C4M: Machine Learning on Sensor Data

Overview

PhysioNet is a widely-used repository of biomedical data and software. It enables researchers around the world to share and reuse resources that underpin clinical studies, promoting reproducible research and lowering barriers to data access. Each year, PhysioNet hosts a challenge in which researchers and hobbyists can compete to see who can create the most accurate models for a given problem, with previous challenges including sleep apnea detection and heart sound classification (https://physionetchallenges.org/).

For this project, you will train a basic machine learning model that is capable of solving the task from PhysioNet Challenge 2019 on sepsis prediction. Early detection and treatment of sepsis are critical for improving sepsis outcomes, where each hour of delayed treatment has been associated with roughly an 4–8% increase in mortality [1, 2]. The challenge gave people access to a dataset containing demographic information, vital signs, and lab reports. Your model will load this data, split it into separate groups for different purposes, and then train and evaluate your model.

Part 1: Preparing for the Challenge

Required Reading Materials

You should also read the details of the challenge at https://physionet.org/challenge/2019/. Pay particular attention to what kind of data you will be working with and what the objective of the challenge is.

Optional Reading Materials

Machine learning is a vast and deep topic, and this assignment will only scratch the surface. Although you should be able to complete this assignment strictly by following our instructions, it may help to read through some materials to familiarize yourself with important concepts. There are hundreds of blogs, videos, and online courses that people use to learn about machine learning. Here are a couple of our favorites:

- Jason Mayes' Machine Learning 101

- Machine Learning Mastery

If you would prefer to look at academic materials, we recommend going through Roger Grosse's course notes for CSC321: http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/. These particular materials may be of interest:

- CSC411 Intro: What is machine learning, history of machine learning
- CSC321 Lecture 2 Slides: Fitting polynomials (useful visualization), generalization
- CSC321 Lecture 2 Notes: Generalization
- CSC321 Lecture 3 Notes: Section 1, intro paragraph of 2.

Package Installation

We will need to take advantage of numerous software packages that will give us access to functions that will make our lives easier. You have already seen one of these packages (`numpy`) in a previous assignment, but the full list is below:

- `numpy` provides efficient implementations of many math operations, particularly ones that involve arrays and matrices.
- `pandas` is an extremely useful library for working with data, especially when your data can be organized into tables (e.g., a `.csv` or `.xlsx` file).
- `scikit-learn` is a tool for data mining, data analysis and machine learning.
- `imbalanced-learn` contains functions to help you work with imbalanced data.
- `cache-em-all` will allow us to save the result of a function so that it will only take a long time the first time you call it. In other words, `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 you call it again, it will only take a couple of seconds to load the saved result. This package was created by a TA from a previous iteration of the course.

`numpy`, `pandas`, and `scikit-learn` are some of the most popular ones for doing data science with Python, so getting familiar with these packages would be a good idea for both this assignment and any future projects you might do on your own.

If you are running your program on a local computer, you will need to install these packages yourself. Google Colab will already have some of these packages pre-installed, but we will confirm this and install the missing packages by running the following command:

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

This is not Python code, but rather a 'bash' command that you would normally run in a local terminal.

Part 2: Loading the Data

Downloading the Data

One way you could get the into your program would be to download the `.zip` file(s) provided by the challenge website, extract their contents, and then putting those files in the working directory of your code. However, this is time-consuming and would require you to repeat the process if the data were to change. What we can do instead is run `bash` commands to download the data programmatically:

```
!wget -nc https://archive.physionet.org/users/shared/challenge-2019/training_setA.zip
!unzip -n training_setA.zip
```

DataFrames in Pandas

The `pandas` library has a special data structure called a `DataFrame`. It is very similar to a table in that it has rows and columns, and each column can have a name assigned to it. We can create a `DataFrame` in many different ways, but for the sake of this example, we will manually create one with random values:

```
np.random.seed(0)
df = pd.DataFrame(np.random.rand(6,2), columns=['hemoglobin', 'bilirubin', 'uric acid'])
```

Similar to a dictionary, you can access specific columns within the `DataFrame` by putting a column name or a list of column names within square brackets:

```
df["bilirubin"] # Retrieves one column
df[["hemoglobin", "bilirubin"]] # Retrieves multiple columns
```

