

Categorizzazione delle personalità sui 16 fattori di Cattell

Anno Accademico 24/25

Componenti del gruppo:

- Francesco Gerardi, 776824, f.gerardi4@studenti.uniba.it

Link repository del progetto:

- <https://github.com/StevenG16/ICON>

Sezioni del documento:

1. [Introduzione del progetto](#)
2. [Preprocessing dei dati](#)
3. [Ragionamento logico](#)
4. [Apprendimento non supervisionato](#)
5. [Apprendimento supervisionato](#)
6. [Conclusione e Sviluppi futuri](#)

Introduzione del progetto:

Questo progetto ha come obiettivo quello di poter distinguere dei gruppi distinti di persone sulla base della loro personalità. Definiti tali gruppi si cerca poi di comprendere a quale gruppo una persona appartenga, permettendo anche di capire se essa sia simile ad altre persone, qualora vi sia un'appartenenza allo stesso gruppo.

Per fare ciò ho scelto di utilizzare un dataset contenente le risposte al test dei 16 fattori di personalità di Cattell, reperibile insieme ad altri dataset simili su openpsychometrics.org.

il dataset è composto da:

- 163 risposte alle domande del questionario, divise in 16 macrocategorie, una per fattore, denominate con lettere maiuscole dalla A alla P, e rappresentate come un intero da 1 a 5. Le singole domande possono essere consultate nella [descrizione dei campi fornita insieme al dataset](#)
- La nazione da cui l'utente ha risposto al questionario, nella forma dei codici dei paesi dello standard ISO 3166.
- la pagina di origine del partecipante al test, denominata con 1 se egli proveniva dalla pagina principale del sito, 2 da google, 3 da facebook, 4 da un url contenente ".edu", 5 da Wikipedia, 6 da una qualunque altra fonte.
- l'accuratezza con cui l'utente crede di aver risposto alle domande, rappresentata come un intero da 1 a 100 (qualora la risposta fosse 0, i dati non venivano inseriti nel dataset)
- il numero di secondi trascorsi dall'inizio del test all'invio delle risposte, rappresentato come un intero maggiore di 0.
- l'età in anni, rappresentata come un intero maggiore di 0.
- il genere dell'utente, rappresentata con 0 = Nessuna Risposta, 1 = Maschio, 2 = Femmina, 3 = Altro.

Preprocessing dei dati: Pulizia del dataset e selezione delle feature

Prima di utilizzare i dati ho scelto di ripulire il dataset dai campi che ho ritenuto non necessari e di sintetizzare le informazioni contenute in tutte le 163 risposte alle domande.

In particolare, ho scelto di eliminare i seguenti campi del dataset per i seguenti motivi:

- pagina di origine del partecipante al test, in quanto ha correlazione minima o nulla con la personalità dell'utente.
- numero di secondi trascorsi dall'inizio del test all'invio delle risposte, in quanto ha poca correlazione con la personalità dell'utente
- età dell'utente, in quanto era utilizzata da chi ha creato il questionario solo per escludere gli utenti minori di 14 anni dalla raccolta dei dati relativi al test, e non ho ritenuto che fosse di alcuna utilità ai fini del progetto.

Ho inoltre riassunto le informazioni contenute nelle 163 risposte nel dataset, in maniera analoga a come viene indicato nel metodo di scoring di tale questionario.

In particolare, ho sommato gli interi delle risposte per ogni categoria, ottenendo così 16 campi, rappresentanti il punteggio totale rispetto al fattore corrispondente.

Una volta fatto questo, ho normalizzato tali punteggi con una normalizzazione Min-Max, per far rientrare tutti i valori nell'intervallo $[0, 1]$, in modo tale da renderli più comodi da utilizzare ed avere i punteggi tutti nello stesso intervallo.

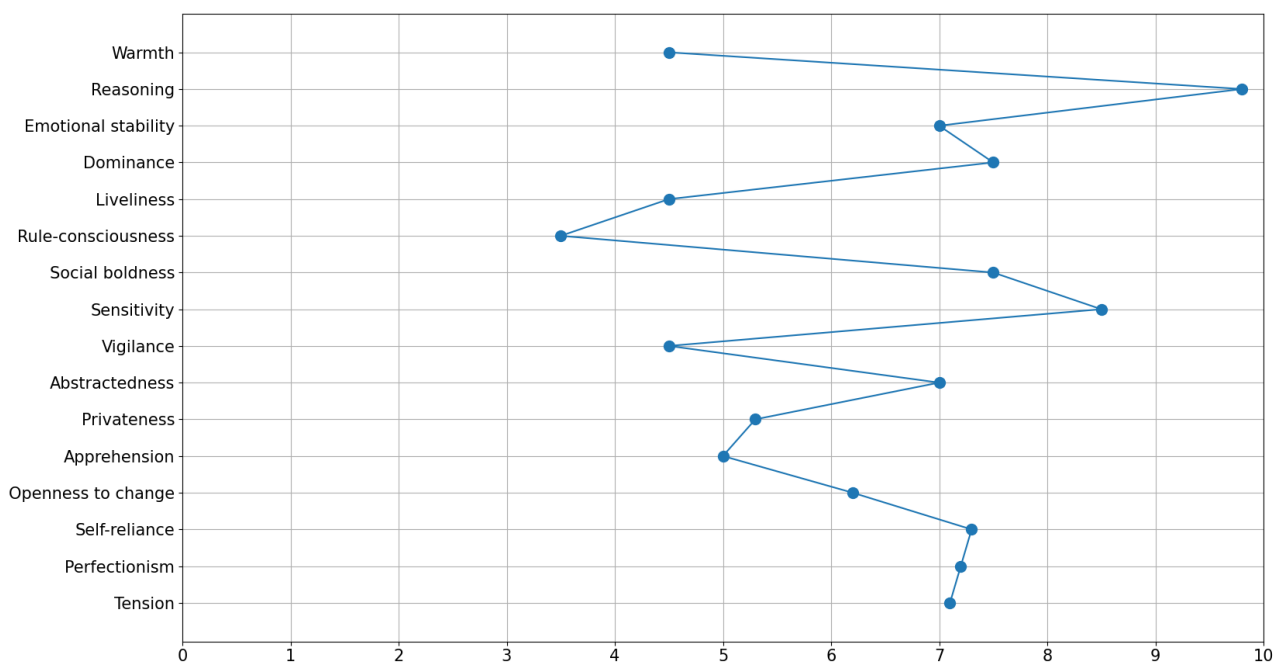
I 16 campi rappresentanti i valori per ogni fattore di personalità, indicati da lettere maiuscole dalla A alla P costituiranno le feature selezionate per il progetto.

I fattori indicati da tali lettere corrispondono alle seguenti caratteristiche:

- A : Warmth
- B : Reasoning
- C : Emotional Stability
- D : Dominance

- E : Liveliness
- F : Rule-Consciousness
- G : Social Boldness
- H : Sensitivity
- I : Vigilance
- J : Abstractedness
- K : Privateness
- L : Apprehension
- M : Openness to change
- N : Self-reliance
- O : Perfectionism
- P : Tension

Ho inoltre scelto di rappresentare graficamente gli esempi, laddove ne avessi bisogno, nella stessa maniera in cui avviene nei risultati di test di questo tipo, ovvero mediante un grafico come quello in questo esempio:



Ogni valore è rappresentato in maniera coerente rispetto alla sua forma numerica all'interno degli esempi nel dataset, con l'unica differenza di essere mostrato su una scala da 0 a 10 anziché in intervallo [0, 1].

Ragionamento logico: Creazione della Knowledge Base

Il ragionamento logico permette di creare una Knowledge Base, utile a specificare una rappresentazione intesa del mondo da rappresentare, formata da assiomi, ovvero proposizioni vere nell'interpretazione intesa, dette regole se hanno una testa e un corpo rappresentante una condizione, oppure dette fatti se formati solo da una testa.

Una volta processati i dati, ho creato un Knowledge Base contenente tutte i dati nel dataset già processati, sottoforma di fatti, ed alcune regole utili ad eseguire query di diverso tipo e con diversi criteri all'interno della KB. Per la creazione della Knowledge Base ho utilizzato il linguaggio prolog, integrandolo con la libreria pyswip, che mi ha permesso di utilizzare la Knowledge Base per fare query direttamente in python.

Le Knowledge base presenti sono:

- **data.pl**: KB contenente i fatti rappresentanti tutti i dati post scrematura iniziale, con ancora le 163 singole risposte alle domande, creato inizialmente, e poi scartato in favore di small_data.pl.

Ho deciso di tenere anche questa versione della KB in caso di eventuali ed ulteriori sviluppi, ma non concretamente utilizzato nel progetto.

- **small_data.pl**: KB effettivamente utilizzata per attingere ai dati utilizzati nelle fasi successive del progetto.

E' composta dai seguenti fatti e regole:

- **person**: fatto contenente tutte le informazioni sul singolo utente, con genere, nazionalità, accuratezza, e punteggio per ognuno dei 16 fattori

- **Regole**

country(COUNTRY, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P) :- person(_, _, COUNTRY1, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P), COUNTRY = COUNTRY1.

country(COUNTRY, Results) :- findall(person(GENDER1, ACCURACY1, COUNTRY, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P), person(GENDER1, ACCURACY1, COUNTRY, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P), Results).

