Table of Contents

1. Spelling Correction System

1.1 Introduction to Candidate Techniques

Spelling corrections are used by most people on a day-to-day basis – for example when typing messages on the phone or using search engines (Huang & Li, 2021). It helps ensure clear and accurate written communication, which is particularly important in the medical field. Proper electronic health records are essential to clinical well-being and dependable research. Mistakes in spelling of free-text entries (including clinical notes, allergy entries, or medication orders) may cause misinterpretation, medical errors, and patient-related risks. Automatic spelling mistakes correction is necessary, as most medical data is still typed in free text that is not structured (Hier et al., 2024).

For this reason, the first part of this assignment deals with the theoretical background of spelling correction models based on Natural Language Processing (NLP). The second part of this assignment consists of the design of a probabilistic spelling correction model and its implementation.

The work by Toleu et al. (2022) summarizes available spelling correction systems, most of which use three steps: detecting errors, generating candidate corrections, and ranking those candidates. The classical methods tend to rely on dictionary lookups, edit distance metrics like Damerau -Levenshtein, and noisy channel models for detection and correction symmetry. The newer approaches combine phonetic and contextual aware models such as ngram and neural network-based models to manage real-word errors more successfully. The goal of candidate generation methods is to reduce the search space by means of such techniques as Levenshtein or embedding-based similarity joins. To rank candidates, the noise channel models are used together with the contextual features to make more precise corrections, which demonstrates that incorporating the error and language models leads to better performance.

Spelling correction generally differentiates between three types of errors: detecting non-word errors, correcting isolated-word errors, and correcting context-dependent errors. Non-word errors are words which are wrongly spelt and simply do not exist (Example: “The patient was

prescribed *aspitin* for pain relief" – The word "aspitin" is a non-existent word). The main two procedures to detect non-word errors are dictionary pairing and N-gram statistical analysis. Dictionary pairing, also called lexicon lookup, checks every token against an vocabulary/dictionary. If the token cannot be found in this vocabulary, the word is identified as a non-word. The N-gram statistical analysis measures the probability of character or word sequences depending on their frequency in a reference corpus to enable the system to mark tokens that strongly diverge in the common language patterns as possible non-words (Huang & Li, 2021).

Isolated-word corrections are commonly performed using edit distance, especially DamerauLevenshtein distance and will be described in more detail in Chapter 1.2.

Real-word errors are words which actually exist, however are used in the wrong context (Example: "He was admitted after a *hearth* attack" – The word "hearth" exists and is spelt correctly, however in this context, "*heart* attack" would be correct). This is especially dangerous in medicine, because a naive spell checker would not flag it as an error. Context models, however, can identify such real-word mistakes. Ideally, a spelling correction model can correct both kinds of errors (Huang & Li, 2021; Luitel et al., 2024).

One probabilistic approach to enable context-dependent corrections are noisy channels. It does not correct the observed text directly but estimates the most probable intended word under the observed word and the likelihoods of various sources of errors. The core components of noisy channel spelling corrections are language models, error model and candidate generation (Luitel et al., 2024).

A prerequisite of correcting real-word errors is contextual understanding, which can be achieved with language models (LMs). For a low-resource setting, a small corpus, as used in the following chapters of this assignment, is sufficient to build such a model. Language Models compute the "probability that a given word occurs in the presence of its context" (Luitel et al., 2024). For this reason, a list words which might be actually intended (candidate words) can be created and ranked with the help of these probabilities (Luitel et al., 2024).

Although language models are able to create contextual understanding, they must still be combined with error models to weight which corrections are actually likely typos. Error models assign higher weighting to candidate words that are likely to generate the observed error form and often work with edit distance, which will be discussed in more detail later on. In order to build a reliable spelling correction system, one has to combine language and an error models. This combination is often modelled through Bayesian inference, which computes the posterior probability of candidate corrections, based on which one can generate a ranking of candidates. As part of this process, candidate generation plays a crucial role, since possible correction options for a misspelled word first need to be produced before they can be ranked (Luitel et al., 2024).

This work presents a spelling correction algorithm that takes advantage of similarity-based spelling corrections via pretrained bio-wordvector word embeddings to treat typos in medical reports. This process, as opposed to rule-based systems, which are limited to dictionaries, can fix mistakes without a predefined set of terms and so it is applicable to the medical field where terminology is complicated and dynamic. Combination of edit distance with embedding similarity results in candidate words after which ranking is done according to frequency and similarity metrics. It was found in experiments that the model was able to correct 12 categories of typographical errors and had an F1 score of 97.48, compared to the traditional rule-based typographical correction systems, like SymSpell. This indicates that embedding-based unsupervised learning is effective in spelling correction in medical text which is specialized (Kim et al., 2021).

Some recent research confirms the usefulness of tools and libraries like those applied in this project. As an example, Bravo-Candel et al. (2021) use pretrained Word2Vec and GloVe embeddings (loaded with a tool such as gensim) to provide context to correct real-word mistakes in Spanish clinical texts. Dashti et al. (2024) demonstrate better Persian medical document correction through text preprocessing, lexical and embeddings. Work in Turkish (Turhan et al., 2025) is yet another example that large language models, as well as metrics of vector similarity, can be utilized to fix spelling in less-resourced, agglutinative languages. Moreover, Laukaitis et al. (2024) provide increased coverage and usefulness to WordNet based resources, which allows nltk.corpus.wordnet to be used in candidate generation and semantic filtering. These results confirm that embedding libraries such as gensim are

designed to be used in this project, and the semantics of the lexicon, the lexical semantics, and the other Python tools in preprocessing and counting statistics are used in this project.

1.2 Edit Distance

The idea of edit distance, also called Levenshtein-distance, is a crucial and widely applied element in correction spelling. It is part of both error modelling and candidate generation (Luitel et al., 2024; Chaabi & Allah, 2021). Edit distance is a string similarity metric that measures how many single-character operations are required to transform one word into another (Huang & Li, 2021). Similarity is measured by computing the minimum number of operations – including insertions, deletions and substitutes. In some earlier analyses, transposition of adjacent characters was also included as an operation, as suggested by Damerau's study. The later method is known as Damerau-Levenshtein-distance and will become relevant in the implementation part of this assignment.

Luitel et al. (2024) recommend an edit distance of 2. Hence, a spelling correction system is searching for candidate words in the vocabulary which are within the corresponding edit distances (Luitel et al., 2024). This will also be the case for the applied part of this assignment.

However, there are limitations coming with edit distance. Assuming that the system suggests several candidate words for a certain misspelt word and all have the same minimal distance, the approach is not capable of ranking one solution above another. As a consequence, correction lists are often unsatisfactory, since plausible and implausible candidate words are treated equally. For this reason researchers introduced weighted versions of edit distance. Instead of all mistakes having the same cost, more common errors are given a lower cost and less likely ones a higher cost. Furthermore, hybrid models gained popularity, since they implement language information (like grammar or context) to decide which correction makes the most sense in a sentence. According to the authors, this combination is particularly effective with segmentation errors, i.e. two valid words can be obtained due to an erroneous addition of a space (e.g., foot ball vs. football). In that situation, edit distance will fail to determine which form is right, but it may be enabled by combining bigram probabilities to promote the candidate that is closer to the surrounding situation. One way to incorporate

contextual meaning is to combine n-gram language models with edit distance (Huang & Li, 2021).

N-gram models are able to predict the probability of a word based on the words that come before it. Unigrams are not suitable to incorporate contextual meaning, since each word's probability is considered in isolation. Bigrams (n = 2) and trigram (n = 3) models use the previous word (or previous words) to estimate the probability of the current word (Huang & Li, 2021). An example for that in the medical field is "*heart attack*". Assuming one uses a medical corpus, the probability of the word *"attack"* occurring after *"heart"* is much higher than after unrelated words, which shows how n-gram models capture contextual patterns that enable more accurate spelling correction. Contrary to this, "*heart banana"* would have a near-zero probability, since this combination of words cannot be found in the corpus.

The paper notes that n rarely exceeds 3 in practice, because higher orders require both very large corpora and become computationally heavy. The prerequisite for implementing bigram models is a corpus which is sufficiently large to include all common sequences of words in the respective field or language (Huang & Li, 2021).

Bigram Probability:

$$P(w_i \mid w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$$

Sentence Probability (in a bigram model):

$$P(w_1, w_2, \ldots, w_n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-1})$$

1.3 Formulation/Design

The objective of this assignment is to create a spelling correction editor with an appropriate graphical user interface. It is supposed to be a probabilistic model for detecting spelling errors. The system has to differentiate between non-word and real-word errors (cf. Chapter 1.1). Furthermore, the spelling correction tool should not just detect spelling errors, but also

suggest correction for the misspelt words. The corrections are then ranked using multiple techniques, such as minimum edit distance, bigram probabilities and hybrid corpus frequencies.

