Università degli Studi di Messina

Dipartimento MIFT

PROGETTO LABORATORIO DI RETI E SISTEMI DISTRIBUITI

Autore: Davide Mento MAT.527917

MESSINA, 21/06/2024

Indice

Capitolo 1: Stato dell'arte

1.1 L'algoritmo KNN (K-Nearest Neighbors)

L'algoritmo KNN è un metodo di classificazione/regressione appartenente alla famiglia degli algoritmi di apprendimento supervisionato. La sua peculiarità risiede nell'approccio lazy-learning, dove memorizza semplicemente l'intero set di dati di addestramento e calcola le predizioni solo al momento della classificazione di un'istanza di test. Questo lo distingue da metodi più complessi come le reti neurali profonde, che richiedono una fase di addestramento intensiva. KNN si basa sull'assunzione che istanze simili siano assegnate a classi simili. In pratica, la classe di un'istanza di test viene predetta basandosi sulle etichette delle istanze più vicine ad essa nello "spazio delle caratteristiche". [Raj+20]

L'algoritmo KNN trova ampio utilizzo in vari settori, tra cui:

- Medicina: per la classificazione e la previsione di malattie e casi clinici. [XB20]
- Marketing: per la raccomandazione personalizzata di prodotti e contenuti. [ZL22]
- Economia: per l'analisi dei mercati e la previsione dei trend. [Alk+13]
- Scienze ambientali: per la classificazione e la previsione dei modelli climatici. [Jan+09]

1.2 Funzionamento

L'algoritmo KNN opera in modalità supervisionata, dove il dataset viene suddiviso in un set di addestramento (train set) e un set di test (test set). Il train set è utilizzato per addestrare l'algoritmo, mentre il test set viene impiegato per valutarne le prestazioni.

1.2.1 Calcolo delle distanze

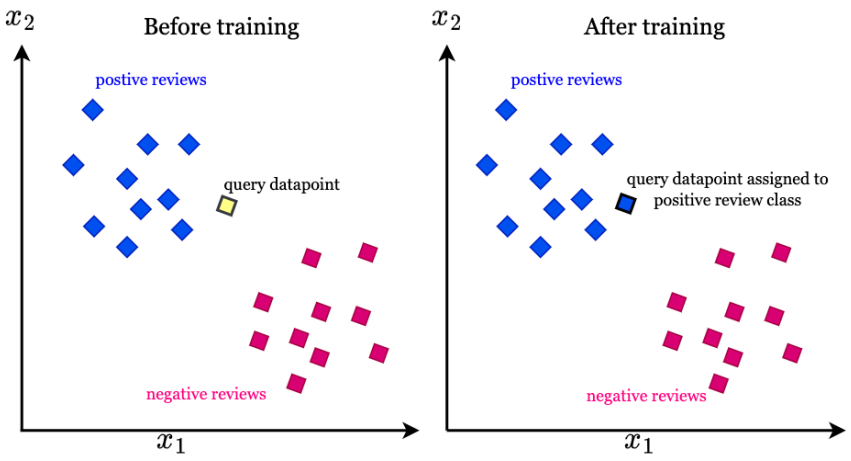
Per classificare un'istanza di test, KNN calcola la distanza tra questa e tutte le istanze presenti nel train set. Le metriche di distanza comunemente utilizzate includono la distanza euclidea, di Manhattan, di Minkowski e altre.

1.2.2 Selezione dei vicini

Dopo il calcolo delle distanze, KNN identifica le K istanze di addestramento più vicine (i "vicini più prossimi") all'istanza di test. Il parametro K è scelto a priori e influenza direttamente il processo decisionale dell'algoritmo. È importante inoltre scegliere un valore di K dispari, in modo da evitare parità nella selezione della classe.

1.2.3 Classificazione

Per la classificazione, KNN utilizza un meccanismo di voto di maggioranza tra le classi dei vicini più prossimi. La classe più frequente tra i vicini determina l'etichetta predetta per l'istanza di test.



1.2.4 Valutazione della precisione

La precisione dell'algoritmo viene valutata utilizzando il test set. È fondamentale scegliere un valore appropriato per K, ottimizzandolo mediante tecniche come la cross-validation o l'elbow method.

1.3 Pregi e difetti dell'algoritmo

L'algoritmo KNN si distingue per la sua semplicità concettuale e per l'intuitività nell'implementazione. Tuttavia, presenta anche alcune limitazioni significative. In particolare, l'algoritmo è computazionalmente costoso in termini di tempo e spazio, limitandone l'efficacia su dataset di grandi dimensioni.

