Elaborato di Analisi e Prestazioni di Internet

QoS on Railway Paths

Prof. Antonio Pescapè

Mario Varlese - Mat. M63/778
Andrea Fresa - Mat. M63/822
Marco Francesco Patriarca - Mat. M63/772
Andrea Pinto - Mat. M63/764

11 giugno 2019

Indice

Traccia

Una compagnia ferroviaria norvegese vuole valutare le prestazioni della connessione dati in funzione di coordinate gps e di metadati legati agli access point sulle varie tratte disponibili. L'obiettivo è quello di capire come varia la QoS in funzione del particolare landscape della Norvegia, in modo tale da poterla migliorare.
Il dataset riporta misure di downlink (DL) rate, uplink (UL) rate e latenza (RTT) effettuate da dispositivi mobili installati sui treni che percorrono la rete ferroviaria norvegese. I dati disponibili sono ottenuti da un totale di 40 nodi di misura installati su 20 treni. A questi dati si aggiungono due ulteriori set di misurazioni:

- Modem metadata: Misurazioni event-based (ad esempio, operatore mobile, tecnologia del mezzo trasmissivo, potenza del segnale) collezionate dai nodi stessi. I valori sono memorizzati ogni qual volta si percepisce un cambiamento o in caso contrario ogni 30 secondi.
- Misurazioni GPS: posizione e velocità (latitudine e longitudine) forniti dalla compagnia ferroviaria con granularità di 10 secondi.

1 Introduzione

L'obiettivo primario del lavoro svolto è stato quello di istruire vari regressori affinché, in funzione dei dati in ingresso, realizzino una stima statistica delle uscite attese. In particolare, si è focalizzata l'attenzione sulla valutazione delle prestazioni del **downlink rate**, espresso in **kilobytes per secondo**. A valle dell'analisi dei due dataset a disposizione, è stato testato il comportamento di vari algoritmi di regressione, effettuando differenti operazioni di *preprocessing* e cercando di individuare i migliori *iperparametri* per ognuno di essi, per migliorarne le prestazioni.
A partire dal problema di regressione, si è poi generato un problema di classificazione equivalente, discretizzando opportunamente l'uscita (res_dl_kbps) in 4 classi:

1. **Low**: per i valori di y minori di 5 Mbps;
2. **Medium**: per i valori di emphy compresi tra 5 Mbps e 15 Mbps;
3. **High**: per i valori di y compresi tra 15 Mbps e 30 Mpbs;
4. **Very High**: per i valori di y maggiori di 30 Mbps.

Nel seguito saranno descritte le metodologie di preprocessing, di regressione e di classificazione utilizzate. Infine saranno descritti e commentati i risultati ottenuti.

2 Descrizione del Dataset e Preprocessing

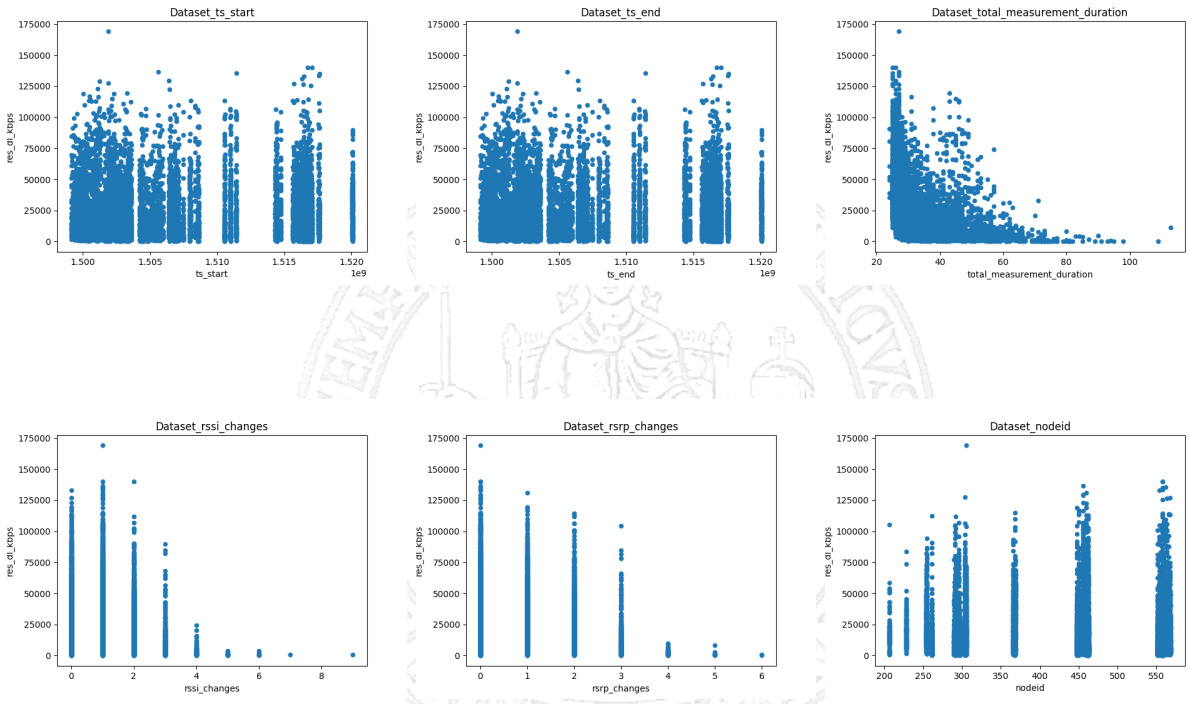
In questa sezione verranno analizzate le metodologie di preprocessing attuate dopo aver constatato la natura del dataset.

