Classification Models

Lecture 01

Douglas Chielle e Lucas Sousa
douglas.chielle@ifce.edu.br / lucas.sousa@ifce.edu.br

Content

1. Logistic Regression
2. Softmax Regression
3. K-Nearest Neighbors
4. Naive Bayes classifiers

Objectives

Upon completion of this lecture, you will be able to:

- Understand the concepts of Logistic Regression, Softmax Regression, K-Nearest Neighbors, and Naive Bayes classifiers;
- Build these models with scikit-learn and use them for classification.

Logistic Regression

Logistic Regression

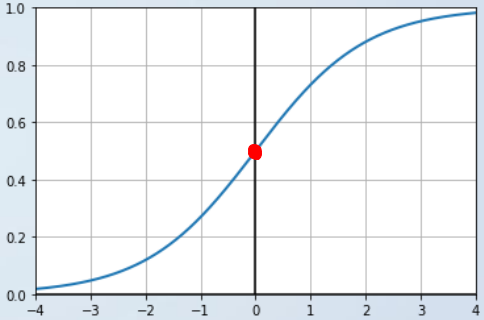
- *Logistic Regression (or Logit* Regression) is a binary classification algorithm used to estimate the probability that an instance belongs to a certain class.
- It is used for situations like pass/fail, win/lose, alive/dead or healthy/sick.

Steps in Logistic Regression

1. Compute a weighted sum of the input features plus bias term ($x^T \theta$);
2. Apply the ***logistic function*** on that sum

$$x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$\hat{p} = \sigma(\mathbf{x}^T \boldsymbol{\theta}) = \frac{1}{1 + e^{-x^T \theta}}$$



3. Predict $\hat{y}$:

$$\hat{y} = \begin{cases} 0 & if \quad \sigma(\mathbf{x}^T \boldsymbol{\theta}) < 0.5 \\ 1 & if \quad \sigma(\mathbf{x}^T \boldsymbol{\theta}) \geq 0.5 \end{cases}$$

Training and Cost Function

- The goal in training is to set the parameters ($\boldsymbol{\theta}$) so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$).
- The cost function used in Logistic Regression is

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \ln\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right) \ln\left(1 - \hat{p}^{(i)}\right)\right]$$

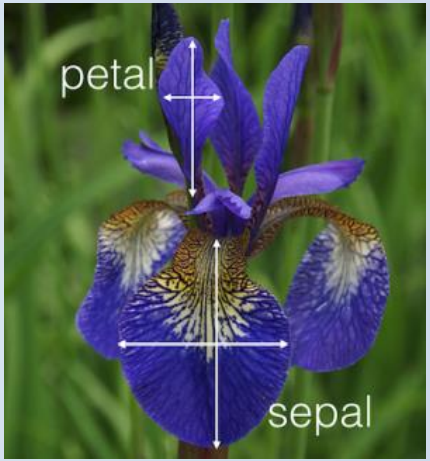
Training and Cost Function

- There is no closed-form equation for this cost function, but it is convex.
- Therefore, Gradient Descent (or any optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too high and you wait long enough).
- The partial derivatives of the cost function w.r.t. the j^{th} model parameter θ_j are given by

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma\left(\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} \right) - y^{(i)} \right) x_j^{(i)}$$

Example

- Let’s see an example of classification with Logistic Regression.
- For this example, we will use the “iris dataset”.
- There are 150 samples in this dataset, 50 in each of the three classes.
- There are four numeric features:
  - sepal length in cm
  - sepal width in cm
  - petal length in cm
  - petal width in cm
- The classes are: *Iris-Setosa*, *Iris-Versicolour* and *Iris-Virginica*



Example

- **Loading the data**

```
import numpy as np
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data[:,2:] # petal length and width in cm
y = (iris.target == 2).astype(np.int) # 1 if Iris virginica, else 0
```

- **Splitting the data into training and test set**

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.80,
 random_state=7)
```

Example

- **Building a Logistic Regression model:**

