MiniAI

Programmer's manual · Manuel du programmeur · Manuale del programmatore · Manualul programatorului · プログラマー マニュアル

Language · Langue · Lingua · Limba · 言語.....**Page · Page · Pagina · Pagina · ページ**

EN English.....2
FR Français.....24
IT Italiano.....46
RO Română.....68

JA 日本語.....90

DevService.net — Raffaele Romano

MiniAI

Educational artificial intelligence: from text to vectors

Programmer's manual

Source code in VB.NET, C# and JavaScript

October 2026

Contents

1. Introduction....4
1.1 What MiniAI is....4
1.2 The same principle as the large AIs....4
1.3 Download contents....4
2. Requirements, compiling and startup....6
2.1 VB.NET version....6
2.2 C# version....6
2.3 JavaScript version....6
2.4 First run with Pinocchio....6
3. Basic concepts....8
3.1 From text to numbers....8
3.2 Cosine similarity....8
3.3 The distributional hypothesis....8
3.4 Retrieval and generation....8
4. The training pipeline....9
4.1 Step 1 – Sentence splitting....9
4.2 Step 2 – Normalization, stems and vocabulary....9
4.3 Step 3 – IDF....9
4.4 Step 4 – Sparse TF-IDF matrix....10
4.5 Step 5 – Co-occurrences....10
4.6 Step 6 – PPMI....10
4.7 Step 7 – Embedding with random projection....11
4.8 Step 8 – Dense sentence vectors....11
4.9 The training log....11
5. Answering a question....12
5.1 A – Question type....12
5.2 B – Question words....12
5.3 C – Expansion with similar words....12
5.4 D – Sentence scoring....12
5.5 E – Context....13
5.6 F – Answer....13
5.7 Reading the trace....13
6. Code structure....15
6.1 Files and correspondences....15
6.2 ElaboratoreTesto....15
6.3 ModelloConoscenza....15
6.4 MotoreDomande....16
6.5 The user interface....16
6.6 Notes on porting between the three languages....17
7. The .mai file format....18
8. Parameters and experiments....19
8.1 Training parameters....19
8.2 Answer parameters....19
8.3 Guided experiments with Pinocchio....19
9. Limits and extensions....21
9.1 Intentional limits....21
9.2 From the toy to real AIs....21
9.3 Possible extensions....21
10. License and liability....23

1. Introduction

1.1 What MiniAI is

MiniAI is my own basic study of artificial intelligence applied to language. It reads a long text (a book, a collection of documents, a manual), turns it into numbers, vectors and matrices, and then answers questions by searching the text for the sentence closest to the question.

It is not a product for production use: it is a laboratory. Every step of the computation is visible, the program writes the sizes of the matrices it builds to the log and, for every answer, it shows the «reasoning trace»: which words it recognized, which ones it added, which sentences it compared and with what scores. The goal is to understand, not to amaze.

The program is distributed as **source code**, in three equivalent versions: **VB.NET**, **C#** and **JavaScript**. It uses no external libraries: everything it needs (from sentence splitting to the random projection of the embeddings) is written out in full in a few hundred commented lines.

1.2 The same principle as the large AIs

Obviously, the amount of data makes the difference. The AIs on the market are trained on billions of words, have billions of parameters and neural networks that learn to generate new text. MiniAI reads a single book and learns a few thousand vectors in less than a second. The basic principle, however, is the same:

- **text becomes numbers**: every word and every sentence is represented by a vector;
- **meaning comes from context**: two words used in similar contexts receive similar vectors (distributional hypothesis);
- **understanding a question means measuring similarities**: the question is transformed into the same vector space and compared with what the system knows;
- **search before answering**: the retrieval of relevant sentences (*retrieval*) is the same mechanism used by RAG systems (*Retrieval-Augmented Generation*), which supply a language model with the documents from which to draw the answer.

MiniAI stops at retrieval: **it does not generate text, it picks an existing sentence**. That is precisely why it is a good starting point: you can see clearly what statistics can do on its own and where more data or more powerful models are needed.

> With the same code, more text gives better results. With a single novel, the «similar words» are often just words that are close together in the story; with dozens of books on the same subject, real synonyms start to appear. This is, on a small scale, the reason why large AIs need enormous amounts of data.

1.3 Download contents

The website offers three zip files, one per language. Each contains the complete program, the sample text and a `LEGGIMI.md` file in Italian.

File	Contents
`MiniAI_VBNET.zip`	Windows Forms project for .NET Framework 4.8 in VB.NET: `MiniAI.sln`, `MiniAI.vbproj`, `compila.bat` and the files `Testo.vb`, `Modello.vb`, `Motore.vb`, `FormPrincipale.vb`, `Program.vb`.
`MiniAI_CSharp.zip`	The same project in C#: `MiniAI.sln`, `MiniAI.csproj`, `compila.bat` and the `.cs` files with the same names.
`MiniAI_JavaScript.zip`	Standalone web page (`index.html`, `stile.css`, `js` folder) and `prova-node.js` for command-line use with Node.js.

File	Contents
`Pinocchio.txt`	In all three zips: *Le avventure di Pinocchio* (The Adventures of Pinocchio) by Carlo Collodi (1883), a public-domain text from Project Gutenberg, used for all the examples in this manual.

The three versions perform the same calculations in the same order. With the same text and the same parameters they produce the same answers, the same trace and the same `.mai` model file, identical byte for byte: a model saved by one version can be opened with the others.

2. Requirements, compiling and startup

2.1 VB.NET version

- Windows 10/11 or Windows Server 2016 and later, with .NET Framework 4.8 (already included in Windows 10 starting with version 1903, in Windows 11 and in Windows Server 2022 and later).
- **With Visual Studio** (2019 or 2022, «.NET desktop development» workload): open `MiniAI.sln` and press F5.
- **Without Visual Studio**: double-click `compila.bat`. The file uses the `vbc.exe` compiler that .NET Framework 4.8 installs in `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319`. The code is written to be compatible with that compiler (Visual Basic 2012), and `MiniAI.exe` is created in the same folder.

The `.vb` files are saved in UTF-8 with BOM: this way the Framework compiler, too, correctly reads the accented letters in comments and messages.

2.2 C# version

- Same requirements as the VB.NET version.
- **With Visual Studio**: open `MiniAI.sln` (project `MiniAI.csproj`) and press F5.
- **Without Visual Studio**: double-click `compila.bat`, which uses `csc.exe` from .NET Framework 4.8. The code is compatible with C# 5, the version of that compiler: no string interpolation or other more recent syntax.

2.3 JavaScript version

- **In the browser**: double-click `index.html`. It works with recent versions of Edge, Chrome and Firefox, even without an Internet connection and without a server: the text never leaves the computer.
- **With Node.js** (version 18 or later), from the program folder: `node prova-node.js Pinocchio.txt` trains the model and opens a question session; `node prova-node.js Pinocchio.txt domande.txt modello.mai` answers every line of `domande.txt` and saves the model; `node prova-node.js modello.mai` restarts from a saved model. The `--traccia` option also prints the reasoning trace.

In the browser, training blocks the page for the duration of the computation (less than a second with Pinocchio); with very long texts the page may seem frozen for a few seconds.

2.4 First run with Pinocchio

1. Tab **1 · Addestramento** (Training): «Apri testo (.txt)...» (Open text) and choose `Pinocchio.txt` (in the JavaScript version you can also drag the file onto the page).
2. Leave the default parameters (window 4, size 128, minimum frequency 2) and press «Addestra» (Train). The log lists the eight steps described in chapter 4.
3. Tab **2 · Domande** (Questions): type *Chi ha costruito il burattino?* (Who built the puppet?) and press Enter. The answer with the nearby sentences appears at the top, the complete trace below.
4. Tab **3 · Esplora la memoria** (Explore the memory): type `naso` (nose) and press «Parole simili» (Similar words).
5. Optional: «Salva modello...» (Save model) creates a `.mai` file that reloads in an instant with «Carica modello...» (Load model).

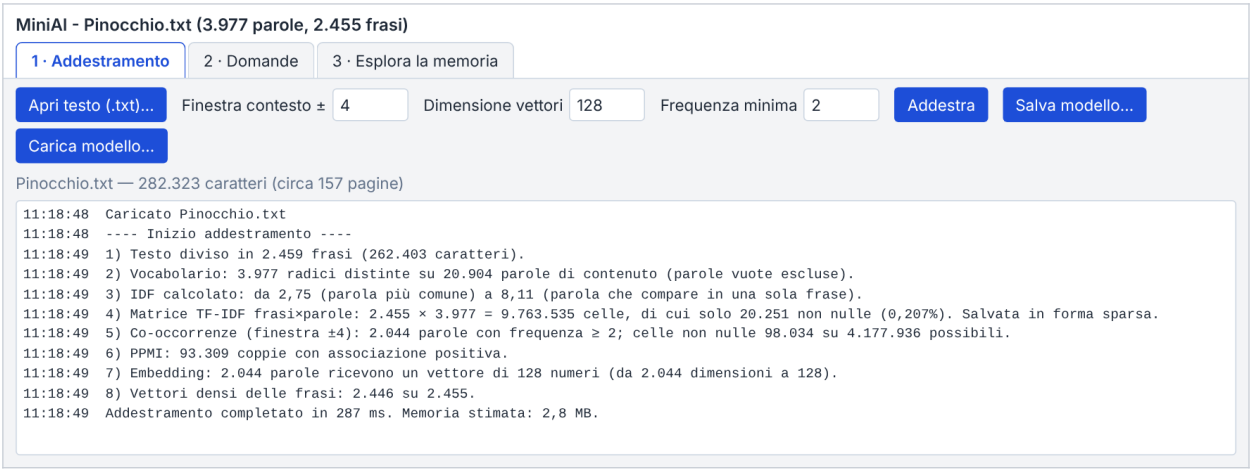

Figure 1 – Addestramento tab after training on Pinocchio (JavaScript version; the Windows versions have the same three tabs)

The text is read as UTF-8; if it contains bytes that are not valid UTF-8, it is read again with the Windows encoding (Windows-1252), so old `.txt` files saved with Notepad also work.

3. Basic concepts

3.1 From text to numbers

A computer does not know what a «burattino» (puppet) is. It can, however, count and perform operations on lists of numbers. All of MiniAI's work consists of turning words and sentences into **vectors**, that is, lists of numbers of fixed length, so that texts with similar meaning give similar vectors.

Two kinds of vectors appear in the program:

- **sparse vectors**: as long as the vocabulary (about 4,000 positions with Pinocchio), almost all zeros; each position corresponds to a word. Only the nonzero positions are stored;
- **dense vectors** (*embedding*): short (128 numbers), all meaningful; no position corresponds to a specific word, it is the set of numbers as a whole that describes the meaning.

3.2 Cosine similarity

To express how similar two vectors are, MiniAI uses the **cosine of the angle** between them:

```
cosine(a, b) = (a · b) / (|a| · |b|)

a · b = a1·b1 + a2·b2 + ... + an·bn      (dot product)
|a|   = square root of (a · a)            (length of the vector)
```

The result is 1 if the vectors point in the same direction, 0 if they have nothing in common, and negative if they are opposite. Direction matters, not length: a long sentence and a short one on the same subject turn out to be similar.

To save computation, all vectors are **normalized** (scaled to length 1) as soon as they are created. From then on the cosine coincides with the simple dot product: this is what the `Normalizza` and `Prodotto` functions of the `ModelloConoscenza` class do.

3.3 The distributional hypothesis

«You shall know a word by the company it keeps» (J. R. Firth, 1957). If two words often appear near the same other words, they probably have a related meaning. MiniAI applies this idea by counting, for each word, which words are near it in the text (co-occurrence matrix) and turning these counts into a dense vector. word2vec, GloVe and the embedding layers of modern language models are based on the same principle.

3.4 Retrieval and generation

There are two ways to answer a question starting from a text:

- **extractive** (retrieval): you search the text for the most relevant passage and return it as is. This is what MiniAI does, and it is the first stage of RAG systems;
- **generative**: a language model produces a new sentence, word by word, each time choosing the most probable word. It requires neural networks trained on enormous amounts of text.

Retrieval has an important advantage: the answer is always a real, verifiable sentence from the document, and the system can honestly say «I don't know any of these words».

4. The training pipeline

Training is performed by the `Addestra` method of the `ModelloConoscenza` class (file `Modello.vb`, `Modello.cs` or `js/modello.js`). It consists of eight steps; each one writes a line to the log. The numbers given in this chapter are those obtained with `Pinocchio.txt` and the default parameters.

4.1 Step 1 – Sentence splitting

If the file comes from Project Gutenberg, `TogliIntestazioneGutenberg` removes the license at the beginning and at the end (it looks for the lines `*** START OF` and `*** END OF`). Then `DividiInFrasi`:

- joins the lines broken by the «hard» line breaks typical of `.txt` files; paragraphs are separated by an empty line;
- splits each paragraph where there is an end-of-sentence mark (. ! ? …), optionally followed by quotation marks, then a space and an uppercase letter, a digit or a dialogue dash;
- discards sentences with fewer than 4 words.

The sentence is the **unit of answer**: MiniAI always answers with a whole sentence. With Pinocchio you get 2,459 sentences out of 262,403 characters.

4.2 Step 2 – Normalization, stems and vocabulary

Each sentence is reduced to its **content words** by the `ParoleContenuto` function:

1. `Normalizza`: everything lowercase, accents removed (Unicode NFD decomposition and removal of diacritical marks), every character that is not a letter or a digit becomes a space. «Perché l'albero?» (Why the tree?) becomes `perche l albero`.
2. The **stop words** (articles, prepositions, auxiliaries, pronouns: about 200 words listed in `Testo`) and single-letter words are discarded: they appear everywhere and say nothing about the content.
3. `Stem`: a deliberately simple stemmer removes a known ending (`-issimo`, `-mente`, `-zione`, `-avano`, `-ando`, `-ato`, `-ito`…), or else the final vowel. «burattino», «burattini» and «burattina» (puppet, puppets, female puppet) all become `burattin`; «andato» and «andavano» (gone, they went) become `and`.

Each new stem receives a sequential number (its **id**) in the `Vocabolario` dictionary. For each stem the occurrences are counted (`Frequenze`) and the most frequent form is remembered (`Forme`), used only to display the word in a readable way. With Pinocchio: **3,977 stems** out of 20,904 content words.

> The stemmer often makes mistakes, and it is instructive to see where: «costruito» (built) never appears in Pinocchio, but its stem `costru` coincides with that of «costruì» (he built), which appears once. The word in the question is therefore recognized thanks to another form of the same verb.

4.3 Step 3 – IDF

Not all words are equally informative: «pinocchio» appears in hundreds of sentences, «abbecedario» (spelling book) in very few. The **IDF** (*inverse document frequency*) measures how rare a word is:

```
idf(t) = ln( (N + 1) / (df(t) + 1) ) + 1

N     = number of sentences
df(t) = number of sentences that contain the word t
```

The +1 in the numerator and in the denominator avoids division by zero; the final +1 guarantees that even the most common word has a positive weight. With Pinocchio the IDF ranges from 2.75 (the most common word, «pinocchio») to 8.11 (words that appear in only one sentence). Here is the same line in the three versions:

```
' VB.NET
Idf(i) = CSng(Math.Log((N + 1.0) / (df(i) + 1.0)) + 1.0)

// C#
Idf[i] = (float)(Math.Log((N + 1.0) / (df[i] + 1.0)) + 1.0);

// JavaScript (Float32Array: the conversion to Single is automatic)
this.idf[i] = Math.log((N + 1.0) / (df[i] + 1.0)) + 1.0;
```

4.4 Step 4 – Sparse TF-IDF matrix

Each sentence becomes a vector as long as the vocabulary. The weight of word t in sentence f is:

```
weight(t, f) = (1 + ln tf(t, f)) · idf(t)      tf = occurrences of t in the sentence

then the vector is normalized to length 1
```

The logarithm dampens repetitions: a word repeated three times is not worth three times as much. The complete sentences × words matrix would have 2,455 × 3,977 = 9,763,535 cells, but only 20,251 are nonzero (0.207%). For this reason each sentence stores only two parallel arrays: `Termini` (the ids of the words present, in ascending order) and `Pesi` (their respective weights). This is the **sparse representation**, the same one used by search engines.

Sentences made up only of stop words do not produce a vector and are discarded: this is why 2,455 sentences are stored and not 2,459.

4.5 Step 5 – Co-occurrences

This is where the «semantic» part begins. Only words with a frequency at least equal to `FrequenzaMinima` (2) are considered: words seen only once do not have enough contexts for a reliable vector. With Pinocchio there are 2,044.

The content words of all sentences are lined up in a single stream. For each word a and for each word b that follows it within `Finestra` positions (4), a weight that decreases with distance is added to both cells (a, b) and (b, a):

```
cooc(a, b) += 1 / distance        distance = 1, 2, ..., Finestra
```

The theoretical 2,044 × 2,044 matrix has 4,177,936 cells, but only 98,034 are nonzero: this one, too, is stored in sparse form, in a dictionary whose key is `a · V + b` (a 64-bit integer).

4.6 Step 6 – PPMI

The raw count favors frequent words: «disse» (said) is near almost everything. The **PMI** (*pointwise mutual information*) compares how many times two words are near each other with how many times you would expect by pure chance:

```
PMI(a, b)  = ln( P(a, b) / ( P(a) · Pc(b) ) )

P(a, b) = cooc(a, b) / total
P(a)    = row sum of a / total
Pc(b)   = (row sum of b)^0.75 / sum of all (row sums)^0.75

PPMI(a, b) = max(0, PMI(a, b))
```

Only the positive values are kept (**PPMI**, *positive* PMI): the pairs that occur together more often than chance. The 0.75 exponent on the context probability is the same trick used by word2vec and reduces the excessive weight of rare contexts. With Pinocchio: 93,309 pairs with positive association.

4.7 Step 7 – Embedding with random projection

Each row of the PPMI matrix is already a vector that describes a word, but it is 2,044 positions long. To get short vectors MiniAI uses a **random projection**:

1. each context word b is assigned a fixed random vector r(b) of `Dimensione` numbers (128), drawn from a normal distribution with standard deviation 1/√D (Box-Muller transform);
2. the embedding of word a is the sum of the vectors of its contexts, weighted by PPMI: e(a) = Σ PPMI(a, b) · r(b);
3. the vector is normalized to length 1.

The **Johnson-Lindenstrauss lemma** guarantees that a random projection approximately preserves the distances and cosines between the original vectors: two words with similar PPMI rows also have similar embeddings. This technique is known as *random indexing*; modern models obtain the same result by learning the vectors with a neural network.

The random generator always starts from the same `Seme` (seed, 12345), so training is **reproducible**: the same text and the same parameters always give the same vectors. With Pinocchio 2,044 words receive an embedding; words that are too rare remain without one (value `Nothing`/`null`).

4.8 Step 8 – Dense sentence vectors

The dense vector of a sentence is the average of the embeddings of its words, weighted by the TF-IDF weight computed in step 4, then normalized. Rare and specific words count more than common ones. With Pinocchio 2,446 sentences out of 2,455 have a dense vector; the others contain only words without an embedding.

4.9 The training log

This is the complete log produced with Pinocchio and the default parameters; the three versions write the same lines:

```
1) Testo diviso in 2.459 frasi (262.403 caratteri).
2) Vocabolario: 3.977 radici distinte su 20.904 parole di contenuto
   (parole vuote escluse).
3) IDF calcolato: da 2,75 (parola più comune) a 8,11 (parola che compare
   in una sola frase).
4) Matrice TF-IDF frasi×parole: 2.455 × 3.977 = 9.763.535 celle, di cui
   solo 20.251 non nulle (0,207%). Salvata in forma sparsa.
5) Co-occorrenze (finestra ±4): 2.044 parole con frequenza ≥ 2; celle non
   nulle 98.034 su 4.177.936 possibili.
6) PPMI: 93.309 coppie con associazione positiva.
7) Embedding: 2.044 parole ricevono un vettore di 128 numeri (da 2.044
   dimensioni a 128).
8) Vettori densi delle frasi: 2.446 su 2.455.
```

Training takes from a few tenths of a second to one second, depending on the computer; the model takes up about 2.8 MB in memory and on disk.

5. Answering a question

The answer is produced by the `Rispondi` method of the `MotoreDomande` class (file `Motore.vb`, `Motore.cs` or `js/motore.js`). The method returns a `Risposta` object with three fields: the text of the answer, the trace and the score of the chosen sentence. The steps are labeled with the letters A to F, the same ones that appear in the trace.

5.1 A – Question type

If among the first three words of the question there is *chi*, *dove*, *quando*, *quanto* (or *quanti*, *quante*, *quanta*), *perché* or *come* (who, where, when, how much/many, why, how), the question type is recognized. It is the only part of the program written «by hand» with fixed rules, and it serves to compare programmed knowledge with knowledge learned from the data.

5.2 B – Question words

The question undergoes the same transformations as the text: normalization, removal of stop words, reduction to the stem. Each stem is looked up in the vocabulary. Words never seen are listed in the trace as «MAI viste nel testo» (NEVER seen in the text): the model knows nothing outside the text it has read. If no word is known, the answer is *Non conosco nessuna delle parole della domanda* (I don't know any of the words in the question).

The recognized words are assigned the TF-IDF weight, as for the sentences: (1 + ln tf) · idf.

5.3 C – Expansion with similar words

For each word in the question, the `Espansione` (3) words with the most similar embedding are looked up. Those with a similarity at least equal to `SogliaEspansione` (0.35) are added to the question with a reduced weight:

```
weight(added word) = idf · similarity · 0.5
```

This is how a question about «vendere» (to sell) can find a sentence that talks about «mercato» (market). In the trace the added words are marked with the ~ symbol. The question vector is then normalized.

The **dense vector of the question** is also computed: the sum of the embeddings of the original words (not of the added ones), weighted by IDF, normalized.

5.4 D – Sentence scoring

Each stored sentence receives three scores:

- **lexical**: cosine between the sparse vector of the question and that of the sentence. The products are summed only over the words in common, so the computation is very fast;
- **semantic**: cosine between the dense vector of the question and that of the sentence (negative values become 0). It rewards sentences that talk about the same subject even with different words;
- **bonus**: a small reward (0.02–0.03) if the sentence contains clues suited to the question type: a proper noun for *chi* (in practice, a word with an initial capital that is not the first word of the sentence), a time word for *quando*, a place for *dove*, a cause for *perché*, a number for *quanto*, a manner for *come*. The bonus applies only to sentences that have at least one word in common with the question.

```
raw = (1 - ps) · lexical + ps · semantic + bonus        ps = PesoSemantico (0.30)
```

5.5 E – Context

In a story, a relevant sentence is often surrounded by relevant sentences. The final score adds a share of the average of the scores of the two neighboring sentences:

```
total(i) = raw(i) + pc · (raw(i-1) + raw(i+1)) / 2     pc = PesoContesto (0.25)
```

Sentences are sorted by descending total score (stable sort: with equal scores, the sentence that comes first in the text wins) and the trace shows the top five.

5.6 F – Answer

The sentence with the highest score is the answer. If the score is lower than `SogliaIncertezza` (0.12) the answer is preceded by *(Risposta incerta: punteggio basso)* (Uncertain answer: low score); if it is zero the program answers *Non ho trovato nulla di pertinente* (I found nothing relevant). Below the answer, `FrasiContesto` sentences before and after (1) are shown, so you can read it in its context.

5.7 Reading the trace

This is the trace of the question *Chi ha costruito il burattino?* with the default parameters:

```
A) TIPO DI DOMANDA: CHI
B) PAROLE DELLA DOMANDA
   ignorate (parole vuote): chi, ha, il
   costrui            radice 'costru'  frequenza     1  IDF 8,11  (senza embedding)
   burattino          radice 'burattin'  frequenza   226  IDF 3,40
C) ESPANSIONE CON PAROLE SIMILI (embedding)
   costrui            -> nessuna sopra soglia
   burattino          -> guardare (0,35)
D) CLASSIFICA (2.455 frasi confrontate)   peso semantico 0,30, peso contesto 0,25
   pos  frase   lessic.  semant.  bonus   TOTALE   parole in comune
     1   1026   0,385    1,000    0,030   0,607    burattino
             — Io sono un burattino.
     2   2140   0,385    1,000    0,000   0,588    burattino
             Io sono un burattino.
     3   1245   0,174    1,000    0,000   0,500    burattino
             Nascono burattini, vivono burattini e muoiono burattini.
```

Several instructive things can be noticed. «costruito» is recognized only through the stem of «costruì», which appears once and therefore has no embedding. The semantic score is 1.000 for sentences that contain only the word «burattino»: their dense vector coincides with that of the question. The first and second sentences have the same lexical score; the first wins only because of the *chi* bonus: after the dialogue dash the word «Io» (I) has an initial capital and the rule mistakes it for a proper noun. A handwritten rule is simple to understand, but just as easy to fool. The answer is wrong from a human point of view (the right answer is «Geppetto»), but it is exactly what the statistics of this text suggest.

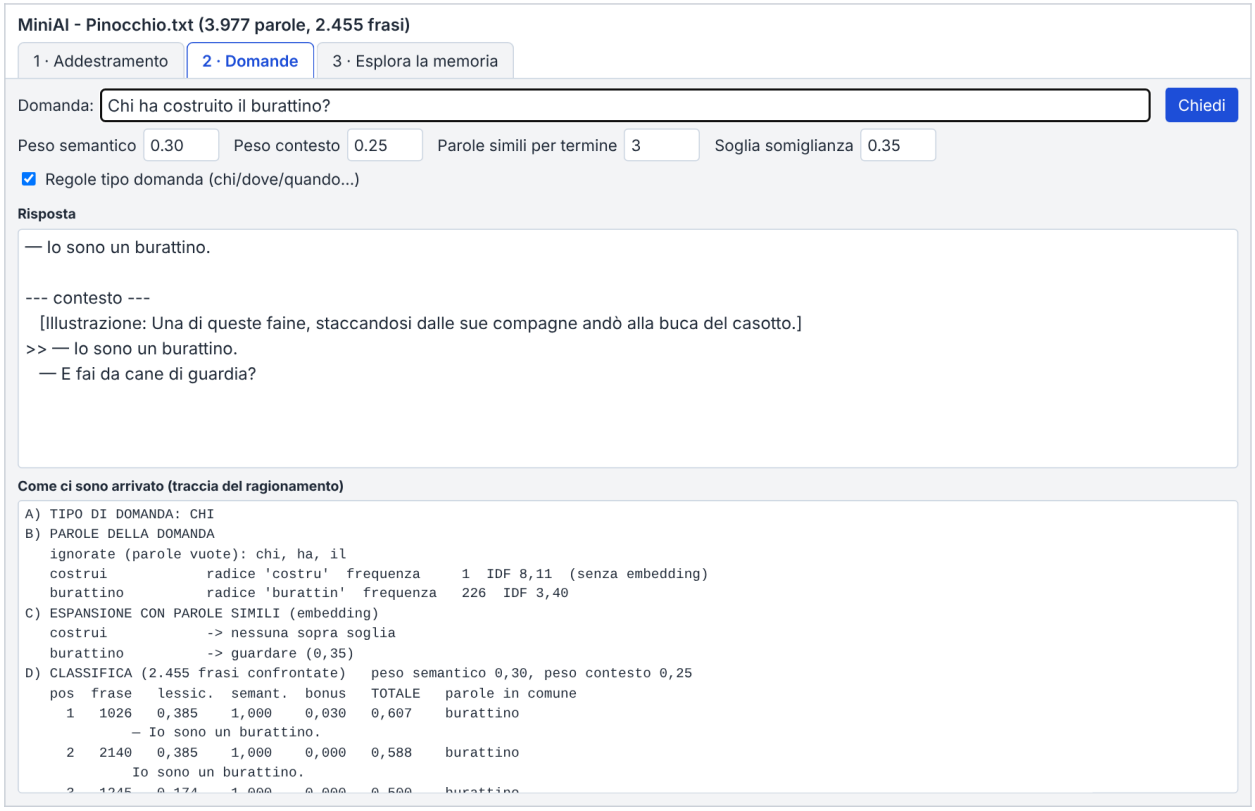

Figure 2 – Domande tab: the answer with its context and, below, the reasoning trace

6. Code structure

6.1 Files and correspondences

The names of classes, methods and variables are in Italian and are the same in the three versions. In JavaScript they follow the conventions of the language (lowercase initial: `rispondi` instead of `Rispondi`).

VB.NET	C#	JavaScript	Contents
`Testo.vb`	`Testo.cs`	`js/testo.js`	Class `ElaboratoreTesto`: sentences, normalization, stop words, stems.
`Modello.vb`	`Modello.cs`	`js/modello.js`	Classes `Frase` and `ModelloConoscenza`: training, similar words, saving and loading.
`Motore.vb`	`Motore.cs`	`js/motore.js`	Classes `Risposta` and `MotoreDomande`: answering questions and trace.
`FormPrincipale.vb`	`FormPrincipale.cs`	`index.html`, `js/app.js`, `stile.css`	Three-tab user interface. The .NET versions build the controls in code, without Designer files.
`Program.vb`	`Program.cs`	`prova-node.js`	Entry point. In JavaScript, `prova-node.js` is the command-line alternative.

6.2 ElaboratoreTesto

It contains only static functions (`Shared` in VB.NET, `static` in C#, an object literal in JavaScript).

Member	Description
`EParolaVuota(parola)`	True if the word (already normalized) is a stop word.
`TogliIntestazioneGutenberg(testo)`	Removes the Project Gutenberg license at the beginning and at the end, if present.
`Normalizza(testo)`	Lowercase, without accents, non-alphanumeric characters replaced by spaces.
`Parole(testo)`	Array of the normalized words.
`Radice(parola)`	Stem of the word (simplified stemmer).
`ParoleContenuto(testo)`	(stem, form) pairs of the content words.
`DividiInFrasi(testo, paroleMinime = 4)`	List of the sentences of the text.

6.3 ModelloConoscenza

Member	Description
`Vocabolario`	Stem → id dictionary.
`Radici`, `Forme`, `Frequenze`	For each id: the stem, the most frequent form, the number of occurrences.

