Allenamento di alcuni modelli predittivi al riconoscimento di siti web malevoli

G. Ciampi, A. Corsuto, F. Mogavero

23 aprile 2018

Indice

Outline

Il nostro lavoro si è incentrato sulla formulazione di un modello predittivo di tipo probabilistico, in grado di stimare la presenza di codice malevolo su siti Web.

Si è partiti da un *dataset*[1], rappresentato da una tabella di $n + 1$ colonne. All'interno del dataset, ogni colonna indica uno specifico attributo del sito web in questione; ci si riferisce comunemente agli attributi come *feature*. Le iniziali n colonne rappresentano dati di vario tipo, tra cui informazioni sulla struttura dell'*URL*, la nazione dove è fisicamente situato il server in cui risiede il sito, ed altro. L'ultima colonna, in posizione $n + 1$, contiene invece una informazione booleana che identifica un sito come malevolo o non malevolo. Nel dataset in questione il 12.13% dei siti è identificato come malevolo, mentre il restante 87.87% dei siti è identificato come non malevolo.

L'insieme di dati preso in considerazione contiene informazioni reali su siti internet, scelti in modo da costituire un campione (ragionevolmente) casuale del Web. Questa considerazione è di fondamentale importanza per valutare l'utilità del modello realizzato: non avrebbe senso infatti tentare di costruire un modello statistico su un campione che non rappresenta la realtà. Altre considerazioni fondamentali riguardo il campione di dati sono quelle a proposito della taglia: è molto difficile rappresentare un insieme di dati estremamente grande come quello dei siti Web con un dataset piccolo; questo requisito si scontra però con quello della fattibilità: la gestione di dataset molto grandi è estremamente costosa dal punto di vista computazionale.

La nostra attività è stata divisa in più fasi. Una prima fase di preprocessing iniziale ci ha permesso di eseguire una pulizia dei dati ad alto livello. Successivamente è stato necessario effettuare una seconda fase di preprocessing, con lo scopo di adattare i dati agli algoritmi da utilizzare. Questo si è reso necessario poiché gran parte degli algoritmi di machine learning non accetta input di tipo stringa, oppure richiede che gli attributi numerici siano codificati in modi specifici. In ultimo abbiamo costruito i modelli e ne abbiamo testato le performance.

[1] Disponibile all'indirizzo: https://www.kaggle.com/xwolf12/malicious-and-benign-websites.

Riguardo alle performance, il nostro obiettivo è stato quello di superare in modo significativo l'accuratezza di 'predizioni banali', possibili conoscendo semplicemente la distribuzione dei dati nel dataset. Ad esempio, sapendo che l'87.87% delle istanze è classificato come malevolo, basterebbe predire ogni volta un sito come non malevolo, per ottenere un'accuratezza dell'87.87%.
Per la realizzazione implementativa del progetto è stato utilizzato `Python`, il linguaggio più diffuso nel campo della *data science*; in particolar modo sono state sfruttate le librerie `NumPy`, `Pandas`, `ScikitLearn` e `TensorFlow`, le quali offrono strumenti matematici ed algoritmici per effettuare tutte le principali operazioni di machine learning, compresa la parte di preprocessing.
Infine, invece di sfruttare un ambiente di sviluppo integrato, abbiamo optato per l'utilizzo dell'ambiente Jupyter Notebook, anch'esso fondamentale nel campo della data science, in quanto risulta essere molto più agile rispetto ad un normale *IDE*, permettendo di compilare ed eseguire istruzioni singolarmente.
Riguardo la scelta del dataset, sul Web vi sono numerosi siti che offrono risorse di questo tipo, e numerosissimi sono i dataset stessi, riguardanti gli argomenti più disparati, da poter sfruttare come base per elaborazioni di tipo machine learning. Tra le varie tematiche, tuttavia, la sicurezza informatica presenta un numero di dataset più esiguo rispetto a quelli disponibili relativamente ad altri campi. Ciò potrebbe essere dovuto alla natura intrinseca della disciplina. Inoltre, solitamente, i dataset reperibili in rete relativi alla sicurezza informatica, presentano dati 'semplici' (ovvero dotati di poche *features*), e spesso *già pronti* (ovvero dati che possono essere direttamente forniti agli algoritmi di machine learning, senza bisogno di uno step intermedio di preparazione).
Uno dei siti web più famosi da cui è possibile attingere per ottenere dataset è Kaggle[2], dal quale è stato prelevato il dataset di cui ci occuperemo. La scelta del dataset è stata effettuata in base a vari fattori, tra i quali il numero di features ($24 + 1$, un valore sufficientemente elevato da non richiedere un preprocessing 'banale'), e la presenza di numerose colonne con dati *grezzi*, come stringhe e valori nulli. Questo aspetto è stato considerato poiché la maggioranza degli algoritmi di Machine Learning opera esclusivamente su dati

[2]Link: https://www.kaggle.com/.

numerici, che a volte devono essere addirittura codificati in modo specifico. Ciò ha reso necessaria una laboriosa fase di preprocessing, che ci ha permesso di prendere confidenza con le principali tecniche adoperate in questo settore. Un altro fattore di cui abbiamo tenuto conto, è il fatto che il dataset in questione è molto recente: i dati sono stati collezionati e pubblicati negli ultimi mesi. Oltre a dare un senso ad eventuali applicazioni pratiche di modelli realizzati tramite questi dati, ciò garantisce che essi siano stati collezionati con la consapevolezza delle possibilità offerte oggi dal machine learning.
L'implementazione del progetto è stata suddivisa in diverse fasi: inizialmente sono stati esplorati brevemente i dati, per comprenderne la strutturazione. Successivamente, per ogni colonna che lo richiedesse, è stata effettuata la fase di preprocessing (ovvero di preparazione dei dati agli algoritmi). In ultimo è stata effettuata la fase di analisi *statistico-probabilistica*. Relativamente a quest'ultima fase, inizialmente sono stati allenati[3] due algoritmi di machine learning, uno basato su SupportVector Machines, ed uno basato su Random Forest. Successivamente è stato allenato un classificatore basato su **DeepNeuralNetworks**. In tutti e tre i casi, per l'allenamento e l'impostazione degli iperparametri[4] degli algoritmi ci siamo serviti degli strumenti forniti da ScikitLearn, i quali sono in grado di fornire indicazioni precise circa le performance degli algoritmi relativamente a diverse configurazioni di iperparametri, testando ripetutamente gli algoritmi con settaggi differenti.
A questo punto gli algoritmi sono stati allenati con gli iperparametri ritenuti migliori[5] e ne sono state valutate le performance.
Dopo aver costruito i modelli con i parametri ritenuti migliori, ed averne analizzate le performance, si è notato che sui dati di test, ovvero dati sconosciuti agli algoritmi, tramite i modelli basati su SupportVectorMachines e DeepNeuralNetwork si raggiunge un'accuratezza maggiore del 97%. Nel caso

[3]Formalmente questa fase è chiamata *training phase* in inglese.

[4]Per iperparametro in questo caso si intende un fattore che permette di decidere la strategia di allenamento del modello. Essi sono divisi in numerose categorie; i più comuni sono quelli di 'Regolarizzazione', che impattano sulla complessità del modello

[5]Queste informazioni sono a volte di natura probabilistica, poiché alcuni modelli, come quelli basati su Random Forest, effettuano durante l'allenamento delle scelte casuali. Questo conduce a lievi differenze nelle varie esecuzioni dei test, anche se questi ultimi vengono eseguiti sugli stessi dati e con gli stessi iperparametri

del modello basato su RandomForest, sempre sui dati di test, l'accuratezza si aggira intorno al 95%.
In tutti e tre i casi si è comunque raggiunto l'obiettivo prefissato si superare in modo significativo l'accuratezza della "previsione banale", ovvero l'87.87%.

1 Alcune Nozioni Preliminari

Ai fini di rendere la lettura più agevole, evitando continue digressioni, presentiamo qui i concetti fondamentali della disciplina che saranno necessari per muoversi nelle parti successive del testo

1.1 Cos'è il Machine Learning

Il Machine Learning è una branca della conoscenza che giace nell'intersezione di diverse aree del sapere, tra le quali principalmente informatica e statistica. La necessità di una formazione così peculiare ed al tempo stesso eterogenea, fa sì che, tendenzialmente, vi sia una scarsità di esperti del settore. Si dice infatti che un professionista in questo campo debba essere: "più competente in informatica rispetto ai matematici e più competente in statistica rispetto agli informatici".
Una definizione più formale invece è stata fornita da *Arthur Samuel*, il quale afferma: "Machine Learning is the field of study that gives computer the ability to learn without being explicitly programmed".

Definizione fondamentale. In sostanza, il machine learning si compone di tutti quegli strumenti che permettono, in modo automatico o semiautomatico, di generare conoscenza a partire dai dati. Quest'ultima può essere utilizzata dalla macchina per effettuare predizioni o prendere decisioni.

Gli strumenti del Machine Learning permettono alle macchine di gestire situazioni altrimenti intrattabili, quali il riconoscimento del linguaggio umano.

Esempio 1.1. Un esempio chiave è quello del riconoscimento di cifre numeriche: supponiamo di voler costruire un software che distingua delle cifre disegnate su carta.[6] Ognuna di esse può essere rappresentata (ovvero scritta) in un numero pressoché infinito di modi, che variano sia in base alla calligrafia dell'utente, sia in base a fattori casuali, quali ad esempio lo stress, lo

[6]Un dataset di questo tipo è estremamente popolare ed è reperibile sotto il nome di *MNIST*.

strumento utilizzato o la posizione. È immediato notare l'impossibilità della realizzazione di un software del genere con le tecniche tradizionali: ciò richiederebbe di trattare (in modo esplicito) infiniti casi possibili. Gli algoritmi di Machine Learning permettono di risolvere situazioni del genere identificando delle similitudini tra i vari membri di ogni classe (ovvero tra più raffigurazioni di una stessa cifra).

Grazie alle nuove possibilità offerte per trattare problemi altrimenti irrisolvibili, il machine learning ha avuto una diffusione estremamente ampia e veloce che lo ha condotto ad essere, ad oggi, uno dei campi di studio principali e maggiormente innovativi di tutto il settore scientifico.

1.2 Apprendimento Supervisionato vs non Supervisionato

Una prima divisione fondamentale nel machine learning è quella tra apprendimento supervisionato e non supervisionato. Per apprendimento supervisionato si intende l'apprendimento effettuato fornendo all'algoritmo, insieme agli altri dati, il 'risultato'. Nel caso dell'*Esempio 1.1* (il riconoscimento di cifre), l'apprendimento supervisionato prevederebbe la costruzione di un modello tramite istanze composte sia da una rappresentazione digitale del disegno, sia da una "etichetta", digitale e non ambigua, che ne indica il contenuto. Al termine della fase di learning, l'algoritmo sarà in grado, data in input la rappresentazione digitale del disegno di una cifra decimale, di fornire una predizione più o meno accurata rispetto a quale sia la cifra rappresentata.
L'opposto dell'apprendimento supervisionato è duello non supervisionato. Esso consiste nel costruire un modello a partire da dati per cui non è fornito un 'risultato'. Nell'esempio precedente, si può immaginare di allenare un modello su una serie di immagini, senza fornire all'algoritmo alcuna associazione immagine-valore. Quanto detto potrebbe sembrare contraddittorio: come si può costruire un modello, senza sapere esattamente cosa bisogna modellare? In realtà questo tipo di apprendimento è utilizzato con scopi diversi rispetto al primo, come la clusterizzazione, o l'identificazione di pattern (astratti) all'interno di immagini. In casi del genere, si cerca di individuare somiglianze e

differenze all'interno dei dati, e non il fare una associazione ad un certo valore prefissato.

1.3 Apprendimento Batch ed Apprendimento Online

Altra suddivisione fondamentale nel machine learning, è quella tra apprendimento *batch* ed apprendimento *online*. L'apprendimento online consiste in un 'training infinito', che permette al modello di apprendere ogniqualvolta si presentano nuovi dati. Nella pratica, prima si allena un modello che raggiunge un'accuratezza ritenuta sufficiente; successivamente questo modello è reso disponibile agli utenti finali, ed esso continua ad apprendere anche dalle richieste di questi ultimi. Riferendosi all' *Esempio 1.1*, un modello supervisionato, dopo aver fornito all'utente una previsione, il modello potrebbe ricevere dall'utente un feedback (ad esempio il valore corretto), tramite il quale eseguire una nuova iterazione dell'apprendimento.
Nell'apprendimento batch, invece, vi è una suddivisione netta tra la fase di training e quella operativa. In questi casi i modelli sono prima allenati e poi resi disponibili per l'utilizzo. In questi casi i modelli non sono più in grado di apprendere dopo la distribuzione.