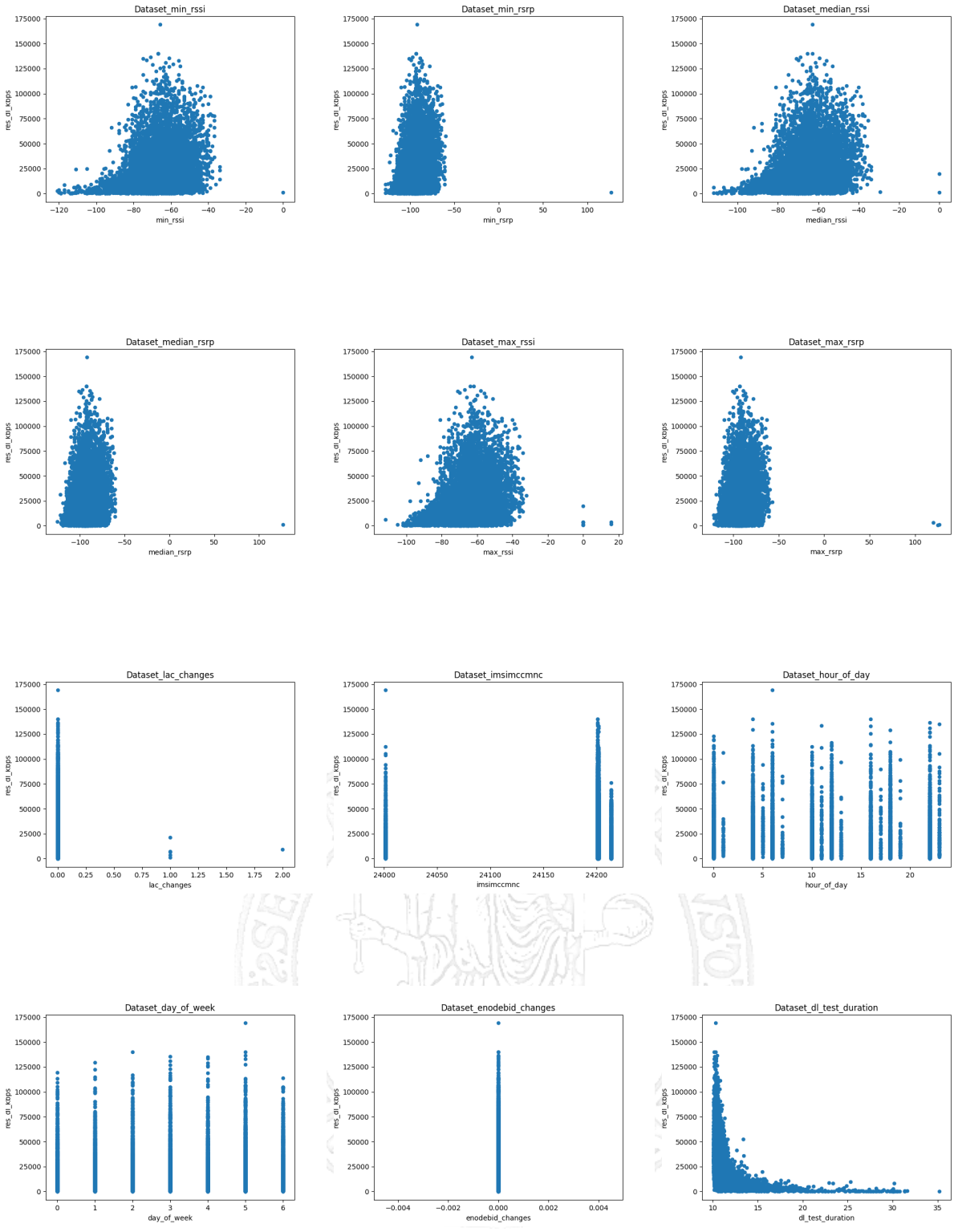
2.1 Analisi dei dati a disposizione

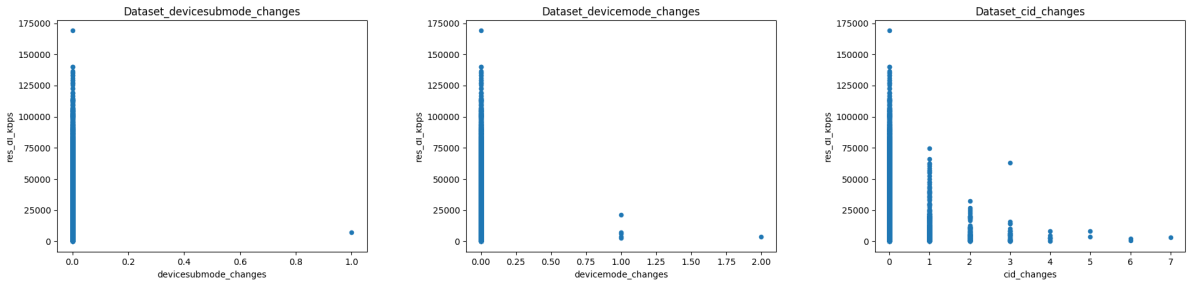
Prima di poter effettuare qualsiasi operazione di regressione è stato necessario analizzare la natura del dataset a disposizione, al fine di valutare:

- significato di ciascuna feature;
- eventuali *outlier*;
- eventuali *missing values*.

Il primo approccio al dataset è stato effettuato considerando tutte le features a disposizione: si è notata la presenza di numerosi missing values, oltre che quella di alcuni outlier su alcune features. Di seguito sono mostrate le distribuzioni delle varie features del dataset **QoS_railway_paths_nodeid_iccid_feature_extraction.csv**




Dataset_min_rssi
res_dl_kbps
min_rssi
Dataset_min_rsrp
res_dl_kbps
min_rsrp
Dataset_median_rssi
res_dl_kbps
median_rssi
Dataset_median_rsrp
res_dl_kbps
median_rsrp
Dataset_max_rssi
res_dl_kbps
max_rssi
Dataset_max_rsrp
res_dl_kbps
max_rsrp
Dataset_lac_changes
res_dl_kbps
lac_changes
Dataset_imsimccmnc
res_dl_kbps
imsimccmnc
Dataset_hour_of_day
res_dl_kbps
hour_of_day
Dataset_day_of_week
res_dl_kbps
day_of_week
Dataset_enodebid_changes
res_dl_kbps
enodebid_changes
Dataset_dl_test_duration
res_dl_kbps
dl_test_duration



Per evitare il fenomeno dell'*overfitting* durante la fase di apprendimento degli algoritmi di regressione, si è deciso di effettuare un'analisi della correlazione dei dati, al fine di scegliere solo le features che tra di loro risultassero meno correlate. Di seguito, in Figura 1, si riporta la *scatter_matrix* che illustra eventuali correlazioni tra le features:

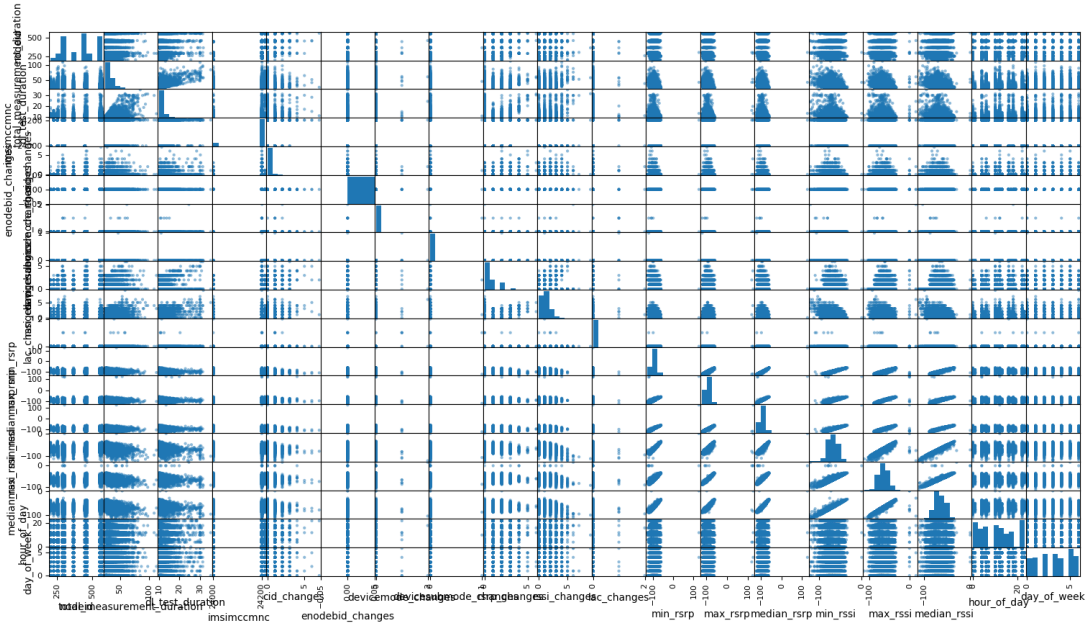


Figura 1: Scatter Matrix – correlazione delle features

Come si evince dalla Figura 1, ci sono alcune variabili che presentano correlazione. Dunque, la procedura di selezione delle features corretta deve tener conto di tale peculiarità del dataset.

2.2 Workflow

Selezionato il dataset su cui lavorare, è possibile riassumere la procedura di classificazione/regressione in poche macrofasi fondamentali:

1. trattamento dei missing values nel dataset;

2. selezione delle features;
3. suddivisione del dataset in una porzione per il *training* e una porzione per il *testing* degli algoritmi di intelligenza artificiale;
4. utilizzo di regressori (classificatori[1]) e ricerca di iperparametri ottimi;
5. calcolo dello score e salvataggio dei risultati.

In Figura 2 è mostrato un grafico che illustra il flusso seguito.

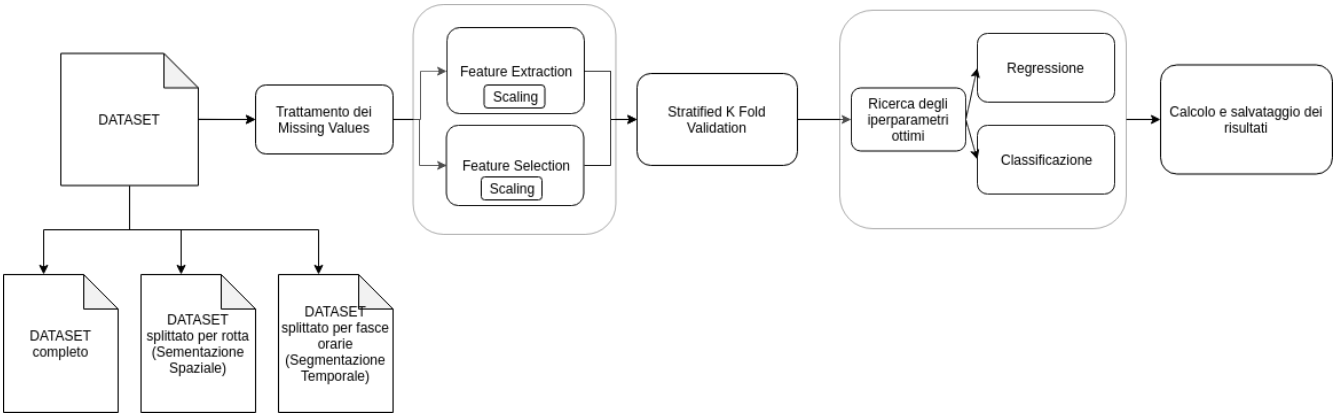


Figura 2: Workflow

Prima di entrare nel dettaglio di ciascuna fase è bene evidenziare che tutta questa procedura è stata ripetuta:

- per il problema di regressione;
- per il problema di classificazione equivalente.