You can take this a step further to assign values to all the rows of a given column. For example, the following code converts the hemoglobin values from g/dL to g/L by multiplying them by a factor of 10:

```
df["hemoglobin"] = df["hemoglobin"] * 10
```

You can also create new columns with a default value for all of the rows. In this example, we have created a new column for people's mood with the default value `"happy"`:

```
df["mood"] = "happy"
```

Each row is assigned an numerical **index** value. By default, each row's index is the same as its row number (i.e., the first row has an index of 0, the second row has an index of 1, etc). However, this may not always be true since there may be situations when you need to either index your rows differently or shuffle your rows while keeping track of their original position. You can access specific rows of a `DataFrame` using either its position in the `DataFrame` (with the method `.loc[]`) or its index (with the method `.iloc[]`):

```
df.loc[0] # The first row in df
df.iloc[0] # The row in df with the index 0
```

For the purposes of this assignment, you should just be aware that each row is associated with a numerical index.

Loading a Single File

We will first create a helper function called `load_single_file()` that will load a single file from the dataset folder that you just created. These files are in the `.psv` format. Just like how a `.csv` file contains values separated by commas (`,`), a `.psv` file contains values separated by pipes (`|`). Your function should return a `DataFrame` after doing the following:

1. Read in a file using the `pandas` function `read_csv()`. As its name suggests, this function typically expects values to be separated by commas; however, the optional `sep` argument enables us to specify the character that the file uses to separate values. Here is an example of this function being used for a `.psv` file:

```
df = pd.read_csv("training/p00001.psv", sep="|")
```

2. Create a new column called `patient` that contains the name of the file from which we retrieved this data. We will use the file name as a sort of patient ID to keep track of which rows belong to which patient.
3. Create a column called `hour` that represents when each row was collected. You may recall from the challenge description that each row in the file represents an hour. This means that we can use the row number (or index) to as the hour value for each row:

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

Loading All of the Data

Now that we can read one file, we will write a helper function called `load_data()` that will read all of the data files and put them into a single `DataFrame`. One way we could do this is by continuously appending rows to a single `DataFrame`, but that is very slow. Instead, what we will do is read each file in our dataset, append the resulting `DataFrame` to a list, and then use the `pandas` function `concat()` to concatenate the list of `DataFrames` into a single one. Here are the steps your function should follow:

1. Get a list of all the filenames in the dataset using the `get_data_files()` function we have provided.
2. Call the `load_single_file()` helper function you wrote earlier on each filename in that list and append the result to a list.

3. Use the function `pd.concat()` to combine the `DataFrames`. Note that this is only possible because each `DataFrame` in the list has the same columns.
4. Pre-process the data using the `clean_data()` function we have provided.

Once you are sure this function is working correctly, you can uncomment the `@Cachable("data.csv")` line above the function. The next time you run the function, its result will be saved and subsequent calls to the function will load the saved data. If you realize there was a mistake in your implementation and you have to fix it, you will need to delete the `cache` folder; otherwise, your fixed code will not be called and the old saved data will still be returned.

Data Exploration

Once you have loaded the data, run the following commands to understand some high-level characteristics of your dataset:

```
df = load_data()
print("Column names:", df.columns)
print("Number of samples:", len(df))
print("Distribution of sepsis vs. non-sepsis samples:", df["SepsisLabel"].value_counts())
```

Moving forward, we will consider all of the columns other than the patient ID to be our **features** (the inputs to our model) and our **label** (the desired output of our model) to be `SepsisLabel`. Because we are working with data that has labels, we will be using a type of learning called **supervised learning**. More specifically, we are trying to predict an output y given an input X and we have examples of (X, y) pairs that we can use to train our system. This is in contrast to **unsupervised learning** where we would only have X at our disposal.

There are two types of tasks within supervised learning: **regression** and **classification**. Regression involves continuous labels like white blood cell count or the price of a house. Classification involves discrete labels like healthy/sick or cat/dog/bird; the different possible values that a discrete label can take for a given problem are known as **classes**. We will be doing binary (2-class) classification in this assignment.