1.4 Note sulla Preparazione dei Dati

La preparazione dei dati è un aspetto fondamentale del machine learning. Quasi sempre i dati devono essere trattati e codificati in maniera opportuna, prima di essere forniti al modello: lievi variazioni nella codifica adottata possono determinare variazioni significative nelle performance dello stesso modello.
La maggioranza degli algoritmi di machine learning riesce a trattare solo dati di tipo numerico; alcuni, addirittura, pongono dei vincoli anche sul modo in cui questi valori debbano essere distribuiti[7].
La codifica di valori che non siano numerici per natura, come stringhe, o *strettamente* numerici, come indirizzi IP, richiede la considerazione di fattori tutt'altro che banali.

[7] I modelli basati su *SVM*, introdotti nella sezione 3.2, richiedono che gli input numerici siano centrati sul valore medio.

La questione fondamentale riguardante le codifiche dell'input sta nel fatto che gli algoritmi di machine learning tendono per loro natura a considerare dati rappresentati da numeri 'vicini' come maggiormente correlati.

Esempio 1.2. Supponiamo di avere un attributo che identifica delle distanze, rappresentato da stringhe del tipo: $Vicino$, $Lontano$, $MoltoLontano$. Un meccanismo automatico di codifica, che non conosce l'italiano, potrebbe fare un'associazione da stringa ad intero, del tipo: $Vicino = 1$, $Lontano = 3$, $MoltoLontano = 2$. Poiché, per molti modelli, valori numericamente vicini sono trattati come maggiormente simili, un certo modello potrebbe trattare la stringa $Vicino$ come più simile alla stringa $MoltoLontano$, e meno simile rispetto alla stringa $Lontano$.

Errori di questo tipo possono avere un impatto estremamente significativo sull'allenamento dei modelli predittivi, impedendo a questi ultimi di cogliere associazioni fondamentali tra i dati; conducendo a risultati dalla scarsa accuratezza.

1.5 Overfitting ed Underfitting

Concetti fondamentali riguardanti la costruzione dei modelli sono quelli di overfitting ed underfitting. Durante l'allenamento, il modello identifica dei pattern presenti nei dati, che saranno utilizzati per fare delle previsioni sui dati nuovi. Purtroppo però, i dati forniti in input all'algoritmo sono, nella stragrande maggioranza dei casi, solo un campione minimale rispetto a tutti i dati esistenti riguardo al problema in questione.

Obiettivo. A causa di ciò, partendo dall'assunzione fondamentale che il dataset scelto rappresenti proporzionalmente la totalità dei dati esistenti, si vuole che il modello sia in grado di *generalizzare* rispetto ai dati presentati, astraendo concetti 'globali', in modo da *performare* bene nelle predizioni su istanze nuove.

Durante l'apprendimento, gli algoritmi cercano di modellare i dati minimizzando una funzione di errore che misura la distanza tra le predizioni ed i

valori reali; funzioni di questo tipo sono dette comunemente *loss function*. Per ottenere un modello che generalizzi bene rispetto a dati nuovi, esso deve essere costruito senza minimizzare la loss function né troppo, né troppo poco. Quando il modello minimizza troppo la loss function, esso apprende *troppo* dai dati di training, finendo per non essere in grado di generalizzare bene sui dati nuovi: questa condizione è nota come overfitting. "*Il modello potrebbe adattarsi a caratteristiche che sono specifiche solo del training set, ma che non hanno riscontro nel resto dei casi; perciò, in presenza di overfitting, le prestazioni (cioè la capacità di adattamento/previsione) sui dati di allenamento aumenteranno, mentre le prestazioni sui dati non visionati saranno peggiori.*"[8]

L'underfitting è invece la condizione opposta, in cui il modello non ha appreso *abbastanza* dai dati ed ottiene perciò performance scarse.

Un esempio chiave è quello dell'approssimazione di una curva a partire da alcuni punti campionati da essa. La *Figura 1.1* mostra, in blu, le curve che rappresentano i modelli ottenuti approssimando dei punti campionati dal grafico di una funzione, nei quali è stato applicato un leggero rumore casuale (di tipo gaussiano).

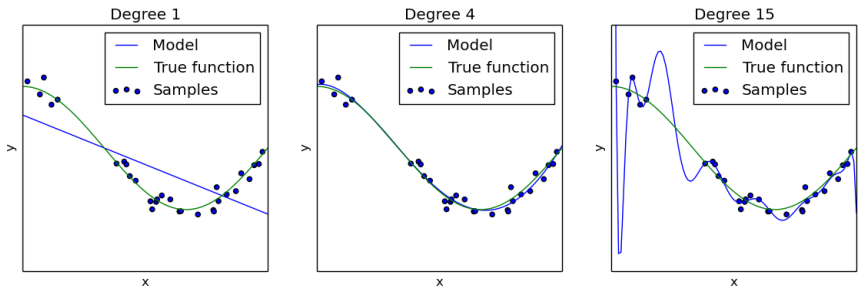


Figura 1.1: Overfitting ed Underfitting. Le figure mostrano l'approssimazione di una curva tramite tre polinomi, rispettivamente di grado 1, 4 e 15.

Il rumore presente sui dati può essere visto come una rappresentazione della casualità degli eventi del mondo reale, i quali non sono sempre deterministici. Matematicamente, nell'esempio, l'obiettivo è la costruzione di una funzione modello (la curva in blu) che, dato un insieme di punti, riesca ad approssimare in maniera accettabile la funzione originale (la curva in verde). Si cerca, dunque, mediante l'apprendimento sui dati di testing, di formare un modello

[8]Descrizione su Wikipedia: https://it.wikipedia.org/wiki/Overfitting.

che sia quanto più possibile valido nel caso generale.

Come si evince dalla *Figura 1.1*, il primo modello del grafico consiste in un polinomio di grado 1 (una retta). A causa della sua scarsa complessità (il grado troppo basso) esso non riesce ad approssimare n un modo sufficientemente accurato i punti, neppure quelli ricevuti per l'allenamento (caso *underfitting*). Il terzo modello è troppo complesso rispetto ai dati. Esso porta quasi a zero la distanza tra ogni punto usato per l'apprendimento e la curva di approssimazione, minimizzando eccessivamente la loss function; in pratica l'algoritmo si è adattato completamente ai valori dei dati studiati nel training senza fare le dovute astrazioni. Questo modello, è fortemente condizionato dal rumore gaussiano e finisce per identificare dei pattern anche all'interno del rumore presente nei dati, il che è ovviamente fuorviante.

Esempio 1.3. Per chiarire ulteriormente, forniamo un esempio che tratta la questione ad alto livello. Si consideri il problema di volere prevedere il tempo che un'auto da corsa impiegherà per compiere un singolo giro, in un circuito fissato; tentando di risolverlo tramite il machine learning. Si supponga di avere un dataset contenente le informazioni che riguardano un migliaio di giri effettuati, dalla medesima auto, in varie edizioni della stessa gara nello stesso circuito, parametrizzati tramite attributi quali le condizioni atmosferiche, gli pneumatici utilizzati ed altro.
Si supponga di allenare il modello tramite 10 righe presenti nell'insieme di partenza, i cui valori per ciascun attributo sono identici alla tupla da esaminare. Per 9 di esse risulta sempre lo stesso crono di 1:27:xxx min. Nel decimo, per via di un'avaria (informazione non presente tra gli attributi), l'auto ha impiegato 2:35:yyy min. Un algoritmo che presenta overfitting, considera in modo significativo anche il valore legato all'avaria al motore, prevedendo che in tali condizioni la velocità media dell'auto in un singolo giro possa essere, erroneamente, $\frac{1{:}27{:}xxx \times 9 + 2{:}35{:}yyy}{10}$: in tal modo la componente casuale gioca un ruolo predominante nella predizione.

Tornando alla nostra figura, un polinomio di grado 4, raffigurato al centro, riesce ad approssimare molto bene la funzione di partenza; senza presentare nessuna delle due problematiche presentate.

Purtroppo non è possibile stabilire in modo rigoroso quale sia la complessità adeguata affinché il modello colga solo i pattern che si ritengono giusti; sebbene sia enorme la quantità di tecniche studiate appositamente per la *regolarizzazione*: ma forse è proprio questo il senso dell'esistenza del machine learning.

1.6 Testing e Validazione

Dopo aver costruito dei modelli, è necessario valutarne le performance. A questo scopo servono le fasi di testing e validazione. Nel caso dell'apprendimento supervisionato ciò può avvenire tramite un semplice rapporto tra successi e numero di tuple testate. Per far ciò si fa uso di un altro sottoinsieme di dati del dataset, disgiunto e complementare rispetto a quello usato nell'apprendimento (la loro unione da il dataset, la loro intersezione è vuota). Anche questa fase può presentare insidie per nulla banali. Il fattore principale da tenere a mente durante questo stadio è che i dati a disposizione sono limitati. In situazioni standard i dati vengono suddivisi in due parti: un train_set, sul quale eseguire l'allenamento, ed un test_set, su cui eseguire il testing. Da qui in avanti ci riferiremo a questi due insiemi di dati come `train_set` e `test_set`, per indicare le istanze specifiche riferite ai nostri dati. Il problema fondamentale legato a questo approccio consiste nel fatto che, una volta allenato il modello, qualora le sue performance sul test_set risultino scarse, non si hanno dati nuovi su cui testarlo nuovamente.
Qualora dopo il test, infatti, le performance dovessero risultare scarse, saremmo tentati di apportare modifiche al modello già costruito in modo da migliorarle: ad esempio modificando i valori degli iperparametri degli algoritmi, fino a raggiungere delle performance ritenute buone anche sul test_set. Tuttavia, come illustrato nell'*Esempio 1.4*, ciò può causare problemi di overfitting, poiché un approccio di questo tipo porta ad un modello plasmato per comportarsi al meglio solo su un determinato dataset.

Esempio 1.4. Supponendo di avere un algoritmo di generazione del modello che prenda in input un solo iperparametro. Si consideri una suddivisione del dataset in train_set e test_set.

Con lo scopo di ricercare il miglior iperparametro che minimizzi il margine di errore sul test_set, si effettuino n esecuzioni differenti dell'allenamento, ognuna con valori diversi degli iperparametri, su una coppia di train_set e test_set definita in partenza. Per ognuna di esse, si valutano le performance di successo e di insuccesso della fase di testing: si supponga che il modello dalle performance migliori sia reso disponibile nel mondo reale. Le prestazioni potrebbero essere molto diverse dalle aspettative. Questo perché l'istanza dell'algoritmo in questione non è quella che generalizza meglio sui dati, ma quella che si comporta meglio nel caso specifico del nostro dataset. Dunque l'istanza in questione garantisce le migliori prestazioni su quel test_set, ma nulla garantisce che ciò corrisponda alle migliori prestazioni nel caso generale, anzi, è elevata la probabilità di aver costruito un modello troppo specifico per quel test_set.

Per evitare situazioni di questo tipo, sono state ideate delle tecniche specifiche. La più utilizzata, consiste in uno split ulteriore dei dati in train_set, validation_set e test_set. Tramite questa metodologia, l'algoritmo viene prima allenato sul train_set, per poi essere testato sul validation_set. A questo punto si possono apportare al modello tutte le modifiche ritenute necessarie testandolo solo sul validation_set.
Infine, *quando si è sicuri che non si possa fare di meglio*, ne vengono valutate le effettive performance sul test_set, sconosciuto all'algoritmo, considerato un campione *rappresentante dei dati provenienti dal mondo esterno.* Giunti sin qui, però, i giochi sono fatti, e non va commesso lo stesso errore della situazione precedente: se il modello ha performance scarse, va semplicemente eliminato, poiché, con i dati a disposizione, non sarebbe possibile testarlo nuovamente.
Allenare nuovamente, modificando gli iperparametri nell'ottica di migliorare le performance sul test set, condurrebbe come nel caso precedente ad un modello 'plasmato' su di uno specifico dataset.
Una tecnica leggermente più sofisticata è quella della cross-validation. Essa consiste in uno split iniziale del dataset in train_set - test_set; successivamente, il train_set viene splittato ulteriormente in k sottoinsiemi. Le fasi di allenamento e testing vengono eseguite esattamente k volte nel modo seguente: ad ogni passo si scelgono $k-1$ sottoinsiemi, e si allena il modello

su questi. Successivamente si testa il modello così allenato sul k-esimo sottoinsieme (che funge da validation_set), valutandone le performance. Per una misura complessiva dell'accuratezza viene effettuata una media dell'accuratezza di ognuno dei k modelli, che, com'è evidente, sono ognuno indipendente dall'altro. Quanto descritto è schematizzato in *Figura 1.2*.