Capitolo 2: Descrizione del Problema

L'algoritmo KNN è afflitto da limitazioni significative in termini di efficienza computazionale. Questo metodo è intrinsecamente lento e oneroso da eseguire, specialmente su dataset di grandi dimensioni o complessi a livello strutturale.
L'implementazione sequenziale di KNN comporta il calcolo delle distanze tra l'istanza da classificare e tutti i punti nel set di addestramento. Con l'aumentare della dimensione del dataset, questo processo diventa sempre più dispendioso in termini di tempo e risorse computazionali, andando a limitare la scalabilità dell'algoritmo.
Per superare queste sfide, si rende necessaria l'adozione di un approccio distribuito. L'implementazione distribuita di KNN, consente di distribuire il carico di lavoro su un cluster di macchine. Questo approccio sfrutta il calcolo parallelo per migliorare significativamente le prestazioni, consentendo di gestire grandi volumi di dati in modo più efficiente e riducendo i tempi di elaborazione.
Attraverso l'adozione di una soluzione distribuita, è possibile non solo mantenere l'accuratezza predittiva di KNN, ma anche migliorare radicalmente la scalabilità e l'efficienza operativa dell'algoritmo.

Capitolo 3: Implementazione

Per implementare l'algoritmo distribuito, è stato utilizzato il framework Ray per il linguaggio Python, e sfruttato il paradigma MapReduce per la suddivisione del carico.

3.1 Dataset

Per il dataset è stato selezionato Iris, un noto dataset per gli algoritmi di classificazione. Il dataset è composto da 150 istanze di fiori di iris, ognuno di questi caratterizzato da quattro attributi: lunghezza e larghezza del sepalo, lunghezza e larghezza del petalo. Il dataset Iris è comunemente utilizzato per la classificazione supervisionata, dove l'obiettivo è predire la specie di iris (Iris Setosa, Iris Versicolor, Iris Virginica) in base alle misurazioni morfologiche fornite.

3.1.1 Data augmentation

Dato che il dataset presenta solo 150 istanze, viene fatta una operazione di data augmentation per incrementare la dimensione del dataset. Per fare ciò viene utilizzato il metodo RandomOverSampler.
Partendo dal dataset di addestramento, il RandomOverSampler duplica casualmente campioni dalla classe minoritaria fino a che il numero di campioni in questa classe non raggiunge un numero predeterminato.

```
iris = load_iris()
X = iris.data
Y = iris.target

class_names = {0: 'Setosa', 1: 'Versicolor', 2: 'Virginica'}

augmented_data = RandomOverSampler(sampling_strategy={0: 1000, 1:
    1000, 2: 1000}, random_state=1)
X_resampled, Y_resampled = augmented_data.fit_resample(X, Y)
```

3.1.2 Aggiunta di rumore

Al nuovo dataset viene introdotto del rumore casuale secondo una distribuzione normale (Gaussiana). Questo procedimento permette di simulare in modo più efficacie delle nuove istanze, migliorando la generalizzazione dell'algoritmo durante la fase di train. Ulteriormente, il dataset viene diviso in 70% train, e 30% test. Il rumore Gaussiano è ampiamente utilizzato in campi come reti neurali e computer vision, con lo scopo di riprodurre la casualità presente in natura.

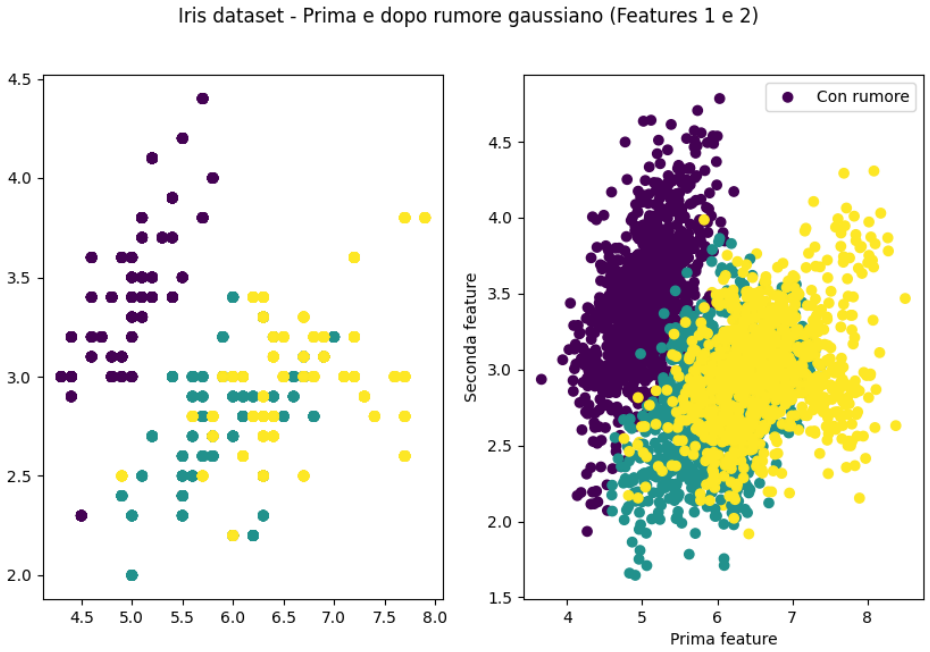


Figura 1: Dati prima e dopo il rumore gaussiano

```
noise_factor = 0.2
X_resampled_noisy = X_resampled + noise_factor * np.random.randn(*
    X_resampled.shape)

X_train, X_test, Y_train, Y_test = train_test_split(
    X_resampled_noisy, Y_resampled, test_size=0.3, random_state=1)
```

