UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria "Enzo Ferrari"

Corso di Perfezionamento a crediti in:

SCHOOL IN AI: DEEP LEARNING, VISION AND LANGUAGE FOR INDUSTRY

DEEP LEARNING PER L'INGEGNERIA DEL SOFTWARE: MLP PER LA JIT DEFECT PREDICTION

AUTORE

Gerardo Festa

Anno Accademico 2021-2022

Indice

SEZIONE 1

Introduzione

Il seguente progetto si pone l'obiettivo di verificare se il Deep Learning possa essere una valida alternativa ai modelli di Shallow Learning nell'ambito del Software Engineering, più nello specifico nella sezione del Just In Time Defect Prediction. In particolare, verrà utilizzato un multi-layer perceptron per cercare di classificare correttamente le istanze che contengono un difetto software all'interno di repository Github, per poi paragonare i risultati con quelli ottenuti dai più canonici metodi di Machine Learning.

1.1 Software Defect Prediction

La Defect Prediction, tema centrale nell'ingegneria del software, consiste nel predire se una unità di lavoro, sia essa un commit o un file presente in un commit, possa portare il software a fault o bug.

La Just In Time (JIT) Defect Prediction, in particolare, si occupa della predizione a livello di commit, cercando di segnalare al programmatore che il suo commit sta introducendo un difetto - o ha buona probabilità di farlo -, proprio mentre questo sta effettuando l'operazione di push alla repository software. Questa caratteristica è ciò che la distingue dalla defect prediction tradizionale, che viene vista sì come una tecnica efficace per individuare difetti a lungo termine su progetti software, ma ha un'applicabilità piuttosto limitata nel concreto, come evidenziato da *Kamei et al.* [1]. Generalmente, un dataset dedito alla JIT Defect Prediction si compone da un set di metriche di processo e di prodotto. Queste, formalizzate da *Kamei*

et al. [1] e *Rahman et al.* [2], tendono a valutare il processo di produzione del software, cercando di analizzarlo da più punti di vista; le metriche, dunque, non fanno riferimento diretto al codice, come fanno quelle spesso utilizzate nella Defect Prediction tradizionale (che includono, ad esempio, le metriche di McCabe o di Chidamber & Kemerer), ma si riferiscono a come gli sviluppatori gestiscono la repository e le modifiche effettuate su essa. Ad esempio, oltre alle metriche più banali come linee di codice aggiunte o rimosse, troviamo metriche quali l'esperienza dello sviluppatore, il numero di sviluppatori che hanno contribuito al commit, il numero di cambiamenti dei file e l'entropia della modifica. Nel 2019 *Pascarella, Palomba e Bacchelli* [3] hanno proposto una variante della JIT Defect Prediction, chiamata Just In Time Fine-Grained Defect Prediction. Questa, invece di indicare solamente quando un commit potrebbe introdurre un difetto, ha l'obiettivo di indicare con precisione quale file (o sottoinsieme di file) coinvolto nel commit è quello difettivo, in un tentativo di migliorare quanto la precisione quanto l'efficacia della previsione. Il vantaggio ulteriore di scendere a questo livello di precisione è quello di consentire di avere un dataset più ampio di osservazioni, anche limitando il numero di progetti analizzati.

1.2 JIT Defect Prediction Dataset

Nel 2022, *Lomio et al.* [4] presentano una versione aggiornata del lavoro del 2019, nel quale vengono testate tecniche di anomaly detection su dataset poco bilanciati come quelli utilizzati negli studi sull'evoluzione del software e sulla presenza di difetti. In questo paper viene presentato e reso disponibile un dataset composto da 32 progetti software open-source, per un totale di 193000 file, di cui il 37% introduce un difetto. L'introduzione del difetto rappresenta quindi la variabile target dello studio, mentre le feature utilizzate per la predizione sono 24, riportate nella Tabella 1.1

Data la natura numerica dei dati e data la dimensione del dataset, questo si presta bene all'applicazione di un modello deep quale il MultiLayer Perceptron. Pertanto, nel seguente studio viene testata l'efficacia dell'utilizzo di un MLP per la predizione dei difetti, con una comparativa finale con i metodi di Machine Learing illustrati nel paper da cui provengono i dati.