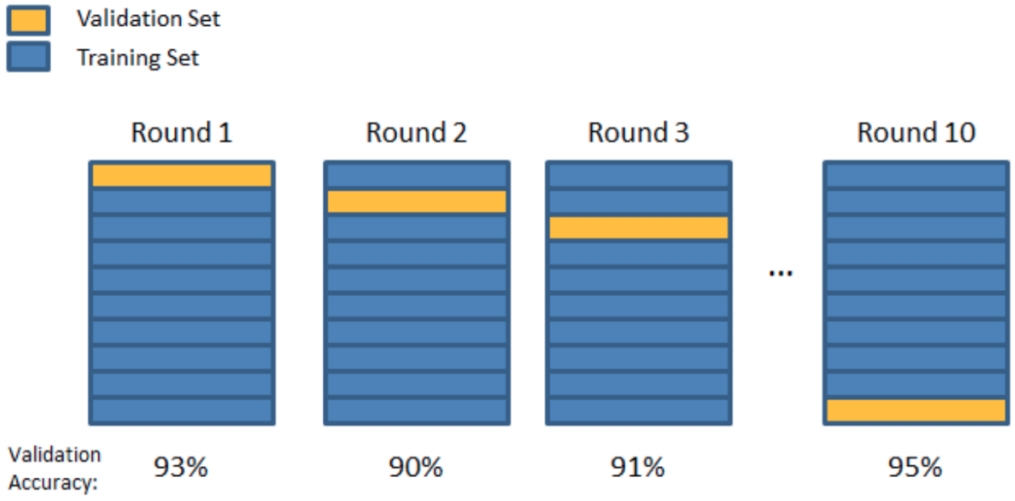


Figura 1.2: Schema che illustra la tecnica della cross-validation.

Anche in questo caso, dopo aver raggiunto quelle che si ritengono essere le performance migliori possibili, bisogna testare i dati sul test_set facendo attenzione a non fare modifiche successive.

1.7 Le Librerie Fondamentali

Python è certamente il linguaggio più usato nel mondo della data-science, per rendersene conto è sufficiente ricercare i libri di testo che la riguardano: eccetto una buona fetta di libri che riguardano il linguaggio R, tutti gli altri sono basati su Python. La motivazione consiste nel fatto che Python mette a disposizione dell'utente una serie di librerie ove tutti gli algoritmi principali necessari per le varie fasi sono già implementati: ciò permette all'utente di focalizzarsi prevalentemente su processi ad alto livello, piuttosto che curare anche gli aspetti legati all'implementazione a basso livello. Sebbene anche altri linguaggi forniscano delle librerie di questo tipo, nulla può competere con la completezza di quello che è un vero e proprio ecosistema, fornito da Python.

Le librerie che sono state utilizzate (che sono le principali) si possono dividere, grossolanamente, in tre gruppi: il primo è composto dalla libreria NumPy, che permette di manipolare agevolmente vettori ad n-dimensioni, e dalla libreria Pandas, che fornisce delle strutture dati estremamente comode (tra cui la struttura DataFrame, ormai fondamentale) ed alcuni strumenti per l'analisi. Queste librerie sono entrambe contenute nell'ecosistema SciPy, che fornisce anche altri utilissimi strumenti, come la libreria Matplotlib, la quale permette di creare e manipolare grafici in modo molto diretto e semplice. Questo primo insieme è formato da librerie che essenzialmente forniscono strumenti che servono a manipolare i dati.
Il secondo gruppo è formato dalla libreria ScikitLearn: questa fornisce una implementazione di moltissimi algoritmi di machine learning, oltre a quella di numerose classi ideate per il preprocessing dei dati.
Il terzo ed ultimo gruppo è composto esclusivamente dalla libreria TensorFlow: questa potentissima libreria fornisce tutte le classi e i metodi necessari per creare e manipolare reti neurali.

2 Preparazione dei Dati

2.1 Esplorazione

Una volta scaricato il dataset dalla pagina Kaggle, per esplorarlo va importato all'interno dell'ambiente Jupyter Notebook, che è stato preferito rispetto ad un *IDE* per via della sua maggiore *agilità*, in quanto permette di compilare ed eseguire ogni istruzione singolarmente. Poiché i dati del set sono in formato .csv (comma-separated values, estensione estremamente comune nella data science), è facile importarli in Jupyter Notebook tramite la funzione `load_csv()`. Quest'ultima inserisce i dati in una struttura tabellare detta DataFrame, definita nella libreria Pandas. Terminato l'*import* dati, è buona pratica escluderne subito una parte, da destinare al testing degli algoritmi; i rimanenti saranno dedicati alla fase di training.
Per effettuare questa separazione è stata adoperata la classe `StratifiedShuffleSplit`, messa a disposizione da Python. Essa effettua uno split delle righe di tutto il dataset in due sottoinsiemi disgiunti: un `train_set` per l'allenamento ed un `test_set` per il testing, le quali **composizioni sono casuali**. Inoltre, assieme al permettere di scegliere le dimensioni del `test_set` (nel nostro caso il 20% dell'insieme di partenza), questa classe consente di effettuare lo split mantenendo inalterate le frequenze di distribuzione di determinati valori rispetto alla loro distribuzione nel dataset iniziale. Nel nostro caso, è sembrato opportuno effettuare lo split lasciando inalterata la distribuzione di siti web malevoli (il dato su cui effettuare poi le previsioni, circa il 12.13% sul totale). In tal modo nei due sottoinsiemi, la distribuzione di valori rispettivamente all'ultima colonna è identica all'originale.
Il nostro dataset presenta 24 features, tra cui lunghezza dell'URL, il numero di caratteri speciali all'interno della pagina, paese e regione di appartenenza, lunghezza del contenuto in bytes, etc. .
Sin da subito è possibile notare la presenza di molti valori nulli in varie colonne, codificati in vario modo (`None`, `NaN`, `'None'`), sui quali è necessario intervenire in qualche modo, al fine di renderli omogenei e trattabili dagli algoritmi. Inoltre, molti degli attributi, come il charset, la data di registrazione, il paese in cui il sito è registrato, sono codificati come stringhe: considerando che molti algoritmi di machine learning accettano in input solo valori nu-

merici, bisognerà trovare un modo coerente per trasformarli e trattarli come numeri.

2.2 Le classi Transformer, Pipeline e FeatureUnion

Per il preprocessing, ScikitLearn mette a disposizione tre meccanismi fondamentali: Transformer, Pipeline e FeatureUnion. Il primo, Transformer, è un "paradigma". Esso consente di apportare trasformazioni di vario genere sui dati, modificandoli nella morfologia ma non nella semantica, non alterando dunque il loro significato logico. Oggetti che rispettano questo paradigma sono caratterizzati principalmente da due metodi: `fit()` e `transform()`. Il primo, `fit()`, serve a definire con precisione la strategia e la struttura con le quali le trasformazioni devono essere effettuate; mentre `transform()` le mette in pratica sui dati, modificandoli. Nell'ottica del rispetto di questo paradigma, una classe transformer deve estendere le classi `BaseEstimator` e `TransformerMixin`: tramite ciò si ottiene un terzo metodo, `fit_transform()`, il quale invoca sequenzialmente `fit()` e `transform()` sui dati passati in input.

Esempio 2.1. Per essere più chiari, si supponga di avere un vettore con dati di tipo stringa, che si vuole codificare come numeri interi. Innanzitutto, bisogna chiamare il metodo `fit()`: esso effettuerà una mappatura che associa ogni stringa contenuta nel vettore di partenza ad un numero, come una sorta di dizionario. Successivamente, il metodo `transform()` utilizzerà la mappatura costruita dal metodo precedente per costruire un vettore codificato, da restituire in output. Questo meccanismo fa in modo che `fit()` sia invocato una sola volta, in modo da definire una codifica (mappatura) immutabile; a questo punto tutte le chiamate di `transform()` saranno *consistenti* tra di loro.

Durante la fase di preparazione dei dati, i transformer possono essere visti come dei blocchi fondamentali: ognuno di essi lavora su piccoli gruppi di attributi separatamente. Le classi Pipeline e FeatureUnion, possono essere considerate invece come 'collanti' tra i vari transformer. La prima, all'atto della costruzione, prende in input una sequenza di transformer ordinati. Dopo aver definito una Pipeline, è possibile invocare i suoi metodi `fit()`,

`transform()` e `fit_transform()` come se si trattasse di un singolo transformer: all'atto di questa chiamata, la Pipeline invocherà lo stesso metodo su tutti i transformer, in modo sequenziale, trasmettendo in input ad uno l'output del precedente.
L'uso di questa struttura risulta conveniente per effettuare sequenze di operazioni ordinate sullo stesso sottoinsieme di colonne.
La classe FeatureUnion, invece, è utile per 'concatenare' gli output di transformer e Pipeline differenti. Essa, all'atto della costruzione, riceve in input un elenco *non* ordinato di Pipeline e transformer: li esegue in parallelo, e restituisce una matrice contenente tutti gli outputconcatenati tra di loro in modo orizzontale.
L'uso di quest'ultima classe risulta perché, consentendo di parallelizzare il calcolo tra Pipeline operanti su colonne differenti, permette di migliorare significativamente l'efficienza.

2.3 Una preparazione Iniziale

La preparazione dei dati è stata articolata in due fasi: una fase di preparazione iniziale, ed una di preparazione secondaria. Durante la prima fase, sono state eseguite solo delle operazioni che riguardano la struttura generale dei dati, come l'eliminazione di alcune feature ritenute superflue rispetto agli obiettivi prefissati.
Come specificato in precedenza, i dati sono contenuti inizialmente in una struttura detta DataFrame (definita nella libreria Pandas); ciò rappresenta un primo problema di compatibilità, perché i metodi delle classi transformer (già definite in ScikitLearn) accettano in input oggetti di tipo `numpy.array`. A questo scopo è stata realizzata la classe `DataFrameSelector`: un transformer il cui metodo transform, prendein input un DataFrame, e restituisce i dati in esso contenuti sotto forma di array.
In secondo luogo, è stata realizzata la classe, `ColumnSelector`, utile a selezionare solo alcune delle colonne dell'array bidimensionale output di `DataFrameSelector`, con lo scopo di eliminare le altre. Infatti, le feature *Reg_date*, riguardante la data di creazione del sito, *Update_date* indicante la data dell'ultimo aggiornamento e *URL* sono sembrate inutili e pertanto sono state

immediatamente escluse. In particolare, riguardo quest'ultima feature, siccome ogni sito web presenta un URL diverso, esso non dà (secondo noi) la possibilità di individuare correlazioni con il contenuto (malevolo-non malevolo) del sito.

Per rendere la questione più chiara, la consideriamo da un altro punto di vista. Cercando di formalizzare:

ASSUNZIONE. Sia *www.example.com* un URL. Si può assumere ogni URL come *totalmente* scorrelato da tutti gli altri (non ha alcuna relazione con *www.example2.com*): ogni URL corrisponde ad una struttura a sé stante e non ha nulla a che vedere con qualunque altro sito.

A questo punto, vi sono solo due possibili situazioni: o l'URL della riga da esaminare è presente nel dataset, ed in questo caso si conosce già una risposta, oppure, l'URL non è presente nel dataset, e quindi non è possibile fare assunzioni.
Alla fine della prima fase del preprocessing, è stata utilizzata di nuovo la classe `ColumnSelector`, con lo scopo di eliminare la colonna contenente le etichette (sito malevolo - non malevolo); infatti questa particolare colonna deve essere fornita a parte agli algoritmi.
Per aggregare tutte le trasformazioni della prima fase è stata implementata una Pipeline, avente l'aspetto illustrato in *Figura 2.1*.

```
In [211]: prePipeline = Pipeline([
              ('dataFrameSelector', DataFrameToArray()),
              ('column_selector', Column_selector( [...] ))
          ])
```

Figura 2.1: La Pipeline per la preparazione iniziale.

2.4 Praparazione dei Dati agli Algoritmi

La seconda fase di preparazione dei dati, è stata la più lunga e laboriosa. In questa fase sono state trattate le varie feature in modo specifico, definendo i metodi per trattare i valori nulli, insieme con quelli per trattare le feature non-numeriche, nell'ottica di fornire un input compatibile con gli algoritmi.
La strategia di lavoro per trattare le feature, ovvero le 'colonne' della nostra

matrice bidimensionale, è stata la costruzione di varie Pipeline, una per ogni colonna o gruppo di colonne, raggruppando poi tutti gli output tramite una FeatureUnion, che ha consentito di parallelizzare il lavoro.
Requisito fondamentale per l'utilizzo di Feature Union è che tutte le colonne in input abbiano lo stesso numero di righe. Nonostante possa sembrare ovvio che le colonne soddisfino tale requisito, questo non è scontato a causa della presenza di molti valori nulli. Perciò, bisogna prestare particolare attenzione alle strategie adottate per la loro gestione.

2.5 Le Features *URL_Length* e *Number_Special_Characters*

In questa e nelle prossime sezioni illustreremo le caratteristiche delle varie feature, insieme con le strategie adottate per il preprocessing. Cominceremo con il trattamento delle feature URL_Length e Number_Special_Characters: esse rappresentano rispettivamente la lunghezza dell'URL ed il numero di caratteri speciali contenuti nel payload della HTTP Response. Questi attributi sono già codificati, di per sé, come numeri, quindi non vi è bisogno di trattarli in modo particolare. Tuttavia, le due feature presentano molti valori nulli: per eliminarli, è stata realizzata la classe transformer `NumberCheckNan`, la quale inserisce il carattere 0 nei campi in cui vi sono valori nulli. Inoltre, poiché questo in alcuni casi fornisce performance migliori (o tempi di training più brevi) abbiamo scalato tutti i valori riconducendoli all'intervallo $[-1, 1]$. Questa operazione è stata effettuata utilizzando la classe `StandardScaler`, contenuta nella libreria ScikitLearn. La Pipeline risultante, per queste prime due colonne, è la seguente:

```
In [222]: select_and_scale1 = Pipeline([
              ("column_selector", Column_selector([0,1])),
              ("checkNumberNan", NumberCheckNan()),
              ('std_scaler', StandardScaler())
          ])
```

Figura 2.2: La Pipeline per la trasformazione delle prime colonne.

