Utilizzo IA per rilevamento frodi bancarie

Gabriele Nigro, matricola 758413, g.nigro17@studenti.uniba.it

Link alla repo GitHub: https://github.com/xLightingBlade/ICON-Project

ICON AA 2023-24

Tabella dei contenuti

Link alla repo GitHub: 1
Introduzione 1
Sommario 2
Elenco argomenti trattati 2
Dataset 2
Descrizione 2
Apprendimento supervisionato 6
Apprendimento non supervisionato 8
Reti Neurali 8
Reti bayesiane 10
Conclusioni 12
Riferimenti Bibliografici 13

Introduzione

Obiettivo di questo progetto sviluppato per l'esame di Ingegneria della Conoscenza è quello di esplorare e comparare varie metodologie per il rilevamento di frodi bancarie sulla base dei dati per ogni transazione.

Sommario

Il progetto si compone di una prima parte di caricamento, esplorazione e preprocessing necessario sul dataset, per poi passare alle diverse fasi di machine learning.
Nella parte di apprendimento supervisionato si considerano cinque modelli diversi e come migliorarne le performance.
Per la parte di apprendimento non supervisionato si utilizza una Isolation Forest, anche in quanto metodo di visualizzazione dei dati.
Si prova poi ad utilizzare una rete neurale ai fini della classificazione, ed infine si tenta di apprendere una rete bayesiana a partire da un subset dei dati e a usarla al fine della classificazione.
Infine si confrontano le performance delle varie soluzioni.

Elenco argomenti trattati

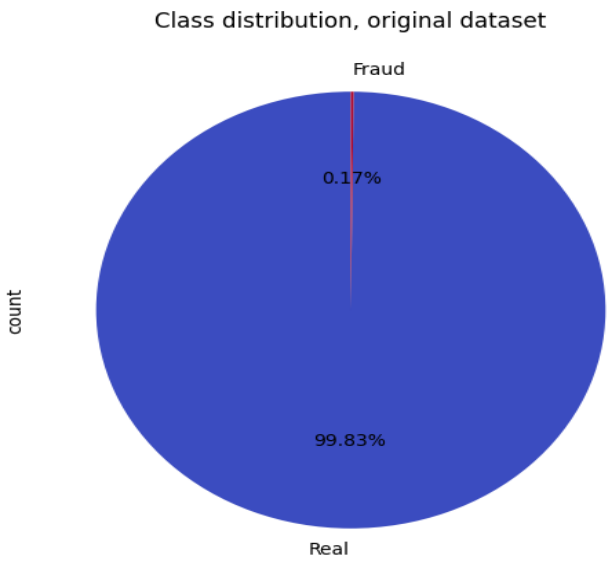
- Il dataset preso in considerazione
- Apprendimento supervisionato, non supervisionato, reti neurali
- Reti bayesiane

In maniera più o meno ampia, trattati nei capitoli 7-8-9-10 del libro di testo [1]

Dataset

Descrizione

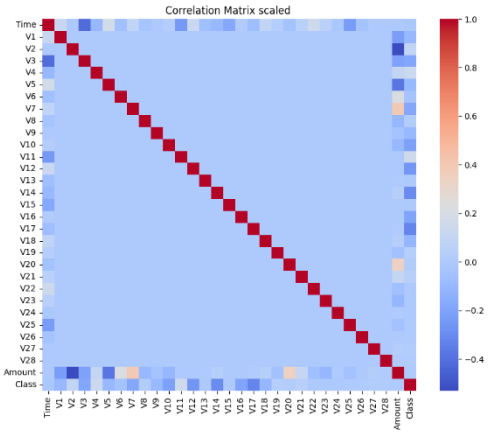
Il dataset utilizzato è il Credit Card Fraud Detection prodotto dal Machine Learning Group dell'Università di Bruxelles. Esso consiste di 31 features: 28 feature, nominate V1, V2, …, V28 sono il risultato dell'applicazione di PCA sui dati originali, vi sono poi altre due feature numeriche separate chiamate 'Amount' e 'Time', rappresentanti rispettivamente l'importo della transazione e il numero di secondi trascorsi tra ogni transazione e la prima del dataset.
Sorge subito un chiaro problema di interpretabilità poiché le feature Vn sono il risultato di una PCA, dunque combinazioni lineari delle feature dei dati originali, le quali non vengono rivelate per motivi di confidenzialità dei dati.
Il dataset è composto di 284,807 record, dei quali minuscola parte è etichettata come "Fraud (1)":

