Fault Diagnosis del Tennessee Eastman Process tramite reti neurali, tecniche di machine learning ed algoritmi statistici.

Corso di Manutenzione Preventiva per la Robotica e l'automazione intelligente

- Prof. Alessandro Freddi
- Dr. Francesco Ferracuti

Studenti:

- D'Agostino Lorenzo
- Lanciotti Antonio
- Olivieri Davide

Abstract

Nella relazione seguente verranno descritti i passaggi e gli script sviluppati in linguaggio **MATLAB** (R2021b) per lo sviluppo di processi di fault diagnosis del banchmark Tennessee Eastman Process.

In primo luogo sono state utilizzate tecniche di machine learning e di deep learning per l'identificazione analizzando e processando i dati come "*serie temporali*". A tal proposito sono state sviluppate reti di tipo *LSTM* i cui risultati sono stati quelli ottenuti tramite l'utilizzo di *Alberi Decisionali* ed algoritmi *Ensamble* secondo le linee guida descritte nel seguente link.

In secondo luogo sono stati testati *algoritmi di stampo statistico* adattati al **dataset** open source ottenuto tramite il modello *Simulink* del processo.

Infine sono stati utilizzati gli output generati dall'algoritmo di sopra per addestrare un classificatore.

Dataset

Il dataset in questione è composto di 4 file .RData (convertiti in .CSV) contenenti le simulazioni effettuate sul modello del processo. In particolare i file sono organizzati come segue:

- **TEP_FaultFree_Training.csv**: contiene le simulazioni fault free da utilizzare in fase di train,
- **TEP_FaultFree_Testing.csv**: contenente i dati di testing fault free,
- **TEP_Faulty_Training.csv**: contenete le simulazioni per ogni classe di fault da utilizzare in fase di training,
- **Tep_Faulty_Testing.csv**: contenente i dati di test faulty.

Le tabelle contenute all'interno del dataset contengono dati circa **52 sensori** oltre che informazioni circa la simulazione e la classe di fault associata:

- **faultNumber**: da 0 a 20 (**0 → fault free**),
- **simulationRun**: indice della simulazione,
- **sample**: indice delle misurazioni associate alla *simulationRun*.

L'intervallo temporale che intercorre tra un *sample* e l'altro è di 3 minuti.

È importante tenere presente che nelle simulazioni faulty il fault viene introdotto dopo un periodo di tempo descritto nel medium.

Approccio con reti neurali LSTM

Come anticipato sono state utilizzate reti neurali LSTM, particolari tipi di reti ricorrenti molto utilizzate nel forecasting, classificazione e nella data imputation di serie temporali, per cui adatte al nostro scopo.

Preprocessing

Facendo riferimento al file "*./Reti&ML/train_completo.m*" il dataset viene processato in primo luogo filtrando i primi 20 *sample* (1 ora) di ogni *simulationRun* al fine di eliminare la fase di inizializzazione in cui i fault non sono presenti. Questo filtraggio è stato effettuato anche per il dataset **fault free** al fine di mantenere il dataset totale il più bilanciato possibile.

```
training_data = [readtable("TEP_FaultFree_Training.csv");
readtable("TEP_Faulty_Training.csv")];
training_data.(1) = [];
training_data = training_data(training_data.sample > 20, :);
```

Per addestrare le reti sono stat utilizzate delle finestre temporali anche qui di 1 ora (20 campioni). Le reti neurali in MATLAB utilizzano tensori di dimensione **f**x**l**x**n** dove **f** rappresenta il numero di features ed è **l** la dimensione della finestra temporale considerata.

```
%% Costruzione dataset di train
seq_len = 20;
n_features = 52;

% Creo il vettore delle labels
training_labels = categorical(training_data.faultNumber(mod(training_data.sample,seq_len)==1));

% Costruisco il tensore di training
training_data = squeeze(num2cell(reshape(table2array(training_data(:, 4:end))', 52 ,seq_len,
[]),[1 2]));
```

Rete neurale LSTM

La rete neurale è stata sviluppata utilizzando il **[Deep Learning Toolbox](Deep Learning Toolbox - MATLAB)** messo a disposizione all'interno dell'ambiente MATLAB. La rete reurale utilizzata è la seguente:

```
model = [
    sequenceInputLayer(n_features, "Normalization", 'zscore')    % rescale-zero-one, zscore
    lstmLayer(128,'OutputMode','sequence', 'GateActivationFunction','sigmoid')
    lstmLayer(128,'OutputMode','last', 'GateActivationFunction','sigmoid')
    fullyConnectedLayer(300)
    dropoutLayer(0.5)
    fullyConnectedLayer(128)
    dropoutLayer(0.8)
    batchNormalizationLayer
    fullyConnectedLayer(21)
    softmaxLayer
    classificationLayer];
```

Come mostrato, il primo layer effettua la normalizzazione automatica del dataset, mentre l'ultimo (**fullyConnectedLayer(21)**), seguito dalla funzione di attivazione *softmax*, esegue la classificazione.

Eseguendo lo script viene effettuato automaticamente il training della rete ed il salvataggio della rete addestrata.