3.1.3 PCA (Principal Component Analysis)

Per semplificare il dataset viene effettuata un'analisi delle componenti principali (PCA). Questa tecnica permette di ridurre la dimensionalità del dataset mantenendo le informazioni delle istanze. Nel nostro caso, il dataset Iris contiene quattro caratteristiche, di conseguenza vengono ridotte a tre in modo che la prima componente principale spieghi la massima varianza nei dati, la seconda componente spieghi la seconda massima varianza, e così via. Questa tecnica inoltre, permette di organizzare e visualizzare i punti in uno spazio euclideo.

```
pca = PCA(n_components=3)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

3.2 Calcolo delle distanze

Per il calcolo delle distanze sono stati utilizzati due metodi principali: Euclideo, e Euclideo distribuito.

3.2.1 Distanza euclidea

La distanza euclidea è tra i metodi più utilizzati per il calcolo delle distanze nell'algoritmo KNN. La distanza euclidea tra due punti $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ e $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ nello spazio euclideo è definita come:

$$\text{dist}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

Questo calcolo viene implementato e semplificato con la libreria NumPy.

```
def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))
```

3.2.2 Matrice delle distanze

É possibile grazie alla quantità di risorse disponibili, utilizzare un altro approccio che migliora le prestazioni dell'algoritmo. Infatti, come vedremo dopo, ad ogni macchina viene assegnata una porzione del train set. Su di questa, vengono calcolate le distanze in broadcast, andando a formare una matrice in cui ogni riga corrisponde alle distanze di un punto dal test set rispetto a tutti i punti del train set. Questo metodo è facilmente applicabile se il carico viene suddiviso su più macchine, ma come vedremo, per quanto concerne l'algoritmo sequenziale (su singola macchina), questa implementazione riscontra problemi di memoria.

```
def euclidean_distance_matrix(a, b):
    return np.sqrt(np.sum((a[:, np.newaxis] - b) ** 2, axis=2))
```

Consideriamo un esempio pratico utilizzando il nostro dataset. Dataset di Training (3 punti, 3 dimensioni):

- Punto A: $[2.31, 1.45, 0.12]$ (Classe Setosa)
- Punto B: $[-1.02, 0.33, -0.23]$ (Classe Versicolor)
- Punto C: $[0.56, -1.78, 2.34]$ (Classe Virginica)

Dataset di Test (2 punti, 3 dimensioni):

- Punto X: $[1.00, 0.50, -0.10]$
- Punto Y: $[-0.50, -1.00, 1.50]$

Calcolo delle Distanze Euclidee:
La distanza euclidea tra due punti $P = [p_1, p_2, p_3]$ e $Q = [q_1, q_2, q_3]$ è data da:

$$d(P, Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}$$

Calcolo delle Distanze tra i Punti di Test e di Training:

- Distanze per il punto X $[1.00, 0.50, -0.10]$:

$$\approx [1.63\ 2.03\ 3.37]$$

- Distanze per il punto Y $[-0.50, -1.00, 1.50]$:

$$\approx [3.98\ 2.24\ 1.56]$$

Organizziamo queste distanze in una matrice delle distanze:

	A	B	C
X	1.63	2.03	3.37
Y	3.98	2.24	1.56

Supponiamo che i punti di training appartengano alle seguenti classi:

- A: Classe Setosa
- B: Classe Versicolor
- C: Classe Virginica

Per il punto X $[1.00, 0.50, -0.10]$:

- Ordiniamo le distanze nella prima riga: $[1.63(A), 2.03(B), 3.37(C)]$
- I k vicini più prossimi (supponendo $k = 1$) sono: A (distanza 1.63)
- Classe predetta per X: Classe Setosa

Per il punto Y $[-0.50, -1.00, 1.50]$:

- Ordiniamo le distanze nella seconda riga: $[1.56(C), 2.24(B), 3.98(A)]$
- I k vicini più prossimi (supponendo $k = 1$) sono: C (distanza 1.56)
- Classe predetta per Y: Classe Virginica

3.3 Algoritmo sequenziale

Algorithm 1 KNN Sequenziale

