Data Mining in Python

Witek ten Hove

10/24/22

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 Learning 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 for reading csv from the `pandas` library

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

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

```
rawDF.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 32 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   id                 569 non-null    int64
 1   diagnosis          569 non-null    object
```

```
 2   radius_mean         569 non-null    float64
 3   texture_mean        569 non-null    float64
 4   perimeter_mean      569 non-null    float64
 5   area_mean           569 non-null    float64
 6   smoothness_mean     569 non-null    float64
 7   compactness_mean    569 non-null    float64
 8   concavity_mean      569 non-null    float64
 9   points_mean         569 non-null    float64
 10  symmetry_mean       569 non-null    float64
 11  dimension_mean      569 non-null    float64
 12  radius_se           569 non-null    float64
 13  texture_se          569 non-null    float64
 14  perimeter_se        569 non-null    float64
 15  area_se             569 non-null    float64
 16  smoothness_se       569 non-null    float64
 17  compactness_se      569 non-null    float64
 18  concavity_se        569 non-null    float64
 19  points_se           569 non-null    float64
 20  symmetry_se         569 non-null    float64
 21  dimension_se        569 non-null    float64
 22  radius_worst        569 non-null    float64
 23  texture_worst       569 non-null    float64
 24  perimeter_worst     569 non-null    float64
 25  area_worst          569 non-null    float64
 26  smoothness_worst    569 non-null    float64
 27  compactness_worst   569 non-null    float64
 28  concavity_worst     569 non-null    float64
 29  points_worst        569 non-null    float64
 30  symmetry_worst      569 non-null    float64
 31  dimension_worst     569 non-null    float64
dtypes: float64(30), int64(1), object(1)
memory usage: 142.4+ KB
```

The dataset has `python rawDF.shape()[0]` variables (columns).

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.drop(['id'], axis=1)
cleanDF.head()
```

```
  diagnosis  radius_mean  ...  symmetry_worst  dimension_worst
0         B        12.32  ...          0.2827          0.06771
1         B        10.60  ...          0.2940          0.07587
2         B        11.04  ...          0.2998          0.07881
3         B        11.28  ...          0.2102          0.06784
4         B        15.19  ...          0.2487          0.06766

[5 rows x 31 columns]
```

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 both outcomes, using the `value_counts()` function. When we set the normalize setting to `True` we get the the proportions.

```
cntDiag = cleanDF['diagnosis'].value_counts()
propDiag = cleanDF['diagnosis'].value_counts(normalize=True)
cntDiag
```

```
B    357
M    212
Name: diagnosis, dtype: int64
```

```
propDiag
```

```
B    0.627417
M    0.372583
Name: diagnosis, dtype: float64
```

Looking again at the results from the `info()` function we notice that The variable `diagnosis` is coded as text (`object`). Many models require that the label is of type `category`. The '`pandas` library contains a function that can transform a `object` type to `category`.

```
catType = CategoricalDtype(categories=["B", "M"], ordered=False)
cleanDF['diagnosis'] = cleanDF['diagnosis'].astype(catType)
cleanDF['diagnosis']
```

```
0      B
1      B
2      B
3      B
4      B
      ..
564    B
565    B
566    M
567    B
568    M
Name: diagnosis, Length: 569, dtype: category
Categories (2, object): ['B', 'M']
```

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

```
cleanDF[['radius_mean', 'area_mean', 'smoothness_mean']].describe()
```

```
       radius_mean    area_mean  smoothness_mean
count   569.000000   569.000000       569.000000
mean     14.127292   654.889104         0.096360
std       3.524049   351.914129         0.014064
min       6.981000   143.500000         0.052630
25%      11.700000   420.300000         0.086370
50%      13.370000   551.100000         0.095870
75%      15.780000   782.700000         0.105300
max      28.110000  2501.000000         0.163400
```

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.

```
def normalize(x):
  return((x - min(x)) / (max(x) - min(x))) # distance of item value - minimum vector value

testSet1 = np.arange(1,6)
testSet2 = np.arange(1,6) * 10
```

```
  print(f'testSet1: {testSet1}\n')
```

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

```
  print(f'testSet2: {testSet1}\n')
```

```
testSet2: [1 2 3 4 5]
```

```
  print(f'Normalized testSet1: {normalize(testSet1)}\n')
```

```
Normalized testSet1: [0.   0.25 0.5  0.75 1.  ]
```

```
  print(f'Normalized testSet2: {normalize(testSet2)}\n')
```

```
Normalized testSet2: [0.   0.25 0.5  0.75 1.  ]
```

and apply it to all the numerical variables in the dataframe.

