Data Mining in Python

Witek ten Hove

10/24/2022

Table of contents

Preface

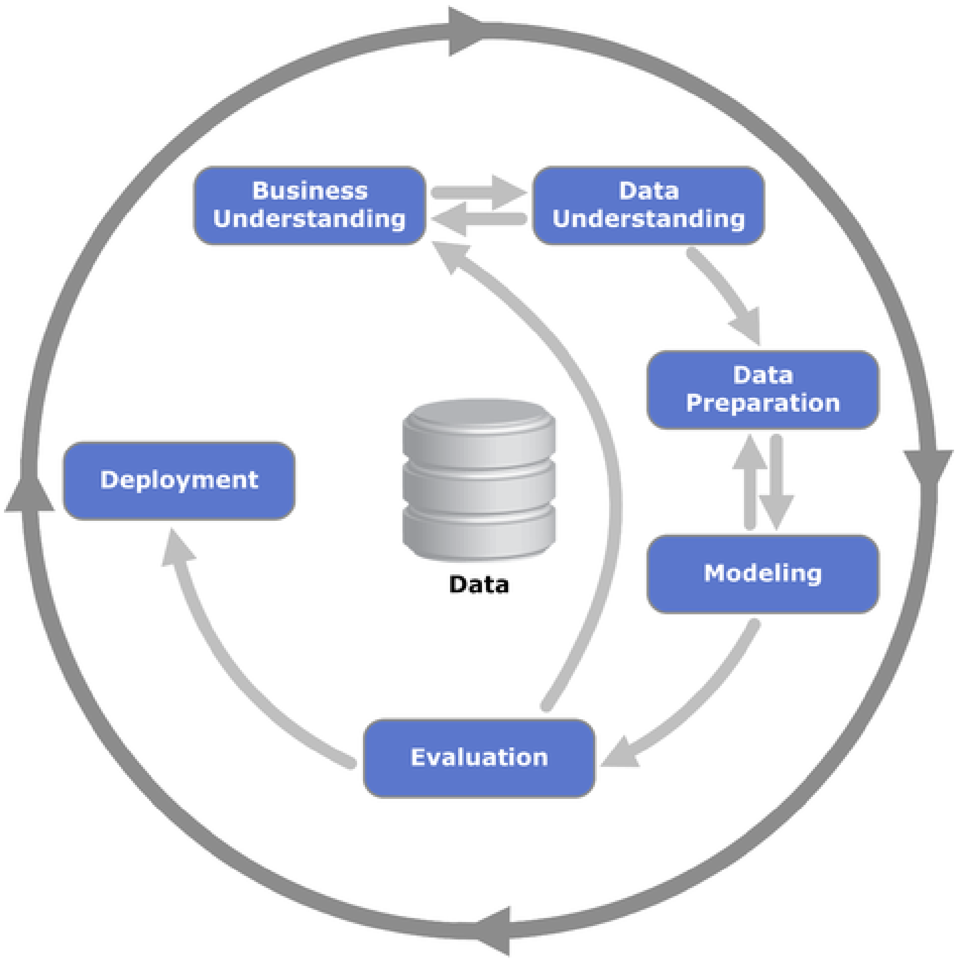


Figure 1: CRISP-DM Model taken from: https://commons.wikimedia.org/wiki/File:CRISP-DM_Process_Diagram.png

Prerequisites

Before starting this module make sure you have:

- access to the book *Provost, F., & Fawcett, T. (2013). Data Science for Business: What you need to know about data mining and data-analytic thinking. O'Reilly Media, Inc.*
- installed Anaconda
- a Github account

Purpose of this course

The general learning outcome of this course is:

> The student is able to perform a well-defined task independently in a relatively clearly arranged situation, or is able to perform in a complex and unpredictable situation under supervision.

The course will provide you with a few essential data mining skills. The focus will lie on non-linear modeling techniques - k-Nearest Neighbors (kNN) and Naive Bayes classification.