Inoltre, fissato il problema (classificazione o regressione), tale procedura è stata ripetuta:

- utilizzando il dataset *QoS_railway_paths_nodeid_iccid_feature_extraction*
- effettuando una segmentazione temporale sul dataset;
- effettuando una segmentazione spaziale (per rotta) sul dataset.

Infine, fissato il problema (regressione o classificazione equivalente) e il dataset su cui lavorare, si è iterata la procedura al fine di trovare:

- il numero di componenti principali ottimo da utilizzare per la fase di *feature extraction*;
- il numero K di features migliori da selezionare durante la fase di *feature selection.*

[1] nel caso di problema di classificazione equivalente

2.3 Dataset

Come illustrato in precedenza, i vari esperimenti sono stati effettuati andando a valutare cosa succede utilizzando tutto il dataset, utilizzando un dataset splittato per rotta (splitting spaziale) oppure utilizzando un dataset suddiviso per fascia oraria (slitting temporale). Sui tre dataset, al fine di trovare il numero ottimo di componenti principali per la fase di feature extraction (dimensionality reduction – PCA) e il numero ottimo K di features per la fase alternativa di feature selection (dimensionality reduction – Select K Best), si è deciso di far variare, in maniera ricorsiva, il numero di componenti/features, valutando contestualmente lo score generato dai classificatori/-regressori (**accuratezza** nel caso del problema di classificazione e **MSE** nel caso del problema di regressione).

2.3.1 Segmentazione temporale

Per quanto riguarda la **segmentazione temporale**, si è scelto di utilizzare solamente due fasce orarie (da 12 ore), poiché la dimensione del dataset non è elevatissima. Dunque, sulla base di ciò, sono stati generati due dataset:

- Day.csv;
- Night.csv.

2.3.2 Segmentazione spaziale

Per la **segmentazione spaziale**, invece, è stato necessario, al fine di generare tanti dataset quante sono le rotte, utilizzare i metadati forniti dal dataset **QoS_railway_paths_latlong_nsb_gps_segment_mapping_mobility_apu2**. In particolare, in quest'ultimo dataset sono presenti quattro informazioni fondamentali:

- ID di rotta;
- nodeid;
- timestamp di inizio misurazione;
- timestamp di fine misurazione.

Dunque, sulla base dell'id di rotta, si sono selezionati tutti gli elementi del dataset corrispondenti a quella specifica rotta e sono poi state utilizzate le features (nodeid, timestampStart, TimestampEnd) come indici per prelevare gli elementi dal dataset *QoS_railway_paths_nodeid_iccid_feature_extraction*.

2.4 Trattamento dei *missing values*

Dopo aver constatato l'assenza di una quantità significativa di dati per alcune features, si è deciso di sottoporre il dataset ad un processo di **imputing** che consente di riempire i missing values. Sono state testate due tecniche per effettuare tale operazione:

- **Mean Imputing**: consiste nel sostituire i missing values di una colonna con il valore medio di tale colonna;

- **Mode Imputing**; consiste nel sostituire i missing values di una colonna con il valore più frequente di tale colonna.

A valle di tale operazione, sono stati effettuati degli esperimenti di regressione, argomentati meglio in seguito, su tutte le features, dai quali si è notata l'assenza di una differenza significativa in termini di prestazioni rispetto all'utilizzo di una tecnica di imputing piuttosto che dell'altra. Per tale motivo, tutti i successivi esperimenti sono stati eseguiti utilizzando come tecnica di imputing solamente quella che prevede il riempimento dei missing values con i rispettivi valori medi di ciascuna colonna.

2.5 Selezione delle feature

Per la fase di selezione delle features sono state utilizzate due differenti metodologie:

- **feature extraction**: consiste nell'applicazione della PCA, che effettua una trasformazione di spazio prendendo in considerazione soltanto le colonne ottenute a valle di tale processo aventi una maggiore varianza;
- **feature selection**: consiste nell'applicazione della funzione *SelectKBest()*, che assegna uno score a ciascuna feature e seleziona le k features aventi uno score maggiore. Tale score è funzione della correlazione.

Entrambi tali approcci necessitano di una procedura di **scaling** dei dati. In particolare, per quanto riguarda la metodologia che contempla l'utilizzo della PCA, non si è dovuto scegliere un metodo di scaling appropriato, poiché tale tecnica prevede essa stessa l'utilizzo dello **standard scaling** (media nulla e varianza unitaria) come fase propedeutica alla trasformazione dello spazio. Per quanto riguarda, invece, la tecnica di feature selection, sono stati effettuati diversi test per capire quali tra i seguenti approcci di scaling fosse il più appropriato:

- **min-max scaling**: consiste nello scalare i dati nell'intervallo [0,1], considerando 0 il valore minimo e 1 il valore massimo;
- **standard scaling**: consiste nello scalare i dati in valori a media nulla e varianza unitaria (tale metodo è quello usato dalla PCA);
- **robust scaling**: consiste nello scalare i dati mediante l'utilizzo dei percentili.

A valle di tali esperimenti si è capito che, per la metodologia di feature selection, il meccanismo di scaling più adatto da utilizzare è il **robust scaling**, perché, a differenza degli altri due, è robusto agli outliers.

2.5.1 Feature extraction

La fase di **feature extraction** ha previsto, dunque, l'applicazione della Principal Component Analysis. Questa consiste nel generare un ulteriore spazio multi-dimensionale che sia combinazione lineare delle features del dataset. Le componenti principali generate saranno incorrelate tra di loro. Le variabili generate spiegano, in ordine decrescente, la varianza del dataset. Quindi, la prima componente principale spiegherà la maggior percentuale di varianza del dataset originale, l'ultima componente principale spiegherà la minor parte della varianza del dataset. L'obiettivo

principale è quello di ridurre il numero di dimensioni nel dataset, quindi scegliere il numero di features ottime che spieghino assieme la maggior parte del dataset.
Per valutare il numero di componenti da utilizzare è necessario valutare la derivata della varianza in funzione del numero di componenti principali. Se tale valore scende al di sotto di una certa soglia, allora significa che la varianza non aumenta sufficientemente all'aumentare del numero di componenti principali, quindi è opportuno fermarsi.
Tale valore di soglia, in genere, risulta essere, in un grafico che mette in relazione il numero di componenti e la percentuale di varianza spiegata, coincidente con un punto a sinistra del ginocchio della curva.
È stato provato un approccio di questo genere, ma i risultati in termini di MSE non erano soddisfacenti. Quindi è stato preferito aumentare il numero di componenti. I risultati ottenuti utilizzando la procedura di feature extraction sono comunque stati meno soddisfacenti rispetto a quelli ottenuti con la procedura di feature selection.

2.5.2 Feature selection

Per la fase di **feature selection** è stata utilizzata la metodologia *SelectKBest*. Tale metodologia prevede il calcolo di uno score per ogni feature (**F-regression**, in caso di problemi di regressione; **F-classif**, in caso di problemi di classificazione).
È comunque necessario specificare che lo score attribuito ad ogni variabile sia calcolato in maniera indipendente dalle altre variabili. Ciò vuol dire che tale score è attribuito ad ogni singola variabile e non alle varie combinazioni di esse.
Un possibile miglioramento di questa fase potrebbe consistere nell'applicazione di un approccio che valuti lo score tenendo in considerazioni varie combinazioni di più features.

3 Regressione e Classificazione

3.1 Regressione

L'obiettivo della regressione è quello di ottenere una variabile in funzione di altre variabili opportunamente pesate.
Per quanto riguarda i metodi di regressione, sono stati utilizzati differenti approcci.
In particolare, sono stati utilizzati i seguenti regressori:

- K-Nearest Neighbors;
- Decision Tree;
- Random Forest.

3.1.1 K-Nearest Neighbors

Con l'algoritmo di regressione **K-Nearest Neighbors**, durante la fase di training viene diviso lo spazio delle uscite in più porzioni, in base a ciò che viene appreso. In fase di testing, poi, la scelta del valore y che viene predetto dal regressore è dipendente dai suoi k vicini. Il valore k, quindi,

è un valore che deve essere scelto opportunamente e che, dunque, deve essere necessariamente tarato. Per la scelta del numero di vicini k da considerare, è stato utilizzato una *grid search cross validation*. Con questo approccio, scelto un modello ed un insieme di iperparametri da tarare con opportuni valori, viene effettuato un prodotto cartesiano tra i possibili iperparametri e viene scelta la combinazione ottima. Un ulteriore valore da considerare è la metrica utilizzata per valutare la distanza dai vicini. Anche questo è un parametro da scegliere opportunamente.
È inoltre opportuno considerare la differenza tra classificazione con KNN e regressione con KNN: nel primo caso, la classe di appartenenza è scelta in base alla maggioranza dei voti dei k vicini, nel secondo caso, invece, essendo l'uscita continua e non discreta, viene effettuata una media. La media può essere sia pesata che non pesata. Non conoscendo le misurazioni effettuate, non è stato possibile valutare il rumore su di esse, quindi è stato scelto un approccio con una media standard.

