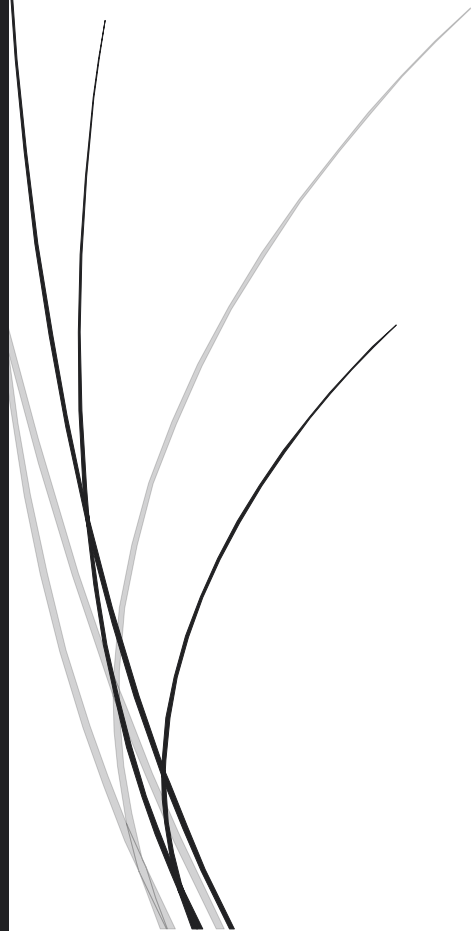
Progetto Icon

Classificazione del bias politico

Algoritmi di classificazione del documento su bias politico.



Gruppo di lavoro: Federico Troisi, 774469,
f.troisi@studenti.uniba.it
https://github.com/Naahbi/Progetto_ICon_Troisi.git

Indice

Sommario

Introduzione

Obiettivi del progetto

L'obiettivo del progetto è analizzare il contenuto testuale di un documento, al fine di identificarne l'orientamento politico.

Si vuole verificare se la forma e la struttura del testo esprimano un bias riconducibile a una posizione politica di sinistra, destra o invece ha un orientamento più neutrale, di centro.

A tale scopo, il progetto propone un sistema di classificazione automatica in grado di assegnare ciascun testo a una delle tre categorie politiche, vagliando più metodologie, tra cui tecniche di machine learning supervisionato, knowledge base semantica e reti Bayesiane.

Contesto e motivazione

L'idea dietro il progetto è quella di fornire uno strumento valido, in grado di valutare l'imparzialità e l'affidabilità di una fonte informativa, permettendo al fruitore di un servizio mediatico di poter accedere ad un contenuto in piena consapevolezza.

Si rende infatti necessario uno strumento simile, vista la crescente polarizzazione dei contenuti mediatici, che passa spesso inosservata a causa della difficoltà di notare eventuali bias al momento di una prima lettura.

Elenco argomenti di interesse

Capitolo: 'Modelli di Machine Learning Supervisionato'. Tratta l'apprendimento supervisionato, con riferimento alla regressione lineare, agli alberi decisionali e alle tecniche di ensemble learning, soffermandosi su addestramento e valutazione dei modelli.

Capitolo: 'Knowledge Base'. Tratta il concetto di knowledge base, la sua costruzione e l'utilizzo di prolog per interrogare attraverso query la base di conoscenza.

Capitolo: 'Reti Bayesiane'. Tratta il ragionamento e l'apprendimento in caso di incertezza, soffermandosi sulla costruzione di una belief network per il task di classificazione, valutando i risultati ottenuti.

Tecnologie e strumenti utilizzati

Per risolvere questo task si è passati per diversi tentativi, vagliando metodi e tecnologie diverse, valutando tra svariate possibilità.

Si è prima reso necessario individuare un dataset valido, che contenesse gli articoli già etichettati per categoria.
Il dataset scelto, vista la disponibilità ampia di articoli e il buon lavoro di etichettatura già fatta, è il '***PoliticalBias***[1]', di proprietà della compagnia ***Valurank***[2].
Il dataset è formato da tre colonne, 'sinistra', 'destra', 'centro'. Per ogni riga avremo tre articoli che trattano il medesimo argomento, ma su testate giornalistiche diverse e con diversa etichetta.

Utilizzando la libreria ***Newspaper3K***[3] è stato possibili scaricare la maggioranza degli articoli, mantenendo sempre 3 fonti diverse per ogni notizia, correttamente etichettate.

[1] PoliticalBias, https://huggingface.co/datasets/valurank/PoliticalBias/tree/main .
[2] Valurank, https://huggingface.co/valurank .
[3] Newspaper3k, Libreria per analisi e gestione di articoli online, https://newspaper.readthedocs.io/en/latest/ .

A quel punto si è utilizzato una tecnica di pre-processing e vettorizzazione dei documenti, calcolando matrici sparse TF-IDF[4], e si è passati al training degli algoritmi.

Prima sono stati allenati e valutati gli algoritmi di Machine Learning Supervisionato, dividendo il dataset in training e testing e usando diverse metriche per valutarne l'efficacia.