The correction system is based on two corpora – one corpus contains scientific abstracts from the biomedical background and the other contains random English sentences to cover general English language. For both corpora, unigram and bigram counts are calculated and combined later on to boost accuracy.

Language Corpora:

- *PubMed 200k RCT:* https://www.kaggle.com/datasets/matthewjansen/pubmed-200krtc
- *Random English Sentences:* https://www.kaggle.com/datasets/nikitricky/randomenglish-sentences

In regards to error detection, inputs from users of the spelling correction editor are tokenized with Regular Expressions. Non-words are then checked simply by comparing it against the entire vocabulary of both corpora, the real-words are determined by using bigram language models and comparing the contextual probability.

Since the spelling correction tool must also be able to give suggestions for correction, candidate words must be generated. With the help of Edit Distance, candidates are generated up to a distance of 2.

The GUI of the text editor can accept a maximum number of 500 characters for user input and highlights spelling errors. When clicking on the misspelt word, it shows correction suggestions with their corresponding Minimum Edit Distance scores in a ranked order. Moreover, it provides a Corpus Browser, which is a sorted list of all words by frequency with a search functionality.

The correction spelling editor was tested with the help of example sentences (both with medical and non-medical contexts) generated by ChatGPT. The limitations will be described in Chapter 1. 5.

1.4 Implementation

a) Import necessary libraries and medical text corpus

```
#Import libraries and load the CSV
import pandas as pd
import re
from collections import Counter
from math import log
import math
from nltk.corpus import wordnet as wn
import nltk
import tkinter as tk
from tkinter import ttk, messagebox
from numpy.linalg import norm
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec

# Load the dataset
df = pd.read_csv("NLP_train.csv")

# Displaying the first rows
print(df.head())
print(df.info())
```

b) Build the medical corpus and tokenization

```
#Build the medical corpus
# join all abstract_text into one big corpus
texts = df["abstract_text"].astype(str).tolist()
corpus = " ".join(texts)

# tokenize: keep letters and '@' (numbers replaced in this dataset)
TOK_RE = re.compile(r"[A-Za-z@]+")
tokens = TOK_RE.findall(corpus.lower())

len(tokens), tokens[:20]
```

```
(50183758,
 ['the',
  'emergence',
  'of',
  'hiv',
  'as',
  'a',
  'chronic',
  'condition',
  'means',
  'that',
  'people',
  'living',
  'with',
  'hiv',
  'are',
  'required',
  'to',
  'take',
  'more',
  'responsibility'])
```

During tokenization, the whole medical corpus is split into lowercase and word-like units, where a word is defined as a sequence of letters or “@”. Number were replaced by @ in the Kaggle dataset. Below the code, one can see that the corpus contains more than 50 million tokens. Additionally, the first 20 tokens of the corpus are displayed.

c) Creation of unigram and bigram count, probability distribution and vocabulary

```
#Build Unigram and Bigram Models
uni = Counter(tokens)
bigrams = list(zip(tokens[:-1], tokens[1:]))
bi = Counter(bigrams)

# totals
N_tokens = sum(uni.values())
V = len(uni)

# Probability Smoothing
def p_uni(w, k=1.0):
    return (uni.get(w, 0) + k) / (N_tokens + k * V)

def p_bi(w1, w2, k=1.0):
    return (bi.get((w1, w2), 0) + k) / (uni.get(w1, 0) + k * V)

# vocab set
VOCAB = set(uni.keys())
```

At this point, the unigram (for non-word errors) and the bigram models (for real-word errors) were built. That means that the unigram and bigram counts were turned into a probability distribution. La Place Smoothing was applied to avoid 0-probabilities. On top of that, the first (solely medical-based) vocabulary was built as a set in Python, since it enables a fast membership test.

d) Creation of a frequency-sorted list of all unique tokens from the corpus

```
# full, frequency-sorted list (descending)
sorted_vocab = sorted(uni.items(), key=lambda x: (-x[1], x[0]))

def search_vocab(q, top=30):
    q = q.lower()
    return [(w, c) for (w, c) in sorted_vocab if q in w][:top]

# examples
sorted_vocab[:20]
print(sorted_vocab[:20])
```

```
[('@', 3750167), ('the', 2173891), ('of', 1715701), ('and', 1670719), ('in', 1345241), ('to', 939291), ('with', 755421), ('a', 750779), ('were', 653116), ('was', 595127), ('patients', 524715), ('for', 488957), ('group', 427238), ('p', 359232), ('or', 338495), ('at', 293007), ('treatment', 247640), ('on', 237590), ('study', 233969), ('after', 213605)]
```

This frequency sorted list is needed later on to fulfil the requirement of the Corpus Browser in the editor. Below the code, you can see the 20 most frequent tokens and how often they appear. ‘@’ (replacing numbers) appears the most often (3750167 times), followed by ‘the’ and ‘of’.

e) Candidate generation with minimum edit distance (= 2)

```python
#Candidate Generation with Minimum Edit Distance
alphabet = "abcdefghijklmnopqrstuvwxyz@"

def edits1(word):
    splits = [(word[:i], word[i:]) for i in range(len(word)+1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R) > 1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in alphabet]
    inserts    = [L + c + R               for L, R in splits for c in alphabet]
    return set(deletes + transposes + replaces + inserts)

def known(words):
    """Return only words that are actually in the vocabulary."""
    return [w for w in words if w in VOCAB]

def candidates(word, max_edits=2):
    """Generate spelling correction candidates for a word."""
    cands = set()
    e1 = edits1(word)
    cands.update(known(e1))
    if max_edits >= 2:
        for w in list(e1)[:5000]:
            cands.update(known(edits1(w)))
    if word in VOCAB:
        cands.add(word)
    return list(cands)
```

At this point, the candidate generation step for spelling correction is implemented. This code creates all possible correction candidate words within 2 edits of the input, including deletion, insertion, substitution and transposition of characters and then checks if they exist in the vocabulary. This ensures, that the system offers a set of candidate correction after any misspelling. These candidates can be ranked later, by frequency or context.

f) Dropping '@' from Suggestions

```python
#Drop @ from suggestions

USE_AT = False
if not USE_AT and '@' in VOCAB:
    VOCAB.discard('@')
```

Dropping the placeholder '@' (@ replaced all numbers in the corpus). Keeping it in the corpus would artificially inflate the vocabulary with a meaningless token that doesn't contribute to spelling correction or context modeling.

g) Contextual scoring with smoothed probabilities

```python
#La Place Smoothing for Candidate Words

def p_uni_smoothed(w, k=1.0):
    return (uni.get(w, 0) + k) / (N_tokens + k * len(uni))

def p_bi_smoothed(w1, w2, k=1.0):
    return (bi.get((w1, w2), 0) + k) / (uni.get(w1, 0) + k * len(uni))

def logscore_context(left, w, right, k=1.0, lam=0.7):
    """
    Interpolate bigram(s) with unigram:
      score = lam*(log P(w|left) + log P(right|w)) + (1-lam)*log P(w)
    If left/right missing, those terms drop out.
    """
    s = 0.0
    used = 0
    if left:
        s += log(p_bi_smoothed(left, w, k)); used += 1
    if right:
        s += log(p_bi_smoothed(w, right, k)); used += 1
    if used:
        return lam * s + (1 - lam) * log(p_uni_smoothed(w, k))
    else:
        return log(p_uni_smoothed(w, k))
```

To make sure that no candidate correction gets completed discarded, La Place Smoothing is applied again to calculate unigram and bigram probabilities. With the help of interpolation, these probabilities are combined to score how well a candidate word fits in its left and right context. This ensures that the spelling correction model prefers contextually plausible words.

h) Calculation of the Damerau-Levenshtein distance between the original word and the candidate word

```python
#Damerau-Levenshtein distance between the original word and the candidate word
def approx_edit_distance(a, b):
    # True Damerau-Levenshtein: insertions, deletions, substitutions, transpositions
    return nltk.edit_distance(a, b, transpositions=True)
```

Since the candidates need to show a score in the GUI, the Damerau-Levenshtein distance between the original word and the candidate word is calculated and then combined with other scores (like bigram probability and semantics) later in order to derive an overall ranking of plausible corrections. The use of Damerau–Levenshtein distance is suitable here, since it extends the standard edit distance by also considering transpositions of adjacent characters (e.g., teh → the), which are among the most common human typing errors. This makes the candidate generation and scoring process more robust and realistic.

i) User Input Tokenization

```python
#User Input Tokenization

TOK_RE = re.compile(r"[A-Za-z@]+")

def tokenize_user(text):
    return TOK_RE.findall(text.lower())
```

The code tokenizes the input of the spelling correction editor. The tokens are compared to the vocabulary, scored with the unigram/bigram models, and passed through the error models (edit distance etc.) to detect & suggest corrections.