Tabella 1.1: Descrizione delle variabili

Metrica	Acronimo	Descrizione
COMM	Commit Count	Numero di cambiamenti per il file fino al commit considerato
ADEV	Active Dev Count	Numero di sviluppatori che hanno modificato il file
DDEV	Distinct Dev Count	Numero di sviluppatori che hanno modificato il file fino al commit considerato
ADD	Normalized Lines Added	Numero di linee aggiunte al file nel commit (normalizzato)
DEL	Normalized Lines Deleted	Numero di linee rimosse al file nel commit (normalizzato
OWN	Owner's Contributed Lines	Booleano che indica se la modifica è stata fatta dal creatore del file
MINOR	Minor Contributor Count	Numero di sviluppatori che hanno modificato meno del 5% del file
SCTR	Changed Code Scattering	Numero di package modificati nel commit
NADEV	Neighbor's Active Dev Count	Numero di sviluppatori che hanno modificato i file coinvolti nel commit in cui il file esaminato è stato modificato
NDDEV	Neighbor's Distinct Dev Count	Numero cumulativo di sviluppatori che hanno modificato file coinvolti nei commit in cui il file in esame è stato modificato
NCOMM	Neighbor's Commit Count	Numero di commit che hanno modificato file contenuti nei commit che hanno modificato il file in esame
NSCTR	Neighbor's Change Scattering	Numero di package coinvolti in commit che modificano il file in esame
OEXP	Owner Experience	Linee modificate dall'autore del file nel progetto
EXP	All Committer's Experience	Media dell'esperienza di tutti i commiter
ND	Number of modified Directories	Numero di cartelle modificate
Entropy	Distribution of modified code across each file	Entropia delle modifiche al file fino al commit considerato
LA	Lines of code Added	Numero di linee di codice aggiunte al file nel commit
LD	Lines of code Deleted	Numero di linee di codice rimosse dal file nel commit
LT	Lines of code in a file before the change	Linee di codice nel file prima della modifica
AGE	Avg interval between last and current change	Intervallo medio tra le modifiche
NUC	Number of Unique Changes	Numero di volte in cui il file è stato modificato da solo
CEXP	Commiter Experience	Numero di commit che contengono il file fatti dal committer fino al commit in esame
REXP	Recent developer experience (last x months)	Numero di commit fatti al file nell'ultimo mese dal committer
SEXP	Developer EXPerience on a Subsystem	Numero di commit fatti dallo sviluppatore nel package che contiene il file

SEZIONE 2

MLP per la JIT Defect Prediction

2.1 Preprocessing dei dati e divisione in train e test set

I dati, già in forma numerica, richiedono pochi ritocchi. Viene effettuato uno scaling dei dati tramite la classe StandardScaler fornita dalla libreria SciKit Learn. Tuttavia, la variabile OWN, insieme alla variabile target, viene estromessa da questo schema, in quanto variabile binaria. Il caricamento dei dati viene effettuato tramite Pandas, mentre c'è da approfondire sulla scelta di train e test set: per testare al meglio l'efficacia del MLP, vengono effettuati tre diversi esperimenti che si differenziano l'uno dall'altro proprio in base all'approccio con il quale vengono scelti l'insieme di addestramento e di testing. In un primo momento, si è verificato il corretto funzionamento della rete con un semplice metodo di campionamento stratificato, ovvero il train_test_split messo a disposizione da Sklearn. Dunque, la distribuzione delle classi nel dataset originale viene rispettata nello split in train e test set. La rete e il suo utilizzo sono presenti nel file FirstNN.ipynb

La seconda tipologia di campionamento e di utilizzo della rete è stata realizzata in 10Fold-CV.ipynb, dove, utilizzando TensorFlow e Keras per semplificare il processo, viene utilizzata la 10 Fold Cross Validation per ottenere una stima quanto più reale possibile dell'errore. Oltre a far questo, vengono testate varie combinazioni di iperparametri (tematica trattata in seguito) nel tentativo di fare tuning sulla rete