Part 3: Machine Learning Basics

Splitting the Data

We want the model we train to be able to produce accurate predictions on data it has not seen before. One way we can do this is by splitting our data into two sets: (1) a **training dataset** that we use to train the model, and (2) a **test set** that will only be used to evaluate the model. The `scikit-learn` library provides a function called `train_test_split()` that splits data into train and test splits for us:

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

The parameter `test_size` is the proportion of data that should be used for the test set. In this example, 20% of the data goes to the test set `test_df`, which means the remaining 80% goes to the training set `train_df`.

Training a Classifier

There are many different types of classifiers available in `scikit-learn`, but we will start with a decision tree classifier. Look through the online documentation for `DecisionTreeClassifier` and complete the following steps for `train_simple()`:

1. Separate the features and the labels for both `train_df` and `test_df`. For your convenience, some of the code your ran earlier has variables called `feature_cols` and `label_col`.
2. Create a new instance of the `DecisionTreeClassifier` with `class_weight="balanced"`.
3. Pass the training data through the classifier so it can identify the best structure and model weights that will maximize its accuracy; this process is often known as **fitting**.
4. Use your trained model to **predict** labels for the training features.
5. Use the `evaluate()` function we have provided to print out accuracy metrics that explain how often the predicted labels matched the expected labels.
6. Repeat the previous two steps using the test dataset.

Making Your Results Repeatable

What happens if you run `train_simple()` multiple times? The results are not consistent because there are multiple parts of our code that rely on randomness: how we split the data into training and test sets, how the model fits itself to training data, etc. Inconsistent results make it hard to replicate our work, so we should have a way to be able to produce the same result every time.

Many random number generators are not truly random; they are actually pseudorandom in that they generate numbers based on a **seed**. Therefore, if we set the value of the seed, we can control the sequence of random numbers the generator produces. We can do this by using the following lines of code:

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

You can set the value of your random seed to be whatever you want; as long as you keep the seed the same, your program should produce the same results. These lines of code should

be ran before any part of your program that involves randomness. We recommend putting it next to the `import` statements from the first part of the assignment. We also need to pass this seed to the `train_test_split()` function:

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

Re-run your `train_simple()` function multiple times to make sure it produces the same results each time.

Part 4: Improving Your Pipeline

Dealing with Overfitting

How well does the model work on the training dataset? How about on the test dataset? The model likely worked better on the training data for a variety of reasons. The test dataset may have some rows that are completely different from what was used to train the model, which would lead to a higher chance of misclassification. The model might also be memorizing the training data so well that it lacks the flexibility to generalize to unseen data that is roughly similar; we call this phenomenon **overfitting**.

One way of addressing overfitting is by tuning the complexity of the model. A model that is too simple may not have enough flexibility to capture the complex nature of the dataset, while a model that is too complicated may be so flexible that it can memorize the dataset; the best model will often lie between these two extremes. For the decision tree classifier, the model complexity is dictated by the width and depth of the decision tree, which we can control as follows:

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

Try different values for the model complexity parameters in your `train_simple()` implementation. We suggest varying those parameters in increments of 10 and looking for general trends; otherwise, you may overoptimize the complexity of your model for your particular data split.

Dealing with Class Imbalance

Recall that there is a significant difference between the number of positive (`SepsisLabel = 1`) and negative examples (`SepsisLabel = 0`) in our dataset. When a dataset is significantly imbalanced, classifiers may become biased because there are too few examples of a particular class.

Within the `DecisionTreeClassifier`, we can set `class_weight="balance"` to tell the classifier to weigh classes according 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 incorrect predictions on positive examples twice as bad as a incorrect predictions on negative examples while training.

Another way to deal with class imbalance is by adjusting how the data is sampled. In this case, we are going to **undersample** the data, which means that were are going to keep all of the data in the minority class and decreasing the size of the majority class. The alternative would be **oversampling**, which means that we would be keeping the size of the majority class and repeating examples in the minority class. In your `train_simple()` implementation, use the `RandomUnderSampler` from `imbalanced-learn` to undersample the training data before you fit your model:

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

Better Cross Validation

In the previous section, we split the data into training and test sets to see how well our model would generalize to unseen data. However, those splits were not as independent as they should be. The `train_test_split()` function randomly splits the rows in the dataset, but multiple rows belong to same patient. This means that data from patient P_1 can appear in both the training and test datasets. If that happens, the model is essentially "peeking at the answers" ahead of time, so even if a model has high accuracy on the test dataset, we cannot argue that it will work for other unseen patients.