j) Baseline Suggestion Ranking (**for the Medical Corpus only**, no optimizations included yet).

```
def suggest_for_token(tokens, idx, topk=5, k=1.0, lam=0.7, edit_penalty=0.75):
    """
    Simple baseline suggestion generator (medical corpus only).
    No POS, no IDF, no normalization.
    """
    w = tokens[idx].lower()
    left  = tokens[idx-1].lower() if idx-1 >= 0 else None
    right = tokens[idx+1].lower() if idx+1 < len(tokens) else None

    cand_list = candidates(w, max_edits=2)
    if not cand_list:
        return []

    ranked = []
    for c in cand_list:
        ed = approx_edit_distance(w, c)
        # force numeric conversion to avoid str issues
        raw_score = logscore_context(left, c, right, k=k, lam=lam)
        score = float(raw_score) - edit_penalty * ed
        ranked.append({
            "cand": c,
            "score": round(score, 3),
            "edit": ed
        })

    ranked.sort(key=lambda x: (-x["score"], x["edit"], x["cand"]))
    return ranked[:topk]
```

This is the core suggestion function of candidate words. It takes a wrongly spelt words and generates candidate corrections. It then assigns a score to each candidate by combining the language model with edit distance. It favors words that are closer to the original spelling and return the suggestion with the highest score in the GUI (edit_penalty).

k) Detect and Suggest Corrections

```
def detect_and_suggest(text, topk=5, ctx_margin=1.0):
    toks = tokenize_user(text)
    results = []

    for i, w in enumerate(toks):
        w = w.lower()
        if w == '@':   # skip placeholder tokens
            continue

        in_vocab = w in VOCAB
        suggestions = suggest_for_token(toks, i, topk=topk)

        # If it's a non-word: show suggestions if any
        if not in_vocab and suggestions:
            results.append({
                "index": i,
                "word": w,
                "type": "NON-WORD",
                "suggestions": suggestions   # already dicts
            })
            continue

        # If it's a real word: only flag if best alternative beats the original by ctx_margin
        if in_vocab and suggestions:
            left  = toks[i-1].lower() if i-1 >= 0 else None
            right = toks[i+1].lower() if i+1 < len(toks) else None
            orig_score = float(logscore_context(left, w, right))

            best_cand = suggestions[0]["cand"]
            best_score = suggestions[0]["score"]
            best_ed = suggestions[0]["edit"]
```

```
            if best_cand != w and (best_score - orig_score) >= ctx_margin:
                results.append({
                    "index": i,
                    "word": w,
                    "type": "REAL-WORD?",
                    "orig_score": round(orig_score, 3),
                    "best_delta": round(best_score - orig_score, 3),
                    "suggestions": suggestions   # already dicts
                })

    return toks, results
```

This function processes the sentence(s) of the user. If the token cannot be found in the vocabulary, it is identified to be a non-word and correction suggestions are shown. If the word is in the vocabulary but context (= the interpolation of unigrams and bigrams = raw_score) suggests a better alternative, it is flagged as a real-word and better candidates are suggested (including their scores).

l) Testing the detect_and_suggest function with a more lenient context margin

```
tests = [
    "The patints were given aspirin daily.",      # patients
    "The study focused on diabtes treatment.",    # diabetes
    "He suffered from a hearth attack.",          # heart
    "The drug aspitin was administered.",         # aspirin
]

for t in tests:
    toks, res = detect_and_suggest(t, topk=5, ctx_margin=0.8) #more lenient ctx_margin
    print("\nINPUT:", t)
    print("TOKENS:", toks)
    for r in res:
        print(r)
```

m) Error Replacement

```python
def apply_corrections(text, results, ctx_only=True):
    """
    ctx_only=True: for REAL-WORD? apply only when flagged (i.e., delta >= margin)
    """
    toks = tokenize_user(text)
    idx_to_best = {}
    for r in results:
        if not r.get("suggestions"):
            continue
        best = r["suggestions"][0]["cand"]
        if r["type"] == "NON-WORD":
            idx_to_best[r["index"]] = best
        elif r["type"] == "REAL-WORD?":
            # already passed the margin filter in detect_and_suggest
            idx_to_best[r["index"]] = best

    corrected = [ (idx_to_best[i] if i in idx_to_best else t) for i, t in enumerate(toks) ]
    return " ".join(corrected)
```

This function is the final correction step, since it takes the results from the detection stage and automatically replaces errors with their top-ranked suggestions. If a words is identified as a non-word, always replace it. If it is a real-word, only replace it if it passed the margin filter (ctx_margin = 1) in detect_and_suggest.

n) Testing detect_and_suggest for the medical-only corpus

```python
#Testing detect_and_suggest for the medical corpus
text = "The patints suffered a hearth attack."
toks, res = detect_and_suggest(text)
print("TOKENS:", toks)
for r in res:
    print(r)

corrected = apply_corrections(text, res)
print("\nCorrected:", corrected)
```

```
TOKENS: ['the', 'patints', 'suffered', 'a', 'hearth', 'attack']
{'index': 1, 'word': 'patints', 'type': 'REAL-WORD?', 'orig_score': -23.673, 'best_delta': 12.572, 'suggestions': [{'cand': 'patients', 'scor
e': -11.101, 'edit': 1}, {'cand': 'patient', 'score': -13.414, 'edit': 2}, {'cand': 'parents', 'score': -18.068, 'edit': 2}, {'cand': 'point
s', 'score': -19.623, 'edit': 2}, {'cand': 'ratings', 'score': -19.975, 'edit': 2}]}
{'index': 4, 'word': 'hearth', 'type': 'REAL-WORD?', 'orig_score': -22.731, 'best_delta': 8.645, 'suggestions': [{'cand': 'heart', 'score': -
14.086, 'edit': 1}, {'cand': 'health', 'score': -16.241, 'edit': 1}, {'cand': 'healthy', 'score': -17.504, 'edit': 2}, {'cand': 'heat', 'scor
e': -19.692, 'edit': 2}, {'cand': 'death', 'score': -19.743, 'edit': 2}]}

Corrected: the patients suffered a heart attack
```

A series of test sentences with intentional spelling inaccuracies was developed to test the hybrid spell-checker. The system tokenizes the input and identifies misspelled words and produces the ranked correction suggestions with relevant scores and edit distances per sentence.

o) Adding a general English corpus to boost the performance of the spelling correction system (Loading and tokenization of general English corpus)

```python
#Adding a general English corpus to boost the performance of the spelling correction (Loading and tokenization of general English corpus)

# Load the general English corpus
with open("sentences.txt", "r", encoding="utf-8") as f:
    general_text = f.read()

# Tokenize: keep only letters
general_tokens = TOK_RE.findall(general_text.lower())
print("General corpus size:", len(general_tokens))
print("Sample:", general_tokens[:20])
```

```
General corpus size: 9076
Sample: ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog', 'my', 'mum', 'tries', 'to', 'be', 'cool', 'by', 'saying', 'th
at', 'she', 'likes']
```

Identical to step b, the second dataset containing random English sentences was loaded and tokenized. With 9076 tokens is the general English corpus significantly smaller than the medical corpus of over 50 million tokens. This step was needed, since the editor flagged very basic words like “went” or “walk”, since they are rarer in the medical corpus.

p) Building unigram and bigram counts for the general English corpus

```python
#Building unigram and bigram counts for general corpus

uni_general = Counter(general_tokens)
bi_general = Counter(zip(general_tokens[:-1], general_tokens[1:]))

print("Unique words in general corpus:", len(uni_general))
```

This step creates the unigram and bigram counts similar to step c. This corpus has 2563 unique words.

q) Building a hybrid corpus

```python
#Building the hyprid corpus

# Merge unigram counts
uni_hybrid = uni + uni_general   # uni = your PubMed unigrams

# Merge bigram counts
bi_hybrid = bi + bi_general      # bi = your PubMed bigrams

# Hybrid totals
N_tokens_hybrid = sum(uni_hybrid.values())
V_hybrid = len(uni_hybrid)

# Hybrid vocab
VOCAB_HYBRID = set(uni_hybrid.keys())
```

Since both corpora combined are supposed to be the basis of the spelling correction editor, an overall hybrid corpus is created. On top of that, the hybrid vocabulary was built as a set in Python, since it enables a fast membership test.

r) Semantic Context Scoring

```python
#Semantic Context Scoring
def get_context_vector(tokens, idx, window=2):
    ctx_words = []
    for j in range(max(0, idx-window), min(len(tokens), idx+window+1)):
        if j != idx and tokens[j] in word_vectors:
            ctx_words.append(word_vectors[tokens[j]])
    if ctx_words:
        return np.mean(ctx_words, axis=0)
    return None

def semantic_similarity(candidate, context_vec):
    if candidate in word_vectors and context_vec is not None:
        cand_vec = word_vectors[candidate]
        return float(np.dot(cand_vec, context_vec) / (norm(cand_vec) * norm(context_vec)))
    return 0.0
```