Il terzo e ultimo approccio è basato sul sampling effettuato nel paper di origine. La soluzione adoperata, oltre ad essere motivata dalla sperimentazione di un possibile utilizzo reale del modello, risulta anche consistente a livello logico. Si tratta, infatti, di una "estensione" della tecnica del Leave One Out; in questo caso, la rete viene allenata sui dati relativi a 31 dei 32 progetti software, per poi essere testata su quello escluso. Come nell'utilizzo della 10FoldCV, iterativamente la selezione coprirà tutti e 32 i progetti software. Dal punto di vista applicativo, certamente questo sarebbe il metodo di utilizzo. Un utente potrebbe utilizzare la rete già allenata su altri progetti open source per verificare se, facendo un commit nella sua repository software, c'è possibilità di introduzione di un difetto. Dal punto di vista logico, si è certi che nel test set le osservazioni siano effettivamente nuove per la rete, in quanto nel training set si trovano solo osservazioni di altri progetti (non c'è possibilità di predizione sulla base di una "dipendenza" con osservazioni precedenti).

2.2 MultiLayer Perceptron

La rete ha una dimensione di input sempre pari a 24 nodi, ovvero il numero delle features del dataset e un nodo di output in modo da poter determinare la classe di appartenenza. Per quanto riguarda gli hidden layer, ce ne sono due composti da 13 nodi caduno in FirstNN e LeaveOneProjectOut, mentre in 10FoldCV per la natura del tuning, la dimensione è variabile.

2.2.1 FirstNN

Dopo una fase di sperimentazione con la funzione di attivazione della rete, la scelta è ricaduta su tanh, che riporta i migliori risultati in termini di recall (pressappoco uguali sulle altre metriche), seppure si differenzi poco da relu. L'ottimizzazione viene gestita con il metodo ADAM, mentre il criterio della loss è la Binary Cross Entropy With Logit Loss, data la natura binaria della classificazione. Questa loss si basa sulle probabilità delle predizioni, dunque penalizza, con un valore di loss più alto, laddove la predizione ha un risultato "misto" (ad esempio la probabilità che sia classe positiva è il 51%). L'applicazione della BCE con Logit Loss comporta che all'output layer venga applicata automaticamente la funzione di attivazione sigmoidale; questa soluzione è riportata essere più consistente dell'applicazione della sigmoid seguita dall'utilizzo della Binary Cross Entropy tradizionale.

Il training del modello avviene su 50 epoche, con una batch size di 1024 e learning rate fissato a 0.001, con il train loader che carica i batch del training set per il modello, divisi, naturalmente, in variabile dipendente (y, il booleano che indica se quel file contiene un difetto o meno) e variabili indipendenti (x, le 24 feature numeriche utilizzate per la predizione).
Per evitare l'accumulo del gradiente, per ogni batch viene esplicitato l'azzeramento dello stesso, e viene computata loss ed accuracy. L'accuracy consiste nell'applicazione della sigmoide sull'output della rete (nb. il calcolo è indipendente dalla computazione della loss, quindi non c'è sovrapposizione con la sigmoid applicata dalla BCE with Logit Loss), per poi contare quante delle predizioni della rete sono corrette. Infine, il risultato viene trasformato in percentuale. L'output della loss e dell'accuracy avviene per ogni epoca.
Finito il training, inizia il testing, naturalmente senza modifica del gradiente. I dati del training set vengono dati in input al modello e per ogni risultato ricevuto si applica la sigmoide, si arrotonda per stabilire se la predizione tende alla classe 1 o alla classe 0. I risultati vengono inseriti in una lista, in modo da poter poi ottenere un report delle performance tramite confusion matrix, f1 e Auc Roc, confrontandoli con le vere y.

