low training loss means that the model is doing well and producing better and accurate predictions. The training loss here would be around 8 which is pretty high (the training loss should be almost close to 1). So the model still predicted incoherent way, as our previous tries did.

While making these strides, the trialing model model prediction obsession continued: it made a stupid prediction composed of standard words pertaining to type of radiation over and over again, which indicated that the model failed to learn anything more sophisticated than a bad word association in my dataset due to random happenstance. This misconduct is a direct sign of overfitting, as the model starts to over-lean on training data and being non-generalizable patterns. Overfitting will happen when the model cannot generalize in the right way to the new data, resulting to its prediction power is being limited.

When machine learning model starts overfitting [31]. It means model learns not only the underlying patterns in the training data but also learns the noise and outliers to the model. Which results in great performance for the training set but bad generalisability to new, unseen data. Like memorizing answers to certain questions instead of actually understand the big picture concepts.

5.4 Performance improvement

The code used in this section can be found in Appendix A.4.

The standard names of the most recent version (version 90) were first separated according to the version in which each name initially appeared in the experiment's fourth attempt. This stage gives training a more structured framework and is crucial for examining how the data has changed over time.

A smaller dataset that corresponded to versions 2 through 45 was then chosen. Two primary factors led to the selection of these versions. First, because the amount of data is easier to handle, training the model is quicker and more effective when a smaller dataset is used. Second, using a smaller dataset also lessens the possibility of overfitting, which is essential for enhancing the model's capacity for generalization.

To maximize the model's performance, the trainer parameters had to be changed after the data was ready. Weight decay, which controls the penalty applied to large weights during training; num_train_epochs, which determines the number of complete cycles of the training dataset to be completed; and learning rate, which controls the step size the model takes when adjusting its parameters during training, were some of the parameters that were changed. The quality of the training is immediately affected by changing these parameters, which improves fine-tuning and lowers training loss.

One important parameter that quantifies the discrepancy between the model's predictions and the actual labels during training is training loss. Since a low training loss means that the model is successfully learning from the training data, the primary objective is to reduce this value in order to increase the model's accuracy. A very low training loss, on the other hand, can be a sign of overfitting, in which the model has learned the data by heart rather than recognizing broad patterns that can be applied to fresh data. This phenomenon, which indicates that the model has lost its capacity to generalize correctly, becomes apparent when the validation loss begins to rise while the training loss keeps down.

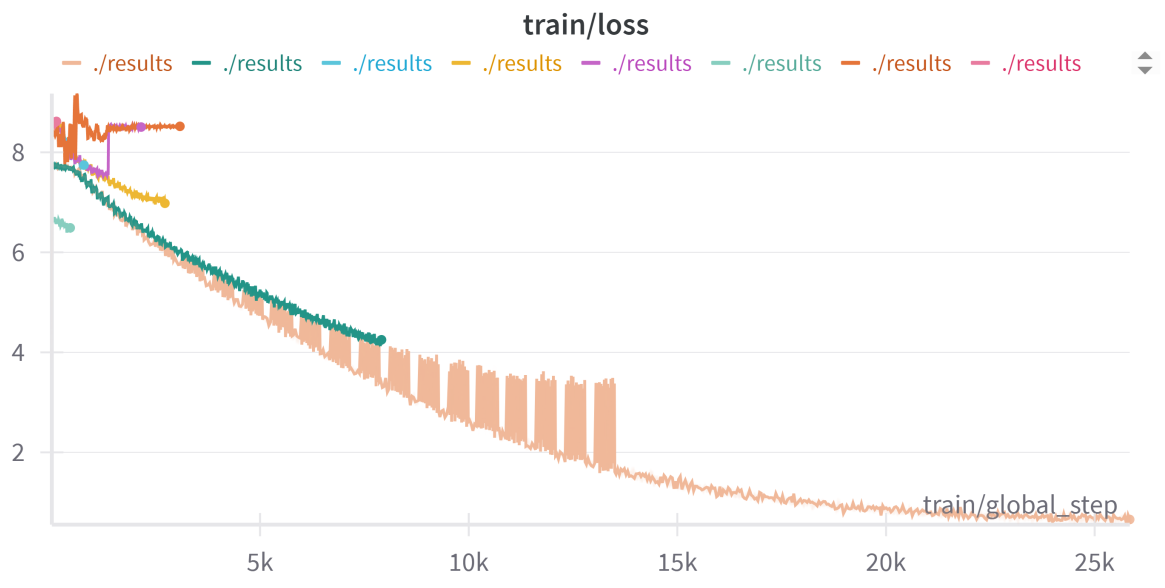


Figure 5.1: Example of different training values in our model.

The trainer has undergone a number of modifications, as seen in the above figure, from which we can infer a number of inferences. First, as the orange and purple training lines show, the training loss will not reduce if the parameters are not optimal for training the model or are not properly adjusted. These lines demonstrate how ineffectively the model is improving under those specific conditions.

