Softwarewerkzeuge der Bioinformatik

Prof. Dr. Volkhard Helms
PD Dr. Michael Hutter, Markus Hollander,
Andreas Denger, Marie Detzler, Larissa Fey

Saarland University
Department of Computational Biology

Winter semester 2020/2021

Project 3

Deadline: February 25, 2021

Machine Learning

In this project, you will analyze a microarray dataset containing gene expression data from cancer patients, and from healthy patients for reference. First, you will learn some basic Python programming, so you can solve the exercises without any prior knowledge about coding. Next, you will learn how to plot the data with the *seaborn* package, calculate the pairwise correlations between the samples, and perform a hierarchical clustering. In the final exercise you will train and optimize a machine learning pipeline on training data, and test its performance on an independent test set.

Submit the finished notebook (the *.ipynb* file) to
`andreas.denger@bioinformatik.uni-saarland.de`. You can download a Jupyter Notebook from Kaggle by clicking on *File→Download*.
The notebook has to contain all of the code that you used to solve the exercises. Use Markdown cells (i.e. cells containing written text) to clarify which code cells belong to which exercise, and to write notes, explanations and descriptions for the steps and results in your analysis.
Also write to the Email address above if you have any questions, or if you want schedule a meeting on Microsoft Teams to go through parts of exercises you didn't understand.

Exercise 3.1: Preparation (5 Points)

In the last tutorial, you have learned how to use kaggle.com to create and execute code in Jupyter Notebooks. Find the gene expression dataset assigned to your group on the courses website, and upload it to a new notebook.

(a) Visit kaggle.com and create a new notebook. Give it a name you can recognize, such as *SWW Project 3 Group X*. If you close the notebook, you can find it again in your user account on the top right of the start page.

(b) Upload the dataset that was assigned to your group to the notebook, using the **+Add data** button on the top right. Afterwards, it should be in your *input* folder underneath the button.

Exercise 3.2: Python basics (15 Points)

The purpose of this exercise is to teach you the basic python skills needed for the rest of the project, so that you can solve the exercises even if you have no previous programming experience.

(a) In this project, you mainly need to call *functions* with *parameters* and assign their *return values* to *variables* with the = operator. You can find documentation on all of the built-in functions under https://docs.python.org/3/library/functions.html. Here, we call the *list()* function with no parameters, which returns an empty list object. An *object* can contain data, and usually comes with functions that can be applied to that data, or to something else. The list is then assigned to the variable *my_list*, and the *print()* function with the list as its parameter is called to see what the empty object looks like:

```
my_list = list()
print(my_list)
```

The *list* object comes with some functions of its own. You can find them in the Python reference manual, Chapter 5.1. We are going to append three integers to the list, and print the resulting list object:

```
my_list.append(1)
my_list.append(2)
my_list.append(2+3)
print(my_list)
```

The list and its contents are now saved to the environment of the notebook, and can be recalled later.

List elements can be accessed by using square brackets. Counting starts at 0:

```
print(my_list[0])
```

Look through the documentation for lists in the reference manual. Write code that removes the first occurrence of the integer 5, then inserts a 5 at position 0, and prints the list.

(b) In this project, you will be using several *Python Packages*, i.e. libraries containing code that is not part of the standard python distribution. In order to access the functions provided by a package, you need to import it first:

```
import numpy
```

The *numpy* package provides a data structure that acts similar to Pythons *list*, but has additional functions for scientific computing. Create a *numpy array* from your list:

```
my_array = numpy.array(my_list)
print(my_array)
```

The functions and attributes of numpy arrays can be found here:
https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray.

Look through the functions, and write code that prints both the mean value and the sum of your array.

Exercise 3.3: Data exploration with pandas & seaborn (30 Points)

The *pandas* package provides the *DataFrame* and *Series* objects. DataFrames contain tabular data, similar to an Excel sheet. Series are another type of list-like objects. Each column in a DataFrame is a Series.
Seaborn is a package that is used for data visualization, mainly to create plots from DataFrames. The reference manuals for DataFrames, Series and Seaborn can be found here, here and here, respectively.