Member	Description
`Idf`	Array of Single (`Float32Array` in JavaScript) with the IDF of each word.
`Embedding`	For each id, a vector of `Dimensione` Single values, or Nothing/null.
`Frasi`	List of `Frase` objects: `Testo`, `Termini` (sorted ids), `Pesi` (TF-IDF, norm 1), `Denso` (dense vector or null).
`Dimensione`, `Finestra`, `FrequenzaMinima`, `Seme`	Training hyperparameters (128, 4, 2, 12345).
`Origine`	Name of the text file from which the model was trained.
`Addestra(testo, log)`	Runs the eight steps of chapter 4; `log` receives one line per step.
`IdTermine(radice)`	Id of the stem, or -1.
`Simili(id, k)`	The k words with the closest embedding, as (id, cosine) pairs in descending order.
`Salva(percorso)` / `Carica(percorso)`	Writing and reading of the `.mai` file (chapter 7). In JavaScript `salva()` returns a `Uint8Array` and `carica(dati)` receives the bytes of the file.
`Prodotto(a, b)`, `Normalizza(v)`	Dot product (computed in Double) and normalization to length 1.
`StimaMB()`	Estimated memory used, in MB.

6.4 MotoreDomande

It is created by passing the model to the constructor; the parameters are public fields that the user interface updates before each question (see chapter 8).

```
' VB.NET
Dim motore As New MotoreDomande(modello)
motore.PesoSemantico = 0.5
Dim r As Risposta = motore.Rispondi("Dove vive la fata?")
Console.WriteLine(r.Testo)

// C#
var motore = new MotoreDomande(modello) { PesoSemantico = 0.5 };
Risposta r = motore.Rispondi("Dove vive la fata?");

// JavaScript
const motore = new MiniAI.MotoreDomande(modello);
motore.pesoSemantico = 0.5;
const r = motore.rispondi('Dove vive la fata?');
console.log(r.testo, r.punteggio);
```

6.5 The user interface

The window has three tabs: **Addestramento** (opening the text, parameters, log, saving and loading the model), **Domande** (question, answer parameters, answer and trace) and **Esplora la memoria**. In the last one you can see the 25 words most similar to a given word along with the first numbers of its vector, the similarity between two words, and the vocabulary of the 150 most frequent words with stem, frequency, IDF and whether an embedding is present.

In the .NET versions, training runs in a separate thread (`Task.Run`) and the log reaches the user interface through `IProgress`, so the window stays responsive. In the JavaScript version everything happens in the page thread; the `.mai` file is saved as a browser download.



Figure 3 – Esplora la memoria tab: the words with the vector closest to «naso»

6.6 Notes on porting between the three languages

Getting bit-identical results in three languages requires some care, which is useful to know if you modify the code:

- **Random numbers**: JavaScript does not allow you to set the seed of `Math.random`. `js/modello.js` contains `RandomNet`, an exact reproduction of the algorithm of .NET's `System.Random` (Knuth's subtractive generator), so the random vectors are the same.
- **Single and Double**: .NET uses `Single` (32-bit) for vectors and weights and `Double` (64-bit) for sums and dot products. JavaScript always computes in 64 bits; this is why the vectors are `Float32Array` and every product of two Single values is rounded with `Math.fround`, as .NET would do.
- **Dictionary order**: the .NET `Dictionary`, if no elements are removed, returns the keys in insertion order; JavaScript's `Map` does the same. Since the order of the sums changes the last decimal places, the code keeps the same order everywhere.
- **Sorting**: the ranking uses a stable sort (`OrderByDescending` in .NET, `Array.prototype.sort` in JavaScript); `Simili` uses `List.Sort` in .NET and `sort` in JavaScript, with equal results except for identical cosines.
- **Formatting**: the messages use the Italian number format (2.459; 2,75; 0,207%). The JavaScript version has a small `Formato` object that reproduces it without depending on the browser settings.
- **Compatibility with Framework 4.8**: the VB.NET version is compatible with Visual Basic 2012 and the C# version with C# 5, the languages of the compilers included in Windows, so that you can compile with `compila.bat` without installing anything.

7. The .mai file format

The trained model is saved in a binary file with the `.mai` extension. The format is that of .NET's `BinaryWriter`: 32-bit integers and Single numbers in little-endian; strings are in UTF-8, preceded by their length in bytes, encoded in groups of 7 bits (the highest bit of each byte indicates that another byte follows). A missing vector is written as length 0.

Field	Type	Notes
Signature	string	Always `MINIAI01`.
Dimensione, Finestra, FrequenzaMinima, Seme	4 × Int32	Hyperparameters used in training.
Origine	string	Name of the text file.
V	Int32	Number of words in the vocabulary.
for each word (V times)		Stem (string), Form (string), Frequency (Int32), IDF (Single), Embedding (vector).
N	Int32	Number of sentences.
for each sentence (N times)		Text (string), number of terms n (Int32), then n pairs id (Int32) + weight (Single), and finally the dense vector (vector).
vector	Int32 + Single…	Length followed by the values; length 0 = no vector.

With Pinocchio the file is about 2.8 MB. The co-occurrence and PPMI matrices are not saved: they are needed only during training. To answer, the vocabulary, IDF, embeddings and sentence vectors are enough.

8. Parameters and experiments

8.1 Training parameters

Parameter	Default	Effect
Finestra contesto ± (Context window ±)	4 (1–10)	How many words to the right and left count as context. Narrow window: more grammatical similarities; wide: more related to the subject.
Dimensione vettori (Vector size)	128 (16–512)	Length of the embeddings. More numbers = more faithful projection, but more memory and computation.
Frequenza minima (Minimum frequency)	2 (1–20)	Below this threshold the word receives no embedding. High values: fewer words, more reliable vectors.
Seme (Seed)	12345	Seed of the random generator (code only). Changing it changes the vectors, not the average quality.

8.2 Answer parameters

Parameter	Default	Effect
Peso semantico (Semantic weight)	0,30	0 = only identical words (lexical), 1 = only embedding similarity.
Peso contesto (Context weight)	0,25	How much the two neighboring sentences count.
Parole simili per termine (Similar words per term)	3	Words added to the question for each word. 0 = no expansion.
Soglia somiglianza (Similarity threshold)	0,35	Minimum similarity for a similar word to be added.
Regole tipo domanda (Question type rules)	enabled	Enables or disables the bonuses for chi/dove/quando/quanto/perché/come.
FrasiContesto, SogliaIncertezza	1; 0,12	Code only: sentences shown around the answer, and the threshold below which the answer is declared uncertain.

8.3 Guided experiments with Pinocchio

1. *Chi ha costruito il burattino?* with **Peso semantico** 0, then 0.30, then 1. With 0 and 0.30 the answer is «— Io sono un burattino.» (I am a puppet.); with 1 it becomes «Nascono burattini, vivono burattini e muoiono burattini.» (They are born puppets, live as puppets and die puppets.): only the subject counts, not the exact words.
2. *Chi ha mangiato Geppetto?* (Who ate Geppetto?) The answer is «— E chi te li ha mangiati?» (And who ate them for you?). The model is a «bag of words»: it does not distinguish who eats from who is eaten, and word order does not matter.
3. *Che cosa vende Geppetto per comprare l'abbecedario?* (What does Geppetto sell to buy the spelling book?) finds the title of chapter VIII: «Geppetto rifà i piedi a Pinocchio, e vende la propria casacca per comprargli l'Abbecedario» (Geppetto makes Pinocchio new feet and sells his own coat to buy him a spelling book). The expansion adds «casacca» (coat) as a word similar to «comprato» (bought): you can see it in the trace with the ~ symbol.

4. Set **Parole simili per termine** to 0 and repeat the previous question: without «casacca» the right sentence drops to fourth place and the answer becomes «— E perchè mi avete comprato?» (And why did you buy me?).
5. In the Esplora tab, search for `fata`, `geppetto`, `naso` (fairy, geppetto, nose). Many similar words are just words that are close together in the story («naso» → «giunti», «spropositato», «allungo»), not synonyms: with a single book the statistics are poor.
6. Retrain with **Finestra** 1 and then 8, and compare the words similar to `naso`. With window 8, «spropositato», «allungo», «punta» (enormous, lengthened, tip) appear: words from the same scene.
7. Type *astronave* (spaceship): the program answers that it does not know any of the words. It knows nothing outside the text it has read, and it says so.
8. Disable **Regole tipo domanda** and repeat *Chi ha costruito il burattino?*: without the bonus the answer becomes «Io sono un burattino.» (without the dash). The rules are the only handwritten knowledge; everything else comes from the data.
9. Train with another text, for example a technical manual or a collection of company procedures in `.txt` format, and ask questions about its content: the program does not depend on Pinocchio.

9. Limits and extensions

9.1 Intentional limits

- The stemmer is crude and designed only for Italian; the stop words are Italian.
- Word order is ignored («bag of words» model): subject and object cannot be told apart.
- Negations («non», not) are stop words: «è vero» (it is true) and «non è vero» (it is not true) come out the same.
- The embeddings come from a single text, so they reflect the proximities in that text and not the general meaning of the words.
- The search compares the question with all sentences: this is perfectly fine for thousands of sentences, not for millions.
- The answers are always existing sentences: the program does not summarize and does not combine information taken from different sentences.

9.2 From the toy to real AIs

Aspect	MiniAI	Large language models
Data	One book: about 20,000 content words	Trillions of words
Text unit	Word stems	Tokens (word pieces) from a fixed vocabulary
Word vectors	Co-occurrences + PPMI + random projection, 128 numbers	Learned by a neural network, thousands of numbers
Context	Fixed window of ±4 words	Attention mechanism over thousands of tokens, order included
Learned numbers	About 600,000 (word and sentence embeddings)	Billions of parameters
Answer	Picks an existing sentence	Generates new text, word by word
Training	Less than a second on a PC	Weeks on thousands of specialized processors

The differences are huge, but the substance is the same: representing language with vectors learned from context and measuring similarities. A modern RAG system does, on a much larger scale, what MiniAI does in steps B–E: it turns the question into a vector, looks for the closest passages and passes them to the language model, which uses them to formulate the answer.

9.3 Possible extensions

- **Generating sentences**: a Markov chain over words (or bigrams) trained on the same text, to see the step from choosing to generating.
- **Learning the vectors**: replace the random projection with a small word2vec network (skip-gram with negative sampling), still without external libraries.
- **More text**: train on several files together; this is the most effective improvement.
- **Bigrams**: add frequent pairs of consecutive words to the vocabulary («fata turchina», the blue-haired fairy) to preserve some word order.
- **Negations**: join «non» to the following word («non_vero») instead of discarding it.
- **Other languages**: make the stop words and suffixes configurable.

- **Fast search**: an inverted index (word → sentences) for the lexical score and an approximate index for the dense vectors, if the sentences grow to hundreds of thousands.

10. License and liability

MiniAI is distributed free of charge, like all DevService.net programs, and is used at your own risk: it is a study project, provided as is, without warranties of any kind. The source code may be read, modified and used for learning and experimenting.

The text `Pinocchio.txt` (*Le avventure di Pinocchio*, Carlo Collodi, 1883) is in the public domain and comes from Project Gutenberg (www.gutenberg.org); the Project Gutenberg license is included in the file itself.

For reports and suggestions, use the contact form on the DevService.net website.

MiniAI

Intelligence artificielle didactique : du texte aux vecteurs

Manuel du programmeur

Sources en VB.NET, C# et JavaScript

Octobre 2026

Sommaire

1. Introduction....26
1.1 Qu'est-ce que MiniAI....26
1.2 Le même principe que les grandes IA....26
1.3 Contenu du téléchargement....26
2. Configuration requise, compilation et lancement....28
2.1 Version VB.NET....28
2.2 Version C#....28
2.3 Version JavaScript....28
2.4 Premier lancement avec Pinocchio....28
3. Notions de base....30
3.1 Du texte aux nombres....30
3.2 La similarité cosinus....30
3.3 L'hypothèse distributionnelle....30
3.4 Recherche et génération....30
4. Le pipeline d'entraînement....31
4.1 Étape 1 – Découpage en phrases....31
4.2 Étape 2 – Normalisation, racines et vocabulaire....31
4.3 Étape 3 – IDF....31
4.4 Étape 4 – Matrice TF-IDF creuse....32
4.5 Étape 5 – Cooccurrences....32
4.6 Étape 6 – PPMI....32
4.7 Étape 7 – Embeddings par projection aléatoire....33
4.8 Étape 8 – Vecteurs denses des phrases....33
4.9 Le journal de l'entraînement....33
5. Répondre à une question....34
5.1 A – Type de question....34
5.2 B – Mots de la question....34
5.3 C – Expansion avec des mots similaires....34
5.4 D – Score des phrases....34
5.5 E – Contexte....35
5.6 F – Réponse....35
5.7 Lire la trace....35
6. Structure du code....37
6.1 Fichiers et correspondances....37
6.2 ElaboratoreTesto....37
6.3 ModelloConoscenza....37
6.4 MotoreDomande....38
6.5 L'interface....38
6.6 Remarques sur le portage entre les trois langages....39
7. Le format du fichier .mai....40
8. Paramètres et expériences....41
8.1 Paramètres d'entraînement....41
8.2 Paramètres de réponse....41
8.3 Expériences guidées avec Pinocchio....41
9. Limites et extensions....43
9.1 Limites voulues....43
9.2 Du jouet aux IA réelles....43
9.3 Extensions possibles....43
10. Licence et responsabilité....45

1. Introduction

1.1 Qu'est-ce que MiniAI

MiniAI est une étude personnelle de base sur l'intelligence artificielle appliquée au langage. Il lit un texte long (un livre, un recueil de documents, un manuel), le transforme en nombres, en vecteurs et en matrices, puis répond aux questions en cherchant dans le texte la phrase la plus proche de la question.

Ce n'est pas un produit destiné à la production : c'est un laboratoire. Chaque étape du calcul est visible, le programme écrit dans le journal les dimensions des matrices qu'il construit et, pour chaque réponse, il affiche la « trace du raisonnement » : quels mots il a reconnus, lesquels il a ajoutés, quelles phrases il a comparées et avec quels scores. Le but est de comprendre, pas d'épater.

Le programme est distribué sous forme de **code source**, en trois versions équivalentes : **VB.NET**, **C#** et **JavaScript**. Il n'utilise aucune bibliothèque externe : tout ce qui est nécessaire (du découpage en phrases à la projection aléatoire des embeddings) est écrit en toutes lettres en quelques centaines de lignes commentées.

1.2 Le même principe que les grandes IA

Évidemment, la quantité de données fait la différence. Les IA présentes sur le marché sont entraînées sur des milliards de mots, possèdent des milliards de paramètres et des réseaux de neurones qui apprennent à générer du texte nouveau. MiniAI lit un seul livre et apprend quelques milliers de vecteurs en moins d'une seconde. Le principe de base, pourtant, est le même :

- **le texte devient des nombres** : chaque mot et chaque phrase est représenté par un vecteur ;
- **le sens naît du contexte** : deux mots employés dans des contextes similaires reçoivent des vecteurs similaires (hypothèse distributionnelle) ;
- **comprendre une question, c'est mesurer des similarités** : la question est transformée dans le même espace vectoriel et comparée à ce que le système connaît ;
- **chercher avant de répondre** : la recherche des phrases pertinentes (*retrieval*) est le même mécanisme que celui des systèmes RAG (*Retrieval-Augmented Generation*), qui fournissent à un modèle de langage les documents dont il tire la réponse.

MiniAI s'arrête à la recherche : **il ne génère pas de texte, il choisit une phrase existante**. C'est justement pour cela qu'il constitue un bon point de départ : on voit clairement ce que la statistique parvient à faire seule et où il faut davantage de données ou des modèles plus puissants.

> Avec le même code, davantage de texte donne de meilleurs résultats. Avec un seul roman, les « mots similaires » ne sont souvent que des mots voisins dans le récit ; avec des dizaines de livres sur le même sujet, de vrais synonymes commencent à apparaître. C'est, en petit, la raison pour laquelle les grandes IA ont besoin d'énormes quantités de données.

1.3 Contenu du téléchargement

Le site propose trois fichiers zip, un par langage. Chacun contient le programme complet, le texte d'exemple et un fichier `LEGGIMI.md` en italien.

Fichier	Contenu
`MiniAI_VBNET.zip`	Projet Windows Forms pour .NET Framework 4.8 en VB.NET : `MiniAI.sln`, `MiniAI.vbproj`, `compila.bat` et les fichiers `Testo.vb`, `Modello.vb`, `Motore.vb`, `FormPrincipale.vb`, `Program.vb`.
`MiniAI_CSharp.zip`	Le même projet en C# : `MiniAI.sln`, `MiniAI.csproj`, `compila.bat` et les fichiers `.cs` portant les mêmes noms.

Fichier	Contenu
`MiniAI_JavaScript.zip`	Page web autonome (`index.html`, `stile.css`, dossier `js`) et `prova-node.js` pour l'utilisation en ligne de commande avec Node.js.
`Pinocchio.txt`	Dans les trois zip : *Le avventure di Pinocchio* (Les Aventures de Pinocchio) de Carlo Collodi (1883), texte du domaine public provenant du Project Gutenberg, utilisé pour tous les exemples de ce manuel.

Les trois versions effectuent les mêmes calculs dans le même ordre. Avec le même texte et les mêmes paramètres, elles produisent les mêmes réponses, la même trace et le même fichier de modèle `.mai`, identique octet par octet : un modèle enregistré par une version s'ouvre avec les autres.

2. Configuration requise, compilation et lancement

2.1 Version VB.NET

- Windows 10/11 ou Windows Server 2016 et versions ultérieures, avec .NET Framework 4.8 (déjà présent dans Windows 10 à partir de la version 1903, dans Windows 11 et dans Windows Server 2022 et versions ultérieures).
- **Avec Visual Studio** (2019 ou 2022, charge de travail « Développement .NET Desktop ») : ouvrez `MiniAI.sln` et appuyez sur F5.
- **Sans Visual Studio** : double-cliquez sur `compila.bat`. Le fichier utilise le compilateur `vbc.exe` que le .NET Framework 4.8 installe dans `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319`. Le code est écrit de manière compatible avec ce compilateur (Visual Basic 2012) et `MiniAI.exe` est créé dans le même dossier.

Les fichiers `.vb` sont enregistrés en UTF-8 avec BOM : ainsi, même le compilateur du Framework lit correctement les lettres accentuées des commentaires et des messages.

2.2 Version C#

- Mêmes prérequis que la version VB.NET.
- **Avec Visual Studio** : ouvrez `MiniAI.sln` (projet `MiniAI.csproj`) et appuyez sur F5.
- **Sans Visual Studio** : double-cliquez sur `compila.bat`, qui utilise `csc.exe` du .NET Framework 4.8. Le code est compatible avec C# 5, la version de ce compilateur : pas d'interpolation de chaînes ni d'autres syntaxes plus récentes.

2.3 Version JavaScript

- **Dans le navigateur** : double-cliquez sur `index.html`. Fonctionne avec des versions récentes d'Edge, de Chrome et de Firefox, même sans connexion Internet et sans serveur : le texte ne quitte jamais l'ordinateur.
- **Avec Node.js** (version 18 ou ultérieure), depuis le dossier du programme : `node prova-node.js Pinocchio.txt` entraîne le modèle et ouvre une session de questions ; `node prova-node.js Pinocchio.txt domande.txt modello.mai` répond à chaque ligne de `domande.txt` et enregistre le modèle ; `node prova-node.js modello.mai` repart d'un modèle enregistré. L'option `--traccia` affiche aussi la trace du raisonnement.

Dans le navigateur, l'entraînement bloque la page pendant la durée du calcul (moins d'une seconde avec Pinocchio) ; avec des textes très longs, la page peut sembler figée pendant quelques secondes.

2.4 Premier lancement avec Pinocchio

1. Onglet **1 · Addestramento** (Entraînement) : « Apri testo (.txt)... » (Ouvrir un texte) et choisissez `Pinocchio.txt` (dans la version JavaScript, vous pouvez aussi faire glisser le fichier sur la page).
2. Laissez les paramètres par défaut (fenêtre 4, dimension 128, fréquence minimale 2) et appuyez sur « Addestra » (Entraîner). Le journal liste les huit étapes décrites au chapitre 4.
3. Onglet **2 · Domande** (Questions) : tapez *Chi ha costruito il burattino?* (Qui a construit la marionnette ?) et appuyez sur Entrée. La réponse apparaît en haut avec les phrases voisines, la trace complète en dessous.
4. Onglet **3 · Esplora la memoria** (Explorer la mémoire) : tapez `naso` (nez) et appuyez sur « Parole simili » (Mots similaires).
5. Facultatif : « Salva modello... » (Enregistrer le modèle) crée un fichier `.mai` qui se recharge en un instant avec « Carica modello... » (Charger le modèle).

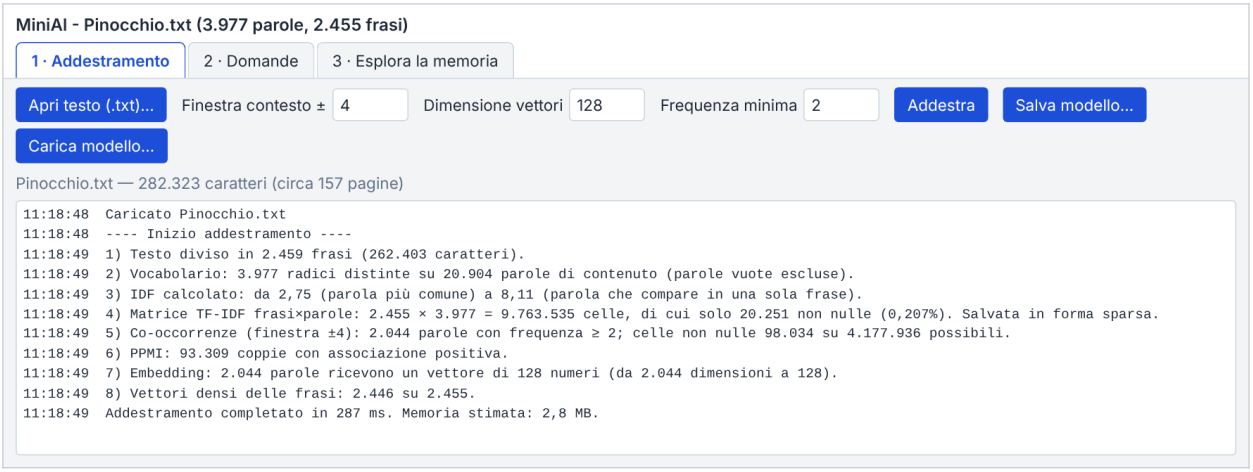

Figure 1 – Onglet Addestramento après l'entraînement sur Pinocchio (version JavaScript ; les versions Windows ont les trois mêmes onglets)

Le texte est lu en UTF-8 ; s'il contient des octets non valides en UTF-8, il est relu avec le codage Windows (Windows-1252), de sorte que les anciens fichiers `.txt` enregistrés avec le Bloc-notes fonctionnent aussi.

3. Notions de base

3.1 Du texte aux nombres

Un ordinateur ne sait pas ce qu'est un « burattino » (marionnette). Il sait en revanche compter et effectuer des opérations sur des listes de nombres. Tout le travail de MiniAI consiste à transformer des mots et des phrases en **vecteurs**, c'est-à-dire en listes de nombres de longueur fixe, de sorte que des textes de sens similaire donnent des vecteurs similaires.

Le programme utilise deux types de vecteurs :

- **vecteurs creux** : aussi longs que le vocabulaire (environ 4 000 positions avec Pinocchio), presque entièrement nuls ; chaque position correspond à un mot. Seules les positions non nulles sont mémorisées ;
- **vecteurs denses** (*embeddings*) : courts (128 nombres), tous significatifs ; aucune position ne correspond à un mot précis, c'est l'ensemble des nombres qui décrit le sens.

3.2 La similarité cosinus

Pour exprimer à quel point deux vecteurs sont similaires, MiniAI utilise le **cosinus de l'angle** qu'ils forment :

```
cosinus(a, b) = (a · b) / (|a| · |b|)

a · b = a1·b1 + a2·b2 + ... + an·bn      (produit scalaire)
|a|   = racine carrée de (a · a)         (longueur du vecteur)
```

Le résultat vaut 1 si les vecteurs pointent dans la même direction, 0 s'ils n'ont rien en commun, et des valeurs négatives s'ils sont opposés. C'est la direction qui compte, pas la longueur : une phrase longue et une phrase courte sur le même sujet sont jugées similaires.

Pour économiser des calculs, tous les vecteurs sont **normalisés** (ramenés à une longueur de 1) dès leur création. À partir de ce moment, le cosinus coïncide avec le simple produit scalaire : c'est ce que font les fonctions `Normalizza` et `Prodotto` de la classe `ModelloConoscenza`.

3.3 L'hypothèse distributionnelle

« Vous connaîtrez un mot par la compagnie qu'il fréquente » (J. R. Firth, 1957). Si deux mots apparaissent souvent près des mêmes autres mots, ils ont probablement un sens voisin. MiniAI applique cette idée en comptant, pour chaque mot, quels mots se trouvent près de lui dans le texte (matrice de cooccurrence) et en transformant ces comptages en un vecteur dense. word2vec, GloVe et les couches d'embedding des modèles de langage modernes reposent sur le même principe.

3.4 Recherche et génération

Il existe deux façons de répondre à une question à partir d'un texte :

- **extractive** (recherche) : on cherche dans le texte le passage le plus pertinent et on le renvoie tel quel. C'est ce que fait MiniAI et c'est le premier stade des systèmes RAG ;
- **générative** : un modèle de langage produit une phrase nouvelle, mot par mot, en choisissant à chaque fois le mot le plus probable. Cela nécessite des réseaux de neurones entraînés sur d'énormes quantités de texte.

La recherche a un avantage important : la réponse est toujours une phrase réelle du document, vérifiable, et le système peut dire honnêtement « non conosco nessuna di queste parole » (je ne connais aucun de ces mots).

4. Le pipeline d'entraînement

L'entraînement est effectué par la méthode `Addestra` de la classe `ModelloConoscenza` (fichier `Modello.vb`, `Modello.cs` ou `js/modello.js`). Il se compose de huit étapes ; chacune écrit une ligne dans le journal. Les nombres cités dans ce chapitre sont ceux obtenus avec `Pinocchio.txt` et les paramètres par défaut.

4.1 Étape 1 – Découpage en phrases

Si le fichier provient du Project Gutenberg, `TogliIntestazioneGutenberg` supprime la licence initiale et finale (il cherche les lignes `*** START OF` et `*** END OF`). Ensuite, `DividiInFrasi` :

- réunit les lignes coupées par les retours à la ligne « fixes » typiques des fichiers `.txt` ; les paragraphes sont séparés par une ligne vide ;
- découpe chaque paragraphe là où se trouve un signe de fin de phrase (. ! ? …), éventuellement suivi de guillemets, puis d'une espace et d'une lettre majuscule, d'un chiffre ou d'un tiret de dialogue ;
- écarte les phrases de moins de 4 mots.

La phrase est l'**unité de réponse** : MiniAI répond toujours par une phrase entière. Avec Pinocchio, on obtient 2 459 phrases sur 262 403 caractères.

4.2 Étape 2 – Normalisation, racines et vocabulaire

Chaque phrase est réduite à ses **mots de contenu** par la fonction `ParoleContenuto` :

1. `Normalizza` : tout en minuscules, accents supprimés (décomposition Unicode NFD et élimination des signes diacritiques), tout caractère qui n'est ni une lettre ni un chiffre devient une espace. « Perché l'albero? » devient `perche l albero`.
2. Les **mots vides** (articles, prépositions, auxiliaires, pronoms : environ 200 mots listés dans `Testo`) et les mots d'une seule lettre sont écartés : ils apparaissent partout et ne disent rien du contenu.
3. `Radice` : un stemmer volontairement simple supprime une terminaison connue (`-issimo`, `-mente`, `-zione`, `-avano`, `-ando`, `-ato`, `-ito`…), ou bien la voyelle finale. « burattino », « burattini » et « burattina » deviennent tous `burattin` ; « andato » et « andavano » deviennent `and`.

Chaque nouvelle racine reçoit un numéro séquentiel (son **id**) dans le dictionnaire `Vocabolario`. Pour chaque racine, on compte les occurrences (`Frequenze`) et on retient la forme la plus fréquente (`Forme`), utilisée uniquement pour afficher le mot de façon lisible. Avec Pinocchio : **3 977 racines** sur 20 904 mots de contenu.

> Le stemmer se trompe souvent, et il est instructif de voir où : « costruito » (construit) n'apparaît jamais dans Pinocchio, mais sa racine `costru` coïncide avec celle de « costruì », qui apparaît une fois. Le mot de la question est donc reconnu grâce à une autre forme du même verbe.

4.3 Étape 3 – IDF

Tous les mots ne sont pas également informatifs : « pinocchio » apparaît dans des centaines de phrases, « abbecedario » (abécédaire) dans très peu. L'**IDF** (*inverse document frequency*) mesure la rareté d'un mot :

```
idf(t) = ln( (N + 1) / (df(t) + 1) ) + 1

N     = nombre de phrases
df(t) = nombre de phrases contenant le mot t
```

Le +1 au numérateur et au dénominateur évite les divisions par zéro ; le +1 final garantit que même le mot le plus courant ait un poids positif. Avec Pinocchio, l'IDF va de 2,75 (le mot le plus courant, « pinocchio ») à 8,11 (mots qui n'apparaissent que dans une seule phrase). Voici la même ligne dans les trois versions :

```
' VB.NET
Idf(i) = CSng(Math.Log((N + 1.0) / (df(i) + 1.0)) + 1.0)

// C#
Idf[i] = (float)(Math.Log((N + 1.0) / (df[i] + 1.0)) + 1.0);

// JavaScript (Float32Array : la conversion en Single est automatique)
this.idf[i] = Math.log((N + 1.0) / (df[i] + 1.0)) + 1.0;
```

4.4 Étape 4 – Matrice TF-IDF creuse

Chaque phrase devient un vecteur aussi long que le vocabulaire. Le poids du mot t dans la phrase f est :

```
poids(t, f) = (1 + ln tf(t, f)) · idf(t)        tf = occurrences de t dans la phrase

puis le vecteur est normalisé à une longueur de 1
```

Le logarithme atténue les répétitions : un mot répété trois fois ne vaut pas le triple. La matrice complète phrases × mots aurait 2 455 × 3 977 = 9 763 535 cellules, mais seules 20 251 sont non nulles (0,207 %). C'est pourquoi chaque phrase ne mémorise que deux tableaux parallèles : `Termini` (les id des mots présents, par ordre croissant) et `Pesi` (les poids correspondants). C'est la **représentation creuse**, celle qu'utilisent les moteurs de recherche.