After a successful completion of the course, a student:

- is able to prepare data for a given non-linear model
- train en test a non-linear model
- evaluate the quality of a trained model

Structure of the course

Table 1: Course overview

Week nr.	Module name	Readings
2	Onboarding and Introduction to the Course	Provost / Fawcett Ch.3
3-4	Lazy Learning with kNN	Provost / Fawcett Ch.6 + 7
5-6	Probabilistic Kearning with Naive Bayes classification	Provost / Fawcett Ch.9
7	Project Application	

Through the whole of the program you'll be cooperating within a team where you will combine and compare the results of the different case studies. At the end of the course you will present with your team what you have learned from analyzing and comparing the different case studies.

About the author

Witek ten Hove is a senior instructor and researcher at HAN University of Applied Sciences. His main areas of expertise are Data en Web Technologies.

Through his extensive business experience in Finance and International Trade and thorough knowledge of modern data technologies, he is able to make connections between technology and business. As an open source evangelist he firmly believe in the power of knowledge sharing. His mission is to inspire business professionals and help them exploit the full potential of smart technologies.

He is the owner of Ten Hove Business Data Solutions, a consultancy and training company helping organizations to achieve maximum business value through data driven solutions.

1 Setting up your data science environment

1.1 Working with Quarto

1.2 Working with Git and Github

1.3 Using Python virtual environments

2 Lazy learning with k-Nearest Neighbors

2.1 Business Case: Diagnosing Breast Cancer

Breast cancer is the top cancer in women both in the developed and the developing world. In the Netherlands it is the most pervasive form of cancer ("WHO | Cancer Country Profiles 2020" n.d.). In order to improve breast cancer outcome and survival early detection remains the most important instrument for breast cancer control. If machine learning could automate the identification of cancer, it would improve efficiency of the detection process and might also increase its effectiveness by providing greater detection accuracy.

2.2 Data Understanding

The data we will be using comes from the University of Wisconsin and is available online as an open source dataset ("UCI Machine Learning Repository: Breast Cancer Wisconsin (Diagnostic) Data Set" n.d.). It includes measurements from digitized images from from fine-needle aspirates of breast mass. The values represent cell nuclei features.

For convenience the data in csv format is stored on Github. We can access it directly using a function dedicated to reading csv from the `readr` package.

```r
url <- "https://raw.githubusercontent.com/businessdatasolutions/courses/main/data%20mining
rawDF <- read_csv(url)
```

