Transofmer in politica: Fine-tuning di (Ro)BERT(a) per un task di *Politic Ideology Detection*

Greta Gorzoni
greta.gorzoni@studio.unibo.it

Francesca Schiavone
francesca.schiavon10@studio.unibo.it

26 giugno 2025

1 Introduzione

Il presente progetto parte dal proposito di operare un fine-tuning su un modello BERT pre-addestrato, adattando uno script per problemi di *Text Classification* in lingua inglese sul benchmark GLUE a uno specifico task e relativo dataset in lingua italiana. Nello scegliere il task e il dataset con cui lavorare, sono state passate in rassegna le ultime campagne di EVALITA e si è deciso di lavorare su un task di *Politic Ideology Detection*.
Sono stati testati diversi modelli al fine di trovare quello più adatto, sono state apportate modifiche allo script *run_glue_no_trainer.py* e sono stati modificati gli iperparametri in virtù dei limiti e delle problematiche riscontrate nei primi tentativi di addestramento, arrivando alla configurazione presentata nel seguente report che ha raggiunto un'accuratezza del 78% nel *validation set* e del 70% nel *test set*.

Il report è così suddiviso: nella sezione 2 è analizzata la struttura del Dataset scelto; successivamente, nella sezione 3 sono passati in rassegna i dettagli tecnici del fine-tuning: dal preprocessing del dataset, alle modifiche effettuate sullo script *run_glue_no_trainer.py*, alla scelta degli iperparametri. Infine, la valutazione del modello.

Il dataset utilizzato e i codici impiegati sono pubblicamente accessibili nella repository GitHub `MNNLP_project`.

2 PoliticIT EVALITA 2023

La scelta del dataset è ricaduta su PoliticIT (EVALITA 2023). Il task principale di PoliticIT è l'estrazione del posizionamento politico a partire da messaggi testuali pubblicati su piattaforme social da esponenti politici italiani. Il dataset è collezionato a partire da tweet di parlamentari italiani, raccolti e anonimizzati. Le etichette associate non sono state annotate soggettivamente, ma sono state assegnate sulla base delle informazioni pubbliche disponibili (appartenenza partitica).

Il task si articola in tre sotto-compiti:

- **Subtask A** – Gender: classificazione.
- **Subtask B** – Political Ideology (binary): classificazione binaria del posizionamento politico in Left o Right.
- **Subtask C** – Political Ideology (multi-class): classificazione multiclassificata in quattro etichette (Left, Moderate-Left, Moderate-Right, Right).

In questo lavoro abbiamo deciso di concentrarci sul Subtask B, ovvero l'identificazione automatica del posizionamento politico in classificazione binaria.

2.1 Dataset

Il dataset messo a disposizione per la competizione EVALITA 2023 rientra nella macro-area dei task di *Autorship Attribution*. Si tratta di messaggi tweet raccolti tra il 2020 e il 2022. Il dataset contiene 139794 istanze ed è reso disponibile in file .csv divisi in *training* e *test set*. Le colonne presenti nel dataset sono le seguenti:

- **label** = ID del tweet;
- **gender** = genere dell'autore $self - assigned$;
- **ideology_binary** = posizionamento politico binario (*Left, Right*);
- **ideology_multiclass** = posizionamento politico multiclassificato (*Left, Moderate-Left, Moderate-Right, Right*);
- **tweet** = testo del tweet.

In questo lavoro sono state utilizzate le colonne *tweet* e *ideology_binary*, quest'ultima come colonna target per il task di *Political Ideology Detection*.

3 IL fine-tuning: da BERT a RoBERTa

In questa sezione è descritta la pipeline adottata per il fine-tuning di un modello pre-addestrato. In primo luogo vengono descritte le operazioni di pre-processing fatte sul dataset, successivamente viene analizzato lo script *run_glue_no_trainer.py* e gli adattamenti fatti su di esso per il presente progetto. Sono poi illustrati gli iperparametri selezionati e infine vengono presentati i risultati ottenuti e confrontati con il benchmark riportato sulla pagina ufficiale della competizione.

