Classificazione di fiori di Iris nelle diverse specie

July 5, 2019

1 Classificazione di fiori di Iris nelle diverse specie

Progetto per il corso di Programmazione di Applicazioni Data Intensive
Laurea in Ingegneria e Scienze Informatiche, Università di Bologna
Lorenzo Paganelli
lorenzo.paganelli3@studio.unibo.it

1.1 Introduzione

Iris è un genere di piante della famiglia delle Iridaceae, che comprende oltre 300 specie. Noi prendiamo in considerazione unicamente tre specie: **Iris virginica**, **Iris setosa** e **Iris versicolor**, ciò che vogliamo fare è classificare i fiori di iris nelle diverse specie menzionate, sulla base di un insieme di caratteristiche. In particolare vengono considerate quattro caratteristiche per ciascun fiore: - `SepalLengthCm`: Lunghezza del sepalo in cm - `SepalWidthCm`: Larghezza del sepalo in cm - `PetalLengthCm`: Lunghezza del petalo in cm - `PetalWidthCm`: Larghezza del petalo in cm

Si prendono cioè in considerazione due tratti caratteristici del fiore: il **sepalo** e il **petalo**, per ciascuno di questi si conosce la lunghezza e la larghezza. Si sappia che il **sepalo** è una foglia modificata che fa parte del calice del fiore

1.2 Analisi esplorativa

Importiamo le librerie necessarie e scarichiamo il dataset, se non è già presente

```
In [1]: %matplotlib inline
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

```
In [2]: import os.path
        if not os.path.exists("Iris.csv"):
            from urllib.request import urlretrieve
            urlretrieve("https://bitbucket.org/paagamelo/iris-species/downloads/Iris.csv", "Iri
```

Il dataset viene convertito in un DataFrame pandas, per renderlo meglio fruibile

```
In [3]: iris = pd.read_csv("Iris.csv", index_col="Id", dtype={"Species": "category"})
```

Osserviamo un estratto del DataFrame: questo ha 5 colonne, come intuibile, contenenti le quattro caratteristiche menzionate prima e l'informazione sulla specie di ciascuna osservazione

```
In [4]: iris.head(3)
```

```
Out[4]:     SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm      Species
        Id
        1             5.1           3.5            1.4           0.2  Iris-setosa
        2             4.9           3.0            1.4           0.2  Iris-setosa
        3             4.7           3.2            1.3           0.2  Iris-setosa
```

Osserviamo ora la *forma* del DataFrame: disponiamo di 150 osservazioni in tutto, quindi il dataset è piuttosto ridotto. Tipicamente i modelli di conoscenza allenati su insiemi di dati ridotti tendono ad andare facilmente in overfitting, questo perchè meno dati ci sono tanti più sono i modelli che riescono a descrivere lo stesso fenomeno

```
In [5]: iris.shape
```

```
Out[5]: (150, 5)
```

Verifichiamo la presenza di valori mancanti, in questo caso non ce ne sono

```
In [6]: iris.isnull().any()
```

```
Out[6]: SepalLengthCm    False
        SepalWidthCm     False
        PetalLengthCm    False
        PetalWidthCm     False
        Species          False
        dtype: bool
```

Con il metodo `describe()` otteniamo una serie di statistiche sulle colonne numeriche del DataFrame, cioè tutte a parte quella relativa alla specie

```
In [7]: iris.describe()
```

```
Out[7]:        SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
        count     150.000000    150.000000     150.000000    150.000000
        mean        5.843333      3.054000       3.758667      1.198667
        std         0.828066      0.433594       1.764420      0.763161
        min         4.300000      2.000000       1.000000      0.100000
        25%         5.100000      2.800000       1.600000      0.300000
        50%         5.800000      3.000000       4.350000      1.300000
        75%         6.400000      3.300000       5.100000      1.800000
        max         7.900000      4.400000       6.900000      2.500000
```

Una prima informazione ricavabile è che i dati relativi al petalo risultano molto più *variabili* di quelli relativi al sepalo. Infatti la deviazione standard media delle informazioni sul **sepalo** è:

```
In [8]: iris.describe().loc["std", ["SepalLengthCm", "SepalWidthCm"]].mean()
```

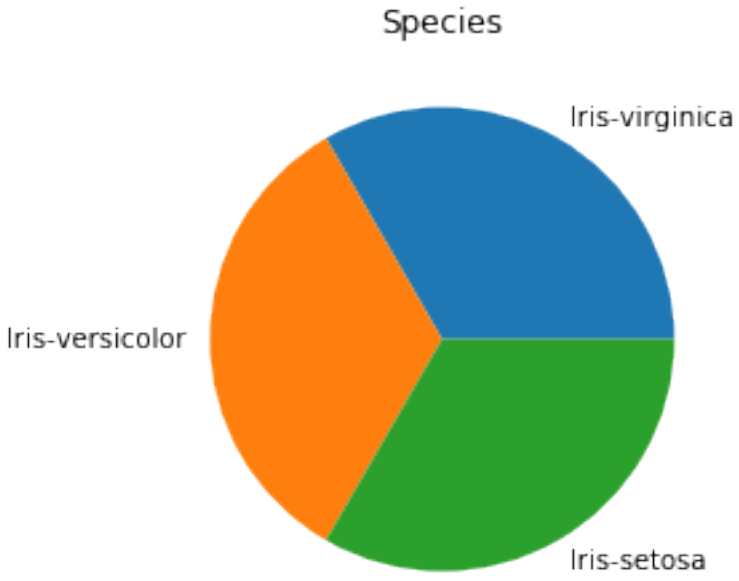
```
Out[8]: 0.6308302196700183
```

Mentre la deviazione standard media delle informazioni sul **petalo** è circa il doppio:

```
In [9]: iris.describe().loc["std", ["PetalLengthCm", "PetalWidthCm"]].mean()

Out[9]: 1.2637905808265515
```

Consideriamo ora la colonna `Species`, mostrando la frequenza delle diverse specie in un grafico a torta. Osserviamo che il dataset è perfettamente bilanciato

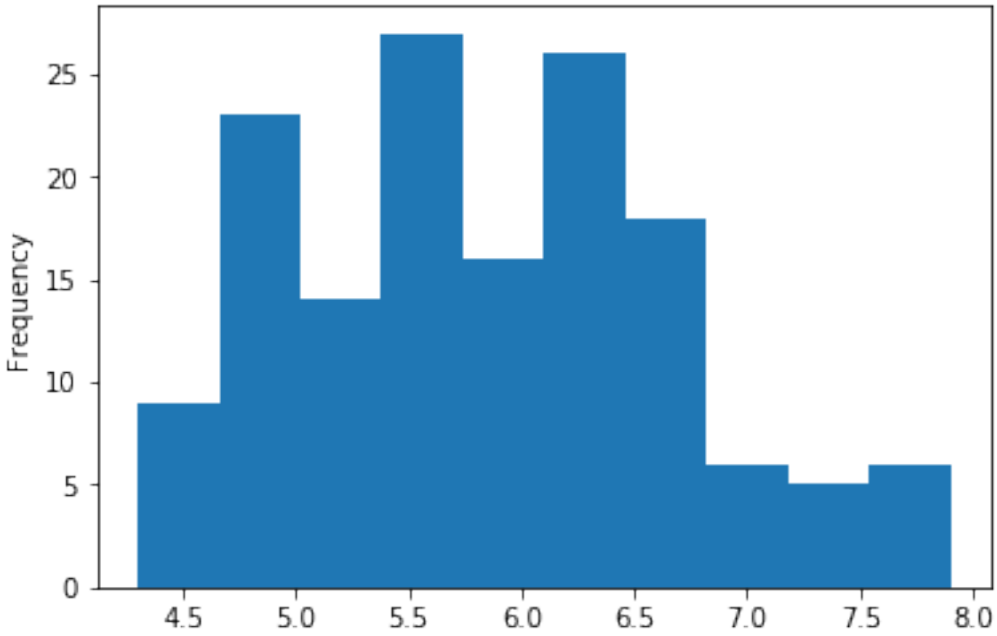
```
In [10]: chart = iris["Species"].value_counts().plot.pie()
         chart.set_title("Species")
         chart.set_ylabel('')

Out[10]: Text(0, 0.5, '')
```



Analizziamo ora le colonne numeriche del DataFrame, visualizzando per ciascuna un istogramma e una raccolta di statistiche ottenibili con il metodo `describe()`

Osserviamo che i dati relativi alla **lunghezza del *sepalo*** presentano tre picchi distinti, potrebbe essere che ciascun picco sia relativo a una delle tre specie. In realtà, come vedremo poi, non è così

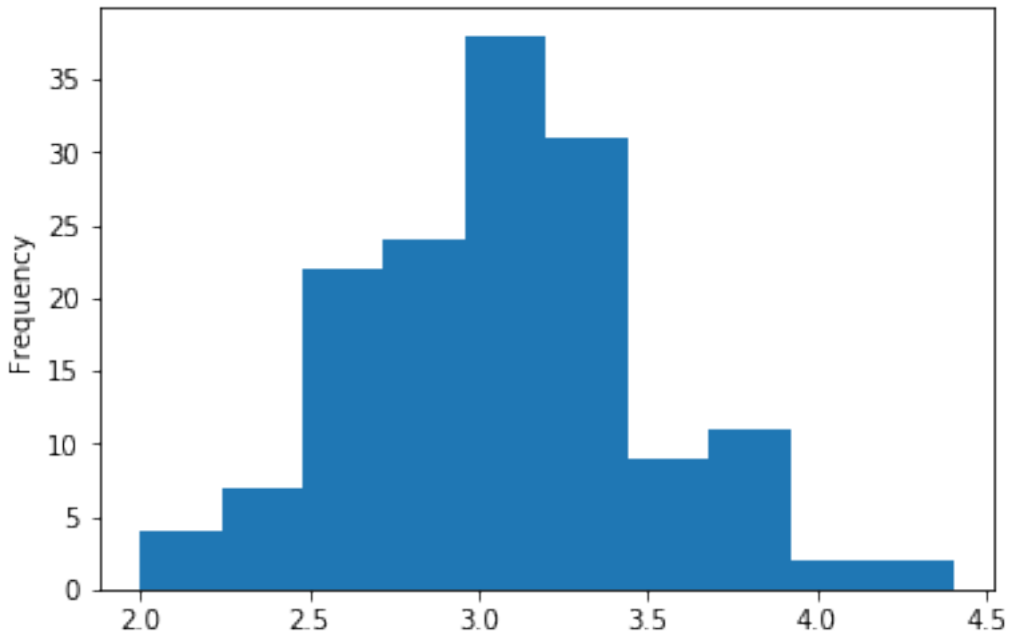
```
In [11]: iris["SepalLengthCm"].plot.hist()

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x121016c50>
```




```
In [12]: iris["SepalLengthCm"].describe()