To fix this issue, we will split our data using a variant of **k-fold cross-validation**. A typical k-fold cross-validation procedure goes as follows:

1. Rather than splitting the data into a single pair of training and testing sets, we split the data into k number of sets (called **folds**) of equal size.
2. For each fold, we do the following:
   (a) We select one fold to hold out as the test set and use the remaining $k-1$ folds collectively as the training set.
   (b) We fit a model on the training set and evaluate on the test set as we would normally do.
   (c) We save the prediction results for the training and test datasets, but we discard the model.
3. At this point, all of the folds will have been the test set at least once, which means we will have a prediction for each sample. We can therefore calculate the performance of our modeling approach the same way as we did before.

This still does not solve our problem of maintaining independence, which is why we need to use a variant of k-fold cross-validation. We will stratify our data according to the `patient`

column before we split it into folds so that rows from the same patient can only appear in one fold. `scikit-learn` has an object called `GroupKFold` that will do this for us:

```
X = data[feature_cols]
y = data[label_col]
group = data[stratify_col]
kf = GroupKFold(n_splits=5)
kf.split(X, y, group)
```

We have implemented the skeleton for a stratified 5-fold cross-validation procedure in `train_stratified()`. Your job is to complete this function by writing code that trains a classifier within the loop. You may re-use your `train_simple()` function as you see fit.

When this function is complete and you run it, you should expect to see a drop in accuracy since you are no longer "cheating" during model training; nevertheless, this is a more accurate representation of how well your approach will be able to generalize for unseen patients.

Part 5: Putting It All Together

Now that you have written and iterated upon a pipeline for model training, it is time for you to find a configuration that maximizes cross-validated test accuracy. There are numerous aspects of the pipeline that you can tweak, but here are the easiest ones to start with:

- **Model architecture:** We have been using the `DecisionTreeClassifier` in `scikit-learn`, but there are many others at your disposal. We have imported a few others that you can try — `KNeighborsClassifier`, `RandomForestClassifier`, and `MLPClassifier` — but you can also explore other ones on `scikit-learn`'s website.
- **Model parameters:** We examined how adjusting the model complexity can help us avoid overfitting. As you try different models, be sure to also explore different parameters.
- **Number of folds:** The skeleton we wrote for `train_stratified()` performs 5-fold cross-validation because of the parameter we passed to `GroupKFold`. What happens when you increase the number of folds to 10? What happens when you decrease the number of folds to 2 or 3?

You could also improve your pipeline by looking into feature pre-processing, feature selection, and automated hyperparameter tuning. However, these topics are outside of the scope of this course. If you are interested in learning more, feel free to reach out to the instructors!

The final deliverable for this assignment is a report that explains the different configurations you tried, the accuracy those configurations achieved, and written explanations of why you believe those results happened. Since there are so many configurations to choose from and each person may be splitting the data differently, we are not expecting everyone to achieve a definitive correct answer. What we are looking for is careful experimentation and viable

explanations for the results of those experiments. You may find it helpful to use tables or graphs to systematically present your accuracy numbers. The report should be single-column, single-spaced, and no longer than 3 pages including tables and graphs. Save the report as either `report.pdf` or `report.docx`.

What To Submit

- ☐ `project3.ipynb` (as a Google Colab link): Your completed Jupyter notebook.
- ☐ `report.pdf / report.docx`: A report that demonstrates a thorough investigation of different pipeline configurations.

References

[1] Kumar, A., Roberts, D., Wood, K.E., Light, B., Parrillo, J.E., Sharma, S., Suppes, R., Feinstein, D., Zanotti, S., Taiberg, L. and Gurka, D., 2006. Duration of hypotension before initiation of effective antimicrobial therapy is the critical determinant of survival in human septic shock. Critical care medicine, 34(6), pp.1589-1596.

[2] Seymour, C.W., Gesten, F., Prescott, H.C., Friedrich, M.E., Iwashyna, T.J., Phillips, G.S., Lemeshow, S., Osborn, T., Terry, K.M. and Levy, M.M., 2017. Time to treatment and mortality during mandated emergency care for sepsis. New England Journal of Medicine, 376(23), pp.2235-2244.