# Table of Contents

# 1

# 4. Predicting categories with Naive Bayes and SVMs

In this chapter you will learn about two popular classification machine learning algorithms: The Naive Bayes & the Linear Support Vector Machines. The Naive Bayes algorithm is a probabilistic model that works well at predicting classes/categories while the linear support vector machines uses a linear decision boundary in order to predict classes/categories.

In this chapter you will learn about:

- The theoretical concept behind the Naive Bayes Algorithm explained in mathematical terms
- Implementing the Naive Bayes algorithm in scikit-learn
- How the linear support vector machines algorithm works under-the-hood
- Optimizing the hyper-parameters of the linear SVMs graphically

## The Naive Bayes Algorithm

The Naive Bayes algorithm makes use of the Bayes theorem in order to classify classes/categories. The word 'Naive' is given to the algorithm because the algorithm assumes that all attributes are independent of each other. This is not realistically possible as every attribute/feature in a dataset is related to each other in some way or the other.

Despite being 'Naive' the algorithm does well in actual practice. The formula for Bayes theorem is given below:

$$\boxed{p(h|\mathcal{D}) = \frac{p(\mathcal{D}|h)p(h)}{p(\mathcal{D})}}$$

Bayes Theorem Formula

We can split the algorithm displayed above into the following components:

- **p(h|D):** This is the probability of some hypothesis taking place provided that we have a dataset. An example of this is - The probability of a fraud transaction taking place provided that we have a dataset that consists of fraudulent & non-fraudulent transactions.
- **p(D|h):** This is the probability of having the data given some hypothesis. An example of this is - The probability of having a dataset that contains fraudulent transactions.
- **p(h):** This is the probability of the hypothesis taking place in general. An example of this is - The average probability of fraudulent transactions taking place in the mobile industry is 2%.
- **p(D):** This is probability of having the data before knowing any hypothesis. An example of this is - The probability that a dataset of mobile transactions can be found without knowing what we want to do with it (i.e Predict Fraud Mobile Transactions)

In the formula above the the **p(D)** can be re-written in terms of **p(h) and p(D|h)** as follows:

$$\boxed{p(h|\mathcal{D}) = \frac{p(\mathcal{D}|h)p(h)}{p(\mathcal{D}|h)p(h) + p(\mathcal{D}|\neg h)(1 - p(h))}}$$

Let's have a look at how we can implement this in the method of predicting classes in the case of the mobile transaction example:

| **p(D\|h)** | **p(h)** | **p(D\|-h)** | (1 - p(h)) |
|---|---|---|---|
| 0.8 | 0.08 | 0.02 | 0.92 |

Substituting the values in the table above in the Bayes Theorem formula produces a result of 0.77. This means that the classifier predicts that there is a 77% probability that a transaction will be predicted as fraud with the data given above.

## Implementing the Naive Bayes Algorithm in scikit-learn

Now that you have learnt how the Naive Bayes Algorithm generates predictions we will now implement the same classifier using scikit-learn in order to predict if a particular transaction is fraud or not.

The first step is import the data, create the feature & target arrays and split the data into training and test sets.

We can do this by using the code shown below:

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('fraud_prediction.csv')

df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features

features = df.drop('isFraud', axis = 1).values
target = df['isFraud'].values

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42, stratify = target)
```

The next step is to build the Naive Bayes classifier. We can do this using the code shown below:

```
from sklearn.naive_bayes import GaussianNB

#Initializing an NB classifier

nb_classifier = GaussianNB()

#Fitting the classifier into the training data

nb_classifier.fit(X_train, y_train)