Using the `str()` function we can have some basic information about the dataset.

```r
str(rawDF)
```

```
spec_tbl_df [569 x 32] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ id                : num [1:569] 87139402 8910251 905520 868871 9012568 ...
 $ diagnosis         : chr [1:569] "B" "B" "B" "B" ...
 $ radius_mean       : num [1:569] 12.3 10.6 11 11.3 15.2 ...
 $ texture_mean      : num [1:569] 12.4 18.9 16.8 13.4 13.2 ...
 $ perimeter_mean    : num [1:569] 78.8 69.3 70.9 73 97.7 ...
 $ area_mean         : num [1:569] 464 346 373 385 712 ...
 $ smoothness_mean  : num [1:569] 0.1028 0.0969 0.1077 0.1164 0.0796 ...
 $ compactness_mean : num [1:569] 0.0698 0.1147 0.078 0.1136 0.0693 ...
 $ concavity_mean   : num [1:569] 0.0399 0.0639 0.0305 0.0464 0.0339 ...
 $ points_mean      : num [1:569] 0.037 0.0264 0.0248 0.048 0.0266 ...
 $ symmetry_mean    : num [1:569] 0.196 0.192 0.171 0.177 0.172 ...
 $ dimension_mean   : num [1:569] 0.0595 0.0649 0.0634 0.0607 0.0554 ...
 $ radius_se        : num [1:569] 0.236 0.451 0.197 0.338 0.178 ...
 $ texture_se       : num [1:569] 0.666 1.197 1.387 1.343 0.412 ...
 $ perimeter_se     : num [1:569] 1.67 3.43 1.34 1.85 1.34 ...
 $ area_se          : num [1:569] 17.4 27.1 13.5 26.3 17.7 ...
 $ smoothness_se    : num [1:569] 0.00805 0.00747 0.00516 0.01127 0.00501 ...
 $ compactness_se   : num [1:569] 0.0118 0.03581 0.00936 0.03498 0.01485 ...
 $ concavity_se     : num [1:569] 0.0168 0.0335 0.0106 0.0219 0.0155 ...
 $ points_se        : num [1:569] 0.01241 0.01365 0.00748 0.01965 0.00915 ...
 $ symmetry_se      : num [1:569] 0.0192 0.035 0.0172 0.0158 0.0165 ...
 $ dimension_se     : num [1:569] 0.00225 0.00332 0.0022 0.00344 0.00177 ...
 $ radius_worst     : num [1:569] 13.5 11.9 12.4 11.9 16.2 ...
 $ texture_worst    : num [1:569] 15.6 22.9 26.4 15.8 15.7 ...
 $ perimeter_worst  : num [1:569] 87 78.3 79.9 76.5 104.5 ...
 $ area_worst       : num [1:569] 549 425 471 434 819 ...
 $ smoothness_worst : num [1:569] 0.139 0.121 0.137 0.137 0.113 ...
 $ compactness_worst: num [1:569] 0.127 0.252 0.148 0.182 0.174 ...
 $ concavity_worst  : num [1:569] 0.1242 0.1916 0.1067 0.0867 0.1362 ...
 $ points_worst     : num [1:569] 0.0939 0.0793 0.0743 0.0861 0.0818 ...
 $ symmetry_worst   : num [1:569] 0.283 0.294 0.3 0.21 0.249 ...
 $ dimension_worst  : num [1:569] 0.0677 0.0759 0.0788 0.0678 0.0677 ...
 - attr(*, "spec")=
  .. cols(
  ..   id = col_double(),
  ..   diagnosis = col_character(),
  ..   radius_mean = col_double(),
  ..   texture_mean = col_double(),
  ..   perimeter_mean = col_double(),
  ..   area_mean = col_double(),
  ..   smoothness_mean = col_double(),
  ..   compactness_mean = col_double(),
  ..   concavity_mean = col_double(),
  ..   points_mean = col_double(),
  ..   symmetry_mean = col_double(),
  ..   dimension_mean = col_double(),
  ..   radius_se = col_double(),
  ..   texture_se = col_double(),
  ..   perimeter_se = col_double(),
  ..   area_se = col_double(),
  ..   smoothness_se = col_double(),
  ..   compactness_se = col_double(),
  ..   concavity_se = col_double(),
  ..   points_se = col_double(),
  ..   symmetry_se = col_double(),
  ..   dimension_se = col_double(),
  ..   radius_worst = col_double(),
  ..   texture_worst = col_double(),
  ..   perimeter_worst = col_double(),
  ..   area_worst = col_double(),
  ..   smoothness_worst = col_double(),
  ..   compactness_worst = col_double(),
  ..   concavity_worst = col_double(),
  ..   points_worst = col_double(),
  ..   symmetry_worst = col_double(),
  ..   dimension_worst = col_double()
  .. )
 - attr(*, "problems")=<externalptr>
```

The dataset has 32 variables (columns) and 569 observations (rows).

2.3 Preparation

The first variable, `id`, contains unique patient IDs. The IDs do not contain any relevant information for making predictions, so we will delete it from the dataset.

```
cleanDF <- rawDF[-1]
head(cleanDF)
```

```
# A tibble: 6 x 31
  diagnosis radius_mean textur~1 perim~2 area_~3 smoot~4 compa~5 conca~6 point~7
  <chr>           <dbl>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 B                12.3     12.4    78.8    464.  0.103   0.0698  0.0399  0.037
2 B                10.6     19.0    69.3    346.  0.0969  0.115   0.0639  0.0264
3 B                11.0     16.8    70.9    373.  0.108   0.0780  0.0305  0.0248
4 B                11.3     13.4    73      385.  0.116   0.114   0.0464  0.0480
5 B                15.2     13.2    97.6    712.  0.0796  0.0693  0.0339  0.0266
6 B                11.6     19.0    74.2    410.  0.0855  0.0772  0.0548  0.0143
# ... with 22 more variables: symmetry_mean <dbl>, dimension_mean <dbl>,
#   radius_se <dbl>, texture_se <dbl>, perimeter_se <dbl>, area_se <dbl>,
#   smoothness_se <dbl>, compactness_se <dbl>, concavity_se <dbl>,
#   points_se <dbl>, symmetry_se <dbl>, dimension_se <dbl>, radius_worst <dbl>,
#   texture_worst <dbl>, perimeter_worst <dbl>, area_worst <dbl>,
#   smoothness_worst <dbl>, compactness_worst <dbl>, concavity_worst <dbl>,
#   points_worst <dbl>, symmetry_worst <dbl>, dimension_worst <dbl>, and ...
# i Use `colnames()` to see all variable names
```