Testing

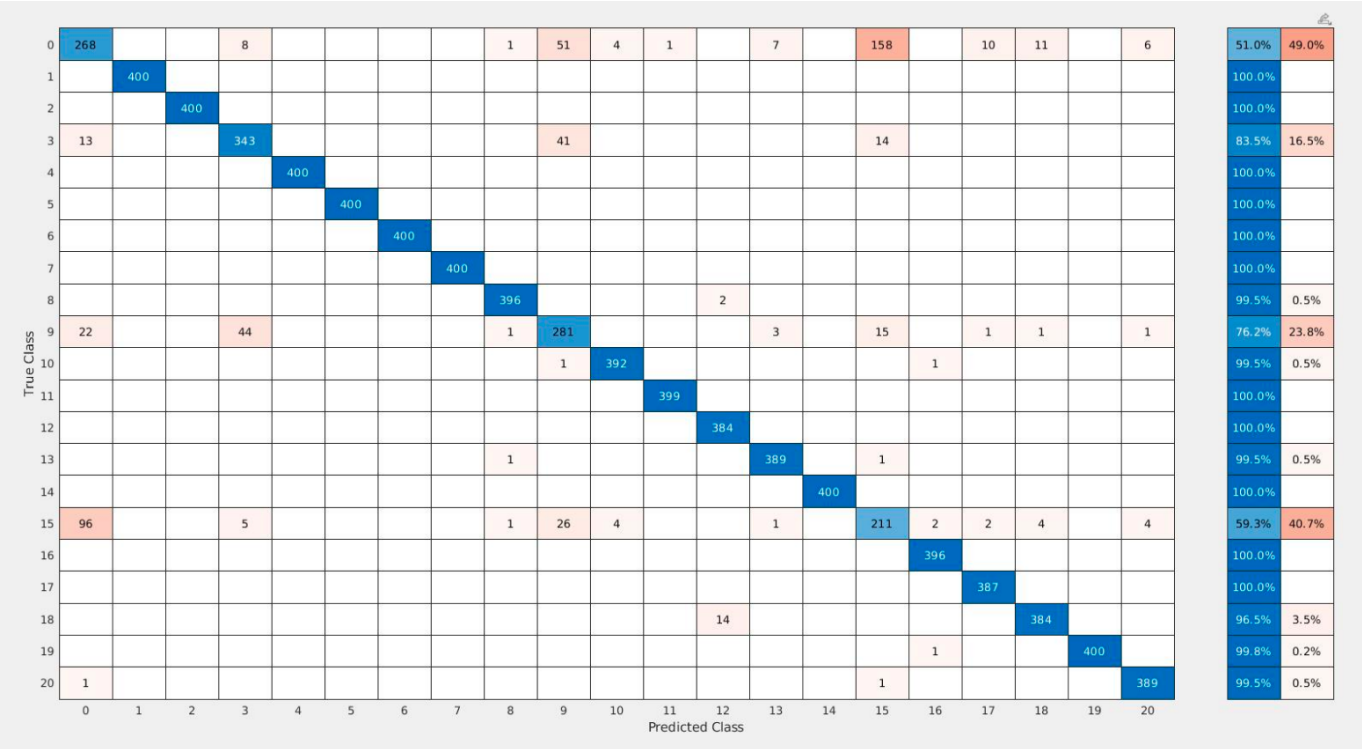
Facendo riferimento al file *./Reti&ML/test_processing.m*, in questo caso i dati vengono caricati all'interno dell'ambiente di programmazione utilizzando delle particolari strutture dati di tipo busy (Datastore) che non effettuano subito il load dei dati, ma solo su richiesta. In più permettono di caricare all'interno della RAM solo dei chunck di dimensione predefinita dei dati. Questa scelta è stata necessaria per lo sviluppo del codice sui nostri portatili data la dimensionalità del dataset di test.

Questo script carica automaticamente la rete neurale addestrata (*vedi sopra*) ed effettua previsioni per poi graficare la matrice di confusione associata. Oltre a questo viene anche salvato il dataset generato:

```
%% Saving dataset
dataset_test = [tot_data_test, tot_label_test];
save("dataset_test", "dataset_test");

%% Load networks
net = load("TNetworks/tnet-zscore.mat");
net = net.trained_model;
net.Layers


%% Make predictions and confusion chart
num_el = size(tot_data_test, 1);
predictions_rows = predict(net, tot_data_test);
predictions = zeros(num_el, 1);
for row_idx = 1:num_el
    [a, idx] = max(predictions_rows(row_idx, :));
    predictions(row_idx) = idx - 1;
end

%% Confusion chart
cm = confusionchart(predictions, table2array(cell2table(tot_label_test)));
cm.RowSummary = 'row-normalized';
```

Risultati

Riportiamo i risultati effettuati sul dataset di test sopra descritto:



Come possiamo vedere i risultati ottenuti sono del tutto conformi a quelli riportati nel *medium*. Possiamo ritenerci soddisfatti del funzionamento della rete neurale che riporta un'**accuratezza del 93%**. In particolare possiamo notare come la rete abbia però difficoltà nella classificazione di dati provenienti da simulazioni fault free. Questo può essere un problema in fase di produzione perchè porterebbe alla frequente generazione di falsi allarmi.

Sarebbe possibile ridurre questo effetto complicando la rete neurale sviluppata o, più semplicemente, addestrare un predittore in grado di distinguere le classi **Faulty** e **Fault Free** e, nel caso **Faulty** andare a classificare il fault.

Approccio Decision Tree e Random Forest

L'addestramento dei modelli decisionali di tipo **Decision Tree / Random Forest** è stato effettuato mediante l'utilizzo del plug-in MATLAB chiamato **Classification Learner**.

Preprocessing

In questo caso il processing del dataset risulta essere molto più semplice in quanto non vengono valutate serie temporali ma i sample vengono classificati prendendo in considerazione un'unica misurazione.

Facendo riferimento al file "*./Reti&ML/data_processing_for_RF.m*" sono state prese in considerazione unicamente le prime 40 simulazioni per quanto riguarda il dataset fault free, e le prime 25 simulazioni per il dataset faulty, al fine di replicare i risultati riportati nel *medium*:

```
load train_dataset.mat;

%% Filter dataset
condition = data.simulationRun <= 40 & data.faultNumber == 0 ...
    | data.simulationRun <= 25 & data.faultNumber ~= 0;
data = data(condition, :);

%% Normalize columns
data = normalize(data, "zscore", ...
    "DataVariables", data.Properties.VariableNames(4:end));
```

Va notato che il file contenente i dati "*train_dataset.mat*" deve essere generato come descritto nella sezione precedente. Esso dovrà contenere al suo interno entrambe le tabelle dei dati faulty e fault free, filtrate dei primi 20 sample di ogni simulation run, al fine di eliminare dai dati faulty la fase di inizializzazione del modello simulink in cui esso evolve senza la presenza del fault considerato.

