C4M: PhysioNet 2019 Challenge - Sepsis Prediction

Overview

PhysioNet provides access to an open dataset of physiological signals and each year they host a challenge. Challenges in previous year include detecting sleep apnea using ECG, predicting hypotensive episodes and classifying heart sounds. For a full list see: https://physionet.org/challenge/.
This years challenge is to predict sepsis before it occurs based on demographic information, vital signs and lab reports. You will be attempting a simplified version of this project where we have made some of the design decisions for you.

Step 0: Background concepts

This assignment requires you to use machine learning models and methods. This is a lot of background material here and this assignment will barely scratch the surface. I would highly recommend going through Roger Grosse's course notes for the CSC321 course. http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/.
At the very least, you should go over the following lectures and try to understand the listed concepts.
CSC411 Intro: What is machine learning, history of machine learning
CSC321 Lecture 2 Slides: Fitting polynomials (useful visualization), Generalization
CSC321 Lecture 2 Notes: Generalization (very important concept)
CSC321 Lecture 3 Notes: Section 1, intro paragraph of 2.

Step 1: Familiarize yourself with the challenge

Read the details of the challenge at https://physionet.org/challenge/2019/. Pay particular attention to what data are available in the dataset and try to build a good understanding of what the objective of the challenge is.

Step 2: Download required files

Download training_setA.zip from the PhysioNet 2019 challenge website and extract it.
Download starter code: https://c4m-uoft.github.io/projects/physionet/physionet.ipynb

You may also need to install some python packages. To install these run

```
$ conda install scikit-learn numpy pandas
$ pip install cache-em-all imbalanced-learn
```

Scikit-learn, numpy and pandas are some of the most used python packages for data science. Scikit-learn is a tool for data mining, data analysis and machine learning. Numpy provides a fast implementation of arrays, matrices and common math operations. Pandas is an extremely useful library for working with data, especially when your data can be organized into tables (e.g. a csv or excel file). Getting familiar with these packages would be a good idea for both this assignment and if you want to do any kind of data analysis in Python. Imbalanced-learn contains functions to help you work with imbalanced data.
The final package, cache-em-all, allows for saving the result of a function. In this assignment, some functions can take 5 or more minutes to run. So, cache-em-all allows you to save the result of the function so that it only takes 5 minutes the first time the function is called. Whenever the function is called again, it only takes a second or two to run. Full disclosure, cache-em-all was created by one of the TAs.

Helper function: load_single_file

First, lets create a helper function that will load a single file from the dataset. If you look at the data in the training folder, you will see files like 'p00001.psv'. This is a pipe separated file, similar to a comma separated file, except each column is separated by a '|' instead of a comma. This helper function will take as input the path to one of these files (e.g. 'training/p00001.psv'), use pandas to load the contents of the file, append some extra data and return the result.
First, you should look up the pandas `read_csv` function. It is a very simple way to read a csv file. For example,

```
1 df = pd.read_csv("training/p00001.psv")
```

Will read "myfile.csv" and load it into the variable df. Pandas uses whats called a dataframe, which you can think of as a table, to represent data. If you want to see what a dataframe looks like, try running the above code in your notebook and printing df (hint: you may have to look at the parameters of `read_csv` , specifically, one called sep).
This helper function also adds some extra information to the dataframe that we will be using later on. With a data frame, you can access a specific column using the square brackets (similar to a dict). So for example, if you read in one of the training files, you can run

```
1 df["HR"]
```

To view just the heart rate's for this file. You can also use this to assign values to a column. So for example,

```
1 df["Hgb"] = df["Hgb"] * 10
2 df["feeling"] = "Happy"
```

Line 1 will multiply all hemoglobin values by 10 (convert from g/dL to g/L). Line 2 will create a new column called "feeling" and fill all rows with the value "happy".

Now back to the helper function. Once you've read in the file, lets create a new column called "patient" that contains the name of the file this data came from. We will use the file name as a sort of patient ID to figure out later on which data belongs to which patient. Secondly, lets also create a column called "hour". If you remember from the PhysioNet data description, each row in the file represents an hour. This is fine for when our data is saved in a file, however as we're working with the data, we want to shuffle rows around a bit. Therefore, we would like the explicitly store the hour in a separate column. This can be achieved with the following code

```
df["hour"] = df.index
```

In pandas, the index is a very important concept, however, it can also get somewhat complicated. Essentially, the index is used to identify and look up rows in the data frame. When we read in the data using the `read_csv` function, pandas created an index for us where the index value of each row was it's row number in the csv file (i.e. the first row had an index value of 0, the second row had an index value of 1, etc.). So now, when we say assign the value of the index to the "hour" column, we are setting the hour to also be the row number. So by now, your `load_single_file` function should read the csv into a dataframe and create new columns for "patient" and "hour". You can now return this dataframe.