2.2.2 10 Fold Cross Validation

La 10-Fold Cross Validation viene utilizzata sia per avere una stima più realistica dell'errore, ma anche per fare tuning degli iperparametri della rete. Esclusivamente per questa applicazione, viene sfruttato Keras al posto di PyTorch. La rete viene creata allo stesso modo, eccetto per il layer di output che questa volta contiene l'attivazione sigmoidale. In questo caso, inoltre viene utilizzato SGD al posto di ADAM e viene introdotto il weight decay.
I parametri che vengono tunati sono l'hidden size (inteso come numero di layer hidden), il learning rate, il momentum e proprio il parametro del weight decay. Per ogni attributo vengono scelti 3 valori, in seguito verranno generate tutte le 81 possibili combinazioni e per ognuna di esse verrà eseguito il training e il testing del modello tramite 10 Fold Cross Validation. Su questa rete si è potuto sperimentare in maniera più efficace con le funzioni di attivazione. Tra quelle testate, la migliore è risultata la LeakyReLU, ovvero una versione della ReLU che non azzera i valori negativi, ma li moltiplica per un coefficiente (usato 0.1). Inoltre, viene sfruttato il meccanismo dell'early stopping per fermare il processo di training quando la loss non riporta miglioramenti per un certo numero di epoche. I risultati ottenuti su tutte le combinazioni vengono riportate in un file .csv, che verrà discusso nella sezione dei risultati.

2.2.3 Leave One Project Out

Con questa tecnica vengono isolate tutte le righe del dataset relative a un progetto e vengono utilizzate come test set, mentre l'allenamento della rete avviene sui rimanenti 31 progetti. Iterativamente, tutti i progetti saranno testati. I pesi del modello, il gradiente, la loss e l'accuracy vengono azzerate ad ogni iterazione, in modo che per ognuna delle 32 combinazioni di train e test set, il modello parta da zero.

SEZIONE 3

Risultati

FirstNN è stato il primo tentativo di utilizzare la rete. Lo split singolo che viene effettuato per train e test set comporta il fatto che i risultati non siano propriamente generalizzabili a tutto il dataset, in quanto potrebbero essere influenzati proprio dai dati che finiscono nei due insiemi (nonostante il campionamento stratificato). Tuttavia, empiricamente, la funzione di attivazione tanh si è rivelata più performante della relu e della sigmoidale, dunque è stata scelta. Per lo split utilizzato in questo esperimento, i risultati non sono particolarmente positivi. Nonostante l'accuracy sia del 69%, quando si tratta di individuare la classe positiva, la rete fa difficoltà: il numero dei falsi negativi (istanze realmente difettate, ma classificate come non difettate) è più alto dei veri positivi. In termini assoluti (sulla dimensione del dataset) non ci sono tantissimi falsi positivi, ma quando si comparano i numeri con i due precedentemente citati, si nota ancor di più la difficoltà della rete - vedi 3.1

I risultati della 10 Fold Cross Validation invece, mostrano che la combinazione utilizzata in FirstNN è già una delle migliori. Infatti, molte delle combinazioni testate danno risultati molto simili (anche variando funzione di attivazione) sia in termini di loss che in termini di accuracy, con eccezioni per quelle reti che presentano un learning rate troppo basso o la combinazione learning rate alto e momentum alto, che oscillano molto nei risultati. Il decay non sembra avere un particolare impatto sulla rete. Tutto sommato, è evidente che con un problema di questo tipo ci siano poche ottimizzazioni da poter fare.

L'applicazione della Leave One Project Out è la più interessante, in quanto permette di confrontare i risultati con quelli ottenuti tramite Machine Learning dagli autori del paper

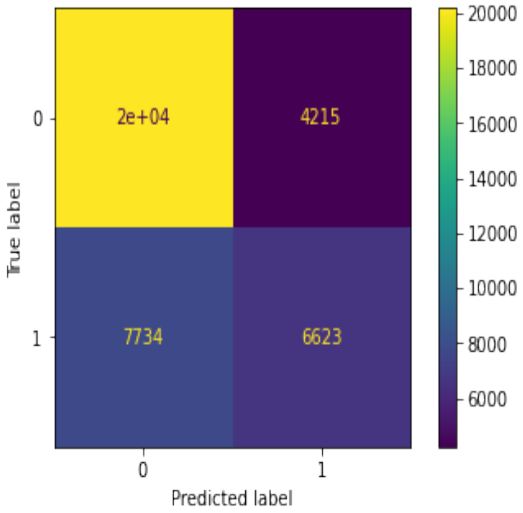

Figura 3.1: FirstNN - Matrice di confusione

da cui il dataset è tratto. Per ogni rete addestrata e testata, vengono salvati auc-roc score, f1 score, precision e recall per poi analizzare la distribuzione di queste tramite boxplot. Per questa analisi viene utilizzato R.
Come visibile dal grafico 3.2, il MultiLayerPerceptron medianamente si classifica al secondo posto per la curva AUC-ROC, con una variabiltà piuttosto ristretta. In linea di massima, lo score AUC-ROC oscilla tra 0.6 e 0.65, indice di una classificazione accettabile, soprattutto quando comparata ai modelli di machine learning, eccezion fatta per il classificatore Extra Trees.