Les phrases composées uniquement de mots vides ne produisent pas de vecteur et sont écartées : c'est pourquoi les phrases mémorisées sont au nombre de 2 455 et non de 2 459.

4.5 Étape 5 – Cooccurrences

Ici commence la partie « sémantique ». On ne considère que les mots dont la fréquence est au moins égale à `FrequenzaMinima` (2) : les mots vus une seule fois n'ont pas assez de contextes pour un vecteur fiable. Avec Pinocchio, ils sont 2 044.

Les mots de contenu de toutes les phrases sont mis bout à bout en un flux unique. Pour chaque mot a et pour chaque mot b qui le suit à moins de `Finestra` positions (4), on ajoute aux deux cellules (a, b) et (b, a) un poids qui décroît avec la distance :

```
cooc(a, b) += 1 / distance        distance = 1, 2, ..., Finestra
```

La matrice théorique 2 044 × 2 044 compte 4 177 936 cellules, mais les cellules non nulles sont 98 034 : elle aussi est mémorisée sous forme creuse, dans un dictionnaire dont la clé est `a · V + b` (un entier sur 64 bits).

4.6 Étape 6 – PPMI

Le comptage brut favorise les mots fréquents : « disse » (dit) se trouve près de presque tout. La **PMI** (*pointwise mutual information*) compare le nombre de fois où deux mots sont voisins avec le nombre de fois auquel on s'attendrait par pur hasard :

```
PMI(a, b)  = ln( P(a, b) / ( P(a) · Pc(b) ) )

P(a, b) = cooc(a, b) / total
P(a)    = somme de la ligne a / total
Pc(b)   = (somme de la ligne b)^0,75 / somme de toutes les (sommes de ligne)^0,75

PPMI(a, b) = max(0, PMI(a, b))
```

On ne garde que les valeurs positives (**PPMI**, *positive* PMI) : les paires qui apparaissent ensemble plus souvent que le hasard. L'exposant 0,75 sur la probabilité du contexte est la même astuce que dans word2vec et réduit le poids excessif des contextes rares. Avec Pinocchio : 93 309 paires avec une association positive.

4.7 Étape 7 – Embeddings par projection aléatoire

Chaque ligne de la matrice PPMI est déjà un vecteur qui décrit un mot, mais elle compte 2 044 positions. Pour obtenir des vecteurs courts, MiniAI utilise une **projection aléatoire** :

1. à chaque mot-contexte b est attribué un vecteur aléatoire fixe r(b) de `Dimensione` nombres (128), tirés d'une distribution normale d'écart type 1/√D (transformation de Box-Muller) ;
2. l'embedding du mot a est la somme des vecteurs de ses contextes, pondérée par la PPMI : e(a) = Σ PPMI(a, b) · r(b) ;
3. le vecteur est normalisé à une longueur de 1.

Le **lemme de Johnson-Lindenstrauss** garantit qu'une projection aléatoire conserve approximativement les distances et les cosinus entre les vecteurs d'origine : deux mots aux lignes PPMI similaires ont aussi des embeddings similaires. C'est une technique connue sous le nom de *random indexing* ; les modèles modernes obtiennent le même résultat en apprenant les vecteurs avec un réseau de neurones.

Le générateur aléatoire part toujours de la même `Seme` (graine, 12345), l'entraînement est donc **reproductible** : le même texte et les mêmes paramètres donnent toujours les mêmes vecteurs. Avec Pinocchio, 2 044 mots reçoivent un embedding ; les mots trop rares en restent dépourvus (valeur `Nothing`/`null`).

4.8 Étape 8 – Vecteurs denses des phrases

Le vecteur dense d'une phrase est la moyenne des embeddings de ses mots, pondérée par le poids TF-IDF calculé à l'étape 4, puis normalisée. Les mots rares et spécifiques comptent davantage que les mots courants. Avec Pinocchio, 2 446 phrases sur 2 455 ont un vecteur dense ; les autres ne contiennent que des mots sans embedding.

4.9 Le journal de l'entraînement

Voici le journal complet produit avec Pinocchio et les paramètres par défaut ; les trois versions écrivent les mêmes lignes :

```
1) Testo diviso in 2.459 frasi (262.403 caratteri).
2) Vocabolario: 3.977 radici distinte su 20.904 parole di contenuto
   (parole vuote escluse).
3) IDF calcolato: da 2,75 (parola più comune) a 8,11 (parola che compare
   in una sola frase).
4) Matrice TF-IDF frasi×parole: 2.455 × 3.977 = 9.763.535 celle, di cui
   solo 20.251 non nulle (0,207%). Salvata in forma sparsa.
5) Co-occorrenze (finestra ±4): 2.044 parole con frequenza ≥ 2; celle non
   nulle 98.034 su 4.177.936 possibili.
6) PPMI: 93.309 coppie con associazione positiva.
7) Embedding: 2.044 parole ricevono un vettore di 128 numeri (da 2.044
   dimensioni a 128).
8) Vettori densi delle frasi: 2.446 su 2.455.
```

L'entraînement prend de quelques dixièmes de seconde à une seconde, selon l'ordinateur ; le modèle occupe environ 2,8 Mo en mémoire et sur le disque.

5. Répondre à une question

La réponse est produite par la méthode `Rispondi` de la classe `MotoreDomande` (fichier `Motore.vb`, `Motore.cs` ou `js/motore.js`). La méthode renvoie un objet `Risposta` avec trois champs : le texte de la réponse, la trace et le score de la phrase choisie. Les étapes sont désignées par les lettres A à F, les mêmes que celles qui apparaissent dans la trace.

5.1 A – Type de question

Si parmi les trois premiers mots de la question se trouve *chi* (qui), *dove* (où), *quando* (quand), *quanto* (combien ; ou *quanti*, *quante*, *quanta*), *perché* (pourquoi) ou *come* (comment), le type de question est reconnu. C'est la seule partie du programme écrite « à la main » avec des règles fixes ; elle sert à comparer la connaissance programmée avec celle apprise à partir des données.

5.2 B – Mots de la question

La question subit les mêmes transformations que le texte : normalisation, élimination des mots vides, réduction à la racine. Chaque racine est cherchée dans le vocabulaire. Les mots jamais vus sont listés dans la trace comme « MAI viste nel testo » (JAMAIS vus dans le texte) : le modèle ne sait rien en dehors du texte lu. Si aucun mot n'est connu, la réponse est *Non conosco nessuna delle parole della domanda* (Je ne connais aucun des mots de la question).

Les mots reconnus reçoivent le poids TF-IDF, comme pour les phrases : (1 + ln tf) · idf.

5.3 C – Expansion avec des mots similaires

Pour chaque mot de la question, on cherche les `Espansione` (3) mots dont l'embedding est le plus similaire. Ceux dont la similarité est au moins égale à `SogliaEspansione` (0,35) sont ajoutés à la question avec un poids réduit :

```
poids(mot ajouté) = idf · similarité · 0,5
```

C'est ainsi qu'une question sur « vendere » (vendre) peut trouver une phrase qui parle de « mercato » (marché). Dans la trace, les mots ajoutés sont signalés par le symbole ~. Le vecteur de la question est ensuite normalisé.

On calcule aussi le **vecteur dense de la question** : la somme des embeddings des mots d'origine (pas de ceux ajoutés), pondérés par l'IDF, normalisée.

5.4 D – Score des phrases

Chaque phrase mémorisée reçoit trois scores :

- **lexical** : cosinus entre le vecteur creux de la question et celui de la phrase. On additionne les produits uniquement sur les mots communs, le calcul est donc très rapide ;
- **sémantique** : cosinus entre le vecteur dense de la question et celui de la phrase (les valeurs négatives deviennent 0). Il favorise les phrases qui traitent du même sujet, même avec des mots différents ;
- **bonus** : une petite prime (0,02–0,03) si la phrase contient des indices adaptés au type de question : un nom propre pour *chi* (en pratique, un mot à initiale majuscule qui n'est pas le premier de la phrase), un mot de temps pour *quando*, un lieu pour *dove*, une cause pour *perché*, un nombre pour *quanto*, une manière pour *come*. Le bonus ne s'applique qu'aux phrases qui ont au moins un mot en commun avec la question.

```
brut = (1 - ps) · lexical + ps · sémantique + bonus        ps = PesoSemantico (0,30)
```

5.5 E – Contexte

Dans un récit, une phrase pertinente est souvent entourée de phrases pertinentes. Le score final ajoute une partie de la moyenne des scores des deux phrases voisines :

```
total(i) = brut(i) + pc · (brut(i-1) + brut(i+1)) / 2     pc = PesoContesto (0,25)
```

Les phrases sont triées par score total décroissant (tri stable : à score égal, c'est la phrase qui vient en premier dans le texte qui l'emporte) et la trace affiche les cinq premières.

5.6 F – Réponse

La phrase ayant le score le plus élevé est la réponse. Si le score est inférieur à `SogliaIncertezza` (0,12), la réponse est précédée de *(Risposta incerta: punteggio basso)* (réponse incertaine : score faible) ; s'il est nul, le programme répond *Non ho trovato nulla di pertinente* (Je n'ai rien trouvé de pertinent). Sous la réponse sont affichées `FrasiContesto` phrases avant et après (1), pour la lire dans son contexte.

5.7 Lire la trace

Voici la trace de la question *Chi ha costruito il burattino?* avec les paramètres par défaut :

```
A) TIPO DI DOMANDA: CHI
B) PAROLE DELLA DOMANDA
   ignorate (parole vuote): chi, ha, il
   costrui            radice 'costru'  frequenza     1  IDF 8,11  (senza embedding)
   burattino          radice 'burattin'  frequenza   226  IDF 3,40
C) ESPANSIONE CON PAROLE SIMILI (embedding)
   costrui            -> nessuna sopra soglia
   burattino          -> guardare (0,35)
D) CLASSIFICA (2.455 frasi confrontate)   peso semantico 0,30, peso contesto 0,25
   pos  frase   lessic.  semant.  bonus   TOTALE   parole in comune
     1   1026   0,385    1,000    0,030   0,607    burattino
             — Io sono un burattino.
     2   2140   0,385    1,000    0,000   0,588    burattino
             Io sono un burattino.
     3   1245   0,174    1,000    0,000   0,500    burattino
             Nascono burattini, vivono burattini e muoiono burattini.
```

On remarque plusieurs choses instructives. « costruito » n'est reconnu qu'à travers la racine de « costruì », qui apparaît une fois et n'a donc pas d'embedding. Le score sémantique vaut 1,000 pour les phrases qui ne contiennent que le mot « burattino » : leur vecteur dense coïncide avec celui de la question. La première et la deuxième phrase ont le même score lexical ; la première ne l'emporte que grâce au bonus *chi* : après le tiret de dialogue, le mot « Io » (je) a une initiale majuscule et la règle le prend pour un nom propre. Une règle écrite à la main est simple à comprendre, mais tout aussi facile à tromper. La réponse est fausse d'un point de vue humain (la bonne réponse est « Geppetto »), mais c'est exactement ce que suggère la statistique de ce texte.

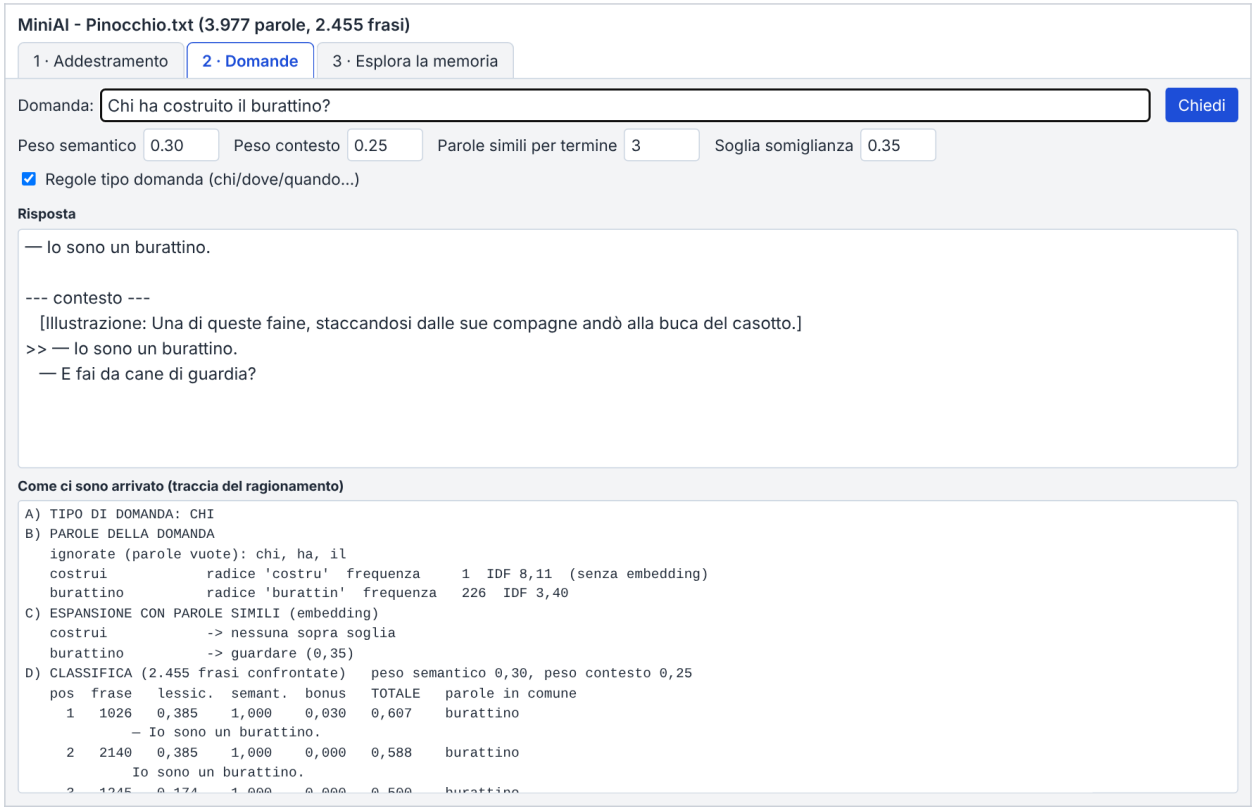

Figure 2 – Onglet Domande : la réponse avec son contexte et, en dessous, la trace du raisonnement

6. Structure du code

6.1 Fichiers et correspondances

Les noms des classes, des méthodes et des variables sont en italien et sont les mêmes dans les trois versions. En JavaScript, ils suivent la convention du langage (initiale minuscule : `rispondi` au lieu de `Rispondi`).

VB.NET	C#	JavaScript	Contenu
`Testo.vb`	`Testo.cs`	`js/testo.js`	Classe `ElaboratoreTesto` : phrases, normalisation, mots vides, racines.
`Modello.vb`	`Modello.cs`	`js/modello.js`	Classes `Frase` et `ModelloConoscenza` : entraînement, mots similaires, enregistrement et chargement.
`Motore.vb`	`Motore.cs`	`js/motore.js`	Classes `Risposta` et `MotoreDomande` : réponse aux questions et trace.
`FormPrincipale.vb`	`FormPrincipale.cs`	`index.html`, `js/app.js`, `stile.css`	Interface à trois onglets. Les versions .NET construisent les contrôles dans le code, sans fichier du Concepteur.
`Program.vb`	`Program.cs`	`prova-node.js`	Point d'entrée. En JavaScript, `prova-node.js` est l'alternative en ligne de commande.

6.2 ElaboratoreTesto

Contient uniquement des fonctions statiques (`Shared` en VB.NET, `static` en C#, un objet littéral en JavaScript).

Membre	Description
`EParolaVuota(parola)`	True si le mot (déjà normalisé) est un mot vide.
`TogliIntestazioneGutenberg(testo)`	Supprime la licence initiale et finale du Project Gutenberg, si elles sont présentes.
`Normalizza(testo)`	Minuscules, sans accents, caractères non alphanumériques remplacés par des espaces.
`Parole(testo)`	Tableau des mots normalisés.
`Radice(parola)`	Racine du mot (stemmer simplifié).
`ParoleContenuto(testo)`	Paires (racine, forme) des mots de contenu.
`DividiInFrasi(testo, paroleMinime = 4)`	Liste des phrases du texte.

6.3 ModelloConoscenza

Membre	Description
`Vocabolario`	Dictionnaire racine → id.

Membre	Description
`Radici`, `Forme`, `Frequenze`	Pour chaque id : la racine, la forme la plus fréquente, le nombre d'occurrences.
`Idf`	Tableau de Single (`Float32Array` en JavaScript) contenant l'IDF de chaque mot.
`Embedding`	Pour chaque id, un vecteur de `Dimensione` Single, ou bien Nothing/null.
`Frasi`	Liste d'objets `Frase` : `Testo`, `Termini` (id triés), `Pesi` (TF-IDF, norme 1), `Denso` (vecteur dense ou null).
`Dimensione`, `Finestra`, `FrequenzaMinima`, `Seme`	Hyperparamètres de l'entraînement (128, 4, 2, 12345).
`Origine`	Nom du fichier texte à partir duquel le modèle a été entraîné.
`Addestra(testo, log)`	Exécute les huit étapes du chapitre 4 ; `log` reçoit une ligne par étape.
`IdTermine(radice)`	Id de la racine, ou -1.
`Simili(id, k)`	Les k mots dont l'embedding est le plus proche, sous forme de paires (id, cosinus) par ordre décroissant.
`Salva(percorso)` / `Carica(percorso)`	Écriture et lecture du fichier `.mai` (chapitre 7). En JavaScript, `salva()` renvoie un `Uint8Array` et `carica(dati)` reçoit les octets du fichier.
`Prodotto(a, b)`, `Normalizza(v)`	Produit scalaire (calculé en Double) et normalisation à une longueur de 1.
`StimaMB()`	Mémoire occupée estimée, en Mo.

6.4 MotoreDomande

Il se crée en passant le modèle au constructeur ; les paramètres sont des champs publics que l'interface met à jour avant chaque question (voir le chapitre 8).

```
' VB.NET
Dim motore As New MotoreDomande(modello)
motore.PesoSemantico = 0.5
Dim r As Risposta = motore.Rispondi("Dove vive la fata?")
Console.WriteLine(r.Testo)

// C#
var motore = new MotoreDomande(modello) { PesoSemantico = 0.5 };
Risposta r = motore.Rispondi("Dove vive la fata?");

// JavaScript
const motore = new MiniAI.MotoreDomande(modello);
motore.pesoSemantico = 0.5;
const r = motore.rispondi('Dove vive la fata?');
console.log(r.testo, r.punteggio);
```

6.5 L'interface

La fenêtre comporte trois onglets : **Addestramento** (ouverture du texte, paramètres, journal, enregistrement et chargement du modèle), **Domande** (question, paramètres de réponse, réponse et trace) et **Esplora la memoria**. Ce dernier permet de voir les 25 mots les plus similaires à un mot avec les premiers nombres de son vecteur, la similarité entre deux mots et le vocabulaire des 150 mots les plus fréquents avec racine, fréquence, IDF et présence de l'embedding.

Dans les versions .NET, l'entraînement s'exécute dans un thread séparé (`Task.Run`) et le journal parvient à l'interface via `IProgress`, de sorte que la fenêtre reste réactive. Dans la version JavaScript, tout se passe dans le thread de la page ; le fichier `.mai` est enregistré comme un téléchargement du navigateur.

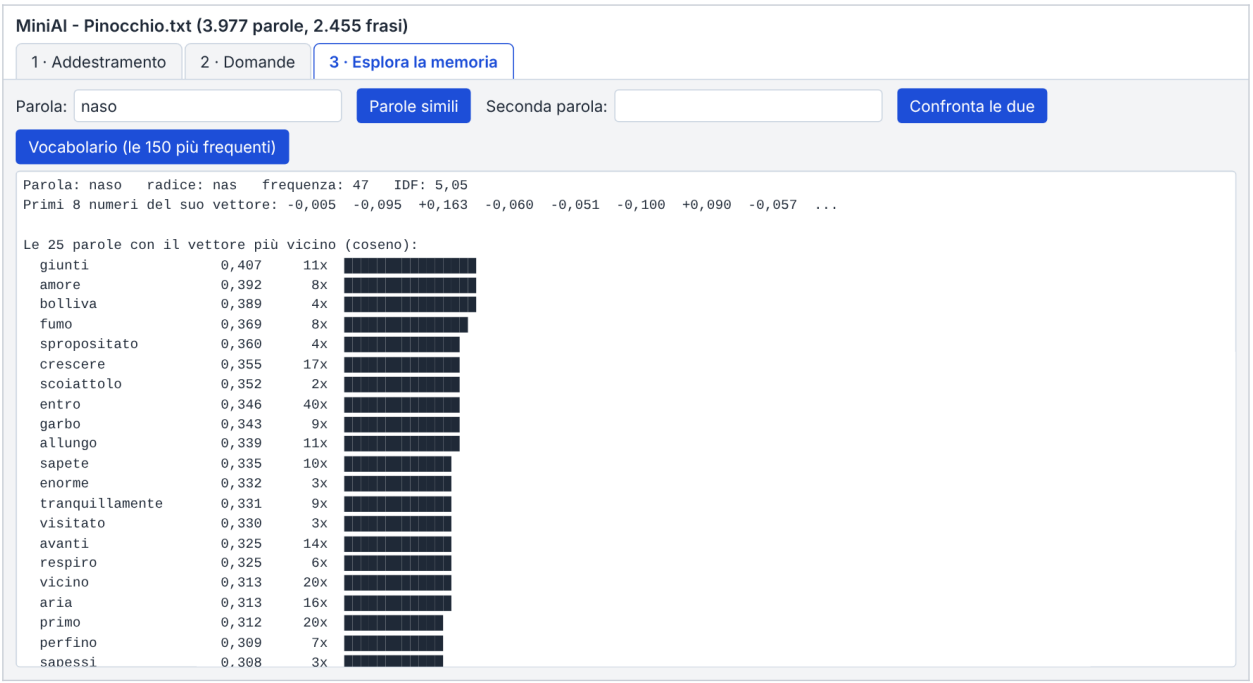

Figure 3 – Onglet Esplora la memoria : les mots dont le vecteur est le plus proche de « naso »

6.6 Remarques sur le portage entre les trois langages

Obtenir des résultats identiques au bit près dans trois langages demande quelques précautions, utiles à connaître si vous modifiez le code :

- **Nombres aléatoires** : JavaScript ne permet pas de fixer la graine de `Math.random`. `js/modello.js` contient `RandomNet`, une reproduction exacte de l'algorithme de `System.Random` de .NET (générateur soustractif de Knuth), de sorte que les vecteurs aléatoires sont les mêmes.
- **Single et Double** : .NET utilise `Single` (32 bits) pour les vecteurs et les poids et `Double` (64 bits) pour les sommes et les produits scalaires. JavaScript calcule toujours sur 64 bits ; c'est pourquoi les vecteurs sont des `Float32Array` et chaque produit entre deux valeurs Single est arrondi avec `Math.fround`, comme le ferait .NET.
- **Ordre des dictionnaires** : le `Dictionary` de .NET, si l'on ne supprime pas d'éléments, renvoie les clés dans l'ordre d'insertion ; la `Map` de JavaScript fait de même. Comme l'ordre des sommes modifie les dernières décimales, le code conserve partout le même ordre.
- **Tris** : le classement utilise un tri stable (`OrderByDescending` en .NET, `Array.prototype.sort` en JavaScript) ; `Simili` utilise `List.Sort` en .NET et `sort` en JavaScript, avec des résultats identiques sauf en cas de cosinus identiques.
- **Formatage** : les messages utilisent le format italien (2.459; 2,75; 0,207%). La version JavaScript dispose d'un petit objet `Formato` qui le reproduit sans dépendre des paramètres du navigateur.
- **Compatibilité avec le Framework 4.8** : la version VB.NET est compatible avec Visual Basic 2012 et la version C# avec C# 5, les langages des compilateurs inclus dans Windows, afin de pouvoir compiler avec `compila.bat` sans rien installer.

7. Le format du fichier .mai

Le modèle entraîné s'enregistre dans un fichier binaire portant l'extension `.mai`. Le format est celui de `BinaryWriter` de .NET : entiers sur 32 bits et nombres Single en little-endian ; les chaînes sont en UTF-8, précédées de leur longueur en octets, codée par groupes de 7 bits (le bit de poids fort de chaque octet indique qu'un autre octet suit). Un vecteur absent s'écrit comme une longueur 0.

Champ	Type	Remarques
Signature	chaîne	Toujours `MINIAI01`.
Dimensione, Finestra, FrequenzaMinima, Seme	4 × Int32	Hyperparamètres utilisés lors de l'entraînement.
Origine	chaîne	Nom du fichier texte.
V	Int32	Nombre de mots du vocabulaire.
pour chaque mot (V fois)		Racine (chaîne), Forme (chaîne), Fréquence (Int32), IDF (Single), Embedding (vecteur).
N	Int32	Nombre de phrases.
pour chaque phrase (N fois)		Texte (chaîne), nombre de termes n (Int32), puis n paires id (Int32) + poids (Single), enfin le vecteur dense (vecteur).
vecteur	Int32 + Single…	Longueur suivie des valeurs ; longueur 0 = aucun vecteur.

Avec Pinocchio, le fichier pèse environ 2,8 Mo. Les matrices de cooccurrence et PPMI ne sont pas enregistrées : elles ne servent que pendant l'entraînement. Pour répondre, le vocabulaire, l'IDF, les embeddings et les vecteurs des phrases suffisent.

8. Paramètres et expériences

8.1 Paramètres d'entraînement

Paramètre	Par défaut	Effet
Finestra contesto ± (Fenêtre de contexte)	4 (1–10)	Nombre de mots à droite et à gauche qui comptent comme contexte. Fenêtre étroite : similarités plus grammaticales ; large : davantage liées au sujet.
Dimensione vettori (Dimension des vecteurs)	128 (16–512)	Longueur des embeddings. Plus de nombres = projection plus fidèle, mais davantage de mémoire et de calculs.
Frequenza minima (Fréquence minimale)	2 (1–20)	En dessous de ce seuil, le mot ne reçoit pas d'embedding. Valeurs élevées : moins de mots, vecteurs plus fiables.
Seme	12345	Graine du générateur aléatoire (dans le code uniquement). La modifier change les vecteurs, pas la qualité moyenne.

8.2 Paramètres de réponse

Paramètre	Par défaut	Effet
Peso semantico (Poids sémantique)	0,30	0 = uniquement les mots identiques (lexical), 1 = uniquement la similarité des embeddings.
Peso contesto (Poids du contexte)	0,25	Importance des deux phrases voisines.
Parole simili per termine (Mots similaires par terme)	3	Mots ajoutés à la question pour chaque mot. 0 = aucune expansion.
Soglia somiglianza (Seuil de similarité)	0,35	Similarité minimale pour qu'un mot similaire soit ajouté.
Regole tipo domanda (Règles de type de question)	activées	Active ou désactive les bonus pour chi/dove/quando/quanto/perché/come.
FrasiContesto, SogliaIncertezza	1; 0,12	Dans le code uniquement : phrases affichées autour de la réponse et seuil en dessous duquel la réponse est déclarée incertaine.

8.3 Expériences guidées avec Pinocchio

1. *Chi ha costruito il burattino?* avec **Peso semantico** à 0, puis 0,30, puis 1. Avec 0 et 0,30, la réponse est « — Io sono un burattino. » ; avec 1, elle devient « Nascono burattini, vivono burattini e muoiono burattini. » : seul le sujet compte, pas les mots exacts.
2. *Chi ha mangiato Geppetto?* (Qui a mangé Geppetto ?) La réponse est « — E chi te li ha mangiati? ». Le modèle est un « sac de mots » : il ne distingue pas celui qui mange de celui qui est mangé et l'ordre des mots ne compte pas.
3. *Che cosa vende Geppetto per comprare l'abbecedario?* (Que vend Geppetto pour acheter l'abécédaire ?) trouve le titre du chapitre VIII : « Geppetto rifà i piedi a Pinocchio, e vende la propria casacca per comprargli l'Abbecedario ». L'expansion ajoute « casacca » (casaque) comme mot similaire à « comprato » : on le voit dans la trace grâce au symbole ~.

4. Réglez **Parole simili per termine** sur 0 et répétez la question précédente : sans « casacca », la bonne phrase descend à la quatrième place et la réponse devient « — E perchè mi avete comprato? ».
5. Dans l'onglet Esplora, cherchez `fata`, `geppetto`, `naso`. Beaucoup de mots similaires ne sont que des mots voisins dans le récit (« naso » → « giunti », « spropositato », « allungo »), et non des synonymes : avec un seul livre, la statistique est pauvre.
6. Réentraînez avec **Finestra** à 1 puis à 8 et comparez les mots similaires à `naso`. Avec une fenêtre de 8 apparaissent « spropositato », « allungo », « punta » : des mots de la même scène.
7. Tapez *astronave* (vaisseau spatial) : le programme répond qu'il ne connaît aucun des mots. Il ne sait rien en dehors du texte lu, et le déclare.
8. Désactivez **Regole tipo domanda** et répétez *Chi ha costruito il burattino?* : sans le bonus, la réponse devient « Io sono un burattino. » (sans tiret). Les règles sont la seule connaissance écrite à la main ; tout le reste provient des données.
9. Entraînez avec un autre texte, par exemple un manuel technique ou un recueil de procédures d'entreprise au format `.txt`, et posez des questions sur son contenu : le programme ne dépend pas de Pinocchio.

9. Limites et extensions

9.1 Limites voulues

- Le stemmer est rudimentaire et conçu uniquement pour l'italien ; les mots vides sont italiens.
- L'ordre des mots est ignoré (modèle « sac de mots ») : le sujet et l'objet ne se distinguent pas.
- Les négations (« non ») sont des mots vides : « è vero » (c'est vrai) et « non è vero » (ce n'est pas vrai) sont considérés comme identiques.
- Les embeddings proviennent d'un seul texte ; ils reflètent donc les proximités de ce texte et non le sens général des mots.
- La recherche compare la question à toutes les phrases : cela convient très bien pour des milliers de phrases, pas pour des millions.
- Les réponses sont toujours des phrases existantes : le programme ne résume pas et ne combine pas des informations tirées de phrases différentes.