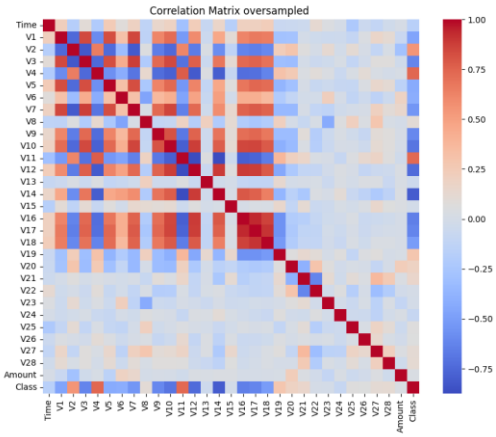


Per confrontare le performance dei vari metodi di apprendimento su un tale dataset, si mantegono due copie dei dati, una del dataset così composto e una sulla quale viene applicato un oversampling della classe di minoranza con la tecnica SMOTE [2], che genera esempi sintetici vicini a quelli reali nello spazio delle features. Usando l'implementazione disponibile di Imbalanced Learn con strategia di sampling 'auto', si ottiene un dataset col circa 45% di record fraudolenti. Il fatto che non si raggiunga un 50/50 sul totale dei record è dovuto al fatto che, come da best practice, l' oversampling va effettuato esclusivamente sul training split e mai altrove: fare ciò altererebbe la distribuzione dei dati nei dati di test, che invece devono continuare a rappresentare il mondo reale

In dataset così fortemente sbilanciati, una metrica come l'accuratezza diventa inutile e bisogna considerarne altre. Per questo particolare caso d'uso, siamo decisamente interessati a massimizzare la Recall, ossia minimizzare il numero di frodi etichettate come transazioni genuine. Questo tipo di errore, noto come falso negativo, ha conseguenze ben peggiori del suo contrario: far passare una transazione fraudolenta è decisamente peggio di bloccarne per sbaglio una genuina.

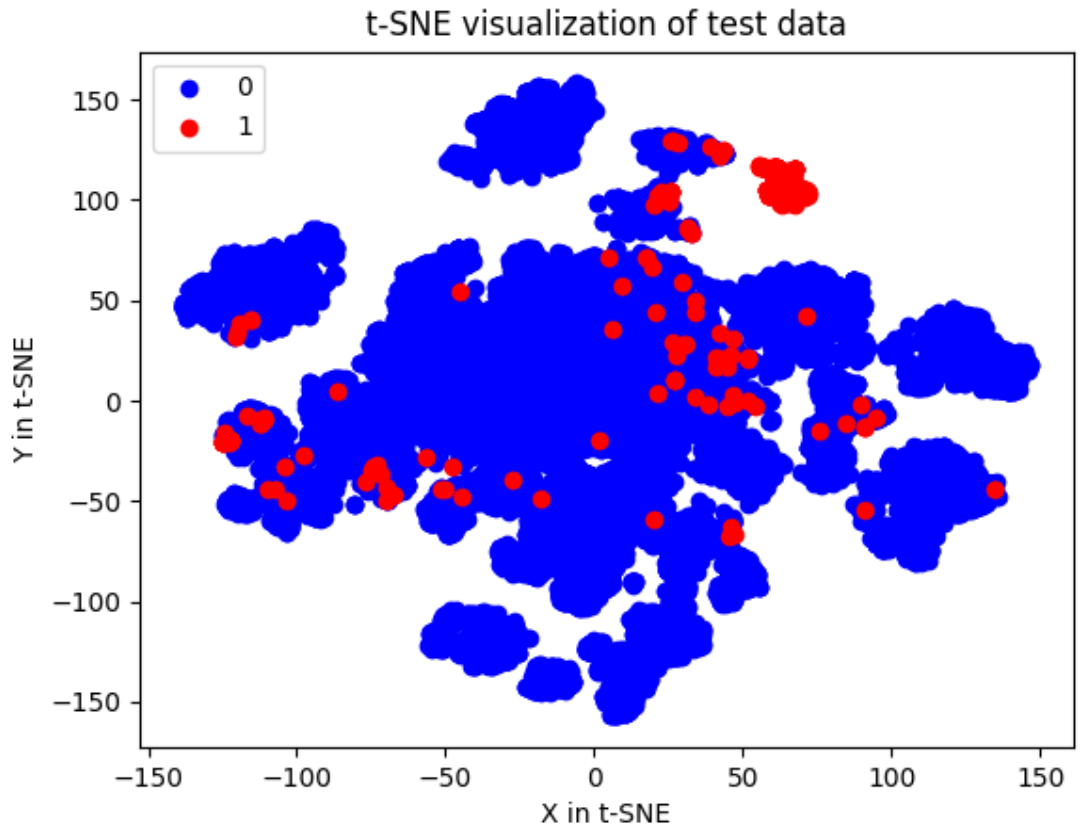
Il dataset si presenta inoltre con nessuna singola cella con valori mancanti o NaN e con un numero di record duplicati estremamente basso, del quale nessuno appartenente ai pochi casi etichettati come fraudolenti, semplificando un po' il lavoro.
Quello che salta subito all'occhio è che le due features Amount e Time sono di diversi ordini di grandezza rispetto a tutte le altre. Alcune classi di classificatori, come la SVM usata in questo progetto, risentono fortemente di features di diversa magnitudine, perciò applichiamo a queste due colonne uno scaling tramite la classe RobustScaler di Scikit Learn.
Si è anche eseguito il plot delle correlazioni tra le feature nel dataset, sebbene per uno scopo puramente "di curiosità" dato che ai fini dell'apprendimento questa cosa non fa alcuna differenza (è vero che, ad esempio, la regressione logistica lavora anche sul presupposto di assenza di features perfettamente correlate, ma non è questo il caso).