3.1.2 Decision Tree

Il secondo regressore utilizzato è il **Decision Tree**. Con tale algoritmo, nella fase di training viene generata una struttura ad albero in memoria e, in base a questa, viene valutata l'opportuna uscita in fase di testing. Il processo risulta essere di tipo iterativo e, quindi, è necessario definire una massima profondità dell'albero. Il motivo è il seguente: se ciò non fosse fatto, l'albero potrebbe procedere generando una struttura con tante foglie quanti sono i valori visti in fase di training, ma questo causerebbe **overfitting**. Quindi è necessario stabilire una massima profondità dell'albero. Oltre a questo parametro, è necessario definire anche il numero minimo di occorrenze in un percorso per effettuare uno splitting dell'albero. Queste due variabili sono iperparametri tarati con un approccio di tipo *grid search cross validation*. I limiti del Decision Tree sono il rischio di overfitting e, poichè la procedura risulta essere ricorsiva, se la scelta della prima feature è sbagliata, allora l'albero generato non rappresenterà assolutamente il sistema da modellare. Per ovviare a questo problema, oltre al Decision Tree, è stato utilizzato anche un regressore di tipo **Random Forest**.

3.1.3 Random Forest

Con l'algoritmo **Random Forest**, i problemi prima descritti nel Decision Tree possono essere mitigati attraverso l'utilizzo di più strutture ad albero che effettuano regressioni. Inoltre, mentre nel Decision Tree la scelta dell'uscita è legata a tutte le features che vengono completamente utilizzate nella costruzione dell'albero, nel Random Forest viene scelto, per ciascun albero, un subset di features su cui decidere. In questo modo, il valore ottenuto da ciascun singolo albero viene confrontato con le uscite degli altri alberi. Come prevedibile, essendo l'uscita continua, il valore dell'uscita del Random Forest è **la media delle uscite dei singoli alberi.**
Oltre al numero di features che viene utilizzato da ciascun albero per effettuare la regressione, bisogna stabilire anche il numero di alberi del Random Forest. Il primo valore è stato settato di default alla **radice quadrata del numero di feature considerate**, il secondo, invece, è stato opportunamente scelto attraverso una *grid search cross validation*.

3.1.4 Considerazioni sulla scelta degli iperparametri dei regressori

La scelta degli iperparametri ottimi è fondamentale affinché l'apprendimento sia fatto nel modo corretto e non si sfoci nel problema dell'overfitting.
Dunque, a tale scopo, è stata utilizzata una procedura di ricerca che effettua un prodotto cartesiano tra tutte le possibili combinazioni definite per ogni iperparametro. Ad esempio, per l'algoritmo

Decision Tree, gli iperparametri considerati sono il numero minimo di campioni che deve contenere un nodo *min_ semple_ split* e la massima profondità dell'albero, *maxt_ depth*. La grid search cross validation andrà a fare il prodotto cartesiano tra i due vettori rappresentanti i possibili valori assunti da questi parametri, al fine di trovare la combinazione ottima. La funzione utilizzata, messa a disposizione da *sklearn*, effettua questa ricerca effettuando una K-fold validation, al fine di testare e trovare la coppia ottima di questi iperparametri facendo variare la porzione di dataset sul quale si effettua la ricerca.
Questo ragionamento è stato ripetuto per tutti gli algoritmi di regressione (così come per quelli di classificazione) implementati. Inoltre, c'è da dire che il train-set ed il test-set sul quale sono stati lanciati questi algoritmi di regressione/classificazione è stato fatto variare usando a monte una *Stratified 4 Fold Validation*.
Una nota da fare riguarda, appunto, l'utilizzo della Stratified K Fold Validation, essa infatti, per essere applicata anche al problema di regressione, ha bisogno di un'operazione di digitalizzazione dell'uscita, attraverso una suddivisione di questa in più range discreti. Si è preferito, inoltre, utilizzare l'operazione di tipo *stratified* piuttosto che una *normale* K-Fold Validation perché tramite essa ogni set generato è rappresentativo di tutto il dataset.
Dunque, ricapitolando:

1. sul dataset viene effettuata una **Stratified 4-Fold Validation**;
2. per ciascuno dei quattro train-test generati si lanciano gli algoritmi di classificazione/regressione;
3. ciascun algoritmo di classificazione/regressione effettua a sua volta una **Grid Search Cross Validation** per la ricerca degli iperparametri ottimi;
4. alla fine, i risultati ottenuti in termini di score sui quattro dataset sono mediati e salvati su file.

3.2 Classificazione

Per il task di Classificazione è stata classificata la QoS percepita in funzione delle features selezionate o estratte.
Le possibili classi di QoS percepite sono le seguenti:

- Low : Downlink $<$ 5 Mbps;
- Medium : 5 Mbps $<$ Downlink $<$ 15 Mbps;
- High : 15 Mbps $<$ Downlink $<$ 30 Mbps;
- Very High : Downlink $>$ 30 Mbps.

È stato quindi necessario discretizzare l'uscita del dataset (res_dl_kbps) per ottenere un problema di classificazione equivalente a partire dal problema di regressione.
Fatto ciò, è stato possibile procedere come fatto per la regressione. Dunque sono stati utilizzati come classificatori:

- MLP: basato sul *deep learning*;

- Decision Tree: basato sul *machine learning*;
- Random Forest: basato sul *machine learning*.

Dunque, si è effettuata la ricerca degli iperparametri ottimi, sia nel caso degli algoritmi di machine learning, sia nel caso della rete neurale. Bisogna inoltre notare che, a differenza del caso della regressione, per la classificazione si è scelto di utilizzare anche un algoritmo di deep learning. Dunque, in questo caso va tenuto in conto il fatto che gli algoritmi di deep learning, oltre la relazione di ingresso-uscita, apprendono dai dati anche il set ottimo e astratto di features da utilizzare.
Per brevità, si evita di spiegare gli algoritmi di machine learning utilizzati per la classificazione, poiché essi sono analoghi a quelli usati per la regressione.

3.2.1 MultiLayer Perceptron – MLP

L'algoritmo MLP genera una rete neurale che consiste di 3 layer ed effettua un apprendimento su 200 epoche. In particolare, è stata utilizzata una funzione di *early stopping* per prevenire l'overfitting. Anche in questo caso, è stata utilizzata una ricerca degli iperparametri ottimi, e, in particolare, di:

- numero di neuroni per ogni layer;
- funzione di attivazione dei layer interni.

Tale ricerca è stata effettuata utilizzando sempre una *Grid Search Cross Validation*.

3.2.2 Considerazioni sulla scelta degli iperparametri dei classificatori

In questo caso, la tipologia di iperparametri utilizzata per gli algoritmi di classificazione (machine learning) è la stessa utilizzata nel caso di quelli usati negli algoritmi di regressione. Ovviamente, è stata ripetuta la fase di addestramento e ricerca degli iperparametri ottimi e, nel caso del Decision Tree Classifier e nel caso del Random Forest Classifier, la ricerca è stata effettuata attraverso una *Grid Search Cross Validation*, che, come detto, opera ricercando i parametri ottimi utilizzando una Cross Validation. Per quanto riguarda, invece, la rete neurale realizzata, la Grid Search Cross Validation utilizza 5 fold per identificare gli iperparametri ottimi (numero di neuroni degli hidden layer, e funzione di attivazione).

4 Valutazione delle prestazioni

Gli algoritmi di regressione e classificazione sono stati lanciati fissando il numero di features da utilizzare, se eseguita la fase di feature selection, e, analogamente, fissando il numero di componenti principali da estrarre, se eseguita la fase di feature extraction. In particolare, dagli esperimenti effettuati in fase di regressione, risulta che i parametri ottimi sono:

- numero K di features ottimo: 3;
- numero di componenti principali ottimo: 11.

Sia nei casi di task di regressione, sia nei casi di task di classificazione con machine learning e sia, infine, nel caso di task di classificazione con deep learning, il dataset è stato splittato utilizzando una Stratified K-Fold Validation con K=4. La differenza è che nel caso di algoritmi di machine learning, a monte è stata fatta una fase di feature extraction/selection, mentre nel caso di MLP solo una fase iniziale di scaling.

4.1 Problema di regressione

Come detto, a partire dal dataset **QoS_railway_paths_nodeid_iccid_feature_extraction**, sono state fatte analisi in tre direzioni:

- utilizzando il dataset per intero;
- utilizzando una porzione del dataset specifica che contempla i campioni relativi ad una stessa fascia oraria;
- utilizzando una porzione del dataset specifica che contempla i campioni relativi ad una specifica rotta.