This part of the code handles the semantic context scoring of the spellchecker. The get_context_vector builds a semantic representation of the local context by averaging embeddinfgs of neighboring words. The semantic_similarity checks how well a candidate word fits semantically into the context, which gives an additional layer of scoring besides frequency and edit distance.

s) Filtering out words from the hybrid corpus that appear less than 15 times

```python
#Filtering out words from the hybrid corpus that appear less than 15 times

token_counts = Counter(tokens)

# filter out very rare words (likely misspellings)
VOCAB_HYBRID = {w for w, c in token_counts.items() if c >= 15}
```

By retaining the rare and blacklisted words in the vocabulary, the system would treat them as valid resulting in a false negative of the common misspellings, which would be no longer corrected. They might also be found in the list of candidates and even ranked higher than the correct words based on frequency, creating noisiness in the unigram and bigram models. Filtering them off gives the vocabulary a clean and trusted source of generating correct corrections.

t) Building hybrid ‘documents’

```
# Build hybrid "documents"
# Use PubMed abstracts + general English sentences
pubmed_docs = df["abstract_text"].astype(str).tolist()
general_docs = general_text.split("\n")

docs = pubmed_docs + general_docs

# Count document frequencies
doc_freqs = Counter()
for doc in docs:
    unique_words = set(TOK_RE.findall(doc.lower()))
    doc_freqs.update(unique_words)

N_docs = len(docs)
print("Docs counted:", N_docs)
print("Sample df:", list(doc_freqs.items())[:10])
```

```
Docs counted: 2212585
Sample df: [('hiv', 13105), ('making', 2672), ('a', 597165), ('management', 18978), ('the', 1264429), ('physical', 22393), ('condition', 890
0), ('as', 133734), ('for', 407227), ('their', 40928)]
```

This built hybrid documents to calculate document frequencies, so that your system can use IDF weighting to favor meaningful terms and avoid overly common words dominating the scoring. A document in the medical corpus is a single abstract and a sentence in the general English corpus. Below you can see the total number of documents and the first 10 entries of that dictionary.

u) Updating the probability functions for the hybrid corpus

```
#Updating probability functions

def p_uni_hybrid(w, k=1.0):
    return (uni_hybrid.get(w, 0) + k) / (N_tokens_hybrid + k * V_hybrid)

def p_bi_hybrid(w1, w2, k=1.0):
    return (bi_hybrid.get((w1, w2), 0) + k) / (uni_hybrid.get(w1, 0) + k * V_hybrid)
```

The smoothed probability estimator for the unigram and bigram models are updated for the hybrid corpus.

v) Making the hybrid vocabulary the new default for the entire system

```
# Hybrid vocab browsing
sorted_vocab_hybrid = sorted(uni_hybrid.items(), key=lambda x: (-x[1], x[0]))

def search_vocab_hybrid(q, top=30):
    q = q.lower()
    return [(w, c) for (w, c) in sorted_vocab_hybrid if q in w][:top]

# --- Aliases so rest of code uses hybrid by default
VOCAB = VOCAB_HYBRID
p_uni = p_uni_hybrid
p_bi = p_bi_hybrid
```

This code makes the hybrid vocabulary the new default for the entire system. It also provides a helper to browse/search it, and ensures all probability functions now run on the hybrid models.

w) Inverse Document Frequency

```
# Inverse Document Frequency

def idf_score(word, doc_freqs, N_docs):
    """Inverse document frequency: penalize very common words."""
    df = doc_freqs.get(word, 1)
    return math.log((N_docs + 1) / (df + 1)) + 1
```

This function assigns a weight to every word according to its level of information in documents. Words that occur everywhere (such as 'the') score low whereas rarer words (such as 'aspirin') score highly and thus are more informative semantically helpful in ranking corrections.

x) Cosine Similarity for Semantic Contextuality

```
#Cosine Similarity for Contextuality
def semantic_similarity(w, ctx_vec):
    """
    Cosine similarity between a candidate word and the context vector.
    Returns a score in [-1, 1], usually 0–1 for embeddings.
    """
    if w not in word_vectors or ctx_vec is None:
        return 0.0
    v = word_vectors[w]
    return float(np.dot(v, ctx_vec) / (norm(v) * norm(ctx_vec)))
```

This function gives the system an extra layer of checking semantics. This function examines the similarity of a candidate word to its context in semantically similar way, with the help of embedded vectors. It uses cosine similarity. If the score is close to 0, there is no relation to the context. If the score is close to 1, the candidate word fits the meaning of its surrounding context.

y) POS-Tagging

```
#POS-Tagging

nltk.download("averaged_perceptron_tagger")

def same_pos(word1, word2):
    """
    Return True if word1 and word2 share the same POS tag.
    """
    try:
        pos1 = nltk.pos_tag([word1])[0][1]
        pos2 = nltk.pos_tag([word2])[0][1]
        return pos1 == pos2
    except:
        return True
```

This function checks whether the word that was typed in and the candidate role have the same grammatical role (Parts-of-Speech Tag). That enables the spell checker to exclude candidate

word that are grammatically different anyways (for example, a wrongly spelt noun shouldn't be replaced by a verb).

z) Preparation and Loading of Pre-Trained Word Embeddings (GloVe)

```
#Preparation and Loading of Pre-Trained Word Embeddings
glove_input_file = "glove.6B.50d.txt"
word2vec_output_file = "glove.6B.50d.word2vec.txt"
glove2word2vec(glove_input_file, word2vec_output_file)

word_vectors = KeyedVectors.load_word2vec_format(word2vec_output_file, binary=False)
```

GloVe embeddings are trained word vectors which are dense numerical vectors, in a high dimension space. GloVe learns semantic and syntactic relationships, unlike one-hot encodings which assume that each word is independent by using the global co-occurrence statistics of large text corpora. This implies that similar meaning words such as "aspirin" and "ibuprofen" will be placed in similar vectors. With semantic similarity in your spelling corrector, you can use GloVe embeddings to rank candidate words according to their similarity with their surrounding environment, meaning the system can select corrections that make more imminent sense, not only statistically but also semantically. This comes in particularly handy with real world mistakes, when both choices are possible words, and only one of them makes sense in the situation (Khan et al., 2022).

A) Detect and Suggest Hybrid

```python
def detect_and_suggest_hybrid(text, topk=5, ctx_margin=0.8, k=3, lam=0.8, edit_penalty=0.5):
    toks = tokenize_user(text)
    results = []

    for i, w in enumerate(toks):
        w_norm = w.lower()
        if w_norm == '@':
            continue

        in_vocab = w_norm in VOCAB_HYBRID
        suggestions = suggest_for_token(toks, i, topk=topk, k=k, lam=lam, edit_penalty=edit_penalty)

        # NON-WORD
        if not in_vocab and suggestions:
            results.append({
                "index": i,
                "word": w_norm,
                "type": "NON-WORD",
                "suggestions": suggestions
            continue

        # REAL-WORD
        if in_vocab and suggestions:
            left = toks[i-1].lower() if i-1 >= 0 else None
            right = toks[i+1].lower() if i+1 < len(toks) else None
            orig_score = float(logscore_context(left, w_norm, right, k=k, lam=lam))

            # dictionary style access
            best_cand = suggestions[0]["cand"]
            best_score = suggestions[0]["score"]
            best_ed = suggestions[0]["edit"]

            MAX_REALWORD_EDIT = 1
            if (
                best_ed <= MAX_REALWORD_EDIT
                and best_cand != w_norm
                and (best_score - orig_score) >= ctx_margin
            ):
                results.append({
                    "index": i,
                    "word": w_norm,
                    "type": "REAL-WORD",
                    "orig_score": round(orig_score, 3),
                    "best_delta": round(best_score - orig_score, 3),
                    "suggestions": suggestions
                })

    return toks, results
```

In contrast to the detect_and_suggest function of the medical corpus only, the hybrid one is broader and smarter. While the first detect_and_suggest relied on medical unigram and bigram models only, the hybrid one uses richer scoring by using interpolated scores, edit penalties and a confidence margin to decide when to actually flag a word.

B) Hybrid Apply Corrections

```python
# For hybrid 'Apply Correction'
def apply_corrections_hybrid(text, results, ctx_only=True):
    """
    ctx_only=True: for REAL-WORD? apply only when flagged (i.e., delta >= margin)
    """
    toks = tokenize_user(text)
    idx_to_best = {}
    for r in results:
        if not r.get("suggestions"):
            continue
        best = r["suggestions"][0]["cand"]
        if r["type"] == "NON-WORD":
            idx_to_best[r["index"]] = best
        elif r["type"] == "REAL-WORD?":
            # already passed the margin filter in detect_and_suggest_hybrid
            idx_to_best[r["index"]] = best

    corrected = [ (idx_to_best[i] if i in idx_to_best else t) for i, t in enumerate(toks) ]
    return " ".join(corrected)
```

Similar to step m, the apply_corrections function was applied to the hybrid corpus. Apply_corrections_hybrid rewrites a text by applying the best corrections suggested by the hybrid spellchecker. Non-words are always corrected, and real-word errors are corrected only if they already passed the confidence margin during detection, ensuring conservative but reliable corrections.