```
1: procedure KNN_CLASSIFY(X_train, Y_train, X_test, k)
2:     prediction ← empty list
3:     for test_point in X_test do
4:         distances ← empty list
5:         for train_point, c in zip(X_train, Y_train) do
6:             d ← calculate_euclidean_distance(test_point, train_point)
7:             append (c, d) to distances
8:         end for
9:         sort distances by distance d
10:        k_nearest_neighbors ← first k elements of distances
11:        k_nearest_classes ← extract classes from k_nearest_neighbors
12:        most_common_class ← most_common_class(k_nearest_classes)
13:        append most_common_class to prediction
14:    end for
15:    return prediction
16: end procedure
```

Il funzionamento dell'algoritmo KNN è relativamente semplice e intuitivo. I dati X rappresentano le istanze del dataset, mentre Y rappresentano le classi assegnate a queste istanze. Per ogni istanza del set di test (punto da classificare), si calcolano le distanze da ciascun punto del set di addestramento. Questo processo comporta l'uso di un loop annidato, evidenziando l'inefficienza temporale dell'algoritmo. Inoltre, l'algoritmo utilizza diverse strutture dati per tenere traccia delle distanze calcolate, dimostrando anche un'efficienza spaziale limitata.

3.3.1 Miglioramenti algoritmo sequenziale

Fino ad un certo numero di punti, è possibile utilizzare la matrice delle distanze. Queso comporta dei cambiamenti a livello di logica e pseudocodice. Questa sarà la logica utilizzata nei test successivi, in quanto l'algoritmo sequenziale classico non soffre di problemi di memoria, ma comporta una complessità temporale di $O(n^2P)$, estremamente lento per essere paragonato all'implementazione distribuita. Questa implementazione invece, soffre di problemi di memoria, ma si ritrova estremamente più veloce rispetto alla classica soluzione sequenziale.

Algorithm 2 KNN Classification

```
1:  procedure EUCLIDEAN_DISTANCE_MATRIX(a, b)
2:      return distance_matrix
3:  end procedure
4:  procedure KNN_CLASSIFY(X_train, Y_train, X_test, k)
5:      prediction ← empty list
6:      distances_matrix ← EUCLIDEAN_DISTANCE_MATRIX(X_test, X_train)
7:      print distances_matrix
8:      for _, distances in enumerate(distances_matrix) do
9:          k_nearest_neighbors ← argsort(distances)[:k]
10:         k_nearest_classes ← Y_train[k_nearest_neighbors]
11:         most_common_class ← most common class in k_nearest_classes
12:         append most_common_class to prediction
13:     end for
14:     return prediction
15: end procedure
```

3.3.2 Codice algoritmo sequenziale

Per l'algoritmo sequenziale si è utilizzato NumPy sia per il calcolo delle distanze che per il sort delle distanze. Per la selezione della classe da assegnare al punto invece, è stata utilizzata la classe Counter di Python, con il metodo most_common.

```
def knn_classify(X_train, Y_train, X_test, k):
    prediction = []
    distances_matrix = euclidean_distance_matrix(X_test, X_train)
    print(distances_matrix)
    for _, distances in distances_matrix:
        k_nearest_neighbors = np.argsort(distances)[:k]
        k_nearest_classes = Y_train[k_nearest_neighbors]
        most_common_class = Counter(k_nearest_classes).most_common
    (1)[0][0]
        prediction.append(most_common_class)

    return prediction
```

3.4 Algoritmo distribuito

Per l'implementazione dell'algoritmo distribuito è stato utilizzato il paradigma MapReduce. Il paradigma si suddivide in due fasi principali:

- Map: La fase di mappatura consiste nell'applicazione di una funzione a ciascuna partizione del dataset fornita alla macchina del cluster. Nel nostro caso, questa fase viene utilizzata per calcolare le distanze tra i punti di test e i punti di addestramento, e la selezione delle K classi più vicine.

- Shuffle: I dati ottenuti dai mapper vengono "mescolati" e suddivisi in chunk per essere consumati dai reducer.

- Reduce: Nella fase di riduzione, i reducer prendono in input i risultati dei relativi mapper per aggregarli. Nel KNN, i reducer determinano la classe più comune di ogni punto appartenente al chunk di dati ottenuto dal mapper.

Pseudocodice mapper e reducer

Algorithm 3 KNN Mapper

```
1: procedure KNN_Mapper(X_train, Y_train, X_test_partition, partition_index, k)
2:     distances_matrix ← Distance_Matrix(X_test_partition, X_train)
3:     mapper_results ← empty list
4:     for i, distances in enumerate(distances_matrix) do
5:         k_nearest_neighbors ← argsort(distances)[:k]
6:         k_nearest_classes ← Y_train[k_nearest_neighbors]
7:         append (partition_index + i, k_nearest_classes) to mapper_results
8:     end for
9:     return mapper_results
10: end procedure
```

Algorithm 4 KNN Reducer

```
1: procedure KNN_Reducer(mapper_results_partition)
2:     predictions ← empty list
3:     for i, k_nearest_classes in mapper_results_partition do
4:         most_common_class ← most common class in k_nearest_classes
5:         append (i, most_common_class) to predictions
6:     end for
7:     return predictions
8: end procedure
```

Come possiamo notare, il carico viene suddiviso tra mapper e reducer: la funzione di map calcola la matrice delle distanze della partizione assegnata a quel mapper, andando infine a trovare le K istanze più vicine al punto. La funzione reduce andrà infine a trovare la moda delle classi dei punti di test forniti dai mapper, assegnando la label all'istanza.

3.4.1 Codice algoritmo distribuito

Funzione assegnata ai mappers:

```
@ray.remote
def knn_mapper(X_train, Y_train, X_test_partition, partition_index,
    k):
    distances_matrix = euclidean_distance_matrix(X_test_partition,
   X_train)
    mapper_results = []
    for i, distances in enumerate(distances_matrix):
        k_nearest_neighbors = np.argsort(distances)[:k]
        k_nearest_classes = Y_train[k_nearest_neighbors]
        mapper_results.append((partition_index + i,
   k_nearest_classes))
        gc.collect()
    return mapper_results
```

Shuffling dei dati:

```
np.random.shuffle(mapper_results)
reducer_data = [[] for _ in range(num_reducers)]
for idx, item in enumerate(mapper_results):
    reducer_data[idx % num_reducers].append(item)
```

Funzione assegnata ai reducer:

```
@ray.remote
def knn_reducer(mapper_results_partition):
    predictions = []
    for i, k_nearest_classes in mapper_results_partition:
        most_common_class = Counter(k_nearest_classes).most_common
   (1)[0][0]
        predictions.append((i, most_common_class))
        gc.collect()
    return predictions
```

Possiamo notare che la logica è pressoché simile a quella dell'algoritmo sequenziale, ma viene suddivisa in map e reduce.

Funzione KNN:

```
def knn(k, num_mappers, num_reducers):
    X_test_partitions = np.array_split(X_test_pca, num_mappers)

    start = perf_counter()

    mapper_futures = [knn_mapper.remote(X_train_pca, Y_train,
    partition, i * len(partition), k) for i, partition in enumerate(
    X_test_partitions)]

    mapper_results = []
    for future in tqdm(ray.get(mapper_futures), desc="Mappatura"):
        mapper_results.extend(future)

    np.random.shuffle(mapper_results)
    reducer_data = [[] for _ in range(num_reducers)]
    for idx, item in enumerate(mapper_results):
        reducer_data[idx % num_reducers].append(item)

    reducer_futures = [knn_reducer.remote(chunk) for chunk in
    reducer_data]
    reducer_results = ray.get(reducer_futures)

    predictions = [0] * len(X_test_pca)
    for reducer_result in reducer_results:
        for i, pred in reducer_result:
            predictions[i] = pred

    accuracy = accuracy_score(Y_test, predictions)
    end = perf_counter()

    conf_matrix = confusion_matrix(Y_test, predictions)

    class_report = classification_report(Y_test, predictions,
    target_names=[class_names[i] for i in range(3)])