Tuttavia quest'ultimo classificatore risulta essere il peggiore quando si valuta F1 (vedi 3.3), media armonica tra precision e recall, mentre proprio in questa metrica il MLP batte tutti gli altri modelli, a pari merito con Local Outlier Factor. Il risultato però ha una variabilità piuttosto alta e in generale nessun modello eccelle in questo score.

Non riportati nell'articolo ma presenti nel replication package, sono anche i dati relativi alla precision e alla recall del modello, che considero metriche interessanti da analizzare nella valutazione delle performance. La precision ha una variabilità molto alta, anche se il valore mediano del Multilayer Perceptron risulta superiore agli altri. Infine, per la recall il MLP fa peggio di tre metodi di machine learning su sei, con un risultato piuttosto scarso.
Ciò significa che seppure la rete si comporti sufficientemente bene da non classificare molte istanze prive di bug come difettate, non ha una sensibilità adatta alla classificazione delle istanze positive (difettate), in quanto molte di queste passano il vaglio del classificatore come

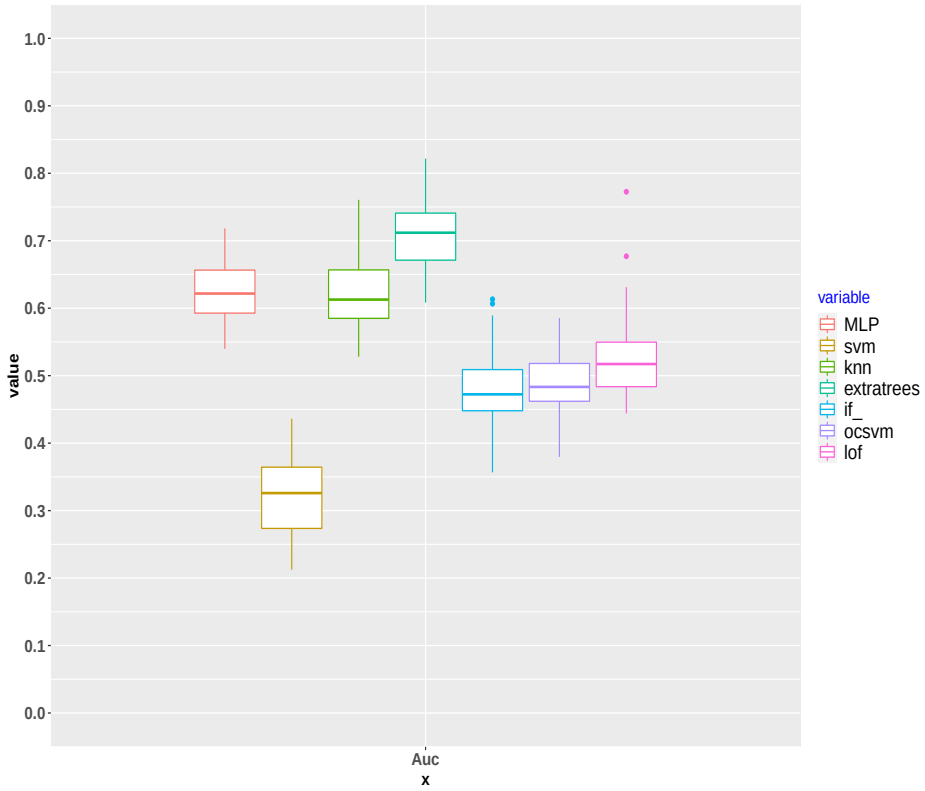

Figura 3.2: Comparativa MLP e ML per AUC-ROC score

negative.

