hw06: Introduction to Kaggle / Titanic Survivors Prediction

Due: See Moodle posting

What you'll learn in this assignment

In this assignment, you'll participate in a competition on Kaggle.com, making predictions from a dataset. Kaggle.com is a website that hosts data science competitions – some for significant cash prizes. It's pronounced "Kaa-gle", with the first syllable rhyming with "wag" like a dog's tail. Participating in Kaggle competitions is a nice thing to put on your resume; I've seen some job advertisements mention such competitions explicitly, in fact. Go to Kaggle.com and create an account.

I set up a couple situations for you that will be similar to some of the challenges you'll face in the semester project: how to properly train and test your model, and how to consistently preprocess your data. At the end of this assignment, you will submit your predictions to Kaggle as a participant in the contest.

---

1) **Getting started in the Kaggle competition**

You may already be aware of the sinking of the ship Titanic, in 1912. If you're interested in some history about it, you could check out https://en.wikipedia.org/wiki/RMS_Titanic, for example.

In this project, you'll try to predict who among a list of passengers survived the sinking of the Titanic. You'll do this using information about the passengers, where their cabins were, etc. First, please copy and rename the hw06-given.py file to *lastname1-lastname2*-hw06.py.

Please go to kaggle.com, sign in, and then visit:
https://www.kaggle.com/c/titanic
Under the "Overview" tab, read the "Evaluation" sections. You can read other sections if you want.

Next, visit the "Data" tab and download the test.csv and train.csv files. Put them in a "data" folder inside of the folder that contains your .py file.

Feel free to browse around the other pages as much as you want, but you won't need them for this assignment.

---

2) **Maintaining the Integrity of Model Testing ("Don't let your system cheat!")**

Note the preprocess function in the given .py file. This function takes two DataFrames as arguments: targetDF and sourceDF. Let's discuss why I've written the preprocess function to take these two arguments.

Recall the philosophical difference between a training set and a testing set. The training set is what you use to develop a model. The testing set is what you use to test the model. You don't want information from your testing set to sneak into your model development process – that would be "cheating", or *data leakage*, giving your model some or all of the "answers" to the test. This would mean your results, while

perhaps seeming high, would not actually give an accurate picture of the generalizability of your model. So, again, it's important to keep the testing set out of the development process for your model.

Now, suppose you're about to preprocess your training set. Some preprocessing is not based on the training set data itself. For example, you can change the type of the Sex attribute to 0 for male, 1 for female, without having to know anything about the actual data in the training set. (For this historical dataset of Titanic passengers, only "male" and "female" are listed for sex.)

But some preprocessing of the training set is actually based on the training set data itself. For example, we might replace missing values in the Age attribute with the median age across the training set. So in that step we *use the training set to preprocess the training set*.

If we preprocess the training set, and build a model using that mutated training set, then we also need to preprocess the testing set in the same way. For example, we couldn't build a model expecting 0 or 1 for Sex, and then test it on "male" and "female" strings. So the testing set needs to be preprocessed in the same way as the training set.

Here's the tricky thing, though. Suppose, in the testing set, we want to replace missing values in the Age attribute with the median age – just like we did in the training set. What median age should we use – the training set's median age, or the testing set's median age? The generally agreed-upon approach is to use the *training* set's median age to fill in missing values for the *testing* set. More generally, we should *use the training set to preprocess the testing set*, rather than using the testing set data to preprocess the testing set. One practical reason is that the training set is typically larger than the testing set. But more importantly, philosophically, if we use the testing set itself to preprocess the testing set, then we're letting the complete testing set sneak into the model testing process. We need to keep the testing set completely separate from the model, feeding just one test row at a time to the model. Otherwise, information from the testing set as a whole is sneaking into the process, and we're not getting a completely accurate picture of the generalizability of the model in the real-world.

So, when we preprocess both the training and testing sets, we need to do this using data only from the training set.

This is why the provided preprocess function takes two arguments. The targetDF is the DataFrame you're actually preprocessing – the one you're mutating. The sourceDF is the DataFrame you're using to do the preprocessing. Note in the preprocessing function how the two DataFrames are used according to this principle. Specifically, in filling in missing Age values, the targetDF is mutated, but the median is computed from the sourceDF.

Look in the provided buildAndTestModel function, at the two calls to the preprocess function. Note that preprocessing for the training set uses the training set as both the targetDF, and the sourceDF. In contrast, the preprocessing for the testing set uses the testing set as the targetDF (of course), but the training set as the sourceDF. This is all according to the principles above.