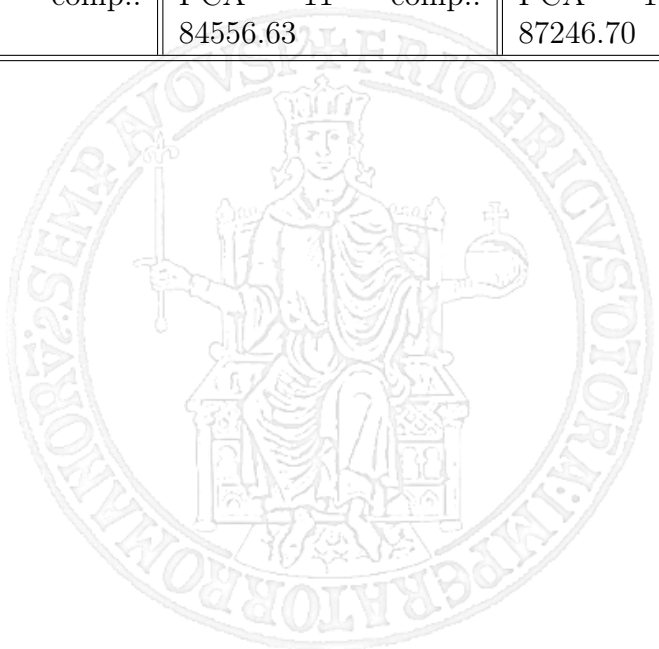
In Tabella 1 sono mostrati i risultati in termini di MSE ottenuti nei vari esperimenti, impostando il numero ottimo di componenti principali, oppure il numero ottimo di features da selezionare.
Tuttavia, è bene sottolineare che, prima di decidere di utilizzare K=3 e Componenti Principali = 11, sono state effettuate varie prove, facendo variare:

- **nel caso di dimensionality reduction – PCA**: il numero di componenti principali selezionate e poi utilizzate come dataset (su cui poi, come detto, è stato applicato lo splitting in train-set e test-set attraverso la Stratified 4-Fold Validation);
- **nel caso di dimensionality reduction – SelectKBest**: il numero K di features utilizzate come dataset (su cui poi, come detto, è stato applicato lo splitting in train-set e test-set attraverso la Stratified 4-Fold Validation).

I risultati ottenuti a valle delle varie prove sono stati salvati in formato *json* e resi disponibili insieme al codice sviluppato.

Tabella 1: Risultati regressione.

Risultati ottenuti utilizzando tutto il dataset		
K-Nearest Neighbors MSE	**Decision Tree MSE**	**Random Forest MSE**
KBest K=3: 51072.66	KBest K=3: 52052.58	KBest k=3: 51026.32
PCA 11 comp.: 113910.24	PCA 11 comp.: 58688.47	PCA 11 comp.: 54311.88
Splitting TEMPORALE – Day		
KBest K=3: 49571.02	KBest K=3: 52943.67	KBest k=3: 51370.19
PCA 11 comp.: 106211.24	PCA 11 comp.: 72583.68	PCA 11 comp.: 68210.83
Splitting TEMPORALE – Night		
KBest K=3: 49465.59	KBest K=3: 53531.22	KBest K=3: 49065.55
PCA 11 comp.: 125060.80	PCA 11 comp.: 82606.89	PCA 11 comp.: 61429.30
Splitting SPAZIALE – Oslo - Bergen		
KBest K=3: 4338.67	KBest K=3: 186.96	KBest k=3: 5790.01
PCA 11 comp.:107737.07	PCA 11 comp.: 87368.63	PCA 11 comp.: 81703.20
Splitting SPAZIALE – Oslo - Trondheim		
KBest K=3: 57160.53	KBest K=3: 50947.59	KBest k=3: 53687.14
PCA 11 comp.: 103481.01	PCA 11 comp.: 84556.63	PCA 11 comp.: 87246.70



Sebbene in tabella siano stati mostrati i risultati solo quando sono state utilizzate 11 componenti principali per la feature extraction e 3 features ottime per la feature selection, i test sono stati, come detto in precedenza, effettuati facendo variare questi parametri. In particolare, questi valori sono stati scelti in relazione al fatto che, arrivati ad un certo numero di componenti principali nel caso della *PCA*, e considerato un certo numero di features ottime nel caso della *SelectKBest*, le prestazioni in termini di MSE iniziano ad assestarsi intorno ad un certo valore.
È stato notato che, affinché i risultati dei regressori utilizzando a monte *PCA* raggiungano quelli ottenuti utilizzando a monte la procedura di *SelectKBest* siano simili, è necessario prendere un numero di componenti principali almeno pari a 11.
Questo probabilmente è legato al fatto che la PCA non è la migliore soluzione per effettuare *dimensionality reduction* su questo dataset. Infatti, la PCA presuppone relazioni lineari tra tutte le fratures, essendo basata sul coefficiente di correlazione di Pearson. Tuttavia, come è evidenziato dalla scatter matrix mostrata in precedenza, non tutte le relazioni tra variabili sono lineari.
Inoltre, sebbene fossero presenti 10 tratte durante la fase di splitting spaziale, ne sono state considerate solo 2 perché risultano le uniche con numero di campioni non inferiore a 200. Sotto tale soglia la regressione sarebbe risultata non efficiente.

4.2 Problema di classificazione equivalente

Come per la regressione, anche per i problemi di classificazione equivalente sono stati effettuati test analoghi. Tuttavia, a causa della complessità computazionale dei test, della potenza di calcolo necessaria e del tempo ridotto a disposizione, non è stato possibile ripetere tutti i test effettuati nel caso della regressione. Per la classificazione, dunque, gli algoritmi sono stati eseguiti solamente:

- sul dataset non segmentato;
- sul dataset segmentato temporalmente, in particolare solo sul dataset night.csv;
- su una sola rotta (*Oslo - Trondheim.* route_id, 12).

Da notare che, sebbene i test siano stati eseguiti solo su questi dataset, il codice per gli altri test è commentato, quindi è possibile eseguirlo qualora lo si desideri.
Inoltre, per gli algoritmi di machine learning utilizzati[2] per la classificazione, a differenza del caso della regressione in cui si faceva variare:

- il numero di componenti principali, nel caso della procedura di feature extraction;
- il numero di features ottime, nel caso della procedura di feature selection.

ora tali parametri sono stati fissati, in relazione alla migliore scelta ricavata dall'analisi dei risultati nel problema di regressione. In particolare, i valori ottimi ritrovati sono, $\mathbf{K} = 3$ per la feature selection e **componenti principali** $= 11$ nel caso della feature extraction.
I risultati ottenuti sono mostrati di seguito nella Tabella 2.

[2]DecisionTree Classifier & RandomForest Classifier

Tabella 2: Risultati classificazione.

Risultati ottenuti utilizzando tutto il dataset		
MLP accuracy	**Decision Tree accuracy**	**Random Forest accuracy**
0.57	KBest K=3: 0.40	KBest K=3: 0.42
	PCA 11 comp.: 0.53	PCA 11 comp.: 0.57
Segmentazione TEMPORALE - Notte		
0.591	KBest K=3:0.437	KBest K=3:0.465
	PCA 11 comp.:0.489	PCA 11 comp.:0.558
Segmaìentazione SPAZIALE - Oslo - Trondheim		
0.52	KBest K=3: 0.42	KBest K=3: 0.42
	PCA 11 comp.: 0.45	PCA 11 comp.: 0.49

Di seguito sono mostrate alcune delle matrici di confusione ottenute. Come si può notare, in caso di dataset completo, la matrice di confusione ha un aspetto a banda (sulla diagonale principale), mentre negli altri casi l'esito dipende da quanti campioni la segmentazione ha rimosso.