3.1 Splitting del data set

Il dataset sopra descritto si presenta suddiviso tra *apprendimento* e *test set*. Abbiamo proceduto a suddividere i dati dell'apprendimento in *training set* 90% e *validation set* 10% grazie all'ausilio della libreria *sklearn*, mantenendo invariato il *test set*. Questa scelta consente di valutare le prestazioni del modello durante la fase di addestramento, potendo dunque monitorare l'overfitting, ottimizzare gli iperparametri e avere quindi uno strumento di controllo. Il dataset risulta composto dal numero di istanze mostrate in tabella 1.

Dataset Split	Numero di istanze	Percentuale
Training Set	93.477	90%
Validation Set	10.362	10%
Test Set	36.167	—

Tabella 1: Suddivisione del dataset tra training, validation e test set.

3.2 *run_glue_no_trainer.py*

run_glue_no_trainer.py è uno script messo a disposizione da Hugging Face come esempio di fine-tuning di un modello pre-addestrato per un task di *text classification*. La differenza principale nella versione *no_trainer* è rendere visibile e modificabile il processo di apprendimento. Infatti, lo script

run_glue_no_trainer.py, pensato per prendere un modello pre-addestrato dalla libreria *transformer* e fare il fine-tuning sul dataset GLUE, è facilmente adattabile e personalizzabile.

3.3 Adattamento del dataset

Una delle prime modifiche apportate allo script riguarda la fase di caricamento dei dati. Lo script originale prevedeva l'utilizzo diretto di un dataset da Hugging Face tramite la libreria `datasets`, presupponendo una struttura già conforme (in particolare la presenza di una colonna denominata `label`). Tuttavia, nel nostro caso i dati sono forniti in formato `.csv` e le etichette binarie del posizionamento politico sono contenute nella colonna `ideology_binary`.

Per risolvere l'incompatibilità e prevenire l'errore `"Column 'label' not in the dataset"`, abbiamo caricato manualmente i dati con la libreria `pandas`, rinominando le colonne in modo che corrispondessero a quelle attese dallo script. I dati sono stati poi convertiti in oggetti `DatasetDict` della libreria `datasets` per renderli compatibili con il flusso previsto. Di seguito il codice implementato:

```
308 train_df = pd.read_csv(args.train_file)
309 valid_df = pd.read_csv(args.validation_file)
310
311 train_df = train_df[['tweet', 'ideology_binary'
      ]].rename(columns={"tweet": "text", "
      ideology_binary": "label"})
312 valid_df = valid_df[['tweet', 'ideology_binary'
      ]].rename(columns={"tweet": "text", "
      ideology_binary": "label"})
313
314 raw_datasets = DatasetDict({
315 "train": Dataset.from_pandas(train_df),
316 "validation": Dataset.from_pandas(valid_df)})
```

In questo modo, il formato dei dati rispetta le aspettative dello script originale, che richiede colonne denominate `text` e `label` per poter eseguire il preprocessing e il fine-tuning del modello.

3.4 Tokenizzazione

I testi in input vengono preprocessati tramite un tokenizzatore specifico per il modello scelto, caricato automaticamente da HuggingFace.

Il tokenizzatore viene inizializzato qui:

```
363     tokenizer = AutoTokenizer.from_pretrained(
364         args.model_name_or_path, use_fast=not
               args.use_slow_tokenizer,
               trust_remote_code=args.
               trust_remote_code
365     )
```

Qui avviene la tokenizzazione vera e propria :

```
436     result = tokenizer(
437         texts if sentence2_key is None else
               texts,
438         padding=padding,
439         max_length=args.max_length,
440         truncation=True
441         )
```

Il codice converte il testo in una forma che il modello può comprendere (cioè, una sequenza di numeri, ciascuno rappresentante un *token* nel vocabolario del modello). Se la sequenza di testo è più lunga di max_length, verrà troncata. Se è più corta, verrà completata con padding.