Ricapitolando, la Pipeline prende in input una matrice che contiene tutte le feature, che viene passata in input al metodo `fit_transform()` del primo transformer, ovvero `Column_selector` . A questo punto, `Column_selector`

crea una nuova matrice che contiene solo le colonne con indice 0 ed 1, e la restituisce in output. La matrice è passata in input a `Number_check_nan`, che sostituisce con uno zero i valori nulli. Quest'ultimo output diventa input della classe `StandardScaler`, che scala i valori in modo che essi siano compresi tra -1 ed 1. L'output di quest'ultimo transformer costituisce l'output dell'intera Pipeline.

2.6 La Feature *Charset*

La Pipeline successiva si occupa del trattamento della feature 'charset'. I valori di questo attributo sono codificati come stringhe, in più molte celle contengono valori nulli. Innanzitutto, è stato ritenuto opportuno trattare i valori nulli come se costituissero un valore a parte, in parole povere, come se si trattasse di un particolare charset. Questo perché è stato valutato che questa potesse essere una informazione rilevante ai fini della comprensione della natura del sito.
Per codificare in modo numerico l'informazione contenuta in questo attributo, è stata adoperata la classe `LabelBinarizer`, contenuta in ScikitLearn. Il metodo `fit()` di questa classe prende in input una matrice, le cui entry sono codificate come stringhe, e ne crea una nuova, dove, per ogni colonna in input, vi sono n colonne in output, dove n è il numero di possibili entry *differenti* all'interno della colonna di partenza.
In altre parole, il metodo `fit()` di LabelBinarizer, tratta singolarmente ogni colonna della matrice in input (ottenuta tramite `Column_selector` nella Pipeline). Per ogni colonna sono estratti tutti i valori univoci, successivamente, per ogni elemento dell'insieme così ottenuto, viene creata una nnuova colonna inizialmente vuota: se n è la cardinalità dell'insieme delle stringhe distinte, n sarà il numero di colonne create. Tale operazione viene ripetuta per tutte le colonne in input a LabelBinarizer. In totale dunque, si avranno tante nuove colonne quanti sono i valori possibili per ogni colonna in input.
Inizialmente, tramite il metodo `fit()`, viene definita una relazione di tipo *one-to-one* che associa ogni stringa distinta di ogni colonna in input ad una nuova colonna. Successivamente, tramite l'invocazione del metodo

`transform()`, per ogni riga della matrice in input, viene creata una riga della matrice in output, dove, per ogni cella, il valore è settato ad uno se la colonna in questione corrisponde al valore contenuto nella cella della matrice in input; il valore è settato a zero altrimenti.
Per facilitare la comprensione di questo meccanismo, forniamo un esempio:

Esempio 2.2. Si supponga di avere un vettore colonna, dove ogni valore può essere soltanto la stringa a oppure la stringa b. Il vettore avrà un aspetto del genere: $w = [a, b, b, a, ..., b]^T$. Passandolo in input al metodo `fit_transform()` di label binarizer, si otterrà una matrice con due colonne ed n righe, codificata nel modo seguente: $o = [[1, 0], [0, 1], [0, 1], [1, 0], ..., [0, 1]]$. Come si può vedere, il valore a è stato codificato come $[1, 0]$, mentre il valore b è stato codificato come $[0, 1]$.

Questo tipo di codifica, detta comunemente *one-hot-encoding*, può causare un notevole incremento nel numero di colonne, qualora vi siano molti valori diversi tra di loro. Una alternativa a questa codifica, è fornita dalla classe `LabelEncoder`. Essa crea, invece di una associazione ad un 'vettore binario', una associazione con un numero intero. Nel caso precedente, avrebbe associato a con il valore zero, b con il valore uno, e se vi fosse stato un valore c lo avrebbe associato a due. Questo tipo di codifica è conveniente poiché può essere rappresentata tramite un vettore ad una dimensione, invece di una matrice.
Pur avendo dei vantaggi dal punto di vista computazionale, una codifica del genere non è l'ideale per il nostro caso, perché molti algoritmi di machine learning tendono a vedere valori numerici vicini come 'più simili'. Nel caso specifico, l'algoritmo avrebbe potuto identificare il valore c come più vicino a b piuttosto che ad a, ma questo nel caso nomi di charset non avrebbe assolutamente senso. Una codifica di tipo *one-hot* ci assicura invece che ogni possibile valore sia visto come indipendente rispetto agli altri.
La Pipeline che si occupa del charset è stata quindi strutturata nel modo seguente:

```
In [736]: charsetPipeline = Pipeline([
              ('Column_selector', Column_selector(col_indexes=[2])),
              ('checkNan', CheckNan()),
              ('stringUpper', StringUpper()),
              ('columnLabelBinarizer',ColumnLabelBinarizer())
          ])
```

Figura 2.3: La Pipeline per la trasformazione della feature Charset.

Due note finali riguardanti questa struttura: la classe `check_nan` si occupa di rimuovere i valori nulli da colonne di tipo stringa (sostituendoli con un carattere speciale). La classe `string_upper` è utilizzata per fare l'uppercase di tutte le entry, siccome a volte lo stesso valore è presente sia in uppercase, sia in lowercase. La classe `column_LabelBinarizer`, invece, è semplicemente una classe *wrapper* di `LabelBinarizer`, utilizzata per risolvere alcuni problemi di compatibilità sorti dall'introduzione della versione 0.19 di ScikitLearn.

2.7 La Feature *Server*

Un discorso leggermente diverso è stato fatto riguardo il trattamento della feature *Server*. I valori di questa colonna sono tutti codificati come stringhe, inoltre, molte delle stringhe sono codificate secondo il pattern `server.version`. Anche in questo caso, dato che non sarebbe stato opportuno utilizzare una codifica che permettesse di 'correlare' valori diversi, è stata sfruttata la codifica fornita da `LabelBinarizer`. Oltre questo, siccome, come già accennato, ad una prima esplorazione dei dati è risultato che fossero molto popolari server di tipo Apache (seppure di versioni differenti), è stato deciso di introdurre una nuova colonna, sempre binaria, che contenesse 1 qualora la stringa di partenza contenesse la sottostringa 'Apache', 0 altrimenti, nell'ottica di valutare eventuali problematiche relative al server Apache che siano *cross-versioning*. Per effettuare queste trasformazioni sono state implementate due differenti pipeline: una per il label binarizer, identica a quella descritta nel precedente paragrafo, l'altra per aggiungere la nuova colonna. Per rendere più efficiente il tutto, però, è stato fatto affidamento alla classe `FeatureUnion`, fornita dalla libreria ScikitLearn e presentata all'inizio di questa sezione. La FeatureUnion realizzata relativamente alla colonna Server prende in input le due pipeline, le esegue in parallelo, e restituisce una matrice dove le colonne output di una sono concatenate a quella in output dell'altra. La classe, così definita, ha

l'aspetto seguente:

```
In [739]: server_feature_union = FeatureUnion([
              ("server_pipeline_1", Server_Apache_Pipeline),
              ("Server_Pipeline", Server_Pipeline)
          ])
```

Figura 2.4: La FeatureUnion per la gestione della feature server.

2.8 La Feature *Cache_Control*

La feature trattata successivamente è *Cache_Control*. Essa contiene dati di tipo stringa, che presentano una difficoltà in più rispetto ai casi già visti. Le celle di questa colonna infatti, contengono più valori, separati dal carattere ';'. Ciascun valore rappresenta un header HTTP relativo alla gestione dei dati nella cache del browser. È stato necessario, perciò, costruire un nuovo transformer per gestire questa situazione. La codifica adottata è stata realizzata *ad-hoc* nell'ottica di dare un output simile a quello di `LabelBinarizer`, con la differenza che, a causa della nuova problematica presentata, per ogni riga della matrice in output potranno essere presenti più caratteri 1 invece che uno solo. Quanto detto è stato implementato all'interno della classe `cache_handler`, e, in accordo con il paradigma proposto da ScikitLearn, questa trasformazione è stata divisa in due fasi. La prima è stata realizzata nel metodo `fit()`, il quale, preso in input il vettore colonna originario, si occupa di listare tutti i possibili valori presenti, inserendoli in una struttura di tipo dizionario che viene poi salvata in una variabile di istanza. La seconda fase è realizzata dal metodo `transform()`, che, considerando i valori salvati nella variabile di istanza dal metodo precedente, crea e restituisce in output la tabella formattata nel modo descritto in `fit()`. Facendo riferimento all'*Esempio 2.2*, la colonna in input potrebbe avere un aspetto del tipo $w = [a, ab, c, cb, ..b]$. Un vettore del genere, viene codificato, dalla classe così definita, nel modo seguente: $o = [[1, 0, 0], [1, 1, 0], [0, 0, 1], [0, 1, 1], ..., [0, 1, 0]]$.

2.9 Le Feature *Country* e *Province*

Le feature *Country* e *Province* indicano rispettivamente il paese e la provincia (o stato) restituiti dall'interrogazione *whois* sul dominio in questione. Anche in questo caso i dati sono codificati come stringhe e sono presenti nella tabella

alcuni valori nulli; inoltre, come capitato per il charset, molto spesso la stessa stringa è codificata a volte in uppercase, altre in lowercase. Nella gestione di queste due ultime problematiche sono state riusate le classi `check_nan` e `string_upper`, mostrate in precedenza. Per quanto riguarda la codifica vera e propria, anche in questa occasione è stata sfruttata la classe `LabelBinarizer`, per ottenere gli stessi vantaggi discussi in precedenza.
È opportuno notare che, in questo caso, per quanto concerne l'attributo indicante la provincia di appartenenza, non vale 'strettamente' lo stesso ragionamento applicato in precedenza. Alcuni valori infatti (ad esempio le province appartenenti ad uno stesso stato) sono maggiormente correlati di altri, dunque si potrebbe sostenere che sarebbe opportuno conservare queste correlazioni. Nonostante ciò, abbiamo comunque utilizzato la codifica di `LabelBinarizer`, poiché questa correlazione è mantenuta dalla presenza della feature Country.

2.10 La Feature *Within_Domain*

La feature *Within_Domain* indica, per ogni dominio, una parte dell'URL, rappresentato tramite il pattern `domainName.suffisso`, (ad esempio unisa.it). Tale attributo è ovviamente rappresentato come stringa.
Poiché nella maggioranza dei casi questo valore risulta diverso per ogni istanza, e considerando che dati di questo tipo non hanno valore in analisi del genere, è stato deciso di non conservare l'attributo nella sua interezza, considerando solo l'estensione finale (i.e. *.it*, *.com*, ecc...). Per la realizzazione di ciò è stato implementato un transformer *ah-hoc*, denominato `extension_extractor`, che, dato un vettore di url, ritorna un vettore contenente solo le estensioni degli URL in input. A questo punto le estensioni sono state codificate tramite `LabelBinarizer`.

2.11 La Feature *TCP Port*

La feature *TCP port* è una attributo di tipo numerico che indica la porta TCP tramite la quale scambiare informazioni con il sito web. In questo caso, pur essendo la feature di tipo tipo numerico, è stato comunque necessario cambiare il tipo di codifica. Infatti, anche per le porte TCP, il fatto che due numeri siano *vicini*, non indica (necessariamente) una correlazione. Per essere

più precisi, molti algoritmi di machine learning, durante la fase di training, calcolano dei '*pesi*', che fungono da coefficienti da moltiplicare per il valore numerico delle feature (con una relazione di tipo *one-to-one*, nel senso che ogni feature è moltiplicata sempre con lo stesso coefficiente), per arrivare al risultato finale. Si supponga di avere due porte, una con un valore prossimo a 400, ed una con valore prossimo a 500: nonostante esse potrebbero non essere correlate affatto, sarebbero moltiplicate per un valore comune, il quale, per forza di cose, tratterebbe la porta prossima a 400 in un modo molto più simile alla porta prossima a 500 di quanto non possa fare con una porta con valore prossimo a 10; nonostante essa, in linea teorica, potrebbe avere una correlazione forte con quest'ultima. Rappresentando le porte come numeri progressivi si forzano correlazioni tendenzialmente inesistenti.

2.12 Le Feature Rimanenti

A questo punto restano solo poche altre feature, tutte di tipo numerico. Esse sono elencate di seguito, assieme ad una loro breve descrizione.
Feature legate agli Header del protocollo HTTP:

- *Content_Length*: Si riferisce al rispettivo header di una HTTP Response indicante la lunghezza in byte del payload che esso contiene. I malware potrebbero modificare il valore contenuto in questo campo in modo tale che non corrisponda alla reale lunghezza del payload.