Si è poi passati alla costruzione di una knowledge base, che è poi salvata su un file leggibile in Prolog[5], assieme ad un secondo file contenente le rules, così da rendere possibile l'interrogazione sempre in linguaggio Prolog[5].
Anche l'efficacia della knowledge base è stata valutata.

Per concludere si è allenata una belief network sulla dipendenza tra termini di un documento e classe del documento, permettendo di fare inferenza su documenti nuovi, potendoli così classificare.

Preparazione del dataset

Per poter addestrare gli algoritmi di Supervised Machine Learning e costruire la Belief Network, si è resa indispensabile la preparazione del dataset per queste task, passando da un file **csv** contenente unicamente i link agli articoli, precedentemente etichettati, ad un file **csv** che salvi gli articoli come testo.

Per poter ottenere questo dataset, si è fatto uso della libreria ***Newspaper3K***[3], che ha permesso il download di titolo e corpo dell'articolo. Si è poi salvati su file tutti gli articoli di cui era possibile il download, assicurandosi di installare una tripla di articoli (stesso argomento, articoli e classi diverse) unicamente se interamente disponibile, per evitare dataset sbilanciati.

[4] TF-IDF, Term Frequency-Inverse Document Frequency
[5] Prolog, https://it.wikipedia.org/wiki/Prolog .

```python
def download_dataset(directory):
    df = pd.read_csv('resources/articles_link.csv')
    articles = []
    mapping = {'center': 0, 'left': 1, 'right': 2}
    for idx, row in tqdm(df.iterrows(), total=len(df), desc='Download Articles'):
        record = []
        error = False
        for bias in ['Left', 'Right', 'Center']:
            url = row[bias]
            try:
                article = Article(url)
                article.download()
                article.parse()

                if article.text and len(article.text) > 50:
                    record.append({
                        'bias': bias.lower(),
                        'url': url,
                        'text': article.text,
                        'title': article.title
                    })
            except:
                error = True
                print(f'Error downloading {url}')
                break
        if not error:
            for _ in record:
                articles.append()
    df = pd.DataFrame(articles)
    df['combined'] = df['title'].str.cat(df['text'], sep=' ', na_rep='')
    df['true_label_enc'] = df['bias'].map(mapping)
```

FUNZIONE PER IL DOWNLOAD DEL CONTENUTO DEGLI ARTICOLI E COSTRUZIONE DEL DATASET

Costruzione del dataframe

Una volta ottenuto il file **csv** contenente etichetta, titolo e corpo per ogni articolo, è stato possibile costruire una matrice TF-IDF[4], cioè una matrice che permette una rappresentazione numerica di un insieme di dati, dove ogni riga rappresenta un documento, ogni colonna un termine, e ogni valore indica quanto il termine è importante nel documento, tenendo conto della frequenza del documento (TF), e della sua rarità all'interno della collezione(IDF).

Nella matrice non sono previste eventuali stopwords[6], eliminate durante la creazione della matrice. Ogni termine è stato prontamente tokenizzato con tecniche di pre-processing del testo, cercando di ridurre il numero di termini, conservando la medesima informazione.

[6] Stopwords, parole dallo scarso valore informativo.

```python
def text_preprocessing(text):
    nlp = spacy.load('en_core_web_sm')
    doc = nlp(text.lower())
    # estraiamo i token eliminando punteggiatura e spazi bianchi
    tokens = [
        token.lemma_ for token in doc
        if not token.is_stop and not token.is_punct and not token.is_space
    ]
    return ' '.join(tokens)
```

METODO DI PRE-PROCESSING DEL TESTO

```python
def get_TF_IDF_matrix(dataframe):
    # Istanziamo un vectorizer, in grado di restituire una matrice di token TF-IDF, che usi la log(TF)
    vectorizer = TfidfVectorizer(
        preprocessor=text_preprocessing(),
        ngram_range=(1, 2),
        min_df=5,
        max_df=0.9,
        stop_words='english',
        sublinear_tf=True
    )

    matrix = vectorizer.fit_transform(dataframe['combined'])
    return matrix, vectorizer
```

METODO PER OTTENERE LA MATRICE TF-IDF

Passando il dataset completo alla funzione *get_TF_IDF_matrix*, si è ottenuta la sparse matrix, che viene ampiamente utilizzata per tutto il progetto.

Modelli di Machine Learning Supervisionato

Regressione Lineare

Descrizione del modello

La regressione lineare è un modello predittivo che stima una variabile target come una combinazione delle feature in ingresso: $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$. Attraverso l'addestramento di modello, si cerca di stimare i coefficienti dell'equazione affinché si riduca l'errore quadratico medio.
In classificazione multi-classe, come nel caso di progetto, si possono adottare due strategie. È possibile adottare una strategia one-vs-rest, cioè addestriamo tre modelli diversi in classificazione binaria, dove le due classi saranno [classe_n, altre_classi].
Adottando una tecnica di softmax regression, in cui per ogni classe si crea una funzione lineare. A quel punto si trasformano i valori di funzione trovati in un'unica formula $softmax(u_j) = \frac{exp(u_j)}{\sum^k exp(u_k)}$, che fornirà probabilità
positive.
Per default, il metodo della libreria python utilizzata, ***scikit-learn***[7], in caso di classificazione multi-classe utilizza la softmax regression.

```
def train_logistic_regression(matrix, dataframe, max_iter=1000):
    model = LogisticRegression(max_iter=max_iter)

    x_train, x_test, y_train, y_test = train_test_split( *arrays: matrix, dataframe['true_label_enc'], test_size=TEST_SIZE,
                                                        random_state=SEED)

    model.fit(x_train, y_train)

    return model
```