C) Testing

```
tests = [
    "The patints were given aspitin to reduce pain.",
    "High blodsugar levels are dangerous for diabtes patients.",
    "The doctor monitored the insuline dosage carefully.",
    "He sufferd an atack yesterday."
]


for t in tests:
    toks, res = detect_and_suggest_hybrid(t, topk=5, ctx_margin=0.8)
    print("\nINPUT:", t)
    print("TOKENS:", toks)

    for r in res:  # r is a dictionary now
        print(f" -> Word: {r['word']} ({r['type']})")
        for s in r["suggestions"]:
            print(f"    cand={s['cand']}, score={s['score']}, edit={s['edit']}")
```

```
INPUT: The patints were given aspitin to reduce pain.
TOKENS: ['the', 'patints', 'were', 'given', 'aspitin', 'to', 'reduce', 'pain']
 -> Word: patints (NON-WORD)
    cand=patients, score=-7.459, edit=1
    cand=patient, score=-12.865, edit=2
    cand=parents, score=-15.299, edit=2
    cand=points, score=-15.31, edit=2
    cand=ratings, score=-16.886, edit=2
 -> Word: aspitin (NON-WORD)
    cand=aspirin, score=-17.063, edit=1
    cand=ascitic, score=-22.759, edit=2
```

A series of test sentences with intentional spelling inaccuracies was developed to test the hybrid spell-checker. The system tokenizes the input and identifies misspelled words and produces the ranked correction suggestions with relevant scores and edit distances per sentence.

D) The GUI function with all component and design requirements (+ starting the GUI)

[32]:
```
%gui tk
```

[ ]:
```
import tkinter as tk
from tkinter import ttk, messagebox

MAX_CHARS = 500

# Regex for tokenization
try:
    TOK_RE
except NameError:
    TOK_RE = re.compile(r"[A-Za-z@]+")

# --- tokenize with spans (for highlighting & click mapping)
def tokenize_with_spans(text):
    spans = []
    for m in TOK_RE.finditer(text.lower()):
        spans.append((m.group(0), m.start(), m.end()))
    return spans

class SpellApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("MedSpell Studio — Hybrid Corpus")
        self.geometry("980x620")
        self.configure(bg="#3E2723")  # dark brown background
        self._build_style()
        self._build_layout()
        self._bind_keys()
```

```
        # state
        self.spans = []
        self.tokens = []
        self.detect_results = []
        self.issue_indices = []
        self.issue_cursor = 0

        # demo text (changed here)
        demo = "The doctor prescribed insuline for the patient."
        self.txt.insert("1.0", demo)
        self._enforce_limit()
        self.run_check()

    # ---------- UI ----------
    def _build_style(self):
        style = ttk.Style(self)
        style.theme_use("clam")
        style.configure(".", background="#3E2723", foreground="#FFF3E0")
        style.configure("TButton", padding=6, background="#5D4037", foreground="#FFF3E0")
        style.map("TButton", background=[("active", "#6D4C41")])
        style.configure("TLabel", background="#3E2723", foreground="#FFF3E0")
        style.configure("TFrame", background="#3E2723")
        style.configure("Treeview", background="#4E342E", fieldbackground="#4E342E",
                        foreground="#FFF3E0", rowheight=24)
        style.configure("Treeview.Heading", background="#5D4037", foreground="#FFF3E0")

    def _build_layout(self):
        # Top toolbar
        bar = ttk.Frame(self); bar.pack(fill="x", padx=10, pady=10)
        ttk.Button(bar, text="Check (Ctrl/Cmd+Enter)", command=self.run_check).pack(side="left", padx=4)
        ttk.Button(bar, text="Auto-correct Top", command=self.auto_correct).pack(side="left", padx=4)
        ttk.Button(bar, text="Prev Issue", command=lambda: self.jump_issue(-1)).pack(side="left", padx=4)
        ttk.Button(bar, text="Next Issue", command=lambda: self.jump_issue(+1)).pack(side="left", padx=4)
        ttk.Button(bar, text="Corpus Word List", command=self.open_vocab_browser).pack(side="left", padx=4)
        self.lbl_status = ttk.Label(bar, text="Ready."); self.lbl_status.pack(side="right")
```

```
# Split view
split = ttk.Panedwindow(self, orient="horizontal"); split.pack(fill="both", expand=True, padx=10, pady=(0,10))

# Left: editor
left = ttk.Frame(split); split.add(left, weight=3)
self.txt = tk.Text(left, wrap="word", height=20, undo=True,
                   bg="#4E342E", fg="#FFF3E0", insertbackground="#FFF3E0",
                   selectbackground="#6D4C41", font=("Menlo", 12))
self.txt.pack(fill="both", expand=True, side="left")
sb = ttk.Scrollbar(left, command=self.txt.yview); sb.pack(side="right", fill="y")
self.txt.configure(yscrollcommand=sb.set)

# Char counter
meta = ttk.Frame(self); meta.pack(fill="x", padx=10, pady=(0,10))
self.lbl_count = ttk.Label(meta, text=f"0/{MAX_CHARS} chars")
self.lbl_count.pack(side="right")

# Right: Inspector
right = ttk.Frame(split); split.add(right, weight=2)
ttk.Label(right, text="Suggestion Inspector", font=("TkDefaultFont", 11, "bold")).pack(anchor="w", padx=4, pady=(0,6))
cols = ("Index", "Word", "Type", "Best", "ΔScore", "Edit")
self.tree = ttk.Treeview(right, columns=cols, show="headings", height=16)
for c, w in zip(cols, (60, 120, 100, 140, 80, 60)):
    self.tree.heading(c, text=c)
    self.tree.column(c, width=w, anchor="center")
self.tree.pack(fill="both", expand=True)
self.tree.bind("<<TreeviewSelect>>", self._on_tree_select)

# Suggestions detail + actions
act = ttk.Frame(right); act.pack(fill="x", pady=6)
self.btn_apply_best = ttk.Button(act, text="Replace Best", command=self.replace_best, state="disabled")
self.btn_apply_best.pack(side="left", padx=4)
self.btn_apply_sel  = ttk.Button(act, text="Replace Selected…", command=self.replace_selected, state="disabled")
self.btn_apply_sel.pack(side="left", padx=4)
```

```
      ttk.Label(right, text="Candidates for selected word:").pack(anchor="w", padx=4, pady=(6,2))
      s_cols = ("Candidate", "Score", "Edit")
      self.tree_sug = ttk.Treeview(right, columns=s_cols, show="headings", height=6)
      for c, w in zip(s_cols, (160, 100, 60)):
          self.tree_sug.heading(c, text=c)
          self.tree_sug.column(c, width=w, anchor="center")
      self.tree_sug.pack(fill="x", padx=0, pady=(0,6))

      # highlight tag styles
      self.txt.tag_config("miss_nonword", underline=True, foreground="#FF7043")
      self.txt.tag_config("miss_real", underline=True, foreground="#FFB74D")
      self.txt.tag_config("focus_word", background="#6D4C41")

      # bind events
      self.txt.bind("<KeyRelease>", lambda e: (self._enforce_limit(), self._clear_highlights()))
      self.txt.bind("<Button-1>", self._click_in_text)

  def _bind_keys(self):
      self.bind_all("<Control-Return>", lambda e: self.run_check())
      self.bind_all("<Command-Return>", lambda e: self.run_check())

  # ---------- logic ----------
  def _enforce_limit(self):
      content = self.txt.get("1.0", "end-1c")
      if len(content) > MAX_CHARS:
          self.txt.delete("1.0", "end")
          self.txt.insert("1.0", content[:MAX_CHARS])
          content = self.txt.get("1.0", "end-1c")
      self.lbl_count.config(text=f"{len(content)}/{MAX_CHARS} chars")

  def _clear_highlights(self):
      for tag in self.txt.tag_names():
          if tag.startswith("miss_") or tag == "focus_word":
              self.txt.tag_remove(tag, "1.0", "end")
```

```
def _highlight_results(self):
    self._clear_highlights()
    flagged = {r["index"]: r for r in self.detect_results}
    self.issue_indices = list(flagged.keys())
    for i, (_, s, e) in enumerate(self.spans):
        if i in flagged:
            kind = flagged[i]["type"]
            tag = "miss_nonword" if kind == "NON-WORD" else "miss_real"
            self.txt.tag_add(tag, f"1.0+{s}c", f"1.0+{e}c")

def run_check(self):
    content = self.txt.get("1.0", "end-1c")
    self.spans = [(t, s, e) for t, s, e in tokenize_with_spans(content)]
    self.tokens = [t for t, _, _ in self.spans]
    try:
        # 🔧 fixed: call detect_and_suggest_hybrid
        toks, results = detect_and_suggest_hybrid(content, topk=5, ctx_margin=1.8)
    except Exception as e:
        messagebox.showerror("Error", f"detect_and_suggest failed:\n{e}")
        return
    self.detect_results = results or []
    self._highlight_results()
    self._refresh_inspector()
    n = len(self.detect_results)
    self.lbl_status.config(text=("No issues found ✅" if n == 0 else f"Found {n} potential issue(s)."))

def _refresh_inspector(self):
    self.tree.delete(*self.tree.get_children())
    for r in self.detect_results:
        idx = r["index"]
        word = r["word"]
        typ = r["type"]
        best = r["suggestions"][0]["cand"] if r.get("suggestions") else "-"
        delta = r.get("best_delta", "")
        editd = r["suggestions"][0]["edit"] if r.get("suggestions") else ""
```

```
            self.tree.insert("", "end", iid=str(idx), values=(idx, word, typ, best, delta, editd))
        self.tree_sug.delete(*self.tree_sug.get_children())
        self.btn_apply_best.config(state="disabled")
        self.btn_apply_sel.config(state="disabled")

    def _on_tree_select(self, event=None):
        sel = self.tree.selection()
        self.tree_sug.delete(*self.tree_sug.get_children())
        self._clear_focus_word()
        if not sel:
            self.btn_apply_best.config(state="disabled")
            self.btn_apply_sel.config(state="disabled")
            return
        idx = int(sel[0])
        self._focus_token(idx)
        rec = next((r for r in self.detect_results if r["index"] == idx), None)
        if not rec:
            self.btn_apply_best.config(state="disabled")
            self.btn_apply_sel.config(state="disabled")
            return
        for s in rec.get("suggestions", []):
            self.tree_sug.insert("", "end", values=(s["cand"], s["score"], s["edit"]))
        state = "normal" if rec.get("suggestions") else "disabled"
        self.btn_apply_best.config(state=state)
        self.btn_apply_sel.config(state=state)

    def _focus_token(self, idx):
        if 0 <= idx < len(self.spans):
            _, s, e = self.spans[idx]
            self.txt.tag_add("focus_word", f"1.0+{s}c", f"1.0+{e}c")
            self.txt.see(f"1.0+{s}c")

    def _clear_focus_word(self):
        self.txt.tag_remove("focus_word", "1.0", "end")
```

