Language proximity analysis

Adrian Bagiński, Mateusz Cakała, Krzysztof Olczyk, Jan Zakroczymski

October 2025

1 Introduction

Program's goal is to compare words from different languages and find out how similar they are to each other using special algorithms. Then it creates a graph picturing how similar are they to each other. Levenshtein's distance method is going to be used in process. Code is written in Python programming language.

2 Prototype version - October 15th 2025

First and very basic version of the program that will be developed later. It has few files with various function realizing functionalities as taking or saving data and creating a comparison of languages.

2.1 file_utils.py

This part of the program defines a set of utility functions responsible for handling input and output operations related to word lists and similarity matrices. They serve as supporting tools for the main part of the program, which compares words between two languages and calculates their similarity.

2.1.1 `get_words_from_file(file_path)`

Purpose: Reads a list of words from a text file and returns them as a Python list.

Description:

- Opens the file specified by `file_path` in UTF-8 encoding.
- Reads all lines, strips unnecessary whitespace using `.strip()`.
- Filters out empty lines so that only valid words remain.

Relevance: This function is likely used to load vocabulary lists for each language being compared. The resulting lists are then used in similarity calculations between languages.

2.1.2 `save_words_to_file(words, file_path)`

Purpose: Saves a list of words to a text file, ensuring that the output directory exists.

Description:

- Creates the output directory using `os.makedirs()` if it does not already exist.
- Writes each word from the `words` list to a new line in the file.

Relevance: This function allows saving processed or filtered word lists that can later be used for further comparison or analysis.

2.1.3 `save_similarity_matrix(words1, words2, matrix, file_path)`

Purpose: Saves a complete similarity matrix to a CSV file, where each cell represents the similarity between a word from the first language (`words1`) and a word from the second language (`words2`).

Description:

- Ensures the output directory exists before saving.
- Opens a CSV file in UTF-8 encoding and writes:
  - A header row containing all words from `words2` (the column labels).
  - For each word in `words1`, a new row containing the similarity scores from `matrix`.
- Each similarity value is formatted to two decimal places using `f"{v:.2f}"`.

Comment translation (from Polish):

> "Saves the full similarity matrix to a CSV file. Rows: `words1`, Columns: `words2`."

Relevance: This function provides a structured and human-readable representation of the computed word similarities, facilitating analysis of how closely related the two compared languages are.

2.2 graph_utils.py

2.2.1 `save_similarity_matrix_csv(matrix, languages, topic, folder="results/graphs")`

Purpose: Saves a language similarity matrix to a CSV file. Each cell of the matrix represents the similarity value between two languages. The file is named according to a given topic, and stored in a specified folder.

Description:

- Ensures that the output directory exists using `os.makedirs()` with `exist_ok=True`.

- Constructs the output file path dynamically, combining the given `folder` and the file name based on the provided `topic` (e.g., `topic_matrix.csv`).
- Opens a new CSV file for writing in UTF-8 encoding.
- Writes the first row as a header, where columns correspond to the list of `languages`.
- Iterates over the `languages` and their corresponding rows in the `matrix`, writing each language name followed by its similarity values.

Relevance: This function is designed to export a matrix that shows the degree of similarity between multiple languages, possibly computed for a particular `topic` or lexical domain. It helps visualize or analyze cross-linguistic relationships by providing a structured and easily readable CSV representation.

2.3 similarity.py

2.3.1 `compute_similarity(word1, word2)`

Purpose: Computes the similarity between two words and returns a numerical value in the range from 0 to 1, where 1 indicates identical words and 0 represents complete dissimilarity.

Description:

- Utilizes the `textdistance` library, specifically the `Levenshtein` distance algorithm.
- The function calls `textdistance.levenshtein.normalized_similarity(word1, word2)`, which measures the similarity by comparing the minimum number of character edits (insertions, deletions, substitutions) required to transform one word into the other.
- The resulting similarity score is normalized to a range between 0 and 1, providing a standardized metric of word resemblance.