Che permettono di filtrare il dataset per nazione, in due formati diversi, non utilizzate concretamente, ma mantenute per eventuali ulteriori utilizzi.

gender(GENDER, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P) :- person(GENDER1, _, _, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P), GENDER1 = GENDER.

gender(GENDER, Results) :- findall(person(GENDER, ACCURACY1, COUNTRY1, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P), person(GENDER, ACCURACY1, COUNTRY1, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P), Results).

Che permettono di filtrare il dataset per genere, in due formati diversi, non utilizzate concretamente, ma mantenute per eventuali ulteriori utilizzi.

person_factors(A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P) :- person(_, _, _, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P).

Regola utilizzata per filtrare il dataset, ottenendo per ogni utente solamente gli score dei 16 fattori.

person_factors_weighted(WEIGHT, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P) :- person(_, WEIGHT, _, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P).

Regola utilizzata per filtrare il dataset, ottenendo per ogni utente gli score dei 16 fattori e l'accuratezza delle risposte, utilizzata poi come peso nelle fasi successive del progetto.

- **small_data_clustered.pl:** KB contenente fatti analoghi a quelli in small_data.pl, con la rimozione di genere, nazionalità ed accuratezza,

e con l'aggiunta del cluster di appartenenza dell'utente, determinato nella fase di apprendimento non supervisionato.

Tale KB verrà utilizzata per ricavare il dataset necessario alla fase di apprendimento supervisionato.

Apprendimento Non Supervisionato:

Determinare i gruppi di utenti

L'apprendimento non supervisionato è quella parte del machine learning in cui la fase di addestramento è fatta su dei dati non provvisti di feature obiettivo.

L'obiettivo in tale tipo di apprendimento è infatti quello di cercare pattern nei dati, ad esempio cercando di ricostruire una classificazione naturale dei dati mediante la procedura del clustering.

Il clustering partiziona gli esempi di un dataset in classi naturali, in modo che ogni cluster predica i valori delle feature dei suoi esempi.

Esso si divide in hard clustering, in cui ogni esempio è assegnato ad un solo cluster, e soft clustering, in cui un esempio appartiene a più cluster con una certa probabilità.

La fase di apprendimento non supervisionato è stata utilizzata all'interno del progetto per poter dividere gli utenti in gruppi in base a quanto la loro personalità sia simile.

Per fare ciò ho utilizzato l'algoritmo di hard clustering KMeans, la cui implementazione utilizzata è quella fornita dalla libreria scikit-learn.

Ottimizzazione del numero dei cluster

Per eseguire il clustering dei dati in maniera ottimale è necessario determinare innanzitutto quale sia il numero ideale di cluster su cui eseguire l'algoritmo KMeans.

E' possibile determinare tale numero attraverso il cosiddetto metodo del gomito, che consiste nell'individuare il valore di k (numero di cluster) che fornisce la massima riduzione dell'errore rispetto al precedente valore di k .

Ho così utilizzato la libreria yellowbrick per poter determinare il valore ottimale di k nelle seguenti funzioni:


```
def elbow_test(dataset, max_clusters, weights = None):
    kmeans = KMeans(n_init=10, init="random")
    visualizer = KElbowVisualizer(kmeans, k = (1, max_clusters))
    dataset = np.array(dataset)
    visualizer.fit(dataset, sample_weight = weights)
    return visualizer
```

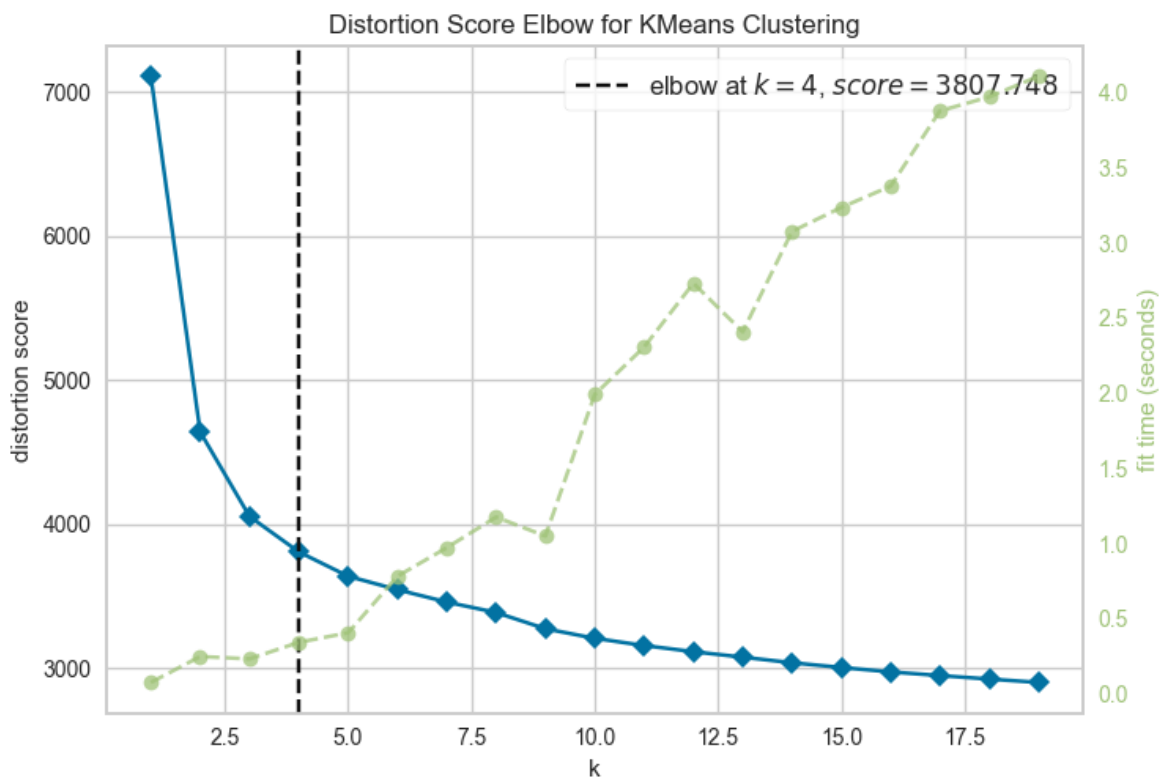
Questa funzione permette di eseguire il metodo del gomito con dataset, numero di cluster ed eventuali pesi, fornendo tali dati come parametri, e restituendo un oggetto a partire dal quale sarà possibile visualizzare il grafico con i risultati.

```
# Main for elbow method
def visualize_elbow():
    max_clusters = 20
    dataset = np.array(query_to_dataset(data_files[data_file], queries[query_name]))
    weight_index = 0
    weights = [np.log(data.pop(weight_index))/np.log(100) for data in dataset]
    visualizer = elbow_test(dataset, max_clusters, weights=weights)
    data_visualization.plot_elbow(visualizer)
```

Essa fa da main per la determinazione del valore ottimale di cluster per l'algoritmo KMeans.

Esegue una query alla KB contenente il dataset, calcola i pesi con cui effettuare il clustering (di cui parlerò in seguito) ed esegue il metodo del gomito fermandosi ad un numero massimo di cluster pari a 20 (da prove precedenti ho notato che valori superiori sarebbero stati innecessari).

Il risultato ottenuto dall'esecuzione di tale funzione è stato il seguente:



Come è possibile notare il valore ottimale di k determinato è stato $k = 4$.

D'ora in poi il clustering KMeans verrà sempre fatto utilizzando tale valore per l'iperparametro `num_clusters`.

Clustering dei dati

Una volta determinato il valore ottimale di cluster da usare, sono passato alla fase di clustering vera e propria, svolta dalle funzioni **`cluster_dataset`** e **`cluster_data`**:

```
def cluster_dataset(dataset, num_clusters, weights = None):
    kmeans = KMeans(n_clusters=num_clusters, n_init=10, init="random", random_state=seed)
    kmeans.fit(dataset, sample_weight=weights)
    return (kmeans)
```

Tale funzione accetta come parametri un dataset, il numero di cluster e una lista di pesi, della stessa lunghezza del dataset. Esso esegue l'algoritmo KMeans con il numero di cluster fornito in input ed assegna ogni esempio del dataset ad un cluster, il cui centroide è calcolato usando una media pesata sui pesi in `weights`. Infine

restituisce un oggetto contenente tutte le attribuzioni degli esempi ai cluster insieme ad altre informazioni.

Le feature scelte per questa fase di apprendimento non supervisionato sono quindi unicamente gli attributi dalla A alla P, rappresentanti il punteggio per ogni fattore della personalità, normalizzato in intervallo [0, 1].

```
# Main for clustering
def cluster_data(visualize = False):
    dataset = query_to_dataset(data_files[data_file], queries[query_name])
    weight_index = 0
    num_clusters = 4
    weights = [np.log(data.pop(weight_index))/np.log(100) for data in dataset]
    dataset = np.array(dataset)

    kmeans = cluster_dataset(dataset, num_clusters, weights=weights)
    centroids = kmeans.cluster_centers_
    labels = kmeans.labels_

    clusters = []
    for i in range(num_clusters):
        indices = np.where(labels == i)[0]
        clusters.append(dataset[indices])

    if visualize:
        visualize_clusters(num_clusters, dataset, weights=weights)
        i = 0
        for centroid in centroids:
            img_name = f"cluster_center_{i}"
            data_visualization.plot_factors(centroid, img_name)
            i += 1

    return clusters
```

Essa costituisce la funzione principale per la fase di apprendimento non supervisionato.