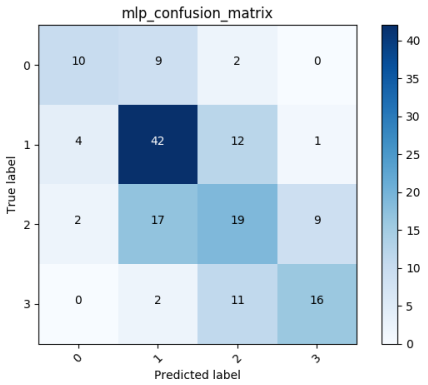


Figura 3: Matrice di confusione usando MLP sul dataset segmentato per rotta 12

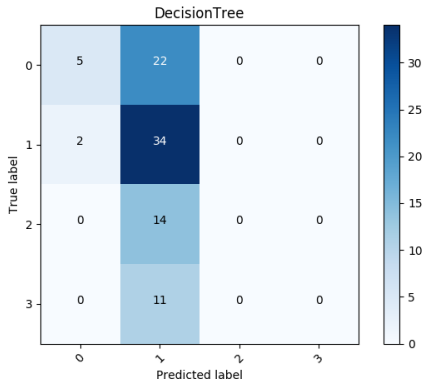


Figura 5: Matrice di confusione usando decision tree e 3-best feature sul dataset segmentato per rotta 12

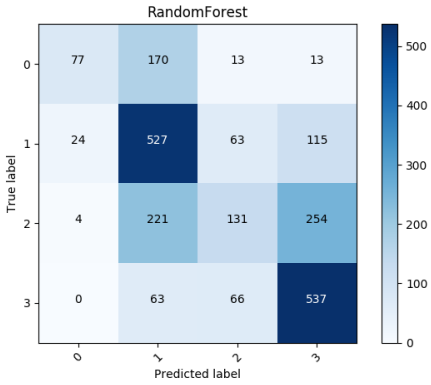


Figura 4: Matrice di confusione usando Random Forest e PCA con 11 componenti

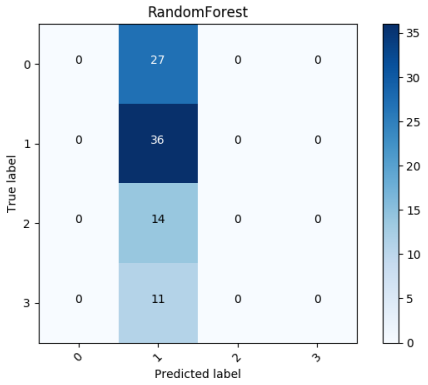


Figura 6: Matrice di confusione usando random forest e 3-best feature, sul dataset segmentato per rotta 12

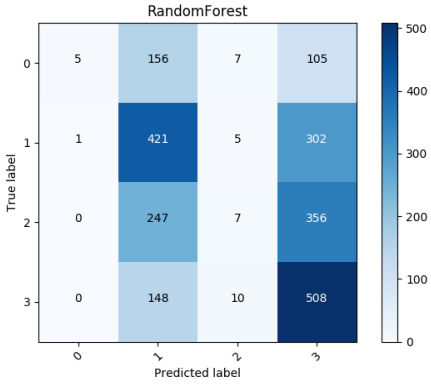


Figura 7: Matrice di confusione usando random forest con 3-Best features sul dataset non segmentato

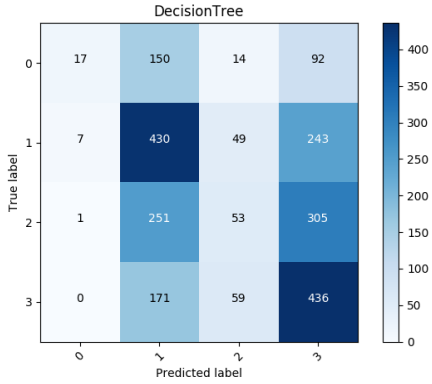


Figura 8: Matrice di confusione usando Decision Tree con 3-Best features sul dataset non segmentato

Dunque, come possiamo notare dai risultati mostrati in precedenza, l'algoritmo che si comporta meglio è quello di deep learning, che realizza una rete neurale a 3 livelli nascosti. Tuttavia, il prezzo da pagare nell'utilizzo di un MLP, ancora non troppo complesso, sta nella durata dei tempi

di esecuzione, nettamente superiori rispetto ai due algoritmi di machine learning implementati. Si può inoltre notare che le prestazioni di Decision Tree e Random Forest sono circa le stesse.

5 Conclusioni

In questo lavoro, dunque, sono stati affrontati i problemi di regressione e classificazione equivalente, utilizzando differenti porzioni del dataset e differenti features. Sono state seguite, inoltre, differenti strade e confrontati i diversi risultati a valle dei vari esperimenti condotti.
Come visto dagli esperimenti, la maggior parte delle volte i risultati migliori si ottengono utilizzando la feature selection, anche se il valore di MSE non supera mai una certa soglia di bontà (discorsi analoghi possono essere ripetuti per l'accuratezza).
Un lavoro futuro che potrebbe essere effettuato, riguarda l'utilizzo di un'altra procedura di feature extraction (ad esempio la Kernel PCA), dato che la PCA presuppone relazioni di linearità tra le features.
Anche la feature selection potrebbe essere soggetta a miglioramento, in quanto la SelectKBest seleziona le K features aventi score più alto. Tale score è calcolato attraverso un *F-test* il quale non cattura relazioni non lineari tra le features.
Possibili miglioramenti, inoltre, potrebbero essere raggiunti utilizzando differenti tipologie di algoritmi di deep learning, oppure variando il numero di layer e complicando così la rete neurale, pagando però in termini di tempi di addestramento e testing dell'algoritmo.
Infine, si è visto che in genere l'utilizzo di segmentazione, e spaziale, e temporale, migliora i risultati. Tuttavia, il problema principale è che, essendo in partenza il dataset dal quale si vanno ad estrarre campioni, relativi ad una stessa rotta o relativi ad una stessa fascia oraria, composto di solo circa novemila campioni, i dataset estratti a valle di segmentazione risultano talvolta costituiti da troppi pochi campioni. Dunque l'apprendimento non è ottimale.

6 Appendice

In questa sezione sarà presentato il codice sviluppato in python per risolvere i problemi di regressione/classificazione [1] in maggiore dettaglio. In particolare, sono state utilizzate le librerie [2, 3] per sviluppare i vari algoritmi presentati.
In tale sezione saranno presentate solo alcune delle funzioni realizzate per ottenere i risultati su mostrati. Il codice sviluppato è presente integralmente nella repository git `https://www.github.com/MarioVar/API`

6.1 Funzioni utilizzate per lo splitting del dataset (Training/Test)

```
def save_stratified_r2(filename,knn,dt,rf):
  #salvataggio parametri di tuning
  dic={}
  dic['knn_r2']=knn
  dic['dt_r2']=dt
  dic['rf_r2']=rf
  with open(filename+".json","a+") as file:
    file.write("mean_squared_error " + json.dumps(dic) + "\n")

#continous = True -> y da digitalizzare in 10 bins
#continous = False -> y gia' discreta
def stratifiedKFold_validation(X , Y,continous=True):
  if continous==True:
    bins = np.linspace(0,30000, 1000)
    Y = np.digitize(Y , bins)
  if isinstance(X,(np.ndarray)):
    X = pd.DataFrame(X=np.int(X[1:,1:]),    # values
    index=X[1:,0],    # 1st column as index
    columns=X[0,1:])  # 1st row as the column names
  if isinstance(Y,(np.ndarray)):
    Y = pd.Series(Y)
  folds = StratifiedKFold(n_splits=4 , random_state=42 , shuffle= True)
  ln_scores = []
  knn_scores = []
  dt_scores = []
  rf_scores = []
  for train_index , test_index in folds.split(X , Y):
    X_train , X_test = X.loc[train_index] , X.loc[test_index]
    Y_train , Y_test = Y.loc[train_index] , Y.loc[test_index]
    if continous==True:
      knn_dict , dt_dict, rf_dict = rg.start_regression_tun(X_train , X_test
    , Y_train , Y_test)
      knn_scores.append(knn_dict['r2'])
      dt_scores.append(dt_dict['r2'])
      rf_scores.append(rf_dict['r2'])
    elif continous==False:
```

```
36        dt_dict, rf_dict = rg.start_classification_tun(X_train , X_test ,
      Y_train , Y_test)
37        dt_scores.append(dt_dict['accuracy'])
38        rf_scores.append(rf_dict['accuracy'])
39
40
41
42    if continous== True:
43      print("mean_squared_error StratifiedKFold Validation KNN Regression: ",
      np.mean(knn_scores))
44      print("mean_squared_error StratifiedKFold Validation DT Regression: ",np
      .mean(dt_scores))
45      print("mean_squared_error StratifiedKFold Validation RF Regression: ",np
      .mean(rf_scores))
46      return np.mean(knn_scores), np.mean(dt_scores) , np.mean(rf_scores)
47    else:
48      #print("Accuracy StratifiedKFold Validation MLP Classification: " ,np.
      mean(mlp_scores))
49      print("Accuracy StratifiedKFold Validation DT Classificatio: ",np.mean(
      dt_scores))
50      print("Accuracy StratifiedKFold Validation RF Classification: ",np.mean(
      rf_scores))
51      return np.mean(dt_scores) , np.mean(rf_scores)
```