Helper function: load_data

Now lets work on another helper function called load_data. The first one read in a single file and created a dataframe, now this helper function will read in all data files and put them into a single data frame.
In pandas, adding rows to a dataframe is possible, however it is very slow. Instead, what we will do is read each file in our dataset and append it to a list and then use pandas concat function to concatenate the list of dataframes into a single dataframe. You should use the load_single_file helper function you wrote earlier.
Then we will use the pandas `concat` function to concatenate the list of dataframes into a single dataframe (Note: only do this when you know that each dataframe has the same columns. With this physionet data, we do know this to be the case).
To help you with this, we have provided a function called "get_data_files" that will give you a list of all the files in the dataset. Once you have this list, you can iterate over it and use your `load_single_file` function to load each file. Append the result of this to a list and after the for loop, use the `concat` .
Once you have the concatenated dataframe, use the following code to clean up the data. This calls a function we provided that normalizes data, removes nans and cleans up the index a bit.

```
df = clean_data(df)
```

At the end of the `load_data` function, return the concatenated and cleaned dataframe.
Once you are sure this function is working correctly, you can un-comment the `@Cachable("data.csv")` line above the function. The next time you run the function, it's result will be saved. Subsequent calls to the function will load the saved data. If you realize there was a mistake in the function implementation and you have to fix it, you will also need to delete the "cache"

folder. Otherwise, your fixed code will not be called and instead, the old saved data will be returned.

Data exploration

Use the helper functions you implemented to load the data. Try to understand what data is available. What are the different columns in the dataframe? How many rows? How many rows with where SepsisLabel is 0, how many where it is 1?

```
print(df.colums)
print(len(df))
print(df["SepsisLabel"].value_counts())
```

Machine Learning 1: Basics

In this section, we implement the function `train_simple`.
Now that we can load our data, lets try some machine learning. Because we are working with data that has labels (i.e. we know which rows are associated with sepsis), we will be using a type of learning called supervised learning. More specifically, given an input X, we are trying to predict an output y and we have examples of X, y pairs that we can use to train our system. In contrast, with unsupervised learning we would only have X and we would be trying to learn something about X.
Within supervised learning, there are two types of tasks, classification and regression. With classification, y is discrete (e.g. sepsis or not sepsis, picture is of cat, dog or bunny). With regression, y is continuous (e.g. white blood cell count, price of a house).
Remember the concept of generalization? We want the model we train to be able to predict accurately on data it has not seen before. One way we can do this is by splitting our data into two sets. One set, which we call the train set, will be used to train the model. The other, called the test set, will only be used to evaluate the model.
Scikit-learn provides a function that splits data into train and test. The test size is the proportion of data that should be used for the test set. In this case, 80% of the data is used for training, and 20% is used for testing.

```
train_df, test_df = train_test_split(df, test_size=0.2)
```

Now, the train_df and test_df contain data that should be used for training and testing, respectively. However, they both contain both X and y values. So lets further split these into train_X, train_y, test_X, test_y. As a hint, remember that you can access specific columns of a dataframe.

```
df1 = df["col_a"]

col_list = ["col_b", "col_c"]
df2 = df[col_list]
```

will create df1 with just col1 and col2 from df. Also, for your convenience, we defined variables called feature_cols and label_col that contain the column names for X and y, respectively.

Once you have X and y for both train and test sets, lets create a classifier.

```
clf = DecisionTreeClassifier(class_weight="balanced")
```

There are many different types of classifiers available in scikit-learn. We have already imported some of them and you can experiment with which ones work well.
After creating the classifier, we need to fit/train it on our data. Lets call the fit function with our training data.
Once the model is fitted, we can evaluate how well it worked. To do this we can use the following code. What this does is it uses our model to predict what it thinks the y value should be for a given X. Then, using the models predicted y value and the actual y value, we compute evaluation accuracy, precision and recall using the provided helper function called evaluate.
First, lets see how well the model performs on the train set. Remember, this is the same data that was used to fit/train the model. We can expect to get fairly high performance on this data. Next, lets see how well the model does on unseen data (i.e. is the model generalizing). This can be done similar to the code below, but just by using the test X and y values. If the model performs significantly worse on the test set than the train set, we can say that the model is overfitting to the train set.

```
y_pred = clf.predict(X_test)
evaluate(y_train, y_pred, "Train")
```