We can therefore conclude that the trainer settings have been properly adjusted by comparing the pink and blue lines; the only difference between them is the number of epochs. Two significant inferences can be made from this. First, since the model has more chances to fine-tune its parameters and perform better, a larger number of epochs increases the trainer's chances of lowering the training loss. However, one of the lines exhibits a quicker decrease in the training loss even though all sets of parameters are identical. This disparity can be explained by a randomness or variability component in the training process, which implies that even when the parameters are the same, the optimization method's inherent randomness may cause the rate of improvement to vary.

The model had a decent fit to the data, as evidenced by the training loss of 0.65 that we achieved after finishing the entire process. We then went ahead and made some forecasts. When we first included descriptions from versions 2 through 45, the model gave back the same standard name as it did in the table. There are two probable explanations for this result: either the model tends to predict the same standard name as in the training data since it was trained using the same descriptions, or the program may have been biased or made a mistake that produced this repeating result.

In light of this, we chose to carry out a fresh experiment to investigate the potential outcomes of incorporating data from the versions that were not used for training. For these descriptions, we anti-cipated receiving precise standard names. The model produced the predictions 'square_of_sea_surface_ temperature' and northward_land_ice_velocity in the first test, which included descriptions of 'land_ice_ temperature' and 'sea_surface_temperature'. Despite being fairly correct, these forecasts were erroneous due to certain details.

The model returned the same standard name, 'land_cover_lccs', in all of the subsequent experiments we conducted using the descriptions 'change_in_land_ice_mass' and 'land_ice_mass'. This identified two problems: first, it should not return the same standard name for two distinct descriptions; second, the forecast was problematically missing the term "ice."

In a last experiment, we included the variables 'change_in_land_ice' and 'tendency_of_land_ice_mass'. The model recognized them as being the same and produced the same result for both.

We chose to look into the underlying reason for the mistakes after evaluating these results, which were near but still incorrect. First, we ruled out problems with the model's adjustment because the trainer seemed to be working well. Since the predictions were coming true, the code was also written appropriately. Consequently, we came to the conclusion that the issue might be caused by database anomalies, which might have resulted in a training strategy that was not appropriate.

After looking through the database more, we found a number of problems: First, the model's fitting was hampered by the standard names that lacked details. Furthermore, we discovered that the model was unable to identify cases where standard names had the same descriptions but noticeable variations in the standard name itself. Additionally, some standard names included labels that were overly descriptive, leaving out crucial details, which led to a discrepancy in the model's predictions. Last but not least, the model was unable to comprehend several descriptions that made reference to or were connected to other parts of the CF conventions. Even though there weren't many of these problems, taken together, they resulted in a large number of predictions that didn't correspond.

We then reexamined whether we were handling the issue appropriately. We made the decision to carefully examine the code and the output it generated. After more examination, we discovered two crucial facts: Initially, the software was only returning one token, which was unexpected because a vector of tokens should have been produced when the standard name was generated. Upon further investigation, we found that the software was generating a list of standard names (ranging from 2 to 45) based on the versions we were entering and identifying the one that most closely matched the input. This functionality proved to be pretty intriguing, even though it was not the result we had hoped for. For a given description, it might be used to find the best standard name available.

5.5 Fine tuning

The code used in this section can be found in Appendix A.5.

In this attempt, the model block has been changed from 'T5ForConditionalGeneration' to 'Bert-ForSequenceClassification', shifting the task from text generation to classification. The dataset has been converted into a Hugging Face Dataset format and split into training and evaluation sets using 'train_test_split'.

In this attempt, the 'BertTokenizerFast' tokenizer has been used, and the process includes fine-tuning, thanks to the Trainer. Despite these changes, the model continues to attempt to classify descriptions into a pre-existing standard name.

5.6 Standard name generation

The code used in this section can be found in Appendix A.6.

In this attempt, we finally achieved the desired goal. First, we obtained the dataset and, as in the previous version, we segmented it by versions to create an appropriate dataset. Next, we imported the 't5-small' tokenizer, a language model developed by Google Research that is designed to perform a wide range of natural language processing tasks, such as translation, summarization, and classification, all under a "text-to-text" framework.

With the tokenizer imported, we created the preprocessing function, which ensures that the inputs are valid strings and that there are no empty values in the data. Additionally, this function tokenizes both the inputs and possible outputs, and also generates the necessary training column for the model. Once this function was defined, it was applied to the entire dataset, creating a new version of the dataset with the processed Description and 'standard_name' columns.

We then split the dataset into two parts: one for training and the other for evaluation. With the data prepared, we loaded the 'T5ForConditionalGeneration' model using f'rom_pretrained('t5-small')' and configured the training parameters and the trainer.

When running the trainer, two key aspects caught our attention. First, the training loss started at much higher values than in previous attempts, which is typical in the initial stages of training. However, unlike in the previous case, this training loss decreased much more quickly, indicating that the model was adjusting to the problem more rapidly. This translates to a more efficient model fitting.