Codice Componente 1: Funzione per la Stratified K Fold Validation

6.2 Funzioni utilizzate per il preprocessing

La funzione utilizzata per la feature selection è mostrata di seguito. Tale funzione, in realtà, prima di effettuare la selezione delle K best features, effettua delle operazioni di preprocessing, andando a:

1. rimuovere colonne di tipo **object**;
2. trattare i valori nulli, tramite una funzione di imputing;
3. effettuare uno scaling sui dati [3]

Inoltre, al fine di rendere generalizzabile il codice e riusabile per la classificazione, è possibile selezionare la funzione da usare per la selezione delle features.

```
1 #Funzione utilizzata per la feature selection
2 def get_main_features(csv_file,feature_to_remove,y_label,i,median_imputing=
      True,function=f_regression):
3   print("Inizio feature selection")
4       #lettura csv
```

[3] Sebbene in figura sia mostrato uno scaling min max, è possibile, modificando il codice, chiamare le altre funzioni di scaling implementate (robust scaling e standard scaling). I risultati presentati in precedenza sono stati ottenuti utilizzando un robust scaling, ma, come detto, c'è poca differenza in termini di prestazioni.

```
5   data = pd.read_csv(csv_file)
6   #se il csv e' quello con le feature di mobilita'
7   if len(data.columns) == 17:
8     data['duration'] = data['res_time_end_s'].sub(data['res_time_start_s'],
      axis=0)
9
10  #rimozione colonne che non sono numeriche
11  newdf = data.select_dtypes(exclude='object')
12
13  main_feature = newdf.columns
14  main_feature = main_feature.drop(feature_to_remove)
15
16
17  #selezione features principali
18  subdataframe=newdf.loc[:,main_feature]
19  y=newdf[y_label]
20
21
22  if median_imputing==True:
23  #imputing dei missing values con media
24    imputer_mean = SimpleImputer()
25    imputed_mean_filled = imputer_mean.fit_transform(subdataframe)
26
27    subdataframe_mean_filled = pd.DataFrame(imputed_mean_filled, columns =
     main_feature)
28    #scaling delle features principali
29    scaled_subdataframe_mean_filled = scaling_dataframe_minmax(
     subdataframe_mean_filled)
30
31    scaled_subdataframe_mean_filled = pd.DataFrame(
     scaled_subdataframe_mean_filled, columns = main_feature)
32
33    #select_k_best per la media
34    selector_mean = SelectKBest(function, k=i)
35    selector_mean.fit(scaled_subdataframe_mean_filled, y)
36
37    k_best_features_mean_filled = selector_mean.transform(
     scaled_subdataframe_mean_filled)
38    selected_columns_mean = selector_mean.get_support(indices=True)
39
40    k_best_features = pd.DataFrame(k_best_features_mean_filled,
41      columns = scaled_subdataframe_mean_filled.columns[
     selected_columns_mean])
42
43  else:
44  #imputing dei missing values con moda
45    imputer_mode = SimpleImputer(strategy = 'most_frequent')
46    imputed_mode_filled = imputer_mode.fit_transform(subdataframe)
47
```

```
    subdataframe_mode_filled = pd.DataFrame(imputed_mode_filled, columns =
    main_feature)


    #scaling delle features principali
    scaled_subdataframe_mode_filled = scaling_dataframe_minmax(
    subdataframe_mode_filled)


    scaled_subdataframe_mode_filled = pd.DataFrame(
    scaled_subdataframe_mode_filled, columns = main_feature)


    #select_k_best per la moda
    selector_mode = SelectKBest(function, k=i)
    selector_mode.fit(scaled_subdataframe_mode_filled, y)

    k_best_features_mode_filled = selector_mode.transform(
    scaled_subdataframe_mode_filled)
    selected_columns_mode = selector_mode.get_support(indices=True)


    k_best_features = pd.DataFrame(k_best_features_mode_filled,
      columns = scaled_subdataframe_mode_filled.columns[
    selected_columns_mode])


  return k_best_features,y,k_best_features.columns
```

Codice Componente 2: Funzioni per feature selection

La funzione utilizzata per la dimensionality reduction è di seguito mostrata. Come mostrato dalla firma della funzione, viene passato in ingresso il parametro **n_comp** che indica il numero di componenti principali da prelevare, effettuata la trasformazione di spazio.

```
#Funzione utilizzata per la feature Extraction: PCA
def pca_preproc(Dataframe,n_comp,show=False):
  print("Inizio feature extraction")
  #scoperta numero di comp princ da usare
  principal_dataframe=pca.fit_transform(Dataframe)
  plt.plot(np.cumsum(pca.explained_variance_ratio_))
  plt.xlabel('number of components')
  plt.ylabel('cumulative explained variance')
  plt.grid()
  if show==True:
    plt.show()
  plt.savefig(str(n_comp)+'_PCA_components_variance.png')
  #Scoperto che il numero di componenti principali da usare e' 2
  principalDF=pd.DataFrame(data=principal_dataframe)

```

```
16
17    return principalDF
```

Codice Componente 3: Funzioni per feature extraction

6.3 Funzioni utilizzate per la regressione

Di seguito si riportano le funzioni utilizzate per la regressione. La prima funzione effettua la taratura degli iperparametri, la seconda, invece, avvia il processo di regressione. Se il parametro *stratified* della funzione *regression* è *True*, allora la funzione *start_regression_tun* sarà eseguita durante l'esecuzione della funzione *stratifiedKFold_validation*.

```
1  def start_regression_tun(X_train, X_test, y_train, y_test):
2
3    knn_dict = {}
4    #K-nearest-neighbor
5    k_opt_array=np.linspace(1,150,150,dtype=int)
6    dist_vect=np.linspace(1,10,10,dtype=int)
7    r2_knn,dist,K=KNN_tun(X_train,X_test,y_train,y_test,k_opt_array,dist_vect)
8    knn_dict.update({'r2' : float(r2_knn)})
9    knn_dict.update({'metrics' : int(dist)})
10   knn_dict.update({'k' : int(K)})
11   #print("R2_KNN: ",r2_knn,"distance_opt: ",dist,"K_opt: ",K)
12
13   r2dt=0;
14   depth=0;
15   samples_min=0;
16
17   #decision tree
18   dt_dict = {}
19   max_depth_array=np.linspace(1,200,20,dtype=int)
20   minSamples_split=np.linspace(2,300,30,dtype=int)
21   r2dt,depth,samples_min=Tuning_DecisionTree(max_depth_array ,
       minSamples_split, X_train , X_test , y_train , y_test)
22   dt_dict.update({'r2' : float(r2dt)})
23   dt_dict.update({'depth' : int(depth)})
24   dt_dict.update({'samples' : int(samples_min)})
25   #print("R2_DT: ",r2dt,"depth_opt: ",depth,"samples_min: ",samples_min)
26
27
28   #random forest
29   rf_dict = {}
30   num_trees_vect=np.linspace(1,200,dtype=int)
31   r2rf,ntrees=random_forest_tun(num_trees_vect,depth,samples_min,X_train ,
       X_test , y_train , y_test)
32   rf_dict.update({'r2' : float(r2rf)})
33   rf_dict.update({'trees' : int(ntrees)})
34   #print("r2_rf: ",r2rf,"num estimators opt RF: ",ntrees)
35   return knn_dict , dt_dict, rf_dict
```

```
36
37 def regression(X,Y,stratified=True,scale=False):
38
39   r2_knn={}
40   r2_dt={}
41   r2_rf={}
42   if stratified==True:
43     r2_knn['knn_mse'] , r2_dt['dt_mse'], r2_rf['rf_mse']=sp.
     stratifiedKFold_validation(X , Y)
44   else:
45     X_train , X_test , Y_train , Y_test=sp.dataset_split(X,Y,scale)
46     knn_dict , dt_dict, rf_dict=start_regression_tun(X_train , X_test ,
     Y_train , Y_test)
47     r2_knn['knn_mse']=knn_dict['r2']
48     r2_dt['dt_mse']=dt_dict['r2']
49     r2_rf['rf_mse']=rf_dict['r2']
50
51 return r2_knn , r2_dt, r2_rf
```