#Extracting the accuracy score from the NB classifier

nb_classifier.score(X_test, y_test)
```

In the code above:

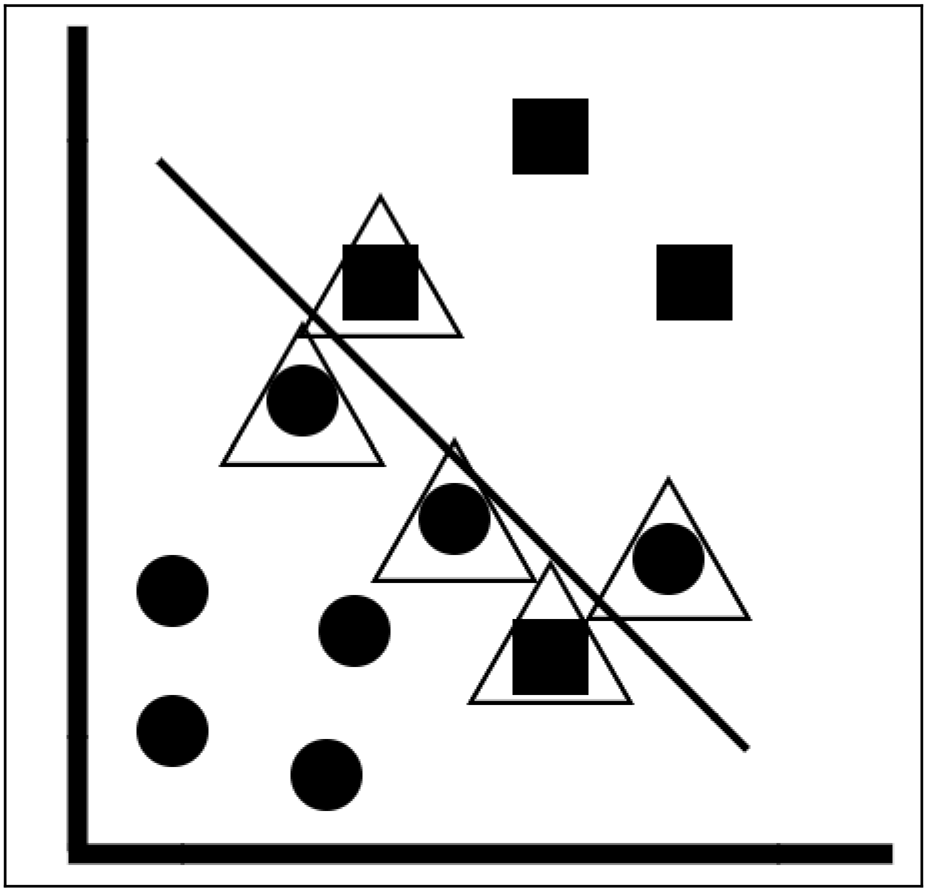
1. We first import the *GaussianNB* module from scikit-learn.
2. Next, we initialize a Naive Bayes Classifier and store it in the variable - "nb_classifier".
3. We then fit the classifier to the training data and evaluate it's accuracy on the test data.

The Naive Bayes Classifier has only one hyper-parameter - which is the prior probability of the hypothesis **p(h).** However:

1. The prior probability is not available to us  in most problems.
2. Even if it is, the value is usually fixed as a statistical fact & therefore hyper-parameter optimization is not done.

## Support Vector Machines (SVMs)

In this section, we will learn about the Support Vector Machines or in specific the Linear Support Vector Machines. In order to understand Support Vector Machines we need to know what Support Vectors are. This is illustrated for you in the image shown below:

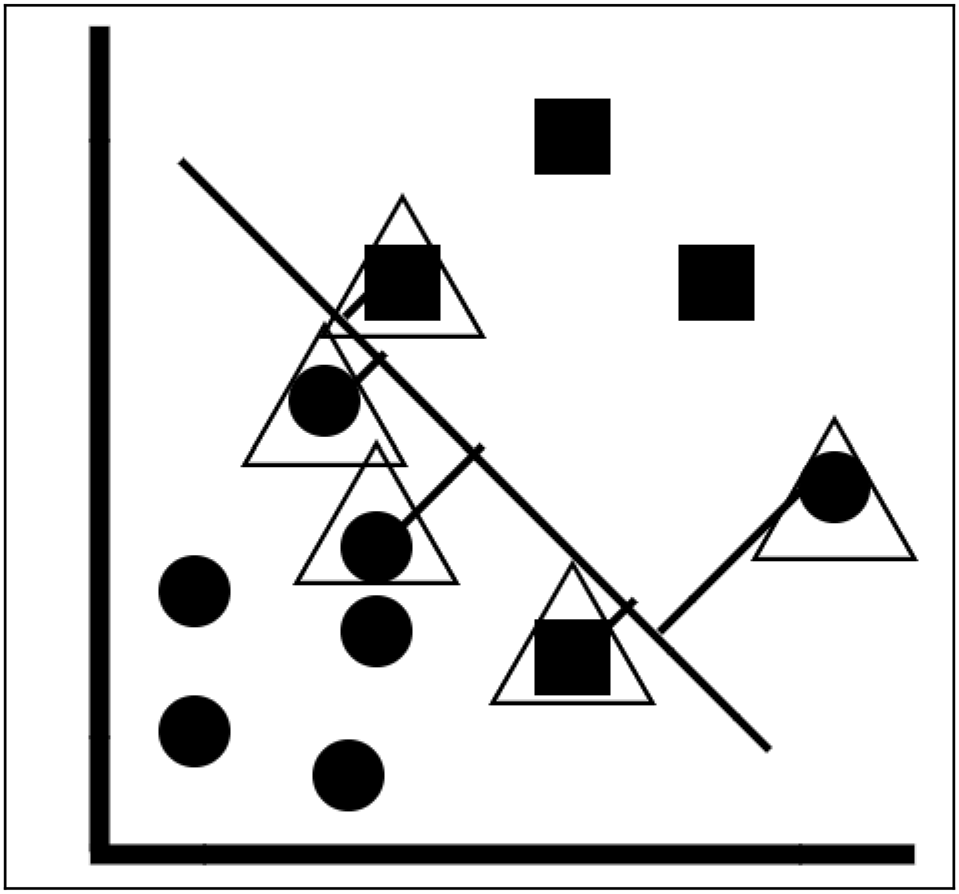


Concept of the support vectors

In the figure above:

1. The linear support vector machine is a form of linear classifier in which a linear decision tree boundary is constructed in which the observations on one side of the boundary (circles) belong to one class while the observations on the other side of the boundary (squares) belong to another class.
2. The support vectors are the observations that have a triangle on them.
3. These are the observations that are either very close to the linear decision boundary or the observations that have be in-correctly classified.
4. We can define which observations we want to make support vectors by defining how close to the decision boundary they should be.
5. This is controlled by the hyper-parameter known as the **inverse regularization strength.**

In order to understand how the Linear Support Vector Machines work we consider the image shown below:



Concept of Max-Margins

In the figure above:

1. The line between the support vectors and linear decision boundary is known as the margin.
2. The goal of the Support Vector Machines is to maximize this margin so that a new data point will be correctly classified.
3. A low value of inverse regularization strength ensures that this margin as big as possible.

## Implementing the linear support vector machine algorithm in scikit-learn

In this section, we will learn how we can implement the linear support vector machines in scikit-learn. The first step is import the data and split it into training & testing sets. We can do this by using the code shown below:

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('fraud_prediction.csv')

df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features

features = df.drop('isFraud', axis = 1).values
target = df['isFraud'].values

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42, stratify = target)
```