METODO CHE RESTITUISCE UN MODELLO DI REGRESSIONE LINEARE PER CLASSIFICAZIONE MULTI CLASSE

Il modello viene addestrato su una porzione di dataset, per questioni di riproducibilità e valutazione corretta, training (80%) e testing (20%) sets vengono fatti utilizzando il medesimo random_state, che assicura la stessa divisione del dataset.

Gli iperparametri del modello, cioè i parametri scelti al momento della creazione del modello che influenzano come il modello lavora e impara, sono:

- C, che influenza la ***regolarizzazione***[8]del modello**.**
- penalty, è il tipo di regolarizzazione, che può essere di tipo L1, absolute loss, e L2, squared loss.
- solver, algoritmo utilizzato da sklearn per trovare i pesi ottimali.
- random_state, necessario per la ripetibilità dei risultati.

Ottimizzazione dei parametri

Attraverso il modulo ***GridsearchCV***[9] è stato possibile provare diverse combinazioni di iperparametri.

[7] Scikit-learn, https://scikit-learn.org/stable/ .
[8] Regolarizzazione, tecnica utilizzata per evitare l'overfitting. Questa modifica la funzione da ottimizzare, funzione di loss, aggiungendo un termine che penalizza i modelli troppo complessi.
[9] GridSearchCV, https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html .

```python
def train_best_logistic_regression(matrix, dataframe, max_iter=1000):
    x_train, x_test, y_train, y_test = train_test_split( *arrays: matrix, dataframe['true_label_enc'], test_size=TEST_SIZE,
                                                        random_state=SEED)

    param_grid = {
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'penalty': ['l1', 'l2'],
        'solver': ['liblinear', 'lbfgs', 'saga']
    }

    grid_search = GridSearchCV(
        LogisticRegression(max_iter=max_iter),
        param_grid,
        cv=5,  # 5-fold cross validation
        scoring='accuracy',
        n_jobs=-1
    )

    grid_search.fit(x_train, y_train)

    best_params = grid_search.best_params_
    best_model = grid_search.best_estimator_

    return best_model, best_params
```

METODO PER TROVARE LA MIGLIOR COMBINAZIONE DI IPERPARAMETRI PER IL TASK DEL PROGETTO PER IL MODELLO DI REGRESSIONE LINEARE.

La miglior configurazione di parametri per il task del progetto risulta essere:

[C:10, penalty: L1, solver: saga]

Il solver 'saga' è un tipo di algoritmo ottimizzato per grandi datasaet, che supporta la regolarizzazione L1.

Alberi Decisionali

Descrizione del modello

Un albero decisionale è un modello predittivo, che utilizza una struttura ad albero per prendere decisioni basate sulle feature di input.

È costituito dai nodi interni, ognuno con una condizione, da cui si diramano i nodi figli, e dalle foglie, ciascuna etichettata con un ***point estimate***[10].

L'albero decisionale, con metodo top down, prende in input un set di condizioni, una funzione da ottimizzare, e un valore soglia, threshold. Si ricerca tra le condizioni quella che ottimizzi la funzione, la si seleziona come nodo e si prosegue iterativamente, in tal modo si arriva alle foglie, dove nessuna condizione è adatta per la divisione.
Qualora la funzione da ottimizzare fosse una funzione specifica, ***logloss***[11], allora parleremmo di ***Entropia***[12], e misureremmo la variazione di entropia data selezionando una condizione dal set. La

[10] Point estimate, predizione effettuata dal modello su un esempio specifico.
[11] Logloss, cross-entropy loss, metrica che valuta quanto valide sono le probabilità predette dal classificatore.
[12] Misura dell'incertezza associata alla distribuzione delle classi, dato un set di esempi, se fossero divisi in classi equamente distribuite avremmo massima incertezza, per questo si vuole ridurre l'entropia, cercando di ottenere massima purezza nei subset ottenuti, massimizzando l'information gain.

differenza di entropia è chiamata information gain, e si sceglie la condizione che massimizza l'information gain.