The variable named **diagnosis** contains the outcomes we would like to predict - 'B' for 'Benign' and 'M' for 'Malignant'. The variable we would like to predict is called the 'label'. We can look at the counts and proportions for both outcomes, using the `tables()` and `prop.tables()`functions.

```
cntDiag <- table(cleanDF$diagnosis)
propDiag <- round(prop.table(cntDiag) * 100 , digits = 1)

cntDiag
```

```
  B   M 
357 212 
```

```
propDiag
```

```
   B    M 
62.7 37.3 
```

The variable is now coded as a type `character`. Many models require that the label is of type `factor`. This is easily solved using the `factor()` function.

```
cleanDF$diagnosis <- factor(cleanDF$diagnosis, levels = c("B", "M"), labels = c("Benign",
head(cleanDF, 10)
```

```
# A tibble: 10 x 31
   diagnosis radius_mean textu~1 perim~2 area_~3 smoot~4 compa~5 conca~6 point~7
   <fct>           <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 Benign           12.3    12.4    78.8    464.  0.103   0.0698  0.0399  0.037 
 2 Benign           10.6    19.0    69.3    346.  0.0969  0.115   0.0639  0.0264
 3 Benign           11.0    16.8    70.9    373.  0.108   0.0780  0.0305  0.0248
 4 Benign           11.3    13.4    73      385.  0.116   0.114   0.0464  0.0480
 5 Benign           15.2    13.2    97.6    712.  0.0796  0.0693  0.0339  0.0266
 6 Benign           11.6    19.0    74.2    410.  0.0855  0.0772  0.0548  0.0143
 7 Benign           11.5    23.9    74.5    404.  0.0926  0.102   0.111   0.0411
 8 Malignant        13.8    23.8    91.6    598.  0.132   0.177   0.156   0.0918
 9 Benign           10.5    19.3    67.4    336.  0.0999  0.0858  0.0300  0.0120
10 Benign           11.1    15.0    71.5    374.  0.103   0.0910  0.0540  0.0334
# ... with 22 more variables: symmetry_mean <dbl>, dimension_mean <dbl>,
#   radius_se <dbl>, texture_se <dbl>, perimeter_se <dbl>, area_se <dbl>,
#   smoothness_se <dbl>, compactness_se <dbl>, concavity_se <dbl>,
#   points_se <dbl>, symmetry_se <dbl>, dimension_se <dbl>, radius_worst <dbl>,
#   texture_worst <dbl>, perimeter_worst <dbl>, area_worst <dbl>,
#   smoothness_worst <dbl>, compactness_worst <dbl>, concavity_worst <dbl>,
#   points_worst <dbl>, symmetry_worst <dbl>, dimension_worst <dbl>, and ...
# i Use `colnames()` to see all variable names
```

The features consist of three different measurements of ten characteristics. We will take three characteristics and have a closer look.

```r
summary(cleanDF[c("radius_mean", "area_mean", "smoothness_mean")])
```

```
  radius_mean       area_mean      smoothness_mean  
 Min.   : 6.981   Min.   : 143.5   Min.   :0.05263  
 1st Qu.:11.700   1st Qu.: 420.3   1st Qu.:0.08637  
 Median :13.370   Median : 551.1   Median :0.09587  
 Mean   :14.127   Mean   : 654.9   Mean   :0.09636  
 3rd Qu.:15.780   3rd Qu.: 782.7   3rd Qu.:0.10530  
 Max.   :28.110   Max.   :2501.0   Max.   :0.16340  
```

You'll notice that the three variables have very different ranges and as a consequence `area_mean` will have a larger impact on the distance calculation than the `smootness_mean`. This could potentially cause problems for modeling. To solve this we'll apply normalization to rescale all features to a standard range of values.