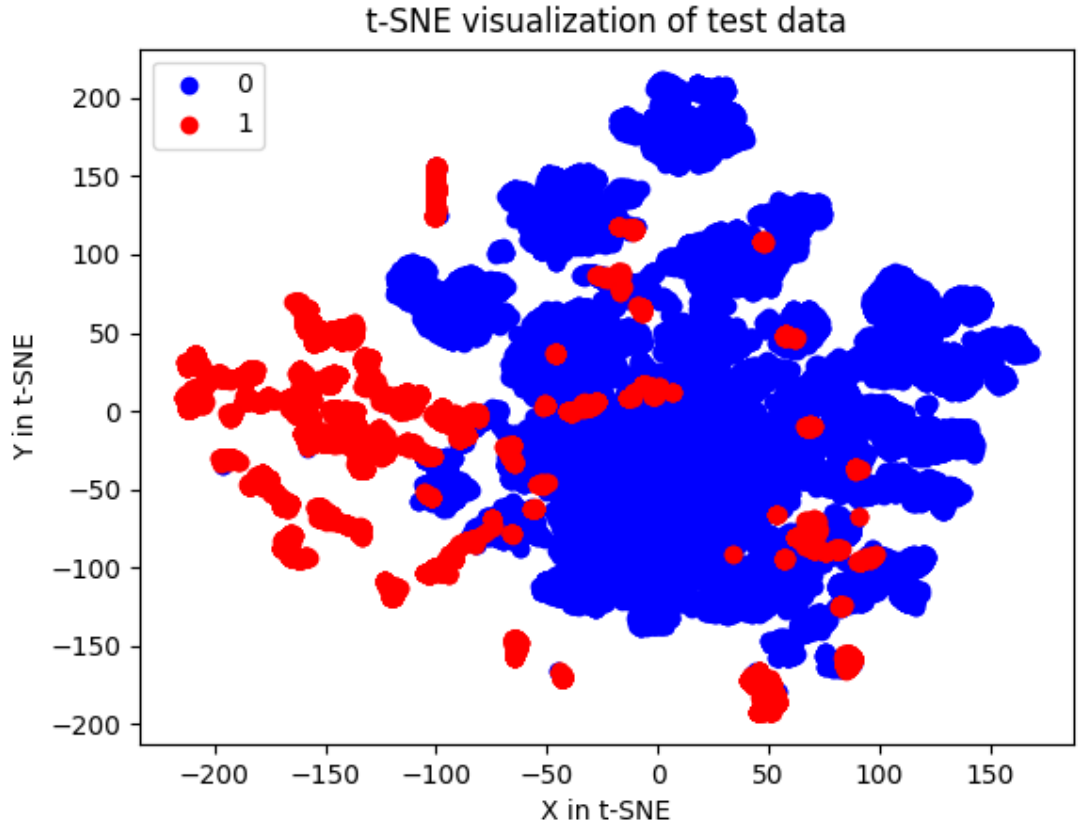


Infine ho provato ad applicare al dataset una tecnica chiamata TSNE, usata per visualizzare in uno spazio 2D dati a più elevata dimensionalità, per cercare un'eventuale "forma" dei dati. Ho testato diverse combinazioni di valori per il parametro di perplexity (il parametro principale del TSNE) e il numero di sample da prelevare dal dataset per costruire il sottoinsieme da usare per l'elaborazione (Non usare l'intero dataset è una scelta quasi obbligata dai tempi di elaborazione dell'algoritmo). Tuttavia questo è risultato un'esperimento abbastanza non degno di nota:

Su dataset non oversampled:



Su dataset oversampled:



Apprendimento supervisionato

La task da apprendere, in questo e i successivi paragrafi, è una task di classificazione binaria: date le 30 feature numeriche in input, vogliamo predire se una transazione ha per valore della feature target, 'Class', il valore 1 (che indica frode) oppure 0 (che indica transazione genuina)

Per questa fase ho scelto di addestrare e comparare cinque modelli di classificatori, tutti tramite implementazione disponibile su Scikit Learn:
LogisticRegression, SVC, RandomForest, DecisionTree e KNeighbors
Per ogni esperimento utilizzando questi classificatori, essi vengono applicati su entrambe le "copie" dei dati (quella solo scalata e quella con oversampling)

Ho effettuato una prima passata di cross_validate utilizzando i classificatori inizializzati senza alcun tipo di iperparametro custom (salvo un classico random state 42) e utilizzando un cross validator della classe StratifiedKFold, particolarmente indicato per i dataset sbilanciati.
Esempio : "LogisticRegression": LogisticRegression(random_state=42)
Dei risultati di queste run si monitorano le seguenti metriche: accuracy, precision, recall e F1 score.