Gli iperparametri del decision tree sono:

- criterion, una funzione che valuta la qualità dello split, che può essere 'gini' o 'entropy'.
- max_depth, profondità massima dell'albero.
- min_samples_split, minimo numero di campioni per dividere un nodo.
- min_samples_leaf, minimo numero di campioni in una foglia.
- max_features, numero di feature da considerare per ogni split.
- random_state, necessario per la ripetibilità dei risultati.

```python
# Metodo che allena un DecisionTreeClassifier, restituendo il modello
def train_decision_tree_classifier(matrix, dataframe, max_depth=10):
    model = DecisionTreeClassifier(
        criterio='entropy',
        max_depth=max_depth,
        random_state=SEED
    )
    x_train, x_test, y_train, y_test = train_test_split( *arrays: matrix, dataframe['true_label_enc'], test_size=TEST_SIZE,
                                                        random_state=SEED)

    model.fit(x_train, y_train)

    return model
```

METODO CHE ADDESTRA UN ALBERO DECISIONALE PER LA CLASSIFICAZIONE MULTICLASSE

Ottimizzazione dei parametri

Attraverso il GridSearchSV si è ricercato la miglior combinazione di iperparametri, ritrovando la seguente:

[criterion:'gini', max_depth:10, min_samples_leaf:2, min_samples_split:2]

Il criterio 'gini' non valuta l'informazione guadagnata, quindi l'entropia dei samples, ma quanto un nodo è 'impuro', cioè quanto mischiate sono le classi all'interno di un nodo dell'albero decisionale.

XGBoost

Descrizione del modello

Per parlare del modello di ***XGBClassification***[13] è necessario introdurre il concetto di gradient boosting.

Il gradient boosting è un metodo di ***ensemble learning***[14]***,*** che utilizza modelli più semplici e deboli per ottenere una predizione più accurata, di solito si utilizzano alberi decisionali poco profondi. Utilizzando alberi decisionali, si parte da una predizione iniziale, si calcola di quanto il modello sbaglia, e si costruisce un nuovo albero per correggere l'errore. Si continua, unendo gli alberi iterativamente, e quindi correggente l'errore residuo dell'albero precedente.

XGBoost sta per eXtreme Gradient Boosting, poiché è un'implementazione ottimizzata e potenziata del classico algoritmo di gradient boosting.

I parametri del classificatore sono:

- n_estimators, il numero di alberi,

[13] XGBClassification, https://xgboost.readthedocs.io/en/latest/python/python_api.html .
[14] Ensable Learning, si combinano le predizioni di modelli semplici per ottenere una predizione nuova.

- objective, il tipo di task che si affronta, se classificazione multiclasse o binaria.
- Num_class, numero di classi per la classificazione multiclasse.
- learning_rate.
- max_depth, profondità degli alberi,
- random_state, necessario per la ripetibilità dei risultati.

```python
# Metodo che restituisce un modello di gradient boost allenanto
2 usages
def train_xgb_model(matrix, dataframe, n_estimators=100, max_depth=6, learning_rate=0.01):
    x_train, x_test, y_train, y_test = train_test_split( *arrays: matrix, dataframe['true_label_enc'], test_size=TEST_SIZE,
                                                        random_state=SEED)
    xgb_model = XGBClassifier(
        objective='multi:softmax',
        num_class=3,
        eval_metric='logloss',
        n_estimators=n_estimators,
        max_depth=max_depth,
        learning_rate=learning_rate,
        random_state=SEED
    )
    xgb_model.fit(x_train, y_train)

    return xgb_model
```

METODO CHE ADDESTRA UN MODELLO DI GRADIENT BOOSTING PER IL TASK DI CLASSIFICAZIONE MULTICLASSE

Nel caso del modello di gradient boosting, non si è andato avanti con la sperimentazione per la ricerca della miglior scelta di iperparametri, sicché computazionalmente e temporalmente dispendiosa per un progetto iniziale come questo.

Risultati e Analisi

Confrontando i migliori modelli addestrati per il task di classificazione multiclasse, mantenendo la medesima divisione tra training e testing set durante l'addestramento di ognuno dei modelli, possiamo notare come il XGBoost, anche senza la miglior combinazione di iperparametri, sia la scelta migliore tra i modelli per il task di classificazione che il progetto si è proposto.

Le metriche utilizzate per valutare e confrontare i modelli sono:

- accuracy, percentuale di predizioni corrette sul totale. Non è la metrica più adatta in caso di dataset sbilanciati, ma non è il caso del dataset scelto.
- f1 Macro, media armonica tra Precision e Recall. Questa metrica è più indicata in caso di dataset sbilanciati.
- Precision, percentuale di predizioni positive che sono corrette.
- Recall, percentuale di veri positivi che sono stati correttamente individuati.

Altre metriche di valutazione dei modelli non sono state scelte per una questione di difficoltà di integrazione col task proposto, e per l'impossibilità di fare un confronto tra modelli in determinati casi. Esempi di altre metriche sono:

- Area sottesa sotto la curva ROC, un grafico che mostra i falsi positivi durante la classificazione contro i veri positivi. In questo caso non è stata adottata poiché più indicata quando si lavora con task di classificazione binaria, o con modelli che restituiscono probabilità di appartenenza piuttosto che valori discreti.
- Log Loss, metrica più indicata in caso di modelli che restituiscono probabilità di appartenenza.

```
def print_model_classification_report(model, x_test, y_test):
    y_pred = model.predict(x_test)
    print(classification_report(y_pred, y_test))
```

FUNZIONE CHE STAMPA IL CLASSIFICATION REPORT DEL MODELLO PASSATO IN INPUT

Model	Accuracy	F1Macro	Precision	Recall
Regressione Lineare	0.65	0.65	0.66	0.65
Albero Decisionale	0.68	0.67	0.71	0.67
XGBoost	0.75	0.69	0.78	0.68

Knowledge Base

Concetti chiave e struttura della Knowledge Base

Una knowledge base è una collezione di fatti e regole, organizzati in un formato che consente l'inferenza e il ragionamento automatico.
Volendo dare una definizione concisa di knowledge base, parleremmo di insieme di proposizioni che sono vere.