Relevance: This function represents the core computational step in the program's language comparison process. By quantifying how similar individual words are, it enables the construction of broader similarity matrices and contributes to the analysis of linguistic relationships between languages.

2.4 translate.py

2.4.1 `translate_word(word, lang)` and `translate_words(words, lang)`

Purpose: Provides automatic translation of words into a specified target language using the Google Translate service. The functions support both single-word and batch translation.

Description:

- The implementation utilizes the `GoogleTranslator` class from the `deep_translator` library.
- `translate_word(word, lang)`:
  - Automatically detects the source language (`source='auto'`) and translates the input `word` into the target language specified by `lang`.
  - Converts the translation result to lowercase to ensure consistency.
  - Handles exceptions gracefully — if an error occurs during translation, an informative error message is printed, and the original word is returned unchanged.
- `translate_words(words, lang)`:
  - Performs batch translation by applying the `translate_word()` function to each word in the provided `words` list.
  - Returns a list of translated words corresponding to the input list.

Relevance: These functions enable multilingual comparison by ensuring that words from different languages are translated into a common reference language before similarity computation. This step is essential for fair and consistent cross-linguistic analysis, especially when measuring lexical similarity between language pairs.

2.5 `main.py` — Program Orchestration and Workflow

Purpose: This script serves as the main entry point of the entire project. It coordinates the process of loading word lists, translating them into multiple languages, computing pairwise word similarities, and saving the resulting data for each topic.

Description:

- **Imports and Configuration:**
  - Imports utility functions from separate modules:
    - `get_words_from_file()`, `save_words_to_file()`, `save_similarity_matrix()` from `utils.file_utils`.
    - `translate_words()` from `utils.translate`.
    - `compute_similarity()` from `utils.similarity`.
  - Defines key configuration variables:
    - `languages = [''en'', ''pl'', ''es'']` — specifies which languages will be compared.
    - `data_dir = "data"` — input folder containing source word lists.
    - `results_dir = "results"` — base folder for all output files.
  - Ensures that required subdirectories exist:

    * `results/translations` — stores translated word lists.
    * `results/similarities` — stores CSV files with similarity matrices.

- **Main Processing Loop:**
  - Iterates through all `.txt` files in the `data` directory. Each file represents a separate *topic* or semantic field.
  - For each file:
    1. Extracts the topic name by removing the `.txt` extension.
    2. Loads the list of words using `get_words_from_file()`.
    3. Translates the list of words into each target language using `translate_words()`.
    4. Saves each set of translated words into separate files within the `results/translations` folder.

- **Similarity Computation:**
  - For every unique pair of languages (`lang1`, `lang2`), computes a word-by-word similarity matrix using `compute_similarity()`.
  - Each element of the matrix represents the normalized similarity score between a word from `lang1` and a word from `lang2`.
  - The resulting matrix is saved as a CSV file in `results/similarities`, named according to the topic and the two compared languages (e.g., `animals_en_pl.csv`).

- **Program Output:**
  - After all topics are processed, a confirmation message is displayed: ``All topics processed successfully! Check results folder.''

Relevance: The `main.py` script integrates all project components into a unified workflow. It automates the full pipeline — from data loading and translation to similarity computation and result storage. This modular structure ensures scalability, allowing new languages or topics to be easily added without modifying the underlying logic of translation or similarity calculation.

3 Graph drawing - October 24th 2025

Prototype version of the program covered only similarity calculation without any graphical representation. Program should create graphs showing graphically how similar to each other compared languages are. For doing so two special functions were added and main file was developed.

3.1 overall_similarity.py

3.1.1 `add_connection(graph, node1, node2, label)`

Purpose: Adds nodes and a labeled connection (edge) between them within a graph structure. This function is used to represent relationships — such as linguistic similarities — as edges connecting nodes that represent individual entities (e.g., languages).