Classification Learner

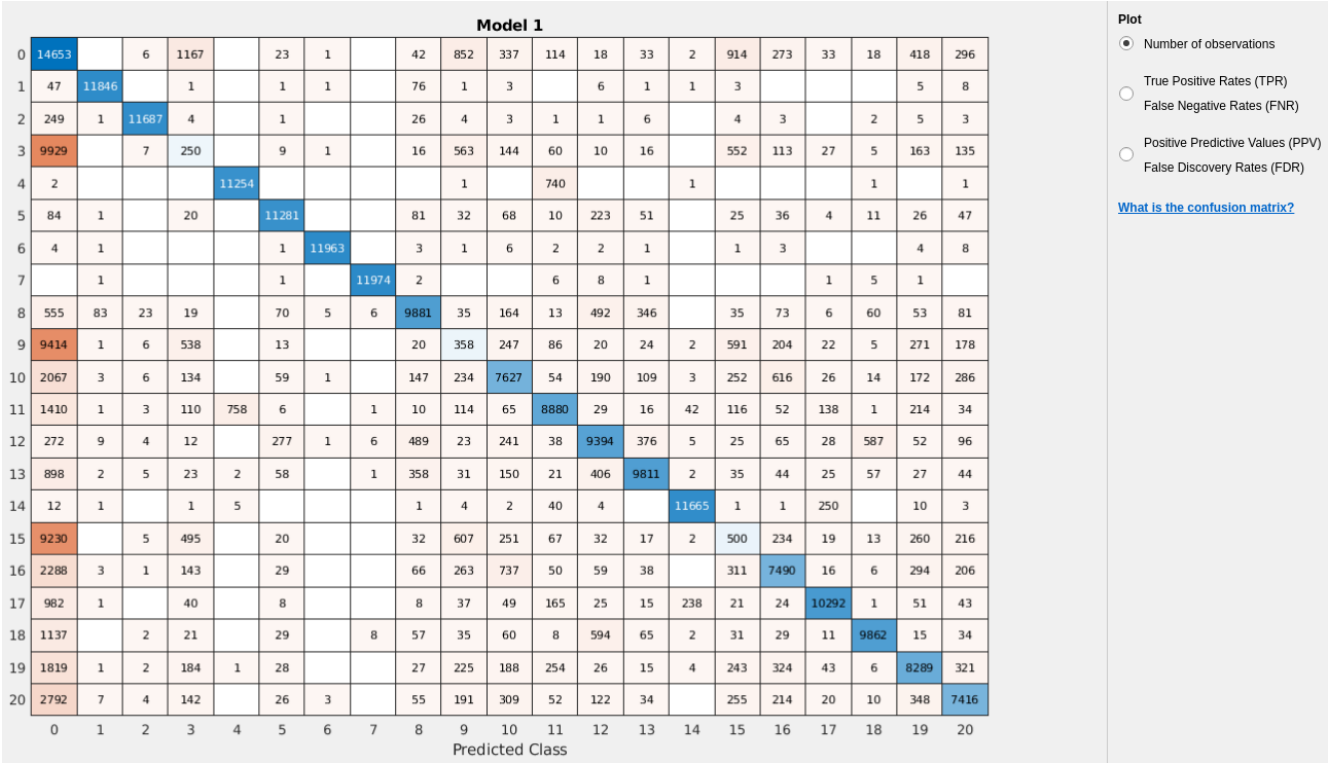
Nel classification learner è possibile effettuare un'ottimizzazione degli iperparamtri. Questa tecnica esegue in maniera automatica l'addestramento di più modelli dello stesso tipo facendo variare gli iperparametri utilizzati dal modello stesso.

Purtroppo questa tecnica risulta essere molto onerosa in termini computazionali per cui è stata effettuata solo nel caso del semplice **Decision Tree**.

L'addestramento del decisore è anch'esso effettuato in maniera automatica dal classification learner, riportiamo per cui i risultati ottenuti:

Optimized Decision Tree

I risultati ottenuti dall'ottimizzazione del decision tree sono riportati nelle immagini seguenti:



Training Results

Accuracy (Validation)	71.9%
Total cost (Validation)	72827
Prediction speed	~640000 obs/sec
Training time	1072 sec

Model Type

Preset: Optimizable Tree
Surrogate decision splits: Off

Optimized Hyperparameters

Maximum number of splits: 6710
Split criterion: Maximum deviance reduction

Hyperparameter Search Range

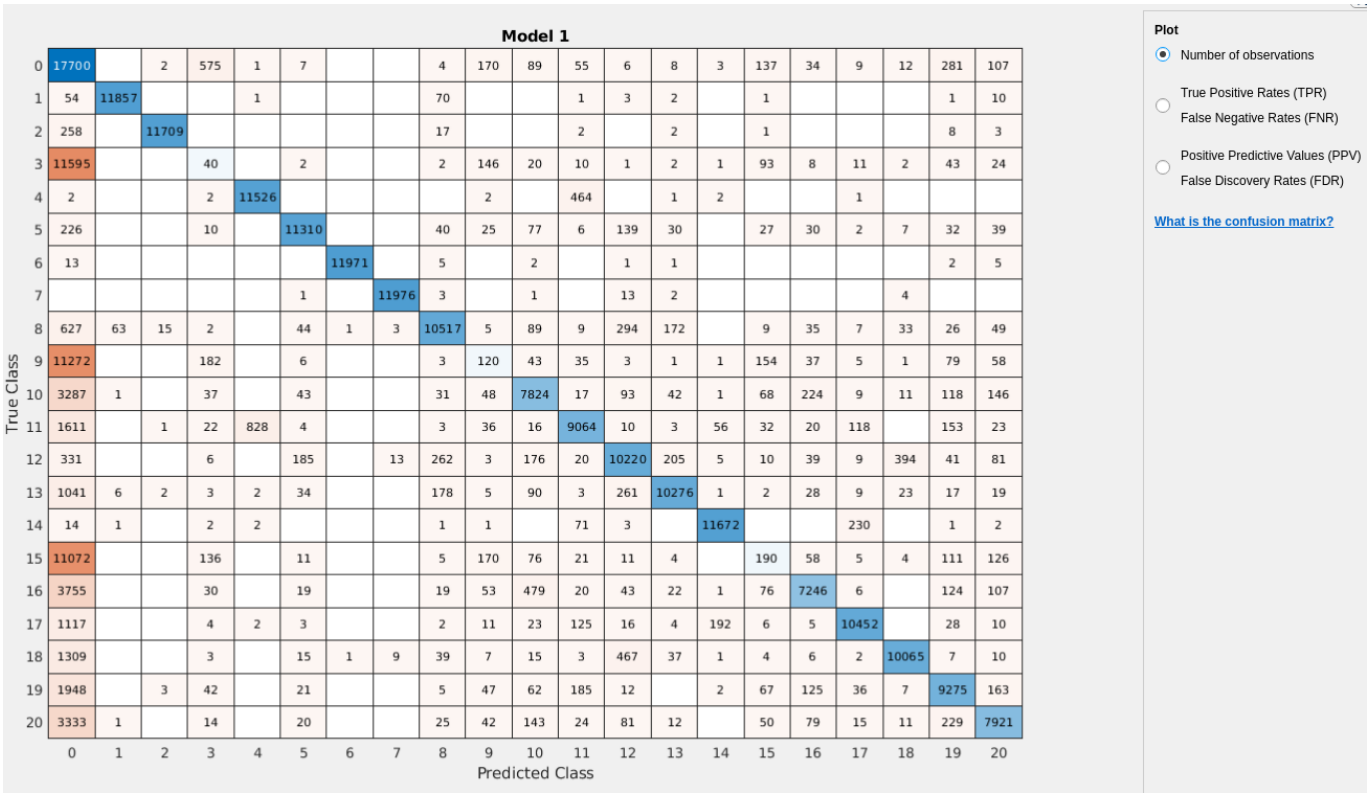
Maximum number of splits: 1-259199
Split criterion: Gini's diversity index, Twoing rule, Maximum deviance reduction