Feature legate a caratteristiche del server e delle sue risorse
Esse sono state catturate dall'autore del dataset tramite l'utilizzo di un *crawler*[9] per la comunicazione.

- *TCP_Conversation_Exchange*: Si riferisce al numero totale di pacchetti TCP inviati dal crawler al server durante la conversazione. Siti web maligni potrebbero fare in modo di far inviare multiple richieste ad un client.

[9]Un crawler (detto anche web crawler, spider o robot), è un software che analizza i contenuti di una rete (o di un database) in un modo metodico e automatizzato. In questo caso è stato sfruttato per analizzare i contenuti scambiati nella sessione HTTP tra client (simulato dal crawler stesso) e server.

- *Remote_IPS*: Si riferisce al numero totale indirizzi IP al quale il crawler si connette quando tenta di esplorare un sito web; valori> 1 possono essere dovuti a pubblicità presenti nella pagina Web, risorse esterne come immagini prese da internet, ma anche redirezioni forzate verso altri server contenenti script malevoli.

Feature legate alla comunicazione tra client e server
Anche queste informazioni sono state raccolte tramite crawler.

- *App_bytes*: Numero di byte inviati dal crawler al server nella comunicazione, escludendo quelli inviati per le query DNS. Un software maligno potrebbe forzare il client, impersonificato dal crawler, ad inviare varie HTTP Request in sequenza.

- *UDP_packets*: Numero di pacchetti UDP generati durante la comunicazione tra client e server durante lo scambio di contenuti di tipo multimediali (es: video streaming).

- *TCP_urg_packets*: Numero di pacchetti TCP con il flag 'urgent' impostato. Alcuni attaccanti fanno abuso di questo flag per bypassare firewall non opportunamente settati per esso.

- *Source_app_packets*: Rappresenta il numero totale di pacchetti inviato dal crawler ai server remoto durante la conversazione.

- *Remote_app_packets*: Inverso del precedente, rappresenta il numero di pacchetti ricevuti dal crawler durante la conversazione.

- *Source_app_bytes*: Correlata ad App_bytes, rappresenta, in byte, la mole di dati totale inviata dal crawler al server, comprendendo anche i byte relativi alle query DNS.

- *Remote_app_bytes*: Inverso del precedente.

- *App_Packets*: Numero totale di pacchetti ip inviati dal crawler al server nella conversazione.

Feature legate alla comunicazione tra client e server DNS
Anche questa informazione è stata raccolta tramite crawler.

- *DNS_query_times*: Numero di query al server DNS effettuate dal crawler. Similmente a quanto accaduto per Remote_IPS, anche in questo caso redirezioni o contenuti nel payload appartenti a diversi FQDN possono rendere necessari multipli accessi al server DNS.

Queste feature sono state le più agevoli da trattare, infatti è stato sufficiente effettuare la gestione dei valori nulli (sostituiti in alcuni casi con degli zeri, ed in altri con il valore medio della colonna, a seconda del significato), e l'utilizzo di `StandardScaler` per scalare i valori in un intervallo compreso tra -1 ed 1. Tutto ciò è stato implementato nei transformer `select_and_scale1` e `select_and_scale2`.

2.13 Assemblaggio delle Pipeline

Dopo aver definito le Pipeline per ogni feature, è stato necessario assemblarle. Ciò è stato facile grazie all'utilizzo della classe `FeatureUnion`, che permette un aumento dell'efficienza parallelizzando tutte le operazioni, come già descritto in precedenza. La feature union, realizzata come descritto, ha la struttura seguente:

```
In [749]: full_pipeline = FeatureUnion([
              ('select_and_scale1', select_and_scale1),
              ("charset_pipeline", charsetPipeline),
              ("server_pipeline", server_feature_union),
              ("cache_pipeline", Cache_Pipeline),
              ("content_length_pipeline", content_length_pipeline),
              ("countryPipeline", countryPipeline),
              ("provincePipeline", provincePipeline),
              ("domain_pipeline", domain_pipeline),
              ('select_and_scale2', select_and_scale2),
              ("portPipeline", portPipeline),
              ('select_and_scale3', select_and_scale3)
          ])
```

Figura 2.5: La FeatureUnion con tutte le Pipeline.

A questo punto abbiamo incorporato la Pipeline mostrata in *Figura 2.1*, creando una Pipeline generale, nel modo seguente:

```
In [752]: data_preparator = Pipeline([
              ("prePipeline", prePipeline),
              ("full_pipeline", full_pipeline)
          ])
```

Figura 2.6: Pipeline completa.

Il metodo `transform()` di quest'ultima prende in input un DataFrame strutturato come la nostra tabella iniziale, per restituire un array bidimensionale

che è possibile trasmettere ad un transformer classico (che, lo ricordiamo, prende in input un array).
L'automatizzazione di questo processo è necessaria non tanto per i dati che serviranno per allenare e testare gli algoritmi, quanto per il trattamento di possibili dati nuovi, magari inesistenti al momento in cui questo codice è stato scritto, purché con la stessa formattazione (per i quali va invocato esclusivamente il metodo `transform()`, in quanto il `fit()` è iniziale e relativo esclusivamente alla matrice da utilizzare nella fase di learning). Si può vedere come, dopo tutto il lavoro di automatizzazione, il processo risulti praticamente immediato:

```
In [753]: data = pd.read_csv("Matriz_Completa.csv")
          data_prepared = data_preparator.fit_transform(strat_train_set)
```

Figura 2.7: Esempio di utilizzo della Pipeline completa.

L'istruzione appena mostrata crea prima la struttura necessaria per ogni pipeline (il metodo `fit()` di ogni Pipeline viene invocato) e poi trasforma i dati in input (data) invocando i `transform()` di ogni Pipeline. La variabile `data_prepared` è pronta per essere data in input agli algoritmi di machine learning per la fase di training.

2.14 Preparazione delle Etichette

Terminata la preparazione delle feature, si passa alla preparazione delle *labels*, ovvero le etichette che identificano un sito come malevolo o non malevolo. Nel caso in esame esse sono rappresentate sotto forma di stringhe ('Benigna'/'Maligna'[10]. Il vettore contenente le etichette, per essere accettato dagli algoritmi di machine learning, deve essere, anch'esso, costituito da soli numeri, e deve essere monodimensionale. Anche questa volta, per ottenere il risultato desiderato è stata utilizzata la classe `LabelBinarizer`. A differenza dei casi precedenti, tuttavia, in questo caso si ha bisogno di un vettore monodimensionale, mentre l'output di `LabelBinarizer` ha la struttura seguente (come sempre è un vettore di vettori colonna, ciascuna delle quali in questo caso ha lunghezza 1): $o = [[0], [1], [0], ..., [1]]$. È possibile ottenere facilmente il risultato desiderato invocando, sul vettore in output, il metodo `flatten()`.

[10]Questo perché il dataset è stato collezionato da spagnoli.

È importante notare che la codifica stringa-numero è decisa dal metodo `fit()`, e che, quindi, anche in questo caso esso deve essere invocato solo una volta (la prima). All'atto pratico, eseguire tale codifica risulta davvero semplice, di seguito vi è il codice da eseguire la prima volta:

```
In [802]: label_binarizer = LabelBinarizer()
          train_set_labels = label_binarizer.fit_transform(strat_train_set["TIPO"])
          train_set_labels = train_set_labels.flatten()
```

Figura 2.8: Preprocessing delle etichette tramite LabelBinarizer.

All'arrivo di etichette nuove, invece, queste devono essere convertite invocando solo `transform()`, utilizzando la stessa istanza di `LabelBinarizer`, nel modo seguente:

```
In [2]: new_labels = label_binarizer.transform(test_set["TIPO"])
        new_labels = new_labels.flatten()
```

Figura 2.9: Preprocessing di etichette nuove tramite il LabelBinarizer già impostato.

Questo paragrafo conclude il discorso intrapreso circa la preparazione dei dati.

3 Allenamento degli Algoritmi

Dopo aver concluso la fase di preparazione dei dati, sono stati allenati alcuni modelli predittivi su una parte di essi e ne sono state valutate le performance. Innanzitutto, occorre notare che nell'intero dataset vi è una presenza del 12.13% circa di siti malevoli: questo indica che con una predizione banale che restituisce sempre 'non malevolo', si raggiunge circa l'87.5% di accuratezza. Il nostro obiettivo è stato perciò il superamento di questa soglia.

3.1 Random-Forest Classifier

3.1.1 Caratteristiche Generali

Il primo modello che è stato allenato è un `RandomForestClassifier`. Esso è un algoritmo di *classificazione* basato su un modello più semplice, detto *Decision Tree* (albero di decisione).
Gli alberi di decisione sono potentissimi algoritmi di machine learning, capaci di eseguire sia classificazione che *regressione*; essi sono in grado di modellare dataset anche molto complessi. Una struttura di questo tipo è in grado di rappresentare una successione di decisioni binarie (sì/no). Nel caso della classificazione, essa modella i nodi interni e le loro connessioni come dei veri e propri percorsi, che portano a dei nodi foglia: ogni nodo foglia indica una classe. Un esempio di albero decisionale è mostrato in *Figura 3.1*.

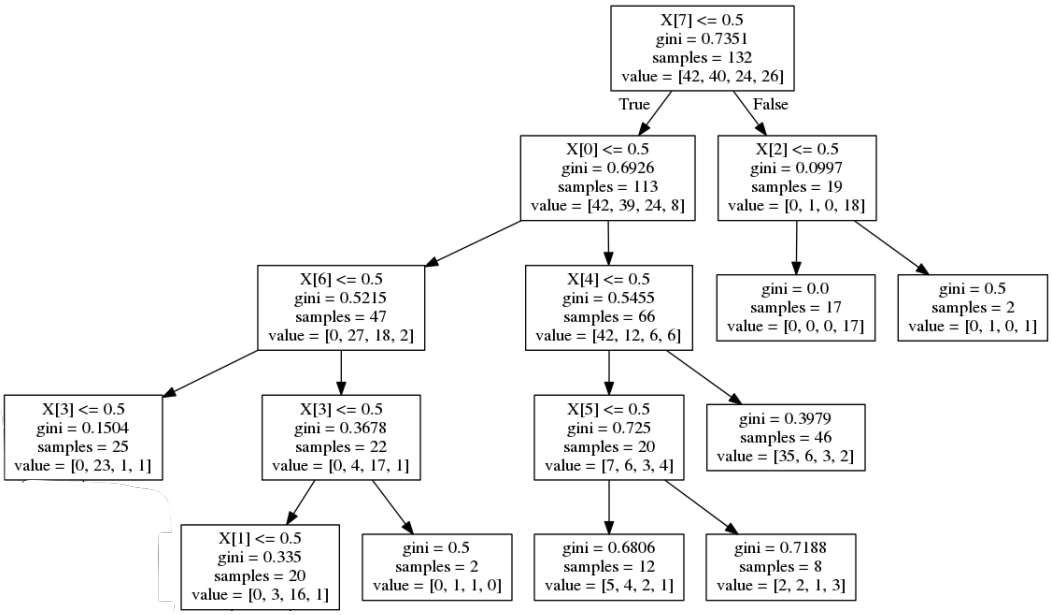


Figura 3.1: Esempio di albero decisionale.

Tipicamente, come nel caso di ScikitLearn, è possibile fissare la profondità massima dei nodi foglia tramite un apposito iperparametro (nel caso di ScikitLearn esso è `max_depth`). Un iperparametro di questo tipo è detto iperparametro di *regolarizzazione*, in quanto regola la complessità del modello (ovvero il trade-off overfitting vs underfitting).

Il modello Random Forest consiste semplicemente in un insieme di alberi di decisione. Ogni albero della 'foresta' è allenato su un sottoinsieme (casuale) delle feature del dataset; inoltre, è possibile definire la profondità massima di tutti gli alberi di decisione, oppure fare in modo che anche questa sia scelta in modo casuale, in modo diverso per ogni albero. Una volta allenata l'intera foresta, essa è in grado di effettuare predizioni combinando quelle di tutti gli alberi decisionali. Algoritmi di questo tipo (cioè basati sull'assemblaggio di modelli più semplici) vengono detti *ensemble methods*.