L'accuracy, per classificazioni su dataset sbilanciati, è una metrica assolutamente fuorviante: se si hanno 99 record positivi ed uno negativo, predirli tutti come positivi porterà una accuracy del 99%.
I risultati di questa prima run, su entrambi i dataset, sono i seguenti:

Classifier	accuracy_mean	accuracy_std	precision_mean	precision_std	recall_mean	recall_std	f1_mean	f1_std
LogisticRegression	0.9992	6e-05	0.87649	0.03156	0.62393	0.03967	0.72772	0.02492
RandomForest	0.99956	4e-05	0.95422	0.0194	0.78254	0.03643	0.85897	0.01516
KNearest	0.99946	5e-05	0.94268	0.02016	0.72973	0.02568	0.82231	0.01799
Support Vector Classifier	0.99939	6e-05	0.94069	0.02023	0.68895	0.05133	0.79371	0.02897
DecisionTreeClassifier	0.99918	9e-05	0.75756	0.03783	0.7785	0.03567	0.76684	0.02429

Oversampled:

Classifier	accuracy_mean	accuracy_std	precision_mean	precision_std	recall_mean	recall_std	f1_mean	f1_std
LogisticRegression	0.96392	0.0006	0.9752	0.00094	0.89658	0.00246	0.93423	0.00118
RandomForest	0.99981	9e-05	0.99956	0.00018	0.99976	0.00032	0.99966	0.00017
KNearest	0.99813	0.00015	0.99362	0.00059	0.99987	0.00026	0.99673	0.00026
Support Vector Classifier	0.98242	0.00078	0.98821	0.00072	0.94983	0.0034	0.96863	0.00145
DecisionTreeClassifier	0.99754	0.00016	0.99423	0.00044	0.99717	0.00057	0.9957	0.00029

Si osserva innanzitutto un chiaro e netto miglioramento dato dall'oversampling ed inoltre il valore dell'accuracy è praticamente sempre al >99%, sebbene le altre metriche non lo siano.
Da queste prime due run i modelli più performanti, prendendo in considerazione la recall come metrica principale, sono RandomForest e KNearest.

Successivamente, ho eseguito una gridsearch per ottimizzare alcuni iperparametri per ogni classificatore.
Definita una grid dei parametri per ogni classificatore, ho utilizzato la funzione GridSearchCv per eseguire la ricerca della combinazione di iperparametri più ottimale. La Grid Search restituisce il classificatore con il miglior punteggio sulla partizione di test, su una metrica specificata (ancora una volta, la recall). Per ognuno di questi si raccoglie in un dataframe i risultati, che sono i seguenti:

No oversampling:

Classifier	phase	C	Accuracy(var)	Precision(var)	Recall(var)	F1(var)	max_depth	n_estimators	n_neighbors
LogisticRegression	training	1000.0	0.99921(0.0)	0.8852(8e-05)	0.62653(8e-05)	0.73369(6e-05)	nan	nan	nan
LogisticRegression	testing	1000.0	0.99919(0.0)	0.87321(0.00085)	0.61987(0.00126)	0.72394(0.00045)	nan	nan	nan
RandomForest	training	nan	1.0(0.0)	1.0(0.0)	0.99949(0.0)	0.99975(0.0)	nan	150.0	nan
RandomForest	testing	nan	0.99956(0.0)	0.95423(0.00039)	0.78458(0.00095)	0.86038(0.00014)	nan	150.0	nan
KNearest	training	nan	0.99956(0.0)	0.95988(0.0)	0.77795(0.00011)	0.85935(4e-05)	nan	nan	5.0
KNearest	testing	nan	0.99946(0.0)	0.94268(0.00041)	0.72973(0.00066)	0.82231(0.00032)	nan	nan	5.0
Support Vector Classifier	training	10.0	0.9998(0.0)	0.99943(0.0)	0.8872(3e-05)	0.93997(1e-05)	nan	nan	nan
Support Vector Classifier	testing	10.0	0.99946(0.0)	0.95291(0.00035)	0.72154(0.00217)	0.82003(0.0007)	nan	nan	nan
DecisionTreeClassifier	training	nan	0.99961(0.0)	0.94927(6e-05)	0.81657(4e-05)	0.8779(2e-05)	5.0	nan	nan
DecisionTreeClassifier	testing	nan	0.99949(0.0)	0.91125(0.00114)	0.7866(0.00176)	0.84271(0.00015)	5.0	nan	nan