What happens if you run this code multiple times? Are the results consistent? Inconsistent results make it hard to replicate our work, therefore we would like a way to make our code produce the same result every time it is run. One way to do this is to set whats called a random seed.
We can do this by adding the following lines right after our import statements.

```
seed = 9001
np.random.seed(seed)
```

we also need to pass this seed to our the train test split function.

```
train_df, test_df = train_test_split(df, test_size=0.2, random_state=seed)
```

In this case, I set a random seed of 9001, but you can set it to whatever you like. But for a given seed, the program should produce the same result. So now your results should be consistent.
Now, how does the model perform? Is the performance on the training set significantly higher than the test set? If so, this indicates overfitting. You can think of this as the model is memorizing the answers rather than learning the concepts. One way around this is to limit the complexity of the model. For example, we can create our model with the parameters shown below. This how deep and wide the decision tree can be, which effectively limits it's ability to memorize answers. The selection of these parameters is also a complex task and you should be careful about how you go about selecting these parameters because you can introduce another form of overfitting. You can end up basically manually optimizing these parameters so that the model performs well on the test set.

```
clf = DecisionTreeClassifier(class_weight="balanced", max_depth=20, max_leaf_nodes=20)
```

Do you remember how many rows we had where the SepsisLabel was 1 (positive examples) and how many there were with 0 (negative examples)? When this proportion is heavily skewed to one side, we call this an imbalance. With a strong imbalance, classifiers may not learn as well because there are too few examples of a particular class (e.g. 0 or 1, cat or dog or bunny). The `class_weight="balance"` is one way to deal with a class imbalance. It tells the classifier to weigh classes relative to how many examples there are in the training data. For example, if there are twice as many negative examples than positive examples, the classifier will consider getting a positive examples wrong twice as bad as a negative examples.
Another way to deal with class imbalance is undersampling. This can be achieved with the code shown below. You can read about it in more detail here. The ratio 0.5 tells the sampler to have a 1:1 ratio of positive and negative examples. A ratio of 0.1 would result in a 1:10 ratio.

```
rus = RandomUnderSampler(0.5, random_state=seed)
X_train, y_train = rus.fit_resample(X_train, y_train)
```

Machine Learning 2: Better cross validation

In the first machine learning section, we talked about splitting data into train and test sets and how that helps with generalizability. We attempted to split data into independent train and test sets. However, it turns out our sets weren't so independent. We were deciding whether each row should go into the training or testing set. However, multiple rows belong to a single patient. If we have data from patient P in the training set, isn't it kind of like cheating to have data from patient P in the test set as well? What we would really like is if the model worked on patients that it had never seen before. To do this, we can use a GroupKFold. K-Fold is form of cross-validation and the simple train/test split we did earlier can be considered a special case of k-fold cross validation. You can read more here.
With a GroupKFold, we ensure that each occurrence of the group variable occurs in only one of the train or test set. So if patient P occurs is assigned to the train set, their data will not be in the test set and vice versa.
We have implemented the skeleton for this GroupKFold cross validation in the function called `train_stratified`. You job is to implement a classifier inside this function. Do not be surprised if the performance of your classifier drops. Not cheating on a test may result in a worse grade, but that grade is more reflective of how much you know.
Experiment with different classifiers, parameters and sampling ratios to find what you think is the best classifier.

Machine Learning 3: Adding in time

So far, we have made prediction on individual rows (i.e. given data at time t, will this patient have sepsis at time $t+6$). However, we would expect that if we have multiple time points of data, we can make better predictions. For example, having 4 hours of data will give us 4 data points for HR, Hgb, RR, etc. and if we can detect trends across these variables, we may be able to make better predictions. While there are more complex models out there that have

a better notion of time baked in, one way to incorporate more data is to just flatten 4 hours of data into a single vector. So now instead of having HR, you would have HR_0, HR_1, HR_2 and HR_3. We have implemented a function that does exactly this transformation. You can call like so:

```
flat_df = flatten(df, hours=4)
flattened_feat_cols = [x for x in df.columns if x not in [label_col, "patient"]]
```

Try experimenting with this flattened data. Does using multiple hours of data improve your prediction accuracy?