Description:

- Accepts a `graph` object (likely a `networkx.Graph` or similar structure).
- Ensures that both `node1` and `node2` exist in the graph by calling `graph.add_node()` for each.
- Adds an edge between the two nodes using `graph.add_edge(node1, node2, label=label)`.
- The `label` parameter can hold metadata, such as a similarity value or a descriptive tag.

Relevance: This function provides the foundational operation for constructing a graph-based model of relationships between languages. By representing languages as nodes and their similarities as edges, it allows the project to visualize and analyze linguistic connections as a network structure.

3.1.2 `diagonal_average(matrix)`

Purpose: Calculates the average value of the diagonal elements in a given matrix, typically representing the degree of correspondence between identical elements (e.g., the same words across languages).

Description:

- Determines the valid diagonal range by selecting the smaller of the matrix's two dimensions (`min(len(matrix), len(matrix[0]))`), ensuring compatibility with non-square matrices.
- Extracts all diagonal elements (`matrix[i][i]` for each valid index).
- Returns the arithmetic mean of the diagonal elements if they exist; otherwise, returns 0.

Relevance: This function serves as a compact metric for summarizing the overall similarity between two corresponding word lists. By focusing on the diagonal, it captures how closely related paired words are between two languages, offering an intuitive measure of lexical alignment or semantic correspondence.

3.2 main.py changes

Overview: The new version of `main.py` significantly extends the functionality of the program. While the original version focused solely on reading data, translating words, computing similarity matrices, and saving results to CSV files, the updated version introduces a *graph-based visualization* component. This enhancement allows users to visualize inter-language relationships based on computed similarity scores.

Key Modifications and Additions:

1. **New Imports:**
   - `networkx` (`import networkx as nx`) — added to support graph creation and manipulation.
   - `matplotlib.pyplot` (`import matplotlib.pyplot as plt`) — introduced for visualizing and saving similarity graphs.
   - `add_connection` and `diagonal_average` imported from `utils.overall_similarity` — new helper functions used for constructing the similarity network and summarizing matrix data.
2. **Graph Creation:**
   - A new `networkx.Graph()` instance (`G`) is created at the start of each topic iteration.
   - For each pair of languages, the script calculates the average similarity using the `diagonal_average()` function.
   - The resulting value (converted to a percentage) is used as a labeled edge between the two language nodes in the graph via `add_connection(G, lang1, lang2, label)`.
3. **Graph Visualization:**
   - Node positions are determined using the `spring_layout()` algorithm with a fixed random seed (`seed=42`) to ensure reproducible layouts.
   - The nodes are drawn with consistent styling:
     - Node color: light blue.
     - Node size: 2000.
     - Labels and edge labels are displayed, with edges annotated using the stored similarity percentages.
   - A plot title is generated dynamically based on the current topic (`plt.title(f"{topic} related words similarity")`).
4. **Graph Output:**
   - Each generated graph is saved as a PNG image in the results directory, named after the processed topic (e.g., `animals_similarity_graph.png`).

- The image is saved with a high resolution (`dpi=300`) and tight bounding box to ensure quality and clean formatting.

5. **Directory Structure and Path Adjustments:**

   - Paths for `data_dir` and `results_dir` have been updated to include relative parent directories (`"../data"` and `"../results"`), likely to align with a reorganized project structure where `main.py` is now placed inside a `src` subdirectory.

6. **Functional Continuity:**

   - All core steps from the previous version remain unchanged:
     (a) Loading word lists.
     (b) Translating words into multiple languages.
     (c) Computing word-by-word similarity matrices.
     (d) Saving translation and similarity data to text and CSV files.
   - The new version builds upon this pipeline by adding visualization and summary statistics, not replacing any existing functionality.

Summary of Impact: The new version transforms the program from a purely data-processing pipeline into a visually interpretable analytical tool. By integrating graph-based visualization, it provides a clear, intuitive overview of inter-language similarity patterns for each topic, making results more accessible for linguistic analysis and presentation.