I fatti rappresentano le informazioni base, vere per definizione all'interno della base di conoscenza.
Le regole sono invece condizioni logiche, che permettono di inferire nuove informazioni, partendo dai fatti.

Si possono distinguere due diverse tipologie di knowledge base, che si distinguono sul tipo dei fatti. Ci sono knowledge base booleane, qualora i fatti possano assumere un semplice valore di verità booleano (vero, falso), oppure con logica fuzzy, in caso si hanno fatti associati ad un valore numerico diverso dal valore di verità, che rappresenta la forza del legame tra fatto e concetto, nel caso del progetto, tra la parola e una classe politica.

La knowledge base costruita per il progetto adotta una logica fuzzy, ogni parola ha infatti un dato score per ogni classe, in tal modo è possibile fare inferenza su un testo sommando gli score dei termini che lo compongono.

Per la creazione della knowledge base ci si è affidati a python, costruendo i fact attraverso metodi python e realizzando funzioni per eseguire delle query.
Al tempo stesso, volendo utilizzare la knowledge base in ambiente Prolog piuttosto che python, il progetto è corredato di due file '.pl' che formano assieme la knowledge base.
Il primo file, 'facts.pl', viene costruito automaticamente in python, e crea per ogni termine un fatto con peso per classe.
Il secondo file, 'rules.pl', è stato realizzato manualmente, e contiene le regole logiche necessarie per poter interrogare la knowledge base.

Processo di costruzione

Per realizzare la Knowledge Base il primo problema era decidere come dovesse essere in grado di valutare un testo e classificarne l'orientamento politico.
Si è optato per un una forma di integrazione multi-modello, in cui si è estratto conoscenza da modelli supervisionati diversi per poter essere combinata in una knowledge base esplicita e interpretabile.

Sono stati addestrati tre modelli, quelli visti in precedenza, che sono regressione lineare, albero di decisione e XGBoost.
L'idea è quella di utilizzare i pesi che questi modelli danno alle feature per poter fare classificazione con la knowledge base.

Una volta scelta questa strategia, si è studiato il come attuarla, poiché la diversa natura dei modelli rappresenta un problema non indifferente durante l'integrazione. Infatti, mentre è possibile prendere dal modello di regressione lineare i coefficienti per ogni feature, e quindi un valore numerico che esplicitamente afferma quanto una feature è rilevante per la classificazione, alberi decisionali, e quindi anche XGBoost, hanno invece per ogni feature un unico valore numerico, che misura l'importanza della feature per la riduzione dell'entropia nel sub set di dati.

Preso atto della differente natura dietro i pesi dei modelli, e anche della necessità di avere un punteggio per ogni feature per ogni classe, si è optato per addestrare diversamente albero decisionale e XGBoost.
Si è infatti adottata una strategia One-Vs-Rest per ogni classe necessaria, in tal modo si sono addestrati tre alberi, uno per classe, e lo score di importanza per feature di ogni albero misurerà il valore di importanza della feature per la classe dell'albero.
Stessa strategia è stata adottata per il XGBoost.
I modelli sono stati addestrati su testi tokenizzati, quindi sulle matrici TF-IDF ottenute da un vectorizer.

```python
def get_weights_tree(matrix, dataframe, class_names, max_depth=10, min_samples_split=2, min_samples_leaf=2):
    x_train, x_test, y_train, y_test = train_test_split( *arrays: matrix, dataframe['true_label_enc'], test_size=TEST_SIZE,
                                                        random_state=SEED)
    label_map = {name: i for i, name in enumerate(class_names)}

    tree_weights_per_class = {}
    for cls in class_names:
        class_idx = label_map[cls]
        y_bin = (y_train == class_idx).astype(int)
        model = DecisionTreeClassifier(
            criterion='gini',
            max_depth=max_depth,
            random_state=SEED,
            min_samples_leaf=min_samples_leaf,
            min_samples_split=min_samples_split
        )
        model.fit(x_train, y_bin)
        tree_weights_per_class[cls] = model.feature_importances_
    return tree_weights_per_class
```

METODO CHE RESTITUISCE I PESI DI OGNI FEATURE PER LE TRE CLASSI DELLA CLASSIFICAZIONE, ADDESTRANDO 3 DIVERSI ALBERI CON STRATEGIA ONE-VS-REST

```python
def get_weights_xgboost(matrix, dataframe, class_names, n_estimators, max_depth, learning_rate):
    x_train, x_test, y_train, y_test = train_test_split( *arrays: matrix, dataframe['true_label_enc'], test_size=TEST_SIZE,
                                                        random_state=SEED)
    label_map = {name: i for i, name in enumerate(class_names)}
    xgb_weights_per_class = {}
    for cls in ['left', 'center', 'right']:
        class_idx = label_map[cls]
        y_bin = (y_train == class_idx).astype(int)

        xgb_model = XGBClassifier(
            objective='binary:logistic',
            eval_metric='logloss',
            n_estimators=n_estimators,
            max_depth=max_depth,
            learning_rate=learning_rate,
            random_state=SEED
        )
        xgb_model.fit(x_train, y_bin)

        xgb_weights_per_class[cls] = xgb_model.feature_importances_
    return xgb_weights_per_class
```