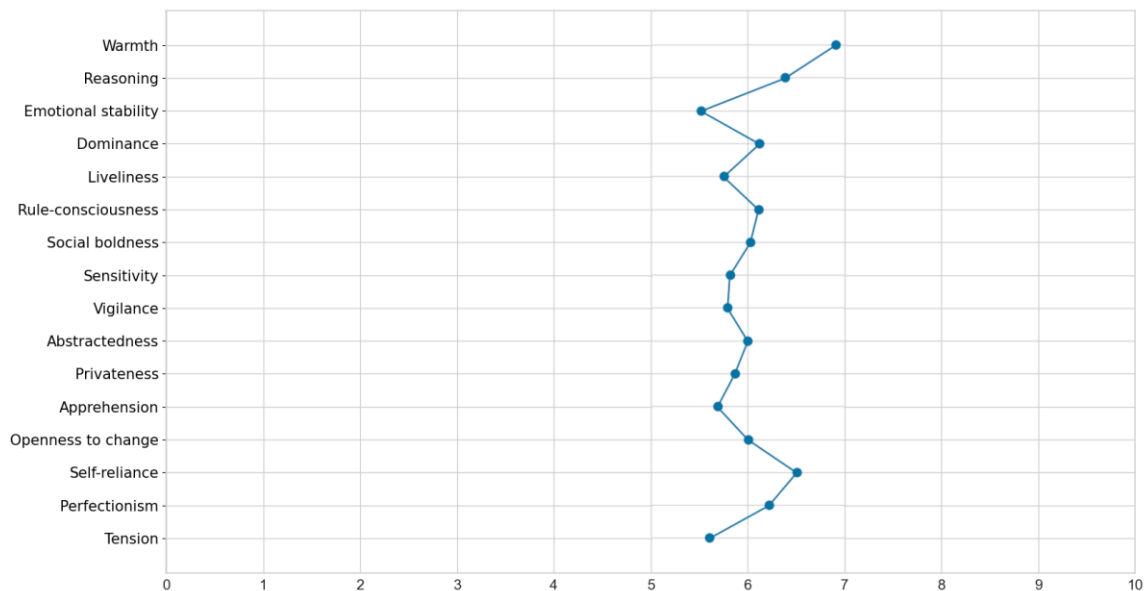
In particolare, ottiene il dataset tramite una query alla KB prolog, per poi formare la lista di pesi da utilizzare nell'algoritmo KMeans. In quanto tra gli attributi presenti nel dataset vi era l'"accuracy", rappresentante su una scala da 1 a 100 quanto l'utente ritenesse che le sue risposte fossero accurate, ho pensato di utilizzare tale valore come peso nel calcolo dei centroidi per ciascun cluster. Inizialmente ho utilizzato i valori "grezzi" dell'attributo "accuracy", ma ho notato che essi avevano un impatto eccessivo in fase di clustering. Per questo motivo ho applicato la funzione logaritmo sull'"accuracy", in modo da ottenere uno scarto non troppo eccessivo tra i valori, per poi normalizzare in intervallo [0, 1].

Dopo aver ottenuto il dataset, la funzione chiama `cluster_dataset` per eseguire il clustering, e ricava il cluster di appartenenza per ogni esempio, che usa per partizionare il dataset originario sulla base del cluster di appartenenza degli esempi.

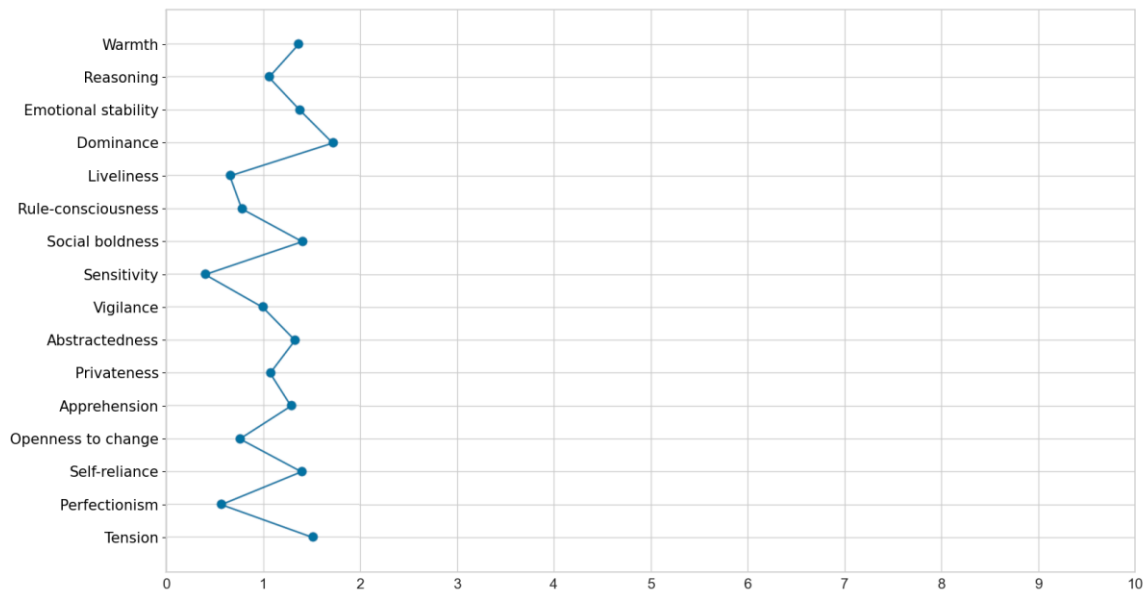
Ho infine deciso di aggiungere la possibilità di poter visualizzare una rappresentazione grafica dei centroidi e della distribuzione degli esempi nei cluster in uno spazio 2D, grazie ad una riduzione della dimensionalità eseguita tramite PCA, fornita dalla medesima libreria `scikit-learn` utilizzata per l'hard clustering con `KMeans`.

I centroidi di ogni cluster così ottenuti sono stati i seguenti:

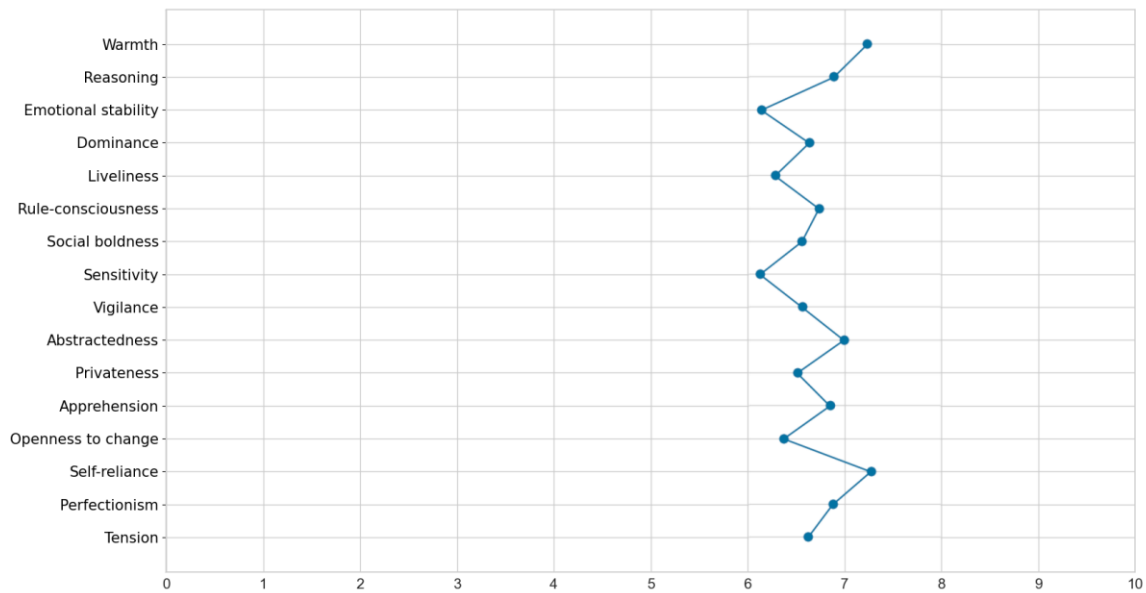
Centroide del cluster 0:



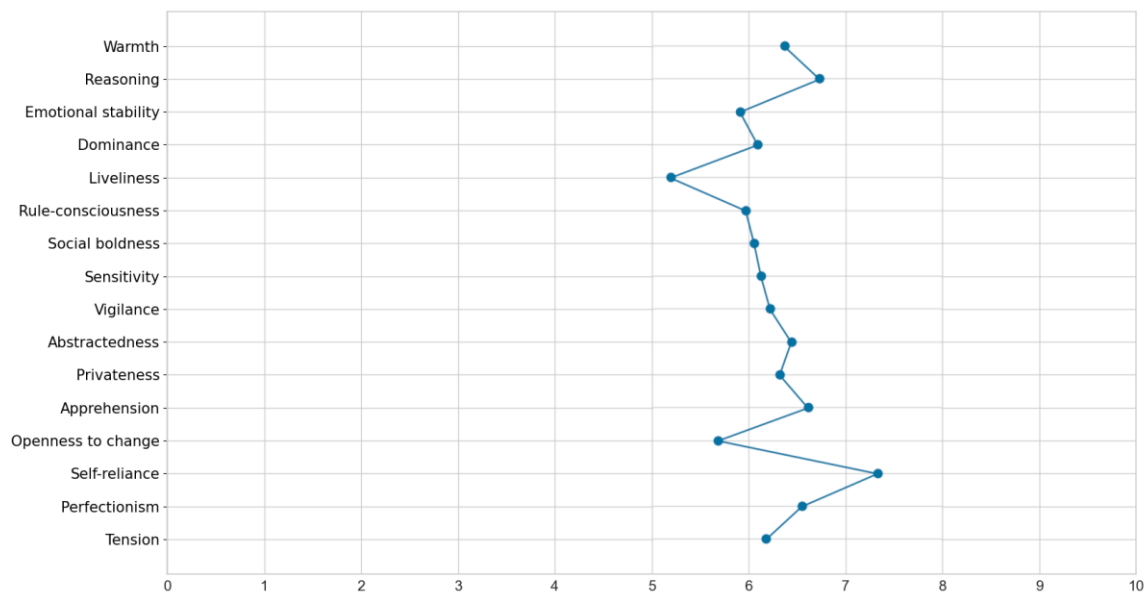
Centroide del cluster 1:



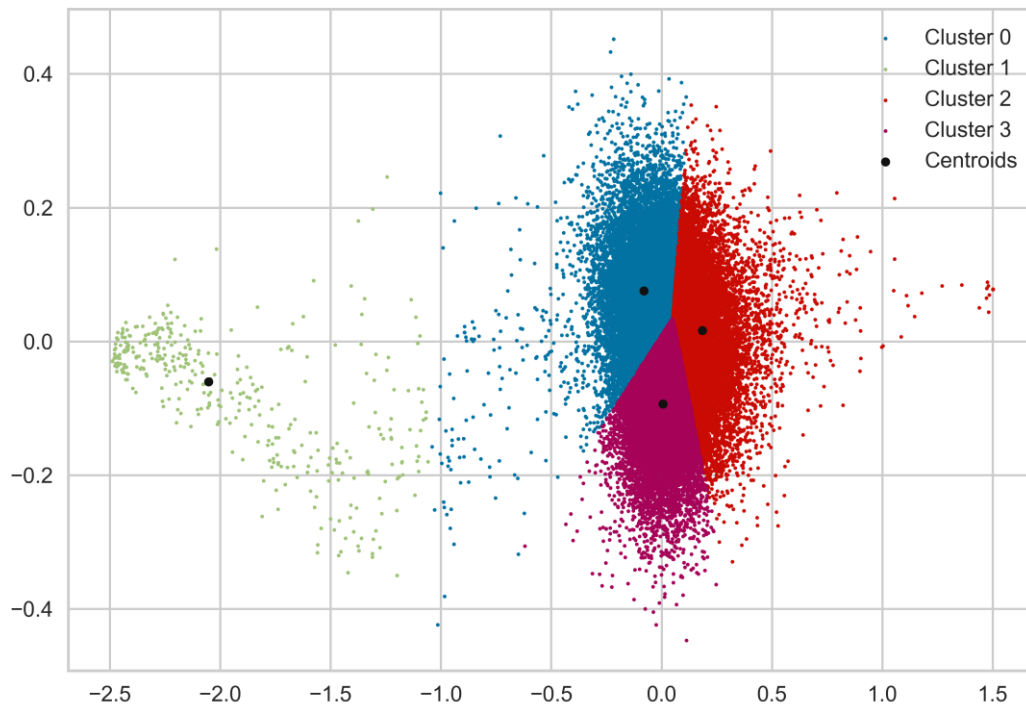
Centroide del cluster 2:



Centroide del cluster 3:



Mentre possiamo osservare come gli esempi siano distribuiti nei cluster nella seguente immagine:



Dopo aver partizionato il dataset in base ai cluster determinati dall'algoritmo KMeans, ho quindi aggiunto ogni esempio, con l'etichetta del corrispettivo cluster, come fatto della Knowledge Base presente nel file `small_prolog_clustered.pl`, in modo da poter poi eseguire efficientemente query per l'ottenimento del dataset per la fase successiva di apprendimento supervisionato.

Apprendimento Supervisionato: Predire il gruppo di appartenenza

L'apprendimento supervisionato è quella parte del machine learning in cui l'addestramento viene fatto su dei dati provvisti di una o più feature obiettivo.

Se tali feature obiettivo da predire hanno un dominio finito, la previsione di tali valori viene detta classificazione, mentre se le feature obiettivo hanno dominio continuo, la previsione di tali valori viene detta regressione.

L'apprendimento supervisionato è stato utilizzato all'interno del progetto per poter classificare gli utenti in un gruppo di appartenenza rispetto alla loro personalità, in

modo da determinare se due o più utenti avessero personalità simili, qualora il gruppo di personalità a cui appartenessero fosse lo stesso.

Per fare ciò ho scelto diversi modelli da valutare e confrontare, per poi scegliere quello con le migliori prestazioni. Le implementazioni utilizzate per i modelli scelti sono quelle facenti parte della libreria scikit-learn.

Come prima cosa ho scelto di creare un oggetto Dataset, che, seppur molto semplice, mi permettesse di organizzare i dati ottenuti dalla query alla knowledge base in modo efficace, dividendo training set da test set, e le feature in input da quella obiettivo, dopo aver ridistribuito casualmente gli esempi all'interno del dataset.

```
class Dataset(object):

    def __init__(self, data = None, test_prob = 0.20, target_index = 0, seed = None):

        if seed:
            random.seed(seed)

        if data is None:
            return

        train_targets = []
        test_targets = []
        train = []
        test = []
        labels = [chr(c) for c in range(ord("A"), ord("P")+1)]
        target_label = "Cluster"
        random.shuffle(data)
        for example in data:
            target = example.pop(target_index)
            is_test = random.random() < test_prob
            if is_test:
                test.append(example)
                test_targets.append(target)
            else:
                train.append(example)
                train_targets.append(target)
        self.train = train
        self.test = test
        self.train_targets = train_targets
        self.test_targets = test_targets
        self.labels = labels
        self.target_label = target_label
```

I modelli che ho selezionato per classificazione nella fase di apprendimento supervisionato sono i seguenti:

- **Albero di Classificazione:** albero binario costituito da nodi interni, etichettati con condizioni sui valori delle feature negli esempi e connessi a due figli attraverso archi etichettati con True e False, e nodi foglia, etichettati con una stima della feature obiettivo (classe dell'esempio da classificare)

- **Regressione logistica:** utilizza una funzione lineare per ogni classe, di cui apprendere i pesi.

I pesi sono presenti in ogni funzione per ogni feature, più un peso aggiuntivo considerato assumendo un' ulteriore feature X_0 dal costante valore 1. Il risultato di ciascuna funzione viene poi compresso utilizzando la funzione softmax, la quale fornisce come output un vettore contenente le predizioni per ciascuna classe del dominio della feature obiettivo, tra le quali viene selezionata la classe con predizione dal valore maggiore.

- **Random forest:** modello di ensemble learning composto da multipli alberi di decisione, addestrati su parti differenti del dataset, in modo da dare diverse predizioni per ogni esempio da classificare, le quali vengono poi combinate tra loro per fornire una predizione finale.

- **Gradient boosted trees:** modello di ensemble learning lineare avente alberi binari come feature, appresi sequenzialmente tramite boosting. Esso predice la classe di appartenenza sommando gli alberi, per poi comprimere il risultato mediante funzione softmax.

Ottimizzare gli iperparametri

Prima di confrontare i modelli tra loro, decidendo quale avesse le migliori prestazioni, ho ottimizzato gli iperparametri per ciascun modello.