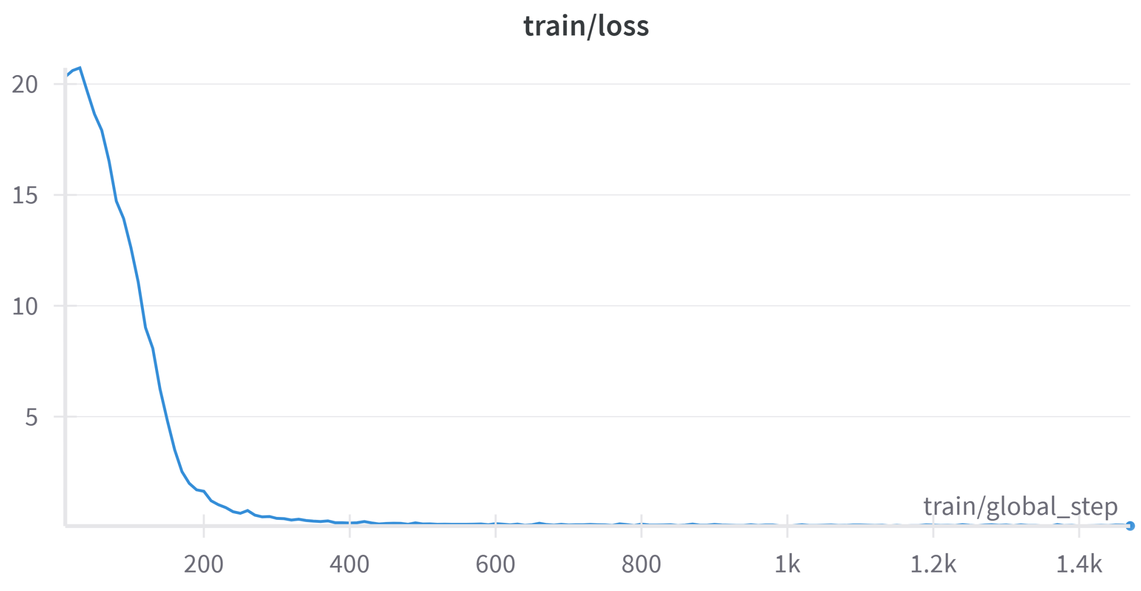


Figure 5.2: Training loss over the training steps

Finally, after completing the training, we saved the model and performed several tests to verify that the results obtained were valid and that the model was generating coherent and appropriate outputs based on the input data. This process allowed us to evaluate the performance of T5-small on specific tasks and compare its results to previous models.

Following this, we conducted an analysis of the results obtained from the model. To do this, we input various descriptions of standard names from the table and attempted to verify that the new standard names generated by the model did not correspond to any existing ones.

Our first test involved the standard name 'beaufort_wind_force'. The model returned exactly the same standard name, which led us to suspect that it was simply reclassifying our input into an already existing one. However, there was a factor that cleared up our doubt: for an unknown reason, the generated standard name starts with an uppercase letter, unlike the original, which is entirely in lowercase. This suggested that the model is indeed generating the standard name, and due to the description's type or specificity, it ended up generating the same standard name.

To confirm this assumption, we tested other standard names. Specifically, we used i'ntegral_wrt_depth _of_sea_water_conservative_temperature_expressed_as_heat_content', and the model's output was 'integral_wrt_time_of_temperature_expressed_as_heat_content_of_sea_water'. These names are similar and correctly align with their respective descriptions. As mentioned in the fourth attempt, some descriptions may skip certain details, which can lead to the model generating a standard name that surprisingly matches an existing one.

In the final phase, we performed two more experiments with our model. First, we input the description of 'tendency_of_mass_fraction_of_cloud_condensed_water_in_air', which is concise and directly describes the formation of the standard name, rather than its meaning. The model's output was 'tendency_of_condensed_water_in_air', which is quite close to the intended meaning, though it omits some details. In contrast, when we input the description for 'sea_surface_foundation_temperature', a longer and more physically descriptive text, the model outputted the exact same standard name. This demonstrated that our model performs well when the description accurately reflects the physical concept behind the standard name.

In summary, while the model sometimes generates the same standard name for similar descriptions, it is capable of adapting well to more descriptive and specific inputs, showing promising results in both general and precise cases.

6 Conclusions

In this project, we have demonstrated that LLMs are powerful tools for automating the generation of standard names for CF conventions, based on the underlying physical principles. The correct assignment of these names is essential for managing data in physics, as it ensures consistency and interoperability between different experiments. In addition to this achievement, we have developed another very useful tool: the sheer number of standard names makes it difficult to identify the most appropriate one, and this tool helps prevent human errors.