Optimizer Options

Optimizer: Bayesian optimization
Acquisition function: Expected improvement per second plus
Iterations: 30
Training time limit: false

Random Forest

I risultati ottenuti dalla Random Forest sono riportati nelle immagini seguenti:



Training Results

Accuracy (Validation)	74.4%
Total cost (Validation)	66269
Prediction speed	~42000 obs/sec
Training time	1371 sec

Model Type

Preset: Boosted Trees
Ensemble method: AdaBoost
Learner type: Decision tree
Maximum number of splits: 6710
Number of learners: 30
Learning rate: 0.1

Optimizer Options

Hyperparameter options disabled

Approccio statistico

La metodologia che andremo a descrivere nel paragrafo seguente basa il suo funzionamento sulla determinazione di una **Funzione di densità di probabilità (PDF)** a partire dalle misurazioni effettuate.

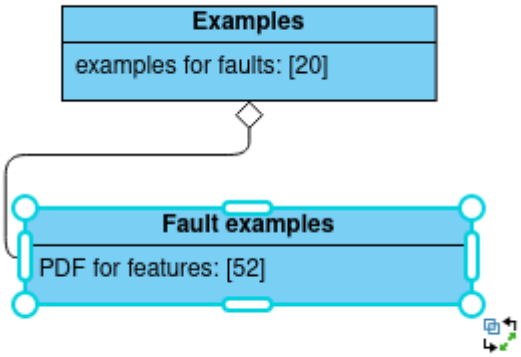
L'idea di base è quella di confrontare, tramite opportune distanze, una *PDF* presa come riferimento, con le altre calcolate a runtime. Tramite queste distanze "*statistiche*" è possibile classificare le diverse classi di guasto senza l'utilizzo di un modello addestrato che può risultare computazionalmente oneroso.

Adattamento dell'algoritmo

L'algoritmo fornitoci utilizza delle finestre temporali per poter creare la PDF all'interno di un dataset ad una variabile. È stato per cui adattato alle dimensionalità del dataset del TEP, generando una PDF per ogni feature.

L'idea di partenza è stata quella di creare quindi una struttura dati che contenesse le PDF per ogni feature di ogni fault, al fine di poter confrontare le serie temporali misurate a runtime con ognuna di queste.

La struttura dati risultante è la seguente:



Per ogni fault vengono generate quindi 52 PDF, una per ogni feature contenuta nel dataset. In MATLAB la struttura dati riportata sopra è stata realizzata tramite "**cellArray**".

Calcolo delle distanze e classificazione

Utilizzando una struttura dati complessa, in particolare quella descritta dall'immagine di sopra, si avranno un totale di distanze elevato (in questo caso $52 * 21 = 1092$). Nasce quindi il problema della definizione di un algoritmo per la classificazione del fault.

In particolare, date due PDF, indichiamo la distanza tra le due come $D(\mathbf{pdf}_i, \mathbf{pdf}_k)$. A questo punto passiamo a definire la distanza tra due insiemi di PDF (IMG_i, IMG_j) come:

$$D(IMG_i,\ IMG_j) = \frac{1}{N} \sum_{k=1}^{N} D(IMG_i(k),\ IMG_j(k))$$

dove $IMG_i = [PDF_i |\ i = 1, \ldots, N\_features]$ è l'insieme delle funzioni di probabilità associate ad uno specifico fault, e $IMG_i(k) = \mathbf{pdf}_k$ è la funzione densità di probabilità associata alla feature k-esima dell'i-esimo fault.

A questo punto definito l'insieme degli esempi per ogni fault come

$$E = [IMG_i,\ i = 0, \ldots,\ n\_faults]$$

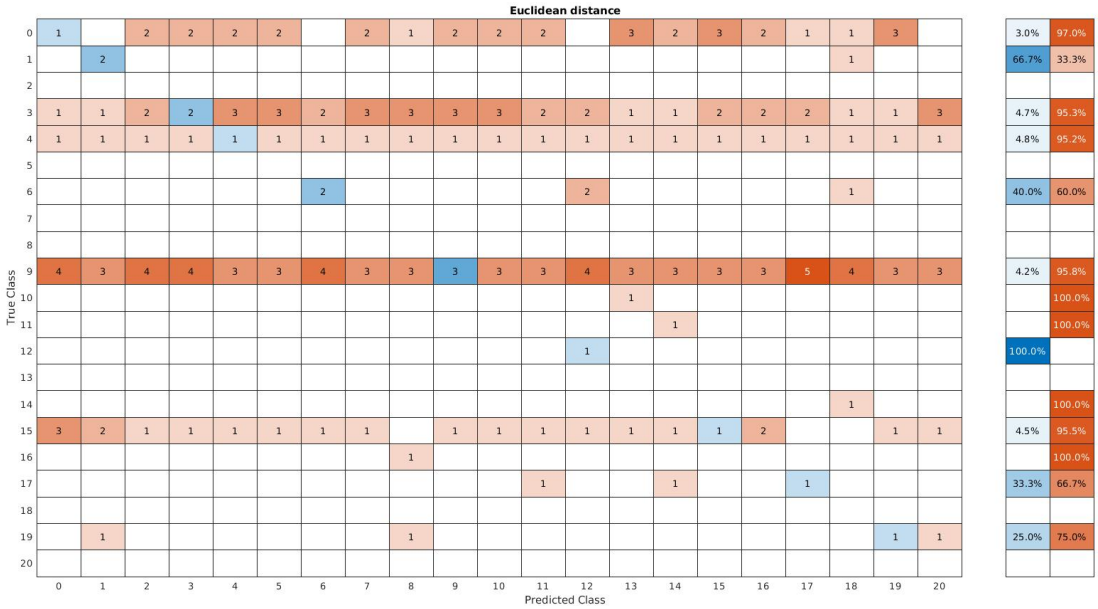
l'insieme delle immagini di riferimento, e $RunIMG$ l'insieme delle PDF calcolate a runtime **che si vuole analizzare**.