Con oversampling:

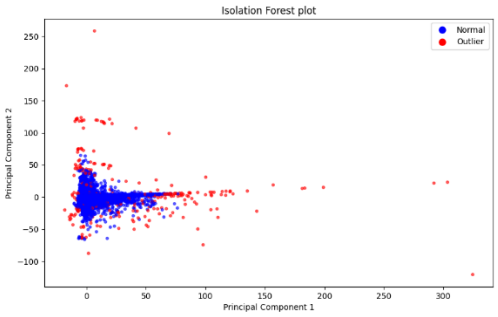
Classifier	phase	C	Accuracy(var)	Precision(var)	Recall(var)	F1(var)	max_depth	n_estimators	n_neighbors
LogisticRegression	training	10.0	0.9548(0.0)	0.97522(0.0)	0.92173(0.0)	0.94772(0.0)	nan	nan	nan
LogisticRegression	testing	10.0	0.9548(0.0)	0.97525(0.0)	0.92171(0.0)	0.94772(0.0)	nan	nan	nan
RandomForest	training	nan	1.0(0.0)	1.0(0.0)	1.0(0.0)	1.0(0.0)	nan	100.0	nan
RandomForest	testing	nan	0.99987(0.0)	0.99977(0.0)	0.99993(0.0)	0.99985(0.0)	nan	100.0	nan
KNearest	training	nan	0.99688(0.0)	0.99309(0.0)	0.99994(0.0)	0.99651(0.0)	nan	nan	15.0
KNearest	testing	nan	0.99646(0.0)	0.99216(0.0)	0.99994(0.0)	0.99604(0.0)	nan	nan	15.0
Support Vector Classifier	training	100000.0	0.99992(0.0)	0.99985(0.0)	0.99998(0.0)	0.99991(0.0)	nan	nan	nan
Support Vector Classifier	testing	100000.0	0.99947(0.0)	0.99895(0.0)	0.99985(0.0)	0.9994(0.0)	nan	nan	nan
DecisionTreeClassifier	training	nan	1.0(0.0)	1.0(0.0)	1.0(0.0)	1.0(0.0)	nan	nan	nan
DecisionTreeClassifier	testing	nan	0.9981(0.0)	0.99696(0.0)	0.99879(0.0)	0.99787(0.0)	nan	nan	nan

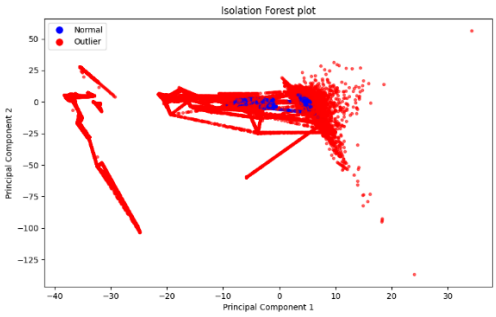
Le metriche registrate, con un focus sempre sulla recall, mostrano un grande passo in avanti dato dall'oversampling e il confronto delle varianze tra i punteggi in train e test ci dicono che non dovrebbe esserci particolare overfitting.

Apprendimento non supervisionato

Nonostante il dataset sia già etichettato in frodi e non, ho voluto sperimentare con una tecnica di apprendimento non supervisionato chiamata Isolation Forest. Si tratta di un algoritmo pensato per rilevare anomalie nei dati usando alberi binari, in modo abbastanza simile ad un albero di decisione, sebbene differisca da quest'ultimo sulle computazioni eseguite.
L'idea di fondo è che i dati anomali sono pochi e facilmente separabili dal resto e per separarli, si partiziona ricorsivamente l'insieme dei dati selezionando una feature a caso su cui splittare, selezionando come valore soglia un valore scelto casualmente all'interno dei range di valori per quella feature. Dati anomali si troveranno assieme con maggior probabilità nella branch più piccola. Ad ogni dato si assegna un anomaly score proporzionale al numero di divisioni necessarie ad isolarlo dal resto dei dati: più divisioni indicano meno anomalia, nel caso particolare dell'implementazione disponibile in Scikit Learn, agli outlier è assegnato -1 e agli inlier 1.

L'esperimento tuttavia non ha dato risultati incoraggianti: sul dataset privo di oversampling si registra un roc_auc_score di appena 0.3, mentre su quello oversampled un comunque migliore 0.8. Nonostante il decente punteggio nel secondo caso, si nota come, plottando l'output dell'algoritmo in uno spazio 2d, il primo caso sembri molto più "esplicativo".
Si rende quindi necessario uno studio più approfondito della materia per capire la natura di queste problematiche.
A seguire, le due visualizzazioni (a destra su oversampled):





Reti Neurali

In soldoni, una rete neurale è un tipo di modello di machine learning ispirato dal meccanismo di ragionamento del cervello e la struttura dei neuroni. È un tipo di modello organizzato in strati, ognuno dei quali applica una qualche funzione lineare f(x), ognuna di queste mappa un certo vettore x in input in un certo vettore f(x) in output, ogni valore del vettore è detto unità o neurone. Ogni funzione lineare è seguita a sua volta da una funzione detta di attivazione, nonlineare, che produce l'output definitivo di uno strato.