METODO CHE RESTITUISCE I PESI DI OGNI FEATURE PER LE TRE CLASSI DELLA CLASSIFICAZIONE, ADDESTRANDO 3 DIVERSI MODELLI DI XGBOOST CON STRATEGIA ONE-VS-REST

Una volta ottenuti questi pesi, si costruiscono delle matrici, che vengono poi normalizzate. A questo punto, si ottengono i pesi finali, combinando i pesi di ogni modello attraverso una media pesata. Si è infatti optato per dare più peso agli score del modello di XGBoost, visti i risultati del training nei casi precedenti.
Questa scelta però può essere liberamente rivalutata, poiché avendo cambiato la strategia di addestramento, anche le prestazioni del modello saranno diverse rispetto a quelle del best model identificato con la grid search, questo è un assunto valido sia per l'albero decisionale che per il XGBoost.
La scelta più adatta sarebbe quella di modificare la funzione per ottenere i pesi dai modelli affinché trovi da sé la miglior combinazione di parametri per il task, cambiando quindi i pesi della media pesata per ottenere gli score finali.

```python
# Metodo che normalizza una matrice per il suo massimo
3 usages
def max_normalization(matrix):
    return matrix / np.max(np.abs(matrix), axis=1, keepdims=True)


# Metodo che calcola e restituisce la somma pesata dei pesi di tre modelli, regression, tree, xgboost,
# per ogni feature della KB
1 usage
def compute_final_weights_from_models(feature_names, matrix, dataframe, classes):
    tree_weights_per_class = get_weights_tree(matrix, dataframe, classes)
    tree_weights = np.array([
        tree_weights_per_class['center'],
        tree_weights_per_class['left'],
        tree_weights_per_class['right']
    ])
    xgb_weights_per_class = get_weights_xgboost(matrix, dataframe, classes)
    xgb_weights = np.array([
        xgb_weights_per_class['center'],
        xgb_weights_per_class['left'],
        xgb_weights_per_class['right']
    ])

    best_logReg_model = train_best_logistic_regression(matrix, dataframe)

    # normalizziamo i pesi su scala [0,1]
    regression_weights = max_normalization(best_logReg_model.coef_)
    tree_weights = max_normalization(tree_weights)
    xgb_weights = max_normalization(xgb_weights)

    final_weights = (0.2 * regression_weights + 0.3 * tree_weights + 0.5 * xgb_weights)
    return final_weights
```

METODO CHE PERMETTE DI OTTENERE I PESI FINALI DI OGNI FEATURE PER OGNI CLASSE

Ottenuti questi score, è poi possibile costruire i fatti della knowledge base, che saranno semplicemente, per ogni feature, lo score del termine per ogni classe.

```python
def build_knowledge_base(feature_name, matrix, dataframe, classes):
    kb = {}
    final_weights = compute_final_weights_from_models(feature_name, matrix, dataframe, classes)
    n_classes, n_features = final_weights.shape

    for i in range(n_features):
        term = feature_name[i]
        kb[term] = {classes[c]: final_weights[c, i] for c in range(n_classes)}
    return kb
```

METODO PER COSTRUIRE LA KNOWLEDGE BASE

```
term_weight('10','center','0.0066').
```

ESEMPIO DI FATTO DELLA KNOWLEDGE BASE COSTRUITA

Costruita la knowledge base, è possibile interrogarla su testo, che deve essere opportunamente tokenizzato utilizzando il medesimo vectorizer usato per le feature della knowledge base, naturalmente.

```python
# Metodo che restituisce la classe di un testo calcolata dalla kb
2 usages
def classify_text(vectorizer, text, knowledge_base, classes):
    feature_names = vectorizer.get_feature_names_out()
    x = vectorizer.transform([text])
    final_weights = np.array([knowledge_base.get(f, 0) for f in feature_names])
    scores = final_weights @ x.toarray().flatten()
    max_idx = scores.argmax()

    return classes[max_idx]
```