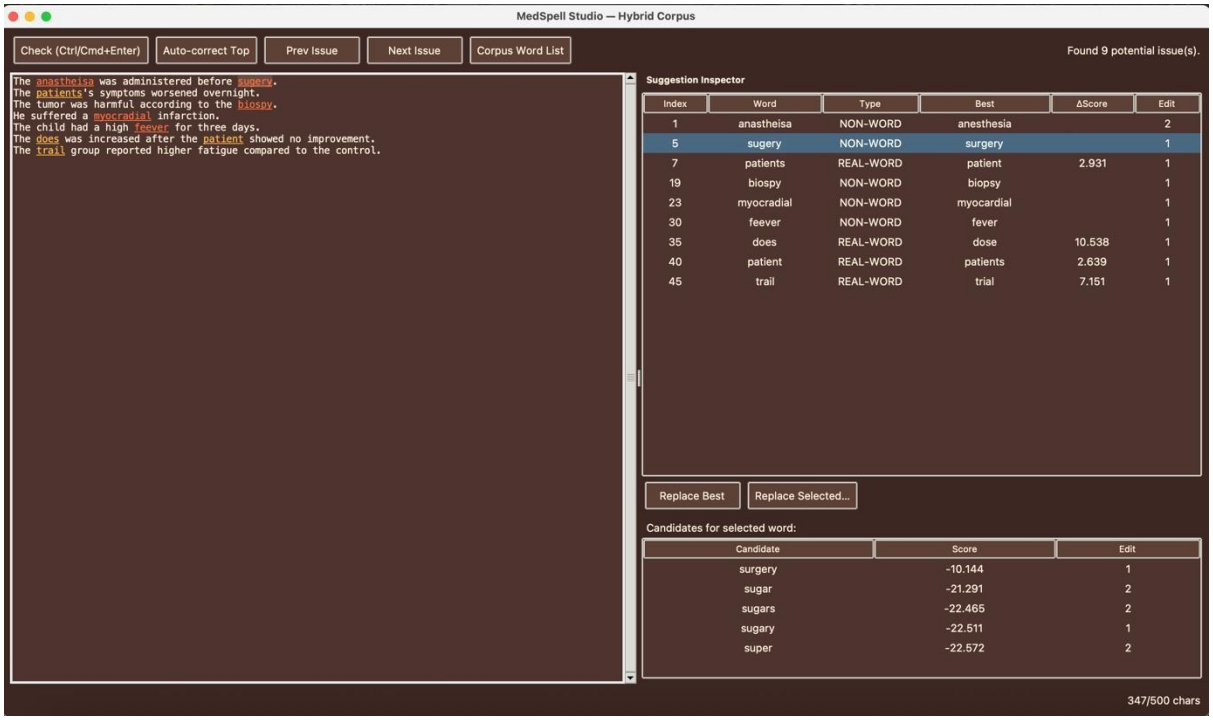
```
    def _click_in_text(self, event):
        index = self.txt.index(f"@{event.x},{event.y}")
        upto = self.txt.get("1.0", index)
        pos = len(upto)
        clicked = None
        for i, (_, s, e) in enumerate(self.spans):
            if s <= pos <= e:
                clicked = i; break
        if clicked is None:
            return
        if str(clicked) in self.tree.get_children():
            self.tree.selection_set(str(clicked))
            self.tree.focus(str(clicked))
            self._on_tree_select()

    def replace_best(self):
        sel = self.tree.selection()
        if not sel: return
        idx = int(sel[0])
        rec = next((r for r in self.detect_results if r["index"] == idx), None)
        if not rec or not rec.get("suggestions"): return
        best = rec["suggestions"][0]["cand"]
        self._replace_token(idx, best)

    def replace_selected(self):
        sel = self.tree.selection()
        sel2 = self.tree_sug.selection()
        if not sel or not sel2: return
        idx = int(sel[0])
        cand = self.tree_sug.item(sel2[0])["values"][0]
        self._replace_token(idx, cand)
```

```
def _replace_token(self, idx, cand):
    if not (0 <= idx < len(self.spans)): return
    _, s, e = self.spans[idx]
    self.txt.delete(f"1.0+{s}c", f"1.0+{e}c")
    self.txt.insert(f"1.0+{s}c", cand)
    self.run_check()

def auto_correct(self):
    content = self.txt.get("1.0", "end-1c")
    try:
        # 🔧 fixed: call detect_and_suggest_hybrid
        toks, results = detect_and_suggest_hybrid(content, topk=5, ctx_margin=1.8)
        corrected = apply_corrections(content, results)
    except Exception as e:
        messagebox.showerror("Error", f"auto-correct failed:\n{e}")
        return
    self.txt.delete("1.0", "end")
    self.txt.insert("1.0", corrected)
    self.run_check()

def jump_issue(self, step):
    if not self.detect_results: return
    self.issue_cursor = (self.issue_cursor + step) % len(self.detect_results)
    idx = self.detect_results[self.issue_cursor]["index"]
    if str(idx) in self.tree.get_children():
        self.tree.selection_set(str(idx))
        self.tree.focus(str(idx))
        self._on_tree_select()

def open_vocab_browser(self):
    vb = tk.Toplevel(self)
    vb.title("Corpus Vocabulary — Hybrid")
    vb.geometry("560x520")
    vb.configure(bg="#3E2723")
    ttk.Label(vb, text="Search vocabulary (Hybrid Corpus):", font=("TkDefaultFont", 10, "bold")).pack(anchor="w", padx=10, pady=(10,4))
```

```
        frm = ttk.Frame(vb); frm.pack(fill="x", padx=10, pady=6)
        qvar = tk.StringVar()
        entry = ttk.Entry(frm, textvariable=qvar, width=30); entry.pack(side="left", padx=(0,6))
        entry.configure(foreground="black", background="white")
        def refresh():
            lb.delete(*lb.get_children())
            q = (qvar.get() or "").lower().strip()
            try:
                items = search_vocab_hybrid(q, top=600) if q else sorted_vocab_hybrid[:600]
            except Exception:
                items = []
            for w, c in items:
                lb.insert("", "end", values=(w, c))

        ttk.Button(frm, text="Search", command=refresh).pack(side="left")
        ttk.Button(frm, text="Top 600", command=lambda:(qvar.set(""), refresh())).pack(side="left", padx=6)

        cols = ("Word", "Count")
        lb = ttk.Treeview(vb, columns=cols, show="headings", height=18)
        for c, w in zip(cols, (300, 120)):
            lb.heading(c, text=c)
            lb.column(c, width=w, anchor="w")
        lb.pack(fill="both", expand=True, padx=10, pady=(0,10))

        refresh()
        entry.bind("<Return>", lambda e: refresh())

# --- Launch app ---
if __name__ == "__main__":
    app = SpellApp()
    app.mainloop()
```

This code defines a Tkinter GUI program named MedSpell Studio - Hybrid Corpus, and which is an interactive spell-checking program. It enables typing or pasting of text, points out possible spelling mistakes (non-words and real-word errors), and also presents correction proposals in a panel called an inspector. Users have the option to substitute the proposed corrections with words manually, auto-correct everything that is wrong, peruse with the hybrid vocabulary, and go through flagged mistakes. The interface has a dark theme, has character limits, and incorporates the detect-andsuggest-hybrid logic of context-dependent corrections.