Out[12]: count    150.000000
         mean       5.843333
         std        0.828066
         min        4.300000
         25%        5.100000
         50%        5.800000
         75%        6.400000
         max        7.900000
         Name: SepalLengthCm, dtype: float64
```

Al contrario, i dati relativi alla **larghezza del *sepalo*** risultano ben distribuiti, mostrando un "andamento" gaussiano

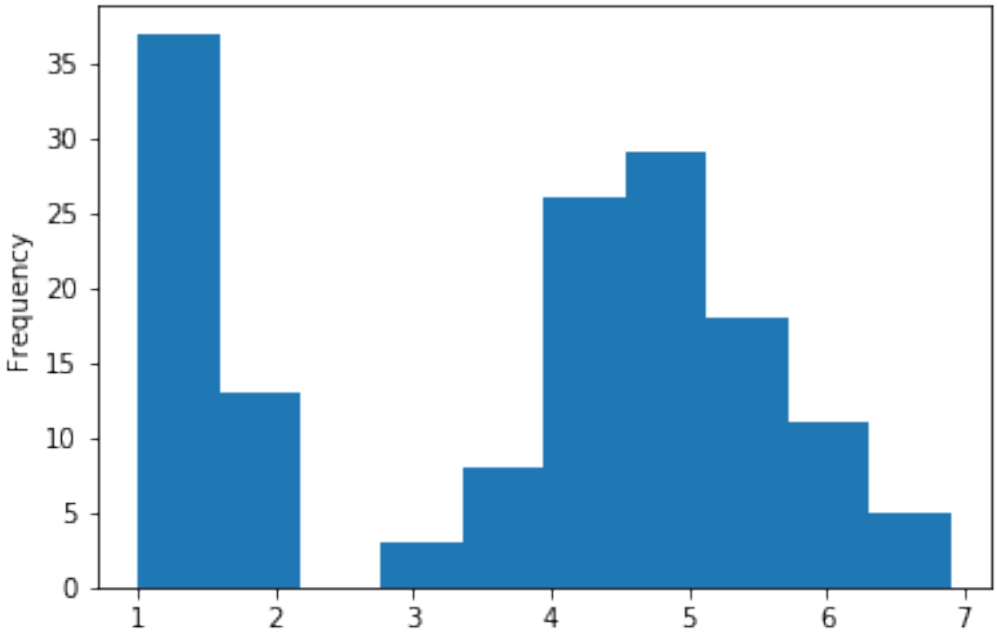
```
In [13]: iris["SepalWidthCm"].plot.hist()

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x121103588>
```




```
In [14]: iris["SepalWidthCm"].describe()

Out[14]: count    150.000000
         mean       3.054000
         std        0.433594
         min        2.000000
         25%        2.800000
         50%        3.000000
         75%        3.300000
         max        4.400000
         Name: SepalWidthCm, dtype: float64
```

Osserviamo poi che i dati relativi alla **lunghezza del *petalo*** presentano un andamento chiaramente bimodale, ossia sono presenti due picchi ben distinti

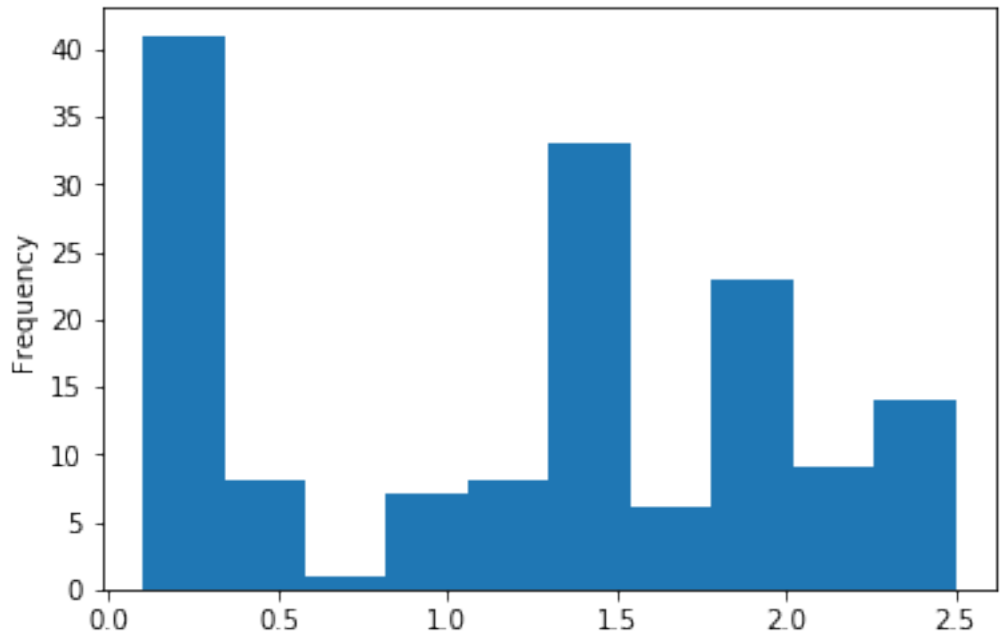
```
In [15]: iris["PetalLengthCm"].plot.hist()

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1211e8ac8>
```




```
In [16]: iris["PetalLengthCm"].describe()

Out[16]: count    150.000000
         mean       3.758667
         std        1.764420
         min        1.000000
         25%        1.600000
         50%        4.350000
         75%        5.100000
         max        6.900000
         Name: PetalLengthCm, dtype: float64
```

Lo stesso si può dire dei dati relativi alla **larghezza del *petalo***

```
In [17]: iris["PetalWidthCm"].plot.hist()

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x1212bcf60>
```



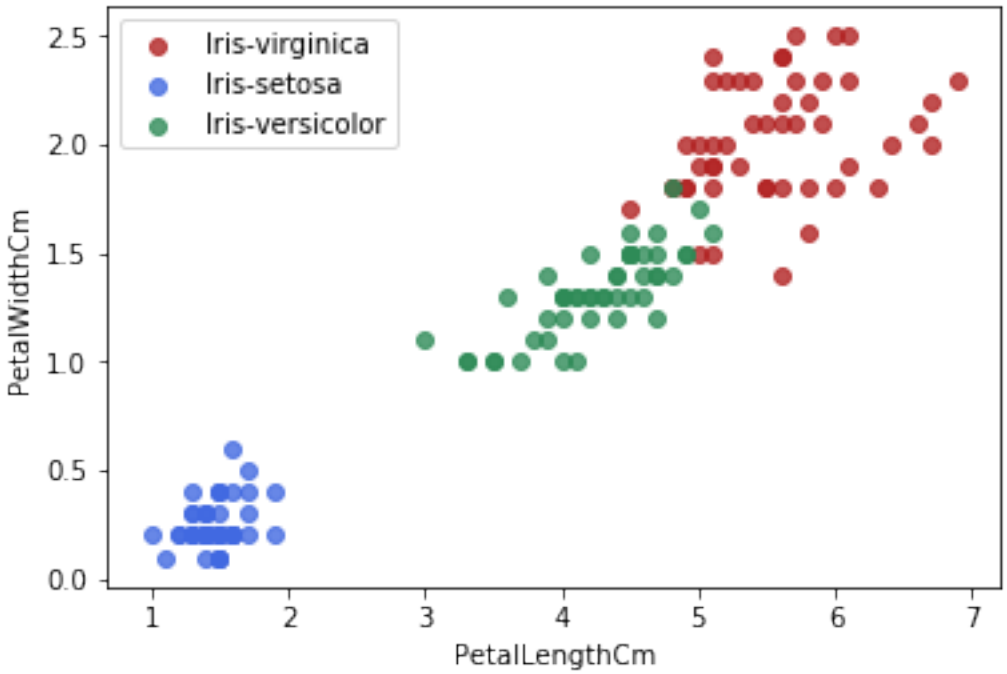

```
In [18]: iris["PetalWidthCm"].describe()

Out[18]: count    150.000000
         mean       1.198667
         std        0.763161
         min        0.100000
         25%        0.300000
         50%        1.300000
         75%        1.800000
         max        2.500000
         Name: PetalWidthCm, dtype: float64
```

Vogliamo ora mostrare la correlazione tra alcune coppie di feature, visualizzando un piano cartesiano con una feature su ciascun asse e con le nostre osservazioni colorate in modo differente a seconda della specie. Definiamo una funzione che ci permetta di fare questo

```
In [19]: species_colors = {"Iris-virginica": "firebrick", "Iris-setosa": "royalblue", "Iris-ve
         def plot_features(x, y):
             for species, color in species_colors.items():
                 plt.scatter(x[iris["Species"] == species], y[iris["Species"] == species], c=c
             plt.xlabel(x.name)
             plt.ylabel(y.name)
             plt.legend()
```

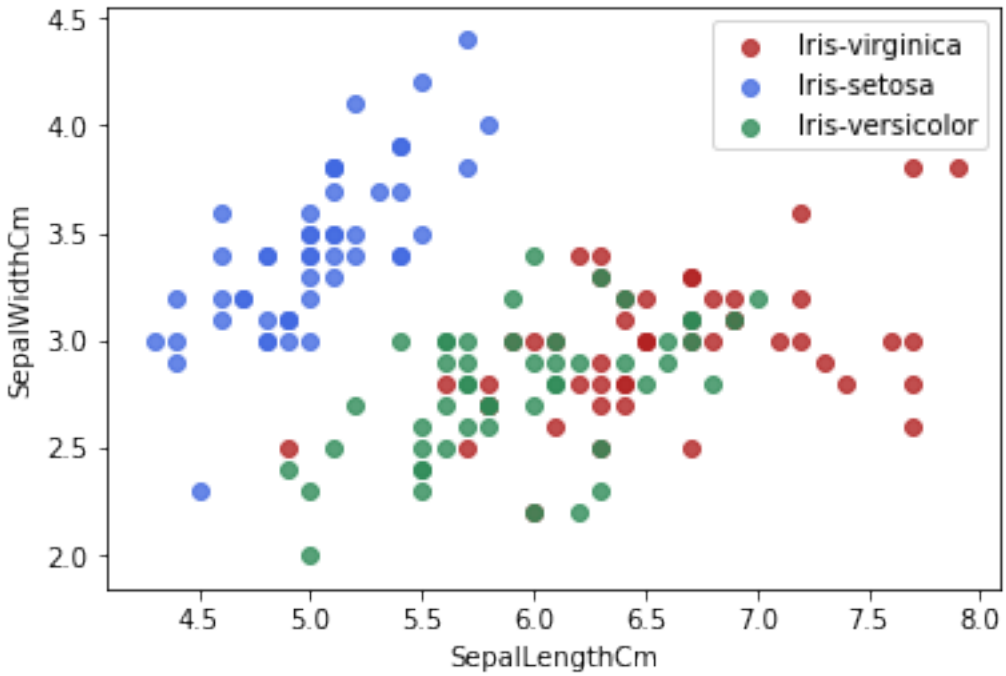
Visualizziamo prima le informazioni sul **petalo** (lunghezza e larghezza), osserviamo che queste già da sole ci permettono di distinguere le diverse specie in modo piuttosto efficace