Per questo progetto si è deciso di usare una semplice rete neurale formata da tre strati "densi", cioè tali che ogni unità di output di uno strato è collegata a tutte le unità dello strato successivo. Come layer di output vi è ancora un layer denso, di una sola unità e applicante funzione di attivazione sigmoide, per poter effettuare classificazione binaria.

Ho utilizzato Keras, l'API ad alto livello di Tensorflow, per definire la struttura base del modello:

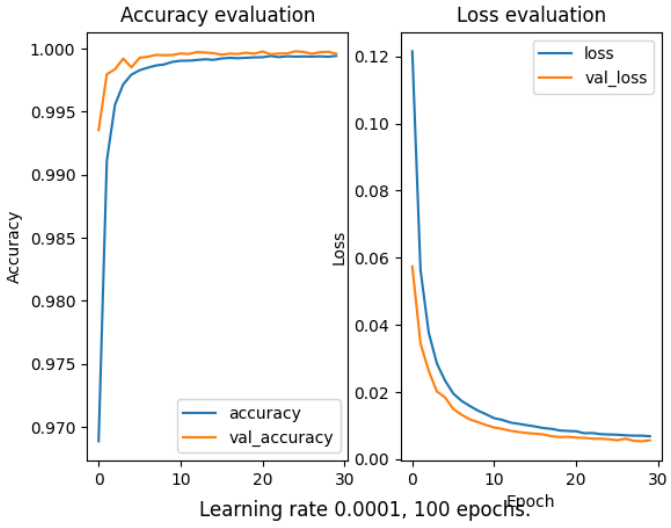
```python
from keras_tuner import HyperModel
import tensorflow as tf
import numpy as np

class MyHypermodel(HyperModel):  1 usage  xLightingBlade
    def __init__(self, inputs):  xLightingBlade
        self.inputs = inputs

    def build(self, hp):  1 usage (1 dynamic)  xLightingBlade
        normalizer = tf.keras.layers.Normalization(axis=-1)
        normalizer.adapt(np.array(self.inputs))
        dense_units = hp.Int('units', min_value=32, max_value=256, step=32)
        model = tf.keras.models.Sequential([
            tf.keras.Input(shape=(30,)),
            normalizer,
            tf.keras.layers.Dense(units=dense_units, activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.l2(0.0001)),
            tf.keras.layers.Dropout(0.3),
            tf.keras.layers.Dense(units=dense_units, activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.l2(0.0001)),
            tf.keras.layers.Dropout(0.3),
            tf.keras.layers.Dense(units=dense_units, activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.l2(0.0001)),
            tf.keras.layers.Dense(units=1, activation='sigmoid'),]
        )

        model.compile(optimizer=tf.optimizers.Adam(learning_rate=0.0001),
                      loss="binary_crossentropy",
                      metrics=['accuracy', tf.keras.metrics.Precision(), tf.keras.metrics.Recall()])
        model.summary()
        return model
```

Utilizzo inoltre keras_tuner per poter effettuare una run di tuning degli iperparametri della rete. In questo caso, l'unico parametro sotto tuning è il numero di unità per layer, il resto viene lasciato come valore fisso. Tra ogni layer denso viene piazzato un layer di Dropout, il quale azzera una certa percentuale dei valori in uscita dal layer, in questo caso il 30%. Assieme al dropout, ogni layer denso applica una regolarizzazione L2, un termine che penalizza i modelli troppo complessi secondo una certa misura. È stato dimostrato in molteplici paper, come in , che regolarizzazione e dropout in combinazione sono un'ottimo modo per ridurre l'overfitting.

Viene dunque avviata la run di tuning, per 30 epoche per trial, con un validation split del 20% e early stopping con pazienza di 5 epoche, monitorando la validation loss. Terminata, si utilizza il modello risultante per chiamare evaluate() sui dati di test, ottenendo queste metriche:

```
Test loss: 0.005788275506347418, test accuracy: 0.9995506405830383, test precision: 0.999011218547821, test recall: 0.9999780058860779
F1 score: 0.9994943784293567
```

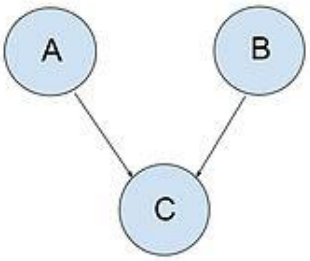
Il plot delle curve di apprendimento ci dice che non sembra esserci particolare overfitting:



Piccolo refuso: 30 epochs, non 100

Reti bayesiane

Consideriamo le feature del dataset come un insieme di variabili aleatorie. Una rete bayesiana (o belief network) è un grafo orientato aciclico (DAG) che ha per nodi le features e archi a collegare feature condizionalmente dipendenti. È l'indipendenza condizionale ad essere implicita qui:
A ===➔ B ===➔ C indica che la variabile C è indipendente da A se si osserva B, che detto "genitore" di C. Dunque la dipendenza tra due variabili A e B è detta condizionale perché non avviene fino a quando non si osserva una terza variabile C, mentre nel caso contrario, *l'indipendenza* condizionale implica che, data una variabile A e un'evidenza B, se una seconda evidenza C non cambia la probabilità di A, allora A e C sono condizionalmente indipendenti data B



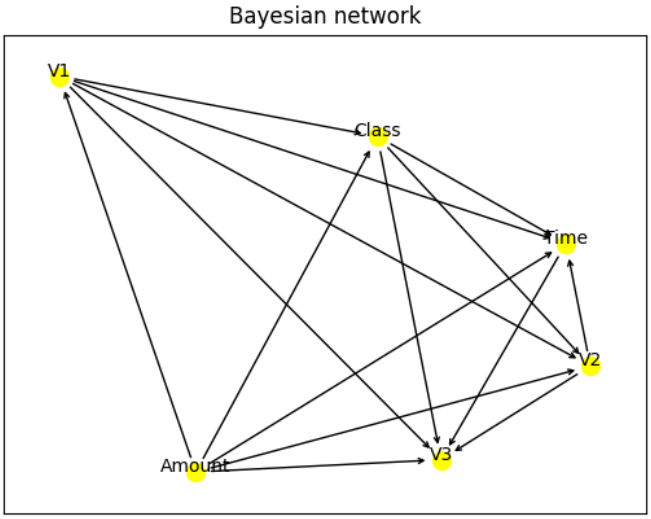
Esempio dove A e B condizionalmente dipendenti data C

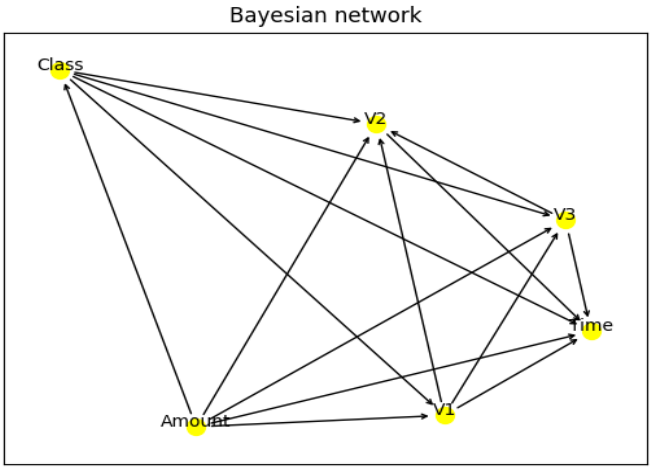
Una rete bayesiana modella tutto ciò e porta inoltre con sé le probabilità condizionali di ogni variabile dati i genitori (ad esempio P(C | A,B))

Su Python è disponibile la libreria pgmpy che mette a disposizione metodi per effettuare inferenza su reti bayesiane e anche per eseguire structure learning di un dataset, ovvero task di machine learning per apprendere automaticamente una struttura di rete bayesiana a partire anche da un

dataframe Pandas come in questo caso specifico, e di calcolare poi le probabilità condizionali delle variabili della rete. La strada più diretta per fare structure learning tramite pgmpy è utilizzare la classe HillClimbSearch() che, nomen omen, conduce una ricerca locale nello spazio dei possibili DAG. In particolare una hill climb è un tipo di ricerca che ambisce a massimizzare una certa valutazione, che nel nostro caso è il Bayesian Information Criteria score, il logaritmo della funzione di verosimiglianza più un termine di penalizzazione per i modelli complessi (regolarizzazione).

Le reti bayesiane apprese sono due, entrambe su 100k record provenienti dal dataset oversampled e discretizzati tramite KBinsDiscretizer() con strategia kmeans e differiscono tra di loro per il numero di bins utilizzati per la discretizzazione, rispettivamente 10 e 20. Tuttavia, per l'apprendimento della struttura si considerano solo poche colonne del dataframe, cioè Amount, Time, V1, V2 e V3, oltre a Class. Questo perché l'aggiunta di una sola altra feature comporta aumenti vertiginosi dei tempi di apprendimento.
La rappresentazione grafica delle due reti bayesiane sono le seguenti:





Una volta ottenuto lo scheletro della rete bayesiana, pgmpy permette di calcolare anche le distribuzioni di probabilità per ogni feature, tramite la classe MaximumLikelihoodEstimator il cui

obiettivo è appunto massimizzare la likelihood, la probabilità dei dati dato il modello, la probabilità che un certo modello possa aver generato i dati che abbiamo. Terminata la computazione si salva il modello in locale per poterlo riutilizzare.
Un'ultima applicazione di questa libreria per gli scopi del progetto è quella di poter fare inferenza statistica sui dati, utilizzando la classe VariableElimination per applicare l'omonimo algoritmo di inferenza esatta.