1.5 Results



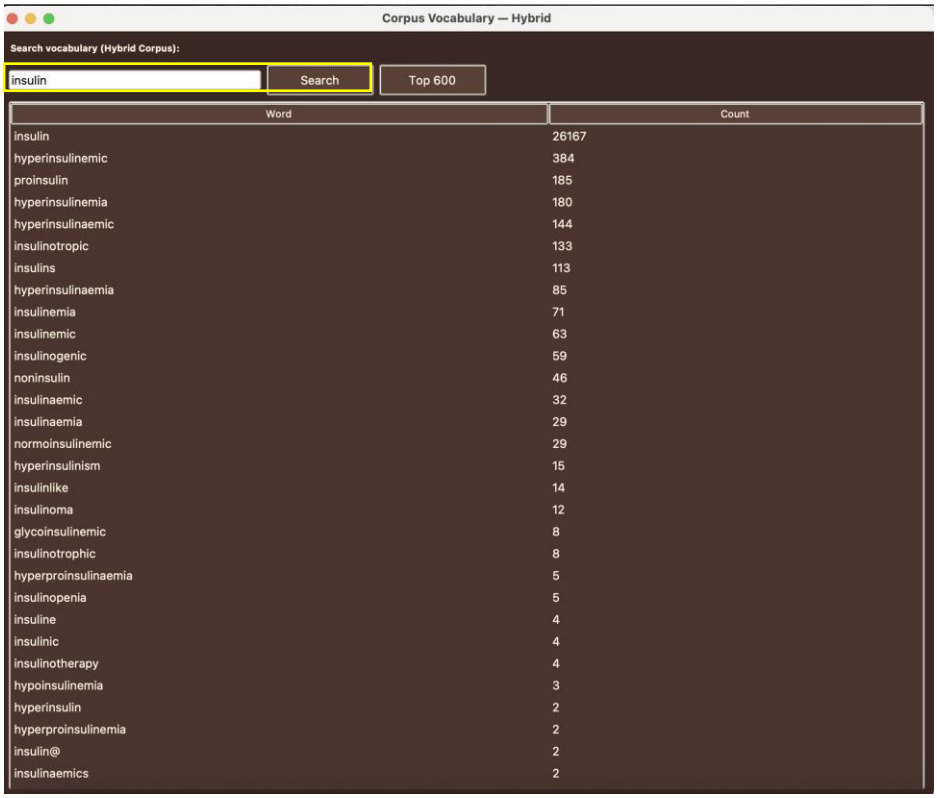
As one can see on the screenshot above, a functioning spelling correction system could be built. Non-word errors are underlined in red, while real-word errors are highlighted in orange. On the right side, one can see the ‘Suggestion Inspector’. The column ‘index’ indicates the tokens position in the token list. The column ‘Type’ assigns the corresponding error type (either ‘Non-Word’ or ‘Real-Word’). ‘Best’ shows the top score candidate word. In the case of the misspelt word ‘anastheisa’, the best candidate word would be ‘anesthesia’. ‘Delta score’ is the difference between the best candidate’s score and the original word’s score in context. When this improvement is larger than your context margin, then a correction is proposed in the system. The last column reflects how many edits were necessary for the candidate word. Taking the example of ‘sugery’, only one edit (inserting a ‘r’) was necessary, while ‘anesthesia’ needed two.

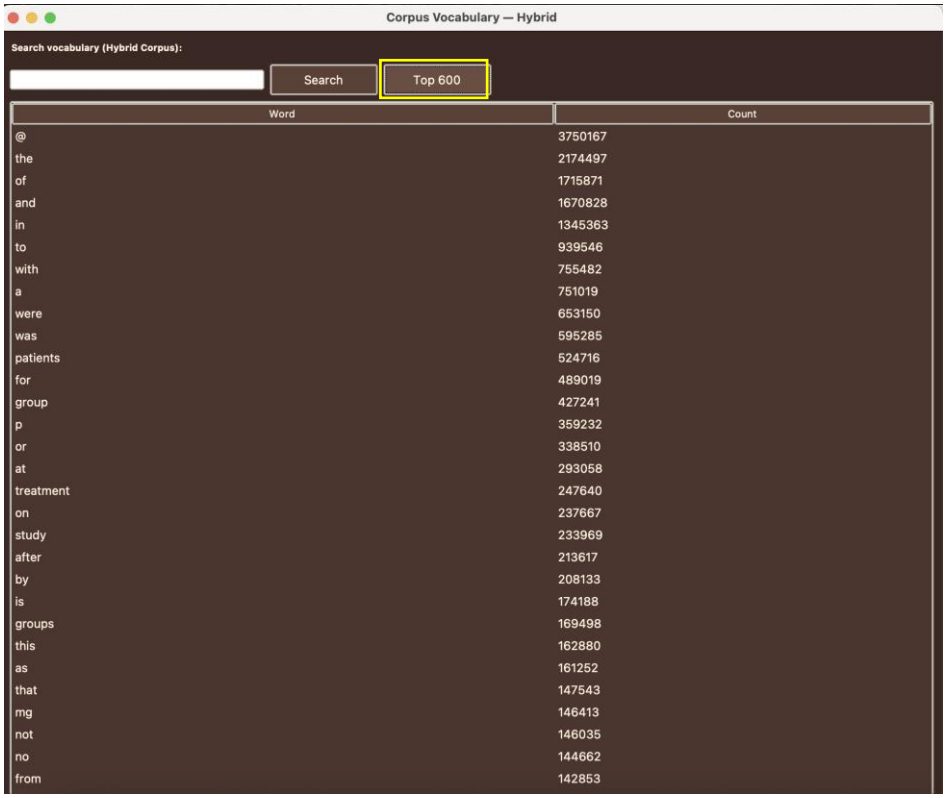
The box in the bottom right contains the candidate list with the respective candidates. The closer the score (log probability) is to 0, the better.

The button ‘Replace Best’ replaces the misspelt word with the top-ranked candidate (from the sorted list), whereas ‘Replace Selected’ lets you select an optional candidate from the candidate list below (‘Candidates for selected word’).

The button ‘Auto-correct Top’ lets the system go through the entire text and replace every flagged token with its top-ranked candidate correction.

“Prev Issue” and “Next Issue” are navigation shortcuts to cycle through all detected errors in your text, highlighting one at a time and showing its suggestions in the inspector.





As one can see on the two screenshots above, the application also includes a Corpus Vocabulary, which makes it possible to check whether certain words are contained by the vocabulary and also the 600 most frequent tokens in the vocabulary (“Top 600”).

1.5.1 Strengths

The strengths of this assignment’s correction spelling editor lies in the use of a hybrid corpus, consisting of both medical texts as well as general English. At the beginning, the model flagged words of sentences like “I went for a walk” as real-word errors, since sentences like this are not common in medical papers. For this reason, the corpus was combined by a general English corpus. After creating this combination of corpora, normal simple sentences were not flagged anymore.

One of the key strengths of this applied assignment, is the application of POS-tagging with NLTK’s pre-trained POS tagger. POS-tagging makes sure that candidate words ensure grammatical consistency. As described earlier, this reduces implausible corrections, since the grammatical role (e.g. noun, verb etc.) matches the grammatical function of the misspelt word.

Furthermore, Inverse Document Frequency (IDF) was implemented as an additional weighting factor. IDF evaluates how informative a word is by penalizing very common words (e.g. “the”, “and”) and assigns higher weight to less frequent and domain-specific words (e.g. “insulin”). The system is then able to choose the candidate which are more contextually probable. The system makes use of IDF and language model probabilities with edit-distance penalties, eliminating the tendency of over-correcting common, but uninformative words, and enhancing the probability of choosing medically informative words in a clinical setting.

Semantics are additionally improved by pre-trained GloVe embeddings. GloVe pre-trained embeddings encode words in a vector space in which the system can determine the cosine similarity between candidate corrections and the context vector based on the surrounding words.