3.5 Stampa Loss Function

Un'altra utile modifica allo script riguarda la **Loss**: infatti, dopo diversi tentativi in cui si è riscontrato che il modello andava verso l'*overfitting* si è deciso di aggiungere un comando che permettesse di monitorare la discesa della Loss:

```
611     if step % 100 == 0:
612         accelerator.print(f"Step {step} - Loss:
               {loss.item()}")
```

Questo consente di avere una visione chiara dell'evoluzione del modello, facilitando l'individuazione di eventuali punti critici nel processo di ottimizzazione.

3.6 Iperparametri

```
1 !python run_glue_no_trainer.py \
2   --model_name_or_path FacebookAI/xlm-roberta-
       base \
3   --train_file politicIT_train.csv \
4   --validation_file politicIT_validation.csv \
5   --max_length 256 \
6   --per_device_train_batch_size 64 \
7   --learning_rate 3e-5 \
8   --num_train_epochs 4 \
9   --output_dir output_roberta
```

Qui di seguito una rassegna dettagliata degli iperparametri scelti.

3.6.1 Il modello

La scelta iniziale del modello pre-addestrato per il fine-tuning è stata *dbmdz/bert-base-italian-cased*. Si tratta di un modello basato sull'architettura BERT BASE, in una versione con un addestramento specifico sulla lingua italiana. Il fatto che le istanze da analizzare fossero testi provenienti da Twitter, dove spesso il linguaggio è più informale, ci ha orientate verso un modello *uncased*: con il modello *cased* l'accuratezza non ha mai superato il 73% sul *validation set*, con il modello *uncased*, invece, le performance sono salite; tuttavia, poiché non ancora soddisfatti dei risultati (in particolare della metrica *recall*, *cfr. infra*), abbiamo passato in rassegna altri modelli

pre-addestrati sull'italiano o multilingue e abbiamo optato, in seguito a una serie di ricerche, per **RoBERTa** poiché il suo addestramento su una quantità maggiore di dati non filtrati e realistici, la rende più robusta anche in domini rumorosi come i social network. Inoltre, la rimozione del task *next sentence prediction* (NSP), che la caratterizza, ha permesso l'ottenimento, in diverse occasioni, di performance migliori. Abbiamo tentato di utilizzare *XLM-RoBERTa-large*, ma la memoria della GPU utilizzata non era sufficiente per 561M parametri (error: *out of memory*). Si è quindi deciso di utilizzare un modello di RoBERTa più leggero: *xlm-roberta-base*.

Di seguito, nella tabella un confronto tra il modello BERT utilizzato nei primi tentativi e il modello RoBERTa utilizzato nello script finale.

	BERT (bert-italian-uncased)	**XLM-RoBERTa (xlm-roberta-base)**
Case sensitivity	Uncased	Cased
Numero di layer	12	12
Numero di parametri	111M	279M
Dimensione embedding	768	768
Testi pretraining	BooksCorpus + Wikipedia (3.3B tokens)	CommonCrawl (2.5TB testo)
Training objective	MLM + NSP	Solo MLM
Tokenizzazione spazi/punteggiatura	Sensibile agli spazi	Più robusta (lessico multilingue)
Linguaggio informale (es. tweet)	Debole (dati puliti)	Alta robustezza (dati rumorosi)

Tabella 2: Confronto tra i modelli *bert-italian-uncased* e *xlm-roberta-base*.