The next step is to build the Linear Support Vector Machine Classifier. We can do this using the code shown below:

```
from sklearn.svm import LinearSVC

#Initializing a SVM model

svm = LinearSVC(random_state = 50)

#Fitting the model to the training data

svm.fit(X_train, y_train)

#Extracting the accuracy score from the training data

svm.score(X_test, y_test)
```

In the code above:

1. We first import the *LinearSVC* module from scikit-learn.
2. Next, we initialize a Linear Support Vector Machine Object with a random state of 50 so that the model produces the same result every time.
3. Finally, we fit the model to the training data and evaluate it's accuracy on the test

data.

Now that we have built the model we can now find & optimize the most ideal value for the hyper-parameters.

## Hyper-parameter optimization for the linear SVMs

In this section we will learn how to optimize the hyper-parameters for the linear support vector machines. In particular there is only one hyper-parameter of interest - the **Inverse Regularization Strength.**

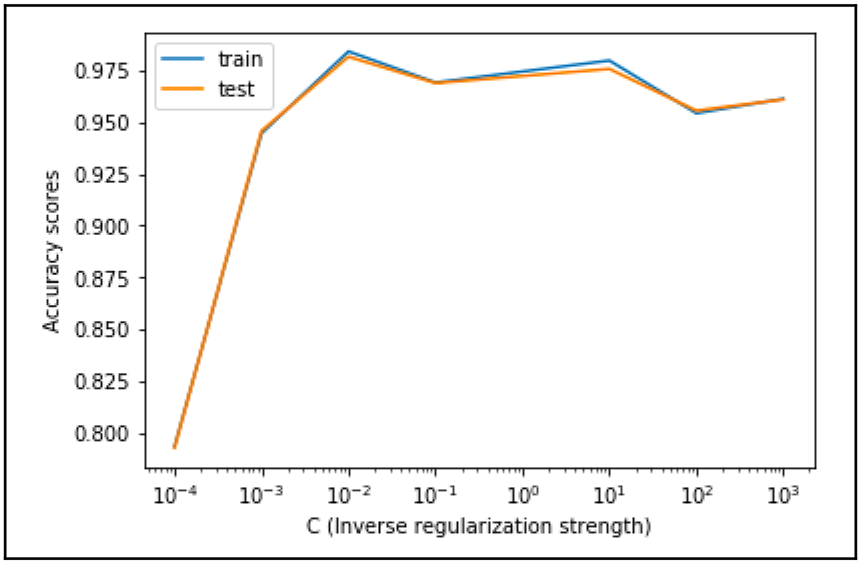
We will explore how to optimize this hyper-parameter both graphically & algorithmically.