Codice Componente 4: Funzioni per la regressione

6.4 Funzioni utilizzate per la classificazione

Di seguito si riportano le funzioni utilizzate per la classificazione: la prima effettua il taratura degli iperparametri e la seconda avvia il processo di classificazione. Se l'opzione *stratified* della funzione *classification()* è *True*, allora la funzione *start_classification_tun()* sarà richiamata nella funzione stratifiedKFold_validation()

```
1 def start_classification_tun(X_train, X_test, y_train, y_test):
2
3   r2dt=0;
4   depth=0;
5   samples_min=0;
6
7   #decision tree
8   dt_dict = {}
9   max_depth_array=np.linspace(1,200,20,dtype=int)
10  minSamples_split=np.linspace(2,300,30,dtype=int)
11  acc,depth,samples_min=Tuning_DecisionTree(max_depth_array ,
     minSamples_split, X_train , X_test , y_train , y_test, classification=
     True)
12  dt_dict.update({'accuracy' : str(acc)})
13  dt_dict.update({'depth' : int(depth)})
14  dt_dict.update({'samples' : int(samples_min)})
15  #print("R2_DT: ",r2dt,"depth_opt: ",depth,"samples_min: ",samples_min)
16
17
18  #random forest
19  rf_dict = {}
```

```
  num_trees_vect=np.linspace(1,200,dtype=int)
  accrf,ntrees=random_forest_tun(num_trees_vect,depth,samples_min,X_train ,
   X_test , y_train , y_test, classifier=True)
  rf_dict.update({'accuracy' : str(accrf)})
  rf_dict.update({'trees' : int(ntrees)})
  #print("r2_rf: ",r2rf,"num estimators opt RF: ",ntrees)

  return dt_dict, rf_dict

def classification(X,Y,stratified=True,scale=False):
  accuracy_dt={}
  accuracy_rf={}
  if stratified==True:
    accuracy_dt['dt_accuracy'], accuracy_rf['rf_accuracy']=sp.
   stratifiedKFold_validation(X , Y, continous = False)
  else:
    X_train , X_test , Y_train , Y_test=sp.dataset_split(X,Y,scale)
    dt_dict, rf_dict =start_classification_tun(X_train , X_test , Y_train ,
   Y_test)
    accuracy_dt['dt_accuracy']=dt_dict['accuracy']
    accuracy_rf['rf_accuracy']=rf_dict['accuracy']
  return  accuracy_dt, accuracy_rf
```

Codice Componente 5: Funzioni per la classificazione

Come detto, nel caso di rete MLP non è necssario utilizzare a monte la selezione delle feature, in quanto la rete apprende, da set di feature, autonomamente quali utilizzare per la classificazione. DI seguito è mostrata la funzione che effettua una stratified e chiama la MLP per ogni iterazione.

```
def Classification_withMLP(name_dataset='Def_',csv_path="../
   QoS_RAILWAY_PATHS_REGRESSION/
   QoS_railway_paths_nodeid_iccid_feature_extraction.csv"):

  feature_to_remove= ['res_dl_kbps' , 'ts_start' , 'ts_end']
  y_label='res_dl_kbps'

  feature_vect,dataframe,y=pr.get_feature(csv_path,feature_to_remove,y_label
   )
  y = mc.CreateClassificationProblem(y,plot=False)

  dataframe=pr.robust_scalint(dataframe)
  dict_mlp=sp.stratifiedKFold_MLP(dataframe , y)
  filename = str(name_dataset)+"_MLP_Classification"
  with open(filename+".json","w") as file:
    file.write(json.dumps(dict_mlp))
```

Codice Componente 6: Funzioni per la classificazione

Di seguito è mostrato il codice della funzione che effettua la ricerca degli iperparametri ottimi dell'MLP, in particolare spaziando in termini di numero di neuroni della rete fissata e funzione di attivazione degli hidden layers.

```
def mlp_tuning(X_train , X_test , Y_train , Y_test):
    model = wp.KerasClassifier(build_fn = multi_layer_perceptron, input_dim
    = X_train.shape[1])

    grid_params = {'activation': ['relu' , 'sigmoid', 'tanh'] , 'num_nodes'
    : np.arange(10 , 120 , 10)}
    cv = GridSearchCV(estimator = model , param_grid = grid_params , n_jobs
    =-1 , verbose = 2 , cv = 5)
    grid_result = cv.fit(X_train , Y_train)
    print("Best: %f using %s" , (grid_result.best_params_))
    numnodes=grid_result.best_params_['num_nodes']
    activation=grid_result.best_params_['activation']
    soft_values, predictions, training_soft_values, training_predictions,
    accuracy, fmeasure, macro_gmean, training_accuracy, training_fmeasure,
    training_macro_gmean=train_test(X_train , Y_train , X_test , Y_test ,
    num_nodes=numnodes, activation=activation)

    print(accuracy)
    return accuracy
```

Codice Componente 7: Funzione ricerca gli iperparametri ottimi della rete MLP

6.5 Funzioni utilizzate per la segmentazione temporale

La funzione che effettua la segmentazione temporale è chiamata *temporal_splitting()* tale funzione definisce le fasce orarie e chiama su ogni fascia la funzione *get_dataset_splitted_by_time()* che genererà i relativi csv: day.csv & night.csv

```
def get_dataset_splittedby_time(range, time):
        data =  pd.read_csv("../QoS_RAILWAY_PATHS_REGRESSION/
    QoS_railway_paths_nodeid_iccid_feature_extraction.csv")
        range_data = data[data['hour_of_day'].isin(range)]
        filename = time + ".csv"
        fullname = os.path.join('..'+'/QoS_RAILWAY_PATHS_REGRESSION/',
    filename)
        range_data.to_csv(fullname)
        return range_data


def temporal_splitting():
        day = get_dataset_splittedby_time(np.linspace(0 , 12 , 1 , dtype =
    int) , "day")
        night = get_dataset_splittedby_time(np.linspace(12 , 23 , 1, dtype=
    int), "night")
        return  day , night
```

Codice Componente 8: Funzioni per la segmentazione temporale

6.6 Funzioni utilizzate per la segmentazione spaziale

Le funzioni che effettuano la segmentazione spaziale (per rotta) sono di seguito illustrate. Esse produrranno tanti dataset quante sono le rotte. In particolare si lavora a partire dal dataset contenente le info sulle rotte, si estraggono per ogni rotta le triple (*timestamt_start,timestamp_end,nodeid*) ed attraverso questa tripla si prelevano i campioni dal dataset di interesse relativi ad una specifica rotta. Tale procedimento è iterato per ogni rotta.

```
def spatial_splitting():
  path_csv='../QoS_RAILWAY_PATHS_REGRESSION/
    QoS_railway_paths_latlong_nsb_gps_segment_mapping_mobility_apu2.csv'
  routes = {}
  routes = get_all_routes(path_csv)
  print(routes)
  j=1
  dataframe_divided_by_routedesc = {}
  dataframe_divided_by_routeid = {}
  for i in routes:
    dataframe_divided_by_routedesc.update({routes[i] : get_nodeid_by_route(i
    )})
    dataframe_divided_by_routeid.update({i : get_nodeid_by_route(i)})
    filename = routes[i] + ".csv"
    print(filename)
    fullname = os.path.join('../QoS_RAILWAY_PATHS_REGRESSION'+'/', filename)
    dataframe_divided_by_routeid[i].to_csv(fullname)
  return dataframe_divided_by_routedesc , dataframe_divided_by_routeid,
    routes

def get_all_routes(path):
  data = pd.read_csv(path)
  new_data =list(set(data['route_id']))
  dict = {}
  for index, i in data.iterrows():
    if i['route_id']  not in dict:
      print(i['route_id'])
      dict.update({ i['route_id'] : i['route_desc']})
  return dict

def get_nodeid_by_route(route_id, pca = False):
  path_csv="../QoS_RAILWAY_PATHS_REGRESSION/
    QoS_railway_paths_latlong_nsb_gps_segment_mapping_mobility_apu2.csv"
  dataapu = pd.read_csv(path_csv)
  data2_single_route= dataapu[dataapu['route_id']==route_id]
  data = pd.read_csv("../QoS_RAILWAY_PATHS_REGRESSION/
    QoS_railway_paths_nodeid_iccid_feature_extraction.csv")
  if pca == True:
    dataset_routeid = pd.merge(data2_single_route[['nodeid','
    res_time_start_s','res_time_end_s']], data,left_on=['nodeid','
    res_time_start_s','res_time_end_s'],right_on=['nodeid','ts_start','
```

```
    ts_end'],how='inner')[data.columns]
35  else:
36    dataset_routeid = pd.merge(data2_single_route[['nodeid','
    res_time_start_s','res_time_end_s']], data,left_on=['nodeid','
    res_time_start_s','res_time_end_s'],right_on=['nodeid','ts_start','
    ts_end'],how='inner')
37 return dataset_routeid
```

Codice Componente 9: Funzioni per la segmentazione spaziale

Riferimenti bibliografici

[1] Andreas C Müller, Sarah Guido, et al. *Introduction to machine learning with Python: a guide for data scientists*. " O'Reilly Media, Inc.", 2016.

[2] Sklearn. `https://scikit-learn.org/stable/`, 2019. [Online; accessed 5-July-2019].

[3] Keras. `https://keras.io/`, 2019. [Online; accessed 5-July-2019].