METODO PYTHON PER CLASSIFICARE UN TESTO ATTRAVERSO UNA KNOWLEDGE BASE

Prolog

Attraverso python è poi possibile installare un ambiente prolog e testare la knowledge base.
Per farlo si è reso prima necessario scrivere un file con le regole della base di conoscenza, 'rules.pl', e uno con i fatti, 'facts.pl'.

```
% Somma dei pesi dei termini per una determinata classe.
% Caso base: lista vuota → score = 0
sum_bias([], _, 0).

% Caso 1: il termine ha un peso per la classe → aggiungilo alla somma ricorsiva
sum_bias([Word|Rest], Class, Score) :-
    term_weight(Word, Class, W),
    sum_bias(Rest, Class, R),
    Score is W + R.

% Caso 2: il termine NON ha un peso per la classe → ignoralo e prosegui
sum_bias([Word|Rest], Class, Score) :-
    \+ term_weight(Word, Class, _),
    sum_bias(Rest, Class, Score).

% Classifica un testo trovando la classe con somma di pesi più alta
classify_text(Words, Class) :-
    sum_bias(Words, left, L),
    sum_bias(Words, center, C),
    sum_bias(Words, right, R),
    max_member((_, Class), [(L, left), (C, center), (R, right)]).
```

REGOLE KNOWLEDGE BASE

Ottenuti questi due file, attraverso la libreria ***PySwip***[15]***,*** che permette di interagire su python con un ambiente prolog, e all'ambiente prolog ***SWI-Prolog***[16]***,*** è possibile interrogare la knowledge base con query prolog.

```python
# Metodo che genera una query di classificazione utilizzabile in linguaggio prolo
1 usage
def generate_prolog_query(text, vectorizer, class_predicate='classify_text'):
    analyzer = vectorizer.build_analyzer()

    tokens = analyzer(text)

    valid_tokens = [t for t in tokens if t in vectorizer.vocabulary_]

    quoted_tokens = [f"'{token}'" for token in valid_tokens]
    token_list = "[" + ", ".join(quoted_tokens) + "]"

    query = f"{class_predicate}({token_list}, Class)."

    return query


# Metodo per classificazione di un testo attraverso KB usando prolog
def classify_text_prolog(rules_dir, facts_dir, text, vectorizer):
    prolog = Prolog()

    prolog.consult(rules_dir)
    prolog.consult(facts_dir)

    query = generate_prolog_query(text, vectorizer)
    results = list(prolog.query(query))

    if results:
        return results[0]['Class']
    else:
        return 'Unknown'
```

METODO PER INTERROGARE LA KNOWLEDGE BASE SU UN TESTO.

[15] PySwip, https://pyswip.org/ .
[16] SWI-Prolog, https://www.swi-prolog.org/ .

Esempi e risultati ottenuti

Per valutare la knowledge base sono state usate le medesime metriche del supervised machine learning, ma i risultati sono stati più deludenti rispetto ai modelli precedenti.

Model	Accuracy	F1Macro	Precision	Recall
Knowledge Base	0.40	0.28	0.63	0.37

Gli scarsi risultati ottenuti sono probabilmente dovuti all'incapacità del modello di lavorare con feature mai viste.
Dato che i pesi affidati al modello vengono passati da tre modelli diversi che si allenano su una parte del dataset, ci saranno sicuramente termini omessi durante l'addestramento. In questo caso la knowledge base assume che questi termini non conosciuti abbiano un peso nullo, e non vengono considerati.

A riprova di questa conclusione, qualora allenassimo i modelli su tutto il dataset, ottenendo uno score finale per ogni termine del dataset, le performance della base di conoscenza salgono di molto, raggiungendo un 70% di accuracy.

Reti Bayesiane

Introduzione teorica alle Reti Bayesiane

Una rete Bayesiana, o Belief Network, è un grafico aciclico diretto, dove ogni nodo X_i rappresenta una ***variabile aleatoria***[17]**,** e ogni arco rappresenta la relazione di probabilità condizionata $P(X_i|Pa(x_i))$, dove $Pa(X_i)$ è il nodo genitore di X_i. Ogni variabile aleatoria dipende unicamente dai genitori.

Alla rete Bayesiana è associata una tabella di probabilità condizionata per ciascun nodo, che specifica le probabilità delle sue possibili istanze date le configurazioni dei suoi genitori.

Costruzione della rete e scelte progettuali

Il grafo che è stato costruito ha come nodo centrale bias_class, che è il genitore di tutte le parole presenti nel vocabolario, le features. Di conseguenza, si considera ogni parola condizionatamente dipendente dall'orientamento politico del testo.

Per ogni parola, la rete calcola la probabilità che la parola sia presente, dato che il documento appartiene ad una specifica classe politica.

Quando vogliamo fare inferenza, il modello calcola per ogni documento la probabilità $P(\text{bias\_class} = c|\text{parole presenti})$, cioè la probabilità che il documento appartenga alla classe c, dato l'insieme di parole presenti. In questo caso il vectorizer è binario, considera se la parola c'è o non c'è.