    return predictions, accuracy, conf_matrix
```

3.4.2 Diagramma MapReduce

È riportato il diagramma MapReduce, con l'obiettivo di mostrare graficamente il funzionamento del paradigma.

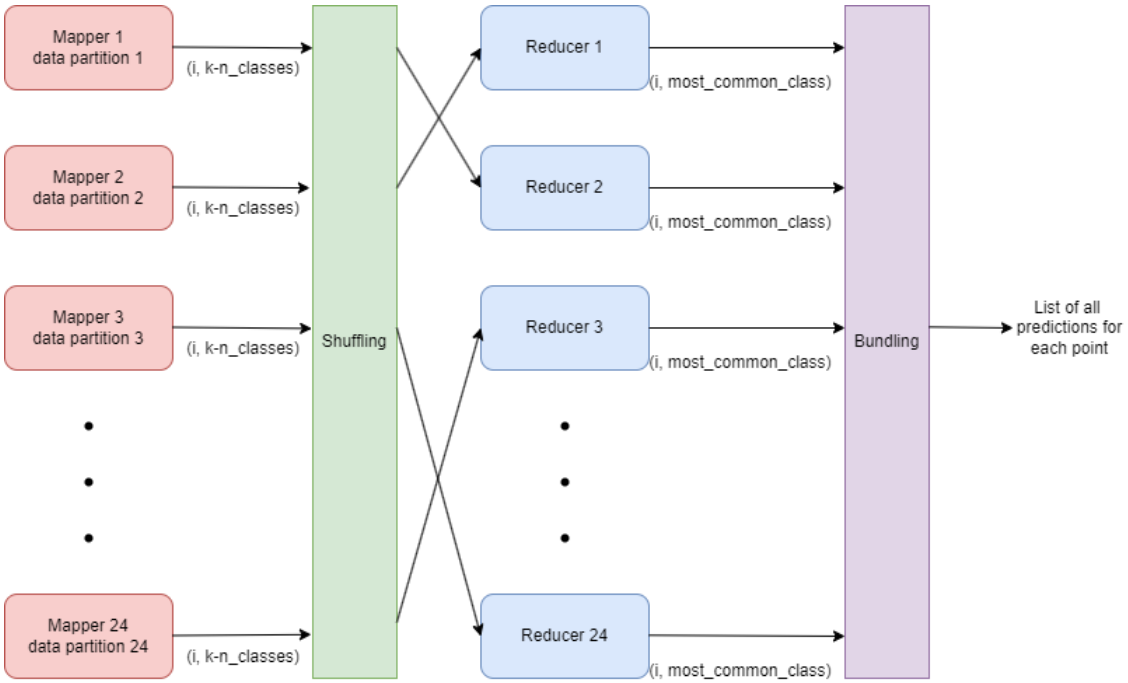


Figura 2: Diagramma MapReduce

Ad ogni mapper viene assegnata una porzione del train set. Su di questa vengono calcolate le K distanze più vicine per ogni punto appartenente alla partizione. Le tuple ottenute dai mapper vengono organizzate e suddivise nello shuffling, per poi essere mandate ai reducer. I reducer calcoleranno e assegneranno la classe ad ogni punto della partizione ottenuta dai mapper. Infine, i risultati di ogni reducer vengono combinati per ottenere le previsioni finali.

Capitolo 4: Risultati sperimentali

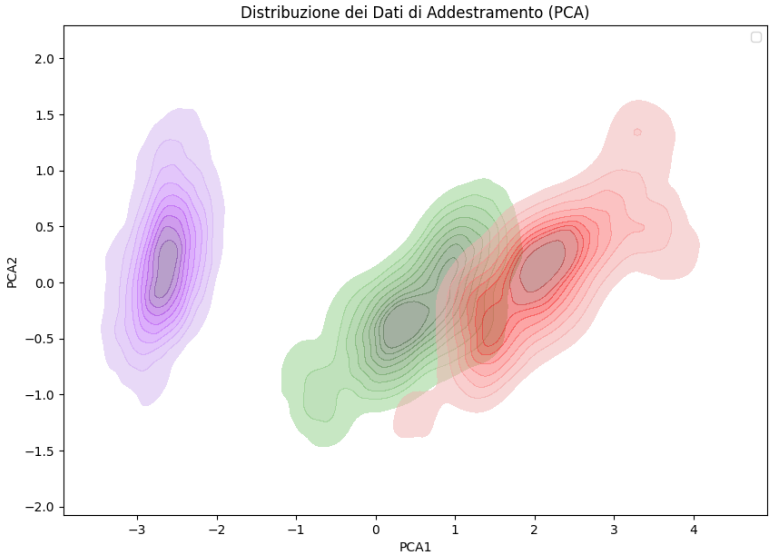
L'algoritmo è stato testato all'interno del laboratorio fornito dall'università. Per i test, sono state utilizzate 8 macchine, per un totale di 64 core di CPU. I test per l'algoritmo sequenziale sono stati eseguiti su una di queste macchine (ogni macchina possiede 32GB di RAM).
Per il calcolo sono stati utilizzati 24 mapper e 24 reducer, in quanto utilizzare un numero elevato di mapper e reducer portavano a svantaggi come overhead e una precisione generale ridotta. Per mantenere quindi delle buone prestazioni è stato trovato un compromesso nel numero di mapper e reducer.

4.1 Risultati del KNN

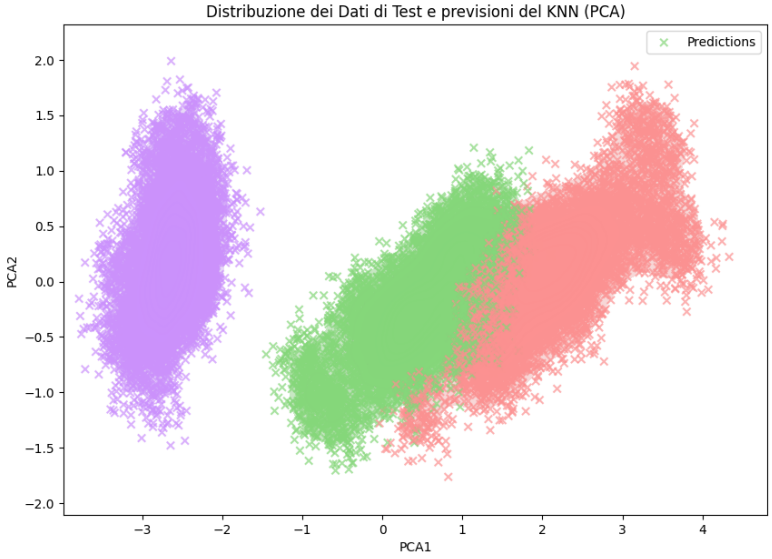
In questa sezione, analizziamo i risultati ottenuti dal KNN. Per ottenere il massimo delle informazioni, è stato utilizzato il dataset più grande (90.000 punti).

4.1.1 Dati di train e test

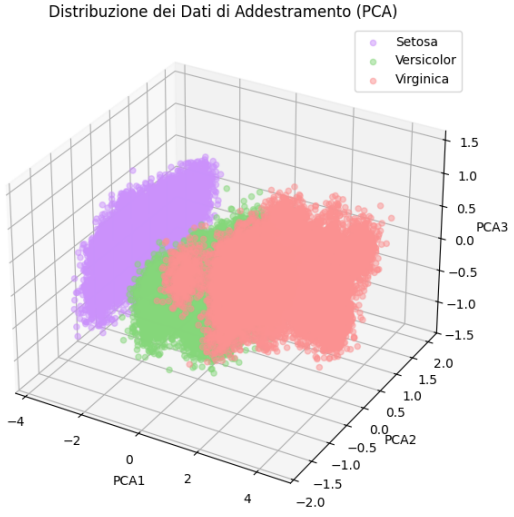
Sono riportati i dati di train e test. Possiamo notare dalla heatmap in che modo il rumore gaussiano ha agito sul dataset, concentrando i punti verso il centro del cluster.



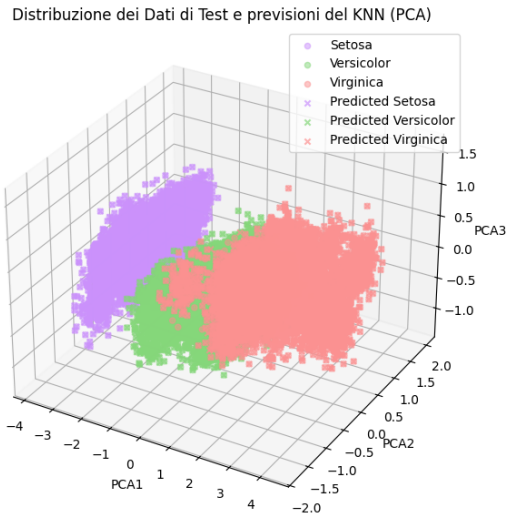
Seguono i dati di test, in cui è possibile notare le previsioni (x nel grafico) effettuate dall'algoritmo. In questo caso, con 90000 punti, la precisione è del 95.70%.



Grazie alla PCA (Principal Component Analysis) è possibile inoltre effettuare plot tridimensionali, che rendono più intuitiva la visualizzazione dei dati.

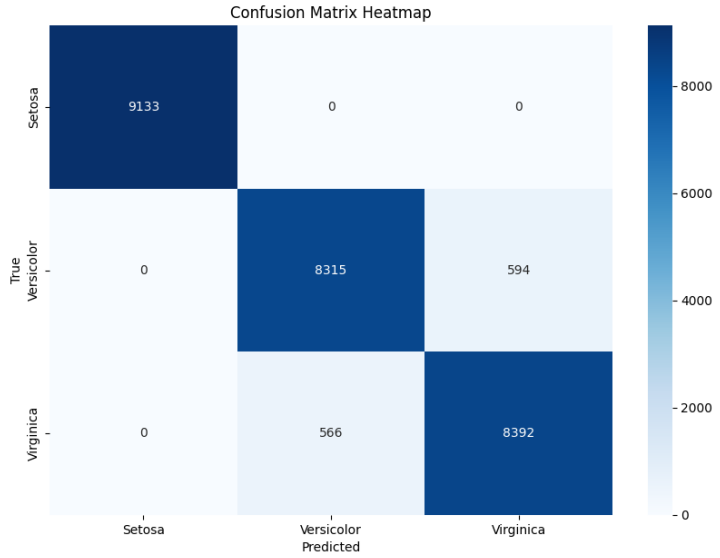


Seguono i dati di test in tre dimensioni. Ogni x corrisponde alla previsione dell'algoritmo.