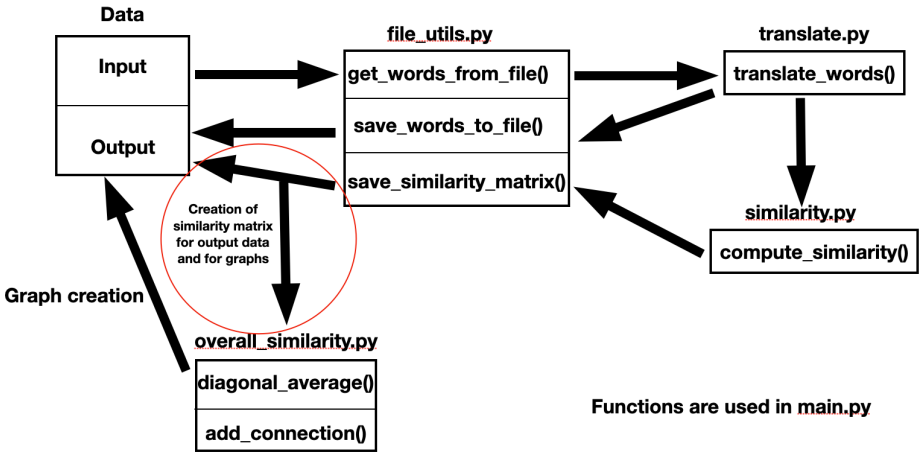
4 Problems that we're facing

Speed: Right now our program is working really slow - it needs much time for compilation. In order to fix that we plan to change some things. First - it shouldn't generate all translations, tables and graphs at once. It should be divided into smaller parts. For example user should be able to choose for example four languages that he want to compare with each other so that program won't have to analyze languages that user isn't interested in. This will improve its performance. Also maybe there is a place for improvement of similarity tables. Right now they compare every world with every world between two languages in one category even if they don't share the same meaning, and this comparison is the only one that interests us.

5 November 23rd 2025

Currently we are working on better user's interface (window instead of console) and word embedding - more advanced way of calculation words similarity. Also we're adding new topics of words.

6 Scheme of data flowing through functions from different files



7 Comparison by embedding

This will be the most advanced part of a project. Next few sections will cover this very topic. The goal is to create a language proximity analysis system utilizing:

- Phonetic dictionaries (IPA) from WikiPron
- Word2Vec algorithm (CBOW) for learning vector representations
- Visualization tools for result analysis

8 System Architecture (embedding)

The system consists of the following modules:

8.1 Data Extraction Module (`src/data/data_scraping`)

8.1.1 Phoneme Extraction

The `phoneme_extractor.py` script downloads phonetic dictionaries from WikiPron:

- Support for over 100 Latin-script languages
- Output format: word + IPA transcription (tab-separated)

- Filtering words containing digits and leading apostrophes
- Saving to `data/<lang>/phonemes.txt`
- Automatic skip if data already exists

8.2 Dataset Module (`src/data/datasets`)

8.2.1 MultilingualWordDataset

PyTorch dataset for character-level data: [language=Python] class MultilingualWordDataset(Dataset): """ Returns: (character, language$_{l}abel$)"""

8.2.2 MultilingualPhonemeDataset

PyTorch dataset for phonetic data: [language=Python] class MultilingualPhonemeDataset(Dataset): """ Parses IPA strings into individual phonemes Returns: (phoneme, language$_{l}abel$)"""

8.3 Word2Vec Module (`src/embeding`)

8.3.1 CBOW Model

Implementation of the Continuous Bag of Words algorithm [4]: $\mathrm{h} = \sum_{-c \leq j \leq c, j \neq 0} \mathbf{v}_{w_{t+j}}$ $p(w_t | w_{t-c}, \ldots, w_{t+c}) = softmax(\mathbf{Wh})$

where:

- c – context window size
- $\mathbf{v}_{w_i}$ – context word embedding
- $\mathbf{h}$ – summed context vector
- $\mathbf{W}$ – output layer weight matrix