Throughout the project, we have sought to address the challenge of training a model based on the physical descriptions of the variables listed in the CF conventions. These data are stored in formats such as netCDF and organized according to the CF conventions, forming the foundation of various studies in atmospheric physics, ocean physics, and Earth sciences. The accurate interpretation and standardization of these variables are crucial for modeling processes, underscoring the importance of having names that are specific and unambiguous.

At the beginning of the experimentation, we clearly encountered difficulties in training models for such a specific task. The complexity of these concepts and the requirement for extreme precision initially prevented the model from adjusting optimally to the problem at hand. This highlighted the importance of tailoring the model's tools to the specific physical needs of our dataset.

After making changes to the model upon which our code was based, we achieved a much faster and more effective training process, demonstrating how crucial it is to select a model that aligns well with our specific needs.

Despite these advances, the model showed limitations when the physical descriptions were not clear enough. This was primarily due to the dataset itself, which contained several elements that produced erroneous results in our model. Some descriptions explained how the variable was formed rather than its physical meaning, while others omitted important elements present in the standard name or had empty entries. This highlights the importance of having a high-quality database to ensure a good fit between the model and the descriptions of the standard names.

Ultimately, this project not only provides a valuable tool for the scientific community but also opens up new pathways for computational and experimental physics. Such tools reduce the workload and human error that were historically part of data handling.

In the long term, this work lays the groundwork for future exploration of LLM integration in physics. Improvements such as employing more powerful models, expanding the dataset used for testing, and incorporating additional critical information, such as physical units and spatial or temporal dependencies, will further enhance the precision and functionality of this system. As a discipline grounded in rigorous data analysis and modeling, physics stands to benefit significantly from these advancements, with artificial intelligence offering new, more accurate ways to understand, model, and predict the processes that govern our planet.

References

[1] IBM. What is a Neural Network?, 2021. Available at: https://www.ibm.com/think/topics/neural-networks.

[2] Google. Google search, 2025.

[3] Artificial Neural Network — How does Artificial Neural Network Work, 2021. Available at: https://www.analyticsvidhya.com/blog/2021/04/artificial-neural-network-its-inspiration-and-the-working-mechanism/.

[4] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[5] Hopfield Network, 2023. Wikipedia. Available at: https://en.wikipedia.org/w/index.php?title=Hopfield_network&oldid=1282790517.

[6] NobelPrize.org. John Hopfield – facts – 2024, 2024. Available at: https://www.nobelprize.org/prizes/physics/2024/hopfield/facts/.

[7] Training energy-based single-layer hopfield and oscillatory networks with unsupervised and supervised algorithms for image classification. *Neural Computing and Applications*, 2023. Available at: https://link.springer.com/article/10.1007/s00521-023-08672-0.

[8] UNIDATA. NetCDF: Network Common Data Form, 2020. Boulder, CO: University Corporation for Atmospheric Research. Available at: https://doi.org/10.5065/D6H70CW6.

[9] Brian Eaton, Jonathan Gregory, Bob Drach, Karl Taylor, Steve Hankin, John Caron, and Rich et al. Signell. Netcdf climate and forecast (cf) metadata conventions, 2023. Available at: https://cfconventions.org/cf-conventions/cf-conventions.html.

[10] Unidata. NetCDF: The NetCDF Data Model, 2023. Available at: https://docs.unidata.ucar.edu/netcdf-c/4.9.3/netcdf_data_model.html.

[11] Introducing CDL (Common Data Language). https://www.unidata.ucar.edu/software/netcdf/workshops/2011/datamodels/Cdl.html, 2011.

[12] Unidata. Introduction — NetCDF User's Guide (NUG), 2023. Available at: https://docs.unidata.ucar.edu/nug/2.0-draft/index.html.

[13] Unidata Program Center. Unidata — University Corporation for Atmospheric Research. https://www.unidata.ucar.edu/, 2025.

[14] Unidata. NetCDF conventions, 2023. Available at: https://www.unidata.ucar.edu/software/netcdf/conventions.html.

[15] Unidata Program Center. Udunits 2.2.28 documentation. url-https://docs.unidata.ucar.edu/udunits/current/, 2025.

[16] Unidata. NUG Attribute Conventions — NetCDF User's Guide (NUG), 2023. Available at: https://docs.unidata.ucar.edu/nug/2.0-draft/nug_conventions.html.

[17] Cf standard names. {https://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html}, 2025.

[18] B. Eaton, J. Gregory, B. Drach, K. Taylor, S. Hankin, et al. NetCDF Climate and Forecast (CF) Metadata Conventions (1.13-draft). CF Community. https://doi.org/10.5281/zenodo.FFFFFF, 2024.

[19] CF Conventions Committee. Guidelines for construction of CF standard names. https://cfconventions.org/Data/cf-standard-names/docs/guidelines.html, 2008. Accessed: 2025-06-05.

[20] Putting it all together - Hugging Face LLM Course, 2023. Available at: https://huggingface.co/learn/llm-course/en/chapter2/6.