```
In [20]: plot_features(iris["PetalLengthCm"], iris["PetalWidthCm"])
```



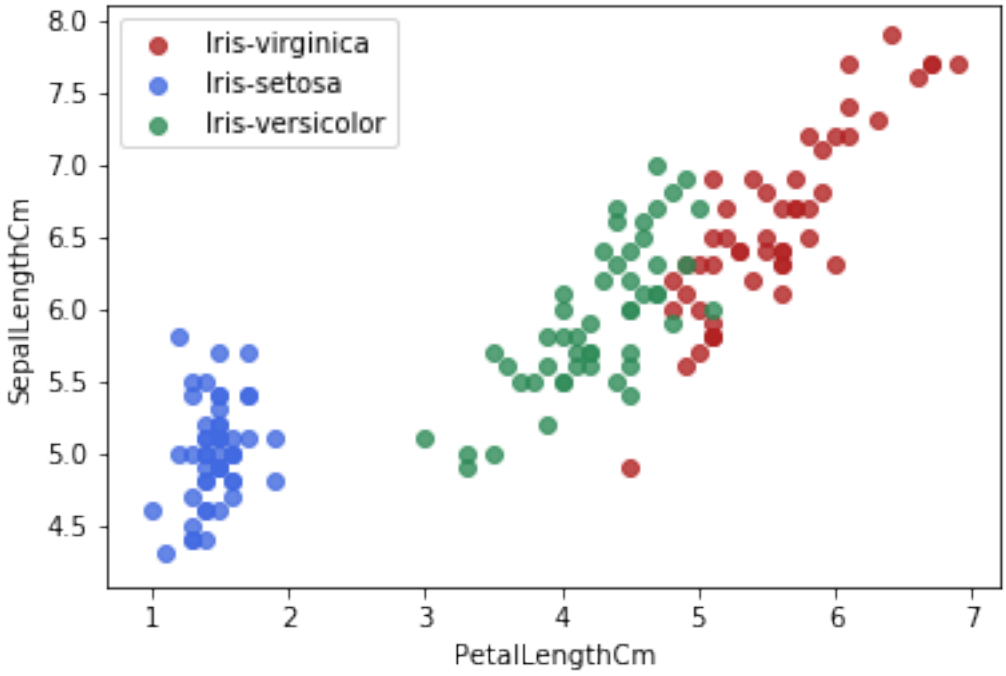
Visualizziamo anche le informazioni sul **sepalo** (lunghezza e larghezza): queste al contrario non permettono di distinguere i nostri dati in modo soddisfacente. In particolare le osservazioni di *iris virginica* e *iris versicolor* non vengono separate

```
In [21]: plot_features(iris["SepalLengthCm"], iris["SepalWidthCm"])
```



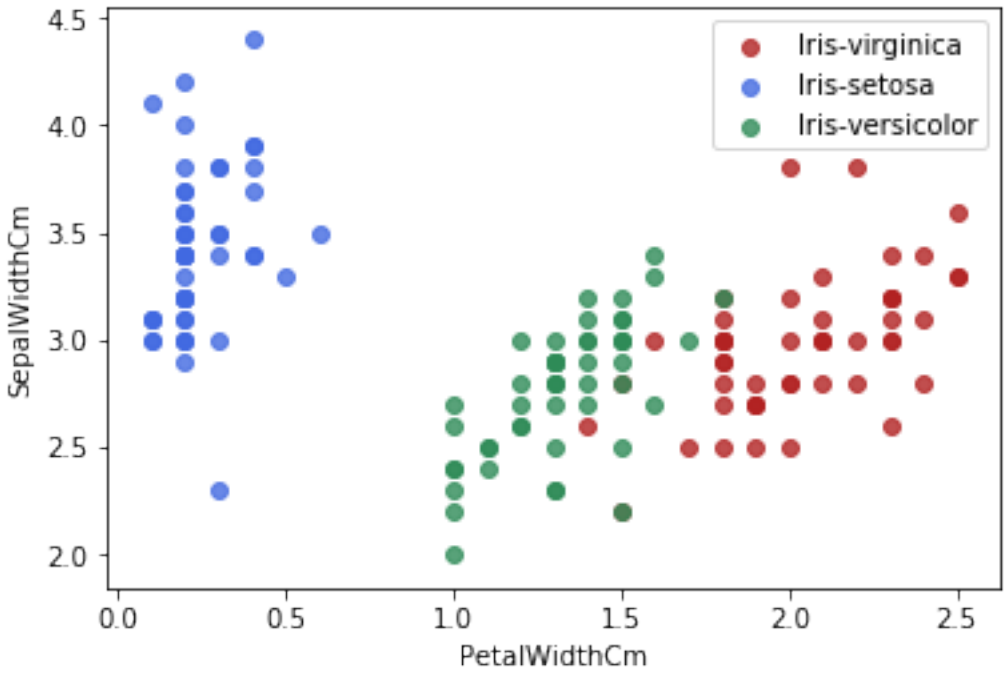
Visualizziamo ora le informazioni relative alla **lunghezza** dei due tratti, anche queste ci permettono di separare i nostri dati in modo efficace

```
In [22]: plot_features(iris["PetalLengthCm"], iris["SepalLengthCm"])
```



Lo stesso si può dire delle informazioni relative alla **larghezza** dei tratti

```
In [23]: plot_features(iris["PetalWidthCm"], iris["SepalWidthCm"])
```



1.3 Primo modello

Vogliamo costruire un primo modello usando unicamente le informazioni relative al **petalo** dei fiori (lunghezza e larghezza), in questo modo - avendo solo due feature - sarà possibile dare una rappresentazione grafica degli iperpiani trovati

Creiamo un nuovo DataFrame `iris_petal` ottenuto dal DataFrame originale rimuovendo le colonne relative al sepalo

```
In [24]: iris_petal = iris.copy()
         iris_petal.drop(["SepalLengthCm", "SepalWidthCm"], axis=1, inplace=True)
```

Convertiamo la colonna categorica `Species` in una colonna numerica, assegnando a ciascuna specie un numero intero crescente partendo da 1. D'ora in poi si farà riferimento a tale numero come come "codice di una specie"

```
In [25]: species_codes = {"Iris-virginica": 1, "Iris-setosa": 2, "Iris-versicolor": 3}
         iris_petal["Species"] = iris_petal["Species"].map(species_codes)
         iris_petal["Species"].value_counts()
```

```
Out[25]: 3    50
         2    50
         1    50
         Name: Species, dtype: int64
```

Nel caso di classificazione multiclasse con iperpiani ci sono due approcci possibili: - **One versus all** (o **one versus rest**) - **Multinomial**

Nel nostro caso scegliamo l'approccio **one vs all**, questo prevede l'indivduazione di un numero di iperpiani pari al numero di classi (nel nostro caso 3): ciascuno deve separare una classe dal resto del mondo. Inoltre l'individuazione di ogni iperpiano è indipendente dalle altre

Iniziamo con l'individuare un iperpiano che separi le osservazioni di *iris virginica* dalle altre, creiamo una serie `target` in cui le osservazioni della specie di interesse conservano il proprio codice, mentre le osservazioni relative al resto del mondo presentano valore 0

```
In [26]: target = iris_petal["Species"].copy()
         target[iris["Species"] != "Iris-virginica"] = 0
         target.value_counts()
```

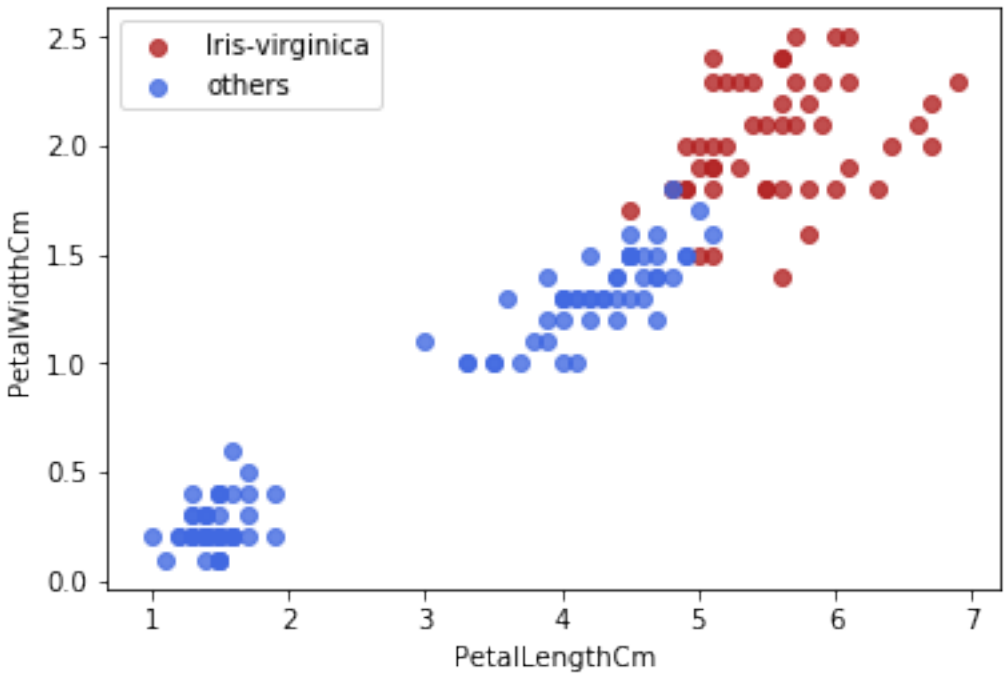
```
Out[26]: 0    100
         1     50
         Name: Species, dtype: int64
```

Definiamo una funzione che ci permetta di visualizzare in un grafico quali osservazioni si vogliono separare dal resto del mondo

```
In [27]: def plot_target(label):
             plt.scatter(iris["PetalLengthCm"][target != 0], iris["PetalWidthCm"][target != 0]
             plt.scatter(iris["PetalLengthCm"][target == 0], iris["PetalWidthCm"][target == 0]
             plt.legend()
             plt.xlabel("PetalLengthCm")
             plt.ylabel("PetalWidthCm")
```

Nonostante le osservazioni di *iris virginica* non siano linearmente separabili dal resto del mondo, visivamente è chiaro che possiamo comunque usare una retta come *decision boundary* in modo efficace

```
In [28]: plot_target("Iris-virginica")
```



Separiamo i nostri dati in training set e validation set, usiamo la serie `target` come colonna y da predire

```
In [29]: from sklearn.model_selection import train_test_split
         X_train, X_val, y_train, y_val = train_test_split(
             iris_petal.drop(["Species"], axis=1),
             target,
             test_size=1/3, random_state=42
         )
```

Silenziamo i warning che risultano piuttosto fastidiosi se si sta eseguendo il codice in jupyter