Gli iperparametri che ho scelto di ottimizzare per ciascun modello sono stati i seguenti:

```
"DecisionTree":{
  "criterion":["gini", "entropy", "log_loss"],
  "splitter":["best"],
  "max_depth":[None, 1, 2, 5, 10, 25, 50],
  "min_samples_split": [2, 5, 10, 25, 50, 100],
  "min_samples_leaf": [1, 2, 5, 10, 25, 50, 100],
},
```

I parametri da ottimizzare sono:

- **criterion**: funzione che misura la qualità degli split di condizione
- **splitter**: criterio di selezione degli split
- **max_depth**: profondità massima dell'albero ammessa
- **min_samples_split**: numero minimo di esempi per dividere un nodo interno anzichè renderlo foglia
- **min_samples_leaf**: numero minimo di esempi necessari a creare un nodo foglia

```
"LogisticRegression": {
  "C":[0.001, 0.01, 0.1, 1, 10, 100, 1000],
  "solver":["lbfgs", "newton-cg", "sag", "saga"],
  "max_iter": list(range(250,2500,250))
},
```

I parametri da ottimizzare sono:

- **C** : inverso della forza di regolarizzazione (più è basso più è forte la regolarizzazione)
- **solver**: algoritmo utilizzato nell'ottimizzazione dei pesi
- **max_iter**: numero massimo di iterazioni permesse all'algoritmo per convergere

```
"RandomForest": {
  "n_estimators": [5, 10, 25, 50, 100],
  "criterion":["gini", "entropy", "log_loss"],
  "max_depth":[None, 1, 2, 5, 10, 25, 50],
  "min_samples_split": [2, 5, 10, 25, 50, 100],
  "min_samples_leaf": [1, 2, 5, 10, 25, 50, 100],
},
```

I parametri da ottimizzare sono:

- **n_estimators**: numero di alberi nella foresta
- **criterion**: funzione che misura la qualità degli split di condizione.
- **splitter**: criterio di selezione degli split
- **max_depth**: profondità massima degli alberi ammessi
- **min_samples_split**: numero minimo di esempi per dividere un nodo interno anziché renderlo foglia
- **min_samples_leaf**: numero minimo di esempi necessari a creare un nodo foglia

```
"GradientBoosting": {
  "loss": ["log_loss", "exponential"],
  "learning_rate": [0.0, 0.1, 1, 10],
  "n_estimators": [10, 50, 100, 250],
  "max_depth": [None, 1, 2, 5, 10, 25],
  "criterion": ["friedman_mse", "squared_error"],
  "min_samples_split": [2, 5, 10],
  "min_samples_leaf": [1, 2, 5, 10],
}
```

I parametri da ottimizzare sono:

- **loss**: funzione di loss da ottimizzare
- **learning_rate**: fattore che diminuisce il contributo di ogni albero
- **n_estimators**: numero di fasi di boosting da eseguire
- **max_depth**: profondità massima degli alberi ammessi
- **criterion**: funzione che misura la qualità degli split di condizione
- **min_samples_split**: numero minimo di esempi per dividere un nodo interno anziché renderlo foglia
- **min_samples_leaf**: numero minimo di esempi necessari a creare un nodo foglia

L'operazione di ottimizzazione degli iperparametri è stata eseguita dalle funzioni **hyperparameter_tuning** e **tune_models**:

```
def hyperparameter_tuning(dataset, model, model_name):
    search = GridSearchCV(model, hyperparams[model_name], cv=5, n_jobs=-1, verbose=3)
    search.fit(dataset.train, dataset.train_targets)
    return search.best_params_
```

Esegue una ricerca sistematica della migliore combinazione dei parametri forniti sul modello in input, valutando le prestazioni con una k-fold cross-validation, con $k = 5$.

La **k-fold** cross validation è una tecnica di validazione che divide il dataset in k partizioni, dette "fold", delle quali $k-1$ compongono il training set, mentre la rimanente verrà usata per validare i risultati del training. Tale processo viene ripetuto tante volte quanto è grande k , usando ogni volta un fold di valutazione diverso ed i restanti come training set.

Alla fine del processo la funzione restituisce la miglior combinazione di parametri trovata.

```
def tune_models():
    print("[*] Querying the dataset...")
    data = data_preprocessing.query_to_dataset(data_preprocessing.data_files["small_prolog_clustered"], data_preprocessing.queries["factors_all_clustered"])
    print("[+] Dataset successfully queried")
    dataset = Dataset(data, target_index=-1)
    print("Targets: ", [dataset.train_targets[i] for i in range(50) ], " ...")

    optimal_parameters = {}
    for model, model_name in models:
        print(f"[*] Searching the best hyperparameters configuration for {model_name}...")
        best_results = hyperparameter_tuning(dataset, model, model_name)
        print(f"[+] Optimal hyperparameters configuration found for {model_name}:\n{best_results}")
        optimal_parameters[model_name] = best_results
    save_item("params_config.pkl", optimal_parameters)
```

Funzione principale per l'ottimizzazione degli iperparametri. Ottiene il dataset mediante query alla KB, crea un oggetto Dataset a partire da esso, e chiama hyperparameter_tuning per ogni modello, salvando infine le configurazioni migliori per tutti i modelli in un file binario.

Dopo il processo di ottimizzazione degli iperparametri, le configurazioni ottimali trovate sono le seguenti:

```
best_hyperparameters = {
    "DecisionTree": {'criterion': 'log_loss', 'max_depth': None, 'min_samples_leaf': 5, 'min_samples_split': 2, 'splitter': 'best'},
    "LogisticRegression": {'C': 1000, 'max_iter': 250, 'solver': 'newton-cg'},
    "RandomForest": {'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100},
    "GradientBoosting": {'criterion': 'squared_error', 'learning_rate': 1, 'loss': 'log_loss', 'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 250}
}
```

Confronto dei modelli

Una volta ottenute le configurazioni ottimali di iperparametri ho valutato i diversi modelli sulle seguenti metriche, in modo da avere dei criteri su cui confrontarli tra loro:

- **accuracy**: numero di predizioni corrette su quelle totali
- **f1**: metrica che combina precision e recall in un solo valore
- **precision**: numero di true positive su true positive + true negative
- **recall**: numero di true positive su true positive + false negative

il calcolo di tali metriche per ogni modello è stato fatto dalle funzioni

model_validation, che calcola la media delle metriche per un modello, su una k-fold con $k = 10$, e **validate_models**, funzione main per il calcolo delle metriche, la quale chiama model_validation per ogni modello (inizializzato con i parametri ottimali) e salva tutti i risultati in un file binario:

```
def model_validation(dataset, model):
    results = {}
    metrics = ["accuracy", "f1_macro", "precision_macro", "recall_macro"]

    kfold = KFold(n_splits = 10)
    for metric in metrics:
        results[metric] = np.mean(cross_val_score(model, dataset.train, dataset.train_targets, scoring=metric, cv = kfold, n_jobs=-1))

    return results

def validate_models():
    print("[*] Querying the dataset...")
    data = data_preprocessing.query_to_dataset(data_preprocessing.data_files["small_prolog_clustered"], data_preprocessing.queries["factors_all_clustered"])
    print("[+] Dataset successfully queried")
    dataset = Dataset(data, target_index=-1)
    print("Targets: ", [dataset.train_targets[i] for i in range(50)], " ...")

    filename = "validation_results.pkl"
    results = {}

    for model, model_name in models:
        print(f"[*] Evaluating metrics for {model_name}...")
        model = model(**best_hyperparameters[model_name])
        result = model_validation(dataset, model)
        print(f"[+] Metrics for {model_name}: \n {result}")
        results[model_name] = result

    print(f"[*] Saving results into {filename}...")
    save_item(filename, results)
    print(f"[+] Results successfully saved into {filename}")
```

Le metriche ottenute per ogni modello dopo tale processo sono le seguenti:

DECISION TREE	
accuracy	0.7985534604966402
f1	0.8424373966127927
precision	0.840855016045537
recall	0.8428123404089117

LOGISTIC REGRESSION	
accuracy	0.9976145094457977
f1	0.9953641002021831
precision	0.9976585698368426
recall	0.993257587541151

RANDOM FOREST	
accuracy	0.9086919305955741
f1	0.9305132763751673
precision	0.9320998680638188
recall	0.9291276088587747

GRADIENT BOOSTING TREES	
accuracy	0.9589391880491049
f1	0.9627680305115268
precision	0.9637352978183047
recall	0.9357408999888845

Come è possibile notare il modello con le prestazioni migliori in assoluto è la **Regressione Logistica**, seguito da **Gradient Boosting Trees**, **Random Forest**, ed infine l'**Albero di Classificazione**.

Prima di utilizzare il modello vero e proprio allenato sull'intero training set, visti i valori molto alti delle metriche, andiamo a verificare l'eventuale presenza di overfitting con un ultimo test.

Test per l'overfitting

Il fenomeno dell'**overfitting** si manifesta nel momento dell'apprendimento quando un modello si specializza troppo sui dati di training, basandosi su regolarità apparenti in essi, ma non presenti nei dati di test o nei dati del mondo reale, perdendo così parte della sua capacità di generalizzazione sui dati, risultando quindi in un calo delle performance del modello sui dati non di training.

Uno dei modi per verificare la presenza di overfitting nei modelli è quella di addestrare i modelli su diverse proporzioni del training set, per poi, sulla base di una qualche metrica, osservarne il comportamento al crescere delle dimensioni del training set, valutando in particolare la qualità delle predizioni fatte sul test set, rispetto a quelle fatte sul training set.

Nel mio caso ho valutato la qualità delle predizioni usando l'accuracy, ed ho scelto come proporzioni del training set 1/10, 1/3, 1/2, 3/4, 1.