```
from sklearn.linear_model import LogisticRegression

Logistic = LogisticRegression(penalty='none')
Logistic.fit(X_train, y_train)
```

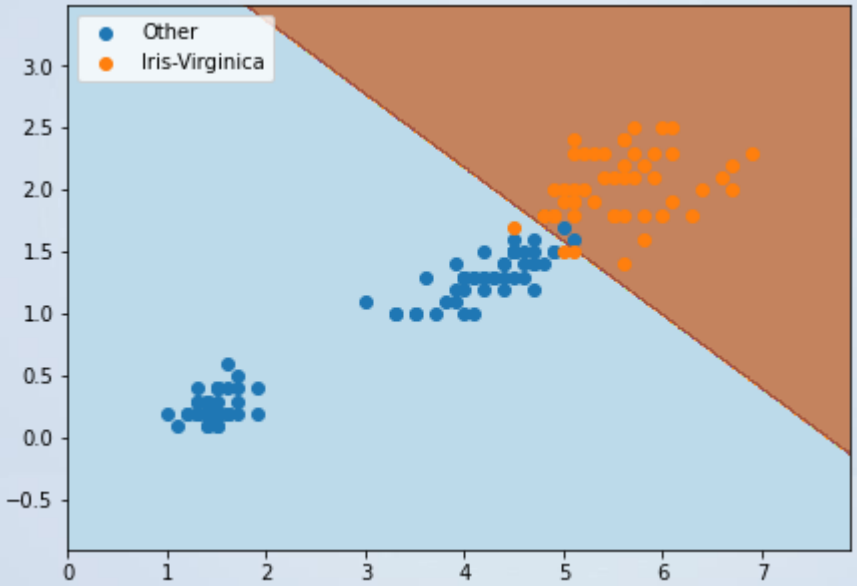
- **Checking its accuracy on the training and test set:**

```
acc_train = Logistic.score(X_train, y_train)
acc_test = Logistic.score(X_test, y_test)
```

- **Printing those values, we get:**
  - Accuracy on training set = 0.98
  - Accuracy on test set = 0.87

Example

- **Decision surface:**



Softmax Regression

Softmax Regression

- Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes.
- Given a test input x, we want to estimate the probability that x belongs to class k, for each $k = 1,\ 2, \cdots, K$.
- To compute that probability, we use the ***softmax function***:

$$\hat{p}_k = \sigma\left(\boldsymbol{x}^T \boldsymbol{\theta}^{(k)}\right)_k = \frac{e^{\boldsymbol{x}^T \boldsymbol{\theta}^{(k)}}}{\sum_{j=1}^{K} e^{\boldsymbol{x}^T \boldsymbol{\theta}^{(j)}}}$$

- The predicted class will be the class with the highest probability.

Training and Cost Function

- The goal in training is to have a model that estimates a high probability for the target class (and a low probability for the other classes).
- The cost function used in Softmax Regression is the *cross-entropy*:

$$J(\boldsymbol{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)} \ln\left(\hat{p}_k^{(i)}\right)\right]$$

$$y_k^{(i)} = 1\left\{y^{(i)} = k\right\}$$

- The gradient vector of this cost function w.r.t. $\boldsymbol{\theta}^{(k)}$ is given by

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\boldsymbol{x}^{(i)}$$

  - $y_k^{(i)}$ is the target probability that the i^{th} instance belongs to class k. In general, it is either equal to 1 or 0.

Example

- **Loading the data**

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data[:,2:] # petal length and width in cm
y = iris.target
```

- **Splitting the data into training and test set**

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.80,
 random_state=7)
```

Example

- **Building a Logistic Regression model:**

```
from sklearn.linear_model import LogisticRegression

Softmax = LogisticRegression(penalty='none', max_iter=1000)
Softmax.fit(X_train, y_train)
```

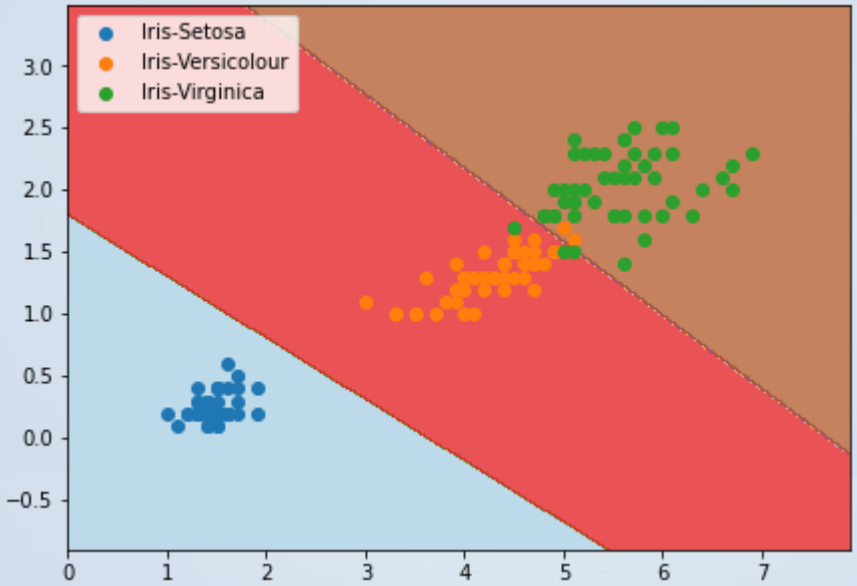
- **Checking its accuracy on the training and test set:**