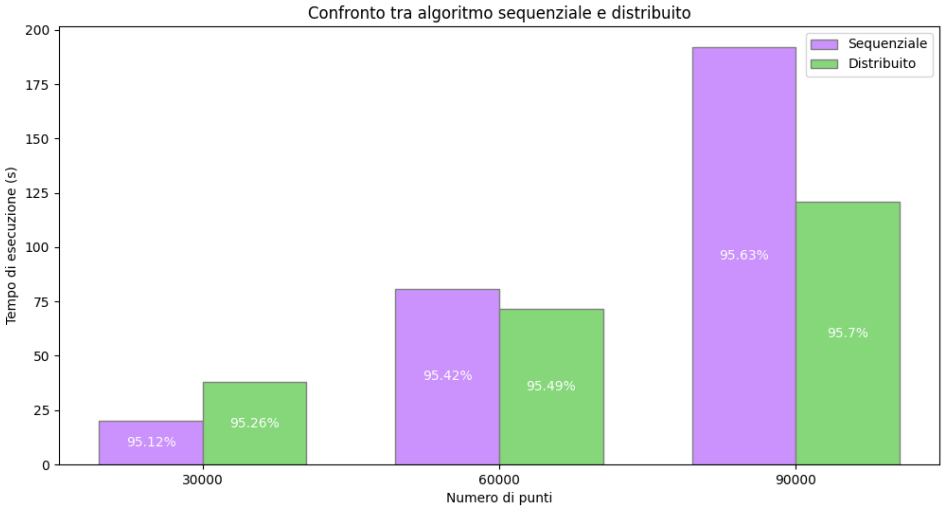
4.1.2 Matrice di confusione

Dai grafici è evidente come il dataset si distribuisca nello spazio. La classe "Setosa" è chiaramente separata dalle altre due classi, formando un cluster isolato. Questa distinzione nello spazio consente all'algoritmo di classificare accuratamente i fiori appartenenti alla classe Setosa. Tuttavia, per le altre due classi, l'algoritmo mostra una maggiore propensione agli errori a causa della loro stretta relazione nello spazio. Questo fenomeno è confermato dai risultati della matrice di confusione:



4.2 Confronto distrubuito vs sequenziale

In questa sezione sono riportati i risultati di entrambi gli approcci.



Possiamo notare come generalmente l'approccio distribuito sia più veloce rispetto al sequenziale, soprattuto nel momento in cui i punti aumentano. Per problemi di memoria relativi all'algoritmo sequenziale, non è stato possibile confrontare gli algoritmi utilizzando più punti.

4.2.1 Punto di crossover

Intorno alle 60.000 istanze si raggiunge il punto di crossover, dove l'algoritmo distribuito diventa più veloce dell'algoritmo sequenziale. Tuttavia, l'algoritmo sequenziale è preferibile per un numero ridotto di punti, poiché l'implementazione distribuita può subire rallentamenti dovuti all' overhead.

4.2.2 Precisione

Dai grafici emerge chiaramente l'importanza della dimensione del set di addestramento sulla precisione del modello. È evidente che aumentando il numero di istanze nel set di addestramento, si ottiene un miglioramento incrementale nella precisione delle previsioni.

4.2.3 Nota sull'algoritmo sequenziale

È importante notare che nel confronto è stato utilizzato l'algoritmo sequenziale migliorato. Questa scelta è stata motivata dalla complessità temporale esponenziale dell'approccio sequenziale classico, il quale mostrava tempi di esecuzione che superavano l'ordine delle ore con il dataset più grande.

Capitolo 5: Conclusioni e futuri sviluppi

Dai risultati ottenuti emerge chiaramente che per un algoritmo computazionalmente intensivo come il KNN, l'adozione di un approccio distribuito offre vantaggi significativi in termini di efficienza temporale e spaziale. È evidente come l'algoritmo KNN sia semplice da implementare e capace di effettuare previsioni accurate quando i dati sono di buona qualità. Tuttavia, è importante notare che anche l'implementazione distribuita presenta limitazioni, specialmente riguardo alla sincronizzazione e all'overhead operativo.