We will write our own normalization function.

```r
normalize <- function(x) { # Function takes in a vector
  return ((x - min(x)) / (max(x) - min(x))) # distance of item value - minimum vector valu
}

testSet1 <- c(1:5)
```

```r
testSet2 <- c(1:5) * 10

cat("testSet1:", testSet1, "\n")
```

```
testSet1: 1 2 3 4 5
```

```r
cat("testSet2:", testSet2, "\n")
```

```
testSet2: 10 20 30 40 50
```

```r
cat("Normalized testSet1:", normalize(testSet1), "\n")
```

```
Normalized testSet1: 0 0.25 0.5 0.75 1
```

```r
cat("Normalized testSet2:", normalize(testSet2))
```

```
Normalized testSet2: 0 0.25 0.5 0.75 1
```

We'll apply the **normalize()** function to each feature in the dataset (so, not on the label) using the **sapply()** function.

```r
nCols <- dim(cleanDF)[2]
cleanDF_n <- sapply(2:nCols,
                    function(x) {
  normalize(cleanDF[,x])
}) %>% as.data.frame()

summary(cleanDF_n[c("radius_mean", "area_mean", "smoothness_mean")])
```

```
  radius_mean       area_mean      smoothness_mean 
 Min.   :0.0000   Min.   :0.0000   Min.   :0.0000  
 1st Qu.:0.2233   1st Qu.:0.1174   1st Qu.:0.3046  
 Median :0.3024   Median :0.1729   Median :0.3904  
 Mean   :0.3382   Mean   :0.2169   Mean   :0.3948  
 3rd Qu.:0.4164   3rd Qu.:0.2711   3rd Qu.:0.4755  
 Max.   :1.0000   Max.   :1.0000   Max.   :1.0000  
```

When we take the variables we selected earlier and look at the summary parameters again, we'll see that the normalization was successful.

We can now split our data into training and test sets.

```
trainDF_feat <- cleanDF_n[1:469,  ]
testDF_feat <- cleanDF_n[470:569,  ]
```

When creating the training and test sets, we've excluded the labels. We'll create separate training and tests sets for them too.

```
trainDF_labels <- cleanDF[1:469,  1]
testDF_labels <- cleanDF[470:569,  1]
```

Now we can train and evaluate our kNN model.

2.4 Modeling and Evaluation

To train the knn model we only need one single function from the `class` package. It takes the set with training features and the set with training label. The trained model is applied to the set with test features and the function gives back a set of predictions.

```
cleanDF_test_pred <- knn(train = as.matrix(trainDF_feat), test = as.matrix(testDF_feat), c
head(cleanDF_test_pred)
```

```
[1] Benign    Benign    Benign    Benign    Malignant Benign
Levels: Benign Malignant
```

Now that we have a set of predicted labels we can compare these with the actual labels. A diffusion table shows how well the model performed.

Here is our own table:

```
confusionMatrix(cleanDF_test_pred, testDF_labels[[1]], positive = NULL, dnn = c("Predictio
```

```
Warning in confusionMatrix.default(cleanDF_test_pred, testDF_labels[[1]], :
Levels are not in the same order for reference and data. Refactoring data to
match.
```