Now that you've read all this and are beginning to see what I'm getting at, I might recommend that you read through this problem again. Please then let me know how I can help with any questions you may have.

---

3) **Changing Types**

Consider the provided code. Let's focus on the following lines for a moment, which convert the Sex attribute into a numerical value.

```
targetDF.loc[:, "Sex"] = targetDF.loc[:, "Sex"].map(lambda v: 0 if v=="male" else v)
targetDF.loc[:, "Sex"] = targetDF.loc[:, "Sex"].map(lambda v: 1 if v=="female" else v)
```

You might reasonably ask why we couldn't just do something like this:

```
targetDF.loc[:, "Sex"] = targetDF.loc[:, "Sex"].map(lambda v: 0 if v=="male" else 1)
```

The one-line version is equivalent to the two-line version only when there are just two possible values. But suppose, for example, that there were also *missing* values in this column. Then the one-line version would set all missing values to 1, while the two-line version would leave missing values alone. Do you see why?

We should not just automatically set missing values to 1. Rather, we should set it according to a more principled approach, like by using the mode (or average for continuous features), or the mode (average) of k neighbors, etc.

So I provide in the given code the two-line version, as an illustration. It turns out that there are no missing values in the Sex attribute, and so either approach would work for that attribute. But I provide the two-line version to illustrate how to be more cautious about missing values. This will be important when you work with other attributes in this assignment.

---

4) **Building a Model**

The algorithm we'll apply here is called logistic regression. We'll talk about it more soon. For now, just know that this algorithm attempts to fit a line (really, a high-dimensional plane) to the data, and outputs a discrete value for a prediction – either 0 or 1.

Let's just use certain columns in our model. Specifically, let's use the Pclass, Sex, Age, SibSp, Parch, Fare, and Embarked columns. We'll refer to these columns as our *predictors*. They're the input columns. Note, also, that the target attribute (the output column) is called "Survived".

In the space indicated in the buildAndTestModel() function, build a LogisticRegression object by calling the constructor. Use Google to discover how to import the class into your code.

What arguments do you need for the LogisticRegression constructor? If you don't specify any, you'll get a warning describing changes to the default behavior of the solver parameter. In order to silence the warning, and to be certain we have identical results, use the named parameter:
`solver='liblinear'`

Next, do 3-fold cross validation by passing the resulting object to cross_val_score as you've done in previous assignments. Split the titanicTrain DataFrame into two: an input DataFrame in which you select just the predictors, and an output Series of the Survived column. Print out the resulting mean score.

(If you feel uncertain about what you're doing here, peek back at prior assignments when we used cross_val_score. This is very similar!)

Unfortunately, when you try to run your code, you'll get an error: "could not convert string to float: 'Q'". (If you get some other error, then something else is wrong, that you'll need to fix first.) Work backwards in the error message stack trace and note that the error begins on the line where you call cross_val_score. The problem is one of our predictors needs to be preprocessed. Logistic regression requires all predictors to be numeric, but one of them is not. Figure out which one and fix it, using the approach for the Sex attribute as a guide. Just so we're all consistent, please assign values like this: Assign 0 to the value that occurs first alphabetically, 1 to the value that occurs second alphabetically, etc. You can hardcode this. Add code as needed in the preprocess function to do this.

(We might reasonably worry about introducing an arbitrary order to nominal values – this is a concern, but we'll ignore it for this homework.)

Once you fix this issue, you'll get another error: "Input contains NaN, infinity or a value too large for dtype('float64')." This is because the attribute you preprocessed above has missing values too! Replace the missing values with the mode for that attribute, using the sourceDF mode, of course. (Don't forget that the mode method returns a Series, so you need to get the element at position 0 of that Series.)

When you fix the above errors (using the alphabetical ordering as suggested), you should get an average accuracy for 3-fold cross validation of 0.7923681257014591.

---

5) **Testing the Model**

You'll note that this section is called "testing the model" – but didn't we already do that with 3-fold cross validation in the previous step? Well, kind of. We took the training set, titanicTrain, and passed it to cross_val_score. That function split up titanicTrain into "training" and "testing" sets for the 3-fold cross validation. Notice, however, that we haven't actually used titanicTest at all yet. titanicTest is our testing set, in a sense, except that we don't actually know the correct answers to titanicTest! The correct answers are known by Kaggle.com. So our next task is to take a model built on titanicTrain, use it to predict titanicTest, and upload the results to Kaggle.com. Kaggle.com will tell us how well we did, without actually telling us which ones we got right or wrong.