3.6.2 La frequenza ha un peso: *Cross Entropy Loss Function*

Nei primi tentativi di addestramento si è notata una tendenza del modello a predire la classe maggioritaria nella fase di *testing*. Abbiamo infatti osservato una maggiore capacità nel riconoscere la classe maggioritaria (recall di 0.80 per *left* contro 0.55 per *right*), a discapito della classe minoritaria.
Abbiamo ipotizzato che ciò dipendesse da uno squilibrio del dataset, infatti le istanze abbinate a etichetta *left* sono maggiori rispetto a quelle con etichetta *right*. Questa situazione può portare il modello a prediligere sistematicamente la classe maggioritaria, riducendo l'efficacia complessiva nella classificazione, in particolare della classe minoritaria.

Al fine di risolvere questa problematica riscontrata abbiamo apportato modifiche all'iperparametro *Loss function*. La funzione standard implementata (*Cross Entropy Loss*) permette di applicare pesi inversamente proporzionali alla frequenza delle classi. Questo evita che il modello, per minimizzare la loss complessiva, tenda a favorire la classe più frequente.
L'inclusione nella funzione di loss dei pesi specifici per ciascuna classe (*Weighted Cross Entropy Loss*), calcolati sulla base della distribuzione osservata nei dati di addestramento, è rappresentata da questi passaggi di codice nello script:

```
323     # Calcolo della distribuzione delle
           etichette nel training set
324     class_counts = train_df['label'].
           value_counts().sort_index().to_numpy()
325     total_samples = sum(class_counts)
326
327     # Calcolo pesi inversamente proporzionali
           alla frequenza
328     class_weights = total_samples / (len(
           class_counts) * class_counts)
329     class_weights = torch.tensor(class_weights,
           dtype=torch.float)
330
331     print("Class weights:", class_weights)
```

I pesi ottenturi sono: `tensor([0.9014, 1.1228])`

```
375     loss_fn = CrossEntropyLoss(weight=
           class_weights.to(accelerator.device))
```

All'interno di model.train():

```
611     loss = loss_fn(outputs.logits, batch["labels
           "])
```

Questa strategia garantisce che gli errori sulle classi meno rappresentate (in questo caso, la classe "right") abbiano un impatto maggiore sul valore della Loss, forzando il modello a trattare equamente entrambe le classi durante l'addestramento, nonostante lo sbilanciamento delle istanze.

Per osservare e mostrare il comportamento della funzione di Loss al netto di queste modifiche abbiamo deciso di realizzare un grafico che comparasse

le due Loss; come si può osservare nella Figura 1, infatti, l'andamento delle due loss nel *training* è differente: la *Cross Entropy Loss* è visibilmente più bassa e più stabile, mentre la *Weighted Cross Entropy Loss* decresce più lentamente e mostra maggiore variabilità. Tuttavia, sia nella fase di addestramento che in quella finale di valutazione, quest'ultima ha mostrato migliori performance: ha evitato l'*overfitting* che invece sembra esserci nel modello che utilizza la *Cross Entropy Loss* standard (nell'ultima epoca l'accuratezza è scesa ed è risalita la *Loss*). L'andamento della Loss suggerisce che il modello con *Weighted CE* non converge rapidamente verso un minimo locale facilmente accessibile come nel caso della *CE standard*, ma esplora una funzione più complessa, portando a una discesa leggermente più lenta e instabile della Loss, ma più robusta e efficace.
In conclusione, i pesi introdotti nella *Weighted Loss* hanno aiutato il modello a trattare meglio le classi sbilanciate: siamo passati da una recall 0.80 per *left* contro 0.55 per *right* ad un risultato più bilanciato: 0.72 *left* contro 0.67 *right*. (I dati dell'evaluation sul test set qui citati sono mostrati in dettaglio nella sezione 3.7).