```
In [30]: import warnings
         warnings.filterwarnings('ignore')
```

Scegliamo un modello "base" per la classificazione: il `Perceptron`. Il parametro `max_iter` indica il numero massimo di iterazioni dopo le quali l'algoritmo arriva a convergenza, di default questo è 1000, lo aumentiamo in modo da avere più precisione

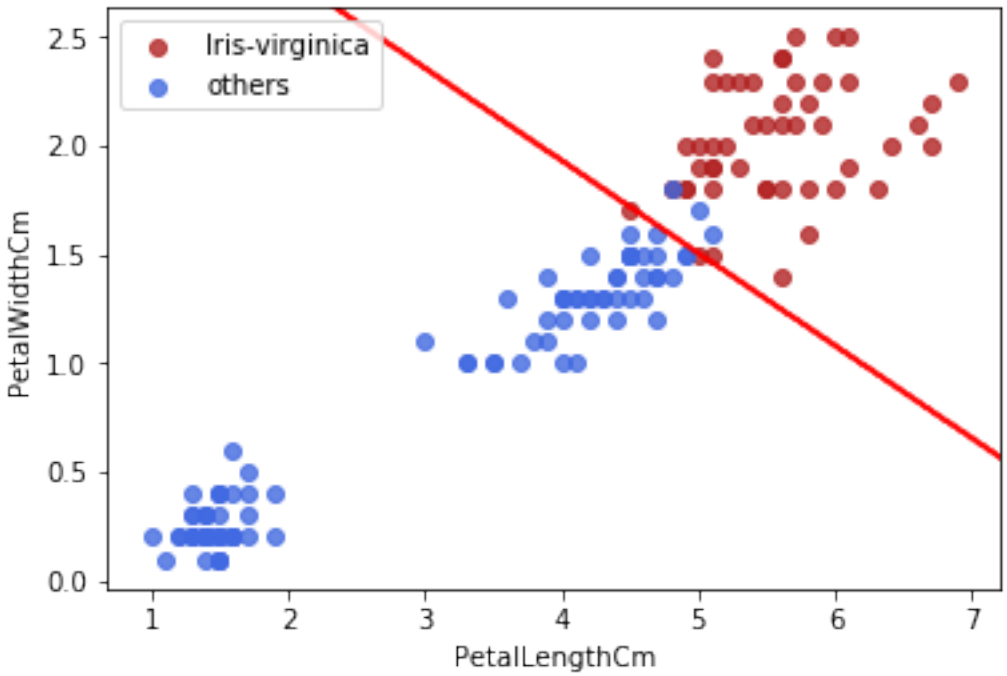
```
In [31]: from sklearn.linear_model import Perceptron
         model1 = Perceptron(max_iter=1000000)
         model1.fit(X_train, y_train)
         model1.score(X_val, y_val)

Out[31]: 1.0
```

Definiamo una funziona che ci permetta di visualizzare il *decision boundary* che abbiamo ricavato

```
In [32]: def plot_decision_boundary(X, y, model, c="r"):
             # Step size of the mesh. Decrease to increase the quality of the VQ.
             h = .01     # point in the mesh [x_min, m_max]x[y_min, y_max].

             # Plot the decision boundary. For that, we will assign a color to each
             x_min, x_max = X.iloc[:, 0].min() - 1, X.iloc[:, 0].max() + 1
             y_min, y_max = X.iloc[:, 1].min() - 1, X.iloc[:, 1].max() + 1
             xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

             # Obtain labels for each point in mesh using the model.
             Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

             x_min, x_max = X.iloc[:, 0].min() - 1, X.iloc[:, 0].max() + 1
             y_min, y_max = X.iloc[:, 1].min() - 1, X.iloc[:, 1].max() + 1
             xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                                  np.arange(y_min, y_max, 0.01))

             Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

             #plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=iris["Species"].map(species_colors), a
             xlim, ylim = plt.xlim(), plt.ylim()
             plt.xlim(xlim); plt.ylim(ylim)
             plt.contour(xx, yy, Z, alpha=0.4, colors=c)
```

Visivamente il risultato è soddisfacente

```
In [33]: plot_target("Iris-virginica")
         plot_decision_boundary(iris_petal.drop(["Species"], axis=1), target, model1)
```

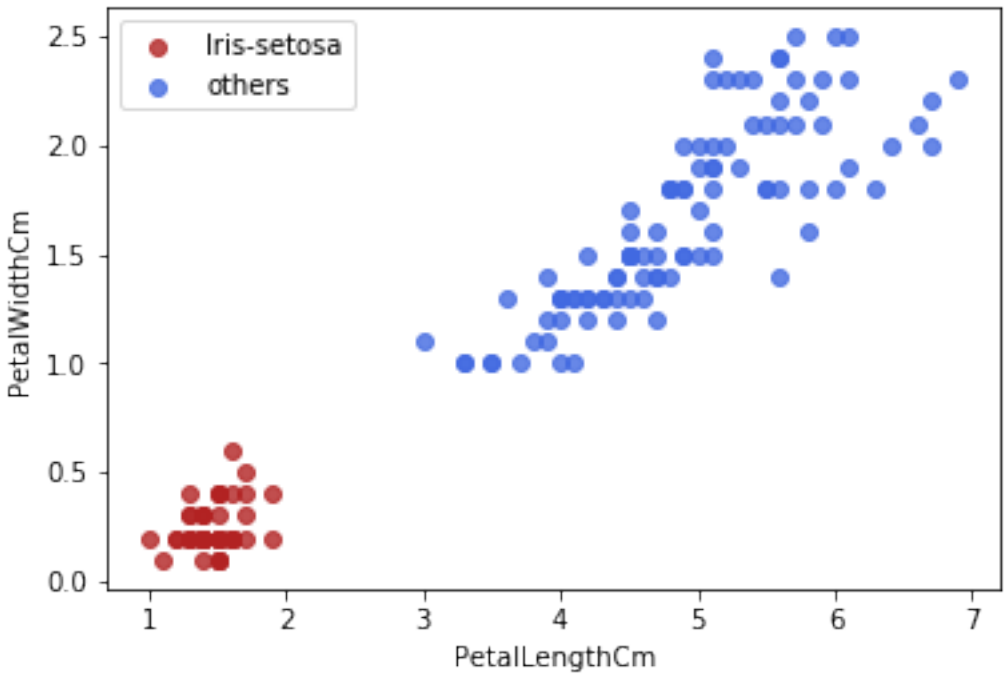


Procediamo ora con l'individuazione di un iperpiano che separi le osservazioni di *iris setosa* dalle altre, aggiorniamo la serie `target` di conseguenza

```
In [34]: target = iris_petal["Species"].copy()
         target[iris["Species"] != "Iris-setosa"] = 0
         target.value_counts()

Out[34]: 0    100
         2     50
         Name: Species, dtype: int64
```

In questo caso le osservazioni di interesse sono linearmente separabili dal resto del mondo

```
In [35]: plot_target("Iris-setosa")
```

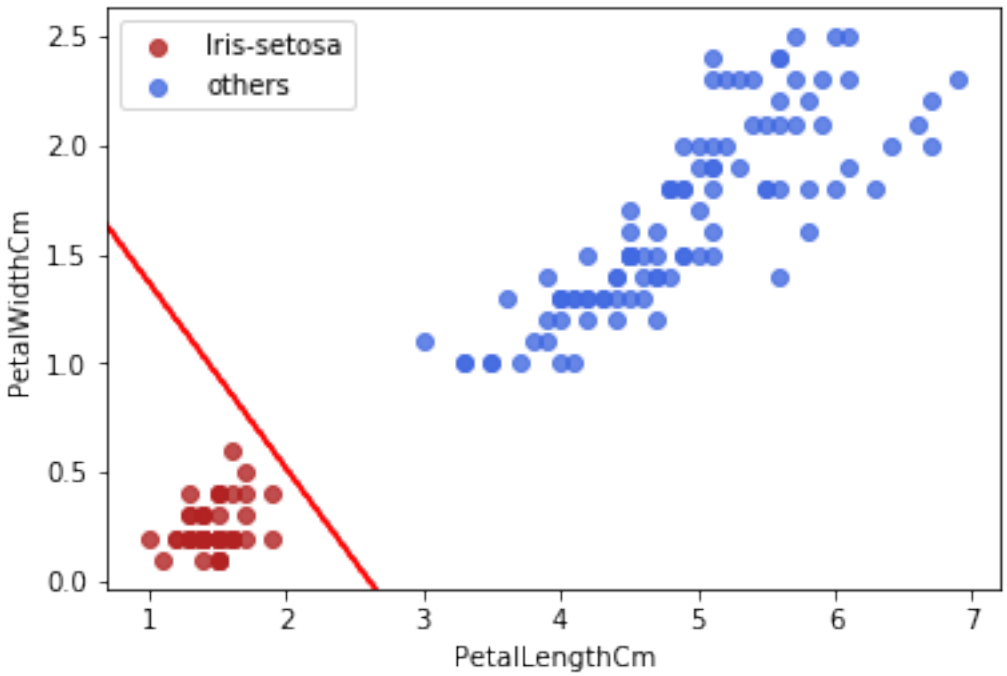


Dividiamo dunque i dati in training e validation set (è necessario farlo ancora in quanto la colonna target è mutata) e addestriamo un modello lineare, allo stesso modo di prima

```
In [36]: X_train, X_val, y_train, y_val = train_test_split(
             iris_petal.drop(["Species"], axis=1),
             target,
             test_size=1/3, random_state=42
         )

In [37]: model2 = Perceptron()
         model2.fit(X_train, y_train)
         model2.score(X_val, y_val)

Out[37]: 1.0
```

Visualizziamo infine l'iperpiano trovato

```
In [38]: plot_target("Iris-setosa")
         plot_decision_boundary(iris_petal.drop(["Species"], axis=1), target, model2)
```



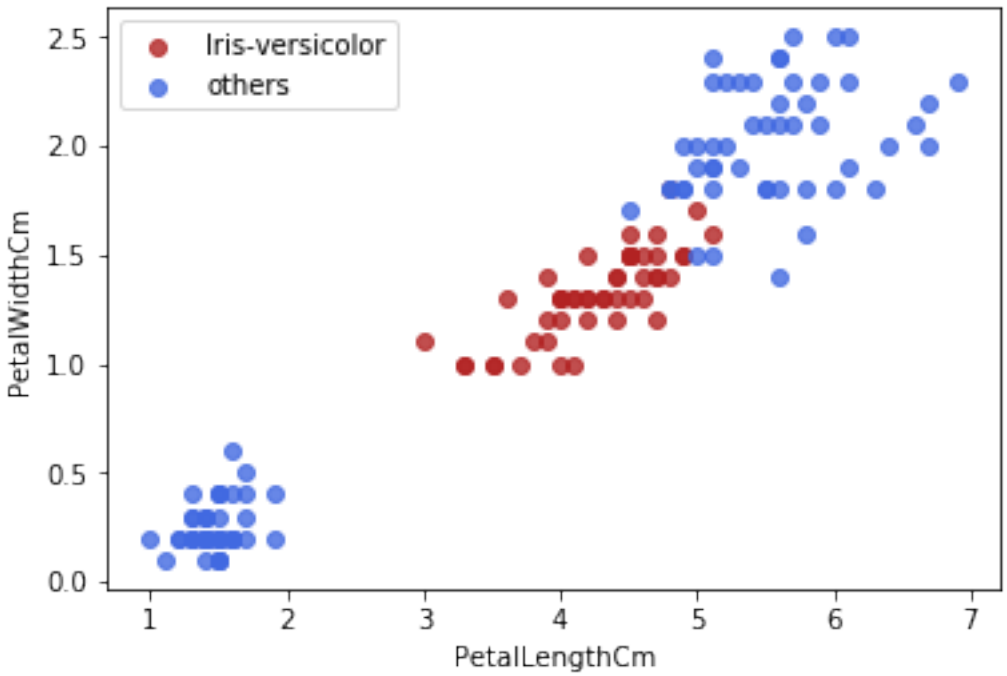
Vogliamo infine individuare un iperpiano che separi le osservazioni di *iris versicolor* dalle altre

```
In [39]: target = iris_petal["Species"].copy()
         target[iris["Species"] != "Iris-versicolor"] = 0
         target.value_counts()
```