8.3.2 Network Architecture

- **Input layer**: Embedding ($|V| \times d$)
- **Hidden layer 1**: Sum of context embeddings
- **Hidden layer 2**: ReLU activation (128 units)
- **Output layer**: Linear ($128 \times |V|$) + LogSoftmax
- **Loss function**: Cross-entropy
- **Optimizer**: Adam

8.3.3 Hyperparameters Configuration

Training parameters are centralized in `src/embeding/hyperparamiters.py`:

- `LANGUAGES`: List of language codes
- `DATA_TYPE`: 'words' or 'phonemes'
- `EMBEDDING_DIM`: Embedding dimension (default: 100)
- `WINDOW_SIZE`: Context window size (default: 2)
- `EPOCHS`: Number of training epochs (default: 10)
- `BATCH_SIZE`: Batch size (default: 128)
- `LEARNING_RATE`: Learning rate (default: 0.001)

8.4 Logging System (`src/logging`)

Centralized logging configuration in `logging_config.py`:

- Separate log files per module in `logs/` directory
- Console output with INFO level
- File output with DEBUG level and timestamps
- Format: `YYYY-MM-DD HH:MM:SS - module - LEVEL - message`

8.5 Testing Module (`tests/`)

8.5.1 Dataset Verification

The `verify_datasets.py` script:

- Loads datasets and writes content to files
- Compares dataset output with original files
- Detects parsing errors (e.g., tab characters in word data)
- Validates both MultilingualWordDataset and MultilingualPhonemeDataset

9 Usage Instructions (embedding)

9.1 Data Extraction

9.1.1 Single Language

[language=bash] Phonemes python src/data$_s$*craping/phoneme*$_e$*xtractor.pypl*

9.1.2 Multiple Languages (Runner)

[language=bash] python src/data/runner.py pl en de

9.2 Model Training

Training parameters are configured in `src/embeding/hyperparamiters.py`: [language=Python] Edit hyperparamiters.py LANGUAGES = ['pl', 'en'] DATA$_TYPE = '$ $phonemes' or 'words' EPOCHS = 10 EMBEDDING_DIM = 100$

Then run training: [language=bash] python src/embeding/train$_c bow.py$

Models are saved to `models/` directory with timestamp.

10 Conclusions (embedding)

10.1 Achievements

1. Implemented complete pipeline for data extraction from Wikipedia
2. Integrated WikiPron phonetic dictionaries (337 languages)
3. Created PyTorch datasets for multilingual data
4. Implemented CBOW algorithm from scratch
5. Created visualization tools (t-SNE, PCA, heatmaps)

10.2 Observations

- CBOW model effectively learns character representations
- Characters with similar usage have higher cosine similarity
- Training loss decreases steadily
- t-SNE/PCA visualizations show sensible structure in embedding space

10.3 Future Directions

1. Training on larger datasets (full Wikipedia dumps)
2. Implementation of Skip-gram as alternative to CBOW
3. Cross-lingual analysis (comparing embeddings of different languages)
4. Negative sampling for training acceleration
5. Hierarchical softmax
6. Visualization tools (t-SNE, PCA, similarity heatmaps)
7. Automated testing suite

11 References (embedding)

1. Mikolov, T., et al. (2013). "Efficient Estimation of Word Representations in Vector Space." arXiv:1301.3781

2. Mikolov, T., et al. (2013). "Distributed Representations of Words and Phrases and their Compositionality." NIPS 2013

3. WikiPron: https://github.com/CUNY-CL/wikipron

4. Zhang, A., et al. Dive into Deep Learning. https://d2l.ai/chapter$_natural-language-processing-pretraining/word2vec.html$