[21] Matheus Oliveira De Souza. LLM from scratch with Pytorch, 2023. Medium. Available at: https://medium.com/@msouza.os/llm-from-scratch-with-pytorch-9f21808c6319.

[22] What Is Tokenization? Types, Use Cases, Implementation, 2023. DataCamp. Available at: https://www.datacamp.com/blog/what-is-tokenization.

[23] Leveraging Text Embeddings with the OpenAI API: A Practical Guide, 2023. DataCamp. Available at: https://www.datacamp.com/tutorial/introduction-to-text-embeddings-with-the-open-ai-api.

[24] An Intuitive Introduction to Text Embeddings, 2023. Stack Overflow. Available at: https://stackoverflow.blog/2023/11/09/an-intuitive-introduction-to-text-embeddings/.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv*, 2017.

[26] R. Pradeep Menon. Introduction to Large Language Models and the Transformer Architecture, 2023. Medium. Available at: `https://rpradeepmenon.medium.com/introduction-to-large-language-models-and-the-transformer-architecture-534408ed7e61`.

[27] Training and validation loss in deep learning, 2023. GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/training-and-validation-loss-in-deep-learning/`.

[28] Python Software Foundation. `https://www.python.org/`, 2025.

[29] Google. Google colaboratory. `https://colab.research.google.com/`, 2025.

[30] The pandas development team. pandas: Python data analysis library. `https://pandas.pydata.org/`, 2025.

[31] Overfitting — Machine Learning — Google for Developers, 2023. Available at: `https://developers.google.com/machine-learning/crash-course/overfitting/overfitting/`.

Appendix

A1 Code for Initial set-up stage

```python
# Required imports
import pandas as pd
import torch
from datasets import Dataset
from transformers import BertTokenizerFast, BertForSequenceClassification

# Step 1: Load and prepare the data
url = 'https://cfconventions.org/Data/cf-standard-names/90/src/cf-standard-name-
    table.xml'
df = pd.read_xml(url, xpath='.//entry')
df.drop(columns=['canonical_units'], inplace=True)
df.rename(columns={'id': 'standard_name', 'description': 'Description'}, inplace
    =True)

# Step 2: Create a HuggingFace dataset and encode labels
dataset = Dataset.from_pandas(df)
label_dict = {label: idx for idx, label in enumerate(df["standard_name"].unique
    ())}
dataset = dataset.map(lambda x: {"labels": label_dict[x["standard_name"]]})

# Step 3: Load the tokenizer
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
print("Tokenizer loaded successfully")

# Ensure descriptions are strings
dataset = dataset.map(lambda x: {'Description': str(x['Description'])})

# Tokenization function
```

```python
def tokenize_function(examples):
    return tokenizer(examples['Description'], padding=True, truncation=True,
  max_length=512)

# Step 4: Tokenize the dataset
dataset = dataset.map(tokenize_function, batched=True)
print(dataset[0])  # Verify tokenization

# Step 5: Load the pretrained model for classification
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
  num_labels=len(label_dict))
print("Model loaded successfully")

# Step 6: Create new examples for prediction
entry1 = 'A variable with the standard_name of cloud_type contains either
  strings which indicate the cloud type, or flags which can be translated to
  strings using flag_values and flag_meanings attributes.'
entry2 = 'The albedo of cloud. Albedo is the ratio of outgoing to incoming
  shortwave irradiance, where shortwave irradiance means that both the
  incoming and outgoing radiation are integrated across the solar spectrum.'
new_data = [entry1, entry2]

# Tokenize the new examples
inputs = tokenizer(new_data, padding=True, truncation=True, return_tensors="pt")

# Step 7: Make predictions
with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits

predictions = torch.argmax(logits, dim=-1)

# Step 8: Display predictions
print("Predictions:")
print(predictions)

# Map predicted indices to label names
predicted_labels = [list(label_dict.keys())[label] for label in predictions.
  tolist()]

# Show the prediction results
for example, label in zip(new_data, predicted_labels):
    print(f"Description: {example} -> Prediction: {label}")
```