```
Out[39]: 0    100
         3     50
         Name: Species, dtype: int64
```

In questo caso le osservazioni di interesse *non* sono linearmente separabili dal resto del mondo, ne tantomento possiamo impiegare una retta come decision boundary in modo efficace. E' dunque inevitabile introdurre delle feature polinomiali, in questo caso è sufficiente fermarsi al secondo grado. Ciò che si sta facendo con questa operazione è mappare i nostri dati in uno spazio a più dimensioni, in cui questi siano separabili in modo efficace con un iperpiano (e dunque con una funzione lineare). Visualizzando tale iperpiano in due dimensioni questo risulterà chiaramente in una funzione non lineare

```
In [40]: plot_target("Iris-versicolor")
```



Dividiamo i dati in training set e validation set ancora una volta

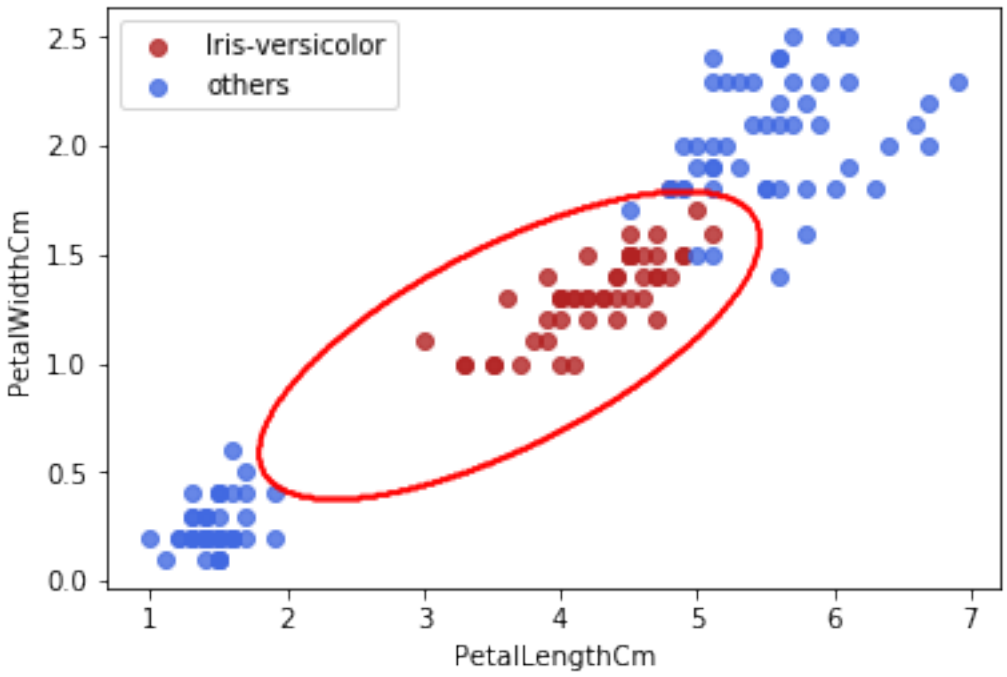
```
In [41]: X_train, X_val, y_train, y_val = train_test_split(
             iris_petal.drop(["Species"], axis=1),
             target,
             test_size=1/3, random_state=42
         )
```

Facciamo uso del filtro `PolynomialFeatures` che ci permette di ricavare feature polinomiali dai nostri dati, lo inseriamo all'interno di una `Pipeline` e alleniamo il nostro modello

```
In [42]: from sklearn.pipeline import Pipeline
         from sklearn.preprocessing import PolynomialFeatures

         model3 = Pipeline([
             ("poly", PolynomialFeatures(degree=2, include_bias=False)),
             ("perc", Perceptron(max_iter=100000))
         ])
         model3.fit(X_train, y_train)
         model3.score(X_val, y_val)

Out[42]: 0.98
```

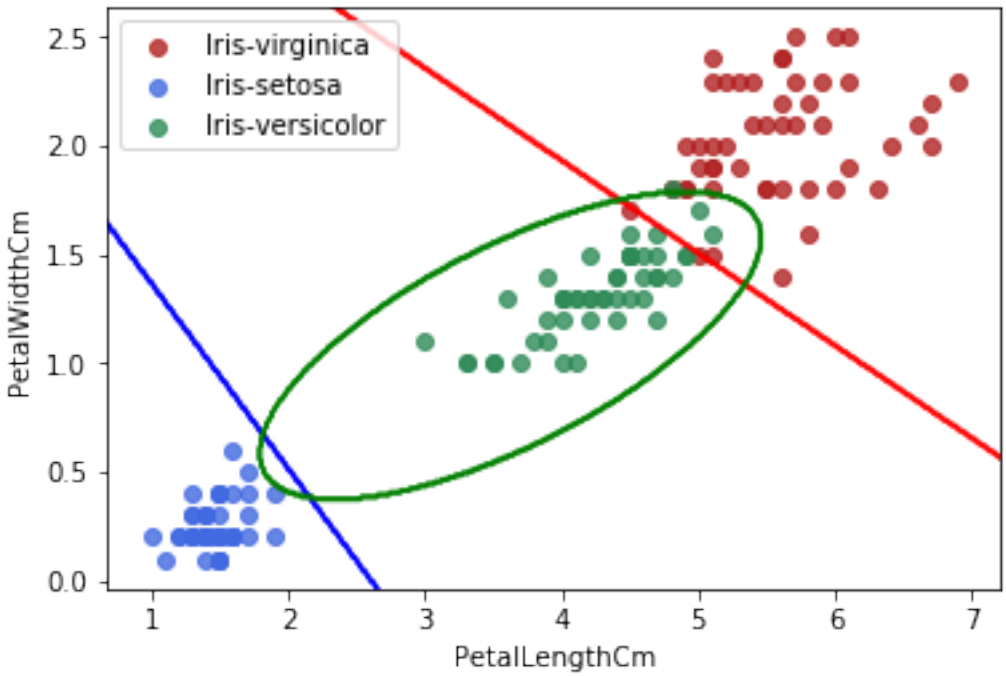
Visualizziamo il decision boundary trovato, come anticipato in due dimensioni questo risulterà in una funzione non lineare

```
In [43]: plot_target("Iris-versicolor")
         plot_decision_boundary(iris_petal.drop(["Species"], axis=1), target, model3)
```



Abbiamo trovato tre iperpiani: ciascuno separa una classe (una specie) dal resto del mondo, visualizziamo congiuntamente i tre decision boundary

```
In [44]: plot_features(iris["PetalLengthCm"], iris["PetalWidthCm"])
         plot_decision_boundary(iris_petal.drop(["Species"], axis=1), target, model1, c="r")
         plot_decision_boundary(iris_petal.drop(["Species"], axis=1), target, model2, c="b")
         plot_decision_boundary(iris_petal.drop(["Species"], axis=1), target, model3, c="g")
```



Possiamo usarli in modo congiunto per classificare un'osservazione in una delle tre specie, la regola di fusione è la seguente: ad ogni istanza x si assegna la classe y corrispondente al piano j che massimizza

$$y = \underset{j=1,\ldots,C}{\text{argmax}} b_j + x^T w_j$$

Definiamo una funzione che faccia questa operazione, si noti che non usiamo la funzione `predict(x)` dei modelli, in quanto restituirebbe un valore binario, usiamo invece la funzione `decision_function(x)` che restituisce una stima della confidenza con la quale il modello in questione classifica l'osservazione in esame. La classe finale sarà quella relativa al modello con confidenza maggiore

```
In [45]: def ovr_predict(x):
             confidence1 = model1.decision_function(x)
             confidence2 = model2.decision_function(x)
             confidence3 = model3.decision_function(x)
             max_confidence = confidence1
             c = 1
             if confidence2 > max_confidence:
                 max_confidence = confidence2
                 c = 2
             if confidence3 > max_confidence:
                 max_confidence = confidence3
                 c = 3
             return c
```

```
def ovr_predict_aggregate(X):
    y = []
    for i in range(0, len(X)):
        if isinstance(X, pd.DataFrame):
            y.append(ovr_predict(X.iloc[[i], :]))
        else:
            y.append(ovr_predict(X[[i], :]))
    return np.array(y)
```

Costruiamo ancora una volta training set e validation set, questa volta usando la colonna `Species` di `iris_petal` (e non `target`) come colonna y. Questo vuol dire che le osservazioni saranno divise nelle tre classi

```
In [46]: X_train, X_val, y_train, y_val = train_test_split(
             iris_petal.drop(["Species"], axis=1),
             iris_petal["Species"],
             test_size=1/3, random_state=42
         )
```

Misuriamo l'accuratezza del modello ottenuto: è del 98%

```
In [47]: hits = y_val == ovr_predict_aggregate(X_val)
         hits.mean()

Out[47]: 0.98
```

Essendo che nei modelli di classificazione l'accuratezza non è l'unica metrica da prendere in considerazione, costruiamo la **matrice di confusione** per il nostro modello. Si ricorda che la matrice di confusione è una matrice in cui la cella in posizione (i,j) indica quanti esempi della classe i-esima sono stati etichettati dal classificatore come di classe j-esima

```
In [48]: from sklearn.metrics import confusion_matrix
         y_val_pred = ovr_predict_aggregate(X_val)
         cm = confusion_matrix(y_val, y_val_pred)
         classes = ["Iris-virginica", "Iris-setosa", "Iris-versicolor"]
         pd.DataFrame(cm, index=classes, columns=classes)
```

Out[48]:

	Iris-virginica	Iris-setosa	Iris-versicolor
Iris-virginica	15	0	1
Iris-setosa	0	19	0
Iris-versicolor	0	0	15

Osserviamo che è presente un unico errore di classificazione, misuriamo ora precision e recall per ciascuna delle tre classi. Si ricorda che la **precision** relativa a una classe inidica la percentuale di esempi classificati come appartanenti a quella classe che sono realmente tali, la **recall** relativa a una classe indica la percentuale di osservazioni della classe in esame che sono stati classificati come tali

```
In [49]: from sklearn.metrics import precision_score, recall_score, f1_score

         precision_score(y_val, y_val_pred, average=None)
```

```
Out[49]: array([1.    , 1.    , 0.9375])

In [50]: recall_score(y_val, y_val_pred, average=None)

Out[50]: array([0.9375, 1.    , 1.    ])
```

Come prevedibile già dalla matrice di confusione, i risultati sono piuttosto soddisfacenti. Misuriamo anche la **f1 score** per ciascuna classe, questa corrisponde a una media armonica di precision e recall

```
In [51]: f1_score(y_val, y_val_pred, average=None)

Out[51]: array([0.96774194, 1.        , 0.96774194])
```

Effettuiamo infine una **media delle diverse f1 score** per avere un unico valore che rispecchi l'efficacia del modello

```
In [52]: f1_score(y_val, y_val_pred, average="macro")