9.2 Du jouet aux IA réelles

Aspect	MiniAI	Grands modèles de langage
Données	Un livre : environ 20 000 mots de contenu	Des milliers de milliards de mots
Unité de texte	Racines de mots	Tokens (morceaux de mots) d'un vocabulaire fixe
Vecteurs des mots	Cooccurrences + PPMI + projection aléatoire, 128 nombres	Appris par un réseau de neurones, des milliers de nombres
Contexte	Fenêtre fixe de ±4 mots	Mécanisme d'attention sur des milliers de tokens, ordre compris
Nombres appris	Environ 600 000 (embeddings de mots et de phrases)	Des milliards de paramètres
Réponse	Choisit une phrase existante	Génère du texte nouveau, mot par mot
Entraînement	Moins d'une seconde sur un PC	Des semaines sur des milliers de processeurs spécialisés

Les différences sont énormes, mais le fond est commun : représenter le langage par des vecteurs appris à partir du contexte et mesurer des similarités. Un système RAG moderne fait, à une échelle beaucoup plus grande, ce que MiniAI fait aux étapes B–E : il transforme la question en vecteur, cherche les passages les plus proches et les transmet au modèle de langage, qui s'en sert pour formuler la réponse.

9.3 Extensions possibles

- **Générer des phrases** : une chaîne de Markov sur les mots (ou sur les bigrammes) entraînée sur le même texte, pour voir le passage du choix à la génération.
- **Apprendre les vecteurs** : remplacer la projection aléatoire par un petit réseau word2vec (skip-gram avec échantillonnage négatif), toujours sans bibliothèque externe.
- **Plus de texte** : entraîner sur plusieurs fichiers à la fois ; c'est l'amélioration la plus efficace.
- **Bigrammes** : ajouter au vocabulaire les paires de mots consécutifs fréquentes (« fata turchina », la fée aux cheveux bleus) pour conserver un peu d'ordre.
- **Négations** : rattacher « non » au mot suivant (« non_vero ») au lieu de l'écarter.

- **Autres langues** : rendre configurables les mots vides et les suffixes.
- **Recherche rapide** : un index inversé (mot → phrases) pour le score lexical et un index approximatif pour les vecteurs denses, si les phrases se comptent par centaines de milliers.

10. Licence et responsabilité

MiniAI est distribué gratuitement, comme tous les programmes de DevService.net, et s'utilise à vos propres risques : c'est un projet d'étude, fourni tel quel, sans garantie d'aucune sorte. Le code source peut être lu, modifié et utilisé pour apprendre et expérimenter.

Le texte `Pinocchio.txt` (*Le avventure di Pinocchio*, Carlo Collodi, 1883) appartient au domaine public et provient du Project Gutenberg (www.gutenberg.org) ; la licence du Project Gutenberg est incluse dans le fichier lui-même.

Pour tout signalement ou suggestion, le formulaire de contact du site DevService.net est à votre disposition.

MiniAI

Intelligenza artificiale didattica: dal testo ai vettori

Manuale del programmatore

Sorgenti in VB.NET, C# e JavaScript

Ottobre 2026

Indice

1. Introduzione....48
1.1 Che cos'è MiniAI....48
1.2 Lo stesso principio delle grandi AI....48
1.3 Contenuto del download....48
2. Requisiti, compilazione e avvio....50
2.1 Versione VB.NET....50
2.2 Versione C#....50
2.3 Versione JavaScript....50
2.4 Primo avvio con Pinocchio....50
3. Concetti di base....52
3.1 Dal testo ai numeri....52
3.2 La somiglianza coseno....52
3.3 L'ipotesi distribuzionale....52
3.4 Recupero e generazione....52
4. La pipeline di addestramento....53
4.1 Passo 1 – Divisione in frasi....53
4.2 Passo 2 – Normalizzazione, radici e vocabolario....53
4.3 Passo 3 – IDF....53
4.4 Passo 4 – Matrice TF-IDF sparsa....54
4.5 Passo 5 – Co-occorrenze....54
4.6 Passo 6 – PPMI....54
4.7 Passo 7 – Embedding con proiezione casuale....55
4.8 Passo 8 – Vettori densi delle frasi....55
4.9 Il log dell'addestramento....55
5. Rispondere a una domanda....56
5.1 A – Tipo di domanda....56
5.2 B – Parole della domanda....56
5.3 C – Espansione con parole simili....56
5.4 D – Punteggio delle frasi....56
5.5 E – Contesto....56
5.6 F – Risposta....57
5.7 Leggere la traccia....57
6. Struttura del codice....59
6.1 File e corrispondenze....59
6.2 ElaboratoreTesto....59
6.3 ModelloConoscenza....59
6.4 MotoreDomande....60
6.5 L'interfaccia....60
6.6 Note sul porting fra i tre linguaggi....61
7. Il formato del file .mai....62
8. Parametri ed esperimenti....63
8.1 Parametri di addestramento....63
8.2 Parametri di risposta....63
8.3 Esperimenti guidati con Pinocchio....63
9. Limiti ed estensioni....65
9.1 Limiti voluti....65
9.2 Dal giocattolo alle AI reali....65
9.3 Possibili estensioni....65
10. Licenza e responsabilità....67

1. Introduzione

1.1 Che cos'è MiniAI

MiniAI è un mio studio di base sull'intelligenza artificiale applicata al linguaggio. Legge un testo lungo (un libro, una raccolta di documenti, un manuale), lo trasforma in numeri, vettori e matrici e poi risponde alle domande cercando nel testo la frase più vicina alla domanda.

Non è un prodotto da usare in produzione: è un laboratorio. Ogni passo del calcolo è visibile, il programma scrive nel log le dimensioni delle matrici che costruisce e, per ogni risposta, mostra la «traccia del ragionamento»: quali parole ha riconosciuto, quali ha aggiunto, quali frasi ha confrontato e con quali punteggi. Lo scopo è capire, non stupire.

Il programma è distribuito come **codice sorgente**, in tre versioni equivalenti: **VB.NET**, **C#** e **JavaScript**. Non usa librerie esterne: tutto ciò che serve (dalla divisione in frasi alla proiezione casuale degli embedding) è scritto per esteso in poche centinaia di righe commentate.

1.2 Lo stesso principio delle grandi AI

Ovviamente la quantità di dati fa la differenza. Le AI presenti sul mercato sono addestrate su miliardi di parole, hanno miliardi di parametri e reti neurali che imparano a generare testo nuovo. MiniAI legge un solo libro e impara qualche migliaio di vettori in meno di un secondo. Il principio di base, però, è lo stesso:

- **il testo diventa numeri**: ogni parola e ogni frase è rappresentata da un vettore;
- **il significato nasce dal contesto**: due parole usate in contesti simili ricevono vettori simili (ipotesi distribuzionale);
- **capire una domanda vuol dire misurare somiglianze**: la domanda viene trasformata nello stesso spazio vettoriale e confrontata con ciò che il sistema conosce;
- **cercare prima di rispondere**: il recupero delle frasi pertinenti (*retrieval*) è lo stesso meccanismo usato dai sistemi RAG (*Retrieval-Augmented Generation*), che forniscono a un modello linguistico i documenti da cui trarre la risposta.

MiniAI si ferma al recupero: **non genera testo, sceglie una frase esistente**. Proprio per questo è un buon punto di partenza: si vede con chiarezza che cosa la statistica riesce a fare da sola e dove servono più dati o modelli più potenti.

> Con lo stesso codice, più testo dà risultati migliori. Con un solo romanzo le «parole simili» sono spesso solo parole vicine nel racconto; con decine di libri sullo stesso argomento iniziano a comparire veri sinonimi. È, in piccolo, il motivo per cui le grandi AI hanno bisogno di enormi quantità di dati.

1.3 Contenuto del download

Il sito offre tre file zip, uno per linguaggio. Ognuno contiene il programma completo, il testo di esempio e un file `LEGGIMI.md` in italiano.

File	Contenuto
`MiniAI_VBNET.zip`	Progetto Windows Forms per .NET Framework 4.8 in VB.NET: `MiniAI.sln`, `MiniAI.vbproj`, `compila.bat` e i file `Testo.vb`, `Modello.vb`, `Motore.vb`, `FormPrincipale.vb`, `Program.vb`.
`MiniAI_CSharp.zip`	Lo stesso progetto in C#: `MiniAI.sln`, `MiniAI.csproj`, `compila.bat` e i file `.cs` con gli stessi nomi.
`MiniAI_JavaScript.zip`	Pagina web autonoma (`index.html`, `stile.css`, cartella `js`) e `prova-node.js` per l'uso da riga di comando con Node.js.

File	Contenuto
`Pinocchio.txt`	In tutti e tre gli zip: *Le avventure di Pinocchio* di Carlo Collodi (1883), testo di pubblico dominio da Project Gutenberg, usato per tutti gli esempi di questo manuale.

Le tre versioni eseguono gli stessi calcoli nello stesso ordine. Con lo stesso testo e gli stessi parametri producono le stesse risposte, la stessa traccia e lo stesso file di modello `.mai`, identico byte per byte: un modello salvato da una versione si apre con le altre.

2. Requisiti, compilazione e avvio

2.1 Versione VB.NET

- Windows 10/11 o Windows Server 2016 e successivi, con .NET Framework 4.8 (già presente in Windows 10 dalla versione 1903, in Windows 11 e in Windows Server 2022 e successivi).
- **Con Visual Studio** (2019 o 2022, carico di lavoro «Sviluppo per desktop .NET»): aprire `MiniAI.sln` e premere F5.
- **Senza Visual Studio**: doppio clic su `compila.bat`. Il file usa il compilatore `vbc.exe` che il .NET Framework 4.8 installa in `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319`. Il codice è scritto in modo compatibile con quel compilatore (Visual Basic 2012) e viene creato `MiniAI.exe` nella stessa cartella.

I file `.vb` sono salvati in UTF-8 con BOM: in questo modo anche il compilatore del Framework legge correttamente le lettere accentate dei commenti e dei messaggi.

2.2 Versione C#

- Stessi requisiti della versione VB.NET.
- **Con Visual Studio**: aprire `MiniAI.sln` (progetto `MiniAI.csproj`) e premere F5.
- **Senza Visual Studio**: doppio clic su `compila.bat`, che usa `csc.exe` del .NET Framework 4.8. Il codice è compatibile con C# 5, la versione di quel compilatore: niente interpolazione di stringhe né altre sintassi più recenti.

2.3 Versione JavaScript

- **Nel browser**: doppio clic su `index.html`. Funziona con Edge, Chrome e Firefox recenti, anche senza connessione a Internet e senza server: il testo non lascia mai il computer.
- **Con Node.js** (versione 18 o successiva), dalla cartella del programma: `node prova-node.js Pinocchio.txt` addestra il modello e apre una sessione di domande; `node prova-node.js Pinocchio.txt domande.txt modello.mai` risponde a ogni riga di `domande.txt` e salva il modello; `node prova-node.js modello.mai` riparte da un modello salvato. L'opzione `--traccia` stampa anche la traccia del ragionamento.

Nel browser l'addestramento blocca la pagina per il tempo del calcolo (meno di un secondo con Pinocchio); con testi molto lunghi la pagina può sembrare ferma per qualche secondo.

2.4 Primo avvio con Pinocchio

1. Scheda **1 · Addestramento**: «Apri testo (.txt)...» e scegliere `Pinocchio.txt` (nella versione JavaScript si può anche trascinare il file sulla pagina).
2. Lasciare i parametri predefiniti (finestra 4, dimensione 128, frequenza minima 2) e premere «Addestra». Il log elenca gli otto passi descritti nel capitolo 4.
3. Scheda **2 · Domande**: scrivere *Chi ha costruito il burattino?* e premere Invio. Sopra compare la risposta con le frasi vicine, sotto la traccia completa.
4. Scheda **3 · Esplora la memoria**: scrivere `naso` e premere «Parole simili».
5. Facoltativo: «Salva modello...» crea un file `.mai` che si ricarica in un istante con «Carica modello...».

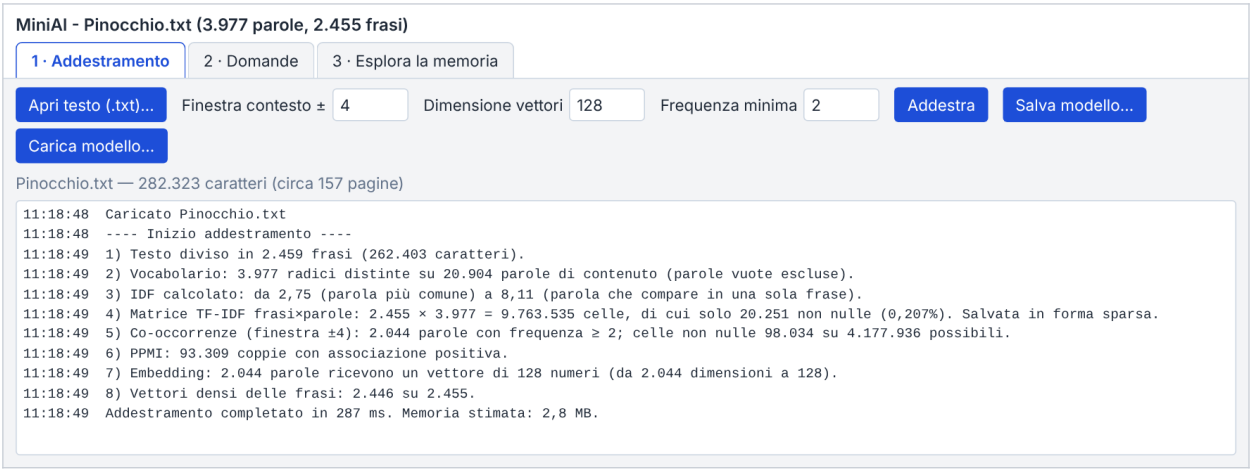

Figura 1 – Scheda Addestramento dopo l'addestramento su Pinocchio (versione JavaScript; le versioni Windows hanno le stesse tre schede)

Il testo viene letto come UTF-8; se contiene byte non validi in UTF-8 viene riletto con la codifica di Windows (Windows-1252), così funzionano anche i vecchi file `.txt` salvati con il Blocco note.

3. Concetti di base

3.1 Dal testo ai numeri

Un computer non sa che cosa sia un «burattino». Sa però contare e fare operazioni su liste di numeri. Tutto il lavoro di MiniAI consiste nel trasformare parole e frasi in **vettori**, cioè liste di numeri di lunghezza fissa, in modo che testi con significato simile diano vettori simili.

Nel programma compaiono due tipi di vettori:

- **vettori sparsi**: lunghi quanto il vocabolario (circa 4.000 posizioni con Pinocchio), quasi tutti zeri; ogni posizione corrisponde a una parola. Si memorizzano solo le posizioni diverse da zero;
- **vettori densi** (*embedding*): corti (128 numeri), tutti significativi; nessuna posizione corrisponde a una parola precisa, è l'insieme dei numeri a descrivere il significato.

3.2 La somiglianza coseno

Per dire quanto due vettori sono simili MiniAI usa il **coseno dell'angolo** fra di essi:

```
coseno(a, b) = (a · b) / (|a| · |b|)

a · b = a1·b1 + a2·b2 + ... + an·bn      (prodotto scalare)
|a|   = radice quadrata di (a · a)        (lunghezza del vettore)
```

Il risultato vale 1 se i vettori puntano nella stessa direzione, 0 se non hanno nulla in comune, valori negativi se sono opposti. Conta la direzione, non la lunghezza: una frase lunga e una corta sullo stesso argomento risultano simili.

Per risparmiare calcoli, tutti i vettori vengono **normalizzati** (portati a lunghezza 1) appena creati. Da quel momento il coseno coincide con il semplice prodotto scalare: è ciò che fanno le funzioni `Normalizza` e `Prodotto` della classe `ModelloConoscenza`.

3.3 L'ipotesi distribuzionale

«Conoscerai una parola dalla compagnia che frequenta» (J. R. Firth, 1957). Se due parole compaiono spesso vicino alle stesse altre parole, probabilmente hanno un significato affine. MiniAI applica questa idea contando, per ogni parola, quali parole le stanno vicino nel testo (matrice di co-occorrenza) e trasformando questi conteggi in un vettore denso. Sullo stesso principio si basano word2vec, GloVe e gli strati di embedding dei modelli linguistici moderni.

3.4 Recupero e generazione

Esistono due modi di rispondere a una domanda partendo da un testo:

- **estrattivo** (recupero): si cerca nel testo il brano più pertinente e lo si restituisce così com'è. È ciò che fa MiniAI ed è il primo stadio dei sistemi RAG;
- **generativo**: un modello linguistico produce una frase nuova, parola per parola, scegliendo ogni volta la parola più probabile. Richiede reti neurali addestrate su enormi quantità di testo.

Il recupero ha un pregio importante: la risposta è sempre una frase vera del documento, verificabile, e il sistema può dire con onestà «non conosco nessuna di queste parole».

4. La pipeline di addestramento

L'addestramento è svolto dal metodo `Addestra` della classe `ModelloConoscenza` (file `Modello.vb`, `Modello.cs` o `js/modello.js`). Si compone di otto passi; ognuno scrive una riga nel log. I numeri riportati in questo capitolo sono quelli ottenuti con `Pinocchio.txt` e i parametri predefiniti.

4.1 Passo 1 – Divisione in frasi

Se il file viene da Project Gutenberg, `TogliIntestazioneGutenberg` elimina la licenza iniziale e finale (cerca le righe `*** START OF` e `*** END OF`). Poi `DividiInFrasi`:

- unisce le righe spezzate dagli a capo «fissi» tipici dei file `.txt`; i paragrafi sono separati da una riga vuota;
- divide ogni paragrafo dove c'è un segno di fine frase (. ! ? …), eventualmente seguito da virgolette, poi uno spazio e una lettera maiuscola, una cifra o un trattino di dialogo;
- scarta le frasi con meno di 4 parole.

La frase è l'**unità di risposta**: MiniAI risponde sempre con una frase intera. Con Pinocchio si ottengono 2.459 frasi su 262.403 caratteri.

4.2 Passo 2 – Normalizzazione, radici e vocabolario

Ogni frase viene ridotta alle sue **parole di contenuto** dalla funzione `ParoleContenuto`:

1. `Normalizza`: tutto minuscolo, accenti tolti (scomposizione Unicode NFD ed eliminazione dei segni diacritici), ogni carattere che non è lettera o cifra diventa uno spazio. «Perché l'albero?» diventa `perche l albero`.
2. Le **parole vuote** (articoli, preposizioni, ausiliari, pronomi: circa 200 parole elencate in `Testo`) e le parole di una sola lettera vengono scartate: compaiono ovunque e non dicono nulla sul contenuto.
3. `Radice`: uno stemmer volutamente semplice toglie una desinenza nota (`-issimo`, `-mente`, `-zione`, `-avano`, `-ando`, `-ato`, `-ito`…), oppure la vocale finale. «burattino», «burattini» e «burattina» diventano tutte `burattin`; «andato» e «andavano» diventano `and`.

Ogni radice nuova riceve un numero progressivo (il suo **id**) nel dizionario `Vocabolario`. Per ogni radice si contano le occorrenze (`Frequenze`) e si ricorda la forma più frequente (`Forme`), usata solo per mostrare la parola in modo leggibile. Con Pinocchio: **3.977 radici** su 20.904 parole di contenuto.

> Lo stemmer sbaglia spesso, ed è istruttivo vedere dove: «costruito» non compare mai in Pinocchio, ma la sua radice `costru` coincide con quella di «costruì», che compare una volta. La parola della domanda viene quindi riconosciuta grazie a un'altra forma dello stesso verbo.

4.3 Passo 3 – IDF

Non tutte le parole sono ugualmente informative: «pinocchio» compare in centinaia di frasi, «abbecedario» in pochissime. L'**IDF** (*inverse document frequency*) misura la rarità di una parola:

```
idf(t) = ln( (N + 1) / (df(t) + 1) ) + 1

N     = numero di frasi
df(t) = numero di frasi che contengono la parola t
```

Il +1 al numeratore e al denominatore evita divisioni per zero; il +1 finale garantisce che anche la parola più comune abbia peso positivo. Con Pinocchio l'IDF va da 2,75 (la parola più comune, «pinocchio») a 8,11 (parole che compaiono in una sola frase). Ecco la stessa riga nelle tre versioni:

```
' VB.NET
Idf(i) = CSng(Math.Log((N + 1.0) / (df(i) + 1.0)) + 1.0)

// C#
Idf[i] = (float)(Math.Log((N + 1.0) / (df[i] + 1.0)) + 1.0);

// JavaScript (Float32Array: la conversione a Single è automatica)
this.idf[i] = Math.log((N + 1.0) / (df[i] + 1.0)) + 1.0;
```

4.4 Passo 4 – Matrice TF-IDF sparsa

Ogni frase diventa un vettore lungo quanto il vocabolario. Il peso della parola t nella frase f è:

```
peso(t, f) = (1 + ln tf(t, f)) · idf(t)        tf = occorrenze di t nella frase

poi il vettore viene normalizzato a lunghezza 1
```

Il logaritmo smorza le ripetizioni: una parola ripetuta tre volte non vale il triplo. La matrice completa frasi × parole avrebbe 2.455 × 3.977 = 9.763.535 celle, ma solo 20.251 sono diverse da zero (0,207%). Per questo ogni frase memorizza soltanto due array paralleli: `Termini` (gli id delle parole presenti, in ordine crescente) e `Pesi` (i rispettivi pesi). È la **rappresentazione sparsa**, la stessa usata dai motori di ricerca.

Le frasi formate solo da parole vuote non producono un vettore e vengono scartate: per questo le frasi memorizzate sono 2.455 e non 2.459.

4.5 Passo 5 – Co-occorrenze

Qui inizia la parte «semantica». Si considerano solo le parole con frequenza almeno pari a `FrequenzaMinima` (2): le parole viste una sola volta non hanno abbastanza contesti per un vettore affidabile. Con Pinocchio sono 2.044.

Le parole di contenuto di tutte le frasi vengono messe in fila in un unico flusso. Per ogni parola a e per ogni parola b che la segue entro `Finestra` posizioni (4) si aggiunge a entrambe le celle (a, b) e (b, a) un peso che decresce con la distanza:

```
cooc(a, b) += 1 / distanza        distanza = 1, 2, ..., Finestra
```

La matrice teorica 2.044 × 2.044 ha 4.177.936 celle, ma quelle non nulle sono 98.034: anche questa è memorizzata in forma sparsa, in un dizionario la cui chiave è `a · V + b` (un intero a 64 bit).

4.6 Passo 6 – PPMI

Il conteggio grezzo premia le parole frequenti: «disse» sta vicino a quasi tutto. La **PMI** (*pointwise mutual information*) confronta quante volte due parole stanno vicine con quante volte ci si aspetterebbe per puro caso:

```
PMI(a, b)  = ln( P(a, b) / ( P(a) · Pc(b) ) )

P(a, b) = cooc(a, b) / totale
P(a)    = somma della riga a / totale
Pc(b)   = (somma della riga b)^0,75 / somma di tutte le (somme di riga)^0,75

PPMI(a, b) = max(0, PMI(a, b))
```

Si tengono solo i valori positivi (**PPMI**, *positive* PMI): le coppie che stanno insieme più spesso del caso. L'esponente 0,75 sulla probabilità del contesto è lo stesso accorgimento di word2vec e riduce il peso eccessivo dei contesti rari. Con Pinocchio: 93.309 coppie con associazione positiva.

4.7 Passo 7 – Embedding con proiezione casuale

Ogni riga della matrice PPMI è già un vettore che descrive una parola, ma è lunga 2.044 posizioni. Per ottenere vettori corti MiniAI usa una **proiezione casuale**:

1. a ogni parola-contesto b viene assegnato un vettore casuale fisso r(b) di `Dimensione` numeri (128), estratti da una distribuzione normale con deviazione standard 1/√D (trasformazione di Box-Muller);
2. l'embedding della parola a è la somma dei vettori dei suoi contesti, pesata con la PPMI: e(a) = Σ PPMI(a, b) · r(b);
3. il vettore viene normalizzato a lunghezza 1.

Il **lemma di Johnson-Lindenstrauss** garantisce che una proiezione casuale conservi approssimativamente le distanze e i coseni fra i vettori originali: due parole con righe PPMI simili hanno anche embedding simili. È una tecnica nota come *random indexing*; i modelli moderni ottengono lo stesso risultato imparando i vettori con una rete neurale.

Il generatore casuale parte sempre dallo stesso `Seme` (12345), quindi l'addestramento è **riproducibile**: stesso testo e stessi parametri danno sempre gli stessi vettori. Con Pinocchio 2.044 parole ricevono un embedding; le parole troppo rare restano senza (valore `Nothing`/`null`).

4.8 Passo 8 – Vettori densi delle frasi

Il vettore denso di una frase è la media degli embedding delle sue parole, pesata con il peso TF-IDF calcolato al passo 4, poi normalizzata. Le parole rare e specifiche contano di più di quelle comuni. Con Pinocchio 2.446 frasi su 2.455 hanno un vettore denso; le altre contengono solo parole senza embedding.

4.9 Il log dell'addestramento

Questo è il log completo prodotto con Pinocchio e i parametri predefiniti; le tre versioni scrivono le stesse righe:

```
1) Testo diviso in 2.459 frasi (262.403 caratteri).
2) Vocabolario: 3.977 radici distinte su 20.904 parole di contenuto
   (parole vuote escluse).
3) IDF calcolato: da 2,75 (parola più comune) a 8,11 (parola che compare
   in una sola frase).
4) Matrice TF-IDF frasi×parole: 2.455 × 3.977 = 9.763.535 celle, di cui
   solo 20.251 non nulle (0,207%). Salvata in forma sparsa.
5) Co-occorrenze (finestra ±4): 2.044 parole con frequenza ≥ 2; celle non
   nulle 98.034 su 4.177.936 possibili.
6) PPMI: 93.309 coppie con associazione positiva.
7) Embedding: 2.044 parole ricevono un vettore di 128 numeri (da 2.044
   dimensioni a 128).
8) Vettori densi delle frasi: 2.446 su 2.455.
```

L'addestramento richiede da qualche decimo di secondo a un secondo, secondo il computer; il modello occupa circa 2,8 MB in memoria e su disco.

5. Rispondere a una domanda

La risposta è prodotta dal metodo `Rispondi` della classe `MotoreDomande` (file `Motore.vb`, `Motore.cs` o `js/motore.js`). Il metodo restituisce un oggetto `Risposta` con tre campi: il testo della risposta, la traccia e il punteggio della frase scelta. I passi sono indicati con le lettere da A a F, le stesse che compaiono nella traccia.

5.1 A – Tipo di domanda

Se fra le prime tre parole della domanda c'è *chi*, *dove*, *quando*, *quanto* (o *quanti*, *quante*, *quanta*), *perché* o *come*, il tipo di domanda viene riconosciuto. È l'unica parte del programma scritta «a mano» con regole fisse e serve a confrontare la conoscenza programmata con quella appresa dai dati.

5.2 B – Parole della domanda

La domanda subisce le stesse trasformazioni del testo: normalizzazione, eliminazione delle parole vuote, riduzione a radice. Ogni radice viene cercata nel vocabolario. Le parole mai viste vengono elencate nella traccia come «MAI viste nel testo»: il modello non sa nulla al di fuori del testo letto. Se nessuna parola è nota, la risposta è *Non conosco nessuna delle parole della domanda*.

Alle parole riconosciute si assegna il peso TF-IDF, come per le frasi: (1 + ln tf) · idf.

5.3 C – Espansione con parole simili

Per ogni parola della domanda si cercano le `Espansione` (3) parole con l'embedding più simile. Quelle con somiglianza almeno pari a `SogliaEspansione` (0,35) vengono aggiunte alla domanda con peso ridotto:

```
peso(parola aggiunta) = idf · somiglianza · 0,5
```

È così che una domanda su «vendere» può trovare una frase che parla di «mercato». Nella traccia le parole aggiunte sono indicate con il simbolo ~. Il vettore della domanda viene poi normalizzato.

Si calcola anche il **vettore denso della domanda**: la somma degli embedding delle parole originali (non di quelle aggiunte), pesati con l'IDF, normalizzata.

5.4 D – Punteggio delle frasi

Ogni frase memorizzata riceve tre punteggi:

- **lessicale**: coseno fra il vettore sparso della domanda e quello della frase. Si sommano i prodotti solo sulle parole in comune, quindi il calcolo è rapidissimo;
- **semantico**: coseno fra il vettore denso della domanda e quello della frase (i valori negativi diventano 0). Premia le frasi che parlano dello stesso argomento anche con parole diverse;
- **bonus**: un piccolo premio (0,02–0,03) se la frase contiene indizi adatti al tipo di domanda: un nome proprio per *chi* (in pratica una parola con l'iniziale maiuscola che non sia la prima della frase), una parola di tempo per *quando*, un luogo per *dove*, una causa per *perché*, un numero per *quanto*, un modo per *come*. Il bonus si applica solo alle frasi che hanno almeno una parola in comune con la domanda.

```
grezzo = (1 - ps) · lessicale + ps · semantico + bonus        ps = PesoSemantico (0,30)
```

5.5 E – Contesto

In un racconto una frase pertinente è spesso circondata da frasi pertinenti. Il punteggio finale aggiunge una parte della media dei punteggi delle due frasi vicine:

```
totale(i) = grezzo(i) + pc · (grezzo(i-1) + grezzo(i+1)) / 2     pc = PesoContesto (0,25)
```

Le frasi vengono ordinate per punteggio totale decrescente (ordinamento stabile: a parità di punteggio vince la frase che viene prima nel testo) e la traccia mostra le prime cinque.

5.6 F – Risposta

La frase con il punteggio più alto è la risposta. Se il punteggio è inferiore a `SogliaIncertezza` (0,12) la risposta è preceduta da *(Risposta incerta: punteggio basso)*; se è zero il programma risponde *Non ho trovato nulla di pertinente*. Sotto la risposta vengono mostrate `FrasiContesto` frasi prima e dopo (1), per leggerla nel suo contesto.