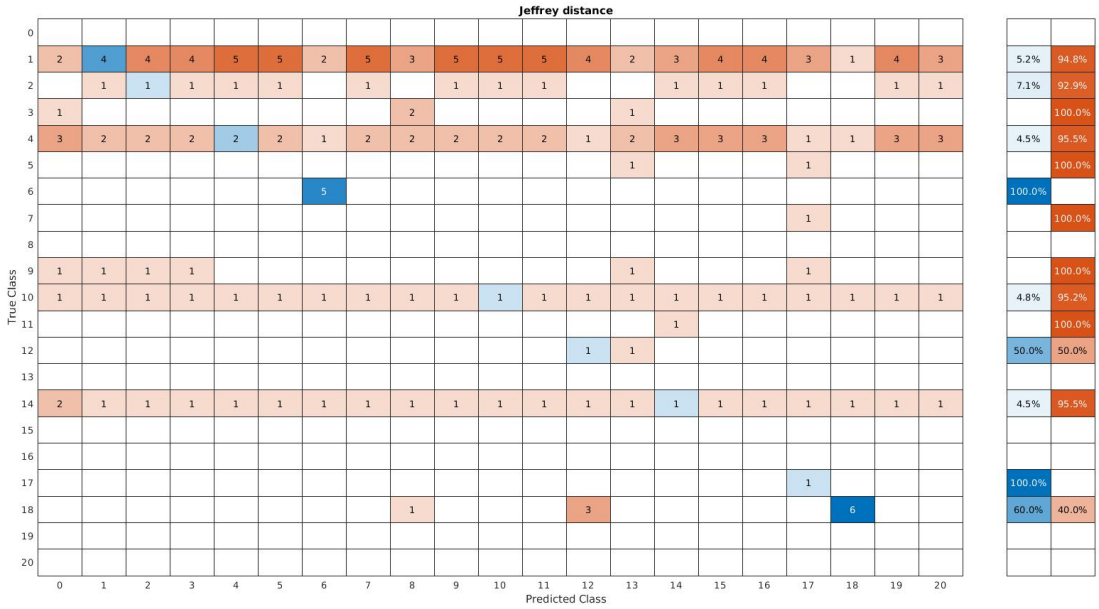
La classificazione viene effettuata tramite la seguente:

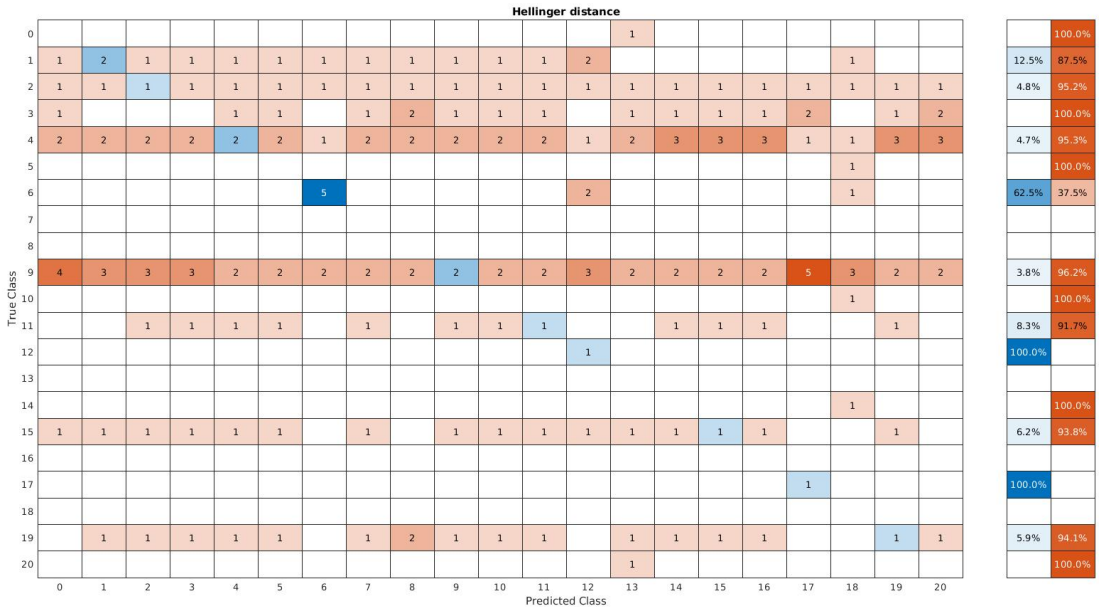
$$C(RunImg) = argmin(\ D(E,\ RunIMG)\ )$$

con $D(E,\ RunIMG) = [D(IMG, RunIMG),\ \forall\ IMG \in E]$

Il file "*./DistanzeStatistiche/trainImageBuild.m*" implementa quanto descritto sopra e dopo aver eseguito il processing dei dati, effettua il tesing e mostra i risultati della classificazione che riportiamo nelle figure sottostanti (sono state utilizzate le tre tipologie di distanze utilizzate dagli autori dell'algoritmo):