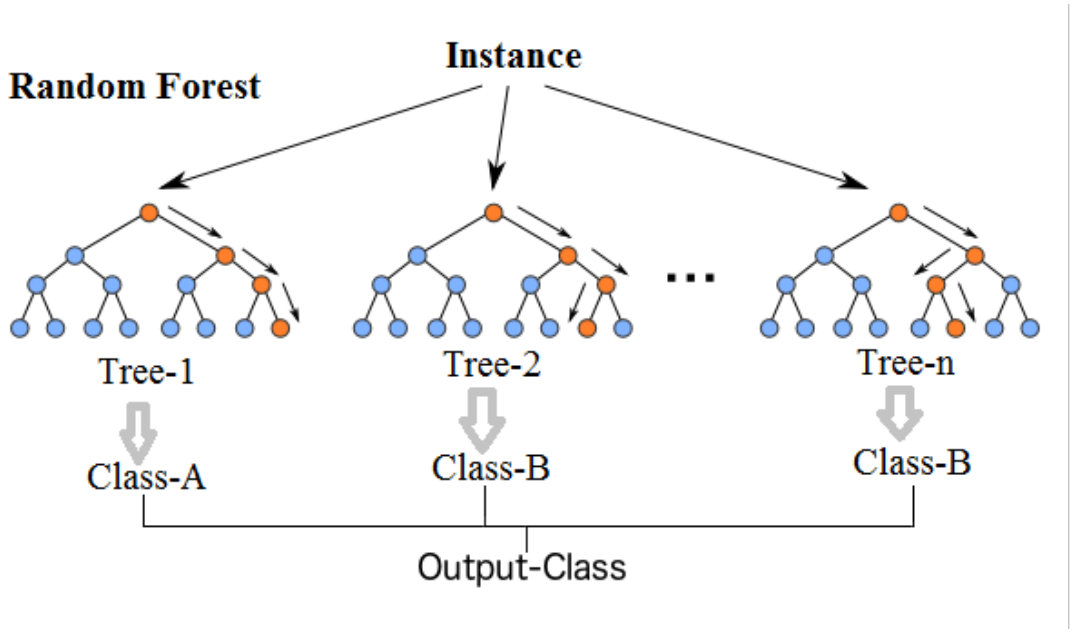


Figura 3.2: Esempio di Foresta Casuale.

3.1.2 Una prima esplorazione...

Per modellare classificatori di tipo Random-Forest, ScikitLearn fornisce la classe `RandomForestClassifier`. Per un primo contatto con il modello, è stata creata un'istanza della classe, la quale è stata allenata sul `train_set` utilizzando dei parametri che sono stati valutati come ragionevoli in base alle caratteristiche strutturali del modello e del dataset. Il codice realizzato per questo primo allenamento è il seguente:

```
In [925]: from sklearn.ensemble import RandomForestClassifier
          rnd_clf = RandomForestClassifier(n_estimators = 100, max_leaf_nodes = 13)
          rnd_clf.fit(train_set_prepared, train_set_labels)

Out[925]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                      max_depth=None, max_features='auto', max_leaf_nodes=13,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
                      oob_score=False, random_state=None, verbose=0,
                      warm_start=False)
```

Figura 3.3: Un primo esempio di `RandomForestClassifier`.

Successivamente è stata definita una funzione per misurare l'accuratezza delle previsioni, e sono state inizialmente calcolate delle predizioni sul `train_set`, ovvero sui dati che l'algoritmo ha usato per fare il training:

```
In [928]: def accuracy_rate(labels, predictions):
              count = 0
              for i in range(len(labels)):
                  if labels[i] == predictions[i]:
                      count += 1
              return count/len(labels)

          # ...

          predizioni_train = rnd_clf.predict(train_set_prepared)
          accuracy_rate(predizioni_train, train_set_labels)

Out[928]: 0.9367977528089888
```

Figura 3.4: Esempio di Foresta Casuale.

L'accuratezza raggiunta in vari test sul modello così allenato oscilla tra il 93 ed il 94%. Abbiamo preferito non testare questo modello sul `test_set` per evitare possibili condizionamenti (dettati dagli eventuali risultati) nei successivi allenamenti dello stesso modello.

3.1.3 Allenamento e Scelta degli iperparametri

Nel caso di `RandomForestClassifier`, gli iperparametri principali da settare sono il numero di alberi di decisione, definito come `n_estimators`, ed il numero massimo di nodi-foglia per ogni albero, chiamato `max_leaf_nodes`. Per scegliere i parametri ottimali, è necessario testare le performance del modello con varie combinazioni di essi. Poiché i loro range sono abbastanza ampi, può essere laborioso eseguire i test necessari manualmente. Per fortuna, ScikitLearn mette a disposizione la classe `GridSearchCV`, che permette di automatizzare questo processo.
Innanzitutto, è necessario specificare una griglia contenente i parametri con cui si vuole che il classificatore sia testato. Questa griglia è rappresentata come un vettore, le cui entry sono dei dizionari, uno per ogni parametro. Quella che è stata definita inizialmente è illustrata nella *Figura 3.5*, la quale contiene dei parametri molto generali:

```
In [931]: param_grid = [
              {"max_leaf_nodes": [5, 8, 10, 12, 15]},
              {'n_estimators': [50, 75, 100, 175, 250, 500]}
          ]
```

Figura 3.5: Griglia di parametri di test per `RandomForestClassifier`.

Tramite tale griglia, l'oggetto `GridSearchCV` esegue sia la fase di training che di testing (sul validation set) del classificatore con ogni possibile combinazione di parametri tra quelli forniti, restituendo la coppia che si è rivelata

maggiormente accurata. Quanto descritto è eseguito dal codice seguente:

```
In [936]: grid_search = GridSearchCV(rnd_clf, param_grid, cv = 5, scoring = 'neg_mean_squared_error')
          grid_search.fit(train_set_prepared, train_set_labels)
          grid_search.best_params_

Out[936]: {'max_leaf_nodes': 19}
```

Figura 3.6: Griglia di parametri di test per `RandomForestClassifier`.

In questo caso, `rnd_clf` è un'istanza di `RandomForestClassifier`, mentre il parametro `scoring` indica quale *loss function* (vedi *sezione 1.5*) si vuole minimizzare. Un parametro che merita una spiegazione a parte è `cv`: esso sta per Cross-Validation, la quale è stata descritta nel paragrafo 1.6. In questo caso essa viene effettuata con $k = 5$ sottoinsiemi del `train_set`.
Dopo aver invocato `fit()`, la variabile `best_params` contiene il risultato desiderato, ovvero la coppia di valori per gli iperparametri che si è comportata meglio.
Al primo tentativo, come vediamo dall'immagine, il contenuto della variabile `best_params` è `max_leaf_nodes = 19`. Il fatto che il parametro `n_estimators` non sia elencato, indica che la modifica di questo valore non ha avuto impatto sulle performance. Dopo aver effettuato delle ricerche a grana più fine, ripetendo le suddette operazioni con differenti valori nei due vettori, è risultato che i valori maggiormente indicati fossero la coppia (`max_leaf_nodes = 24`, `n_estimators = 50`).

3.1.4 Performance

Dopo aver allenato il modello sul `train_set`, con i parametri indicati da grid-search, si riesce ad ottenere un'accuratezza (misurata con la medesima funzione definita nella sottosezione precedente) che sul `train_set` oscilla tra il 94.5 ed il 96%; mentre sul `test_set` oscilla tra il 94 ed il 95%. I parametri forniti da grid-search, hanno quindi permesso di migliorare l'accuratezza in modo piuttosto significativo.
E' necessario sottolineare che l'accuratezza 'oscilla' in un certo range, poiché il modello non è costruito in modo deterministico; come il nome suggerisce, infatti, gli algoritmi Random-Forest hanno una componente casuale, dunque il loro output varia leggermente tra le invocazioni, anche mantenendo la stessa configurazione e lo stesso input.

3.2 Support Vector Machines Classifier

3.2.1 Caratteristiche Generali

Il secondo algoritmo che è stato allenato è un classificatore basato sul modello Support Vector Machines (SVM). Questo modello è uno dei più popolari nell'ambito del machine learning, ed è stato scelto poiché 'adatto a dataset complessi ma piccoli', ossia aventi un buon numero di colonne ma non tante righe, descrizione che calza perfettamente al dataset utilizzato. Il modello SVM ha un funzionamento nella pratica molto semplice: esso cerca di separare le classi (i.e. gli elementi appartenenti alle possibili label) tramite il margine più ampio possibile. La sua natura intrinseca lo rende un modello di classificazione binaria non probabilistico; questo perché nativamente discrimina tra **due sole** classi, e non fornisce una approssimazione sulla probabilità con la quale un'istanza del problema appartenga a ciascuna di esse (a differenza di altri algoritmi che offrono anche tale opportunità).
Per essere più precisi, se il dataset contiene dati ad n features, l'algoritmo vede le tuple come elementi di uno spazio ad n dimensioni, e cerca di separarle costruendo un iperpiano ad $n-1$ dimensioni. Questo iperpiano cerca di raggiungere due obiettivi: il primo è quello di separare le due classi nel modo più accurato possibile; ovvero minimizzare il numero di tuple collocate dalla parte sbagliata. Il secondo obiettivo che si cerca di raggiungere è quello di massimizzare la distanza minima tra le istanze delle due classi e l'iperpiano (tale distanza è detta *margine*).
Questi due obiettivi entrano spesso in conflitto, e poiché alla fine sarà uno solo l'iperpiano scelto, è necessario fare dei compromessi. A questo scopo è utilizzato l'iperparametro C: esso permette di *spostare* l'iperpiano finale verso l'uno o l'altro degli iperpiani che più soddisfano gli obiettivi descritti. Tale parametro è settato di default a 1, che indica esattamente una via di mezzo tra queste due 'condizioni ideali'. Mentre un valore C grande preferirà massimizzare il margine tra l'iperpiano e le istanze delle classi, un C piccolo cercherà di separare le due classi nel modo più accurato possibile.
A questo punto è più semplice comprendere la natura binaria e non probabilistica del modello: esso si limita semplicemente a restituire la collocazione della tupla relativamente all'iperspazio costruito.

A titolo di esempio, mostriamo un grafico che rappresenta la classificazione effettuata da tre differenti algoritmi di classificazione comparata a quella effettuata da un modello SVM, su alcuni dati avente due sole feature (rappresentabili dunque in un piano).

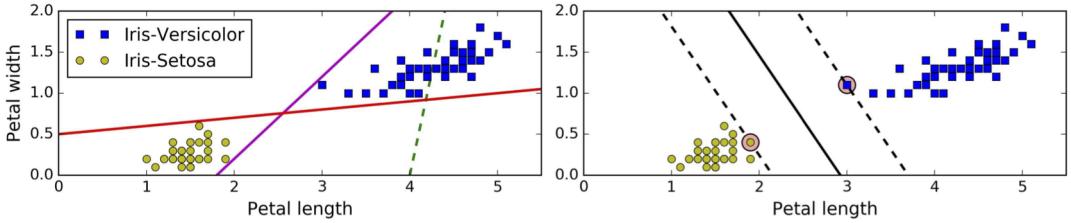


Figura 3.7: Esempio grafico di classificazione eseguita da tre modelli standard (linee rossa, viola e tratteggiata verde), a sinistra, e da un classificatore basato su SVM, a destra.

3.2.2 Scelta degli iperparametri

Per la scelta degli iperparametri, come nel caso precedente, ci si è serviti delle facilitazioni fornite da `GridSearchCV`. Anche stavolta è stata effettuata prima una ricerca su valori più ampi, per poi affinarla successivamente. La prima ricerca è stata effettuata nel modo seguente:

```
In [64]: classifier = LinearSVC()
         param_grid = [
             {'C': [.1, .5, 1, 10, 50, 100, 500]}
         ]
         classifier = LinearSVC()
         grid_search = GridSearchCV(classifier, param_grid, cv = 5, scoring = 'neg_mean_squared_error')
         grid_search.fit(train_set_prepared, train_set_labels)

Out[64]: GridSearchCV(cv=5, error_score='raise',
                estimator=LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
               intercept_scaling=1, loss='squared_hinge', max_iter=1000,
               multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
               verbose=0),
                fit_params=None, iid=True, n_jobs=1,
                param_grid=[{'C': [0.1, 0.5, 1, 10, 50, 100, 500]}],
                pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
                scoring='neg_mean_squared_error', verbose=0)
```

Figura 3.8: Scelta dei parametri migliori per il classificatore svm.

Come nel caso precedente, anche stavolta è stato impostato $cv = 5$. Come output della prima ricerca effettuata, il parametro migliore è risultato essere $C = 1$. Anche affinando la ricerca, questo risulta essere il valore che si comporta meglio.

3.2.3 Performance

A questo punto, come per il modello Random Forest, questo è stato allenato e testato inizialmente sul `train_set`, settando $C = 1$. Ne sono state successivamente analizzate le performance sullo stesso insieme di dati nel modo seguente:

```
In [66]: classifier = LinearSVC(C= 1, loss = "hinge")
         classifier.fit(train_set_prepared, train_set_labels)
         predizioni = classifier.predict(train_set_prepared)
         accuracy_rate(train_set_labels, predizioni)

Out[66]: 0.9859550561797753
```

Figura 3.9: Performance del classificatore SVM con iperparametro ottimale sul `train_set`.