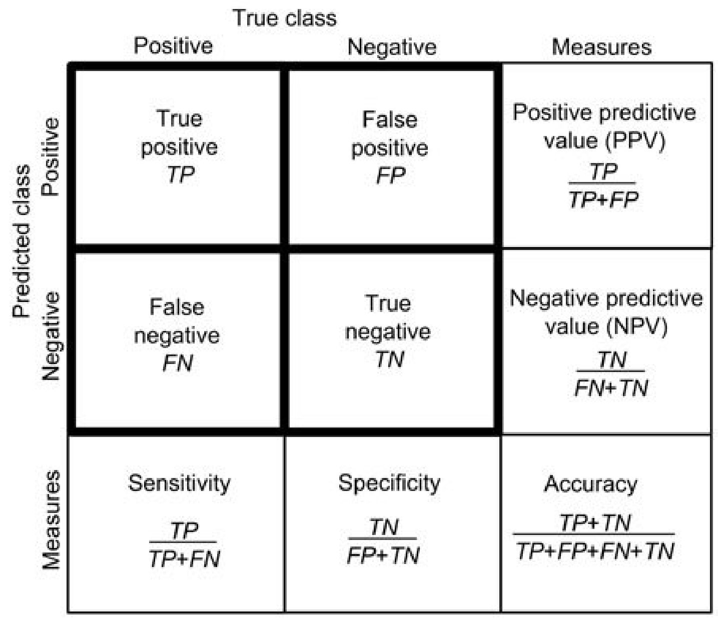


Figure 2.1: Standard diffusion table. Taken from: https://emj.bmj.com/content/emermed/36/7/431/F1.large.jp

```
Confusion Matrix and Statistics

           True
Prediction  Malignant Benign
  Malignant        37      0
  Benign            2     61

               Accuracy : 0.98
                 95% CI : (0.9296, 0.9976)
    No Information Rate : 0.61
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9576

 Mcnemar's Test P-Value : 0.4795

            Sensitivity : 0.9487
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9683
             Prevalence : 0.3900
         Detection Rate : 0.3700
   Detection Prevalence : 0.3700
      Balanced Accuracy : 0.9744

       'Positive' Class : Malignant
```

Questions:

1. *How would you assess the overall performance of the model?*
2. *What would you consider as more costly: high false negatives or high false positives levels? Why?*

3 Probabilistic Learning with Naive Bayes Classification

3.1 Business Case: Filtering Spam

In 2020 spam accounted for more than 50% of total e-mail traffic ("Spam Statistics: Spam e-Mail Traffic Share 2019" n.d.). This illustrates the value of a good spam filter. Naive Bayes spam filtering is a standard technique for handling spam. It is one of the oldest ways of doing spam filtering, with roots in the 1990s.

3.2 Data Understanding

The data you'll be using comes from the SMS Spam Collection ("UCI Machine Learning Repository: SMS Spam Collection Data Set" n.d.). It contains a set of SMS messages that are labeled 'ham' or 'spam'. and is a standard data set for testing spam filtering methods.

```
url = "datasets/smsspam.csv"
rawDF = pd.read_csv(url)
rawDF.head()
```

```
   type                                               text
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
```

The variable `type` is of class `object` which in Python refers to text. As this variable indicates whether the message belongs to the category ham or spam it is better to convert it to a `category` variable.

```
catType = CategoricalDtype(categories=["ham", "spam"], ordered=False)
rawDF.type = rawDF.type.astype(catType)
rawDF.type
```

```
0        ham
1        ham
2       spam
3        ham
4        ham
        ...
5567    spam
5568     ham
5569     ham
5570     ham
5571     ham
Name: type, Length: 5572, dtype: category
Categories (2, object): ['ham', 'spam']
```

To see how the types of sms messages are distributed you can compare the counts for each category.

```
rawDF.type.value_counts()
```

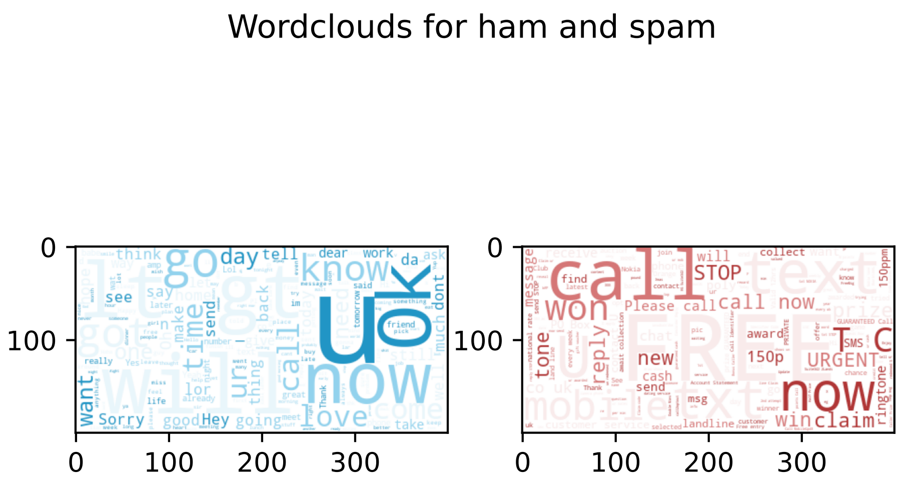
```
ham     4825
spam     747
Name: type, dtype: int64
```