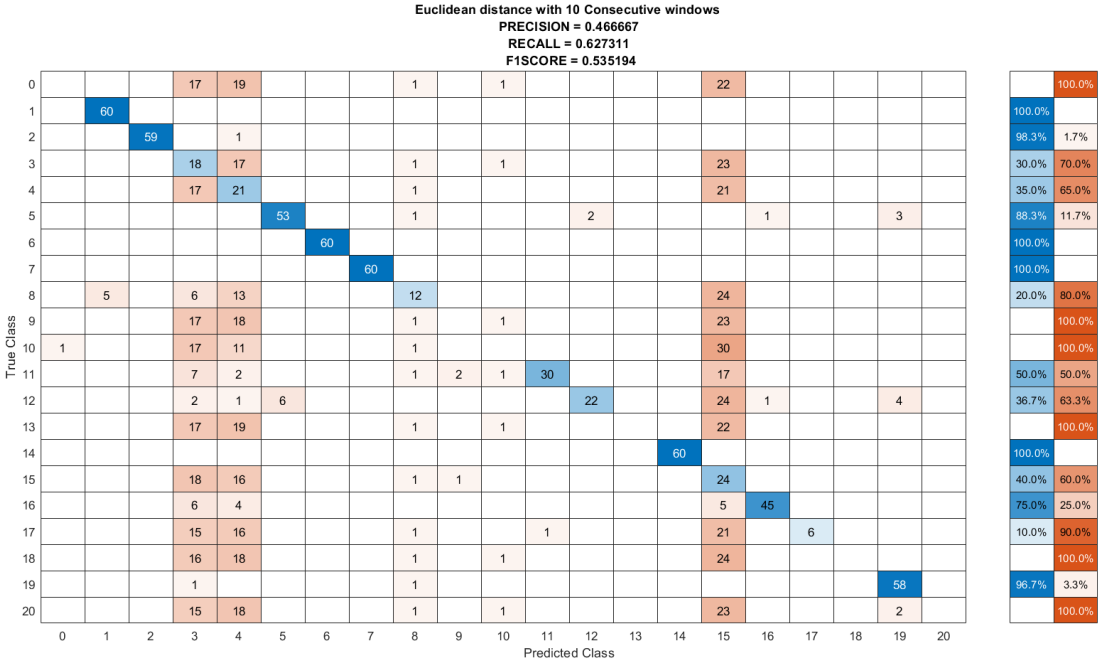
Questo primo approccio costruisce le PDF in forma matriciale e, a causa degli scarsi risultati ottenuti, siamo passati ad una struttura dati di tipo vettoriale.

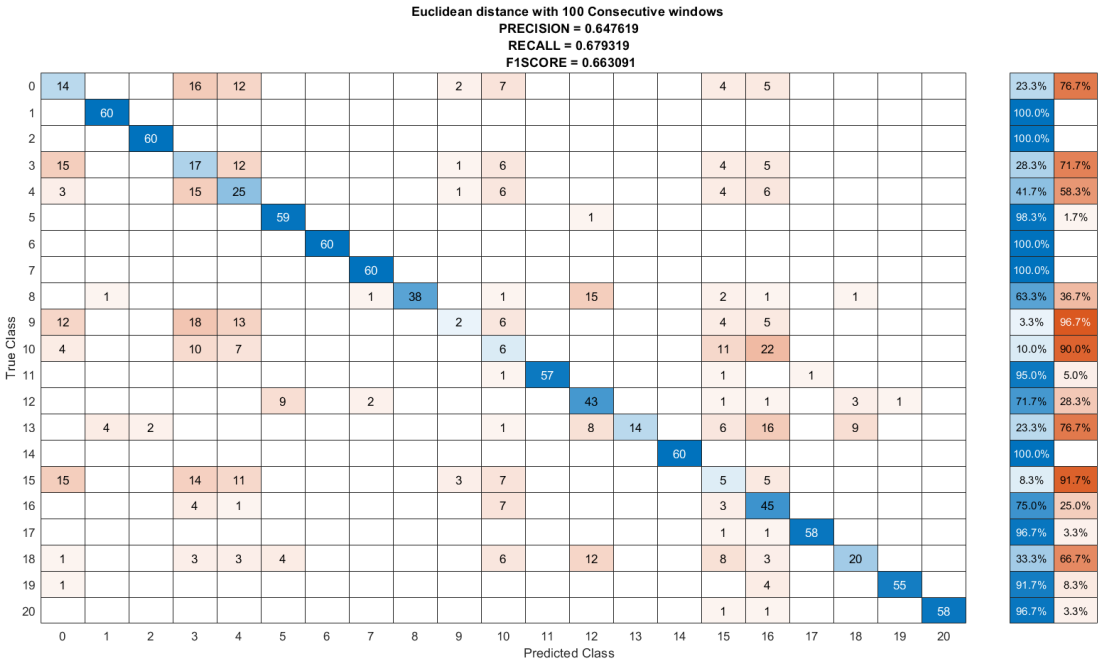
La costruzione delle nuove immagini viene affrontata nel file "*./DistanzeStatistiche/statistical_distance.m*" in cui l'unica differenza significativa con lo script precedente è quella che le matrici dei dati vengono "**flattate**" prima di essere ordinati al fine di ottenere la PDF corrispondente (riga 112) in forma vettoriale.

```
Imagetrain = normalize(sort(Imagetrain,2,"descend"), 2, "norm")
```

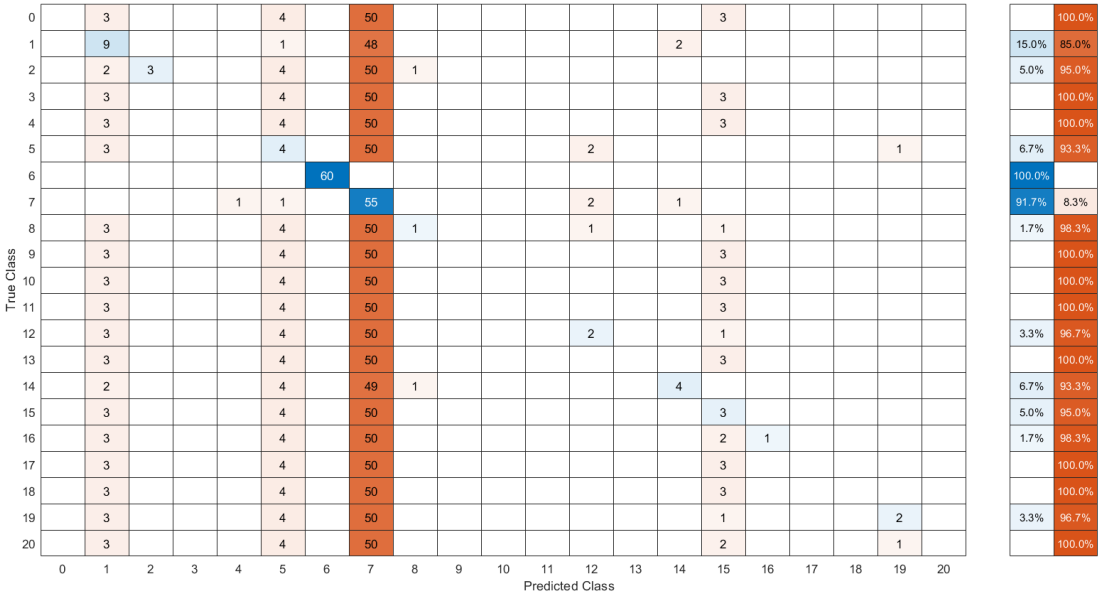
Questa metodologia ha portato ad un significativo miglioramento nelle prestazioni dell'algoritmo.