5.7 Leggere la traccia

Questa è la traccia della domanda *Chi ha costruito il burattino?* con i parametri predefiniti:

```
A) TIPO DI DOMANDA: CHI
B) PAROLE DELLA DOMANDA
   ignorate (parole vuote): chi, ha, il
   costrui            radice 'costru'  frequenza     1  IDF 8,11  (senza embedding)
   burattino          radice 'burattin'  frequenza   226  IDF 3,40
C) ESPANSIONE CON PAROLE SIMILI (embedding)
   costrui            -> nessuna sopra soglia
   burattino          -> guardare (0,35)
D) CLASSIFICA (2.455 frasi confrontate)   peso semantico 0,30, peso contesto 0,25
   pos  frase   lessic.  semant.  bonus   TOTALE   parole in comune
     1   1026   0,385    1,000    0,030   0,607    burattino
             — Io sono un burattino.
     2   2140   0,385    1,000    0,000   0,588    burattino
             Io sono un burattino.
     3   1245   0,174    1,000    0,000   0,500    burattino
             Nascono burattini, vivono burattini e muoiono burattini.
```

Si notano diverse cose istruttive. «costruito» viene riconosciuto solo attraverso la radice di «costruì», che compare una volta e quindi non ha embedding. Il punteggio semantico vale 1,000 per le frasi che contengono solo la parola «burattino»: il loro vettore denso coincide con quello della domanda. La prima e la seconda frase hanno lo stesso punteggio lessicale; la prima vince solo per il bonus *chi*: dopo il trattino di dialogo la parola «Io» ha l'iniziale maiuscola e la regola la scambia per un nome proprio. Una regola scritta a mano è semplice da capire, ma altrettanto facile da ingannare. La risposta è sbagliata dal punto di vista umano (la risposta giusta è «Geppetto»), ma è esattamente ciò che la statistica di questo testo suggerisce.

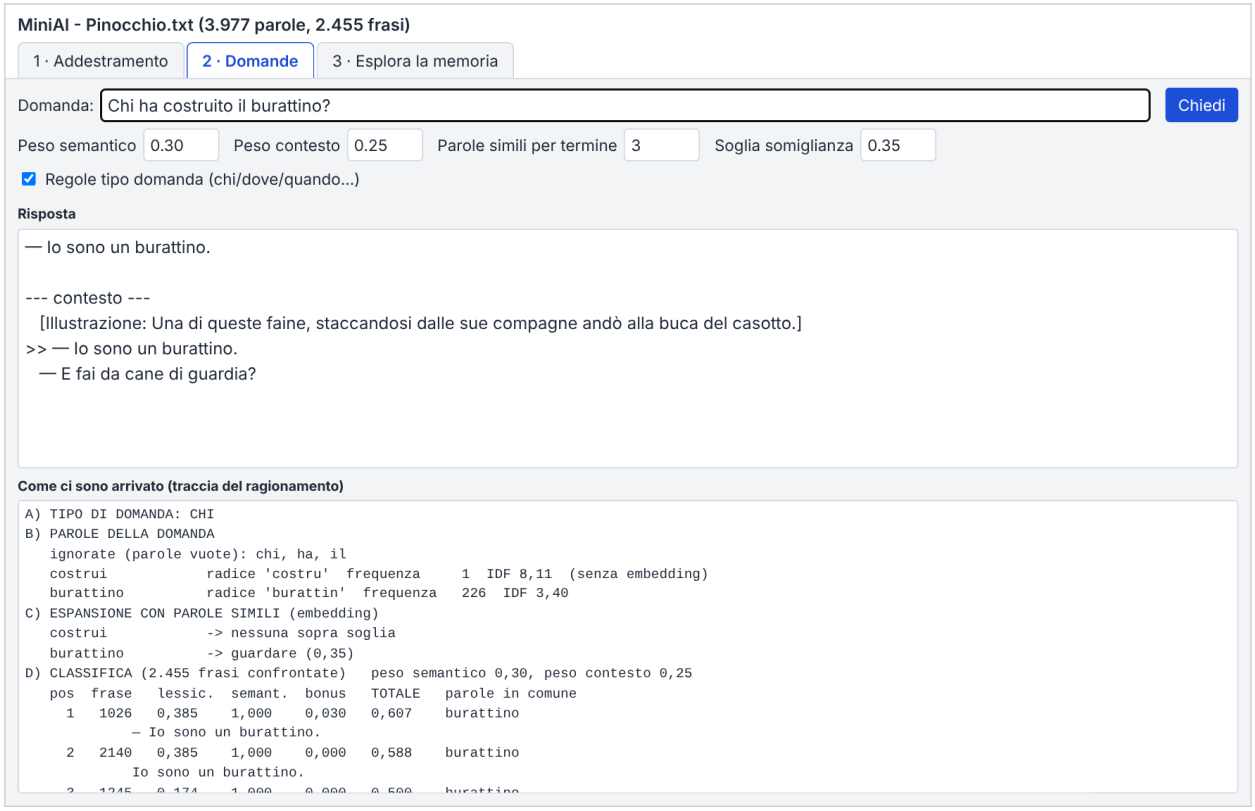

Figura 2 – Scheda Domande: la risposta con il suo contesto e, sotto, la traccia del ragionamento

6. Struttura del codice

6.1 File e corrispondenze

I nomi di classi, metodi e variabili sono in italiano e sono gli stessi nelle tre versioni. In JavaScript seguono la convenzione del linguaggio (iniziale minuscola: `rispondi` invece di `Rispondi`).

VB.NET	C#	JavaScript	Contenuto
`Testo.vb`	`Testo.cs`	`js/testo.js`	Classe `ElaboratoreTesto`: frasi, normalizzazione, parole vuote, radici.
`Modello.vb`	`Modello.cs`	`js/modello.js`	Classi `Frase` e `ModelloConoscenza`: addestramento, parole simili, salvataggio e caricamento.
`Motore.vb`	`Motore.cs`	`js/motore.js`	Classi `Risposta` e `MotoreDomande`: risposta alle domande e traccia.
`FormPrincipale.vb`	`FormPrincipale.cs`	`index.html`, `js/app.js`, `stile.css`	Interfaccia a tre schede. Le versioni .NET costruiscono i controlli da codice, senza file del Designer.
`Program.vb`	`Program.cs`	`prova-node.js`	Punto di avvio. In JavaScript `prova-node.js` è l'alternativa a riga di comando.

6.2 ElaboratoreTesto

Contiene solo funzioni statiche (`Shared` in VB.NET, `static` in C#, un oggetto letterale in JavaScript).

Membro	Descrizione
`EParolaVuota(parola)`	True se la parola (già normalizzata) è una parola vuota.
`TogliIntestazioneGutenberg(testo)`	Toglie licenza iniziale e finale di Project Gutenberg, se presenti.
`Normalizza(testo)`	Minuscolo, senza accenti, caratteri non alfanumerici sostituiti da spazi.
`Parole(testo)`	Array delle parole normalizzate.
`Radice(parola)`	Radice della parola (stemmer semplificato).
`ParoleContenuto(testo)`	Coppie (radice, forma) delle parole di contenuto.
`DividiInFrasi(testo, paroleMinime = 4)`	Lista delle frasi del testo.

6.3 ModelloConoscenza

Membro	Descrizione
`Vocabolario`	Dizionario radice → id.
`Radici`, `Forme`, `Frequenze`	Per ogni id: la radice, la forma più frequente, il numero di occorrenze.
`Idf`	Array di Single (`Float32Array` in JavaScript) con l'IDF di ogni parola.

Membro	Descrizione
`Embedding`	Per ogni id un vettore di `Dimensione` Single, oppure Nothing/null.
`Frasi`	Lista di oggetti `Frase`: `Testo`, `Termini` (id ordinati), `Pesi` (TF-IDF, norma 1), `Denso` (vettore denso o null).
`Dimensione`, `Finestra`, `FrequenzaMinima`, `Seme`	Iperparametri dell'addestramento (128, 4, 2, 12345).
`Origine`	Nome del file di testo da cui è stato addestrato il modello.
`Addestra(testo, log)`	Esegue gli otto passi del capitolo 4; `log` riceve una riga per passo.
`IdTermine(radice)`	Id della radice, oppure -1.
`Simili(id, k)`	Le k parole con embedding più vicino, come coppie (id, coseno) in ordine decrescente.
`Salva(percorso)` / `Carica(percorso)`	Scrittura e lettura del file `.mai` (capitolo 7). In JavaScript `salva()` restituisce un `Uint8Array` e `carica(dati)` riceve i byte del file.
`Prodotto(a, b)`, `Normalizza(v)`	Prodotto scalare (calcolato in Double) e normalizzazione a lunghezza 1.
`StimaMB()`	Memoria occupata stimata, in MB.

6.4 MotoreDomande

Si crea passando il modello al costruttore; i parametri sono campi pubblici che l'interfaccia aggiorna prima di ogni domanda (vedi capitolo 8).

```
' VB.NET
Dim motore As New MotoreDomande(modello)
motore.PesoSemantico = 0.5
Dim r As Risposta = motore.Rispondi("Dove vive la fata?")
Console.WriteLine(r.Testo)

// C#
var motore = new MotoreDomande(modello) { PesoSemantico = 0.5 };
Risposta r = motore.Rispondi("Dove vive la fata?");

// JavaScript
const motore = new MiniAI.MotoreDomande(modello);
motore.pesoSemantico = 0.5;
const r = motore.rispondi('Dove vive la fata?');
console.log(r.testo, r.punteggio);
```

6.5 L'interfaccia

La finestra ha tre schede: **Addestramento** (apertura del testo, parametri, log, salvataggio e caricamento del modello), **Domande** (domanda, parametri di risposta, risposta e traccia) ed **Esplora la memoria**. In quest'ultima si possono vedere le 25 parole più simili a una parola con i primi numeri del suo vettore, la somiglianza fra due parole e il vocabolario delle 150 parole più frequenti con radice, frequenza, IDF e presenza dell'embedding.

Nelle versioni .NET l'addestramento gira in un thread separato (`Task.Run`) e il log arriva all'interfaccia tramite `IProgress`, così la finestra resta reattiva. Nella versione JavaScript tutto avviene nel thread della pagina; il file `.mai` viene salvato come download del browser.



Figura 3 – Scheda Esplora la memoria: le parole con il vettore più vicino a «naso»

6.6 Note sul porting fra i tre linguaggi

Ottenere risultati identici al bit in tre linguaggi richiede alcune attenzioni, utili da conoscere se si modifica il codice:

- **Numeri casuali**: JavaScript non permette di fissare il seme di `Math.random`. `js/modello.js` contiene `RandomNet`, una riproduzione esatta dell'algoritmo di `System.Random` di .NET (generatore sottrattivo di Knuth), così i vettori casuali sono gli stessi.
- **Single e Double**: .NET usa `Single` (32 bit) per vettori e pesi e `Double` (64 bit) per somme e prodotti scalari. JavaScript calcola sempre a 64 bit; per questo i vettori sono `Float32Array` e ogni prodotto fra due valori Single viene arrotondato con `Math.fround`, come farebbe .NET.
- **Ordine dei dizionari**: il `Dictionary` di .NET, se non si cancellano elementi, restituisce le chiavi nell'ordine di inserimento; la `Map` di JavaScript fa lo stesso. Poiché l'ordine delle somme cambia gli ultimi decimali, il codice mantiene lo stesso ordine ovunque.
- **Ordinamenti**: la classifica usa un ordinamento stabile (`OrderByDescending` in .NET, `Array.prototype.sort` in JavaScript); `Simili` usa `List.Sort` in .NET e `sort` in JavaScript, con risultati uguali salvo coseni identici.
- **Formattazione**: i messaggi usano il formato italiano (2.459; 2,75; 0,207%). La versione JavaScript ha un piccolo oggetto `Formato` che lo riproduce senza dipendere dalle impostazioni del browser.
- **Compatibilità con il Framework 4.8**: la versione VB.NET è compatibile con Visual Basic 2012 e la C# con C# 5, i linguaggi dei compilatori inclusi in Windows, per poter compilare con `compila.bat` senza installare nulla.

7. Il formato del file .mai

Il modello addestrato si salva in un file binario con estensione `.mai`. Il formato è quello di `BinaryWriter` di .NET: interi a 32 bit e numeri Single in little-endian; le stringhe sono in UTF-8 precedute dalla lunghezza in byte, codificata in gruppi di 7 bit (il bit più alto di ogni byte indica che segue un altro byte). Un vettore assente si scrive come lunghezza 0.

Campo	Tipo	Note
Firma	stringa	Sempre `MINIAI01`.
Dimensione, Finestra, FrequenzaMinima, Seme	4 × Int32	Iperparametri usati nell'addestramento.
Origine	stringa	Nome del file di testo.
V	Int32	Numero di parole del vocabolario.
per ogni parola (V volte)		Radice (stringa), Forma (stringa), Frequenza (Int32), IDF (Single), Embedding (vettore).
N	Int32	Numero di frasi.
per ogni frase (N volte)		Testo (stringa), numero di termini n (Int32), poi n coppie id (Int32) + peso (Single), infine il vettore denso (vettore).
vettore	Int32 + Single…	Lunghezza seguita dai valori; lunghezza 0 = nessun vettore.

Con Pinocchio il file pesa circa 2,8 MB. Le matrici di co-occorrenza e PPMI non vengono salvate: servono solo durante l'addestramento. Per rispondere bastano vocabolario, IDF, embedding e vettori delle frasi.

8. Parametri ed esperimenti

8.1 Parametri di addestramento

Parametro	Predefinito	Effetto
Finestra contesto ±	4 (1–10)	Quante parole a destra e a sinistra contano come contesto. Finestra stretta: somiglianze più grammaticali; larga: più legate all'argomento.
Dimensione vettori	128 (16–512)	Lunghezza degli embedding. Più numeri = proiezione più fedele, ma più memoria e calcoli.
Frequenza minima	2 (1–20)	Sotto questa soglia la parola non riceve embedding. Valori alti: meno parole, vettori più affidabili.
Seme	12345	Seme del generatore casuale (solo da codice). Cambiandolo cambiano i vettori, non la qualità media.

8.2 Parametri di risposta

Parametro	Predefinito	Effetto
Peso semantico	0,30	0 = solo parole uguali (lessicale), 1 = solo somiglianza degli embedding.
Peso contesto	0,25	Quanto contano le due frasi vicine.
Parole simili per termine	3	Parole aggiunte alla domanda per ogni parola. 0 = nessuna espansione.
Soglia somiglianza	0,35	Somiglianza minima perché una parola simile venga aggiunta.
Regole tipo domanda	attive	Attiva o disattiva i bonus per chi/dove/quando/quanto/perché/come.
FrasiContesto, SogliaIncertezza	1; 0,12	Solo da codice: frasi mostrate attorno alla risposta e soglia sotto cui la risposta è dichiarata incerta.

8.3 Esperimenti guidati con Pinocchio

1. *Chi ha costruito il burattino?* con **Peso semantico** 0, poi 0,30, poi 1. Con 0 e 0,30 la risposta è «— Io sono un burattino.»; con 1 diventa «Nascono burattini, vivono burattini e muoiono burattini.»: conta solo l'argomento, non le parole esatte.
2. *Chi ha mangiato Geppetto?* La risposta è «— E chi te li ha mangiati?». Il modello è un «sacco di parole»: non distingue chi mangia da chi è mangiato e l'ordine delle parole non conta.
3. *Che cosa vende Geppetto per comprare l'abbecedario?* trova il titolo del capitolo VIII: «Geppetto rifà i piedi a Pinocchio, e vende la propria casacca per comprargli l'Abbecedario». L'espansione aggiunge «casacca» come parola simile a «comprato»: si vede nella traccia con il simbolo ~.
4. Impostare **Parole simili per termine** a 0 e ripetere la domanda precedente: senza «casacca» la frase giusta scende al quarto posto e la risposta diventa «— E perchè mi avete comprato?».
5. Nella scheda Esplora cercare `fata`, `geppetto`, `naso`. Molte parole simili sono solo parole vicine nel racconto («naso» → «giunti», «spropositato», «allungo»), non sinonimi: con un solo libro la statistica è povera.
6. Riaddestrare con **Finestra** 1 e poi 8 e confrontare le parole simili a `naso`. Con finestra 8 compaiono «spropositato», «allungo», «punta»: parole della stessa scena.

7. Scrivere *astronave*: il programma risponde che non conosce nessuna delle parole. Non sa nulla al di fuori del testo letto, e lo dichiara.
8. Disattivare **Regole tipo domanda** e ripetere *Chi ha costruito il burattino?*: senza il bonus la risposta diventa «Io sono un burattino.» (senza trattino). Le regole sono l'unica conoscenza scritta a mano; tutto il resto viene dai dati.
9. Addestrare con un altro testo, per esempio un manuale tecnico o una raccolta di procedure aziendali in formato `.txt`, e porre domande sul suo contenuto: il programma non dipende da Pinocchio.

9. Limiti ed estensioni

9.1 Limiti voluti

- Lo stemmer è rozzo e pensato solo per l'italiano; le parole vuote sono italiane.
- L'ordine delle parole è ignorato (modello «sacco di parole»): soggetto e oggetto non si distinguono.
- Le negazioni («non») sono parole vuote: «è vero» e «non è vero» risultano uguali.
- Gli embedding vengono da un solo testo, quindi riflettono le vicinanze di quel testo e non il significato generale delle parole.
- La ricerca confronta la domanda con tutte le frasi: va benissimo per migliaia di frasi, non per milioni.
- Le risposte sono sempre frasi esistenti: il programma non riassume e non combina informazioni prese da frasi diverse.

9.2 Dal giocattolo alle AI reali

Aspetto	MiniAI	Grandi modelli linguistici
Dati	Un libro: circa 20.000 parole di contenuto	Migliaia di miliardi di parole
Unità di testo	Radici di parole	Token (pezzi di parola) di un vocabolario fisso
Vettori delle parole	Co-occorrenze + PPMI + proiezione casuale, 128 numeri	Imparati da una rete neurale, migliaia di numeri
Contesto	Finestra fissa di ±4 parole	Meccanismo di attenzione su migliaia di token, ordine compreso
Numeri appresi	Circa 600.000 (embedding di parole e frasi)	Miliardi di parametri
Risposta	Sceglie una frase esistente	Genera testo nuovo, parola per parola
Addestramento	Meno di un secondo su un PC	Settimane su migliaia di processori specializzati

Le differenze sono enormi, ma la sostanza è comune: rappresentare il linguaggio con vettori appresi dal contesto e misurare somiglianze. Un sistema RAG moderno fa, a una scala molto più grande, ciò che MiniAI fa nei passi B–E: trasforma la domanda in vettore, cerca i brani più vicini e li passa al modello linguistico, che li usa per formulare la risposta.

9.3 Possibili estensioni

- **Generare frasi**: una catena di Markov sulle parole (o sui bigrammi) addestrata sullo stesso testo, per vedere il passo dalla scelta alla generazione.
- **Imparare i vettori**: sostituire la proiezione casuale con una piccola rete word2vec (skip-gram con campionamento negativo), sempre senza librerie esterne.
- **Più testo**: addestrare su più file insieme; è il miglioramento più efficace.
- **Bigrammi**: aggiungere al vocabolario le coppie di parole consecutive frequenti («fata turchina») per conservare un po' di ordine.
- **Negazioni**: unire «non» alla parola seguente («non_vero») invece di scartarlo.
- **Lingue diverse**: rendere configurabili parole vuote e suffissi.

- **Ricerca veloce**: un indice invertito (parola → frasi) per il punteggio lessicale e un indice approssimato per i vettori densi, se le frasi diventano centinaia di migliaia.

10. Licenza e responsabilità

MiniAI è distribuito gratuitamente, come tutti i programmi di DevService.net, e si usa a proprio rischio: è un progetto di studio, fornito così com'è, senza garanzie di alcun tipo. Il codice sorgente può essere letto, modificato e usato per imparare e sperimentare.

Il testo `Pinocchio.txt` (*Le avventure di Pinocchio*, Carlo Collodi, 1883) è di pubblico dominio e proviene da Project Gutenberg (www.gutenberg.org); la licenza Project Gutenberg è inclusa nel file stesso.

Per segnalazioni e suggerimenti è disponibile il modulo contatti del sito DevService.net.

MiniAI

Inteligență artificială didactică: de la text la vectori

Manualul programatorului

Surse în VB.NET, C# și JavaScript

Octombrie 2026

Cuprins

1. Introducere....70
1.1 Ce este MiniAI....70
1.2 Același principiu ca marile AI....70
1.3 Conținutul descărcării....70
2. Cerințe, compilare și pornire....72
2.1 Versiunea VB.NET....72
2.2 Versiunea C#....72
2.3 Versiunea JavaScript....72
2.4 Prima pornire cu Pinocchio....72
3. Concepte de bază....74
3.1 De la text la numere....74
3.2 Similaritatea cosinus....74
3.3 Ipoteza distribuțională....74
3.4 Regăsire și generare....74
4. Pipeline-ul de antrenare....75
4.1 Pasul 1 – Împărțirea în propoziții....75
4.2 Pasul 2 – Normalizare, rădăcini și vocabular....75
4.3 Pasul 3 – IDF....75
4.4 Pasul 4 – Matricea TF-IDF rară....76
4.5 Pasul 5 – Co-ocurențe....76
4.6 Pasul 6 – PPMI....76
4.7 Pasul 7 – Embedding prin proiecție aleatoare....77
4.8 Pasul 8 – Vectorii denși ai propozițiilor....77
4.9 Jurnalul antrenării....77
5. Răspunsul la o întrebare....78
5.1 A – Tipul întrebării....78
5.2 B – Cuvintele întrebării....78
5.3 C – Extinderea cu cuvinte similare....78
5.4 D – Scorul propozițiilor....78
5.5 E – Contextul....78
5.6 F – Răspunsul....79
5.7 Citirea urmei....79
6. Structura codului....81
6.1 Fișiere și corespondențe....81
6.2 ElaboratoreTesto....81
6.3 ModelloConoscenza....81
6.4 MotoreDomande....82
6.5 Interfața....82
6.6 Note despre portarea între cele trei limbaje....83
7. Formatul fișierului .mai....84
8. Parametri și experimente....85
8.1 Parametrii de antrenare....85
8.2 Parametrii de răspuns....85
8.3 Experimente ghidate cu Pinocchio....85
9. Limite și extensii....87
9.1 Limite intenționate....87
9.2 De la jucărie la AI-urile reale....87
9.3 Extensii posibile....87
10. Licență și răspundere....89

1. Introducere

1.1 Ce este MiniAI

MiniAI este un studiu de bază al meu despre inteligența artificială aplicată limbajului. Citește un text lung (o carte, o colecție de documente, un manual), îl transformă în numere, vectori și matrice și apoi răspunde la întrebări căutând în text propoziția cea mai apropiată de întrebare.

Nu este un produs de folosit în producție: este un laborator. Fiecare pas al calculului este vizibil, programul scrie în jurnal dimensiunile matricelor pe care le construiește și, pentru fiecare răspuns, afișează „urma raționamentului”: ce cuvinte a recunoscut, ce cuvinte a adăugat, ce propoziții a comparat și cu ce scoruri. Scopul este să înțelegeți, nu să fiți impresionat.

Programul este distribuit ca **cod sursă**, în trei versiuni echivalente: **VB.NET**, **C#** și **JavaScript**. Nu folosește biblioteci externe: tot ce este necesar (de la împărțirea în propoziții la proiecția aleatoare a embedding-urilor) este scris integral în câteva sute de linii comentate.

1.2 Același principiu ca marile AI

Evident, cantitatea de date face diferența. AI-urile de pe piață sunt antrenate pe miliarde de cuvinte, au miliarde de parametri și rețele neuronale care învață să genereze text nou. MiniAI citește o singură carte și învață câteva mii de vectori în mai puțin de o secundă. Principiul de bază este însă același:

- **textul devine numere**: fiecare cuvânt și fiecare propoziție este reprezentat printr-un vector;
- **sensul se naște din context**: două cuvinte folosite în contexte similare primesc vectori similari (ipoteza distribuțională);
- **a înțelege o întrebare înseamnă a măsura similarități**: întrebarea este transformată în același spațiu vectorial și comparată cu ceea ce sistemul cunoaște;
- **a căuta înainte de a răspunde**: regăsirea propozițiilor relevante (*retrieval*) este același mecanism folosit de sistemele RAG (*Retrieval-Augmented Generation*), care furnizează unui model lingvistic documentele din care să extragă răspunsul.

MiniAI se oprește la regăsire: **nu generează text, ci alege o propoziție existentă**. Tocmai de aceea este un bun punct de plecare: se vede clar ce reușește statistica să facă singură și unde sunt necesare mai multe date sau modele mai puternice.

> Cu același cod, mai mult text dă rezultate mai bune. Cu un singur roman, „cuvintele similare” sunt adesea doar cuvinte apropiate în povestire; cu zeci de cărți pe același subiect încep să apară sinonime adevărate. Este, la scară mică, motivul pentru care marile AI au nevoie de cantități enorme de date.

1.3 Conținutul descărcării

Site-ul oferă trei fișiere zip, câte unul pentru fiecare limbaj. Fiecare conține programul complet, textul de exemplu și un fișier `LEGGIMI.md` în limba italiană.

Fișier	Conținut
`MiniAI_VBNET.zip`	Proiect Windows Forms pentru .NET Framework 4.8 în VB.NET: `MiniAI.sln`, `MiniAI.vbproj`, `compila.bat` și fișierele `Testo.vb`, `Modello.vb`, `Motore.vb`, `FormPrincipale.vb`, `Program.vb`.
`MiniAI_CSharp.zip`	Același proiect în C#: `MiniAI.sln`, `MiniAI.csproj`, `compila.bat` și fișierele `.cs` cu același nume.
`MiniAI_JavaScript.zip`	Pagină web autonomă (`index.html`, `stile.css`, folderul `js`) și `prova-node.js` pentru utilizarea din linia de comandă cu Node.js.

Fișier	Conținut
`Pinocchio.txt`	În toate cele trei arhive zip: *Le avventure di Pinocchio* de Carlo Collodi (1883), text din domeniul public de pe Project Gutenberg, folosit pentru toate exemplele din acest manual.

Cele trei versiuni execută aceleași calcule în aceeași ordine. Cu același text și aceiași parametri produc aceleași răspunsuri, aceeași urmă și același fișier de model `.mai`, identic octet cu octet: un model salvat de o versiune se deschide cu celelalte.

2. Cerințe, compilare și pornire

2.1 Versiunea VB.NET

- Windows 10/11 sau Windows Server 2016 și ulterioare, cu .NET Framework 4.8 (deja prezent în Windows 10 începând cu versiunea 1903, în Windows 11 și în Windows Server 2022 și ulterioare).
- **Cu Visual Studio** (2019 sau 2022, sarcina de lucru „Dezvoltare desktop .NET”): deschideți `MiniAI.sln` și apăsați F5.
- **Fără Visual Studio**: faceți dublu clic pe `compila.bat`. Fișierul folosește compilatorul `vbc.exe` pe care .NET Framework 4.8 îl instalează în `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319`. Codul este scris astfel încât să fie compatibil cu acel compilator (Visual Basic 2012), iar `MiniAI.exe` este creat în același folder.

Fișierele `.vb` sunt salvate în UTF-8 cu BOM: în acest fel și compilatorul din Framework citește corect literele accentuate din comentarii și din mesaje.

2.2 Versiunea C#

- Aceleași cerințe ca pentru versiunea VB.NET.
- **Cu Visual Studio**: deschideți `MiniAI.sln` (proiectul `MiniAI.csproj`) și apăsați F5.
- **Fără Visual Studio**: faceți dublu clic pe `compila.bat`, care folosește `csc.exe` din .NET Framework 4.8. Codul este compatibil cu C# 5, versiunea acelui compilator: fără interpolare de șiruri și fără alte sintaxe mai recente.

2.3 Versiunea JavaScript

- **În browser**: faceți dublu clic pe `index.html`. Funcționează cu versiuni recente de Edge, Chrome și Firefox, chiar și fără conexiune la Internet și fără server: textul nu părăsește niciodată computerul.
- **Cu Node.js** (versiunea 18 sau ulterioară), din folderul programului: `node prova-node.js Pinocchio.txt` antrenează modelul și deschide o sesiune de întrebări; `node prova-node.js Pinocchio.txt domande.txt modello.mai` răspunde la fiecare linie din `domande.txt` și salvează modelul; `node prova-node.js modello.mai` pornește de la un model salvat. Opțiunea `--traccia` afișează și urma raționamentului.

În browser, antrenarea blochează pagina pe durata calculului (mai puțin de o secundă cu Pinocchio); cu texte foarte lungi, pagina poate părea blocată câteva secunde.

2.4 Prima pornire cu Pinocchio

1. Fila **1 · Addestramento** (Antrenare): „Apri testo (.txt)...” (Deschidere text) și alegeți `Pinocchio.txt` (în versiunea JavaScript puteți și să trageți fișierul pe pagină).
2. Lăsați parametrii impliciți (fereastra 4, dimensiunea 128, frecvența minimă 2) și apăsați „Addestra” (Antrenează). Jurnalul enumeră cei opt pași descriși în capitolul 4.
3. Fila **2 · Domande** (Întrebări): scrieți *Chi ha costruito il burattino?* (Cine a construit marioneta?) și apăsați Enter. Deasupra apare răspunsul cu propozițiile vecine, dedesubt urma completă.
4. Fila **3 · Esplora la memoria** (Explorați memoria): scrieți `naso` (nas) și apăsați „Parole simili” (Cuvinte similare).
5. Opțional: „Salva modello...” (Salvare model) creează un fișier `.mai` care se reîncarcă într-o clipă cu „Carica modello...” (Încărcare model).

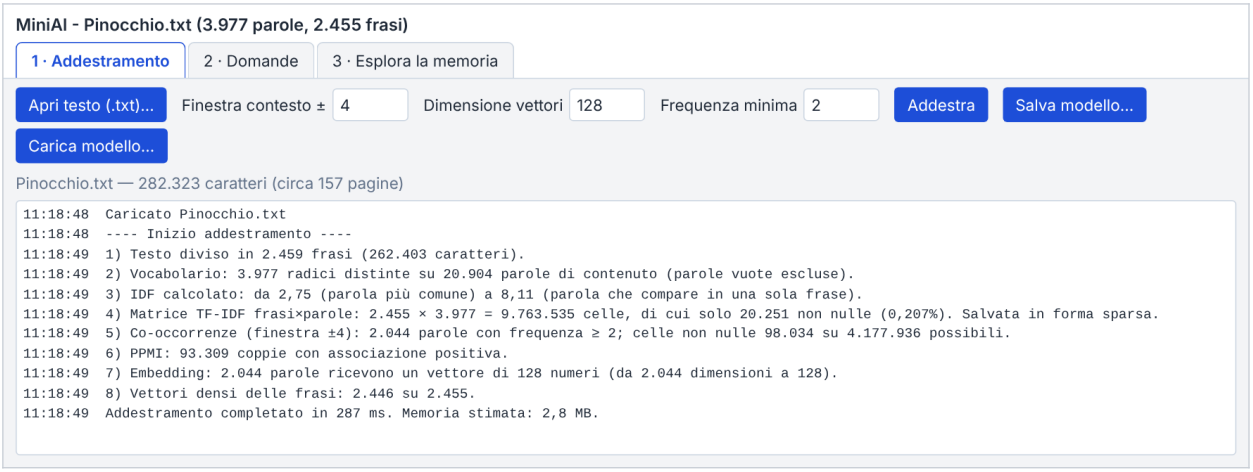

Figura 1 – Fila Addestramento după antrenarea pe Pinocchio (versiunea JavaScript; versiunile Windows au aceleași trei file)

Textul este citit ca UTF-8; dacă conține octeți nevalizi în UTF-8, este recitit cu codificarea Windows (Windows-1252), astfel încât funcționează și fișierele `.txt` vechi salvate cu Notepad.