Out[52]: 0.978494623655914
```

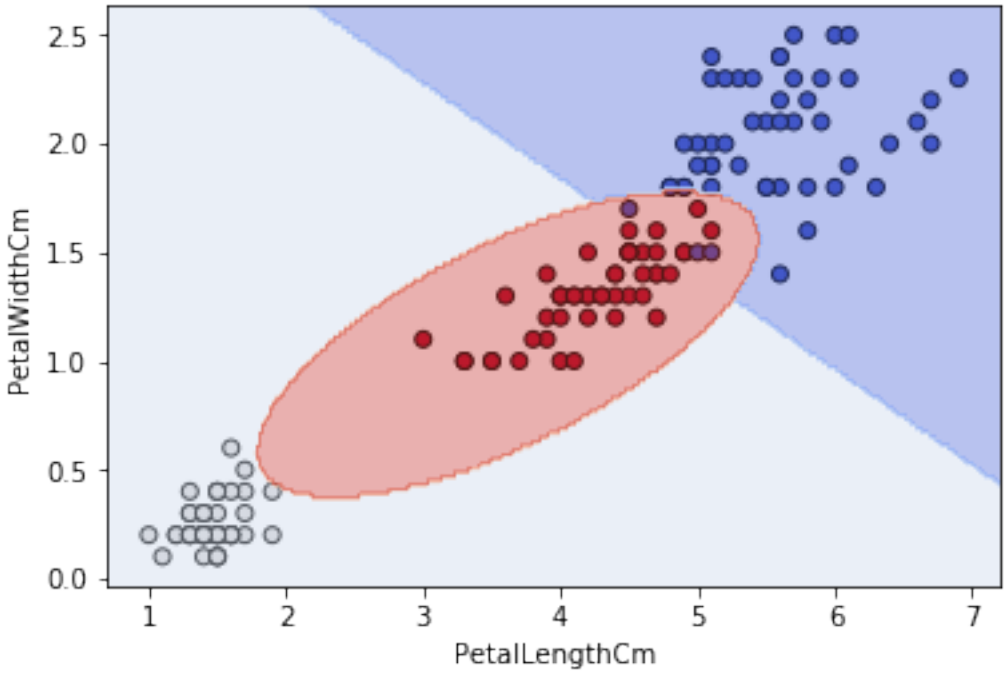
Il risultato ottenuto è piuttosto soddisfacente

L'ultima cosa che vogliamo fare è mostrare i *decision boundaries* "finali" ottenuti dalla fusione dei tre iperpiani, definiamo una funzione che ci permetta di farlo

```
In [53]: def plot_decision_boundaries(model):

             X = iris_petal.drop(["Species"], axis=1)
             y = iris["Species"].map(species_codes)

             def make_meshgrid(x, y, h=.02):
                 x_min, x_max = x.min() - 1, x.max() + 1
                 y_min, y_max = y.min() - 1, y.max() + 1
                 xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
                 return xx, yy

             def plot_contours(xx, yy, **params):
                 Z = model(np.c_[xx.ravel(), yy.ravel()])
                 Z = Z.reshape(xx.shape)
                 out = plt.contourf(xx, yy, Z, **params)
                 return out

             X0, X1 = X.iloc[:, 0], X.iloc[:, 1]
             xx, yy = make_meshgrid(X0, X1)
             plt.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, edgecolors='k')
             xlim, ylim = plt.xlim(), plt.ylim()
             plt.xlim(xlim); plt.ylim(ylim)

             plot_contours(xx, yy, alpha=0.4, cmap=plt.cm.coolwarm)
             plt.ylabel('PetalWidthCm')
             plt.xlabel('PetalLengthCm')
             #plt.show()
```

Osserviamo che, per quanto le nostre misure siano soddisfacenti, il modello finale ottenuto risulta piuttosto aderente ai dati. In particolare l'iperpiano usato per classificare le osservazioni di *iris versicolor* (in figura la regione rossa) potrebbe non essere così efficace su dati futuri, soprattutto se questi risultassero più *sparsi* di quelli attuali

```
In [54]: plot_decision_boundaries(ovr_predict_aggregate) # this may take a while
```



1.4 Modelli più complessi

I modelli allenati fino a questo momento usano solo una parte delle feature a nostra disposizione, inoltre non fanno uso di **standardizzazione** dei dati, **regolarizzazione** dei parametri degli iperpiani (in effetti è stato appena osservato che uno dei tre iperpiani trovati risulta troppo aderente ai dati, cioè è andato in overfitting), e le misure di cui sopra sono state calcolate dividendo i dati con il metodo hold-out, cioè partizionandoli una sola volta. Ciò che si vuole fare ora è allenare modelli più complessi che facciano uso di tutte le feature a nostra disposizione e delle tecniche citate sopra, le metriche di valutazione di questi modelli saranno calcolate con la k-fold cross validation, in modo da ottenere stime più robuste.

Si sappia che la classificazione multiclasse è supportata dai modelli di scikit learn, dunque quanto è stato fatto "a mano" fino a questo momento può essere effettuato in modo automatico

Dividiamo i dati in training e validation set, usando questa volta il DataFrame originale `iris`

```
In [55]: X_train, X_val, y_train, y_val = train_test_split(
             iris.drop(["Species"], axis=1),
             iris["Species"].map(species_codes),
             test_size=1/3, random_state=42
         )
```

Scegliamo come primo algoritmo il `Perceptron`, usato anche in precedenza, introducendo però feature polinomiali, standardizzazione dei dati e regolarizzazione dei parametri. Utilizziamo la grid search per testare diverse combinazioni di iperparametri e trovare quella che fornisce risultati migliori, in particolare testiamo: - feature polinomiali fino al grado 4 - standardizzazione dei dati presente e assente - nessuna regolarizzazione, regolarizzazione con norma 1, con norma 2 e con entrambe (elastic net) - intensità della regolarizzazione (parametro alpha) da $1x10^{-6}$ a $1x10^{3}$

```
In [56]: from sklearn.preprocessing import PolynomialFeatures
         from sklearn.preprocessing import StandardScaler

         model = Pipeline([
             ("poly", PolynomialFeatures(include_bias=False)),
             ("scaler", StandardScaler()),
             ("perc",  Perceptron())
         ])

         grid = {
             "poly__degree": [2,3,4],
             "scaler": [None, StandardScaler()],
             "perc__penalty": [None, "l1", "l2", "elasticnet"],
             "perc__alpha": np.logspace(-6, 3, 10)
         }
```

Di default i modelli vengono confrontati in base all'accuratezza, vogliamo invece che venga usata come metrica la **f1 measure**, tuttavia questa di default non è multiclasse. Dunque è necessario creare uno `scorer` personalizzato basato su una media delle f1 measure delle diverse classi

```
In [57]: from sklearn.metrics import make_scorer
         f1_multiclass = make_scorer(f1_score, greater_is_better=True, average="macro")
```

Lanciamo la grid search sui dati del training set, questi saranno soggetti a k-fold cross validation per trovare gli iperparametri migliori. Ottenuto il modello migliore potremo testarlo sui dati del validation set, che questo non ha mai "visto"

```
In [58]: from sklearn.model_selection import GridSearchCV

         gs = GridSearchCV(model, grid, scoring=f1_multiclass, n_jobs=2)
         gs.fit(X_train, y_train)
         pd.DataFrame(gs.cv_results_).sort_values("rank_test_score").head(3)
```

Out[58]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	\
59	0.003405	0.000232	0.001447	0.000038	
173	0.003161	0.000083	0.001529	0.000089	
197	0.003286	0.000236	0.001525	0.000034	

	param_perc__alpha	param_perc__penalty	param_poly__degree	\
59	0.0001	l1	4	
173	10	None	4	
197	100	None	4	

```
                                              param_scaler  \
59   StandardScaler(copy=True, with_mean=True, with...
173  StandardScaler(copy=True, with_mean=True, with...
197  StandardScaler(copy=True, with_mean=True, with...

                                                params  split0_test_score  \
59   {'perc__alpha': 0.0001, 'perc__penalty': 'l1',...           0.972174
173  {'perc__alpha': 10.0, 'perc__penalty': None, '...           0.972174
197  {'perc__alpha': 100.0, 'perc__penalty': None, ...           0.972174

     split1_test_score  split2_test_score  mean_test_score  std_test_score  \
59            0.941919           1.000000         0.971094        0.023424
173           0.941919           0.969634         0.961377        0.013695
197           0.941919           0.969634         0.961377        0.013695

     rank_test_score  split0_train_score  split1_train_score  \
59                 1            0.939873                 1.0
173                2            0.955468                 1.0
197                2            0.955468                 1.0

     split2_train_score  mean_train_score  std_train_score
59             0.943030          0.960968         0.027630
173            0.928949          0.961472         0.029315
197            0.928949          0.961472         0.029315
```

Il modello migliore si ottiene con feature polinomiali di quarto grado, standardizzazione dei dati, regolarizzazione con norma 1 e intensità della regolarizzazione piuttosto bassa, pari a `0.0001`. Il modello così ottenuto ha una f1 score media di 0.97 sulle porzioni di training set usate per il test

Lo salviamo in una variabile

```
In [59]: perc_model = gs.best_estimator_
```

Vogliamo ora testare la **regressione logistica**, come prima con l'introduzione di feature polinomiali, la standardizzazione dei dati e la regolarizzazione dei parametri. In particolare testiamo: - feature polinomiali fino al grado 4 - standardizzazione dei dati presente e assente - regolarizzazione con norma 1 e con norma 2 - intensità della regolarizzazione (parametro C) da $1x10^{-6}$ a $1x10^{3}$

```
In [60]: from sklearn.linear_model import LogisticRegression

         model = Pipeline([
             ("poly", PolynomialFeatures(include_bias=False)),
             ("scaler", StandardScaler()),
             ("lr",  LogisticRegression())
         ])

         grid = {
             "poly__degree": [2,3,4],
```

```
    "scaler": [None, StandardScaler()],
    "lr__penalty": ["l1", "l2"],
    "lr__C": np.logspace(-6, 3, 10)
}

gs = GridSearchCV(model, grid, scoring=f1_multiclass, n_jobs=2)
gs.fit(X_train, y_train)
pd.DataFrame(gs.cv_results_).sort_values("rank_test_score").head(3)
```

Out[60]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_lr__C	\
66	0.002210	0.000310	0.001112	0.000309	0.1	
72	0.018293	0.004914	0.001305	0.000245	1	
83	0.003920	0.000639	0.001706	0.000412	1	

	param_lr__penalty	param_poly__degree	\
66	l2	2	
72	l1	2	
83	l2	4	

	param_scaler	\
66	None	
72	None	
83	StandardScaler(copy=True, with_mean=True, with...	

	params	split0_test_score	\
66	{'lr__C': 0.1, 'lr__penalty': 'l2', 'poly__deg...	1.0	
72	{'lr__C': 1.0, 'lr__penalty': 'l1', 'poly__deg...	1.0	
83	{'lr__C': 1.0, 'lr__penalty': 'l2', 'poly__deg...	1.0	

	split1_test_score	split2_test_score	mean_test_score	std_test_score	\
66	0.941919	0.969634	0.971116	0.023959	
72	0.941919	0.969634	0.971116	0.023959	
83	0.912381	1.000000	0.971086	0.041200	

	rank_test_score	split0_train_score	split1_train_score	\
66	1	0.970356	0.9855	
72	1	0.985185	1.0000	
83	3	0.985185	1.0000	

	split2_train_score	mean_train_score	std_train_score
66	0.971618	0.975825	0.006861
72	0.985816	0.990334	0.006840
83	0.985816	0.990334	0.006840

Il modello migliore si ottiene con feature polinomiali di secondo grado, nessuna standardizzazione dei dati, regolarizzazione dei parametri con norma 2 e intensità di quest'ultima pari a `0.1`. Il modello così ottenuto ha una f1 score media di 0.97 sulle porzioni di training set usate per il test

Lo salviamo in una variabile