To put it another way, we read in the Kaggle training set and stored it in titanicTrain. That got split up, by cross_val_score, into an experimental training set and experimental testing set. We did this just to test our model ourselves. That was all step (4). Now in step (5), we're going to build a model using the entire Kaggle training set (titanicTrain), get its predictions on the entire Kaggle testing set (titanicTest), and upload our results.

In the designated area of buildAndTestModel, make a new LogisticRegression object as you did in the previous part. Call the fit method on it, passing it the training data as an input DataFrame and output Series. Then call the predict method on it, passing it the testing set as an input DataFrame. If you need a refresher on how to use the fit and predict methods, peek back at hw05, testOneNNClassifier for an example. (That was for our own OneNNClassifier class, but remember we built it according to the same specifications that built-in classes like LogisticRegression use.) And don't forget to select only our predictors for the inputs!

Now, the predict method should return a numpy array of predictions (of type ndarray). Unfortunately, even when you've done everything correctly so far, you'll get another error: "ValueError: Input contains NaN, infinity or a value too large for dtype('float64')." This is the same issue you had with another attribute before. It turns out that in the testing set, there's a missing value for an attribute that didn't have any missing values in the training set. (That's why this particular issue didn't come up in problem (4).) Fix this issue in your preprocess function, replacing the missing value with the median from the *sourceDF* (not the targetDF!).

When you're done, add the following to the top of your file:

```
from collections import Counter
```

Then you should be able to print your predictions like this:

```
    print(predictions, Counter(predictions), sep="\n")
```

(assuming you stored predict's return value in the variable `predictions`)
and get the following output:

```
[0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 1
 1 0 0 0 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 1 0 1 1
 1 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0
 1 1 1 1 0 0 1 1 1 1 0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
 0 0 1 0 0 1 0 0 1 1 0 1 1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1
 0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1 1 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0
 1 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 0 1
 0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 0 1 0 0 0 0
 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0
 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 0
 0 1 0 0 1 1 1 0 0 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 1 1 0 0 0
 0 1 1 1 1 1 0 1 0 0 0]
Counter({0: 261, 1: 157})
```

The last line just counts the number of 0's and 1's, for your convenience in checking your work.

---

6) **Submitting the Model's Results to Kaggle**

So, we've created a model using the Kaggle training set, and generated predictions for the Kaggle testing set. But we don't know if those predictions are accurate or not, because we don't have access to the answers for the Kaggle testing set. So we need to create a .csv file of our predictions and upload that file to Kaggle. Kaggle will then give us a score and show us where we are in the rankings for this contest.

On the Kaggle page for the Titanic contest, go to the "Data" tab. Read what gender_submission.csv is, and download it. Open up gender_submission.csv. These predictions are terrible, but they show us the structure we need to create for our own predictions.

As you may have guessed, Pandas makes this easy. First, we need to make a DataFrame containing just two columns, in order: PassengerId, and Survived. There are multiple ways to do this, but one way is:

```
submitDF = pd.DataFrame(
                {"PassengerId": Series of PassengerId values,
```

```
            "Survived": Series of predictions obtained from predict method
        }
    )
```

In the above code, we're just making a dictionary mapping column name to Series of values, and passing that dictionary to the DataFrame constructor.

Next, we need to write the submitDF variable to a .csv file:

```
submitDF.to_csv("data/filename.csv", index=False)
```

Google the pandas to_csv function for a moment to see what the index=False part is doing and think about why we need it.

With that .csv file generated, on the Kaggle site, hit "Submit Predictions", drag your .csv file to the designated area, and hit "Make Submission". You should end up with a score of 0.76315. We would need to do more work to actually move up in the rankings, but that's fine – our purpose here is just to learn how to participate in a Kaggle contest and how to address some common preprocessing and testing issues.

Preparing for Grading

Run the provided test06 function. You should get the following output:

```
0.792368125701
[0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 1
 1 0 0 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 1 0 1 1
 1 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0
 1 1 1 1 0 0 1 1 1 1 0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
 0 0 1 0 0 1 0 0 1 1 0 1 1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1
 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1 1 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0
 1 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 0 1
 0 0 0 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 0 1 0 0 0 0
 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0
 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 0
 0 1 0 0 1 1 1 0 0 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 1 1 0 0 0
 0 1 1 1 1 1 0 1 0 0 0]
Counter({0: 261, 1: 157})
```

The first number is the mean score resulting from 3-fold cross validation just on the Kaggle training set.

Congratulations! You've just participated in your first Kaggle competition! More importantly, you understand more about preprocessing data, avoiding data leakage, and making models from pre-defined classes.