### Graphical hyper-parameter optimization

In order to optimize the inverse regularization strength we will plot the accuracy scores for the training & testing sets by using the code shown below:

```
import matplotlib.pyplot as plt
from sklearn.svm import LinearSVC

training_scores = []
testing_scores = []

param_list = [0.0001, 0.001, 0.01, 0.1, 10, 100, 1000]

# Evaluate the training and test classification errors for each value of
the parameter

for param in param_list:
    # Create SVM object and fit
    svm = LinearSVC(C = param, random_state = 42)
    svm.fit(X_train, y_train)
    # Evaluate the accuracy scores and append to lists
    training_scores.append(svm.score(X_train, y_train) )
    testing_scores.append(svm.score(X_test, y_test) )
# Plot results

plt.semilogx(param_list, training_scores, param_list, testing_scores)
plt.legend(("train", "test"))
plt.ylabel('Accuracy scores')
plt.xlabel('C (Inverse regularization strength)')
plt.show()
```

This produces a plot as illustrated in the image shown below:



Graphical hyper-parameter optimization

In the code above:

1. We first initialize two empty lists in order to store the accuracy scores for both the training and testing data sets.
2. The next step is to create a list of values of the hyper-parameter which in this case is the inverse regularization strength.
3. We then loop over each value in the hyper-parameter list and build a linear support vector machine classifier with each inverse regularization strength value.
4. The accuracy scores for the training & testing data sets are then appended to the empty lists.
5. Using *matplotlib*, we then create a plot between the inverse regularization strength along the x-axis and the accuracy scores for both the training and test sets along the y-axis.

In the plot above:

1. We can observe that the accuracy score is highest for the training & testing sets for an inverse regularization strength of 10^-2.
2. It is important to pick a value that has a high value of accuracy for both the training & testing sets & not just either one of the data sets.

3. This helps you prevent both over-fitting & under-fitting.

## Hyper-parameter optimization using GridSearchCV

In this section you will learn how to optimize the inverse regularization strength using the GridSearchCV algorithm. We can do this using the code shown below:

```
from sklearn.model_selection import GridSearchCV

#Building the model

svm = LinearSVC(random_state = 50)

#Using GridSearchCV to search for the best parameter

grid = GridSearchCV(svm, {'C':[0.00001, 0.0001, 0.001, 0.01, 0.1, 10]})
grid.fit(X_train, y_train)

# Print out the best parameter

print("The best value of the inverse regularization strength is:",
grid.best_params_)
```

In the code above:

1. We first import the *GridSearchCV* module from scikit-learn.
2. The next step is to initialize a linear support vector machine model with a random state of 50 in order to ensure that we obtain the same results every time we build the model.
3. We then initialize a grid of possible hyper-parameter values for the inverse regularization strength.
4. Finally, we fit the grid of hyper-parameter values to the training set so that we can build multiple linear SVM models with the different values of the inverse regularization strength.
5. The GridSearchCV algorithm then evaluates the model that produces the least generalization error and then outputs the most optimal value of the hyper-parameter.

It's a good practice to compare and contrast the results of the graphical method of hyper-parameter optimization with that of GridSearchCV in order to validate your results.

## Scaling the data for performance improvement

In this section you will learn how scaling & standardization of the data could lead to an improvement in the overall performance of the linear Support Vector Machines. The concept of scaling remains the same as in the case of the previous chapters & will not be discussed here. In order to scale the data we use the code shown below:

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

#Setting up the scaling pipeline

order = [('scaler', StandardScaler()), ('SVM', LinearSVC(C = 0.1,
random_state = 50))]

pipeline = Pipeline(order)

#Fitting the classfier to the scaled dataset

svm_scaled = pipeline.fit(X_train, y_train)

#Extracting the score

svm_scaled.score(X_test, y_test)
```

In the code above:

1. We first import the *StandardScaler* & the *Pipeline* modules from scikit-learn in order to building a scaling pipeline.
2. We then set up the order of the pipeline which specifies that we use the *StandardScaler()* function first in order to scale the data and then build the Linear Support Vector Machine on the scaled data.
3. The *Pipeline()* function is applied to the the order of the pipeline which sets up the pipeline.
4. We then fit this pipeline to the training data and & extract the scaled accuracy scores from the test data.

## Summary

This chapter introduced you to two fundamental supervised machine learning algorithms - the Naive Bayes & the Linear Support Vector Machines. More specifically, you have learnt about:

1. How the Bayes Theorem is used to produce a probability to tell us if a data point belongs to a particular class/category
2. Implementing the Naive Bayes classifier in scikit-learn
3. How the Linear Support Vector Machines works under-the-hood.
4. Implementing the Linear Support Vector Machines in scikit-learn
5. Optimizing the inverse regularization strength both graphically & using the GridSearchCV algorithm.
6. Finally, you learnt how to scale your data for a potential improvement in performance.

In the next chapter, you will learn about the other type of supervised machine learning algorithm that is used to predict numeric values instead of classes/categories - the linear regression!

# Index