A Project Structure (embedding)

```
project/
|-- data/                     # Extracted data
|   `-- <lang>/
|       `-- phonemes.txt    # Word + IPA
|-- src/
|   |-- data/
|   |   |-- data_scraping/  # Extraction scripts
|   |   |-- datasets/       # PyTorch datasets
|   |   `-- runner.py       # Data pipeline
|   |-- embeding/
|   |   |-- cbow.py         # CBOW model
|   |   |-- train_cbow.py   # Training script
|   |   `-- hyperparamiters.py  # Config
|   `-- logging/
|       |-- logging_config.py   # Logger setup
|       `-- add_logging.py      # Helper script
|-- tests/
|   `-- verify_datasets.py  # Dataset tests
|-- logs/                   # Log files
|-- models/                 # Trained models
`-- report/                 # Documentation
```

B Configuration Parameters (embedding)

Parameter	Default value	Description
EMBEDDING_DIM	100	Embedding dimension
WINDOW_SIZE	2	Context window size
EPOCHS	10	Number of epochs
BATCH_SIZE	128	Batch size
LEARNING_RATE	0.001	Learning rate
MAX_TOKENS	None	Token limit (None = all)
DATA_TYPE	'phonemes'	'words' or 'phonemes'
LANGUAGES	['pl']	List of language codes

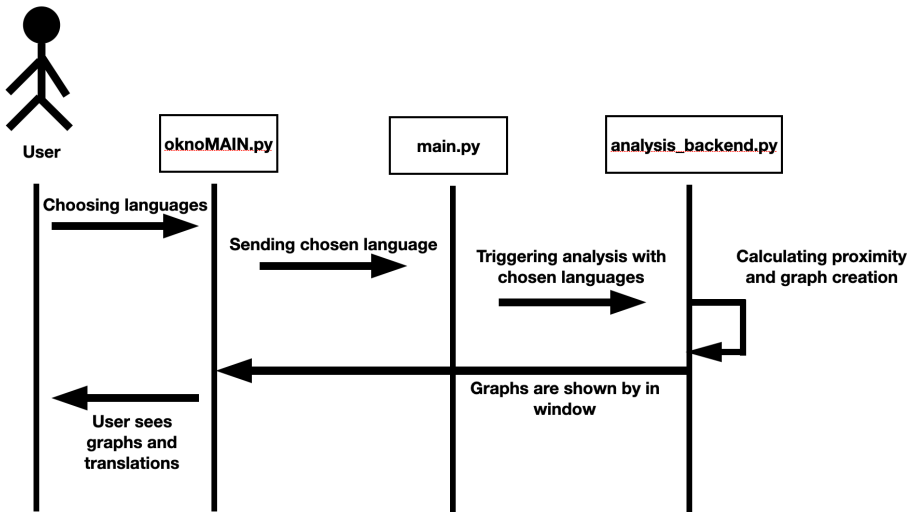
Table 1: CBOW training configuration parameters (`hyperparamiters.py`)

C December 18th - big progress

Embedding: Comparison by embedding was added. This contains trained and prepared models, all required .py and .sh files. This part isn't as versatile as Levensthein comparison because models for embedding are prepared for exact set of languages while Levensthein comparison works for any language (even for fictional words that doesn't exist in any), but at the same time is more advanced.
Front-end: Up to this point we've been working only on back-end - translating, comparing and getting results. However, program should look clearly not only for its creators, but also for its users. For this reason we've added part of the code that creates a window in which user may display translations, categories and comparison results.

D December 30th - important changes

Project structure had some important changes. It was mentioned before that we need front-end and backend divided from backend. From now our project we'll have three most important files - back-end one that's now called `analysis_backend.py`. This file will do proximity calculations and generate graphs of similarity. Next one is front-end one for now called oknoMAIN.py. This file will create a window for user - in this one he may choose which languages he wants to compare with each other and view results of the comparison shown on graphs. There will be also a file main.py that will connect those two files. It's simple and shown on the next page:


User
oknoMAIN.py
main.py
analysis_backend.py
Choosing languages
Sending chosen language
Triggering analysis with chosen languages
Calculating proximity and graph creation
Graphs are shown by in window
User sees graphs and translations