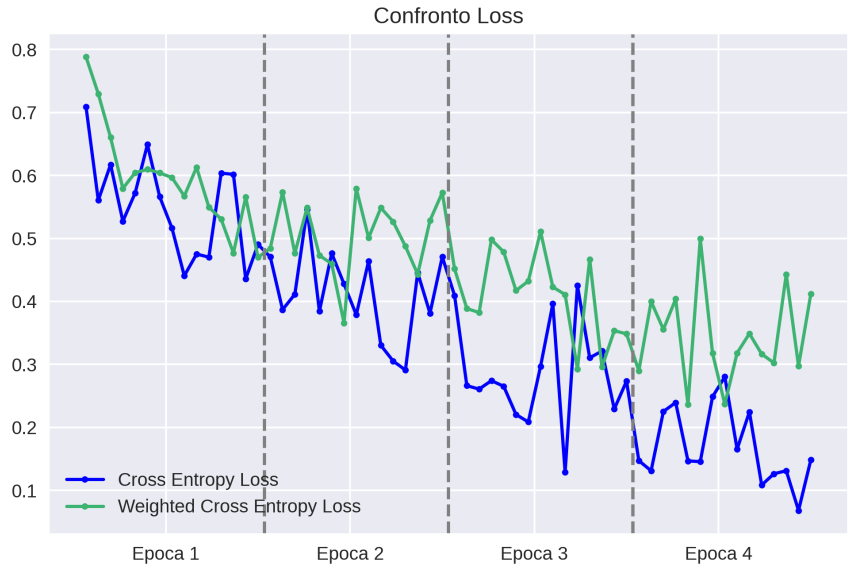

Figura 1: Confronto tra le due loss: *Cross Entropy Loss* in blu, *Weighted Cross Entropy Loss* in verde.

3.6.3 Il passo del modello: riflessioni sul *Learning Rate*

Nel corso dell'esperimento sono state testate diverse configurazioni, con diversi valori di *learing rate*: tra cui $5 \cdot 10^{-6}$, $5 \cdot 10^{-5}$, $3 \cdot 10^{-5}$ e $1 \cdot 10^{-5}$. L'iperparametro del *learning rate* determina quanto velocemente il modello aggiorna i suoi pesi durante l'ottimizzazione, in altre parole, quanto grande è il passo che il modello deve compiere lungo la direzione del gradiente al fine di minimizzare la funzione di loss. Dopo un primo test con un *learning rate* eccessivamente alto ($5 \cdot 10^{-5}$), che ha mostrato performance basse, con una discesa troppo rapida della loss e un'instabilità nel training, nelle configurazioni successive testate, è stato osservato che un *learning rate* troppo basso ($5 \cdot 10^{-6}$) ugualmente portava a performance insoddisfacenti: in particolare, l'impossibilità di testare il modello su un numero alto di epoche (*cfr. infra*), necessarie per un training con un *learning rate* basso, ci ha fatto optare per l'uso di valori intermedi. Le performance migliori sul validation set, infatti, si sono registrate con un learning rate pari a $\mathbf{3 \times 10^{-5}}$, che ha rappresentato un buon compromesso tra stabilità e rapidità di apprendimento. L'efficacia dipende anche dalla dimensione del batch e dal numero di epoche, altri iperparametri con cui è stato opportunamente bilanciato nell'esplorazione delle configurazioni.

3.6.4 Altri iperparametri rilevanti nella configurazione finale:

- ***max_length* = 256**
  Si è deciso di aumentare la lunghezza massima delle sequenze da 128 a 256 token, permettendo di catturare una porzione maggiore del contenuto testuale, migliorando la rappresentazione semantica e riducendo la perdita di informazione dovuta al troncamento.

- ***batch_size* = 64**
  È stato aumentato da 32 a 64, in quanto rappresenta il massimo che la GPU a disposizione (12 GB) è riuscita a sostenere senza incorrere in errori di memoria.

- ***num_train_epochs* = 4**
  Dopo aver sperimentato diverse configurazioni il numero di epoche è stato fissato a **4**. Questo valore si è rivelato un buon compromesso tra accuratezza e rischio di overfitting. Inoltre, il limite di tempo di utilizzo della GPU (12 GB) ha rappresentato un ulteriore vincolo pratico nella scelta.