(a) Import the *pandas* package, load your data into a DataFrame and take a look at the data by writing the name of the DataFrame on the last line of the cell, like you did in the tutorial. On the top row you can see the names of the columns, on the left hand side are the names of the rows.

(1) Assign the column called *type* to a new variable called *labels*. You can read a column from the DataFrame by using square brackets containing the name of the column in quotation marks. Documentation and examples can be found here.

(2) Create a new DataFrame called *features* that contains every column of your data, except for *samples* and *type*. *Hint: take a look at the drop() function in the DataFrame reference, and its "axis" parameter.*

(3) Print the *labels*-Series and *features*-DataFrame to see if everything worked. *features* contains the genes as its columns, and the patients as its rows. *labels* assigns a label (cancer, normal) to every row in *features*.

(4) Use the *value_counts()* function of the *labels* Series. Is your dataset balanced or imbalanced? In a *balanced* dataset, each label has the same number of samples. This will be important later.

(5) For the following exercises, we will need a transposed version of the *features* DataFrame. Use the *transpose()* function of the DataFrame *features*, and assign the result to a variable called *features_t*. Take a look at the contents of the transposed DataFrame.

(b) Next, calculate a pairwise correlation matrix for the patient samples, and visualize it with a heatmap.

(1) The *corr()* method provided by the DataFrame object calculates the pairwise correlation between every pair of columns in a DataFrame. The transposed DataFrame *features_t* that you just created has the patient-samples as its columns. Call the *corr()* method of *features_t*, and assign the result to a new variable called *features_t_corr*.

(2) Import the *seaborn* library. Call the *heatmap()* function on *features_t_corr* to create a heatmap of the correlation matrix:

```
seaborn.heatmap(
    data=features_t_corr,
    xticklabels=labels,
    yticklabels=labels
)
```

Interpret the plot. How do the labels relate to the pairwise correlations?

(c) Finally, create a *clustermap* of *features_t*, just like you created the heatmap. Use *labels* as your *xticklabels*, and set the *yticklabels* to *False*.

(1) The datasets of some groups contain more than 50.000 features. Calculating a clustermap of such a large dataset could take several hours. One way around that is to randomly draw a subset of genes from the data, and create the plot with that. You can use the *sample()* method with the *n* or the *frac* parameter to draw a number or a percentage of genes from the data, respectively. The DataFrame *features_t_sampled* contains 20% of the genes in *features_t*:

```
features_t_sampled = features_t.sample(frac=0.2,random_state=1)
```

The *random_state* parameter assures repeatability by selecting the same random samples every time, depending on what value you assign to it.

(2) You can try different clustering methods with the *method* parameter of *clustermap*. Try *method="ward"* to the function call of *clustermap*, and see if that leads to a better clustering of the columns.

(3) Write a short interpretation of the plot into a Markdown cell.

Exercise 3.4: Machine learning with scikit-learn (40 Points)

In the last part of the project, you will train and evaluate a machine learning algorithm on the data, using scikit-learn (also called *sklearn*). In the tutorial you already used a Support Vector Machine (SVM) with a *linear kernel*, which tries to draw a hyperplane (e.g. a straight line in two-dimensional space) between the classes. This time we will try a *Radial Basis Function (RFB) kernel* (also called *radial Gauss kernel*), which can divide the data in non-linear ways.