Per concludere, la rete creata con MultiLayer Perceptron può essere considerata un'alternativa ai modelli di machine learning tradizionali quando si opera su dataset di Fine Grained Just In Time Defect Prediction come questo. Come visto, ogni classificatore ha i suoi punti di forza e di debolezza - vedesi l'ExtraTree che eccelle in Auc score ma fa male in F1, Precision e Recall - e anche provando con il deep learning si incappa negli stessi problemi. Ciò è sintomo anche di un dataset per il quale, non tanto per la costruzione quanto per le metriche considerate, risulta difficile fare una predizione efficace dei difetti software.

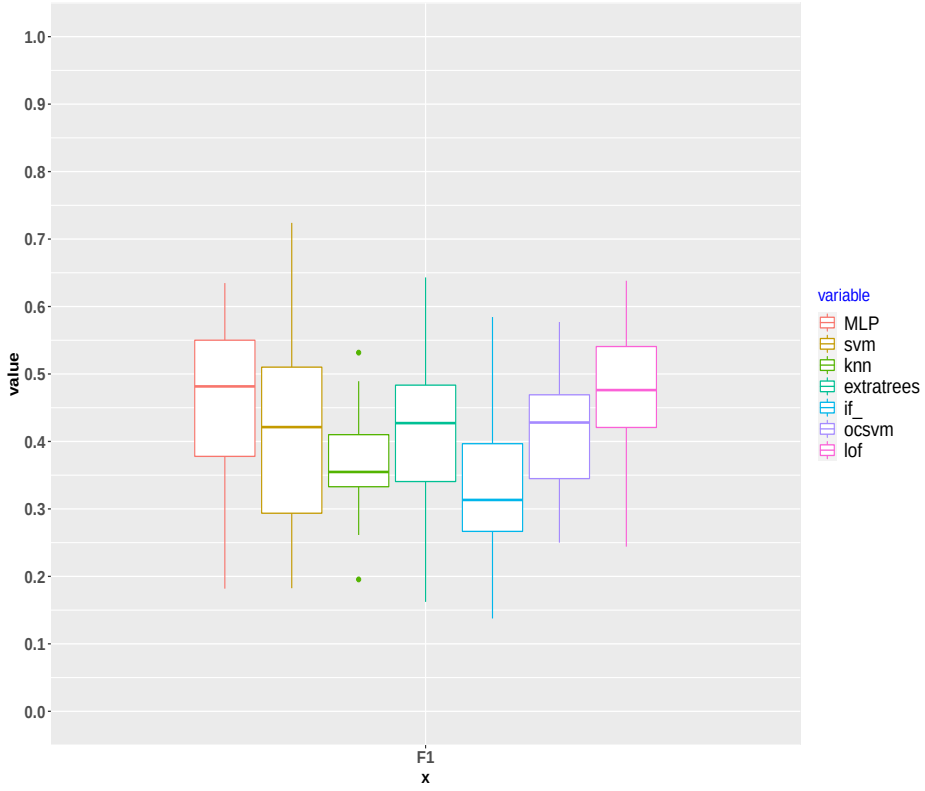