```
In [61]: lr_model = gs.best_estimator_
```

Infine vogliamo provare a fare classificazione usando delle **support vectors machines**, che ci permettono di trovare gli iperpiani *ottimi* per separare i dati. In particolare testiamo: - standardizzazione dei dati presente e assente - kernel lineare, polinomiale (di default di terzo grado) e gaussiano - parametro C, che regola il peso assegnato agli errori di classificazione, da $1x10^{-6}$ a $1x10^{3}$

```
In [62]: from sklearn.svm import SVC

         model = Pipeline([
             ("scaler", StandardScaler()),
             ("svc",  SVC())
         ])

         grid = {
             "scaler": [None, StandardScaler()],
             "svc__kernel": ["linear", "poly", "rbf"],
             "svc__C": np.logspace(-6, 3, 10)
         }

         gs = GridSearchCV(model, grid, scoring=f1_multiclass, n_jobs=2)
         gs.fit(X_train, y_train)
         pd.DataFrame(gs.cv_results_).sort_values("rank_test_score").head(3)
```

```
Out[62]:     mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
         24       0.001717      0.000246         0.001029        0.000232
         54       0.001848      0.000042         0.000838        0.000007
         48       0.001858      0.000074         0.000844        0.000037

                                              param_scaler param_svc__C  \
         24                                           None          100
         54  StandardScaler(copy=True, with_mean=True, with...          100
         48  StandardScaler(copy=True, with_mean=True, with...            1

            param_svc__kernel                                             params  \
         24            linear  {'scaler': None, 'svc__C': 100.0, 'svc__kernel...
         54            linear  {'scaler': StandardScaler(copy=True, with_mean...
         48            linear  {'scaler': StandardScaler(copy=True, with_mean...

             split0_test_score  split1_test_score  split2_test_score  mean_test_score  \
         24                1.0           0.882051                1.0         0.961077
         54                1.0           0.882051                1.0         0.961077
         48                1.0           0.882051                1.0         0.961077

             std_test_score  rank_test_score  split0_train_score  split1_train_score  \
         24        0.055461                1            0.985185                 1.0
         54        0.055461                1            0.985185                 1.0
```

```
48        0.055461                1            0.970356                 1.0

    split2_train_score  mean_train_score  std_train_score
24            0.985816          0.990334         0.006840
54            0.985816          0.990334         0.006840
48            0.957370          0.975908         0.017841
```

Il modello migliore si ottiene senza standardizzazione dei dati, con kernel lineare e parametro C pari a 100. Il modello così ottenuto ha una f1 score media di 0.96 sulle porzioni di training set usate per il test, leggermente inferiore ai modelli precedenti.

Lo salviamo in una variabile

```
In [63]: svm_model = gs.best_estimator_
```

Ciò che vogliamo fare ora è testare i tre modelli migliori sul validation set, che questi non hanno mai visto. Iniziamo misurando **l'accuratezza**

```
In [64]: from sklearn.metrics import accuracy_score
         perc_acc = accuracy_score(y_val, perc_model.predict(X_val))
         lr_acc = accuracy_score(y_val, lr_model.predict(X_val))
         svm_acc = accuracy_score(y_val, svm_model.predict(X_val))
         print("ACCURACY")
         print("Perceptron:           {}".format(perc_acc))
         print("Logistic Regression:  {}".format(lr_acc))
         print("SVM:                  {}".format(svm_acc))
```

```
ACCURACY
Perceptron:           1.0
Logistic Regression:  0.98
SVM:                  0.98
```

Misuriamo poi la f1 **score**

```
In [65]: print("F1 SCORE")
         print("Perceptron:          {}".format(f1_score(y_val, perc_model.predict(X_val), ave
         print("Logistic Regression: {}".format(f1_score(y_val, lr_model.predict(X_val), averag
         print("SVM:                 {}".format(f1_score(y_val, svm_model.predict(X_val), avera
```

```
F1 SCORE
Perceptron:          1.0
Logistic Regression: 0.978494623655914
SVM:                 0.978494623655914
```

Sembra che il modello che fa uso del Perceptron sia leggermente migliore degli altri.

Ciò che possiamo fare ora è stimare l'accuratezza di ciascuno di questi modelli sui dati futuri, conoscendo l'accuratezza di questi sul test set (nel nostro caso chiamato validation set, ma funge

da test set) e la cardinalità di questo insieme. Stabilita la confidenza con la quale vogliamo determinare l'accuratezza futura, ciò che otterremo sarà un intervallo all'interno del quale questa si troverà.

Quello che stiamo facendo è modellare la classificazione come un processo di Bernoulli, dunque approssimiamo la distribuzione di f (l'accuratezza sul test set) a una distribuzione normale standardizzata con media p, dove p è proprio l'accuratezza che si vuole determinare. Per comprendere quanto segue si sappia unicamente che la funzione `accuracy_interval` permette proprio di ricavare l'intervallo all'interno del quale p si troverà

Definiamo una funzione che ci permetta di ricavare il parametro Z a partire dalla confidenza desiderata

```
In [66]: from scipy import stats as st
         def get_Z(conf):
             alpha = 1 - conf
             alpha2 = alpha / 2
             return st.norm.ppf(conf + alpha2)
```

Definiamo infine una funzione che ci permetta di ricavare l'intervallo di accuratezza futura

```
In [67]: import math
         def accuracy_interval(acc, N, conf=0.95):
             Z = get_Z(conf)
             Z2 = Z*Z
             p1 = (2*N*acc + Z2 - Z*math.sqrt(Z2 + 4*N*acc - 4*N*(acc*acc))) / (2*(N + Z2))
             p2 = (2*N*acc + Z2 + Z*math.sqrt(Z2 + 4*N*acc - 4*N*(acc*acc))) / (2*(N + Z2))
             return (p1, p2)
```

Ricaviamo, per ciascuno dei nostri modelli, l'intervallo all'interno del quale cadrà l'accuratezza futura con una confidenza (cioè con una probabilità) del 95%

```
In [68]: print("FUTURE ACCURACY INTERVAL")
         print("Perceptron:          {}".format(accuracy_interval(perc_acc, len(X_val))))
         print("Logistic Regression: {}".format(accuracy_interval(lr_acc, len(X_val))))
         print("SVM:                 {}".format(accuracy_interval(svm_acc, len(X_val))))
```

```
FUTURE ACCURACY INTERVAL
Perceptron:          (0.9286524008666412, 1.0)
Logistic Regression: (0.8950455641036218, 0.9964607407283538)
SVM:                 (0.8950455641036218, 0.9964607407283538)
```

Come prevedibile, il modello che fa uso del Perceptron sembra poco più avvantaggiato. Ciò che possiamo fare ora è stabilire se la differenza tra questo modello e gli altri due sia statisticamente significativa. Modelliamo dunque la differenza d tra le accuratezze con una distribuzione normale: $d \sim N(d_t, \sigma_t)$ e definiamo una funzione che ci permetta di ricavare l'intervallo all'interno del quale si troverà d_t

```
In [69]: def get_difference_interval(e1, e2, n1, n2, conf=0.95):
             Z = get_Z(conf)
```

```
    d = abs(e1 - e2)
    sigma2 = (e1*(1 - e1))/n1 + (e2*(1 - e2))/n2
    d1 = d - Z*math.sqrt(sigma2)
    d2 = d + Z*math.sqrt(sigma2)
    return (d1, d2)
```

Ricaviamo tale intervallo per il modello che fa uso del Perceptron e quello che fa uso di regressione logistica, ciò che otteniamo è che l'intervallo contiene 0 e dunque la differenza tra i due modelli non è statisticamente significativa, ossia è solo frutto del caso. Essendo che la cardinalità del test set è appena di 50 osservazioni, questo risultato era facilmente intuibile.

Si noti che misurando la differenza tra il modello che fa uso del perceptron e quello basato su svm otterremmo lo stesso risultato in quanto l'accuratezza di quest'ultimo sul test set è identica all'accuratezza del modello basato su regressione logistica

```
In [70]: get_difference_interval(1 - perc_acc, 1 - lr_acc, len(X_val), len(X_val))

Out[70]: (-0.01880530708179097, 0.058805307081791006)
```

Avendo dimostrato che il leggero vantaggio del modello basato sul perceptron è solo frutto del caso, la scelta del modello finale potrebbe ricadere indifferentemente su uno qualsiasi di questi. Nel nostro caso scegliamo appunto il modello che fa uso del perceptron.

Una scelta forse più furba è quella di usare congiuntamente i tre modelli per le previsioni future, avvalendosi dell'*ensembling*

1.5 Interpretazione della conoscenza

Vogliamo ora vedere *quali* feature sono più influenti nel modello scelto, per fare questo osserviamo i coefficenti di ciascuno dei tre iperpiani trovati. Per semplicità consideriamo unicamente i coefficenti relativi ai termini di primo grado e ai termini che rappresentano la correlazione tra quelli di primo grado

Consideriamo il primo iperpiano: questo separa le osservazioni di *iris virginica* dalle altre, affianchiamo i coefficenti trovati ai nomi delle feature

```
In [71]: indx = X_train.columns.append(pd.Index(["SepalLengthCm^2",
                                                "SepalLengthCm * SepalWidthCm",
                                                "SepalLengthCm * PetalLengthCm",
                                                "SepalLengthCm * PetalWidthCm",
                                                "SepalWidthCm^2",
                                                "SepalWidthCm * PetalLengthCm",
                                                "SepalWidthCm * PetalWidthCm",
                                                "PetalLengthCm^2",
                                                "PetalLengthCm * PetalWidthCm"]))

In [72]: coefs = perc_model.named_steps["perc"].coef_
```

Si ricorda che questi coefficenti sono standardizzati, per interpretarli nel dominio originale vanno destandardizzati. A noi vanno bene così in quanto ci permettono di vedere facilmente quali sono le feature che "pesano" di più, indipendentemente dalla scala di valori che queste assumo nel dominio originale.

Nel caso del primo iperpiano le tre caratteristiche maggiormente influenti sono: lunghezza del sepalo, correlazione tra lunghezza e larghezza del sepalo e correlazione tra le larghezze dei due tratti. Dunque le decisioni vengono prese considerando per lo più il **sepalo**

```
In [73]: pd.Series(coefs[0][:13], indx).drop(["SepalLengthCm^2", "SepalWidthCm^2", "PetalLengt
```

```
Out[73]: SepalLengthCm                  -2.536656
         SepalWidthCm                   -0.046627
         PetalLengthCm                  -0.662590
         PetalWidthCm                   -0.916573
         SepalLengthCm * SepalWidthCm   -2.097470
         SepalLengthCm * PetalLengthCm  -0.350683
         SepalLengthCm * PetalWidthCm   -0.684804
         SepalWidthCm * PetalLengthCm   -1.844888
         SepalWidthCm * PetalWidthCm    -1.989984
         PetalLengthCm * PetalWidthCm    1.192462
         dtype: float64
```

Nel caso del secondo iperpiano le tre caratteristiche maggiormente influenti sono: larghezza del sepalo, lunghezza del petalo e correlazione tra le lunghezze dei due tratti. In questo caso non c'è un tratto che prevale sull'altro per importanza

```
In [74]: pd.Series(coefs[1][:13], indx).drop(["SepalLengthCm^2", "SepalWidthCm^2", "PetalLengt
```