Il primo esperimento sull'inferenza (esatta) consiste nel generare, usando il modello bayesiano appreso precedentemente, un insieme di record su cui fare inferenza, per predirne la classe. Queste predizioni vengono raccolte ed usate per calcolarne precision e recall.
La prima rete ha ottenuto valori di precision pari a 0.949 e recall pari a 0.916, la seconda valori di 0.96 per le stesse metriche.

Un secondo approccio di più grande utilità sarebbe quello di effettuare l'inferenza su dati reali e non generati tramite la rete. Ho pensato che la scelta giusta fosse quella, sia in fase di structure learning che di inferenza, di prelevare i record dal dataset facendo attenzione a prelevare almeno un valore per bin per feature, ma ciò evidentemente non è bastato ad evitare errori: con tutta probabilità di pesca sempre un qualche record contenente una certa combinazione di valori per le features che la rete non ha mai incontrato, questo genera un'eccezione KeyError:

```
Traceback (most recent call last):  Explain with AI
  File "C:\Users\Gabriele\Desktop\ICON Project\main.py", line 299, in <module>
    main()
  File "C:\Users\Gabriele\Desktop\ICON Project\main.py", line 93, in main
    bayesian_network_inference(bayesian_model, samples)
  File "C:\Users\Gabriele\Desktop\ICON Project\bayesian_utils.py", line 72, in bayesian_network_inference
    res = inference.query(variables = ['Class'], evidence = data)
  File "C:\Users\Gabriele\Desktop\ICON Project\.venv\lib\site-packages\pgmpy\inference\ExactInference.py", line 339, in query
    indexer[index] = phi.get_state_no(
  File "C:\Users\Gabriele\Desktop\ICON Project\.venv\lib\site-packages\pgmpy\utils\state_name.py", line 72, in get_state_no
    return self.name_to_no[var][state_name]
KeyError: 0.0
```

Conclusioni

Le metriche ottenute tramite i metodi menzionati nei punti precedenti sono buone, almeno tanto quanto basta per poter passare a valutare suddetti metodi su altri dati al di fuori del dataset utilizzato. Questo però non vale per quanto ottenuto con l'Isolation Forest, sebbene fosse più un esercizio che una vera utilità, data la già esistente classificazione dei dati.

Un argomento che meriterebbe ulteriore focus, in caso di ulteriori lavori su questo caso, è l'utilizzo di tecniche di under/oversampling come la SMOTE qui utilizzata. Durante le fasi finali di questo progetto, sono venuto a conoscenza di come sia un'opinione, anche abbastanza popolare, che queste tecniche portino ad un reale beneficio più raramente di quanto si pensi e spesso invece siano causa di problemi, sebbene non chiaramente evidenti.

[4] dimostra che smote porta a miglioramenti tangibili solo su classificatori "deboli" come SVM e non su quelli "forti", quelli state-of-the-art come XGBoost, un tipo di classificatore basato sul boosting. Le loro conclusioni sono che, generalmente, non è raccomandato eseguire bilanciamenti come SMOTE ed è meglio preferire direttamente classificatori forti e piuttosto concentrarsi su ottimizare il valore soglia di classificazione.

[3] parla dei metodi di bilanciamento applicati a dei modelli usati per predizioni cliniche e di come abbiano portato ad avere modelli non ben calibrati e non molto utili nel contesto clinico.

Riferimenti Bibliografici

[1] D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3e, Cambridge University Press, https://artint.info/3e/html/ArtInt3e.html

[2] Chawla, N. V. and Bowyer, K. W. and Hall, L. O. and Kegelmeyer, W. P., SMOTE: Synthetic Minority Over-sampling Technique`, Journal of Artificial Intelligence Research, 2002, pages 321–357 http://dx.doi.org/10.1613/jair.953

[3] Ruben van den Goorbergh, Maarten van Smeden, Dirk Timmerman, Ben Van Calster, The harm of class imbalance corrections for risk prediction models: illustration and simulation using logistic regression, *Journal of the American Medical Informatics Association*, Volume 29, Issue 9, September 2022, Pages 1525–1534, https://doi.org/10.1093/jamia/ocac093

[4]Yotam Elor and Hadar Averbuch-Elor, To SMOTE, or not to SMOTE?, 2022, https://arxiv.org/abs/2201.08528