A2 Code for Dataset adaptation stage

```
# Step 1: Imports
import pandas as pd
import torch
from datasets import Dataset
from transformers import BertTokenizerFast, BertForSequenceClassification

# Step 2: Load and preprocess the data
url = 'https://cfconventions.org/Data/cf-standard-names/90/src/cf-standard-name-
    table.xml'
df = pd.read_xml(url, xpath='.//entry')

# Drop unnecessary column and rename for clarity
df.drop(columns=['canonical_units'], inplace=True)
df.rename(columns={'id': 'standard_name', 'description': 'Description'}, inplace
    =True)

# Replace underscores with spaces in the 'standard_name' column
df['standard_name'] = df['standard_name'].str.replace('_', ' ')

# Step 3: Convert to HuggingFace Dataset and create label dictionary
dataset = Dataset.from_pandas(df)
label_dict = {label: idx for idx, label in enumerate(df["standard_name"].unique
    ())}
dataset = dataset.map(lambda x: {"labels": label_dict[x["standard_name"]]})

# Step 4: Load the tokenizer
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
print("Tokenizer loaded successfully")

# Ensure all descriptions are strings
dataset = dataset.map(lambda x: {'Description': str(x['Description'])})

# Define tokenization function
def tokenize_function(examples):
    return tokenizer(examples['Description'], padding=True, truncation=True,
    max_length=512)

# Tokenize the dataset
dataset = dataset.map(tokenize_function, batched=True)
print(dataset[0])  # Verify tokenization

# Step 5: Load the BERT model for sequence classification
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
    num_labels=len(label_dict))
print("Model loaded successfully")

# Step 6: Prepare new input examples for prediction
entry1 = 'A variable with the standard_name of cloud_type contains either
    strings which indicate the cloud type, or flags which can be translated to
    strings using flag_values and flag_meanings attributes.'
```

```
entry2 = 'The albedo of cloud. Albedo is the ratio of outgoing to incoming
    shortwave irradiance, where shortwave irradiance means that both the
    incoming and outgoing radiation are integrated across the solar spectrum.'
new_data = [entry1, entry2]

# Tokenize new examples
inputs = tokenizer(new_data, padding=True, truncation=True, return_tensors="pt")

# Step 7: Run model prediction
with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)

# Step 8: Map predictions to label names
predicted_labels = [list(label_dict.keys())[label] for label in predictions.
    tolist()]

# Step 9: Print the results
print("Predictions:")
print(predictions)

for example, label in zip(new_data, predicted_labels):
    print(f"Description: {example} -> Prediction: {label}")
```

A3 Code for Trainer creation stage

```python
# Step 1: Import Libraries
import pandas as pd
import torch
from datasets import Dataset
from transformers import (
    BertTokenizerFast,
    BertForSequenceClassification,
    TrainingArguments,
    Trainer
)

# Step 2: Load and Preprocess Data
url = 'https://cfconventions.org/Data/cf-standard-names/1/src/cf-standard-name-
    table.xml'
df = pd.read_xml(url, xpath='.//entry')
df.drop(columns=['canonical_units'], inplace=True)
df.rename(columns={'id': 'standard_name', 'description': 'Description'}, inplace
    =True)
df['standard_name'] = df['standard_name'].str.replace('_', ' ')

# Step 3: Create Dataset and Labels
dataset = Dataset.from_pandas(df)
label_dict = {label: idx for idx, label in enumerate(df["standard_name"].unique
    ())}
dataset = dataset.map(lambda x: {"labels": label_dict[x["standard_name"]]})

# Step 4: Load Tokenizer
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
print("Tokenizer loaded successfully")

# Tokenize descriptions
dataset = dataset.map(lambda x: {'Description': str(x['Description'])})
def tokenize_function(examples):
    return tokenizer(examples['Description'], padding=True, truncation=True,
    max_length=400)
dataset = dataset.map(tokenize_function, batched=True)
print(dataset[0])

# Step 5: Load Model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
    num_labels=len(label_dict))
print("Model loaded successfully")

# Step 6: Set Training Arguments
training_args = TrainingArguments(
    learning_rate=5e-5,
    max_grad_norm=1.0,
    output_dir='./results',
    num_train_epochs=5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=10,
)
print("Training parameters configured")

# Step 7: Create Trainer and Train Model
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    eval_dataset=dataset,
)
trainer.train()

# Step 8: Prepare New Inputs for Prediction
entry1 = "A variable with the standard_name of cloud_type contains either
    strings..."
entry2 = '"Sea surface wave radiation stress" describes the excess momentum flux
    ...'
new_data = [entry1, entry2]

# Step 9: Tokenize and Predict
inputs = tokenizer(new_data, padding=True, truncation=True, return_tensors="pt")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
inputs = {key: value.to(device) for key, value in inputs.items()}
model.to(device)

with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
predictions = torch.argmax(logits, dim=-1)

# Step 10: Display Results
predicted_labels = [list(label_dict.keys())[label] for label in predictions.
    tolist()]
print("Predictions:", predictions)
for example, label in zip(new_data, predicted_labels):
    print(f"Description: {example} -> Prediction: {label}")
```