3. Concepte de bază

3.1 De la text la numere

Un computer nu știe ce este un „burattino” (marionetă). Știe însă să numere și să facă operații pe liste de numere. Toată munca lui MiniAI constă în a transforma cuvinte și propoziții în **vectori**, adică liste de numere de lungime fixă, astfel încât texte cu sens similar să dea vectori similari.

În program apar două tipuri de vectori:

- **vectori rari**: lungi cât vocabularul (circa 4.000 de poziții cu Pinocchio), aproape numai zerouri; fiecare poziție corespunde unui cuvânt. Se memorează doar pozițiile diferite de zero;
- **vectori denși** (*embedding*): scurți (128 de numere), toți semnificativi; nicio poziție nu corespunde unui cuvânt anume, ansamblul numerelor descrie sensul.

3.2 Similaritatea cosinus

Pentru a spune cât de similari sunt doi vectori, MiniAI folosește **cosinusul unghiului** dintre ei:

```
cosinus(a, b) = (a · b) / (|a| · |b|)

a · b = a1·b1 + a2·b2 + ... + an·bn      (produs scalar)
|a|   = rădăcina pătrată din (a · a)      (lungimea vectorului)
```

Rezultatul este 1 dacă vectorii au aceeași direcție, 0 dacă nu au nimic în comun și valori negative dacă sunt opuși. Contează direcția, nu lungimea: o propoziție lungă și una scurtă pe același subiect rezultă similare.

Pentru a economisi calcule, toți vectorii sunt **normalizați** (aduși la lungimea 1) imediat după creare. Din acel moment cosinusul coincide cu simplul produs scalar: este ceea ce fac funcțiile `Normalizza` și `Prodotto` ale clasei `ModelloConoscenza`.

3.3 Ipoteza distribuțională

„Vei cunoaște un cuvânt după compania pe care o frecventează” (J. R. Firth, 1957). Dacă două cuvinte apar des lângă aceleași alte cuvinte, probabil au un sens înrudit. MiniAI aplică această idee numărând, pentru fiecare cuvânt, ce cuvinte se află lângă el în text (matricea de co-ocurență) și transformând aceste numărători într-un vector dens. Pe același principiu se bazează word2vec, GloVe și straturile de embedding ale modelelor lingvistice moderne.

3.4 Regăsire și generare

Există două moduri de a răspunde la o întrebare pornind de la un text:

- **extractiv** (regăsire): se caută în text pasajul cel mai relevant și se returnează așa cum este. Este ceea ce face MiniAI și este prima etapă a sistemelor RAG;
- **generativ**: un model lingvistic produce o propoziție nouă, cuvânt cu cuvânt, alegând de fiecare dată cuvântul cel mai probabil. Necesită rețele neuronale antrenate pe cantități enorme de text.

Regăsirea are un avantaj important: răspunsul este întotdeauna o propoziție reală din document, verificabilă, iar sistemul poate spune cinstit „nu cunosc niciunul dintre aceste cuvinte”.

4. Pipeline-ul de antrenare

Antrenarea este efectuată de metoda `Addestra` a clasei `ModelloConoscenza` (fișierul `Modello.vb`, `Modello.cs` sau `js/modello.js`). Se compune din opt pași; fiecare scrie o linie în jurnal. Numerele indicate în acest capitol sunt cele obținute cu `Pinocchio.txt` și parametrii impliciți.

4.1 Pasul 1 – Împărțirea în propoziții

Dacă fișierul provine de pe Project Gutenberg, `TogliIntestazioneGutenberg` elimină licența de la început și de la sfârșit (caută liniile `*** START OF` și `*** END OF`). Apoi `DividiInFrasi`:

- unește liniile întrerupte de trecerile „fixe” la rând nou, tipice fișierelor `.txt`; paragrafele sunt separate de o linie goală;
- împarte fiecare paragraf acolo unde există un semn de sfârșit de propoziție (. ! ? …), eventual urmat de ghilimele, apoi un spațiu și o literă mare, o cifră sau o liniuță de dialog;
- elimină propozițiile cu mai puțin de 4 cuvinte.

Propoziția este **unitatea de răspuns**: MiniAI răspunde întotdeauna cu o propoziție întreagă. Cu Pinocchio se obțin 2.459 de propoziții din 262.403 caractere.

4.2 Pasul 2 – Normalizare, rădăcini și vocabular

Fiecare propoziție este redusă la **cuvintele de conținut** de către funcția `ParoleContenuto`:

1. `Normalizza`: totul cu litere mici, accentele eliminate (descompunere Unicode NFD și eliminarea semnelor diacritice), orice caracter care nu este literă sau cifră devine spațiu. „Perché l'albero?” devine `perche l albero`.
2. Cuvintele **vide** (articole, prepoziții, auxiliare, pronume: circa 200 de cuvinte enumerate în `Testo`) și cuvintele dintr-o singură literă sunt eliminate: apar peste tot și nu spun nimic despre conținut.
3. `Radice`: un stemmer intenționat simplu elimină o terminație cunoscută (`-issimo`, `-mente`, `-zione`, `-avano`, `-ando`, `-ato`, `-ito`…) sau vocala finală. „burattino”, „burattini” și „burattina” devin toate `burattin`; „andato” și „andavano” devin `and`.

Fiecare rădăcină nouă primește un număr progresiv (**id**-ul său) în dicționarul `Vocabolario`. Pentru fiecare rădăcină se numără aparițiile (`Frequenze`) și se reține forma cea mai frecventă (`Forme`), folosită doar pentru a afișa cuvântul într-un mod lizibil. Cu Pinocchio: **3.977 de rădăcini** din 20.904 cuvinte de conținut.

> Stemmerul greșește des și este instructiv să vedeți unde: „costruito” (construit) nu apare niciodată în Pinocchio, dar rădăcina sa `costru` coincide cu cea a lui „costruì”, care apare o dată. Cuvântul din întrebare este deci recunoscut datorită altei forme a aceluiași verb.

4.3 Pasul 3 – IDF

Nu toate cuvintele sunt la fel de informative: „pinocchio” apare în sute de propoziții, „abbecedario” (abecedar) în foarte puține. **IDF** (*inverse document frequency*) măsoară raritatea unui cuvânt:

```
idf(t) = ln( (N + 1) / (df(t) + 1) ) + 1

N     = numărul de propoziții
df(t) = numărul de propoziții care conțin cuvântul t
```

+1 de la numărător și de la numitor evită împărțirile la zero; +1 final garantează că și cuvântul cel mai comun are o pondere pozitivă. Cu Pinocchio, IDF variază de la 2,75 (cuvântul cel mai comun, „pinocchio”) la 8,11 (cuvinte care apar într-o singură propoziție). Iată aceeași linie în cele trei versiuni:

```
' VB.NET
Idf(i) = CSng(Math.Log((N + 1.0) / (df(i) + 1.0)) + 1.0)

// C#
Idf[i] = (float)(Math.Log((N + 1.0) / (df[i] + 1.0)) + 1.0);

// JavaScript (Float32Array: conversia la Single este automată)
this.idf[i] = Math.log((N + 1.0) / (df[i] + 1.0)) + 1.0;
```

4.4 Pasul 4 – Matricea TF-IDF rară

Fiecare propoziție devine un vector lung cât vocabularul. Ponderea cuvântului t în propoziția f este:

```
pondere(t, f) = (1 + ln tf(t, f)) · idf(t)        tf = aparițiile lui t în propoziție

apoi vectorul este normalizat la lungimea 1
```

Logaritmul atenuează repetițiile: un cuvânt repetat de trei ori nu valorează triplu. Matricea completă propoziții × cuvinte ar avea 2.455 × 3.977 = 9.763.535 de celule, dar doar 20.251 sunt diferite de zero (0,207%). De aceea fiecare propoziție memorează doar două array-uri paralele: `Termini` (id-urile cuvintelor prezente, în ordine crescătoare) și `Pesi` (ponderile respective). Este **reprezentarea rară**, aceeași folosită de motoarele de căutare.

Propozițiile formate doar din cuvinte vide nu produc un vector și sunt eliminate: de aceea propozițiile memorate sunt 2.455 și nu 2.459.

4.5 Pasul 5 – Co-ocurențe

Aici începe partea „semantică”. Se iau în considerare doar cuvintele cu frecvența cel puțin egală cu `FrequenzaMinima` (2): cuvintele văzute o singură dată nu au suficiente contexte pentru un vector fiabil. Cu Pinocchio sunt 2.044.

Cuvintele de conținut din toate propozițiile sunt puse în șir într-un singur flux. Pentru fiecare cuvânt a și pentru fiecare cuvânt b care îl urmează în cel mult `Finestra` poziții (4) se adaugă la ambele celule (a, b) și (b, a) o pondere care descrește cu distanța:

```
cooc(a, b) += 1 / distanță        distanță = 1, 2, ..., Finestra
```

Matricea teoretică 2.044 × 2.044 are 4.177.936 de celule, dar cele nenule sunt 98.034: și aceasta este memorată în formă rară, într-un dicționar a cărui cheie este `a · V + b` (un întreg pe 64 de biți).

4.6 Pasul 6 – PPMI

Numărarea brută favorizează cuvintele frecvente: „disse” (spuse) se află lângă aproape orice. **PMI** (*pointwise mutual information*) compară de câte ori două cuvinte stau unul lângă altul cu de câte ori ne-am aștepta din pură întâmplare:

```
PMI(a, b)  = ln( P(a, b) / ( P(a) · Pc(b) ) )

P(a, b) = cooc(a, b) / total
P(a)    = suma rândului a / total
Pc(b)   = (suma rândului b)^0,75 / suma tuturor (sumelor de rând)^0,75

PPMI(a, b) = max(0, PMI(a, b))
```

Se păstrează doar valorile pozitive (**PPMI**, *positive* PMI): perechile care stau împreună mai des decât ar face-o întâmplarea. Exponentul 0,75 aplicat probabilității contextului este același artificiu din word2vec și reduce ponderea excesivă a contextelor rare. Cu Pinocchio: 93.309 perechi cu asociere pozitivă.

4.7 Pasul 7 – Embedding prin proiecție aleatoare

Fiecare rând al matricei PPMI este deja un vector care descrie un cuvânt, dar are lungimea de 2.044 de poziții. Pentru a obține vectori scurți, MiniAI folosește o **proiecție aleatoare**:

1. fiecărui cuvânt-context b i se atribuie un vector aleator fix r(b) de `Dimensione` numere (128), extrase dintr-o distribuție normală cu abaterea standard 1/√D (transformarea Box-Muller);
2. embedding-ul cuvântului a este suma vectorilor contextelor sale, ponderată cu PPMI: e(a) = Σ PPMI(a, b) · r(b);
3. vectorul este normalizat la lungimea 1.

Lema Johnson-Lindenstrauss garantează că o proiecție aleatoare păstrează aproximativ distanțele și cosinusurile dintre vectorii originali: două cuvinte cu rânduri PPMI similare au și embedding-uri similare. Este o tehnică cunoscută ca *random indexing*; modelele moderne obțin același rezultat învățând vectorii cu o rețea neuronală.

Generatorul aleator pornește întotdeauna de la aceeași `Seme` (sămânță, 12345), deci antrenarea este **reproductibilă**: același text și aceiași parametri dau întotdeauna aceiași vectori. Cu Pinocchio, 2.044 de cuvinte primesc un embedding; cuvintele prea rare rămân fără (valoarea `Nothing`/`null`).

4.8 Pasul 8 – Vectorii denși ai propozițiilor

Vectorul dens al unei propoziții este media embedding-urilor cuvintelor sale, ponderată cu ponderea TF-IDF calculată la pasul 4, apoi normalizată. Cuvintele rare și specifice contează mai mult decât cele comune. Cu Pinocchio, 2.446 de propoziții din 2.455 au un vector dens; celelalte conțin doar cuvinte fără embedding.

4.9 Jurnalul antrenării

Acesta este jurnalul complet produs cu Pinocchio și parametrii impliciți; cele trei versiuni scriu aceleași linii:

```
1) Testo diviso in 2.459 frasi (262.403 caratteri).
2) Vocabolario: 3.977 radici distinte su 20.904 parole di contenuto
   (parole vuote escluse).
3) IDF calcolato: da 2,75 (parola più comune) a 8,11 (parola che compare
   in una sola frase).
4) Matrice TF-IDF frasi×parole: 2.455 × 3.977 = 9.763.535 celle, di cui
   solo 20.251 non nulle (0,207%). Salvata in forma sparsa.
5) Co-occorrenze (finestra ±4): 2.044 parole con frequenza ≥ 2; celle non
   nulle 98.034 su 4.177.936 possibili.
6) PPMI: 93.309 coppie con associazione positiva.
7) Embedding: 2.044 parole ricevono un vettore di 128 numeri (da 2.044
   dimensioni a 128).
8) Vettori densi delle frasi: 2.446 su 2.455.
```

Antrenarea durează de la câteva zecimi de secundă la o secundă, în funcție de computer; modelul ocupă circa 2,8 MB în memorie și pe disc.

5. Răspunsul la o întrebare

Răspunsul este produs de metoda `Rispondi` a clasei `MotoreDomande` (fișierul `Motore.vb`, `Motore.cs` sau `js/motore.js`). Metoda returnează un obiect `Risposta` cu trei câmpuri: textul răspunsului, urma și scorul propoziției alese. Pașii sunt indicați cu literele de la A la F, aceleași care apar în urmă.

5.1 A – Tipul întrebării

Dacă printre primele trei cuvinte ale întrebării se află *chi*, *dove*, *quando*, *quanto* (sau *quanti*, *quante*, *quanta*), *perché* sau *come* (cine, unde, când, cât, de ce, cum), tipul întrebării este recunoscut. Este singura parte a programului scrisă „de mână” cu reguli fixe și servește la compararea cunoașterii programate cu cea învățată din date.

5.2 B – Cuvintele întrebării

Întrebarea suferă aceleași transformări ca textul: normalizare, eliminarea cuvintelor vide, reducere la rădăcină. Fiecare rădăcină este căutată în vocabular. Cuvintele nevăzute niciodată sunt enumerate în urmă ca „MAI viste nel testo” (niciodată văzute în text): modelul nu știe nimic în afara textului citit. Dacă niciun cuvânt nu este cunoscut, răspunsul este *Non conosco nessuna delle parole della domanda* (Nu cunosc niciunul dintre cuvintele întrebării).

Cuvintelor recunoscute li se atribuie ponderea TF-IDF, ca pentru propoziții: (1 + ln tf) · idf.

5.3 C – Extinderea cu cuvinte similare

Pentru fiecare cuvânt al întrebării se caută cele `Espansione` (3) cuvinte cu embedding-ul cel mai similar. Cele cu similaritatea cel puțin egală cu `SogliaEspansione` (0,35) sunt adăugate la întrebare cu pondere redusă:

```
pondere(cuvânt adăugat) = idf · similaritate · 0,5
```

Astfel o întrebare despre „vendere” (a vinde) poate găsi o propoziție care vorbește despre „mercato” (piață). În urmă, cuvintele adăugate sunt indicate cu simbolul ~. Vectorul întrebării este apoi normalizat.

Se calculează și **vectorul dens al întrebării**: suma embedding-urilor cuvintelor originale (nu a celor adăugate), ponderate cu IDF, normalizată.

5.4 D – Scorul propozițiilor

Fiecare propoziție memorată primește trei scoruri:

- **lexical**: cosinusul dintre vectorul rar al întrebării și cel al propoziției. Se adună produsele doar pe cuvintele comune, deci calculul este foarte rapid;
- **semantic**: cosinusul dintre vectorul dens al întrebării și cel al propoziției (valorile negative devin 0). Favorizează propozițiile care vorbesc despre același subiect chiar și cu alte cuvinte;
- **bonus**: o mică primă (0,02–0,03) dacă propoziția conține indicii potrivite tipului de întrebare: un nume propriu pentru *chi* (în practică un cuvânt cu inițială mare care nu este primul din propoziție), un cuvânt de timp pentru *quando*, un loc pentru *dove*, o cauză pentru *perché*, un număr pentru *quanto*, un mod pentru *come*. Bonusul se aplică doar propozițiilor care au cel puțin un cuvânt în comun cu întrebarea.

```
brut = (1 - ps) · lexical + ps · semantic + bonus        ps = PesoSemantico (0,30)
```

5.5 E – Contextul

Într-o povestire, o propoziție relevantă este adesea înconjurată de propoziții relevante. Scorul final adaugă o parte din media scorurilor celor două propoziții vecine:

```
total(i) = brut(i) + pc · (brut(i-1) + brut(i+1)) / 2     pc = PesoContesto (0,25)
```

Propozițiile sunt ordonate după scorul total descrescător (sortare stabilă: la scor egal câștigă propoziția care vine mai întâi în text), iar urma afișează primele cinci.

5.6 F – Răspunsul

Propoziția cu scorul cel mai mare este răspunsul. Dacă scorul este mai mic decât `SogliaIncertezza` (0,12), răspunsul este precedat de *(Risposta incerta: punteggio basso)* (Răspuns incert: scor scăzut); dacă este zero, programul răspunde *Non ho trovato nulla di pertinente* (Nu am găsit nimic relevant). Sub răspuns sunt afișate `FrasiContesto` propoziții înainte și după (1), pentru a-l citi în contextul său.

5.7 Citirea urmei

Aceasta este urma întrebării *Chi ha costruito il burattino?* cu parametrii impliciți:

```
A) TIPO DI DOMANDA: CHI
B) PAROLE DELLA DOMANDA
   ignorate (parole vuote): chi, ha, il
   costrui            radice 'costru'  frequenza     1  IDF 8,11  (senza embedding)
   burattino          radice 'burattin'  frequenza   226  IDF 3,40
C) ESPANSIONE CON PAROLE SIMILI (embedding)
   costrui            -> nessuna sopra soglia
   burattino          -> guardare (0,35)
D) CLASSIFICA (2.455 frasi confrontate)   peso semantico 0,30, peso contesto 0,25
   pos  frase   lessic.  semant.  bonus   TOTALE   parole in comune
     1   1026   0,385    1,000    0,030   0,607    burattino
             — Io sono un burattino.
     2   2140   0,385    1,000    0,000   0,588    burattino
             Io sono un burattino.
     3   1245   0,174    1,000    0,000   0,500    burattino
             Nascono burattini, vivono burattini e muoiono burattini.
```

Se observă mai multe lucruri instructive. „costruito” este recunoscut doar prin rădăcina lui „costruì”, care apare o dată și deci nu are embedding. Scorul semantic este 1,000 pentru propozițiile care conțin doar cuvântul „burattino”: vectorul lor dens coincide cu cel al întrebării. Prima și a doua propoziție au același scor lexical; prima câștigă doar datorită bonusului *chi*: după liniuța de dialog, cuvântul „Io” (Eu) are inițială mare, iar regula îl confundă cu un nume propriu. O regulă scrisă de mână este ușor de înțeles, dar la fel de ușor de păcălit. Răspunsul este greșit din punct de vedere uman (răspunsul corect este „Geppetto”), dar este exact ceea ce sugerează statistica acestui text.

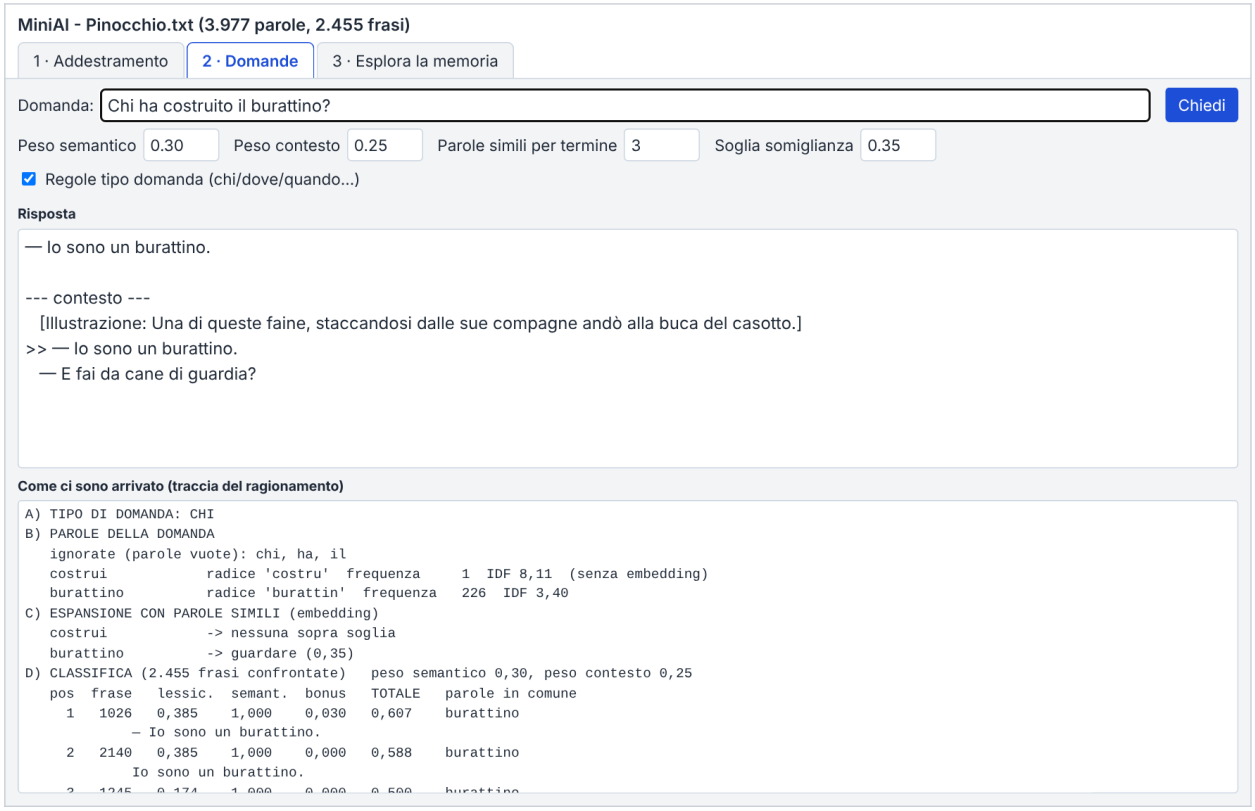

Figura 2 – Fila Domande: răspunsul cu contextul său și, dedesubt, urma raționamentului

6. Structura codului

6.1 Fișiere și corespondențe

Numele claselor, metodelor și variabilelor sunt în italiană și sunt aceleași în cele trei versiuni. În JavaScript urmează convenția limbajului (inițială mică: `rispondi` în loc de `Rispondi`).

VB.NET	C#	JavaScript	Conținut
`Testo.vb`	`Testo.cs`	`js/testo.js`	Clasa `ElaboratoreTesto`: propoziții, normalizare, cuvinte vide, rădăcini.
`Modello.vb`	`Modello.cs`	`js/modello.js`	Clasele `Frase` și `ModelloConoscenza`: antrenare, cuvinte similare, salvare și încărcare.
`Motore.vb`	`Motore.cs`	`js/motore.js`	Clasele `Risposta` și `MotoreDomande`: răspunsul la întrebări și urma.
`FormPrincipale.vb`	`FormPrincipale.cs`	`index.html`, `js/app.js`, `stile.css`	Interfață cu trei file. Versiunile .NET construiesc controalele din cod, fără fișiere Designer.
`Program.vb`	`Program.cs`	`prova-node.js`	Punct de pornire. În JavaScript, `prova-node.js` este alternativa din linia de comandă.

6.2 ElaboratoreTesto

Conține doar funcții statice (`Shared` în VB.NET, `static` în C#, un obiect literal în JavaScript).

Membru	Descriere
`EParolaVuota(parola)`	True dacă cuvântul (deja normalizat) este un cuvânt vid.
`TogliIntestazioneGutenberg(testo)`	Elimină licența de la început și de la sfârșit de la Project Gutenberg, dacă există.
`Normalizza(testo)`	Litere mici, fără accente, caracterele non-alfanumerice înlocuite cu spații.
`Parole(testo)`	Array-ul cuvintelor normalizate.
`Radice(parola)`	Rădăcina cuvântului (stemmer simplificat).
`ParoleContenuto(testo)`	Perechi (rădăcină, formă) ale cuvintelor de conținut.
`DividiInFrasi(testo, paroleMinime = 4)`	Lista propozițiilor din text.

6.3 ModelloConoscenza

Membru	Descriere
`Vocabolario`	Dicționar rădăcină → id.
`Radici`, `Forme`, `Frequenze`	Pentru fiecare id: rădăcina, forma cea mai frecventă, numărul de apariții.
`Idf`	Array de Single (`Float32Array` în JavaScript) cu IDF-ul fiecărui cuvânt.

Membru	Descriere
`Embedding`	Pentru fiecare id, un vector de `Dimensione` valori Single sau Nothing/null.
`Frasi`	Listă de obiecte `Frase`: `Testo`, `Termini` (id-uri ordonate), `Pesi` (TF-IDF, normă 1), `Denso` (vector dens sau null).
`Dimensione`, `Finestra`, `FrequenzaMinima`, `Seme`	Hiperparametrii antrenării (128, 4, 2, 12345).
`Origine`	Numele fișierului text pe care a fost antrenat modelul.
`Addestra(testo, log)`	Execută cei opt pași din capitolul 4; `log` primește o linie pentru fiecare pas.
`IdTermine(radice)`	Id-ul rădăcinii sau -1.
`Simili(id, k)`	Cele k cuvinte cu embedding-ul cel mai apropiat, ca perechi (id, cosinus) în ordine descrescătoare.
`Salva(percorso)` / `Carica(percorso)`	Scrierea și citirea fișierului `.mai` (capitolul 7). În JavaScript, `salva()` returnează un `Uint8Array`, iar `carica(dati)` primește octeții fișierului.
`Prodotto(a, b)`, `Normalizza(v)`	Produs scalar (calculat în Double) și normalizare la lungimea 1.
`StimaMB()`	Memoria ocupată estimată, în MB.

6.4 MotoreDomande

Se creează transmițând modelul constructorului; parametrii sunt câmpuri publice pe care interfața le actualizează înainte de fiecare întrebare (vedeți capitolul 8).

```
' VB.NET
Dim motore As New MotoreDomande(modello)
motore.PesoSemantico = 0.5
Dim r As Risposta = motore.Rispondi("Dove vive la fata?")
Console.WriteLine(r.Testo)

// C#
var motore = new MotoreDomande(modello) { PesoSemantico = 0.5 };
Risposta r = motore.Rispondi("Dove vive la fata?");

// JavaScript
const motore = new MiniAI.MotoreDomande(modello);
motore.pesoSemantico = 0.5;
const r = motore.rispondi('Dove vive la fata?');
console.log(r.testo, r.punteggio);
```

6.5 Interfața

Fereastra are trei file: **Addestramento** (deschiderea textului, parametri, jurnal, salvarea și încărcarea modelului), **Domande** (întrebare, parametri de răspuns, răspuns și urmă) și **Esplora la memoria**. În aceasta din urmă puteți vedea cele 25 de cuvinte cele mai similare cu un cuvânt, împreună cu primele numere ale vectorului său, similaritatea dintre două cuvinte și vocabularul celor mai frecvente 150 de cuvinte, cu rădăcină, frecvență, IDF și prezența embedding-ului.

În versiunile .NET, antrenarea rulează într-un fir de execuție separat (`Task.Run`), iar jurnalul ajunge la interfață prin `IProgress`, astfel încât fereastra rămâne receptivă. În versiunea JavaScript totul se petrece în firul de execuție al paginii; fișierul `.mai` este salvat ca descărcare a browserului.