L'algoritmo utilizza il teorema di Bayes per il calcolo della probabilità$P(C|w_1, \dots w_n) \propto P(C)\prod_{i=1}^{n} P(w_i|C)$.
Il classificatore seleziona la classe con ***massima probabilità a posteriori (MAP)***[18]**.**

[17] Variabile aleatoria, funzione che associa a ogni esito di un esperimento casuale un valore numerico.
[18] Massima probabilità a posteriori, Maximum A Posteriori (MAP), Metodo che prevede di scegliere un modello m che massimizzi $P(Es|m) \cdot P(m)$. Questo metodo tiene conto della probabilità a priori del modello, preferendo modelli semplici e prevenendo l'overfitting.

```python
# Metodo che restituisce un modello di gradient boost allenanto
2 usages
def train_xgb_model(matrix, dataframe, n_estimators=100, max_depth=6, learning_rate=0.01):
    x_train, x_test, y_train, y_test = train_test_split( *arrays: matrix, dataframe['true_label_enc'], test_size=TEST_SIZE,
                                                        random_state=SEED)
    xgb_model = XGBClassifier(
        objective='multi:softmax',
        num_class=3,
        eval_metric='logloss',
        n_estimators=n_estimators,
        max_depth=max_depth,
        learning_rate=learning_rate,
        random_state=SEED
    )
    xgb_model.fit(x_train, y_train)

    return xgb_model
```

METODO CHE COSTRUISCE E ADDESTRA UNA BELIEF NETWORK PER IL TASK DEL PROGETTO

Risultati, inferenza e confronti

Lavorando con probabilità è possibile calcolare la log-loss, oltre alle metriche viste in precedenza, che valuta quanto le probabilità stimate dal modello sono corrette rispetto alle etichette vere.
Questa metrica penalizza le previsioni sbagliate fatte con alta confidenza e premia le previsioni corrette.
Alti livelli di log-loss indicano previsioni altamente sbagliate, fatte con eccessiva sicurezza.

```python
# Funzione per valutare una rete bayesiana attraverso classification report e logloss
def evaluate_belief_network(b_net, x_test_df, y_test_df, classes):
    y_pred = b_net.predict(x_test_df)
    pred_classes = y_pred['bias_class'].to_list()
    print(classification_report(y_test_df, pred_classes))

    y_pred_prob = b_net.predict_probability(x_test_df)
    y_pred_prob = y_pred_prob[classes]
    logloss = log_loss(y_test_df, y_pred_prob)

    print(f'Log-loss : {logloss:.4f}')
```

FUNZIONE PER VALUTARE L'EFFICACIA DI UNA BELIEF NETWORK

Il testing ha evidenziato scarsi risultati, con una log-loss oltre il 5 ed una accuracy del 50%.

Model	Accuracy	F1Macro	Precision	Recall	LogLoss
Belief Network	0.59	0.58	0.59	0.59	5.8

Una log-loss così alta può essere dovuta ad un numero eccessivo di features, come anche alla scelta di costruzione della belief network, che assumendo tra loro indipendenti i termini, risulta essere molto debole.

Conclusioni

Sintesi dei risultati

Col testing terminato, possiamo assumere che il modello più indicato per questo task risulta essere il XGBoost, superando di netto tutte le altre soluzioni proposte.

L'albero decisionale singolo si mostra anche come una valida alternativa, che seppur con performance inferiori del XGBoost, riesce a mantenere un livello di accuracy superiore agli altri modelli.

La base di conoscenza è invece problematica, sicché le sue performance dipendono dall'addestramento di diversi modelli e dalla combinazione dei risultati.

La belief network è altrettanto incerta, in quanto pur avendo un accuracy migliore della base di conoscenza, la log-loss evidenzia i seri limiti del suo utilizzo.

Limiti e considerazioni critiche

La grandezza del dataset, di per sé già limitante, è stata un limite importante nell'addestramento dei modelli, poiché il tempo necessario per sistemare il dataset non era indifferente, pur considerando che molti articoli non sono stati considerati visto l'impossibilità di trovarli online.

Allo stesso tempo, il dispendio del Grid Search ha impedito di ottenere una configurazione di parametri ottimale sia per il XGBoost che per i modelli utilizzati nella knowledge base.

L'assenza di altre informazioni a corredo delle classi o dei termini ha reso la costruzione della belief network molto banale, e infatti la sua semplicità ricade pesantemente sulle sue performance.

Possibili sviluppi futuri

Le migliorie sono svariate, e alcune sono state indagate ma non portate a termine, per questo non sono state esplorate in questa documentazione.

Tra le migliorie possibili c'è certamente l'aumentare il range delle matrici di tokenizzazione, utilizzando ontologie o dizionari online per la ricerca di sinonimi e relazioni, utili come informazioni di corredo per migliorare i risultati di predizione.
Alcuni test sono stati fatti con ***wordnet***[19], ma a quel punto la tokenizzazione richiedeva molto più tempo e risorse, impedendo di poter lavorare nei tempi stabiliti.

Si è previsto di utilizzare librerie per il sentiment analysis, così da migliorare la predizione da parte dei modelli e ottenere in generale statistiche migliori.
Alcuni test sono stati fatti, ma i risultati erano poco soddisfacenti. Per migliorarli sarebbe richiesto un'analisi più approfondita del testo, motivo per cui si è preferito sorvolare momentaneamente su questa possibilità.

Fare feature engineering sui token per identificare quali erano i più divisivi nel dataset è un'altra possibilità di sviluppo, adatta per migliorare specialmente la belief network.
Una ricerca superficiale dei token più divisivi nel dataset è stata fatta, ma i risultati non erano promettenti, questo probabilmente per la limitatezza del dataset.

[19] Wordnet, https://wordnet.princeton.edu/ .