```
  excluded = ['diagnosis'] # list of columns to exclude
  X = cleanDF.loc[:, ~cleanDF.columns.isin(excluded)]
  X = X.apply(normalize, axis=0)
  X[['radius_mean', 'area_mean', 'smoothness_mean']].describe()
```

```
       radius_mean   area_mean  smoothness_mean
count   569.000000  569.000000       569.000000
mean      0.338222    0.216920         0.394785
std       0.166787    0.149274         0.126967
min       0.000000    0.000000         0.000000
25%       0.223342    0.117413         0.304595
50%       0.302381    0.172895         0.390358
75%       0.416442    0.271135         0.475490
max       1.000000    1.000000         1.000000
```

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.

```
y = cleanDF['diagnosis']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=123,
```

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 `sklearn` library. It takes the set with training features and the set with training label and fits a model to the training data. The trained model is applied to the set with test features and the `predict()` function gives back a set of predictions.

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
# make predictions on the test set
```

```
KNeighborsClassifier()
```

```
y_pred = knn.predict(X_test)
```

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:

```
cm = confusion_matrix(y_test, y_pred, labels=knn.classes_)
cm
```

```
array([[106,   1],
       [  2,  62]])
```

```
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=knn.classes_)
disp.plot()
```

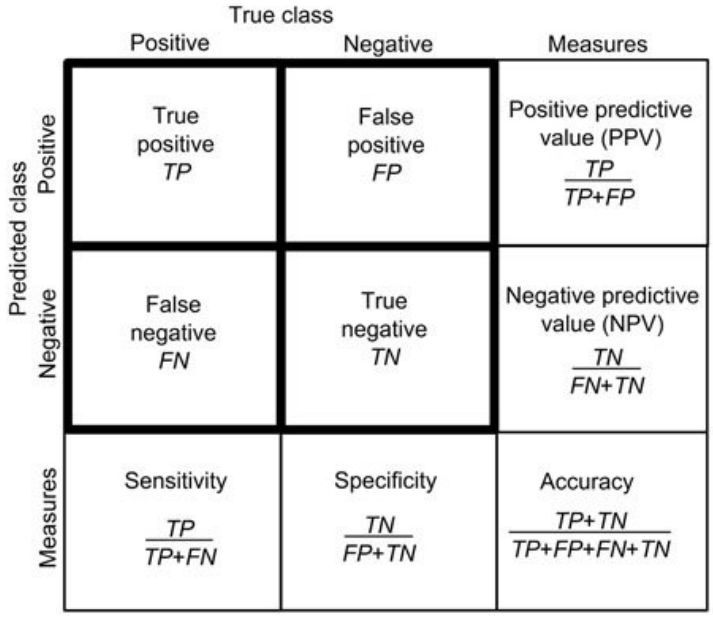
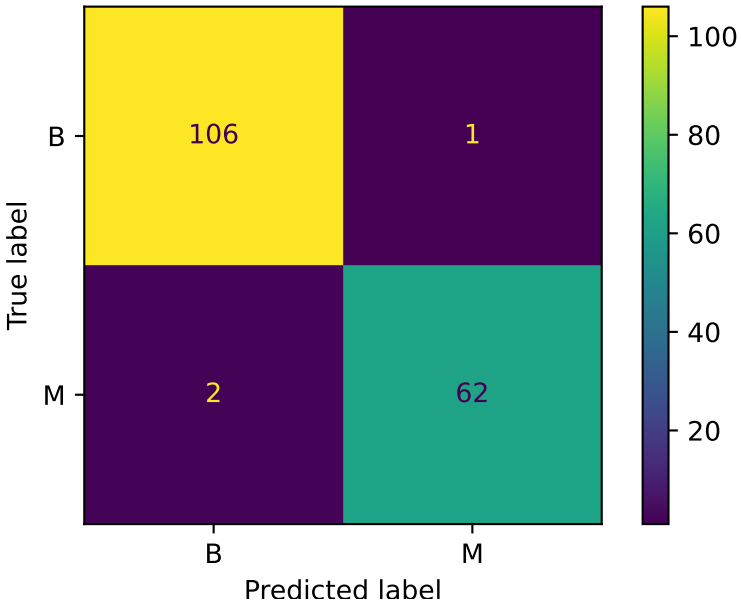