```
Out[74]: SepalLengthCm                  -1.080910
         SepalWidthCm                    2.423661
         PetalLengthCm                  -1.474827
         PetalWidthCm                   -0.927312
         SepalLengthCm * SepalWidthCm    0.824497
         SepalLengthCm * PetalLengthCm  -1.157630
         SepalLengthCm * PetalWidthCm   -0.757020
         SepalWidthCm * PetalLengthCm   -0.977521
         SepalWidthCm * PetalWidthCm    -0.706687
         PetalLengthCm * PetalWidthCm   -0.536565
         dtype: float64
```

Nel caso del terzo iperpiano le tre caratteristiche maggiormente influenti sono: larghezza del petalo, lunghezza del petalo e correlazione tra le larghezze dei due tratti. In questo caso le informazioni più significative provengono dal **petalo**

```
In [75]: pd.Series(coefs[2][:13], indx).drop(["SepalLengthCm^2", "SepalWidthCm^2", "PetalLengt
```

```
Out[75]: SepalLengthCm                   3.493118
         SepalWidthCm                   -4.448004
         PetalLengthCm                   7.707965
         PetalWidthCm                    8.023567
         SepalLengthCm * SepalWidthCm   -0.418566
         SepalLengthCm * PetalLengthCm   3.624429
         SepalLengthCm * PetalWidthCm    4.516591
```

```
SepalWidthCm * PetalLengthCm     6.810834
SepalWidthCm * PetalWidthCm      7.645771
PetalLengthCm * PetalWidthCm     0.727281
dtype: float64
```

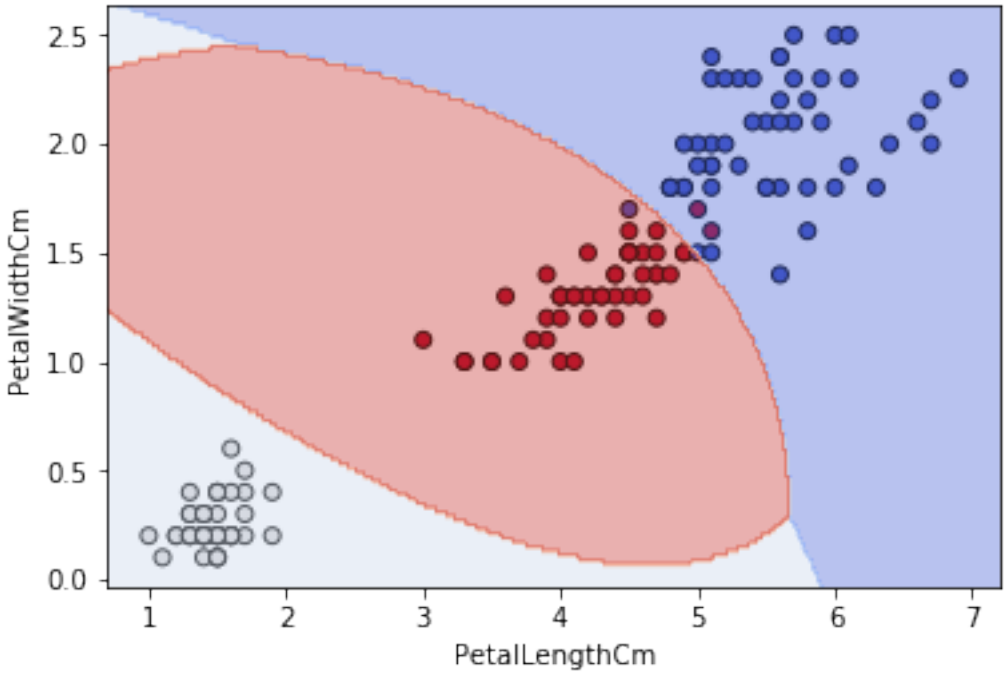
1.6 Interpretazione geometrica

Come ultima cosa, possiamo pensare di allenare nuovamente i nostri modelli sul DataFrame `iris_petal`, in modo da fornire un'interpretazione geometrica di questi

```
In [76]: X_train, X_val, y_train, y_val = train_test_split(
             iris_petal.drop(["Species"], axis=1),
             iris_petal["Species"],
             test_size=1/3, random_state=42
         )

In [77]: perc_model.fit(X_train, y_train)
         perc_model.score(X_val, y_val)

Out[77]: 1.0
```

Vediamo i decision boundaries del modello basato su perceptron

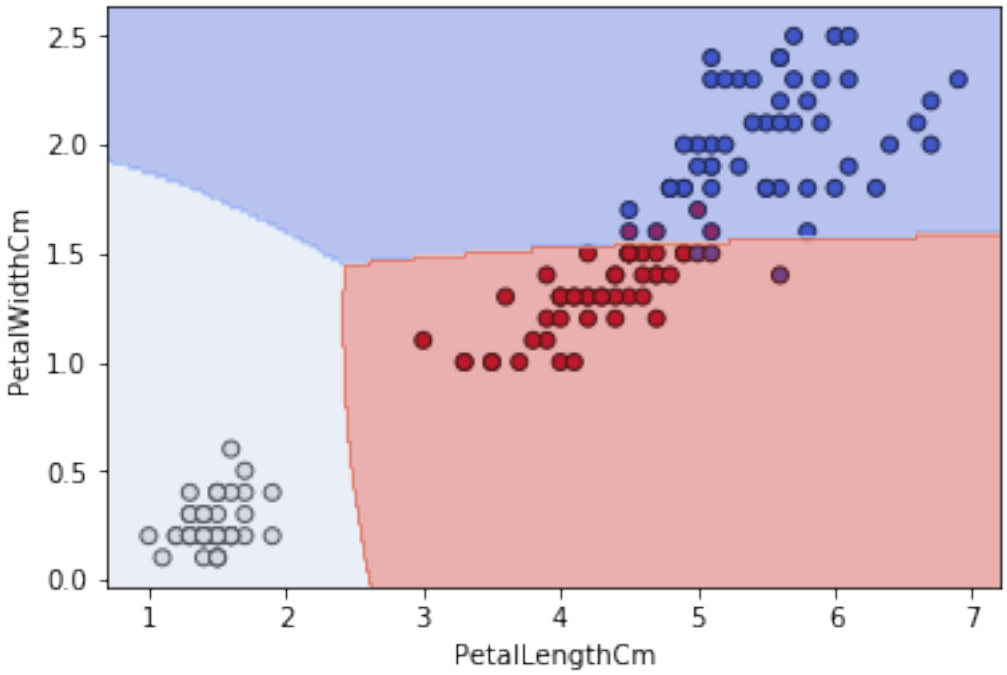
```
In [78]: plot_decision_boundaries(perc_model.predict)
```



```
In [79]: lr_model.fit(X_train, y_train)
         lr_model.score(X_val, y_val)

Out[79]: 0.94
```

Vediamo i decision boundaries del modello basato su regressione logistica

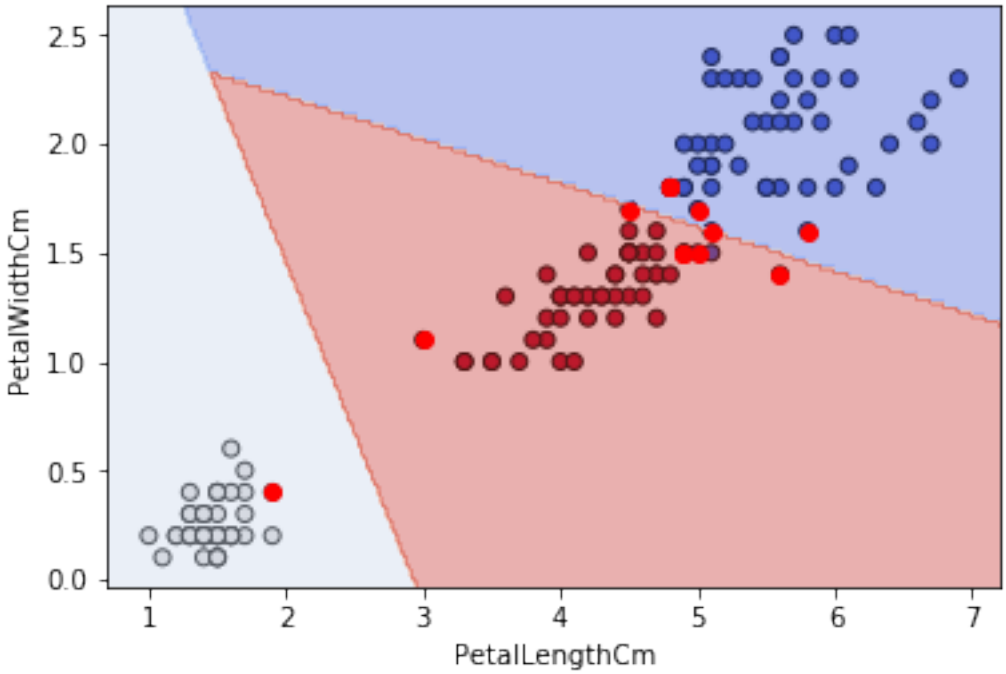
```
In [80]: plot_decision_boundaries(lr_model.predict)
```




```
In [81]: svm_model.fit(X_train, y_train)
         svm_model.score(X_val, y_val)

Out[81]: 0.98
```

Per il modello basato su svm, evidenziamo di rosso i support vectors, ossia i punti più vicini ai bordi, che vengono usati per effettuare le predizioni

```
In [82]: plot_decision_boundaries(svm_model.predict)
         for point in svm_model.named_steps["svc"].support_vectors_:
             plt.plot(point[0], point[1], 'ro')
```



Osserviamo che i tre modelli risultano meno aderenti ai dati rispetto al primo modello, possiamo pensare che saranno più precisi di questo sui dati futuri

1.7 Appendice: ensembling

Definiamo una funzione che ci permetta di predire la specie di un'osservazione x usando i tre modelli descritti sopra in modo congiunto. La tecnica di ensembling qui è del tutto banale: viene definita la probabilità di appartenenza a ogni classe facendo la media delle probabilità espresse dai tre modelli per tale classe, la classe assegnata è quella con maggiore probabilità

```
In [83]: def ensemble_predict(x):
             perc_prob = perc_model.decision_function(x)[0]
             lr_prob = lr_model.decision_function(x)[0]
             svm_prob = svm_model.decision_function(x)[0]
             prob_c1 = (perc_prob[0] + lr_prob[0] + svm_prob[0])/3
             prob_c2 = (perc_prob[1] + lr_prob[1] + svm_prob[1])/3
             prob_c3 = (perc_prob[2] + lr_prob[2] + svm_prob[2])/3
             max_prob = prob_c1
             c = 1
             if prob_c2 > max_prob:
                 max_prob = prob_c2
                 c = 2
             if prob_c3 > max_prob:
                 max_prob = prob_c3
                 c = 3
             return c
```

```
def ensemble_predict_aggregate(X):
    y = []
    for i in range(0, len(X)):
        if isinstance(X, pd.DataFrame):
            y.append(ensemble_predict(X.iloc[[i], :]))
        else:
            y.append(ensemble_predict(X[[i], :]))
    return np.array(y)
```

Visualizziamo i decision boundaries ottenuti dall'ensembling dei tre modelli

```
In [84]: plot_decision_boundaries(ensemble_predict_aggregate)
```