- ***optimizer* = AdamW**
  Vvariante dell'ottimizzatore Adam che permette di gestire Weight Decay
- ***weight_decay* = 0.05**
  Penalizza i pesi grandi nel modello, per evitare l'overfitting

3.7 Evaluation

Per la valutazione del modello è stato creato uno script ulteriore che prende i dati di output generati nella fase di training, fa le predizioni sul *test_set* e restituisce alcune metriche. È possibile consultare la struttura dello script nella cartella *src* sotto la voce *evaluate_model.py* nella repository GitHub sopra citata.

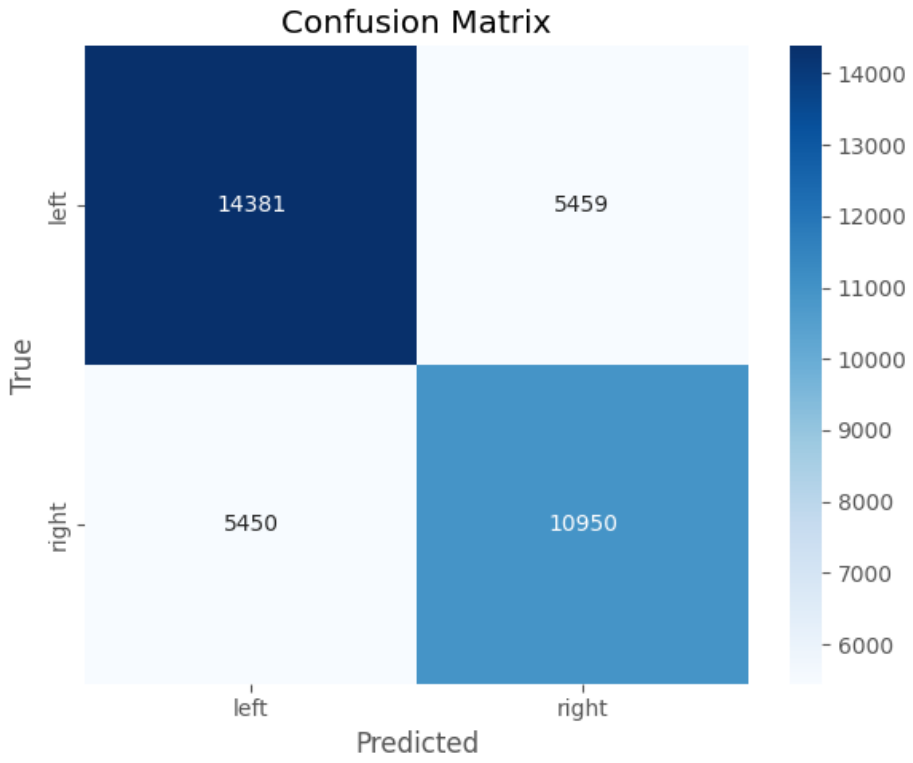

Figura 2: Matrice di Confusione

È stata anche realizzata una matrice di confusione *Figura* 2 che permette la visualizzazione dei falsi positivi FP, falsi negativi FN, veri positivi TP e veri negativi TN per le etichette *left* e *right*. Le metriche di *accuracy*, *recall*, *precision* e *F1_score* sono calcolate a partire dai valori contenuti nella matrice (Tabella 3).

Metrica	**Formula**
Accuracy	$\frac{TP + TN}{TP + TN + FP + FN}$
Precision	$\frac{TP}{TP + FP}$
Recall	$\frac{TP}{TP + FN}$
F1-score	$2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

Tabella 3: Metriche di classificazione

Per quest'ultima fase è stata utilizzata la libreria *sklearn* e i risultati ottenuti sono riassunti nella Tabelle 4 e 5.
Il modello ha ottenuto un'accuratezza del 78% nella fase di validazione, e del 70% nella fase di valutazione . Come accennato in precedenza, la metrica di *recall* si è bilanciata grazie alla *Weight Cross Entropy Loss Function* rispetto ai risultati ottenuti invece con la *Cross Entropy Loss* standard.