```
acc_train = Logistic.score(X_train, y_train)
acc_test = Logistic.score(X_test, y_test)
```

- **Printing those values, we get:**
  - Accuracy on training set = 0.98
  - Accuracy on test set = 0.87

Example

- **Decision surface:**



K-Nearest Neighbors

K-Nearest Neighbors (KNN)

- *"Tell me who you walk with, and I'll tell you who you are."* (Esmeralda Santiago)
- KNN is one of the simplest algorithms in Machine Learning.
- For classification of an instance $\boldsymbol{x}$, KNN checks the class of the k nearest elements to $\boldsymbol{x}$ and will classify $\boldsymbol{x}$ by a plurality vote.
- For regression, KNN will output the average of the target values of the k nearest elements.

K-Nearest Neighbors (KNN)

- Usually, the class of an instance is computed from a simple majority vote of the nearest neighbors.
- However, it is also possible to compute it as a weighted vote, so that the vote of the nearer neighbors contribute more to the decision.
- A common way of doing this is to assign weights proportional to the inverse of the distance from the point being classified.

Exampl

- Let's see how KNN performs on the "iris dataset", considering both the case with two classes and the one with three classes.
- We will use the same exact commands to build the model and check its accuracy.

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5 , weights='uniform')
knn.fit(X_train, y_train)

acc_train = knn.score(X_train, y_train)
acc_test = knn.score(X_test, y_test)
```

distance

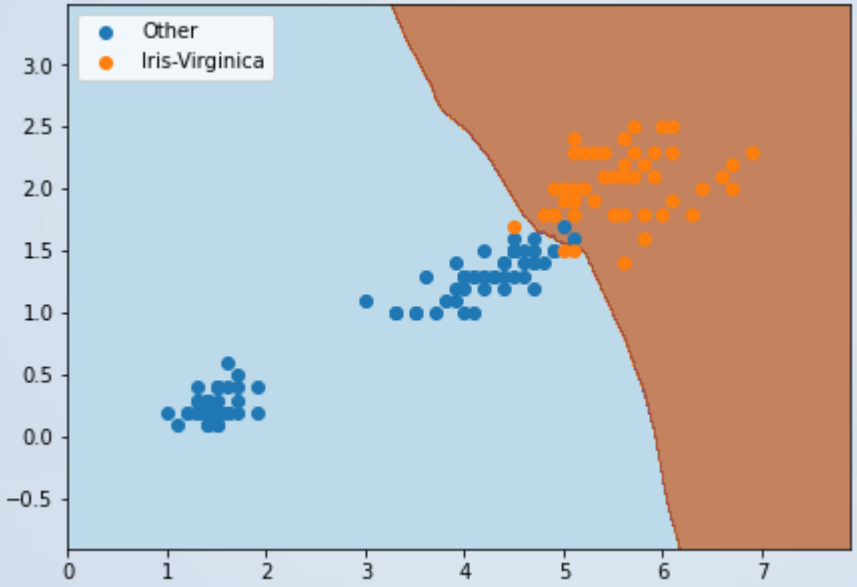
Example

- Here is the model's accuracy for the binary case:
  - Accuracy on training set = 0.98
  - Accuracy on test set = 0.87

- And for the case with three classes:
  - Accuracy on training set = 0.98
  - Accuracy on test set = 0.87