You can also visually inspect the data by creating wordclouds for each sms type.

```
# Generate a word cloud image]
hamText = ' '.join([Text for Text in rawDF[rawDF['type']=='ham']['text']])
spamText = ' '.join([Text for Text in rawDF[rawDF['type']=='spam']['text']])
colorListHam=['#e9f6fb','#92d2ed','#2195c5']
colorListSpam=['#f9ebeb','#d57676','#b03636']
colormapHam=colors.ListedColormap(colorListHam)
colormapSpam=colors.ListedColormap(colorListSpam)
wordcloudHam = WordCloud(background_color='white', colormap=colormapHam).generate(hamText)
wordcloudSpam = WordCloud(background_color='white', colormap=colormapSpam).generate(spamTe

# Display the generated image:
# the matplotlib way:
fig, (wc1, wc2) = plt.subplots(1, 2)
```

```
fig.suptitle('Wordclouds for ham and spam')
wc1.imshow(wordcloudHam)
wc2.imshow(wordcloudSpam)
plt.show()
```



Question:

- *What differences do you notice?*

3.3 Preparation

After you've glimpsed over the data and have a certain understanding of its structure and content, you are now ready to prepare the data for further processing. For the naive bayes model you'll need to have a dataframe with wordcounts.

```
vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(rawDF.text)
wordsDF = pd.DataFrame(vectors.toarray(), columns=vectorizer.get_feature_names_out())
wordsDF.head()
```

```
    00  000  000pes  008704050406  0089  ...  zoe  zogtorius  zoom  zouk  zyada
0  0.0  0.0     0.0           0.0   0.0  ...  0.0        0.0   0.0   0.0    0.0
1  0.0  0.0     0.0           0.0   0.0  ...  0.0        0.0   0.0   0.0    0.0
2  0.0  0.0     0.0           0.0   0.0  ...  0.0        0.0   0.0   0.0    0.0
3  0.0  0.0     0.0           0.0   0.0  ...  0.0        0.0   0.0   0.0    0.0
4  0.0  0.0     0.0           0.0   0.0  ...  0.0        0.0   0.0   0.0    0.0

[5 rows x 8625 columns]
```

To save on computation time you can set a limit on the number of features (columns) in the wordsDF dataframe.

```
vectorizer = TfidfVectorizer(max_features=1000)
vectors = vectorizer.fit_transform(rawDF.text)
wordsDF = pd.DataFrame(vectors.toarray(), columns=vectorizer.get_feature_names_out())
wordsDF.head()
```

```
   000   03   04  0800  08000839402  ...  your  yours  yourself   yr  yup
0  0.0  0.0  0.0   0.0          0.0  ...   0.0    0.0       0.0  0.0  0.0
1  0.0  0.0  0.0   0.0          0.0  ...   0.0    0.0       0.0  0.0  0.0
2  0.0  0.0  0.0   0.0          0.0  ...   0.0    0.0       0.0  0.0  0.0
3  0.0  0.0  0.0   0.0          0.0  ...   0.0    0.0       0.0  0.0  0.0
4  0.0  0.0  0.0   0.0          0.0  ...   0.0    0.0       0.0  0.0  0.0

[5 rows x 1000 columns]
```

The counts are normalized in such a way that the words that are most likely to have predictive power get heavier weights. For instance stopword like *"a"* and *"for"* most probably will equally likely feature in spam as in ham messages. Therefore these words will be assigned lower normalized counts.