L'accuratezza di tali previsioni risulta molto alta: ciò potrebbe essere dovuto sia ad un overfitting sui dati del `train_set` da parte del modello, sia al fatto che effettivamente il modello è stato in grado di generalizzare molto bene. In presenza di dubbi di questo tipo, prima di testare il modello sul `test_set`, sarebbe opportuno testarlo su un altro insieme di dati: il `validation_set`, introdotto nel paragrafo 1.6. Questo approccio risulta conveniente perché, qualora anche le performance sul `test_set` fossero degradate, una volta modificato il modello non saremmo più in grado di discriminare tra overfitting e situazioni 'normali', non avendo più dati nuovi a disposizione.
Tuttavia, in questo caso questa parte del lavoro non è necessaria poiché essa è già stata effettuata da `GridSearchCV`. Questa classe infatti esegue la cross-validation sul `train_set`, e ripete tale validazione per ogni possibile coppia di parametri in input. Quindi essa restituisce in output la coppia che offre performance migliori in assoluto su quel `train_set`, e non avrebbe alcun senso allenare il modello con parametri diversi.
Come per il caso precedente, GridSearchCV è stato eseguito più di una volta, affinando sempre di più la ricerca. Questo ci ha permesso di giungere al settaggio migliore possibile rispetto ai nostri dati.
Al termine di questa fase è stato eseguito l'allenamento dell'algoritmo sul `train_set` con gli iperparamtri scelti (tramite istruzione `classifier.fit()`, come mostrato in *Figura 3.9*. In seguito è stato effettuato il testing sul `test_set` nel modo seguente:

```
In [69]: test_set_prepared = data_preparator.transform(test_set)
         test_set_labels = label_binarizer.transform(test_set["TIPO"])
         predizioni_test = classifier.predict(test_set_prepared)

         accuracy_rate(test_set_labels, predizioni_test)

         /Users/giovanni/anaconda/lib/python3.6/site-packages/sklearn/utils/validation.py:444: DataConversionWarning: Data wit
         h input dtype object was converted to float64 by StandardScaler.
           warnings.warn(msg, DataConversionWarning)

Out[69]: 0.9747191011235955
```

Figura 3.10: Performance del classificatore SVM (lanciato con iperparametro ottimale) sul `test_set`.

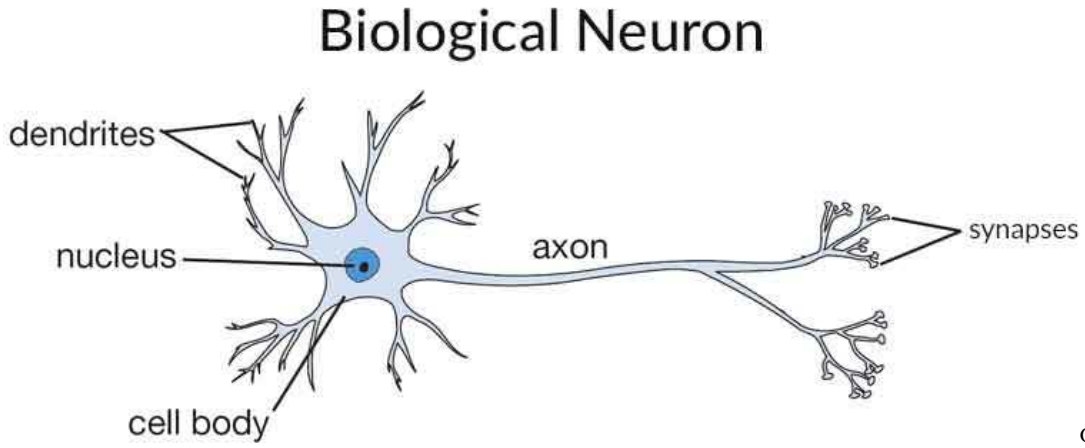
L'accuratezza, come sperato, è risultata molto alta, permettendoci di accantonare immediatamente l'ipotesi overfitting. Considerando che questa volta essa è stata misurata sul `test_set`, composto, come si ricorda, da dati che il classificatore non ha mai visto, si può ritenere questo modello molto affidabile.
Nel caso in cui le performance del modello sul `test_set` fossero state invece degradate, sarebbe stato opportuno accantonarlo definitivamente, in quanto non ci sarebbe stato più modo di tenere sotto controllo l'overfitting.

3.3 Classificazione con Reti Neurali

3.3.1 Struttura Generale di una Rete Neurale

L'ultimo modello allenato è anche il maggiormente complesso: esso è un classificatore basato su reti neurali profonde, appartenente ad un sottoinsieme del machine learning chiamato *deep learning*. Le reti neurali permettono il trattamento di dati che presentano strutture dalla complessità estrema, quali, ad esempio, il linguaggio umano. Rispetto ai più comuni (ed al contempo potenti) algoritmi di machine learning, le reti neurali hanno il vantaggio di offrire ottime performance riguardo la classificazione di dati non linearmente separabili (divisione sulla quale posano tutti gli algoritmi della tipologia precedente per effettuare le proprie classificazioni, i quali, matematicamente, compiono le proprie decisioni suddividendo lo spazio vettoriale in sottospazi di un ordine inferiore). Sebbene la complessità del dataset adoperato non sia paragonabile a quella dei problemi che richiedono l'utilizzo di tecnologie così potenti, è stato comunque scelto di compiere dei tentativi per vedere se avessero potuto garantire delle performance ancora migliori di quelle già

ottenute.
Le reti neurali, formalmente Artificial Neural Networks (*ANNs*), sono dei sistemi di apprendimento automatico che si ispirano alle strutture celebrali di esseri umani ed animali. Il componente fondamentale di una *ANN* è, ovviamente, il *neurone artificiale*. Esso è rappresentato da una funzione matematica che si pone di modellare e simulare il comportamento delle cellule neuronali degli esseri animati (o quantomeno alcuni aspetti di queste). Un neurone biologico si compone di tre parti fondamentali: i *dendriti*, il *corpo*, e l'*assone*.



qui

Tramite i dendriti, degli impulsi elettrici vengono trasmessi alla cellula; possiamo dunque assumerli come il dispositivo di input del neurone. Il corpo, ovvero la parte centrale della cellula, è quella che si occupa di *processare* gli impulsi ricevuti. L'output, se così si può definire, viene trasmesso ad altri neuroni tramite l'assone e le sinapsi ad esso connesse.
Un neurone *artificiale* rappresenta questa struttura come indicato in figura:

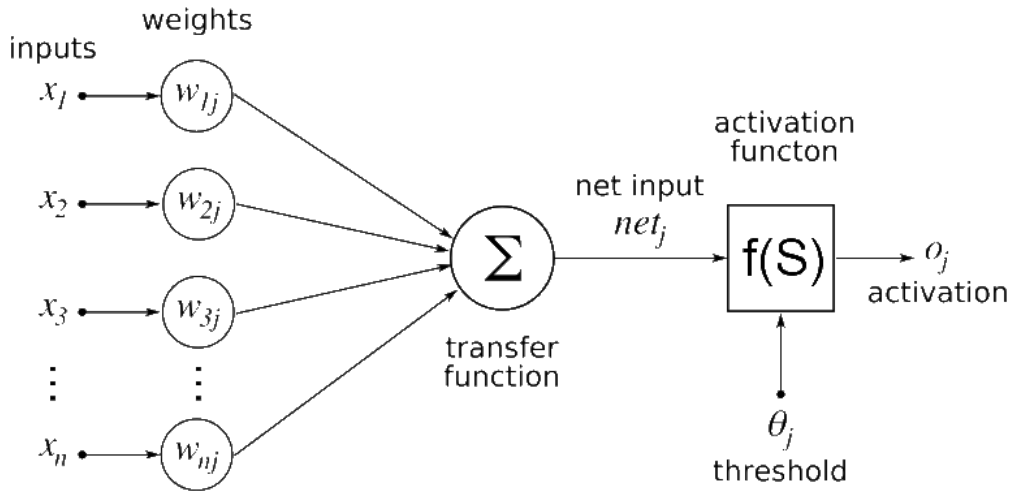


Figura 3.11: Struttura di un neurone artificiale.

Il neurone artificiale prende in input dei valori, i quali rappresentano gli stimoli in input ai dendriti. Questi input sono moltiplicati a dei pesi, e poi sommati da una qualche funzione decisa in base allo scopo. Il valore risultante viene dato in input ad una funzione successiva, detta *funzione di attivazione*. Scelte comuni di funzioni di attivazione sono la funzione *logistica*, la funzione *rectifier* (detta più comunemente *ReLU*) e la funzione *softmax*. L'output, θ_j, viene trasmesso al neurone successivo.

Per costruire una rete neurale, i neuroni artificiali vengono assemblati e disposti in livelli, i quali sono essenzialmente di tre tipi: livelli di input, livelli nascosti (*hidden layers*) e livelli di output. Comunemente, si ha un solo livello di input, dove ogni neurone è associato ad colonna della matrice in input; numerosi livelli nascosti, il cui numero e dimensione variano in base alla complessità dei dati, ed un unico livello di output, che calcola il risultato finale. Questi livelli sono organizzati come nella *Figura 1*.

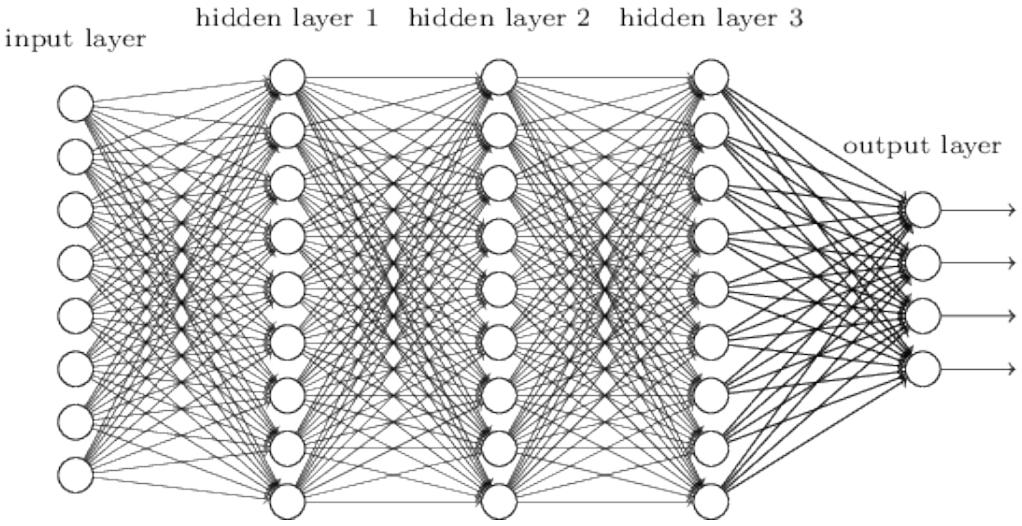


Figura 3.12: Struttura di una rete neurale a 5 livelli.

A differenza dei livelli di input ed output, il cui numero di unità è sostanzialmente fissato, per i livelli nascosti non vi sono formalmente limiti. Ogni livello nascosto può contenere un numero arbitrario di neuroni, e la rete neurale stessa può contenere un numero arbitrario di livelli nascosti. La modifica di questi parametri ha un impatto notevole sia sulla accuratezza della rete, sia sulle risorse computazionali necessarie per il suo allenamento. Non serve, nel caso esaminato, far crescere di molto la rete perché l'allenamento di questa richieda tempi e risorse computazionali assolutamente proibitive.

Come regola generale riguardo problemi molto complessi, vale che la crescita in profondità della rete (l'aumento del numero di livelli nascosti) fa aumentare l'accuratezza della rete in modo esponenziale rispetto all'aumento che si avrebbe con la crescita in larghezza dei singoli livelli. In generale, però, bastano solo pochissimi livelli per avere delle performance molto elevate su problemi semplici: inoltre, se le varie classi sono separabili linearmente, basta un solo livello nascosto. Per quanto riguarda la complessità, invece, mentre la crescita in larghezza dei singoli livelli non inficia in modo significativo sulle risorse computazionali richieste per il training, ciò non è vero per la crescita in profondità, la quale ha invece un impatto notevole. Una rete neurale che contiene almeno due livelli nascosti, è detta rete neurale *profonda* (*deep neural network*).

Nel mondo reale, dopo molti anni dalla diffusione di queste tecnologie, sono state individuate alcune *rules-of-thumb* derivate empiricamente per la struttu-

razione delle reti neurali. Queste regole suggeriscono che il numero di neuroni del primo livello nascosto deve essere proporzionale al numero di neuroni del livello di input, sommato al numero di neuroni del livello di output, diviso due; come riassunto nella seguente formula: $N_{FirstHidden} \simeq \frac{N_{output}+N_{input}}{2}$.
Anche riguardo l'allenamento, le reti neurali presentano meccanismi differenti rispetto a quello degli altri algoritmi. Come abbiamo visto precedentemente, i collegamenti tra i vari neuroni artificiali sono *pesati* da un certo fattore numerico. Lo scopo dell'allenamento è quello di settare questi pesi in modo che input della stessa classe (i.e benigna - maligna nel nostro caso) producano output quanto più possibile simili, e distinguibili da quelli delle altre classi. In tal modo è definita la strategia per la classificazione.
All'atto di costruzione della rete, questi pesi sono settati in modo puramente casuale, dopodiché si procede per iterazioni. Ad ogni iterazione, un numero n di istanze del train set, detto *batch*, è scelto a caso ed usato per *aggiustare* i pesi correnti. Ogni item del train set viene presentato alla rete, e ne viene calcolato l'output. Successivamente, qualora l'output sia sfasato rispetto al valore desiderato, i pesi dei vari neuroni sono modificati tramite un meccanismo detto *backpropagation*, letteralmente *propagazione all'indietro*. Ciò viene ripetuto per ogni elemento del batch; ogni batch corrisponde ad una iterazione dell'allenamento. Anche in questo caso, riguardo la scelta dei parametri non vi sono delle regole fisse. La loro crescita porta verso possibili overfitting e tempi di allenamento che si allungano repentinamente; mentre la loro riduzione porta verso tempi di allenamento più brevi ma possibili underfitting. È immediato notare come modifiche anche molto piccole nelle dimensioni dei batch abbiano un impatto molto più forte rispetto a modifiche al numero di iterazioni.