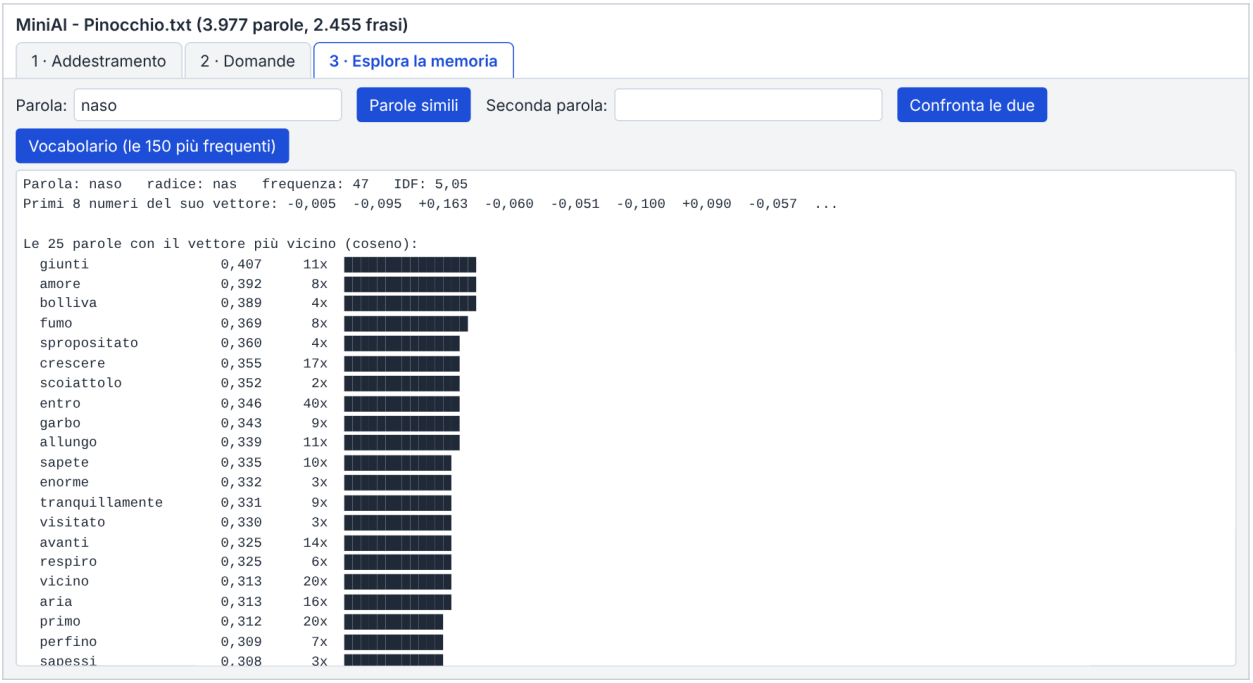

Figura 3 – Fila Esplora la memoria: cuvintele cu vectorul cel mai apropiat de „naso”

6.6 Note despre portarea între cele trei limbaje

Obținerea unor rezultate identice la nivel de bit în trei limbaje necesită câteva precauții, utile de cunoscut dacă modificați codul:

- **Numere aleatoare**: JavaScript nu permite fixarea seminței pentru `Math.random`. `js/modello.js` conține `RandomNet`, o reproducere exactă a algoritmului `System.Random` din .NET (generatorul subtractiv al lui Knuth), astfel încât vectorii aleatori sunt aceiași.
- **Single și Double**: .NET folosește `Single` (32 de biți) pentru vectori și ponderi și `Double` (64 de biți) pentru sume și produse scalare. JavaScript calculează întotdeauna pe 64 de biți; de aceea vectorii sunt `Float32Array`, iar fiecare produs între două valori Single este rotunjit cu `Math.fround`, așa cum ar face .NET.
- **Ordinea dicționarelor**: `Dictionary` din .NET, dacă nu se șterg elemente, returnează cheile în ordinea inserării; `Map` din JavaScript face la fel. Deoarece ordinea adunărilor modifică ultimele zecimale, codul păstrează aceeași ordine peste tot.
- **Sortări**: clasamentul folosește o sortare stabilă (`OrderByDescending` în .NET, `Array.prototype.sort` în JavaScript); `Simili` folosește `List.Sort` în .NET și `sort` în JavaScript, cu rezultate egale, cu excepția cosinusurilor identice.
- **Formatare**: mesajele folosesc formatul italian (2.459; 2,75; 0,207%). Versiunea JavaScript are un mic obiect `Formato` care îl reproduce fără a depinde de setările browserului.
- **Compatibilitatea cu Framework 4.8**: versiunea VB.NET este compatibilă cu Visual Basic 2012, iar cea C# cu C# 5, limbajele compilatoarelor incluse în Windows, pentru a putea compila cu `compila.bat` fără a instala nimic.

7. Formatul fișierului .mai

Modelul antrenat se salvează într-un fișier binar cu extensia `.mai`. Formatul este cel al clasei `BinaryWriter` din .NET: întregi pe 32 de biți și numere Single în little-endian; șirurile sunt în UTF-8, precedate de lungimea în octeți, codificată în grupuri de 7 biți (bitul cel mai semnificativ al fiecărui octet indică faptul că urmează un alt octet). Un vector absent se scrie ca lungime 0.

Câmp	Tip	Note
Semnătură	șir	Întotdeauna `MINIAI01`.
Dimensione, Finestra, FrequenzaMinima, Seme	4 × Int32	Hiperparametrii folosiți la antrenare.
Origine	șir	Numele fișierului text.
V	Int32	Numărul de cuvinte din vocabular.
pentru fiecare cuvânt (de V ori)		Rădăcină (șir), Formă (șir), Frecvență (Int32), IDF (Single), Embedding (vector).
N	Int32	Numărul de propoziții.
pentru fiecare propoziție (de N ori)		Text (șir), numărul de termeni n (Int32), apoi n perechi id (Int32) + pondere (Single), în final vectorul dens (vector).
vector	Int32 + Single…	Lungimea urmată de valori; lungime 0 = niciun vector.

Cu Pinocchio, fișierul are circa 2,8 MB. Matricele de co-ocurență și PPMI nu sunt salvate: sunt necesare doar în timpul antrenării. Pentru a răspunde sunt suficiente vocabularul, IDF, embedding-urile și vectorii propozițiilor.

8. Parametri și experimente

8.1 Parametrii de antrenare

Parametru	Implicit	Efect
Finestra contesto ± (Fereastra de context ±)	4 (1–10)	Câte cuvinte la dreapta și la stânga contează drept context. Fereastră îngustă: similarități mai gramaticale; largă: mai legate de subiect.
Dimensione vettori (Dimensiunea vectorilor)	128 (16–512)	Lungimea embedding-urilor. Mai multe numere = proiecție mai fidelă, dar mai multă memorie și mai multe calcule.
Frequenza minima (Frecvența minimă)	2 (1–20)	Sub acest prag, cuvântul nu primește embedding. Valori mari: mai puține cuvinte, vectori mai fiabili.
Seme	12345	Sămânța generatorului aleator (doar din cod). Schimbând-o se schimbă vectorii, nu calitatea medie.

8.2 Parametrii de răspuns

Parametru	Implicit	Efect
Peso semantico (Ponderea semantică)	0,30	0 = doar cuvinte identice (lexical), 1 = doar similaritatea embedding-urilor.
Peso contesto (Ponderea contextului)	0,25	Cât contează cele două propoziții vecine.
Parole simili per termine (Cuvinte similare per termen)	3	Cuvinte adăugate la întrebare pentru fiecare cuvânt. 0 = nicio extindere.
Soglia somiglianza (Pragul de similaritate)	0,35	Similaritatea minimă pentru ca un cuvânt similar să fie adăugat.
Regole tipo domanda (Reguli pentru tipul întrebării)	active	Activează sau dezactivează bonusurile pentru chi/dove/quando/quanto/perché/come.
FrasiContesto, SogliaIncertezza	1; 0,12	Doar din cod: propozițiile afișate în jurul răspunsului și pragul sub care răspunsul este declarat incert.

8.3 Experimente ghidate cu Pinocchio

1. *Chi ha costruito il burattino?* cu **Peso semantico** 0, apoi 0,30, apoi 1. Cu 0 și 0,30 răspunsul este „— Io sono un burattino.” (— Eu sunt o marionetă.); cu 1 devine „Nascono burattini, vivono burattini e muoiono burattini.” (Marionete se nasc, marionete trăiesc și marionete mor.): contează doar subiectul, nu cuvintele exacte.
2. *Chi ha mangiato Geppetto?* (Cine l-a mâncat pe Geppetto?) Răspunsul este „— E chi te li ha mangiati?” (— Și cine ți le-a mâncat?). Modelul este un „bag of words”: nu distinge cine mănâncă de cine este mâncat, iar ordinea cuvintelor nu contează.
3. *Che cosa vende Geppetto per comprare l'abbecedario?* (Ce vinde Geppetto ca să cumpere abecedarul?) găsește titlul capitolului VIII: „Geppetto rifà i piedi a Pinocchio, e vende la propria casacca per comprargli l'Abbecedario”. Extinderea adaugă „casacca” (haină) ca un cuvânt similar cu „comprato” (cumpărat): se vede în urmă cu simbolul ~.

4. Setați **Parole simili per termine** la 0 și repetați întrebarea anterioară: fără „casacca”, propoziția corectă coboară pe locul al patrulea, iar răspunsul devine „— E perchè mi avete comprato?”.
5. În fila Esplora căutați `fata`, `geppetto`, `naso`. Multe cuvinte similare sunt doar cuvinte apropiate în povestire („naso” → „giunti”, „spropositato”, „allungo”), nu sinonime: cu o singură carte, statistica este săracă.
6. Reantrenați cu **Finestra** 1 și apoi 8 și comparați cuvintele similare cu `naso`. Cu fereastra 8 apar „spropositato”, „allungo”, „punta”: cuvinte din aceeași scenă.
7. Scrieți *astronave* (navă spațială): programul răspunde că nu cunoaște niciunul dintre cuvinte. Nu știe nimic în afara textului citit și declară acest lucru.
8. Dezactivați **Regole tipo domanda** și repetați *Chi ha costruito il burattino?*: fără bonus, răspunsul devine „Io sono un burattino.” (fără liniuță). Regulile sunt singura cunoaștere scrisă de mână; tot restul provine din date.
9. Antrenați cu un alt text, de exemplu un manual tehnic sau o colecție de proceduri ale companiei în format `.txt`, și puneți întrebări despre conținutul său: programul nu depinde de Pinocchio.

9. Limite și extensii

9.1 Limite intenționate

- Stemmerul este rudimentar și gândit doar pentru italiană; cuvintele vide sunt italiene.
- Ordinea cuvintelor este ignorată (model „bag of words”): subiectul și complementul nu se disting.
- Negațiile („non”) sunt cuvinte vide: „è vero” (este adevărat) și „non è vero” (nu este adevărat) rezultă identice.
- Embedding-urile provin dintr-un singur text, deci reflectă vecinătățile din acel text, și nu sensul general al cuvintelor.
- Căutarea compară întrebarea cu toate propozițiile: este perfect pentru mii de propoziții, nu pentru milioane.
- Răspunsurile sunt întotdeauna propoziții existente: programul nu rezumă și nu combină informații luate din propoziții diferite.

9.2 De la jucărie la AI-urile reale

Aspect	MiniAI	Modele lingvistice mari
Date	O carte: circa 20.000 de cuvinte de conținut	Mii de miliarde de cuvinte
Unitatea de text	Rădăcini de cuvinte	Tokenuri (bucăți de cuvânt) dintr-un vocabular fix
Vectorii cuvintelor	Co-ocurențe + PPMI + proiecție aleatoare, 128 de numere	Învățați de o rețea neuronală, mii de numere
Context	Fereastră fixă de ±4 cuvinte	Mecanism de atenție pe mii de tokenuri, inclusiv ordinea
Numere învățate	Circa 600.000 (embedding-uri de cuvinte și propoziții)	Miliarde de parametri
Răspuns	Alege o propoziție existentă	Generează text nou, cuvânt cu cuvânt
Antrenare	Mai puțin de o secundă pe un PC	Săptămâni pe mii de procesoare specializate

Diferențele sunt enorme, dar esența este comună: reprezentarea limbajului cu vectori învățați din context și măsurarea similarităților. Un sistem RAG modern face, la o scară mult mai mare, ceea ce face MiniAI în pașii B–E: transformă întrebarea în vector, caută pasajele cele mai apropiate și le transmite modelului lingvistic, care le folosește pentru a formula răspunsul.

9.3 Extensii posibile

- **Generarea de propoziții**: un lanț Markov pe cuvinte (sau pe bigrame) antrenat pe același text, pentru a vedea trecerea de la alegere la generare.
- **Învățarea vectorilor**: înlocuirea proiecției aleatoare cu o mică rețea word2vec (skip-gram cu eșantionare negativă), tot fără biblioteci externe.
- **Mai mult text**: antrenarea pe mai multe fișiere împreună; este îmbunătățirea cea mai eficientă.
- **Bigrame**: adăugarea în vocabular a perechilor frecvente de cuvinte consecutive („fata turchina”) pentru a păstra puțin din ordine.
- **Negații**: unirea lui „non” cu cuvântul următor („non_vero”) în loc de a-l elimina.
- **Alte limbi**: posibilitatea de a configura cuvintele vide și sufixele.

- **Căutare rapidă**: un index inversat (cuvânt → propoziții) pentru scorul lexical și un index aproximativ pentru vectorii denși, dacă propozițiile ajung la sute de mii.

10. Licență și răspundere

MiniAI este distribuit gratuit, ca toate programele DevService.net, și se folosește pe propriul risc: este un proiect de studiu, furnizat ca atare, fără garanții de niciun fel. Codul sursă poate fi citit, modificat și folosit pentru a învăța și a experimenta.

Textul `Pinocchio.txt` (*Le avventure di Pinocchio*, Carlo Collodi, 1883) este în domeniul public și provine de pe Project Gutenberg (www.gutenberg.org); licența Project Gutenberg este inclusă în fișierul însuși.

Pentru semnalări și sugestii este disponibil formularul de contact de pe site-ul DevService.net.

MiniAI

教育用の人工知能: テキストからベクトルへ

プログラマー マニュアル

VB.NET、C#、JavaScript のソース コード

2026 年 10 月

目次

1. はじめに 93

1.1 MiniAI とは 93

1.2 大規模 AI と同じ原理 93

1.3 ダウンロードの内容 93

2. 要件、コンパイル、起動 95

2.1 VB.NET 版 95

2.2 C# 版 95

2.3 JavaScript 版 95

2.4 Pinocchio での初回起動 95

3. 基本概念 97

3.1 テキストから数値へ 97

3.2 コサイン類似度 97

3.3 分布仮説 97

3.4 検索と生成 97

4. 学習パイプライン 98

4.1 ステップ 1 – 文への分割 98

4.2 ステップ 2 – 正規化、語幹、語彙 98

4.3 ステップ 3 – IDF 98

4.4 ステップ 4 – 疎な TF-IDF 行列 99

4.5 ステップ 5 – 共起 99

4.6 ステップ 6 – PPMI 99

4.7 ステップ 7 – ランダム射影による埋め込み 100

4.8 ステップ 8 – 文の密ベクトル 100

4.9 学習ログ 100

5. 質問への回答 101

5.1 A – 質問の種類 101

5.2 B – 質問の単語 101

5.3 C – 類似語による拡張 101

5.4 D – 文のスコア 101

5.5 E – 文脈 102

5.6 F – 回答 102

5.7 トレースの読み方 102

6. コードの構成 104

6.1 ファイルと対応関係 104
6.2 ElaboratoreTesto 104
6.3 ModelloConoscenza 105
6.4 MotoreDomande 105

6.5 ユーザー インターフェイス 106

6.6 3 つの言語間の移植に関する注意 106

7. .mai ファイルの形式 108

8. パラメーターと実験 109

8.1 学習パラメーター 109

8.2 回答パラメーター 109

8.3 Pinocchio を使ったガイド付き実験 109

9. 制限と拡張 111

9.1 意図的な制限 111

9.2 おもちゃから実際の AI へ 111

9.3 考えられる拡張 111

10. ライセンスと責任 113

1. はじめに

1.1 MiniAI とは

MiniAI は、言語に応用される人工知能について私が行った基礎的な研究です。長いテキスト (1 冊の本、文書の集まり、マニュアルなど) を読み込み、それを数値、ベクトル、行列に変換し、質問に対してはテキストの中から質問に最も近い文を探して回答します。

本番環境で使用するための製品ではなく、実験室です。計算の各ステップが可視化されており、プログラムは構築する行列のサイズをログに書き込み、回答ごとに「推論のトレース」を表示します。つまり、どの単語を認識したか、どの単語を追加したか、どの文を比較し、それぞれどのスコアだったかがわかります。目的は驚かせることではなく、理解することです。

このプログラムは**ソース コード**として、同等の 3 つのバージョン (**VB.NET**、**C#**、**JavaScript**) で配布されています。外部ライブラリは使用しません。必要な処理 (文への分割から埋め込みのランダム射影まで) はすべて、コメント付きの数百行のコードで省略せずに記述されています。

1.2 大規模 AI と同じ原理

当然ながら、データの量が大きな違いを生みます。市販の AI は数十億語で学習され、数十億のパラメーターを持ち、新しいテキストの生成を学習するニューラル ネットワークを備えています。MiniAI は 1 冊の本だけを読み、1 秒未満で数千個のベクトルを学習します。しかし、基本的な原理は同じです。

- **テキストは数値になる**: 各単語と各文はベクトルで表されます。
- **意味は文脈から生まれる**: 似た文脈で使われる 2 つの単語には、似たベクトルが割り当てられます (分布仮説)。
- **質問を理解するとは、類似度を測ること**: 質問は同じベクトル空間に変換され、システムが知っている内容と比較されます。
- **回答する前に検索する**: 関連する文の検索 (*retrieval*) は、RAG (*Retrieval-Augmented Generation*) システムで使われているのと同じ仕組みです。RAG システムは、回答の根拠となる文書を言語モデルに提供します。

MiniAI は検索の段階で止まります。**テキストを生成するのではなく、既存の文を選択します**。だからこそ、出発点として適しています。統計だけで何ができるのか、そしてどこでより多くのデータやより強力なモデルが必要になるのかがはっきりとわかります。

> 同じコードでも、テキストが多いほど良い結果が得られます。小説 1 冊だけでは、「類似語」の多くは物語の中で近くに現れる単語にすぎません。同じテーマの本が数十冊あれば、本当の同義語が現れ始めます。これは、大規模 AI が膨大な量のデータを必要とする理由を小さな規模で示したものです。

1.3 ダウンロードの内容

サイトでは、言語ごとに 1 つずつ、3 つの zip ファイルを提供しています。それぞれに完全なプログラム、サンプル テキスト、およびイタリア語の `LEGGIMI.md` ファイルが含まれています。

ファイル	内容
`MiniAI_VBNET.zip`	.NET Framework 4.8 用の VB.NET による Windows フォーム プロジェクト: `MiniAI.sln`、`MiniAI.vbproj`、`compila.bat`、およびファイル `Testo.vb`、`Modello.vb`、`Motore.vb`、`FormPrincipale.vb`、`Program.vb`。
`MiniAI_CSharp.zip`	C# による同じプロジェクト: `MiniAI.sln`、`MiniAI.csproj`、`compila.bat`、および同じ名前の `.cs` ファイル。
`MiniAI_JavaScript.zip`	単独で動作する Web ページ (`index.html`、`stile.css`、`js` フォルダー) と、Node.js でコマンド ラインから使用するための `prova-node.js`。
`Pinocchio.txt`	3 つの zip すべてに含まれるもの: Carlo Collodi 作『*Le avventure di Pinocchio*』(ピノッキオの冒険、1883 年)。Project Gutenberg から入手したパブリック ドメインのテキストで、このマニュアルのすべての例で使用しています。

3 つのバージョンは、同じ計算を同じ順序で実行します。同じテキストと同じパラメーターを使えば、同じ回答、同じトレース、そしてバイト単位で同一のモデル ファイル `.mai` が生成されます。あるバージョンで保存したモデルは、ほかのバージョンで開くことができます。

2. 要件、コンパイル、起動

2.1 VB.NET 版

- Windows 10/11 または Windows Server 2016 以降で、.NET Framework 4.8 が必要です (Windows 10 バージョン 1903 以降、Windows 11、Windows Server 2022 以降には標準で含まれています)。
- **Visual Studio を使用する場合** (2019 または 2022、ワークロード「.NET デスクトップ開発」): `MiniAI.sln` を開き、F5 キーを押します。
- **Visual Studio を使用しない場合**: `compila.bat` をダブルクリックします。このファイルは、.NET Framework 4.8 が `vbc.exe` をインストールする場所 `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319` にあるコンパイラを使用します。コードはこのコンパイラ (Visual Basic 2012) と互換性があるように書かれており、同じフォルダーに `MiniAI.exe` が作成されます。

`.vb` ファイルは BOM 付き UTF-8 で保存されています。これにより、Framework のコンパイラでもコメントやメッセージ内のアクセント付き文字が正しく読み取られます。

2.2 C# 版

- 要件は VB.NET 版と同じです。
- **Visual Studio を使用する場合**: `MiniAI.sln` (プロジェクト `MiniAI.csproj`) を開き、F5 キーを押します。
- **Visual Studio を使用しない場合**: `compila.bat` をダブルクリックします。これは .NET Framework 4.8 の `csc.exe` を使用します。コードは、このコンパイラのバージョンである C# 5 と互換性があります。文字列補間やその他の新しい構文は使用していません。

2.3 JavaScript 版

- **ブラウザーで使用する場合**: `index.html` をダブルクリックします。最新の Edge、Chrome、Firefox で動作し、インターネット接続やサーバーがなくても使用できます。テキストがコンピューターの外に出ることはありません。

- **Node.js で使用する場合** (バージョン 18 以降)、プログラムのフォルダーから実行します。`node prova-node.js Pinocchio.txt` はモデルを学習して質問セッションを開始します。`node prova-node.js Pinocchio.txt domande.txt modello.mai` は `domande.txt` の各行に回答し、モデルを保存します。`node prova-node.js modello.mai` は保存済みのモデルから再開します。`--traccia` オプションを指定すると、推論のトレースも出力されます。

ブラウザーでは、計算の間 (Pinocchio の場合は 1 秒未満) 学習によってページがブロックされます。非常に長いテキストでは、ページが数秒間止まっているように見えることがあります。

2.4 Pinocchio での初回起動

1. **1 · Addestramento** (学習) タブ: 「Apri testo (.txt)...」(テキストを開く) をクリックし、`Pinocchio.txt` を選択します (JavaScript 版では、ファイルをページにドラッグすることもできます)。
2. パラメーターは既定値 (ウィンドウ 4、サイズ 128、最小頻度 2) のままにして、「Addestra」(学習) をクリックします。ログには、第 4 章で説明する 8 つのステップが表示されます。
3. **2 · Domande** (質問) タブ: *Chi ha costruito il burattino?* (誰があやつり人形を作ったか?) と入力し、Enter キーを押します。上部に回答と近くの文が、下部に完全なトレースが表示されます。
4. **3 · Esplora la memoria** (メモリの探索) タブ: `naso` (鼻) と入力し、「Parole simili」(類似語) をクリックします。
5. 省略可能: 「Salva modello...」(モデルの保存) を使うと `.mai` ファイルが作成され、「Carica modello...」(モデルの読み込み) で瞬時に再読み込みできます。

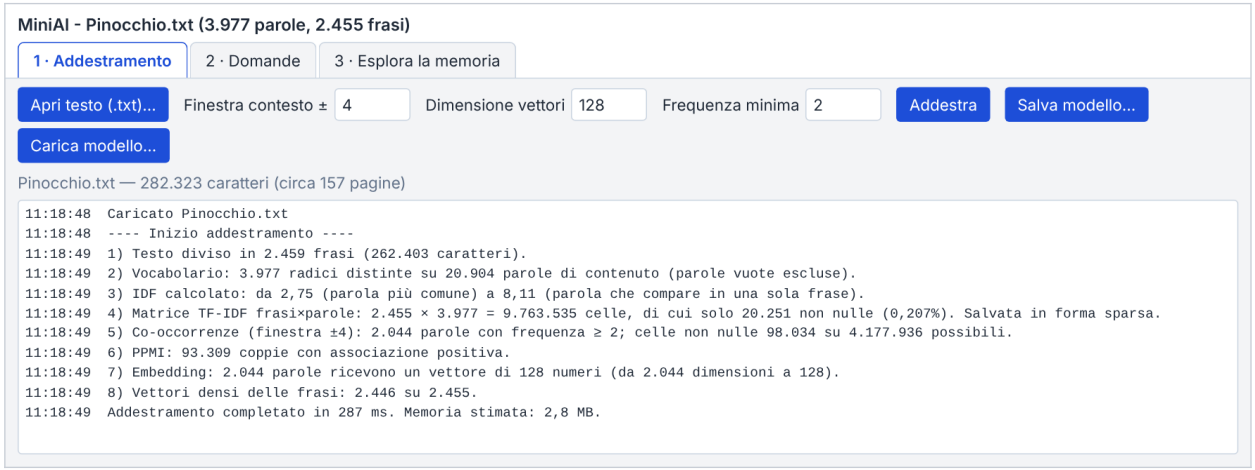

図 1 – Pinocchio で学習した後の Addestramento タブ (JavaScript 版。Windows 版にも同じ 3 つのタブがあります)

テキストは UTF-8 として読み込まれます。UTF-8 として無効なバイトが含まれている場合は、Windows のエンコード (Windows-1252) で読み直されます。そのため、メモ帳で保存された古い `.txt` ファイルも使用できます。

3. 基本概念

3.1 テキストから数値へ

コンピューターは「burattino」(あやつり人形) が何であるかを知りません。しかし、数を数えたり、数値のリストに対して演算を行ったりすることはできます。MiniAI の処理はすべて、単語や文を**ベクトル**、つまり固定長の数値のリストに変換し、意味が似ているテキストが似たベクトルになるようにすることにあります。

プログラムには 2 種類のベクトルが登場します。

- **疎ベクトル**: 長さは語彙と同じ (Pinocchio の場合、約 4,000 個の位置) で、ほとんどがゼロです。各位置が 1 つの単語に対応します。ゼロ以外の位置だけが格納されます。
- **密ベクトル** (*embedding*、埋め込み): 短く (128 個の数値)、すべての値に意味があります。特定の単語に対応する位置はなく、数値全体で意味を表します。

3.2 コサイン類似度

2 つのベクトルがどれだけ似ているかを表すために、MiniAI はベクトル間の**角度のコサイン**を使用します。

```
cosine(a, b) = (a · b) / (|a| · |b|)

a · b = a1·b1 + a2·b2 + ... + an·bn    (内積)
|a|   = (a · a) の平方根          (ベクトルの長さ)
```

結果は、ベクトルが同じ方向を向いていれば 1、共通点がなければ 0、反対向きであれば負の値になります。重要なのは長さではなく方向です。同じテーマについての長い文と短い文は、似ていると判定されます。

計算を省くため、すべてのベクトルは作成された直後に**正規化** (長さ 1 にすること) されます。それ以降、コサインは単純な内積と一致します。これが `Normalizza` 関数と `Prodotto` クラスの `ModelloConoscenza` 関数が行っている処理です。

3.3 分布仮説

「単語の意味は、その単語と共に現れる単語によって知られる」(J. R. Firth、1957 年)。2 つの単語が同じほかの単語の近くによく現れるなら、それらの意味はおそらく似ています。MiniAI はこの考え方を応用し、各単語について、テキスト内でその近くにどの単語があるかを数え (共起行列)、その数を密ベクトルに変換します。word2vec、GloVe、そして最新の言語モデルの埋め込み層も同じ原理に基づいています。

3.4 検索と生成

テキストをもとに質問に回答する方法は 2 つあります。

- **抽出型** (検索): テキストの中から最も関連性の高い部分を探し、そのまま返します。これは MiniAI が行っている処理であり、RAG システムの最初の段階です。
- **生成型**: 言語モデルが、毎回最も確率の高い単語を選びながら、1 語ずつ新しい文を作り出します。膨大な量のテキストで学習したニューラル ネットワークが必要です。

検索には重要な利点があります。回答は常に文書内に実在する検証可能な文であり、システムは「これらの単語はどれも知りません」と正直に答えることができます。

4. 学習パイプライン

学習は `Addestra` クラスの `ModelloConoscenza` メソッド (ファイル `Modello.vb`、`Modello.cs`、または `js/modello.js`) によって行われます。8 つのステップで構成され、各ステップがログに 1 行を書き込みます。この章に記載されている数値は、`Pinocchio.txt` と既定のパラメーターで得られたものです。

4.1 ステップ 1 – 文への分割

ファイルが Project Gutenberg のものである場合、`TogliIntestazioneGutenberg` が冒頭と末尾のライセンスを削除します (`*** START OF` と `*** END OF` の行を探します)。次に、`DividiInFrasi` が以下の処理を行います。

- `.txt` ファイルに典型的な「固定」改行で分断された行を結合します。段落は空行で区切られます。
- 文末記号 (. ! ? …) があり、その後に引用符が続く場合もあり、さらにスペースと大文字、数字、または会話のダッシュが続く位置で、各段落を分割します。
- 4 語未満の文は破棄します。

文は**回答の単位**です。MiniAI は常に文全体で回答します。Pinocchio では、262,403 文字から 2,459 個の文が得られます。

4.2 ステップ 2 – 正規化、語幹、語彙

各文は、**内容語** 関数によって `ParoleContenuto` だけに絞り込まれます。

1. `Normalizza`: すべて小文字にし、アクセントを除去し (Unicode NFD 分解と分音記号の削除)、文字や数字以外の文字はすべてスペースに置き換えます。「Perché l'albero?」は `perche l albero` になります。
2. **ストップ ワード** (冠詞、前置詞、助動詞、代名詞など。`Testo` に約 200 語が列挙されています) と 1 文字の単語は破棄されます。これらはどこにでも現れ、内容について何も語らないからです。
3. 語幹: 意図的に単純化したステマーが、既知の語尾 (`-issimo`、`-mente`、`-zione`、`-avano`、`-ando`、`-ato`、`-ito`… など) または末尾の母音を取り除きます。「burattino」、「burattini」、「burattina」はすべて `burattin` になり、「andato」と「andavano」は `and` になります。

新しい語幹にはそれぞれ、**id** 辞書の中で連番 (その語幹の `Vocabolario`) が割り当てられます。各語幹について出現回数 (`Frequenze`) を数え、最も頻度の高い形 (`Forme`) を記録します。これは単語を読みやすく表示するためだけに使われます。Pinocchio の場合: 20,904 個の内容語から **3,977 個の語幹**が得られます。

> ステマーはよく間違えます。そして、どこで間違えるかを見ることには学ぶ価値があります。「costruito」(作った) は Pinocchio に一度も出てきませんが、その語幹 `costru` は、一度だけ出てくる「costruì」の語幹と一致します。そのため、質問の単語は同じ動詞の別の形によって認識されます。

4.3 ステップ 3 – IDF

すべての単語が同じだけの情報を持つわけではありません。「pinocchio」は数百の文に現れますが、「abbecedario」(読み方の教本) はごくわずかな文にしか現れません。**IDF** (*inverse document frequency*、逆文書頻度) は単語の希少性を測ります。

```
idf(t) = ln( (N + 1) / (df(t) + 1) ) + 1

N   = 文の数
df(t) = 単語 t を含む文の数
```

分子と分母の +1 はゼロ除算を防ぎ、最後の +1 は最も一般的な単語でも重みが正になるようにします。Pinocchio では、IDF は 2.75 (最も一般的な単語「pinocchio」) から 8.11 (1 つの文にしか現れない単語) までの値をとります。3 つのバージョンにおける同じ行を次に示します。

```
' VB.NET
Idf(i) = CSng(Math.Log((N + 1.0) / (df(i) + 1.0)) + 1.0)

// C#
Idf[i] = (float)(Math.Log((N + 1.0) / (df[i] + 1.0)) + 1.0);


// JavaScript (Float32Array: Single への変換は自動で行われます)
this.idf[i] = Math.log((N + 1.0) / (df[i] + 1.0)) + 1.0;
```

4.4 ステップ 4 – 疎な TF-IDF 行列

各文は、語彙と同じ長さのベクトルになります。文 f における単語 t の重みは次のとおりです。

```
重み(t, f) = (1 + ln tf(t, f)) · idf(t)     tf = 文中の t の出現回数


その後、ベクトルを長さ 1 に正規化
```