Nonostante i miglioramenti nelle performance, ottenuti con le modifiche fatte allo script *run_glue_no_trainer.py* e agli iperparametri, i risultati delle performance con il *fine-tuning* da noi realizzato non sono leggermente distanti a EVALITA 2023, ma confrontabili.

Metrica	**Valore**
Accuracy	0.6990
Precision	0.6990
Recall	0.6990
F1-score	0.6990

Tabella 4: Metriche aggregate sul *test set*

Classe	Precision	Recall	F1-score	Supporto
`left`	0.73	0.72	0.73	19840
`right`	0.67	0.67	0.67	16400
Totali				
Accuracy			0.70	36240
Macro avg	0.70	0.70	0.70	36240
Weighted avg	0.70	0.70	0.70	36240

Tabella 5: Metriche per classe sul *test set*

4 Limitazioni e prospettive

Questo progetto è stato realizzato grazie all'utilizzo delle unità di calcolo T4 GPU messe gratuitamente a disposizione da Google. Tuttavia, l'accesso a queste risorse è soggetto a limitazioni giornaliere, che rappresentano uno dei principali vincoli del lavoro: esse riducono infatti la possibilità di eseguire il modello con continuità e limitano la dimensione del modello pre-addestrato che è possibile utilizzare.

In prospettiva, diversi interventi potrebbero contribuire a migliorare le prestazioni del sistema. Tra questi, l'ampliamento del dataset, in particolare con un miglior bilanciamento tra le classi, rappresenta un passaggio cruciale per aumentare la robustezza del modello. Inoltre, l'impiego di risorse computazionali più avanzate e stabili consentirebbe di estendere il numero di esperimenti, esplorare con maggiore sistematicità gli iperparametri e sperimentare architetture più complesse.

5 Bibliografia e sitografia

- Russo, D., Jiménez-Zafra, S. M., García-Díaz, J. A., Caselli, T., Guerini, M., Alfonso Ureña-López, L., Valencia-García, R. (2023). PoliticIT at EVALITA 2023: Overview of the Political Ideology Detection in Italian Texts Task. In M. Lai, S. Menini, M. Polignano, V. Russo, R. Sprugnoli, G. Venturi (Eds.), Proceedings of the Eighth Evaluation Campaign of Natural Language Processing and Speech Tools for Italian. Final Workshop (EVALITA 2023) (CEUR Workshop Proceedings; Vol. 3473). CEUR Workshop Proceedings (CEUR-WS.org).

- García-Díaz, J. A., Colomo-Palacios, R., Valencia-García, R. (2022). Psychographic traits identification based on political ideology: An author analysis study on Spanish politicians' tweets posted in 2020. Future Generation Computer Systems, 130(1), 59-74.

- García-Díaz, J. A., Jiménez Zafra, S. M., Martín Valdivia, M. T., García-Sánchez, F., Ureña López, L. A., Valencia García, R. (2022). Overview of PoliticEs 2022: Spanish Author Profiling for Political Ideology. Procesamiento del Lenguaje Natural, 69, 265-272.

- Task EVALITA 2023 PoliticIT – Political Ideology Detection in Italian Texts (D. Russo, S.M. Jiménez-Zafra, J.A. García-Díaz, T. Caselli, M. Guerini, L.A. Ureña-López, R. Valencia-García)
https://codalab.lisn.upsaclay.fr/competitions/8507

- run_glue_no_trainer.py
https://github.com/huggingface/transformers/blob/main/examples/pytorch/text-classification/run_glue_no_trainer.py

- XLM-RoBERTa (base-sized model)
https://huggingface.co/FacebookAI/xlm-roberta-base

- dbmdz BERT-base-italian-uncased
https://huggingface.co/dbmdz/bert-base-italian-uncased