Di seguito riportiamo i risultati ottenuti al variare della numerosità delle finestre temporali considerate per la costruzione delle PDF:

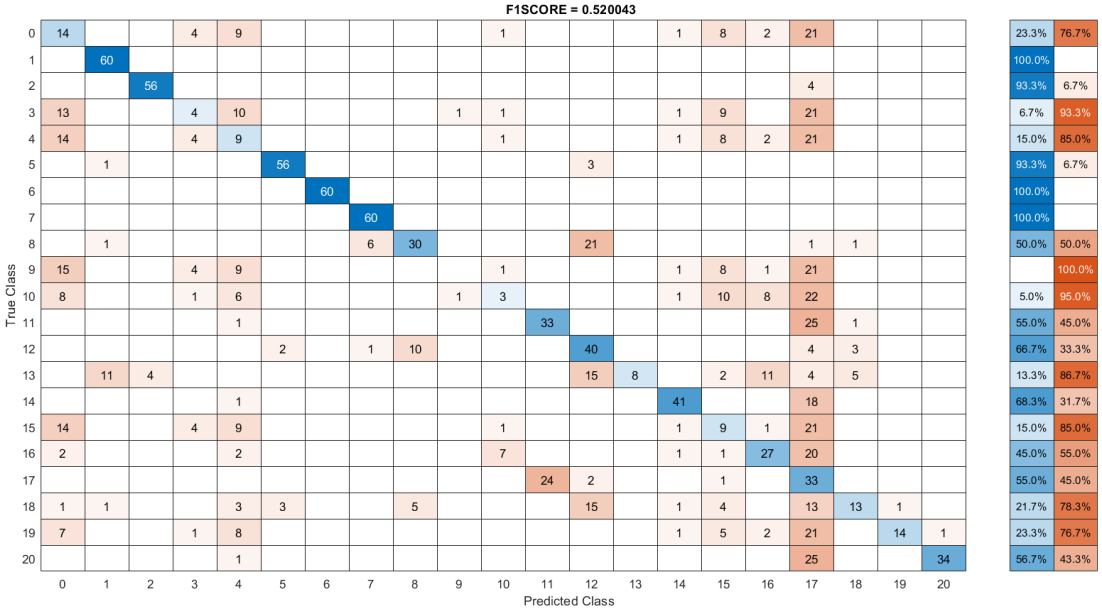


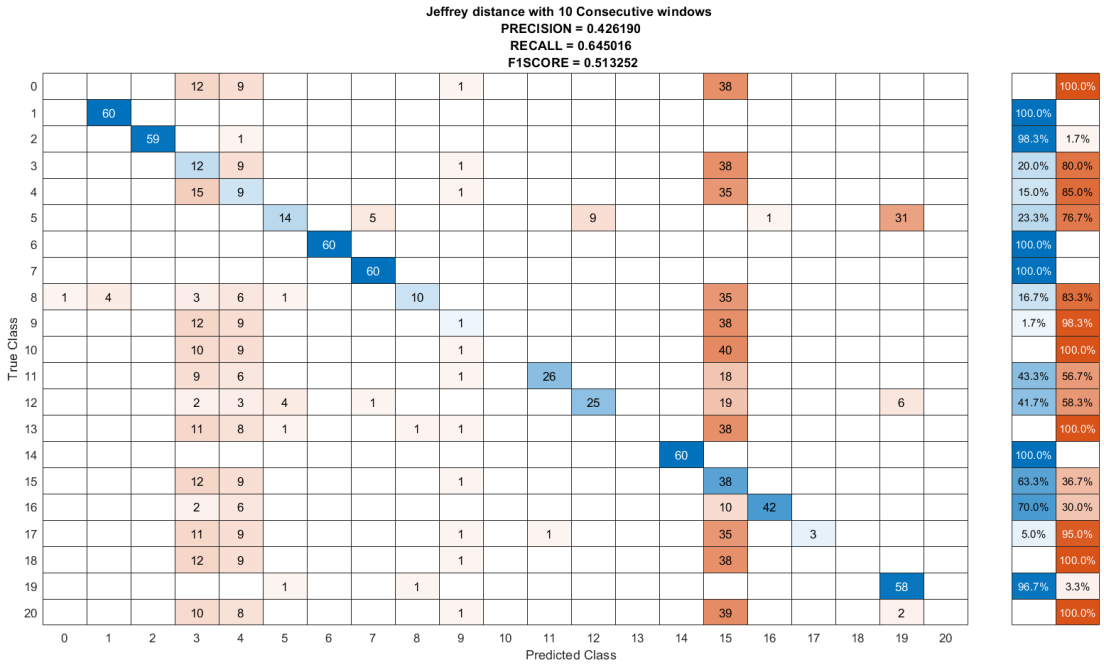


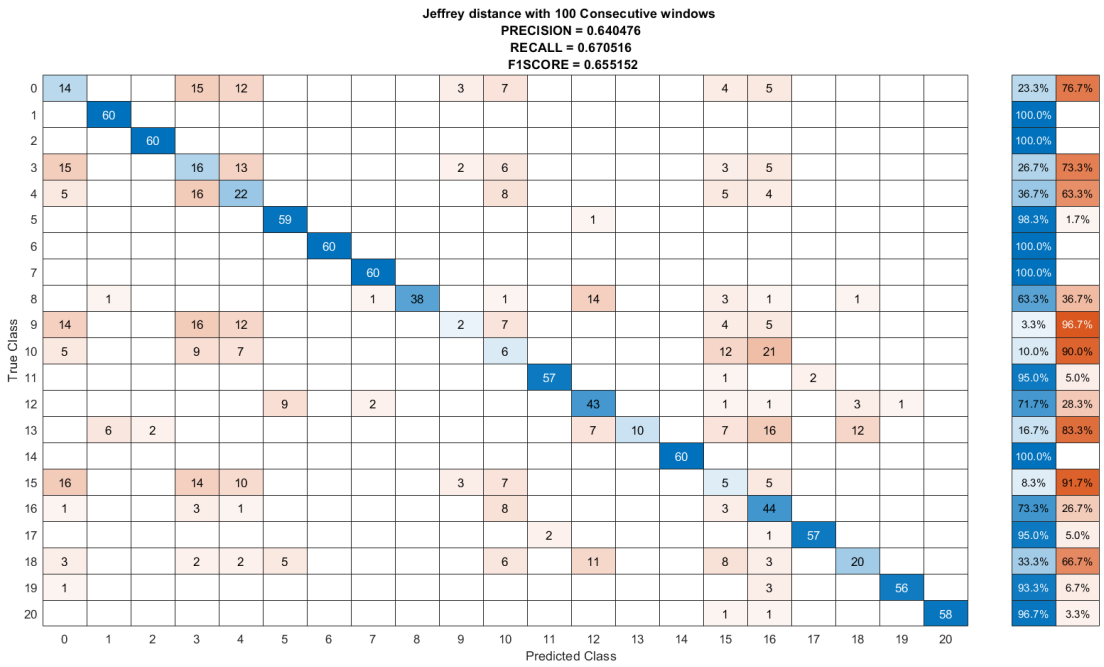
Hellinger distance with 10 Consecutive windows
PRECISION = 0.114286
RECALL = 0.419004
F1SCORE = 0.179588



Hellinger distance with 100 Consecutive windows
PRECISION = 0.479365
RECALL = 0.568264
F1SCORE = 0.520043







Come possiamo vedere dalle immagini e dalle metriche riportate nelle figure, l'algoritmo risulta essere particolarmente sensibile alla numerosità delle finestre temporali utilizzate per il calcolo delle PDF.

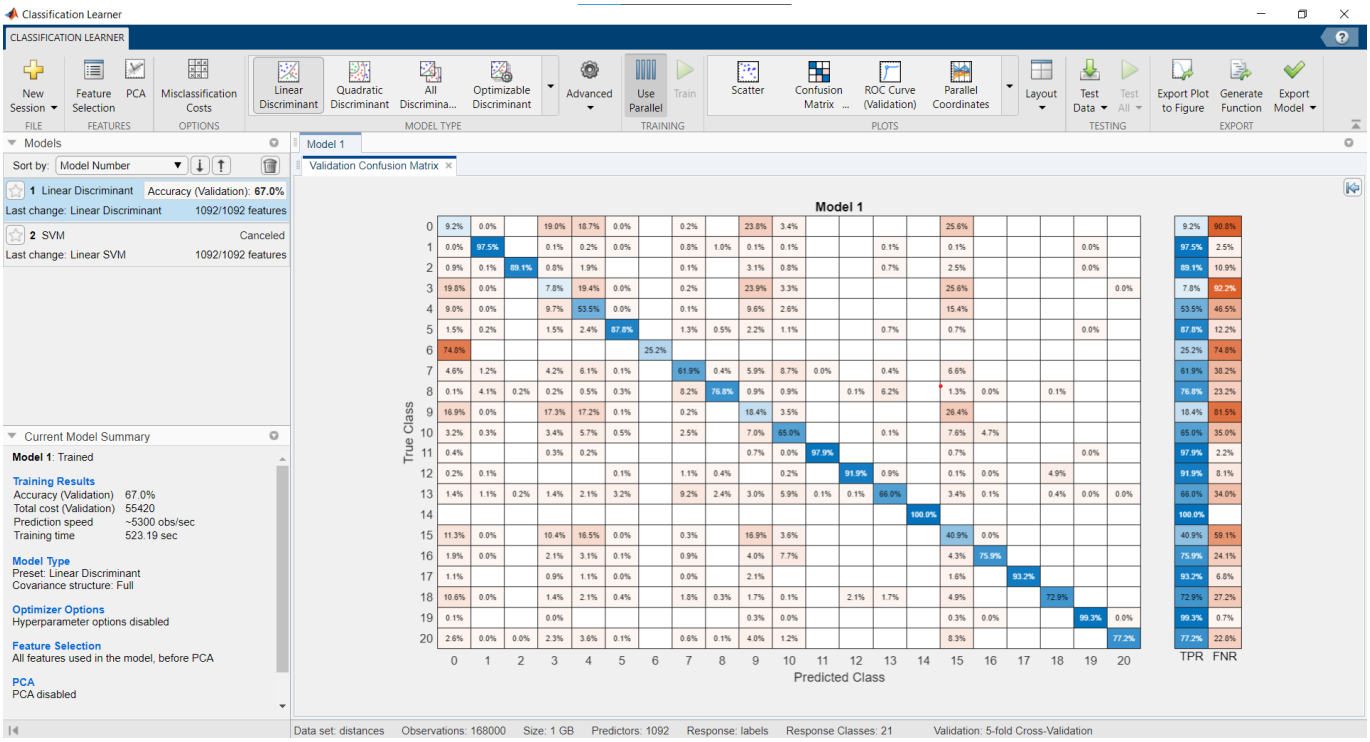
Un numero maggiore di finestre temporali implica un ritardo maggiore nella classificazione in tempo reale, in particolare essendo questo dataset campionato a 3 minuti, 100 finestre corrispondono ad un ritardo di 300 minuti (5 ore) nella classificazione rispetto al campionamento.

Approccio Combinato

A questo punto è stato deciso di utilizzare un approccio combinato, addestrando un classificatore utilizzando le distanze statistiche come features caratteristiche.

Il file di riferimento per la generazione del dataset di addestramento è "./DistanzeStatistiche/dataset_generation_distances.m*.

I risultati seguenti sono stati ottenuti addestrando un **classificatore lineare**.



Abbiamo tentato di addestrare una **SVM** ma le tempistiche erano eccessive. Il dataset generato è stato esportato nel file "./DistanzeStatistiche/distance_dataset.mat" per cui è possibile effettuare il training dei modelli anche effettuando una ottimizzazione degli iperparametri.

Il file "./DistanzeStatistiche/distance_dataset.mat" contiene:

- **distances**: matrice 168000x1092 dei dati,
- **labels**: vettore di 168000 contenente i fault associati ad ogni riga del dataset.