対数は繰り返しの影響を抑えます。3 回繰り返される単語の価値は 3 倍にはなりません。文 × 単語の完全な行列は 2,455 × 3,977 = 9,763,535 個のセルになりますが、ゼロ以外のセルは 20,251 個 (0.207%) だけです。そのため、各文は 2 つの並列配列だけを格納します。`Termini` (含まれる単語の id、昇順) と `Pesi` (それぞれの重み) です。これは**疎表現**と呼ばれ、検索エンジンで使われているものと同じです。

ストップ ワードだけで構成される文はベクトルを生成しないため、破棄されます。格納される文が 2,459 個ではなく 2,455 個なのはこのためです。

4.5 ステップ 5 – 共起

ここから「意味」の部分が始まります。頻度が `FrequenzaMinima` (2) 以上の単語だけを対象にします。一度しか出てこない単語は、信頼できるベクトルを得るのに十分な文脈を持たないからです。Pinocchio では 2,044 語です。

すべての文の内容語を、1 つの連続した流れとして並べます。各単語 a と、その後ろ `Finestra` 個 (4) 以内の位置にある各単語 b について、セル (a, b) と (b, a) の両方に、距離に応じて減少する重みを加算します。

```
cooc(a, b) += 1 / 距離      距離 = 1, 2, ..., Finestra
```

理論上の 2,044 × 2,044 行列は 4,177,936 個のセルを持ちますが、ゼロでないものは 98,034 個です。この行列も疎な形式で、`a · V + b` (64 ビット整数) をキーとする辞書に格納されます。

4.6 ステップ 6 – PPMI

生の共起数は頻出語を優遇します。「disse」(言った) はほとんどすべての単語の近くにあります。**PMI** (*pointwise mutual information*、自己相互情報量) は、2 つの単語が近くに現れる回数を、まったくの偶然で期待される回数と比較します。

```
PMI(a, b) = ln( P(a, b) / ( P(a) · Pc(b) ) )

P(a, b) = cooc(a, b) / 合計
P(a)    = 行 a の和 / 合計
Pc(b)   = (行 b の和)^0.75 / すべての (行の和)^0.75 の和

PPMI(a, b) = max(0, PMI(a, b))
```

正の値だけを残します (**PPMI**、*positive* PMI)。つまり、偶然より頻繁に一緒に現れるペアです。文脈の確率に対する指数 0.75 は word2vec と同じ工夫で、まれな文脈の過大な重みを抑えます。Pinocchio の場合: 正の関連を持つペアは 93,309 個です。

4.7 ステップ 7 – ランダム射影による埋め込み

PPMI 行列の各行はすでに単語を表すベクトルですが、長さが 2,044 あります。短いベクトルを得るために、MiniAI は**ランダム射影**を使用します。

1. 各文脈語 b に、`Dimensione` 個 (128) の数値からなる固定のランダム ベクトル r(b) を割り当てます。数値は標準偏差 1/√D の正規分布から取り出します (Box-Muller 変換)。
2. 単語 a の埋め込みは、その文脈のベクトルを PPMI で重み付けした和です: e(a) = Σ PPMI(a, b) · r(b)。
3. ベクトルを長さ 1 に正規化します。

Johnson-Lindenstrauss の補題により、ランダム射影では元のベクトル間の距離とコサインがおおよそ保たれることが保証されます。PPMI の行が似ている 2 つの単語は、埋め込みも似たものになります。これは *random indexing* として知られる手法です。最新のモデルは、ニューラル ネットワークでベクトルを学習することで同じ結果を得ています。

乱数ジェネレーターは常に同じ `Seme` (シード、12345) から始まるため、学習は**再現可能**です。同じテキストと同じパラメーターからは、常に同じベクトルが得られます。Pinocchio では 2,044 語が埋め込みを受け取ります。まれすぎる単語は埋め込みを持ちません (値は `Nothing`/`null`)。

4.8 ステップ 8 – 文の密ベクトル

文の密ベクトルは、その文の単語の埋め込みを、ステップ 4 で計算した TF-IDF の重みで加重平均し、正規化したものです。まれで特徴的な単語は、一般的な単語より大きく寄与します。Pinocchio では、2,455 個の文のうち 2,446 個が密ベクトルを持ちます。残りの文には、埋め込みを持たない単語しか含まれていません。

4.9 学習ログ

次に示すのは、Pinocchio と既定のパラメーターで生成された完全なログです。3 つのバージョンはすべて同じ行を書き込みます。

```
1) Testo diviso in 2.459 frasi (262.403 caratteri).
2) Vocabolario: 3.977 radici distinte su 20.904 parole di contenuto
   (parole vuote escluse).
3) IDF calcolato: da 2,75 (parola più comune) a 8,11 (parola che compare
   in una sola frase).
4) Matrice TF-IDF frasi×parole: 2.455 × 3.977 = 9.763.535 celle, di cui
   solo 20.251 non nulle (0,207%). Salvata in forma sparsa.

5) Co-occorrenze (finestra ±4): 2.044 parole con frequenza ≥ 2; celle non
   nulle 98.034 su 4.177.936 possibili.
6) PPMI: 93.309 coppie con associazione positiva.
7) Embedding: 2.044 parole ricevono un vettore di 128 numeri (da 2.044
   dimensioni a 128).
8) Vettori densi delle frasi: 2.446 su 2.455.
```

学習には、コンピューターによって 1 秒の数分の一から 1 秒程度かかります。モデルはメモリ上でもディスク上でも約 2.8 MB を占めます。

5. 質問への回答

回答は `Rispondi` クラスの `MotoreDomande` メソッド (ファイル `Motore.vb`、`Motore.cs`、または `js/motore.js`) によって生成されます。このメソッドは、回答のテキスト、トレース、選ばれた文のスコアという 3 つのフィールドを持つ `Risposta` オブジェクトを返します。各ステップは A から F の文字で示され、トレースに表示されるものと同じです。

5.1 A – 質問の種類

質問の最初の 3 語の中に *chi* (誰)、*dove* (どこ)、*quando* (いつ)、*quanto* (どれだけ。または *quanti*、*quante*、*quanta*)、*perché* (なぜ)、または *come* (どのように) があれば、質問の種類が認識されます。これは、固定のルールで「手作業」で書かれたプログラム内の唯一の部分であり、プログラムされた知識とデータから学習した知識を比較するのに役立ちます。

5.2 B – 質問の単語

質問には、テキストと同じ変換 (正規化、ストップ ワードの除去、語幹への還元) が適用されます。各語幹は語彙の中で検索されます。一度も見たことのない単語は、トレースに「MAI viste nel testo」(テキスト中に一度も出現せず) として列挙されます。モデルは、読み込んだテキスト以外のことは何も知りません。既知の単語が 1 つもない場合、回答は *Non conosco nessuna delle parole della domanda* (質問の単語をどれも知りません) になります。

認識された単語には、文の場合と同様に TF-IDF の重み (1 + ln tf) · idf が割り当てられます。

5.3 C – 類似語による拡張

質問の各単語について、埋め込みが最も似ている単語を `Espansione` 個 (3) 探します。類似度が `SogliaEspansione` (0.35) 以上のものは、低い重みで質問に追加されます。

```
重み(追加された単語) = idf · 類似度 · 0.5
```

このようにして、「vendere」(売る) についての質問が、「mercato」(市場) について述べている文を見つけることができます。トレースでは、追加された単語は記号 ~ で示されます。その後、質問のベクトルが正規化されます。

質問の密ベクトルも計算されます。これは、元の単語 (追加された単語ではありません) の埋め込みを IDF で重み付けして合計し、正規化したものです。

5.4 D – 文のスコア

格納されている各文は、3 つのスコアを受け取ります。

- **語彙的スコア**: 質問の疎ベクトルと文の疎ベクトルの間のコサインです。共通する単語についてだけ積を合計するため、計算は非常に高速です。
- **意味的スコア**: 質問の密ベクトルと文の密ベクトルの間のコサインです (負の値は 0 になります)。異なる単語を使っていても同じテーマについて述べている文が優遇されます。
- **ボーナス**: 文に質問の種類に合った手がかりが含まれている場合に与えられる小さな加点 (0.02–0.03) です。*chi* には固有名詞 (実際には、文の最初の単語ではない、頭文字が大文字の単語)、*quando* には時を表す単語、*dove* には場所、*perché* には原因、*quanto* には数値、*come* には方法です。ボーナスは、質問と共通する単語を少なくとも 1 つ持つ文にのみ適用されます。

```
生スコア = (1 - ps) · 語彙的 + ps · 意味的 + ボーナス      ps = PesoSemantico (0.30)
```

5.5 E – 文脈

物語の中では、関連する文は関連する文に囲まれていることがよくあります。最終スコアには、前後 2 つの文のスコアの平均の一部が加算されます。

```
合計(i) = 生スコア(i) + pc · (生スコア(i-1) + 生スコア(i+1)) / 2    pc = PesoContesto (0.25)
```

文は合計スコアの降順に並べ替えられ (安定ソート: スコアが同じ場合はテキストで先に出てくる文が優先されます)、トレースには上位 5 つが表示されます。

5.6 F – 回答

スコアが最も高い文が回答になります。スコアが `SogliaIncertezza` (0.12) 未満の場合、回答の前に *(Risposta incerta: punteggio basso)* (不確かな回答: スコアが低い) が付きます。スコアが 0 の場合、プログラムは *Non ho trovato nulla di pertinente* (関連するものは見つかりませんでした) と回答します。回答の下には、その前後の文が `FrasiContesto` 個 (1) ずつ表示され、文脈の中で読むことができます。

5.7 トレースの読み方

次に示すのは、既定のパラメーターでの質問 *Chi ha costruito il burattino?* のトレースです。

```
A) TIPO DI DOMANDA: CHI
B) PAROLE DELLA DOMANDA
  ignorate (parole vuote): chi, ha, il
  costrui      radice 'costru' frequenza   1 IDF 8,11 (senza embedding)
  burattino     radice 'burattin' frequenza  226 IDF 3,40
C) ESPANSIONE CON PAROLE SIMILI (embedding)
  costrui      -> nessuna sopra soglia
  burattino     -> guardare (0,35)
D) CLASSIFICA (2.455 frasi confrontate)  peso semantico 0,30, peso contesto 0,25
  pos frase  lessic. semant. bonus  TOTALE  parole in comune
   1  1026  0,385  1,000  0,030  0,607  burattino

       — Io sono un burattino.
   2  2140  0,385  1,000  0,000  0,588  burattino
        Io sono un burattino.
   3  1245  0,174  1,000  0,000  0,500  burattino
       Nascono burattini, vivono burattini e muoiono burattini.
```

いくつかの学びがあります。「costruito」は、一度だけ出現するため埋め込みを持たない「costruì」の語幹を通じてのみ認識されます。「burattino」という単語だけを含む文では、意味的スコアが 1.000 になります。それらの密ベクトルが質問の密ベクトルと一致するからです。1 番目と 2 番目の文は語彙的スコアが同じで、1 番目が勝つのは *chi* のボーナスだけによるものです。会話のダッシュの後の「Io」(私) という単語が大文字で始まっているため、ルールがそれを固有名詞と取り違えています。手作業で書かれたルールは理解しやすい反面、同じくらい簡単にだまされます。人間の視点からは回答は間違っています (正しい答えは「Geppetto」です) が、これはまさにこのテキストの統計が示唆する結果です。

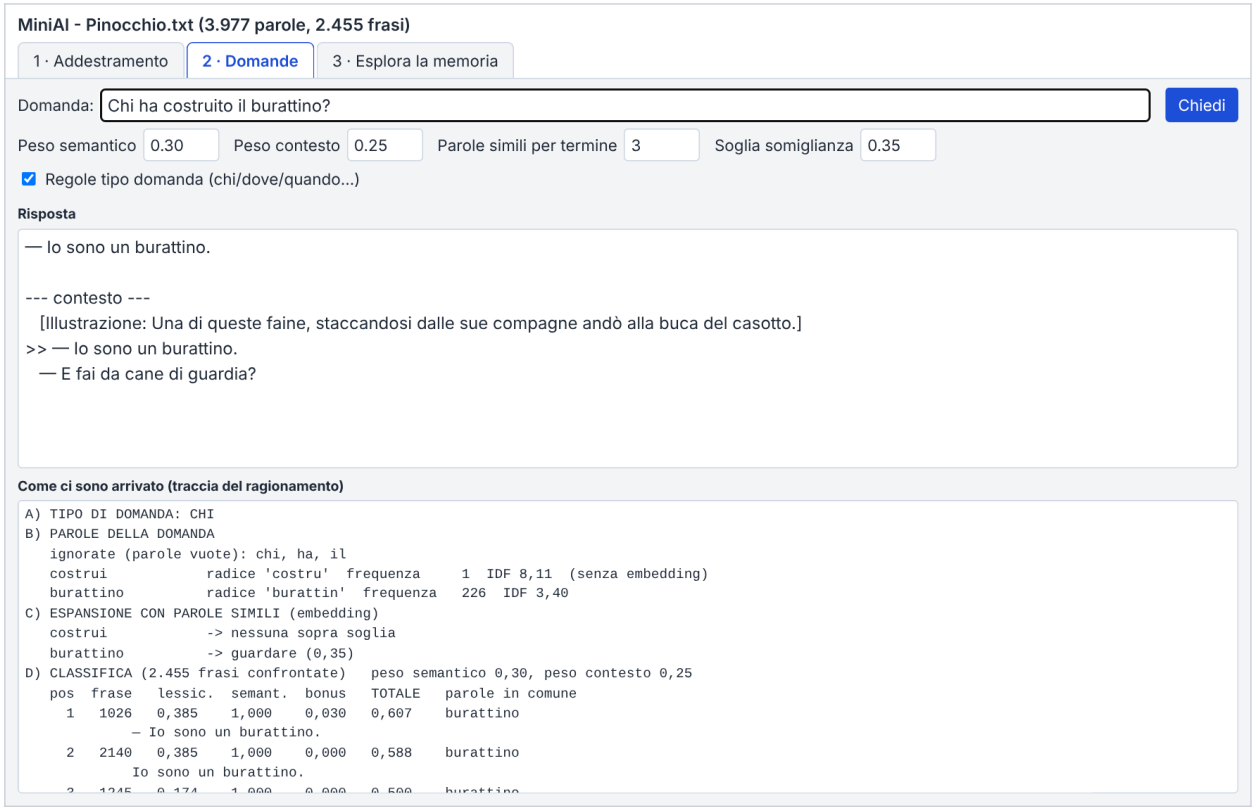

図 2 – Domande タブ: 回答とその文脈、その下に推論のトレース

6. コードの構成

6.1 ファイルと対応関係

クラス、メソッド、変数の名前はイタリア語で、3 つのバージョンで共通です。JavaScript では言語の規約に従います (先頭が小文字: `rispondi` ではなく `Rispondi`)。

VB.NET	C#	JavaScript	内容
`Testo.vb`	`Testo.cs`	`js/testo.js`	`ElaboratoreTesto` クラス: 文、正規化、ストップ ワード、語幹。
`Modello.vb`	`Modello.cs`	`js/modello.js`	`Frase` クラスと `ModelloConoscenza` クラス: 学習、類似語、保存と読み込み。
`Motore.vb`	`Motore.cs`	`js/motore.js`	`Risposta` クラスと `MotoreDomande` クラス: 質問への回答とトレース。
`FormPrincipale.vb`	`FormPrincipale.cs`	`index.html`、`js/app.js`、`stile.css`	3 つのタブを持つインターフェイス。.NET 版は、デザイナー ファイルを使わずにコードからコントロールを構築します。
`Program.vb`	`Program.cs`	`prova-node.js`	エントリ ポイント。JavaScript では `prova-node.js` がコマンド ラインでの代替手段です。

6.2 ElaboratoreTesto

静的関数だけを含みます (VB.NET では `Shared`、C# では `static`、JavaScript ではオブジェクト リテラル)。

メンバー	説明
`EParolaVuota(parola)`	単語 (正規化済み) がストップ ワードであれば True。
`TogliIntestazioneGutenberg(testo)`	Project Gutenberg の冒頭と末尾のライセンスがあれば削除します。

メンバー	説明
`Normalizza(testo)`	小文字化、アクセントの除去、英数字以外の文字をスペースに置換。
`Parole(testo)`	正規化された単語の配列。
`Radice(parola)`	単語の語幹 (簡易ステマー)。
`ParoleContenuto(testo)`	内容語の (語幹, 形) のペア。
`DividiInFrasi(testo, paroleMinime = 4)`	テキストの文のリスト。

6.3 ModelloConoscenza

メンバー	説明
`Vocabolario`	語幹 → id の辞書。
`Radici`、`Forme`、`Frequenze`	各 id について: 語幹、最も頻度の高い形、出現回数。
`Idf`	各単語の IDF を格納する Single の配列 (JavaScript では `Float32Array`)。
`Embedding`	各 id について `Dimensione` 個の Single からなるベクトル、または Nothing/null。
`Frasi`	`Frase` オブジェクトのリスト: `Testo`、`Termini` (並べ替え済みの id)、`Pesi` (TF-IDF、ノルム 1)、`Denso` (密ベクトルまたは null)。
`Dimensione`、`Finestra`、`FrequenzaMinima`、`Seme`	学習のハイパーパラメーター (128、4、2、12345)。
`Origine`	モデルの学習に使用したテキスト ファイルの名前。
`Addestra(testo, log)`	第 4 章の 8 つのステップを実行します。`log` はステップごとに 1 行を受け取ります。
`IdTermine(radice)`	語幹の id。見つからない場合は -1。
`Simili(id, k)`	埋め込みが最も近い k 個の単語を、(id, コサイン) のペアとして降順で返します。

メンバー	説明
`Salva(percorso)` / `Carica(percorso)`	`.mai` ファイル (第 7 章) の書き込みと読み込み。JavaScript では、`salva()` は `Uint8Array` を返し、`carica(dati)` はファイルのバイトを受け取ります。
`Prodotto(a, b)`、`Normalizza(v)`	内積 (Double で計算) と長さ 1 への正規化。
`StimaMB()`	推定使用メモリ (MB 単位)。

6.4 MotoreDomande

コンストラクターにモデルを渡して作成します。パラメーターはパブリック フィールドで、インターフェイスが各質問の前に更新します (第 8 章を参照)。

```
' VB.NET
Dim motore As New MotoreDomande(modello)
motore.PesoSemantico = 0.5
Dim r As Risposta = motore.Rispondi("Dove vive la fata?")
Console.WriteLine(r.Testo)

// C#
var motore = new MotoreDomande(modello) { PesoSemantico = 0.5 };
Risposta r = motore.Rispondi("Dove vive la fata?");

// JavaScript
const motore = new MiniAI.MotoreDomande(modello);
motore.pesoSemantico = 0.5;
const r = motore.rispondi('Dove vive la fata?');
console.log(r.testo, r.punteggio);
```

6.5 ユーザー インターフェイス

ウィンドウには 3 つのタブがあります。**Addestramento** (学習: テキストを開く、パラメーター、ログ、モデルの保存と読み込み)、**Domande** (質問: 質問、回答パラメーター、回答とトレース)、そして **Esplora la memoria** (メモリの探索) です。最後のタブでは、ある単語に最も似ている 25 語とそのベクトルの最初の数値、2 つの単語間の類似度、および最も頻度の高い 150 語の語彙 (語幹、頻度、IDF、埋め込みの有無) を確認できます。

.NET 版では、学習は別のスレッド (`Task.Run`) で実行され、ログは `IProgress` を介してインターフェイスに届くため、ウィンドウは応答性を保ちます。JavaScript 版では、すべてページのスレッドで実行されます。`.mai` ファイルはブラウザーのダウンロードとして保存されます。



図 3 – Esplora la memoria タブ: 「naso」に最も近いベクトルを持つ単語

6.6 3 つの言語間の移植に関する注意

3 つの言語でビット単位まで同一の結果を得るには、いくつかの注意が必要です。コードを変更する場合に知っておくと役立ちます。

- **乱数**: JavaScript では `Math.random` のシードを固定できません。`js/modello.js` には `RandomNet` が含まれています。これは .NET の `System.Random` のアルゴリズム (Knuth の減算型ジェネレーター) を正確に再現したもので、これによりランダム ベクトルが同じになります。
- **Single と Double**: .NET は、ベクトルと重みには `Single` (32 ビット) を、和と内積には `Double` (64 ビット) を使用します。JavaScript は常に 64 ビットで計算します。そのため、ベクトルは `Float32Array` になっており、2 つの Single 値の積はすべて、.NET と同じように `Math.fround` で丸められます。
- **辞書の順序**: .NET の `Dictionary` は、要素を削除しなければ、キーを挿入順に返します。JavaScript の `Map` も同様です。和の順序によって下位の桁が変わるため、コードはどこでも同じ順序を維持しています。
- **並べ替え**: ランキングには安定ソート (.NET では `OrderByDescending`、JavaScript では `Array.prototype.sort`) を使用します。`Simili` は .NET では `List.Sort`、JavaScript では `sort` を使用し、コサインがまったく同じ場合を除いて結果は一致します。

- **書式**: メッセージはイタリア語の書式 (2.459; 2,75; 0,207%) を使用します。JavaScript 版には、ブラウザーの設定に依存せずにこれを再現する小さな `Formato` オブジェクトがあります。
- **Framework 4.8 との互換性**: VB.NET 版は Visual Basic 2012 と、C# 版は C# 5 と互換性があります。これらは Windows に含まれているコンパイラの言語であり、何もインストールせずに `compila.bat` でコンパイルできるようにするためです。

7. .mai ファイルの形式

学習したモデルは、拡張子 `.mai` のバイナリ ファイルに保存されます。形式は .NET の `BinaryWriter` のものです。32 ビット整数と Single の数値はリトル エンディアンで、文字列は UTF-8 で格納され、その前にバイト単位の長さが 7 ビットずつのグループでエンコードされて置かれます (各バイトの最上位ビットは、さらにバイトが続くことを示します)。存在しないベクトルは長さ 0 として書き込まれます。

フィールド	型	注
シグネチャ	文字列	常に `MINIAI01`。
Dimensione、Finestra、FrequenzaMinima、Seme	4 × Int32	学習で使用したハイパーパラメーター。
元のファイル	文字列	テキスト ファイルの名前。
V	Int32	語彙の単語数。
各単語について (V 回)		語幹 (文字列)、形 (文字列)、頻度 (Int32)、IDF (Single)、埋め込み (ベクトル)。
N	Int32	文の数。
各文について (N 回)		テキスト (文字列)、項の数 n (Int32)、続いて id (Int32) + 重み (Single) のペアが n 個、最後に密ベクトル (ベクトル)。
ベクトル	Int32 + Single…	長さの後に値が続きます。長さ 0 = ベクトルなし。

Pinocchio の場合、ファイルのサイズは約 2.8 MB です。共起行列と PPMI 行列は保存されません。これらは学習中にのみ必要です。回答するには、語彙、IDF、埋め込み、文のベクトルがあれば十分です。

8. パラメーターと実験

8.1 学習パラメーター

パラメーター	既定値	効果
Finestra contesto ± (文脈ウィンドウ ±)	4 (1–10)	左右それぞれ何語までを文脈として数えるか。狭いウィンドウ: より文法的な類似性。広いウィンドウ: よりテーマに関連した類似性。
Dimensione vettori (ベクトルのサイズ)	128 (16–512)	埋め込みの長さ。数値が多いほど射影は忠実になりますが、メモリと計算量が増えます。
Frequenza minima (最小頻度)	2 (1–20)	このしきい値を下回る単語は埋め込みを受け取りません。値が大きいほど単語は少なくなり、ベクトルの信頼性は高くなります。
Seme	12345	乱数ジェネレーターのシード (コードからのみ変更可能)。変更するとベクトルは変わりますが、平均的な品質は変わりません。

8.2 回答パラメーター

パラメーター	既定値	効果
Peso semantico (意味的重み)	0,30	0 = 同じ単語のみ (語彙的)、1 = 埋め込みの類似度のみ。
Peso contesto (文脈の重み)	0,25	前後 2 つの文がどれだけ影響するか。
Parole simili per termine (語あたりの類似語数)	3	単語ごとに質問に追加される単語の数。0 = 拡張なし。
Soglia somiglianza (類似度しきい値)	0,35	類似語が追加されるために必要な最小類似度。
Regole tipo domanda (質問の種類のルール)	有効	chi/dove/quando/quanto/perché/come のボーナスを有効または無効にします。

パラメーター	既定値	効果
FrasiContesto、SogliaIncertezza	1; 0,12	コードからのみ変更可能: 回答の前後に表示する文の数と、回答を不確かと判定するしきい値。

8.3 Pinocchio を使ったガイド付き実験

1. *Chi ha costruito il burattino?* を、**Peso semantico** を 0、次に 0.30、次に 1 にして試します。0 と 0.30 では回答は「— Io sono un burattino.」(僕はあやつり人形です) になり、1 では「Nascono burattini, vivono burattini e muoiono burattini.」(あやつり人形として生まれ、あやつり人形として生き、あやつり人形として死ぬ) になります。正確な単語ではなく、テーマだけが重要になるのです。
2. *Chi ha mangiato Geppetto?* (誰が Geppetto を食べたか?) 回答は「— E chi te li ha mangiati?」(じゃあ誰がそれを食べたの?) です。このモデルは「バッグ・オブ・ワーズ」です。食べる側と食べられる側を区別せず、単語の順序は考慮されません。
3. *Che cosa vende Geppetto per comprare l'abbecedario?* (Geppetto は読み方の教本を買うために何を売るか?) は、第 VIII 章のタイトル「Geppetto rifà i piedi a Pinocchio, e vende la propria casacca per comprargli l'Abbecedario」を見つけます。拡張によって、「comprato」の類似語として「casacca」(上着) が追加されます。トレースでは記号 ~ で確認できます。
4. **Parole simili per termine** を 0 に設定して前の質問を繰り返します。「casacca」がないと正しい文は 4 位に下がり、回答は「— E perchè mi avete comprato?」になります。
5. Esplora タブで `fata`、`geppetto`、`naso` を検索します。類似語の多くは同義語ではなく、物語の中で近くに現れる単語にすぎません (「naso」→「giunti」、「spropositato」、「allungo」)。本が 1 冊だけでは、統計は貧弱です。
6. **Finestra** を 1、次に 8 にして学習し直し、`naso` の類似語を比較します。ウィンドウ 8 では「spropositato」、「allungo」、「punta」といった、同じ場面の単語が現れます。
7. *astronave* (宇宙船) と入力します。プログラムは、どの単語も知らないと回答します。読み込んだテキスト以外のことは何も知らず、そのことを明言します。

8. **Regole tipo domanda** を無効にして *Chi ha costruito il burattino?* を繰り返します。ボーナスがないと、回答は「Io sono un burattino.」(ダッシュなし) になります。ルールは手作業で書かれた唯一の知識であり、それ以外はすべてデータから得られています。
9. 別のテキスト、たとえば技術マニュアルや社内手順集などの `.txt` 形式のファイルで学習し、その内容について質問してみてください。プログラムは Pinocchio に依存していません。

9. 制限と拡張

9.1 意図的な制限

- ステマーは粗く、イタリア語専用に設計されています。ストップ ワードもイタリア語です。
- 単語の順序は無視されます (「バッグ・オブ・ワーズ」モデル)。主語と目的語は区別されません。
- 否定 (「non」) はストップ ワードです。「è vero」(本当だ) と「non è vero」(本当ではない) は同じものとして扱われます。
- 埋め込みは 1 つのテキストだけから得られるため、単語の一般的な意味ではなく、そのテキストにおける近さを反映しています。
- 検索では質問をすべての文と比較します。数千の文であれば十分ですが、数百万の文には向きません。
- 回答は常に既存の文です。プログラムは要約を行わず、異なる文から得た情報を組み合わせることもしません。

9.2 おもちゃから実際の AI へ

観点	MiniAI	大規模言語モデル
データ	本 1 冊: 約 20,000 個の内容語	数兆語
テキストの単位	単語の語幹	固定語彙のトークン (単語の断片)
単語のベクトル	共起 + PPMI + ランダム射影、128 個の数値	ニューラル ネットワークで学習、数千個の数値
文脈	±4 語の固定ウィンドウ	数千トークンに対するアテンション機構 (順序も考慮)
学習される数値	約 600,000 個 (単語と文の埋め込み)	数十億のパラメーター
回答	既存の文を選択	新しいテキストを 1 語ずつ生成
学習	PC で 1 秒未満	数千台の専用プロセッサで数週間

違いは非常に大きいものの、本質は共通しています。文脈から学習したベクトルで言語を表し、類似度を測ることです。最新の RAG システムは、MiniAI がステップ B–E で行っていることを、はるかに大きな規模で行っています。質問をベクトルに変換し、最も近い文章を探して言語モデルに渡し、言語モデルがそれを使って回答を組み立てます。

9.3 考えられる拡張

- **文の生成**: 同じテキストで学習した単語 (またはバイグラム) のマルコフ連鎖を使い、選択から生成への一歩を確認します。
- **ベクトルの学習**: ランダム射影を小さな word2vec ネットワーク (ネガティブ サンプリング付き skip-gram) に置き換えます。これも外部ライブラリは使用しません。
- **テキストを増やす**: 複数のファイルをまとめて学習させます。これが最も効果的な改善です。
- **バイグラム**: 頻繁に連続する単語のペア (「fata turchina」) を語彙に追加し、順序の情報を少し保持します。
- **否定**: 「non」を破棄する代わりに、次の単語と結合します (「non_vero」)。
- **他の言語**: ストップ ワードと接尾辞を設定可能にします。
- **高速な検索**: 文の数が数十万になる場合は、語彙的スコアのための転置インデックス (単語 → 文) と、密ベクトルのための近似インデックスを使用します。

10. ライセンスと責任

MiniAI は、DevService.net のすべてのプログラムと同様に無料で配布されており、自己責任で使用するものです。これは研究用のプロジェクトであり、いかなる種類の保証もなく、現状のまま提供されます。ソース コードは、学習や実験のために閲覧、変更、使用することができます。

テキスト `Pinocchio.txt` (『*Le avventure di Pinocchio*』、Carlo Collodi、1883 年) はパブリック ドメインであり、Project Gutenberg (www.gutenberg.org) から入手したものです。Project Gutenberg のライセンスはファイル自体に含まれています。

不具合の報告やご提案には、DevService.net サイトのお問い合わせフォームをご利用ください。