Before we start modeling we need to split all datasets into *train* and *test* sets. The function *train_test_split()* can be used to create balanced splits of the data. In this case we'll create a 75/25% split.

```
xTrain, xTest, yTrain, yTest = train_test_split(wordsDF, rawDF.type)
```

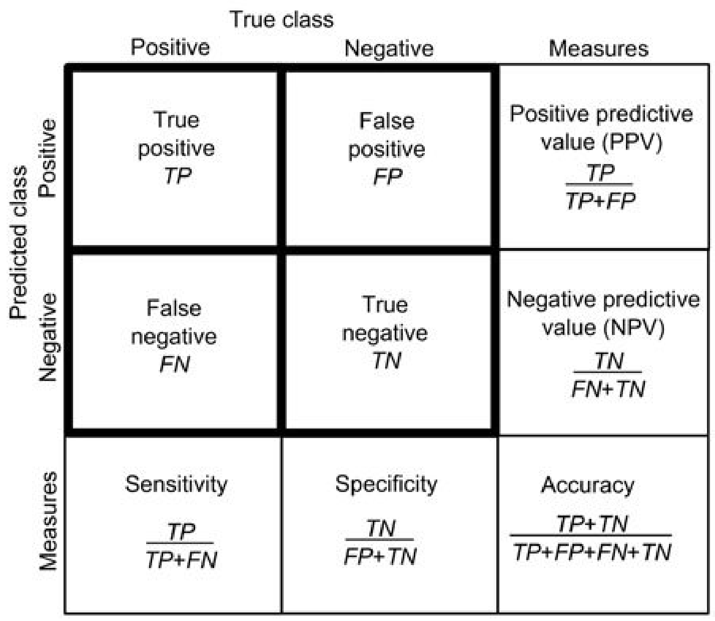
3.4 Modeling and Evaluation

We have now everything in place to start training our model and evaluate against our test dataset. The `naiveBayes()` function is part of the `e1071` package. It takes in the features and labels of our training dataset and returns a trained model.

```
bayes = MultinomialNB()
bayes.fit(xTrain, yTrain)
```

```
MultinomialNB()
```

The model can be applied to the test features using the `predict()` funtion which generates a vector of predictions. Using a confusion matrix we can analyze the performance of our model.



		True class: Positive	True class: Negative	Measures
Predicted class	Positive	True positive *TP*	False positive *FP*	Positive predictive value (PPV) $\frac{TP}{TP+FP}$
Predicted class	Negative	False negative *FN*	True negative *TN*	Negative predictive value (NPV) $\frac{TN}{FN+TN}$
Measures		Sensitivity $\frac{TP}{TP+FN}$	Specificity $\frac{TN}{FP+TN}$	Accuracy $\frac{TP+TN}{TP+FP+FN+TN}$

Figure 3.1: Standard diffusion table. Taken from: https://emj.bmj.com/content/emermed/36/7/431/F1.large.jp

```
yPred = bayes.predict(xTest)
yTrue = yTest
```

```
accuracyScore = accuracy_score(yTrue, yPred)
print(f'Accuracy: {accuracyScore}')
```

```
Accuracy: 0.9791816223977028
```

```
matrix = confusion_matrix(yTrue, yPred)
labelNames = pd.Series(['ham', 'spam'])
pd.DataFrame(matrix,
     columns='Predicted ' + labelNames,
     index='Is ' + labelNames)
```

```
         Predicted ham  Predicted spam
Is ham            1178               3
Is spam             26             186
```

Questions:

1. *What do you think is the role of the* `alpha` *parameter in the MultinomialNB() function?*
2. *How would you assess the overall performance of the model?*
3. *What would you consider as more costly: high false negatives or high false positives levels? Why?*

References

“Spam Statistics: Spam e-Mail Traffic Share 2019.” n.d. *Statista*. Accessed January 10, 2021. https://www.statista.com/statistics/420391/spam-email-traffic-share/.

“UCI Machine Learning Repository: Breast Cancer Wisconsin (Diagnostic) Data Set.” n.d. Accessed January 7, 2021. https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic).

“UCI Machine Learning Repository: SMS Spam Collection Data Set.” n.d. Accessed January 9, 2021. https://archive.ics.uci.edu/ml/datasets/sms+spam+collection.

“WHO | Cancer Country Profiles 2020.” n.d. *WHO*. Accessed January 7, 2021. http://www.who.int/cancer/country-profiles/en/.