(a) The matrix containing the features is called X in sklearn, the list containing the labels is called y, and contains a different integer value for each label. Both X and y are numpy arrays. Convert your pandas DataFrame and Series to numpy arrays:

```
X = features.to_numpy()
y = numpy.where(labels == "normal", 0, 1)
```

(b) The next step will be to split the data, into a training set and an independent test set:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test=train_test_split(X,y,random_state=1)
```

The classifier will be optimized on the training set, and then evaluated using the test set.

(c) Read the Getting Started article on sklearn's website to become familiar with the basic concepts and functions. The website also contains an extensive User Guide, a reference manual (API) for the functions, and many example applications to learn from.

(d) Use the *make_pipeline* function to create a pipeline containing a StandardScaler and a SVC (sklearn.svm.SVC). Don't forget to import the functions from their packages, see the *Getting Started* guide for reference. If you found out in the last exercise that your dataset is imbalanced, you should to set the *class_weight* parameter of the SVM to *"balanced"*, that will likely improve the accuracy. Print the pipeline to see if everything worked.

(e) Now you will optimize the classifier using the training dataset. For that, we are going to use *GridSearchCV*. It takes an estimator, which can be a classifier or a pipeline, as well as a grid of parameters. It will try each combination of parameters, and choose the combination that performs the best.

   (1) Create a GridSearchCV object with your pipeline as its first parameter, and the following parameter grid as its second parameter:

   ```
   param_grid={
       'svc__gamma':[0.1, 1e-2, 1e-3, 1e-4],
       'svc__C':[1, 10, 100, 1000]
   }
   ```

   This will try four different values for the *gamma* parameter, and four values for the *C* parameter of the SVC object in the pipeline, so a total of 16 models will be tested. Set the *scoring* parameter of *GridSearchCV* to "f1". The default option is "accuracy", which can be biased for imbalanced datasets. Print the GridSearchCV object.

   (2) Call the *fit()* method of the *GridSearchCV* object on the training data (*X_train, y_train*). Print the best parameters that were found, as well as the best score.

   Hint: You can find a similar approach using RandomizedSearchCV in the Getting Started guide. Also look at the *Attributes* in the documentation of GridSearchCV.

   (3) What do the parameters you found say about your model? *Hint: look at the slides from lecture 12, and the documentation for RBF kernels on the sklearn website.*

(f) The *GridSearchCV* object now behaves like an estimator, and automatically uses the best parameters chosen during parameter optimization. Call its *score()* function on the test data (*X_test, y_test*) to receive an F1 score for your optimized pipeline.

(g) One thing that could negatively influence the performance of your pipeline is the large number of features in the dataset (more than 20.000 genes). Use *feature selection* to select the best *k* features with regards to their classification performance. The classifier is then trained only on those. Create a *SelectKBest* object that only keeps the 20 best features:

```
from sklearn.feature_selection import SelectKBest
kbest = SelectKBest(k=20)
```

Make another pipeline that contains a StandardScaler, the *kbest* object, another StandardScaler, and a SVC (don't forget the class_weights parameter). Perform the training and scoring with GridSearchCV again for this pipeline, and see if your score improves.

(h) The choice of training and test set can have a great influence on the final score. One *train_test_split* might lead to a score of 1.00, another to 0.50. To account for this potential bias, one can split the dataset into five subsets. Each subset is used for testing once, and the other four together for training. Finally, you calculate the mean and the standard deviation of the five test scores. Perform a cross validation with for your GridSearchCV object, with "f1" as your scoring function. Print the mean and average score across the five runs:

```
from sklearn.model_selection import cross_val_score
res = cross_val_score(gsearch, X,y,scoring="f1", cv=5)
print(res)
print(f"{res.mean()}+-{res.std()}")
```

To get the last 5 points for the project, you have to achieve an average F1 score of at least 0.90. If your model did not achieve the score yet, try different values for *k*, *gamma* and *C*, or read the documentation of SVC and see if you can find any other parameters to optimize with grid search.

Exercise 3.5: Project report (20 Points)

Turn the Jupyter Notebook you just created into a project report. Use Markdown cells to explain the steps you took in your analysis of the dataset, and describe the results and plots.
You can find more information about your datasets here:

- Group 1: GSE57297
- Group 2: GSE60502
- Group 3: GSE41328
- Group 4: GSE12452
- Group 5: GSE22405

Finally, click on **Run All** in the notebook, wait for the calculations to finish, and download the notebook (File→Download). Send the notebook file to the Email address above. Don't forget to include your names.

Have fun!