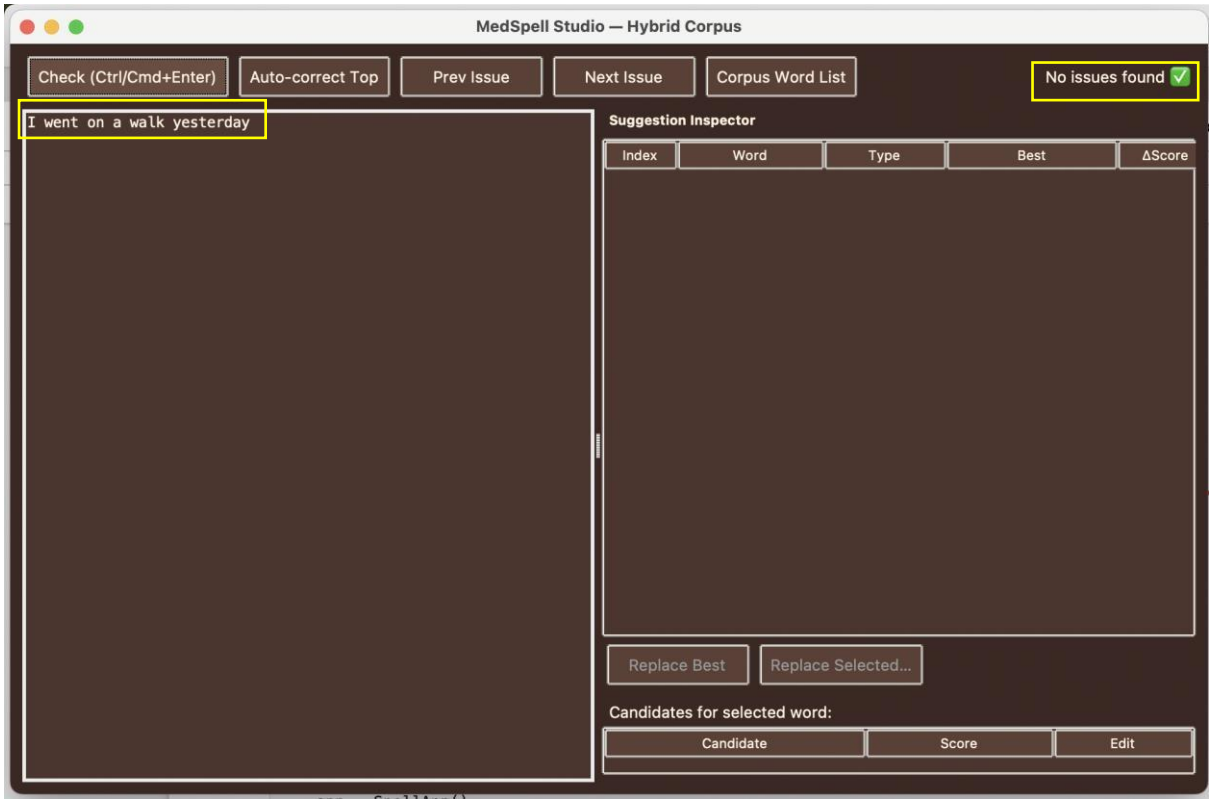


Figure 1: Simple sentences are not flagged anymore after creation of hybrid corpus

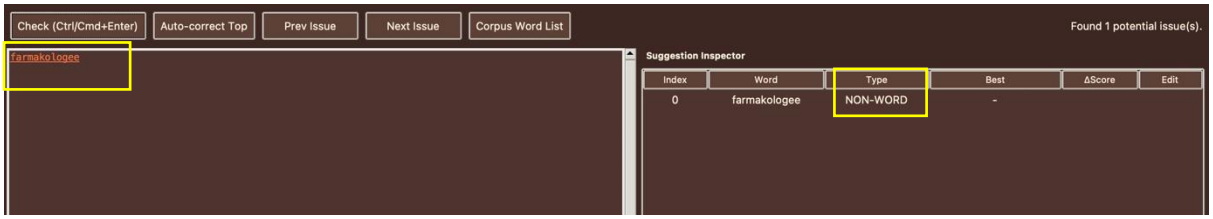
1.5.2 Limitations

In the following the limitations of the practical part of this assignment will be highlighted.

Since we use the PubMed corpus as our main domain corpus, the editor performs the best when the user input is related to the medical domain. Even though a general English corpus was added, the majority of words comes from the PubMed corpus. Thus, the usage of the editor is limited to a specific domain.

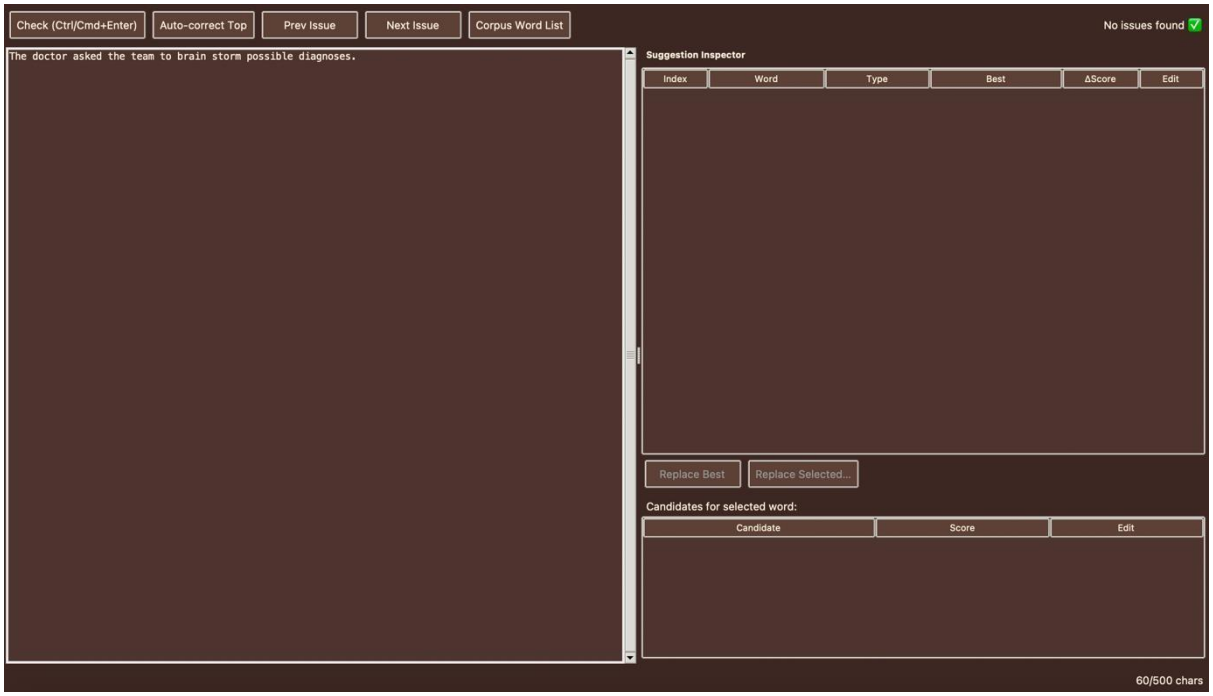
Since we used the PubMed corpus, we have to assume corpus contamination to a certain extent. Since the corpus is a collection of abstracts which might already contain noisy text and spelling mistakes. Since the assignment's system was trained on this data, one cannot rule out, that the model learnt wrong spelling of words or words in a wrong context. Thus, wrongly spelt words might be treated as valid words by the editor. Furthermore, the corpus contains many specialised terms and abbreviations which were not cleaned. Besides this, the corpus also contains abstracts in other languages (such as French or German), which might affect the model negatively, as it is supposed to be used for English sentences only. Spelling rules from other languages might dominate, leading to wrong writing being classified as true.

In addition the corpus contamination, the edit distance of 2 must be mentioned. Even though this edit distance keeps the runtime low and avoids suggesting completely unrelated words, it does not cover spelling mistakes that require more than three edits (Example: "farmakologee" → pharmacology). This assignment's editor would flag this word as a non-word.



Another limitation of this system is that low-frequency words in the hybrid corpus are often not detected as errors, even when they are contextually wrong. When typing in the sentence "The doctor asked the team to *brain storm* possible diagnoses, the system did not identify

‘brain storm’ as an error, even though “brainstorm” would be correct. The underlying problem is that the word ‘brainstorm’ only exists once in the huge hybrid corpus. Thus, one faces the challenge of data sparsity in the corpus. Words that appear only once or very rarely are still treated as valid vocabulary entries, even though there is too little context material to estimate their probabilities reliably.



Word	Count
brainstorming	4
brainstorm	1
brainstormed	1

The use of unigram and bigram models is another constraint. This can be improved with the use of higher order n-gram models that would better represent the context. Nonetheless, this extension would make the storage, and computation needs to become higher as well.