Figura 3.3: Comparativa MLP e ML per F1 score

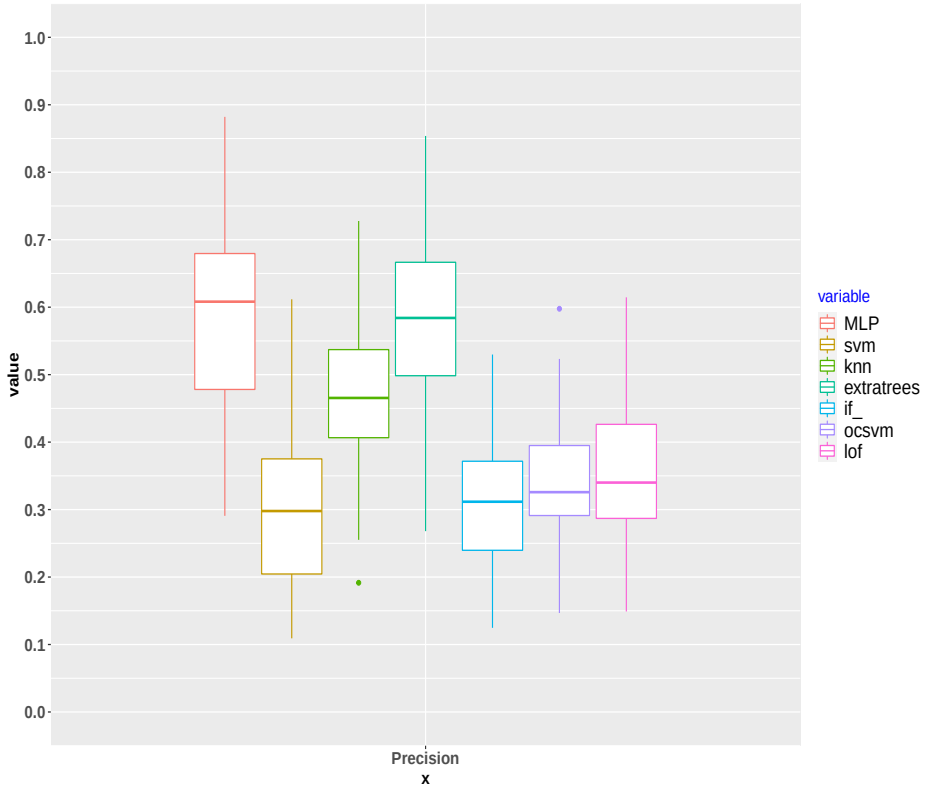

Figura 3.4: Comparativa MLP e ML per Precision

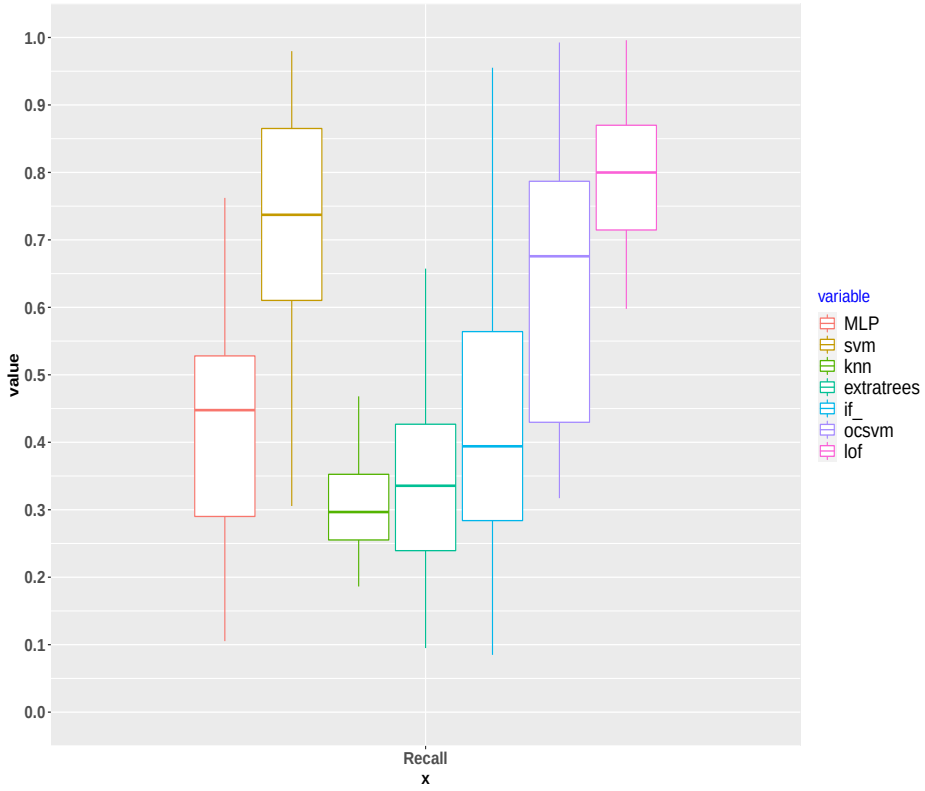

Figura 3.5: Comparativa MLP e ML per Recall

Bibliografia

[1] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013. (Citato alle pagine 1 e 2)

[2] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 432–441, 2013. (Citato a pagina 2)

[3] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019. (Citato a pagina 2)

[4] F. Lomio, L. Pascarella, F. Palomba, and V. Lenarduzzi, "Regularity or anomaly? on the use of anomaly detection for fine-grained just-in-time defect prediction," 05 2022. (Citato a pagina 2)