5.1 Miglioramenti

Per migliorare ulteriormente l'algoritmo, si potrebbe adottare un approccio per determinare in modo efficace un valore significativo di K che non sia scelto arbitrariamente. Trovare questo valore ottimale può migliorare la precisione delle previsioni e ridurre il rischio di over o underfitting del modello.

Inoltre, per affrontare i persistenti problemi di gestione della memoria associati al KNN, noti nella comunità scientifica, potrebbe risultare vantaggioso esplorare nuovi metodi e strutture dati. Questi approcci potrebbero contribuire a ottimizzare l'uso della memoria durante l'elaborazione dei dati, riducendo l'impatto negativo sulla scalabilità e migliorando l'efficienza complessiva dell'algoritmo.

È importante considerare che nonostante le sfide legate alla gestione della memoria, il KNN rimane un potente strumento per la classificazione e la regressione.

Riferimenti bibliografici

[Jan+09] Zahoor Jan et al. «Seasonal to inter-annual climate prediction using data mining KNN technique». In: *Wireless Networks, Information Processing and Systems: International Multi Topic Conference, IMTIC 2008 Jamshoro, Pakistan, April 11-12, 2008 Revised Selected Papers*. Springer. 2009, pp. 40–51.

[Alk+13] Khalid Alkhatib et al. «Stock price prediction using k-nearest neighbor (kNN) algorithm». In: *International Journal of Business, Humanities and Technology* 3.3 (2013), pp. 32–44.

[Raj+20] Nazneen Fatema Rajani et al. «Explaining and improving model behavior with k nearest neighbor representations». In: *arXiv preprint arXiv:2010.09030* (2020).

[XB20] Wenchao Xing e Yilin Bei. «Medical Health Big Data Classification Based on KNN Classification Algorithm». In: *IEEE Access* 8 (2020), pp. 28808–28819. DOI: 10.1109/ACCESS.2019.2955754.

[ZL22] Jianfeng Zou e Hui Li. «Precise Marketing of E-Commerce Products Based on KNN Algorithm». In: *Computational Intelligence and Neuroscience* 2022.1 (2022), p. 4966439. DOI: https://doi.org/10.1155/2022/4966439. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2022/4966439. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2022/4966439.