Il calcolo delle train e test accuracy è stato fatto dalle funzioni **test_model** e **test_models**:

```

def test_model(dataset, model, num_tests = 5):
    train_scores = []
    test_scores = []
    train_sizes = [0.1, 0.33, 0.5, 0.75, 1]

    for size in train_sizes:
        print(f"\tTest size: {int(size*len(dataset.train))}")
        train_accuracies = []
        test_accuracies = []
        for i in range(num_tests):
            if size != 1:
                train_set, _, train_targets, _ = train_test_split(dataset.train, dataset.train_targets, train_size=size)
                model.fit(train_set, train_targets)
                train_predictions = model.predict(train_set)
                train_accuracies.append(accuracy_score(train_targets, train_predictions))
                test_predictions = model.predict(dataset.test)
                test_accuracies.append(accuracy_score(dataset.test_targets, test_predictions))
            else:
                data = dataset.train + dataset.test
                targets = dataset.train_targets + dataset.test_targets
                train_set, test_set, train_targets, test_targets = train_test_split(data, targets, train_size=0.8)
                model.fit(train_set, train_targets)
                train_predictions = model.predict(train_set)
                train_accuracies.append(accuracy_score(train_targets, train_predictions))
                test_predictions = model.predict(test_set)
                test_accuracies.append(accuracy_score(test_targets, test_predictions))

        train_scores.append(train_accuracies)
        test_scores.append(test_accuracies)
    return train_scores, test_scores

```

Nella funzione test model vengono calcolate train e test accuracy per il modello in input, per ogni proporzione del dataset.

Piuttosto che usare nuovamente una k-fold, vista la quantità di dati disponibili, ho scelto di effettuare **holdout** sul dataset, usando come test set il 20% dei dati totali, lasciati inutilizzati finora appositamente per questa fase.

In particolare, per ogni dimensione del training set eccetto la più grande (totalità del training set), per 5 volte ho preso un sottoinsieme casuale del training set di tale dimensione, su cui allenare il modello, per poi testare sui 10k esempi circa facenti parte del test set.

Per la dimensione del training set pari alla totalità del training set originale, ho invece unito training e test set, per poi selezionare casualmente per 5 volte un training set della dimensione originale, utilizzando il resto degli esempi come test set.


```
def test_models():
    print("[*] Querying the dataset...")
    data = data_preprocessing.query_to_dataset(data_preprocessing.data_files["small_prolog_clustered"], data_preprocessing.queries["factors_all_clustered"])
    print("[+] Dataset successfully queried")
    dataset = Dataset(data, target_index=-1)
    print("Targets: ", [dataset.train_targets[i] for i in range(50) ], " ...")

    results = {mn:{ } for m, mn in models}

    for model, model_name in models:
        print(f"[*] Testing {model_name}")
        model = model(**best_hyperparameters[model_name])
        train_scores, test_scores = test_model(dataset, model)
        results[model_name]["train_accuracy"] = train_scores
        results[model_name]["test_accuracy"] = test_scores

    print(f"[+] Results for all models:\n{results}")

    filename = "test_results.pkl"
    print(f"[*] Saving results in {filename}...")
    save_item(filename, results)
    print(f"[+] Results successfully saved into {filename}")
```

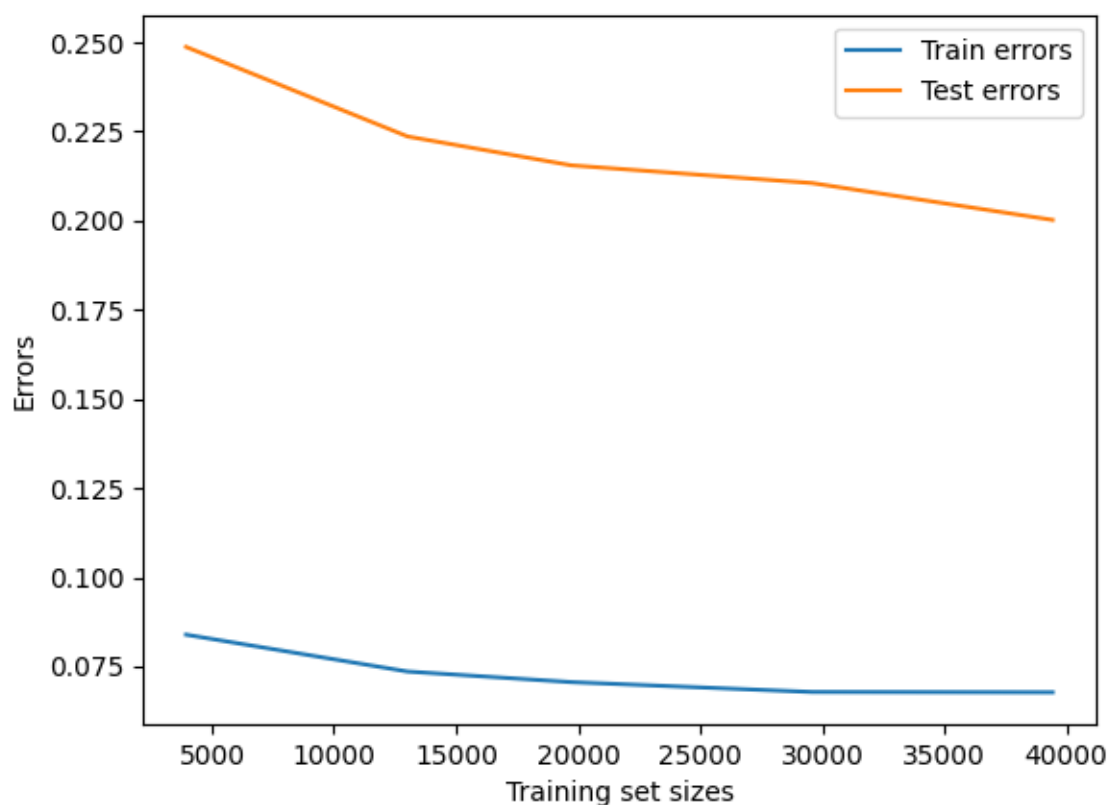
Funzione principale per il test di ogni modello con gli iperparametri ottimizzati.

Chiama test_model per ogni modello e salva i risultati per tutti i modelli in un file binario.

Andiamo ora ad analizzare i risultati ottenuti dall'esecuzione della funzione test_models:

Decision Tree

La curva di apprendimento è la seguente:

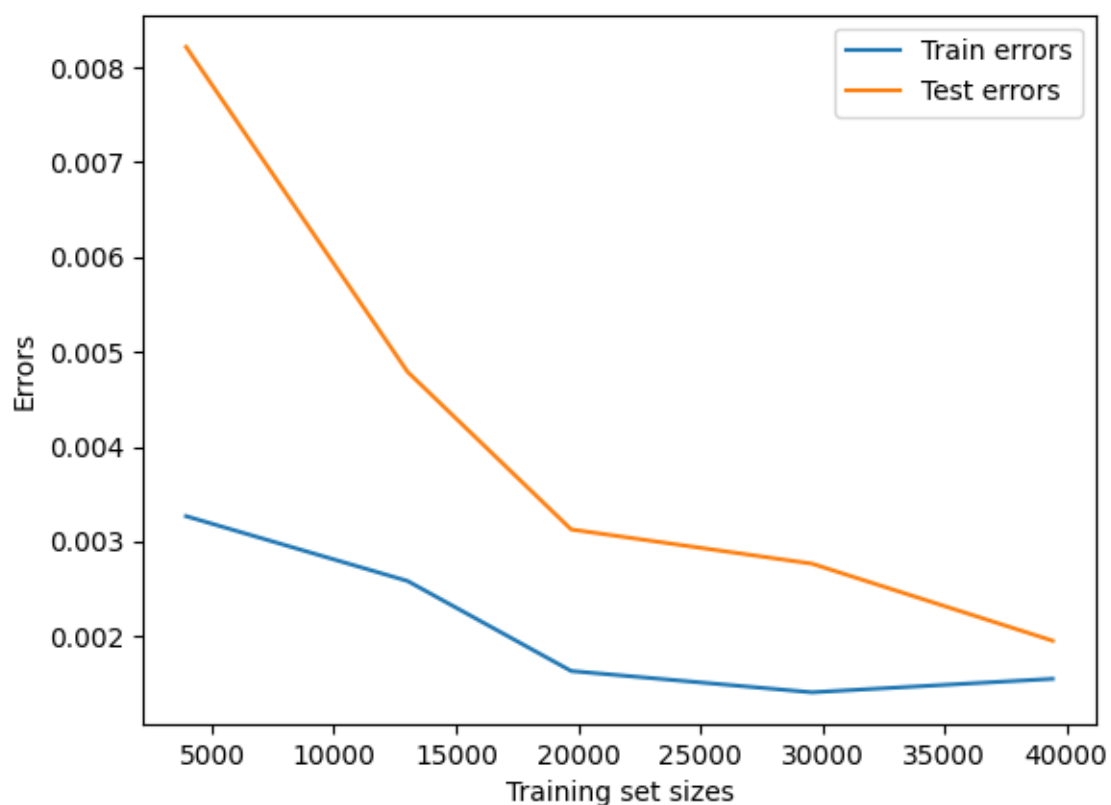


Mentre qui possiamo osservare varianza e deviazione standard degli errori al crescere del training set:

```
----- DecisionTree -----
Train:
  Error Variance: [0.0000037106 0.0000010256 0.0000005239 0.0000004231 0.0000006904]
  Error Std Deviation: [0.0019262851 0.0010127241 0.0007237804 0.0006504813 0.0008308947]
Test:
  Error Variance: [0.0000017650 0.0000074818 0.0000079751 0.0000056252 0.0000145305]
  Error Std Deviation: [0.0013285350 0.0027352862 0.0028240149 0.0023717506 0.0038118925]
```