Figure 2.1: Standard diffusion table. Taken from: https://emj.bmj.com/content/emermed/36/7/431/F1.large.jp

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay object at 0x1ae7dac20>
```

```
plt.show()
```



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
```

Often you'll prefer the relative counts.

```
rawDF.type.value_counts(normalize=True)
```

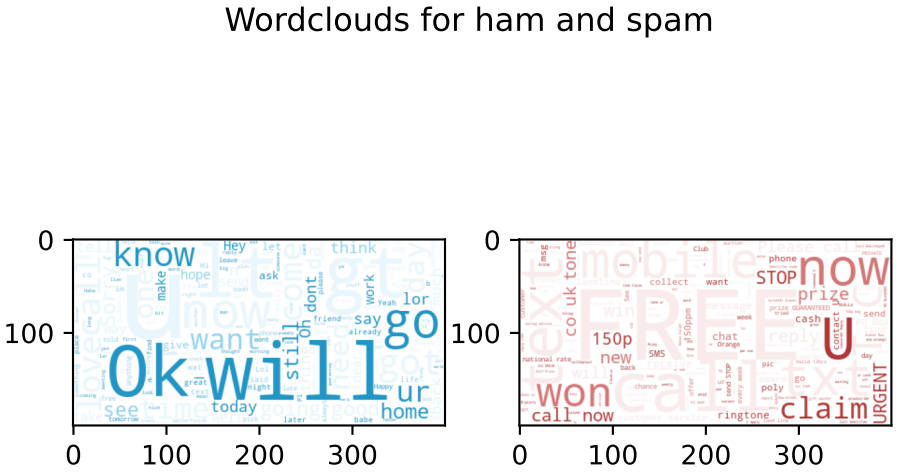
```
ham     0.865937
spam    0.134063
Name: type, dtype: float64
```

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. 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 `MultinomialNB().fit()` function is part of the `scikit learn` package. It takes in the features and labels of our training dataset and returns a trained naive bayes model.

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

```
MultinomialNB()
```

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

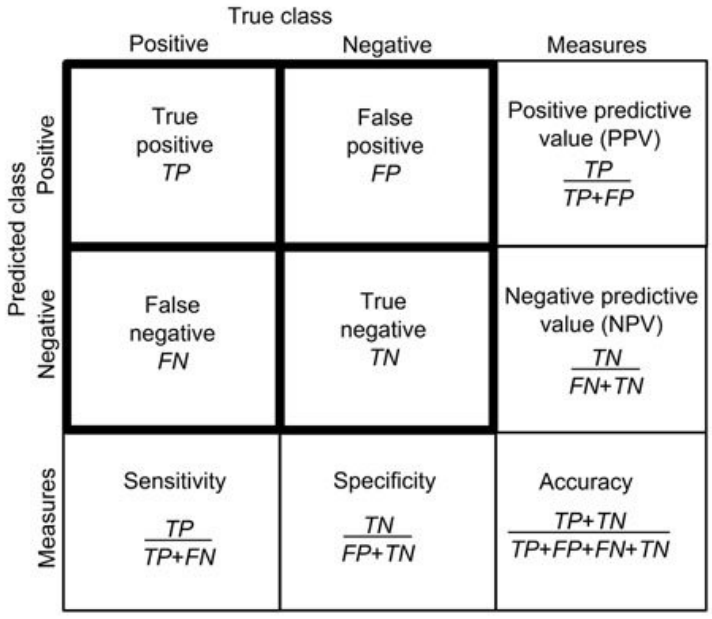


Figure 3.1: Standard diffusion table. Taken from: https://emj.bmj.com/content/emermed/36/7/431/F1.large.jp

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

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

```
Accuracy: 0.9748743718592965
```

```
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            1220               0
Is spam             35             138
```

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/.