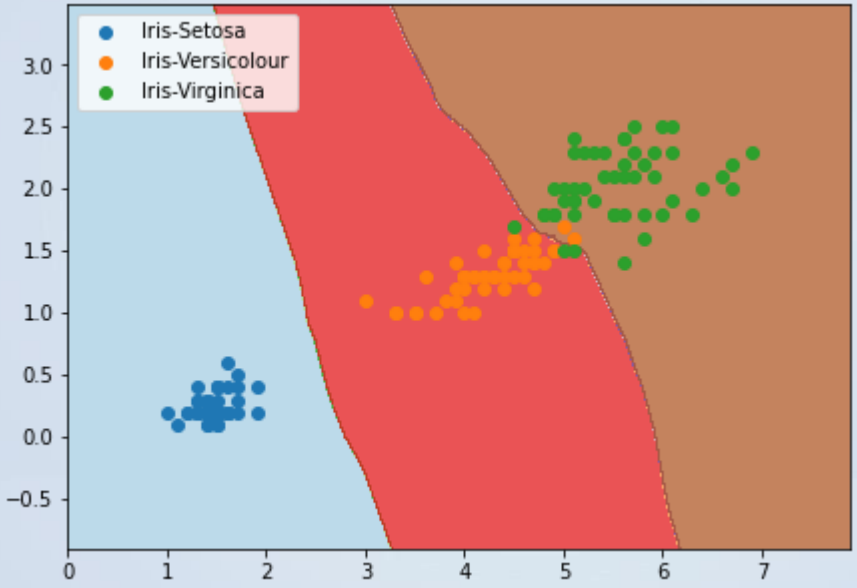
Example

- Decision surface: two classes



Example

- Decision surface: three classes



Naive Bayes Classifier

Naive Bayes classifier

- A naive Bayes classifier is a simple probabilistic classifier based on applying Bayes theorem where every feature is assumed to be *class-conditionally independent*.
- This assumption is made to simplify the computation, and, in this sense, it is considered to be naive.
- Despite its “naive” assumptions, this classifier works well in many real-world problems, like document classification and spam filtering.

Naive Bayes classifier

- Bayes Theorem

$$P(C_k \mid x_1, x_2, \cdots, x_n) = \frac{P(C_k)P(x_1, x_2, \cdots, x_n \mid C_k)}{P(x_1, x_2, \cdots, x_n)}$$

- Assuming the features are independent, given the value of the class variable, the probability of the numerator is simply

$$P(C_k)\prod_{i=1}^{n} P(x_i \mid C_k)$$

- The predicted class will be the one with the highest posterior probability.

Types of Naive Bayes Classifier

- The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid C_k)$.
  - **Multinomial Naive Bayes**:
    - A feature vector $(x_1, \cdots, x_n)$ is a histogram, with x_i counting the number of times event i was observed in a particular instance.
    - It is typically used for document classification, with events representing the occurrence of a word in a single document.
  - **Bernoulli Naive Bayes**:
    - This is like the multinomial Naive Bayes, but the predictors are Boolean (yes/no) variables.
    - Also popular for document classification tasks.
  - **Gaussian Naive Bayes:**
    - Features are continuous values, and it is assumed that they follow a normal distribution.

Laplace Smoothing

- Problems arise when an attribute value in the testing set has no example in the training set.
- In this case, because it has never seen this value before in the training examples, it thinks the probability of seeing it in any class is zero. Therefore, the algorithm will not know hot to make a prediction.
- It is statistically a bad idea to estimate the probability of some event to be zero just because you haven’t seen it before in your finite training set.
- Therefore, it is often desirable to incorporate a small-sample correction, in all probability estimates such that no probability is ever set to be exactly zero.

Laplace Smoothing

- For the Naive Bayes classifier, the posterior probability is:

$$P(C_k \mid x_1, x_2, \cdots, x_n) = \frac{P(C_k) \prod_{i=1}^{n} P(x_i \mid C_k)}{P(x_1, x_2, \cdots, x_n)}$$

- With Laplace Smoothing, it will be:

$$P(C_k \mid x_1, x_2, \cdots, x_n) = \frac{P(C_k) \prod_{i=1}^{n} P(x_i \mid C_k) + \alpha}{P(x_1, x_2, \cdots, x_n) + \alpha K}$$

Usually $\alpha = 1$.

Example – Gaussian NB

- Let's see how Gaussian NB performs on the "iris dataset", considering both the case with two classes and the one with three classes.
- We will use the same exact commands to build the model and check its accuracy.

```
from sklearn.naive_bayes import GaussianNB

gNB = GaussianNB()
gNB.fit(X_train, y_train)

acc_train = gNB.score(X_train, y_train)
acc_test = gNB.score(X_test, y_test)
```