Logistic Regression

La curva di apprendimento è la seguente:

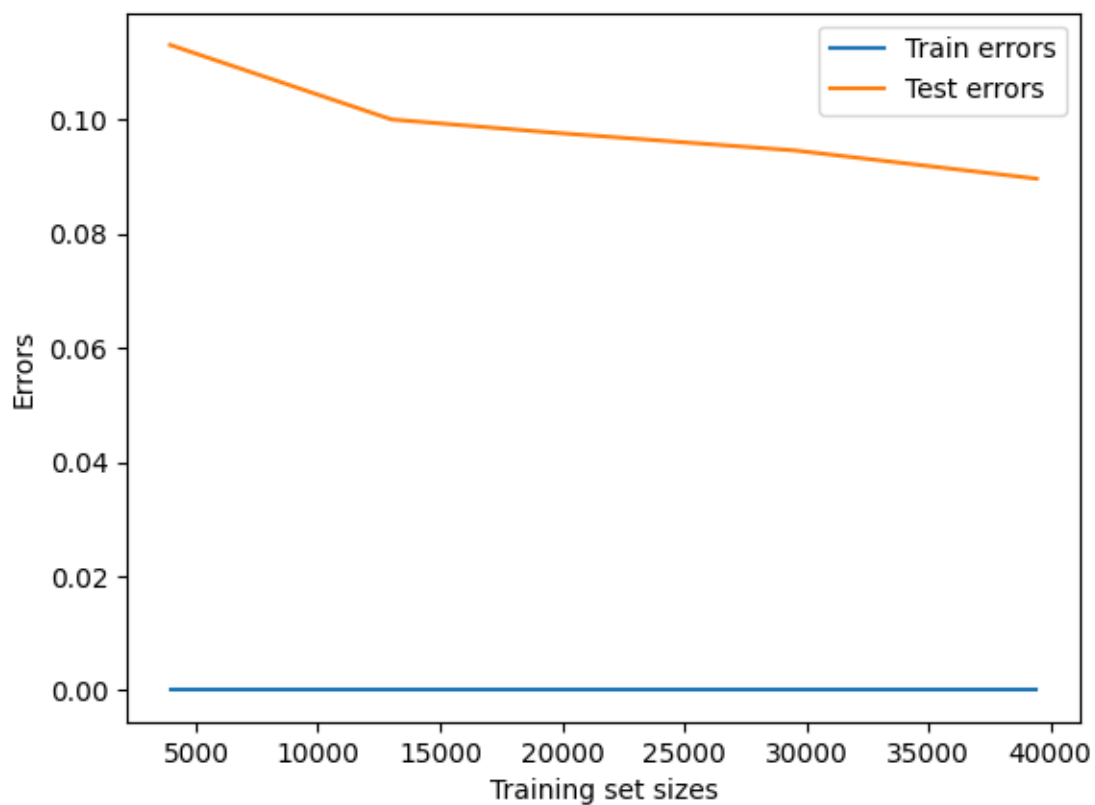


Mentre qui possiamo osservare varianza e deviazione standard degli errori al crescere del training set:

```
----- LogisticRegression -----
Train:
  Error Variance: [0.0000007140 0.0000006427 0.0000000365 0.0000000234 0.0000000486]
  Error Std Deviation: [0.0008449691 0.0008017114 0.0001909983 0.0001531015 0.0002205048]
Test:
  Error Variance: [0.0000005142 0.0000005479 0.0000002426 0.0000000426 0.0000003285]
  Error Std Deviation: [0.0007170476 0.0007402018 0.0004925626 0.0002063459 0.0005731897]
```

Random Forest

La curva di apprendimento è la seguente:

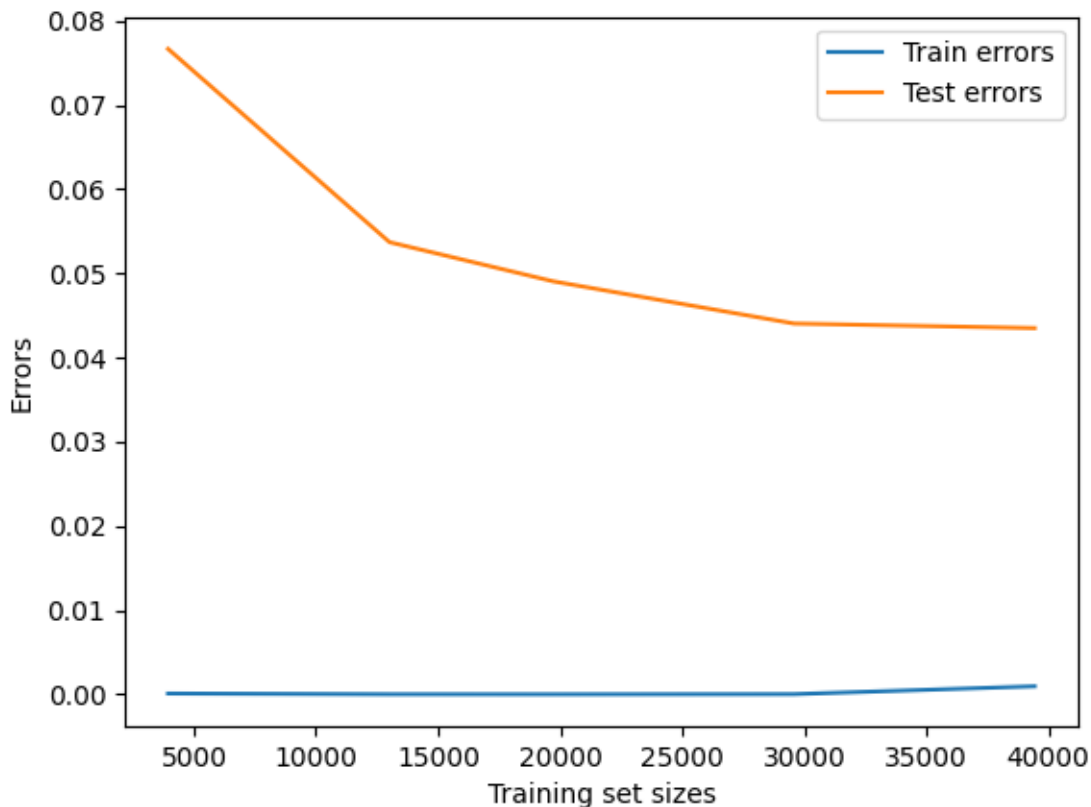


Mentre qui possiamo osservare varianza e deviazione standard degli errori al crescere del training set:

```
----- RandomForest -----
Train:
  Error Variance: [0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000]
  Error Std Deviation: [0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000]
Test:
  Error Variance: [0.0000034625 0.0000038747 0.0000053641 0.0000030504 0.0000066347]
  Error Std Deviation: [0.0018607868 0.0019684147 0.0023160539 0.0017465389 0.0025757818]
```

Gradient Boosting

La curva di apprendimento è la seguente:



Mentre qui possiamo osservare varianza e deviazione standard degli errori al crescere del training set:

```
----- GradientBoosting -----  
Train:  
  Error Variance: [0.0000000156 0.0000000010 0.0000000004 0.0000000016 0.0000033209]  
  Error Std Deviation: [0.0001250378 0.0000309382 0.0000204186 0.0000396866 0.0018223360]  
Test:  
  Error Variance: [0.0000327992 0.0000032352 0.0000023442 0.0000024005 0.0000465783]  
  Error Std Deviation: [0.0057270596 0.0017986588 0.0015310896 0.0015493455 0.0068248285]
```

Osservando le curve di errore decrescenti in tutti i grafici, e visti i valori molto bassi di varianze e deviazioni standard, possiamo concludere che in nessuno dei modelli sembra essere presente il fenomeno di overfitting.

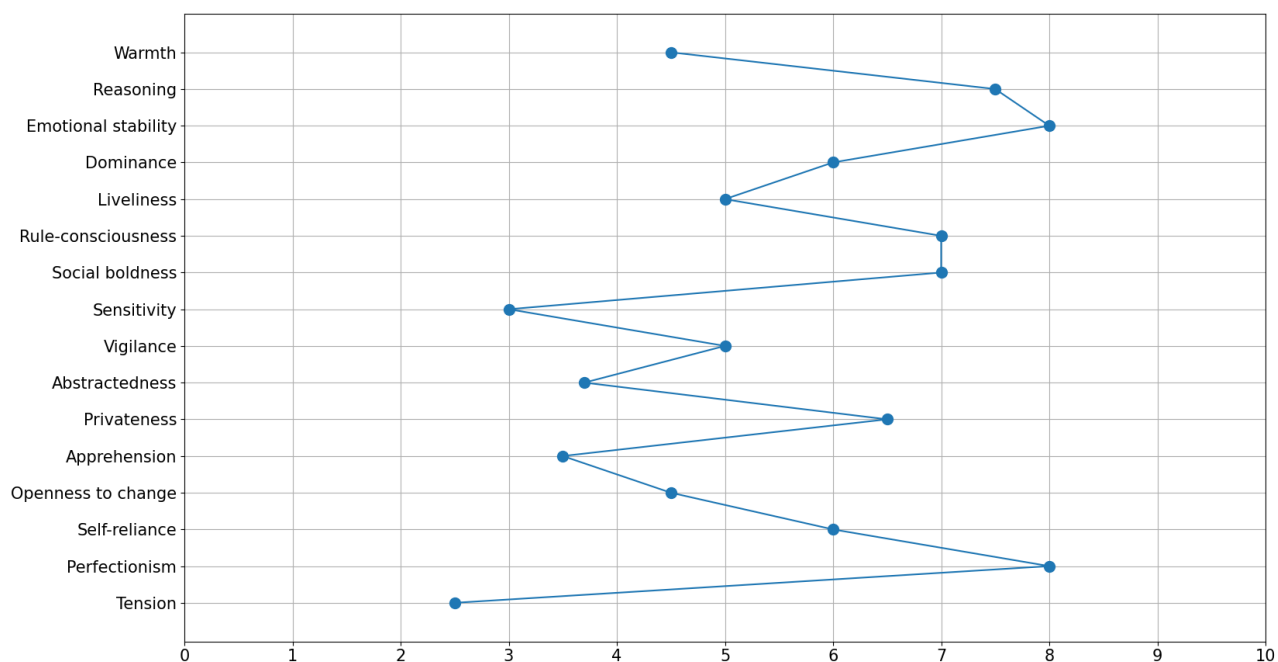
Conclusione e sviluppi futuri:

Alla fine della fase di test dei modelli e di verifica della presenza di overfitting ho quindi adottato come modello da usare per le predizioni in ambito di apprendimento supervisionato il modello di Regressione Logistica.

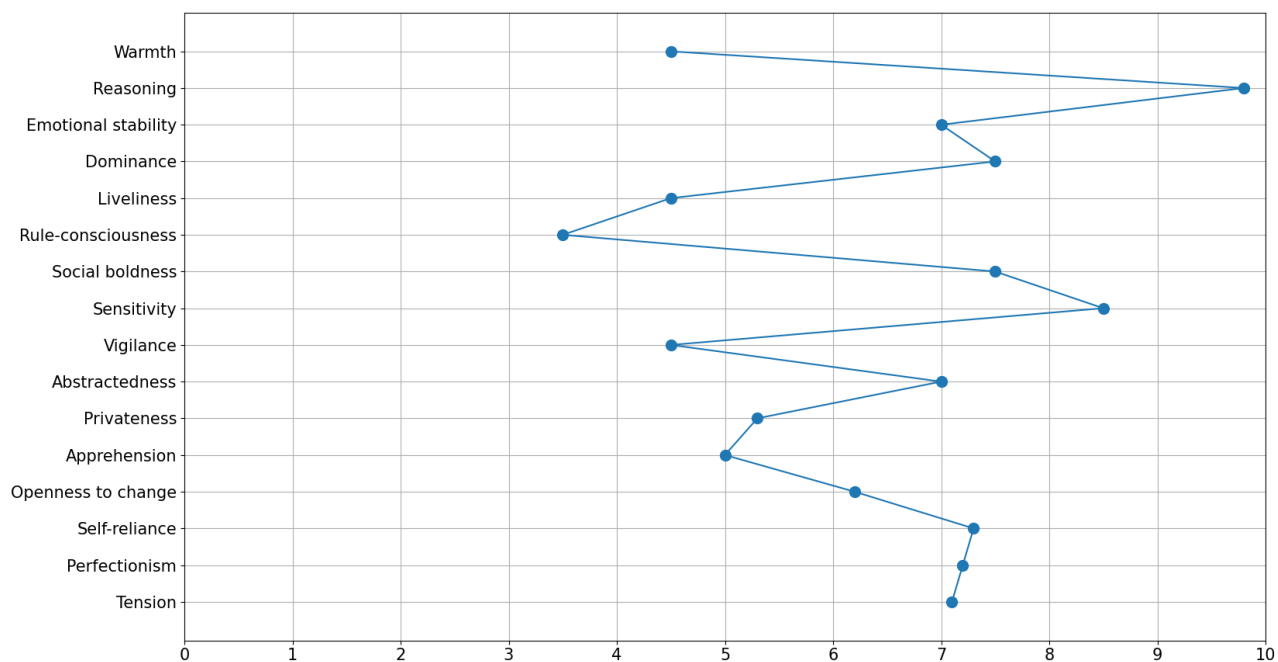
Per provare il modello ottenuto ho quindi preso tre esempi di profili di personalità, documentate usando il test dei 16 fattori di Cattell, e le ho usate per verificare se la classificazione avvenisse in maniera corretta.

I profili selezionati sono quelli di un pilota di aerei di linea, un artista ed uno scrittore. Tracciandole su un grafico i valori dei loro 16 fattori di personalità appaiono come segue:

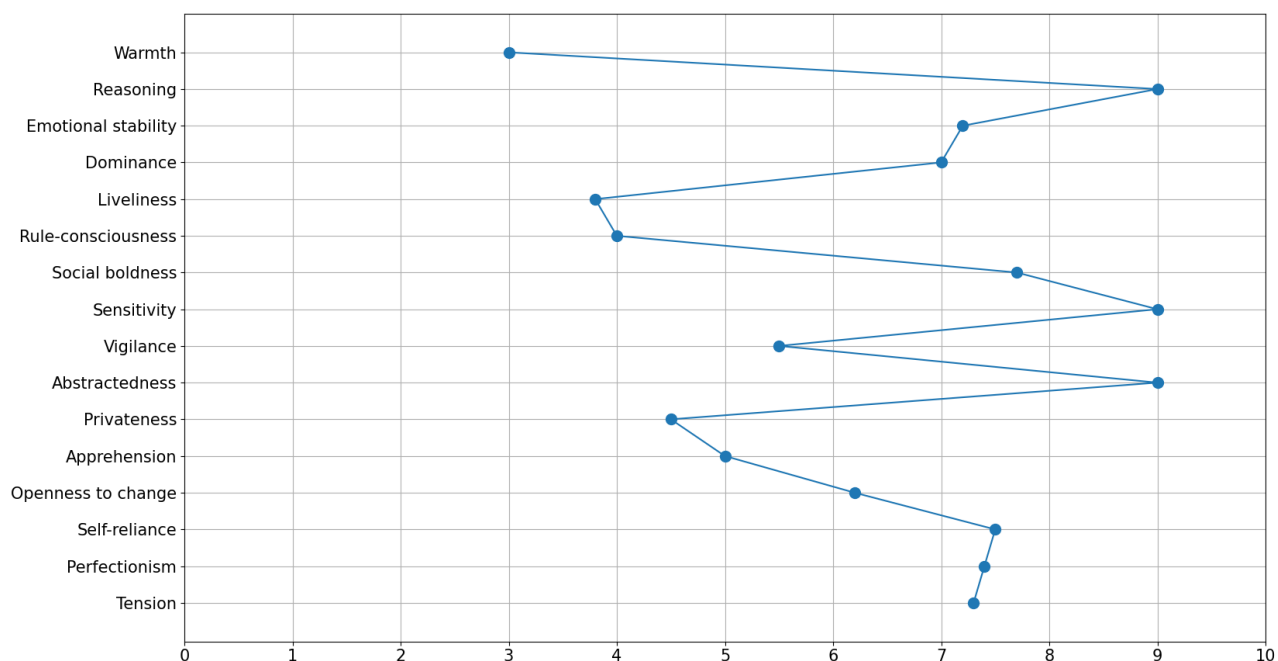
Pilota di aerei di linea



Artista



Scrittore



Osservando i grafici possiamo notare come artista e scrittore abbiano personalità molto simili, mentre il pilota di aerei di linea appaia molto diverso su molti fattori.

Ciò avviene nella funzione **test_on_new**, che ottiene il dataset, crea il modello di Regressione Logistica con gli iperparametri ottimizzati, lo allena sul dataset ed infine esegue la predizione sui nuovi dati forniti.

Verifichiamo ora come viene eseguita la classificazione dei vari profili di personalità:

```
Cluster predictions:  
Pilot: 3  
Artist: 2  
Writer: 2
```

Possiamo osservare quindi che il pilota di linea è stato classificato come appartenente al cluster di personalità con etichetta 3, mentre sia l'artista che lo scrittore siano stati classificati come appartenenti al cluster di personalità con etichetta 2, coerentemente con quanto ci si aspettava vista la somiglianza tra i due profili.

Sviluppi futuri

Eventuali idee per sviluppi futuri potrebbero essere quelle di ampliare con la consulenza di esperti del campo psicologico il progetto, magari riuscendo a dare un significato più concreto ai cluster, anziché limitarsi alle semplici etichette numeriche ora presenti.

Fatto ciò, si potrebbe anche sviluppare un'applicazione che includa anche il questionario da cui raccogliere i dati sull'utente che la utilizza, per poi classificare l'utente in uno dei cluster, fornendogli informazioni aggiuntive sul suo macrogruppo di personalità e presentandogli esempi di personalità note simili alla sua o dargli consigli di vario tipo.

Si potrebbe infine valutare l'idea di considerare come feature tutte le 163 domande presenti inizialmente, anziché utilizzare le 16 feature che le riassumono nei 16 fattori di personalità come fatto da me, per poi confrontare i due modelli e vedere se siano presenti differenze sostanziali tra i due approcci.