A4 Code for Performance improvement stage

```python
# Step 1: Imports
import pandas as pd
import torch
from datasets import Dataset
from transformers import (
    BertTokenizerFast,
    BertForSequenceClassification,
    TrainingArguments,
    Trainer
)

# Step 2: Download and preprocess multiple versions
base_url = "https://cfconventions.org/Data/cf-standard-names/{}/src/cf-standard-name-table.xml"
all_data = []
version_90_standard_names = set()

for version in range(1, 91):
    if version == 38:
        continue

    url = base_url.format(version)
    df = pd.read_xml(url, xpath='.//entry')
    df.rename(columns={'id': 'standard_name', 'description': 'Description'}, inplace=True)
    df['standard name'] = df['standard_name'].str.replace('_', ' ')
    df['version'] = version

    if version == 90:
        version_90_standard_names = set(df['standard_name'])

    all_data.append(df)

# Step 3: Combine and filter the data
final_df = pd.concat(all_data, ignore_index=True)
final_df = final_df.drop_duplicates(subset='standard_name', keep='first')
final_df = final_df[final_df['standard_name'].isin(version_90_standard_names)]
final_df.drop(columns=['grib', 'amip'], errors='ignore', inplace=True)

# Step 4: Select a version range subset
df = final_df
df2 = df[df['version'].between(2, 45)]

# Step 5: Create Dataset and label mapping
dataset = Dataset.from_pandas(df2)
label_dict = {label: idx for idx, label in enumerate(df2["standard_name"].unique())}
dataset = dataset.map(lambda x: {"labels": label_dict[x["standard_name"]]})

# Step 6: Tokenization
```

```python
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
print("Tokenizer loaded successfully")

dataset = dataset.map(lambda x: {'Description': str(x['Description'])})
def tokenize_function(examples):
    return tokenizer(examples['Description'], padding=True, truncation=True,
    max_length=400)

dataset = dataset.map(tokenize_function, batched=True)
print(dataset[0])

# Step 7: Load Model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
    num_labels=len(label_dict))
print("Model loaded successfully")

# Step 8: Training configuration
training_args = TrainingArguments(
    learning_rate=2e-5,
    max_grad_norm=1.0,
    output_dir='./results',
    num_train_epochs=190,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.001,
    logging_dir='./logs',
    logging_steps=10,
)
print("Training parameters configured")

# Step 9: Train the model
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    eval_dataset=dataset,
)
trainer.train()

# Step 10: Prediction
def1 = '
    integral_wrt_depth_of_sea_water_conservative_temperature_expressed_as_heat_content
    '
def2 = 'beaufort_wind_force'

entry1 = str(df[df['standard_name'] == def1]['Description'].iloc[0])
entry2 = str(df[df['standard_name'] == def2]['Description'].iloc[0])
new_data = [entry1, entry2]

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
inputs = tokenizer(new_data, padding=True, truncation=True, return_tensors="pt")
    .to(device)

with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
predictions = torch.argmax(logits, dim=-1)

predicted_labels = [list(label_dict.keys())[label] for label in predictions.
    tolist()]
print(f'Table -> [{def1}, {def2}]         Predictions -> {predicted_labels}')
```

A5 Code for: Fine-tuning stage

```python
# Step 1: Imports
import pandas as pd
import torch
from datasets import Dataset
from sklearn.model_selection import train_test_split
from transformers import (
    BertTokenizerFast,
    BertForSequenceClassification,
    Trainer,
    TrainingArguments
)

# Step 2: Load and preprocess all XML versions
base_url = "https://cfconventions.org/Data/cf-standard-names/{}/src/cf-standard-
    name-table.xml"
all_data = []
version_90_standard_names = set()

for version in range(1, 91):
    if version == 38:
        continue

    url = base_url.format(version)
    df = pd.read_xml(url, xpath='.//entry')
    df.rename(columns={'id': 'standard_name', 'description': 'Description'},
    inplace=True)
    df['standard name'] = df['standard_name'].str.replace('_', ' ')
    df['version'] = version

    if version == 90:
        version_90_standard_names = set(df['standard_name'])

    all_data.append(df)

final_df = pd.concat(all_data, ignore_index=True)
final_df = final_df.drop_duplicates(subset='standard_name', keep='first')
final_df = final_df[final_df['standard_name'].isin(version_90_standard_names)]
final_df.drop(columns=['grib', 'amip'], inplace=True, errors='ignore')

df = final_df
df2 = df[df['version'].between(2, 45)]

# Step 3: Dataset and Tokenization
dataset = Dataset.from_pandas(df2)
label_dict = {label: idx for idx, label in enumerate(df2["standard_name"].unique
    ())}
dataset = dataset.map(lambda x: {"labels": label_dict[x["standard_name"]]})

tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
print("Tokenizer loaded successfully")
```

```

dataset = dataset.map(lambda x: {'Description': str(x['Description'])})
def tokenize_function(examples):
    return tokenizer(examples['Description'], padding=True, truncation=True,
   max_length=400)

dataset = dataset.map(tokenize_function, batched=True)
print(dataset[0])

# Step 4: Train-test split
dataset_df = dataset.to_pandas()
train_df, eval_df = train_test_split(dataset_df, test_size=0.2)
train_dataset = Dataset.from_pandas(train_df, preserve_index=False)
eval_dataset = Dataset.from_pandas(eval_df, preserve_index=False)

# Step 5: Load model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
   num_labels=len(label_dict))
print("Model loaded successfully")

# Step 6: Training configuration
training_args = TrainingArguments(
    learning_rate=2e-5,
    max_grad_norm=1.0,
    output_dir='./results',
    num_train_epochs=100,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.001,
    logging_dir='./logs',
    logging_steps=10,
)

# Step 7: Training
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)
trainer.train()

# Step 8: Make predictions
def1 = '
   integral_wrt_depth_of_sea_water_conservative_temperature_expressed_as_heat_content
   '
def2 = 'beaufort_wind_force'

entry1 = str(df[df['standard_name'] == def1]['Description'].iloc[0])
entry2 = str(df[df['standard_name'] == def2]['Description'].iloc[0])
new_data = [entry1, entry2]
```

```

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
inputs = tokenizer(new_data, padding=True, truncation=True, return_tensors="pt")
    .to(device)

with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
predictions = torch.argmax(logits, dim=-1)

# Step 9: Inspect tokenized input
tokenized_inputs = tokenizer(new_data, padding=True, truncation=True,
    return_tensors="pt")
print(f"Tokens generated for descriptions:")
for i, description in enumerate(new_data):
    print(f"\nDescription: {description}")
    print(f"Tokens: {tokenized_inputs['input_ids'][i]}")
    print(f"Decoded: {tokenizer.decode(tokenized_inputs['input_ids'][i])}")

# Step 10: Map predictions to labels
predicted_labels = [list(label_dict.keys())[label] for label in predictions.
    tolist()]
print(f'Table -> [{def1}, {def2}]         Predictions -> {predicted_labels}')
```