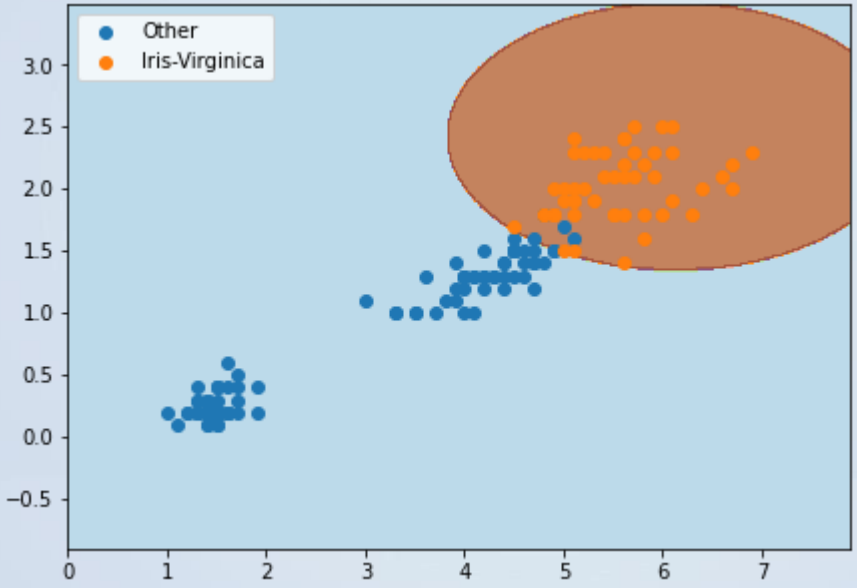
Example – Gaussian NB

- Here is the model's accuracy for the binary case:
  - Accuracy on training set = 0.98
  - Accuracy on test set = 0.93

- And for the case with three classes:
  - Accuracy on training set = 0.98
  - Accuracy on test set = 0.87

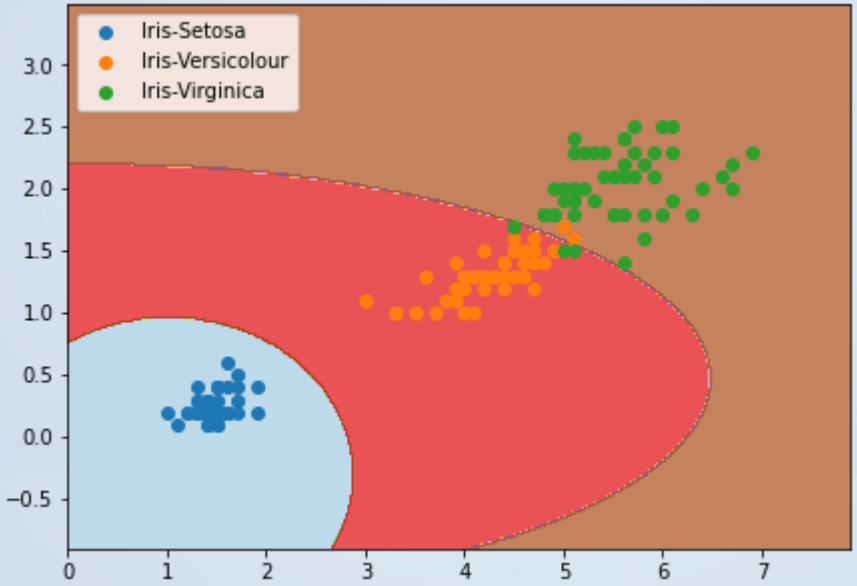
Example

- Decision surface: two classes



Example

- Decision surface: three classes



Example – Multinomial NB

- As this version of the Naive Bayes classifier is mostly used for document classification, let’s load some text data.
- For this example, we will use the "20 newsgroups text dataset”
- This dataset contains 18846 samples divided into 20 topics (classes).
- The data is already divided in two datasets: one for training and the other for testing.

Example – Multinomial NB

- **Loading the data:**

```
from sklearn.datasets import fetch_20newsgroups

categories = ['comp.graphics', 'rec.autos', 'rec.motorcycles',
              'rec.sport.baseball', 'rec.sport.hockey', 'sci.electronics',
              'sci.med', 'sci.space']

twenty_train = fetch_20newsgroups(subset='train', categories=categories,
                                  remove=('headers', 'footers', 'quotes'),
                                  random_state=7)

twenty_test = fetch_20newsgroups(subset='test', categories=categories,
                                 remove=('headers', 'footers', 'quotes'),
                                 random_state=7)
```

Example – Multinomial NB

- We cannot feed text directly to our algorithm.
- So, before building the model, we will convert our raw data to a matrix of token counts using the `CountVectorizer` function:

```
from sklearn.feature_extraction.text import CountVectorizer

vect = CountVectorizer()

X_train = vect.fit_transform(twenty_train.data)
X_test = vect.transform(twenty_test.data)

y_train = twenty_train.target
y_test = twenty_test.target
```

Example – Multinomial NB

- Now we can build our model and evaluate its performance:

```
from sklearn.naive_bayes import MultinomialNB

mNB = MultinomialNB()
mNB.fit(X_train, y_train)

acc_train = mNB.score(X_train, y_train)
acc_test = mNB.score(X_test, y_test)
```

- Here is the model accuracy on the training and test set:
  - Accuracy on training set = 0.91
  - Accuracy on test set = 0.80

References

1. Géron, Aurélien. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.,
2. Rennie, Jason D., et al. "*Tackling the poor assumptions of naive bayes text classifiers.*" Proceedings of the 20th international conference on machine learning (ICML-03). 2003.
3. http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression/


That's all Folks!