3.3.2 Allenamento di un Classificatore basato su Reti Neurali

La libreria TensorFlow semplifica molto al programmatore il lavoro, altrimenti estremamente tedioso, de costruzione e allenamento delle reti neurali. TensorFlow fornisce infatti la classe `DNNClassifier`, che permette di costruire classificatori basati su reti neurali (altamente customizzabili) in pochi e semplici passi. Ecco il codice necessario per realizzare il nostro classificatore.

```
In [342]: import tensorflow as tf
feature_cols = tf.contrib.learn.infer_real_valued_columns_from_input(train_set_prepared)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 300], n_classes = 2,
                                         feature_columns = feature_cols)

WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
INFO:tensorflow:Using default config.
INFO:tensorflow:Using config: {'_task_type': None, '_task_id': 0, '_cluster_spec': <tensorflo
w.python.training.server_lib.ClusterSpec object at 0x13bclad30>, '_master': '', '_num_ps_repl
icas': 0, '_num_worker_replicas': 0, '_environment': 'local', '_is_chief': True, '_evaluation
_master': '', '_tf_config': gpu_options {
  per_process_gpu_memory_fraction: 1
}
, '_tf_random_seed': None, '_save_summary_steps': 100, '_save_checkpoints_secs': 600, '_save_
checkpoints_steps': None, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000,
'_model_dir': None}
WARNING:tensorflow:Using temporary folder as model directory: /var/folders/lg/gcgbjxpj23g6k44
tq1_24zv80000gp/T/tmphp9o872a
```

Figura 3.13: Costruzione della rete neurale.

Illustriamo brevemente il funzionamento del codice appena presentato. Dopo aver importato la libreria, costruiamo un oggetto di tipo `Iterable` che permette di iterare sulle colonne del dataset: un riferimento a questo oggetto è inserito nella variabile `feature_cols`. Successivamente, viene creata l'istanza del classificatore: oltre al parametro `feature_columns`, vengono passati i parametri `hidden_units` e `n_classes`. Il parametro `n_classes` indica semplicemente quante sono le classi che stiamo cercando di distinguere, in questo caso due (sito malevolo - non malevolo). Il parametro `hidden_units`, invece, rappresenta il numero e le dimensioni dei livelli nascosti.
Come si evince dal codice, la rete che abbiamo allenato presenta due livelli nascosti: entrambi da 300 unità, è perciò una rete profonda. Per decidere il numero di neuroni per livello, ci si è riferiti alle regole presentate nel paragrafo precedente (occorre notare inoltre che la classe `GridSearchCV` non è compatibile con questo modello). A questo punto è stato necessario allenare la rete tramite il `train_set`:

```
In [343]: dnn_clf = tf.contrib.learn.SKCompat(dnn_clf) #   Necessario per TensorFlow >= 1.1.0
          dnn_clf.fit(train_set_prepared, train_set_labels, batch_size = 40, steps=500)

          WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
          WARNING:tensorflow:From /Users/giovanni/anaconda/lib/python3.6/site-packages/tensorflow/contr
          ib/learn/python/learn/estimators/head.py:615: scalar_summary (from tensorflow.python.ops.logg
          ing_ops) is deprecated and will be removed after 2016-11-30.
          Instructions for updating:
          Please switch to tf.summary.scalar. Note that tf.summary.scalar uses the node name instead of
          the tag. This means that TensorFlow will automatically de-duplicate summary names based on th
          e scope they are created in. Also, passing a tensor or list of tags to a scalar summary op is
          no longer supported.
          INFO:tensorflow:Create CheckpointSaverHook.
          INFO:tensorflow:Saving checkpoints for 1 into /var/folders/1g/gcgbjxpj23g6k44tq1_24zv80000gp/
          T/tmphp9o872a/model.ckpt.
          INFO:tensorflow:loss = 0.663419, step = 1
          INFO:tensorflow:global_step/sec: 92.6589
          INFO:tensorflow:loss = 0.19718, step = 101 (1.084 sec)
          INFO:tensorflow:global_step/sec: 86.9795
          INFO:tensorflow:loss = 0.107768, step = 201 (1.147 sec)
          INFO:tensorflow:global_step/sec: 97.7245
          INFO:tensorflow:loss = 0.170548, step = 301 (1.026 sec)
          INFO:tensorflow:global_step/sec: 97.9423
          INFO:tensorflow:loss = 0.0465684, step = 401 (1.022 sec)
          INFO:tensorflow:Saving checkpoints for 500 into /var/folders/1g/gcgbjxpj23g6k44tq1_24zv80000g
          p/T/tmphp9o872a/model.ckpt.
          INFO:tensorflow:Loss for final step: 0.024167.

Out[343]: SKCompat()
```

Figura 3.14: Allenamento della rete neurale.

Il cast effettuato nella prima riga è necessario per aggirare alcuni problemi di compatibilità presenti nelle versioni più recenti di TensorFlow. Come si evince dalla figura, inoltre, il metodo `fit()` prende in input i due parametri `batch_size` e `steps`: `batch_size`, rappresenta la dimensione di ogni batch, e `steps` indica invece quante iterazioni, ciascuna prendendo un input di dimensione batch, si devono eseguire per l'allenamento.
Poiché non vi sono regole standard per scegliere questi due parametri, i valori finali li abbiamo scelti in base alle performance. Come si nota dal codice in alto, durante l'allenamento TensorFlow fornisce ad ogni 100 iterazioni le performance della rete sui nuovi batch. In base a questi dati è possibile stimare, seppur in modo grossolano, dei parametri dall'accuratezza ragionevole. Dopo aver effettuato vari test empirici per scegliere la coppia di parametri migliore (su dati di training e validazione, ma non su dati di test!), è stato scelto di allenare la rete per 500 iterazioni, con batch da 40 istanze ciascuno.
Una caratteristica molto comoda delle reti neurali, che ha permesso di effettuare questi test risparmiando del tempo considerevole, è il fatto che le reti possono essere ri-allenate. Dopo aver allenato una rete, infatti, è possibile chiamare nuovamente il suo metodo `fit()`: l'allenamento conseguente partirà dai pesi impostati dall'allenamento precedente, e non da pesi casuali, permettendo di segmentare l'allenamento in varie fasi e testare la rete volta per volta.

3.3.3 Performance

Una volta costruita ed allenata la rete nei metodi descritti, come negli altri casi ne sono state valutato le performance sia sul `train_set` che sul `test_set`. Il codice sottostante riassume le modalità di test ed i risultati.

```
In [344]: predizioni_train = dnn_clf.predict(train_set_prepared)
          accuracy_rate(train_set_labels, predizioni_train["classes"])
          accuracy_rate(train_set_labels, predizioni_train["classes"])

          WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
          INFO:tensorflow:Restoring parameters from /var/folders/1g/gcgbjxpj23g6k44tq1_24zv80000gp/T/tm
          php9o872a/model.ckpt-500

Out[344]: 0.988061797752809
```

Figura 3.15: Performance del classificatore sul `train_set`.

```
In [345]: predicted_test = dnn_clf.predict(test_set_prepared)
          accuracy_rate(test_set_labels.flatten(), predicted_test["classes"])

          WARNING:tensorflow:float64 is not supported by many models, consider casting to float32.
          INFO:tensorflow:Restoring parameters from /var/folders/1g/gcgbjxpj23g6k44tq1_24zv80000gp/T/tm
          php9o872a/model.ckpt-500

Out[345]: 0.9775280898876404
```

Figura 3.16: Performance del classificatore sul `test_set`.

Nei vari test effettuati, l'accuratezza del modello così costruito, sui dati di test, è stata sempre compresa tra il 97 ed il 98%: un risultato davvero notevole, tanto da superare addirittura quella ottenuta tramite SVMs.

4 Conclusioni Finali

Come si è visto, le caratteristiche strutturali dei modelli hanno ripercussioni significative sull'accuratezza raggiunta. Nonostante ciò, anche nel caso Random-Forest, il modello meno 'adatto', l'accuratezza raggiunge livelli piuttosto accettabili ($94-95\%$). Miglioramenti significativi si ottengono allenando i classificatori basati su SVM e reti neurali (in entrambi i casi l'accuratezza è compresa tra il 97 ed il 98%).
Il processo di allenamento dei modelli è stato notevolmente facilitato dalle funzionalità di preprocessing messe a disposizione dalle librerie utilizzate; la riusabilità di tutto il processo e l'universalità del formato finale dei dati rendono istantaneo l'allenamento di nuovi modelli (anche non previsti inizialmente) e facilmente modificabili le strategie di trasformazione dei dati, a seconda delle proprie esigenze. Uno dei maggiori benefici derivanti da queste caratteristiche, è sicuramente la possibilità di utilizzare i dati per allenare algoritmi nuovi e magari ad oggi ancora non disponibili, senza bisogno di modifiche consistenti al preprocessing.

Riferimenti

1 Alcune Nozioni Preliminari

1. *Aurélien Géron*: Hands-On Machine Learning with Scikit-Learn & TensorFlow - capitoli 1, 2, 3.
2. *S. Rogers, M. Girolami*: A First Course in Machine Learning - capitoli 1, 2.
3. *C. M. Bishop*: Pattern Recognition and Machine Learning - capitolo 1.
4. *Scikit-Learn*: An introduction to machine learning with Scikit-Learn - *http://scikit-learn.org/stable/tutorial/basic/tutorial.html*
5. *Scikit-Learn*: Preprocessing Data - sezione 4.3 (*http://scikit-learn.org/stable/modules/preprocessing.html*)
6. *Wikipedia*: Machine Learning - *https://en.wikipedia.org/wiki/Machine_learning*
7. *Wikipedia*: Supervised Learning - *https://en.wikipedia.org/wiki/ Supervised_learning*
8. *Wikipedia*: Unsupervised Learning - *https://en.wikipedia.org/wiki/ Unsupervised_learning*
9. *Wikipedia*: Batch Learning - *https://en.wikipedia.org/wiki/Batch_learning*
10. *Wikipedia*: Overfitting - *https://en.wikipedia.org/wiki/Overfitting*
11. *Wikipedia*: Underfitting - *https://en.wikipedia.org/wiki/Underfitting*
12. *Wikipedia*: Cross Validation - *https://en.wikipedia.org/wiki/Cross-validation_(statistics)*

2 Preparazione dei Dati

riferimenti sezione precedente, in particolare 1, 2, 3, 8, 9.

1. *Scikit-Learn*: Dataset Transformations - *http://scikit-learn.org/stable/data_transforms.html*

3 Allenamento degli Algoritmi

1. *Aurélien Géron*: Hands-On Machine Learning with Scikit-Learn & TensorFlow - capitolo 6, *Decision Trees*

2. *Aurélien Géron*: Hands-On Machine Learning with Scikit-Learn & TensorFlow - capitolo 7, *Ensemble Learning and Random Forest*

3. *Wikipedia*: Decision Tree - *https://en.wikipedia.org/wiki/Decision_tree*

4. *Wikipedia*: Random Forest - *https://en.wikipedia.org/wiki/Random_forest*

5. *Scikit-Learn*: RandomForestClassifier - *http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier*

6. *Aurélien Géron*: Hands-On Machine Learning with Scikit-Learn & TensorFlow - capitolo 5, *Support Vector Machines*

7. *Wikipedia*: Support Vector Machines - *https://en.wikipedia.org/wiki/Support_vector_machines*

8. *Scikit-Learn*: Support Vector Machines - *http://scikit-learn.org/stable/modules/svm.html*

9. *Aurélien Géron*: Hands-On Machine Learning with Scikit-Learn & TensorFlow - capitolo 10, *Introduction to Artificial Neural Networks*

10. *Aurélien Géron*: Hands-On Machine Learning with Scikit-Learn & TensorFlow - capitolo 11, *Training Deep Neural Nets*

11. *Wikipedia*: Biological Neural Network - *https://en.wikipedia.org/wiki/Biological_neural_network*

12. *Wikipedia*: Artificial Neuron - *https://en.wikipedia.org/ wiki/Artificial_neuron*

13. *Wikipedia*: Artificial Neural Network - *https://en.wikipedia.org/wiki/Artificial_neural_network*

14. *TensorFlow*: DeepNeuralNetworkClassifier - *https://www.tensorflow.org/api_docs/python/tf/contrib/learn/DNNClassifier*