A6 Code for Standard name generation stage

```python
# Step 1: Load data from multiple versions
import pandas as pd
from datasets import Dataset
from transformers import T5ForConditionalGeneration, T5Tokenizer, Trainer, TrainingArguments
from sklearn.model_selection import train_test_split
import torch

base_url = "https://cfconventions.org/Data/cf-standard-names/{}/src/cf-standard-name-table.xml"
all_data = []
version_90_standard_names = set()

for version in range(1, 91):
    if version == 38:
        continue
    url = base_url.format(version)
    df = pd.read_xml(url, xpath='.//entry')
    df.rename(columns={'id': 'standard_name', 'description': 'Description'}, inplace=True)
    df['standard name'] = df['standard_name'].str.replace('_', ' ')
    df['version'] = version
    if version == 90:
        version_90_standard_names = set(df['standard_name'])
    all_data.append(df)

final_df = pd.concat(all_data, ignore_index=True)
final_df = final_df.drop_duplicates(subset='standard_name', keep='first')
final_df = final_df[final_df['standard_name'].isin(version_90_standard_names)]
final_df.drop(columns=['grib', 'amip'], inplace=True, errors='ignore')

df = final_df
dataset = Dataset.from_pandas(df)

# Step 2: Tokenizer and preprocessing
tokenizer = T5Tokenizer.from_pretrained('t5-small')

def preprocess_function(examples):
    texts = [str(x) if x else "" for x in examples['Description']]
    summaries = [str(x) if x else "" for x in examples['standard name']]
    inputs = tokenizer(texts, padding="max_length", truncation=True, max_length=512)
    targets = tokenizer(summaries, padding="max_length", truncation=True, max_length=150)
    inputs['labels'] = targets['input_ids']
    return inputs

dataset = dataset.map(preprocess_function, batched=True)

# Step 3: Train-test split
```

```
dataset = dataset.train_test_split(test_size=0.2, seed=42)
train_dataset = dataset['train']
test_dataset = dataset['test']

# Step 4: Load model
model = T5ForConditionalGeneration.from_pretrained('t5-small')

# Step 5: Training configuration
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=10,
)

# Step 6: Train the model
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset
)
trainer.train()
model.save_pretrained('./t5_model')

# Step 7: Load trained model and generate standard names
model = T5ForConditionalGeneration.from_pretrained('./t5_model')
tokenizer = T5Tokenizer.from_pretrained('t5-small')

def generate_sn(text):
    inputs = tokenizer(text, return_tensors="pt", max_length=512, truncation=
    True, padding="max_length")
    sn_ids = model.generate(inputs['input_ids'], max_length=40, min_length=5,
    length_penalty=2.0, num_beams=4, early_stopping=True)
    return tokenizer.decode(sn_ids[0], skip_special_tokens=True).replace(" ", "_
    ")

# Test examples
def1 = 'tendency_of_mass_fraction_of_cloud_condensed_water_in_air'
def2 = 'thickness_of_convective_rainfall_amount'

desc1 = str(df[df['standard_name'] == def1]['Description'].iloc[0])
desc2 = str(df[df['standard_name'] == def2]['Description'].iloc[0])

sn1 = generate_sn(desc1)
sn2 = generate_sn(desc2)

print("Original:", def1)
print("Generated:", sn1)
print("\nOriginal:", def2)
print("Generated:", sn2)
```