COSC 2673 Machine Learning

Lab 04

Objective

- Practice loading datasets
- Perform basic data preparation
- Learn about and perform logistic regression
- Learn how to perform basic parameter tuning

Introduction

In this lab, we will learn more of the machine learning pipeline, including examining and performing basic data cleaning. We then examine how to perform logistic regression, learn two basic metrics to evaluate this (we will cover more metrics in this week's lecture) and perform basic parameter tuning to demonstrate how it can be done. We will apply it to predicting whether NBA rookies will play five years or more.

Datasets

In this lab, we will be using a dataset of NBA rookies, some of their stats and trying to predict whether they will still be playing after 5 years. The data is available on Canvas, in the same location as where you got this lab worksheet.

Data Preprocessing

We will first study how to perform some basic data pre-processing. Starting this week, we will progressively provide less code, and would like you to use previous labs and what you know to perform the tasks. This will help you to become proficient at this.

Open up Anaconda and create a new Jupiter Notebook session. Download the datasets into the working directory of your session.

Once you have the session and data downloaded, first import pandas, sklearn, numpy and matplotlib.pyplot.

Next, we want to load the dataset 'nbaPlayers.csv' into a Pandas dataframe (call it nbaDf, so we can have a consistent variable name to refer to). Hint, use a delimiter of ',' (comma). Remember to check if your dataframe was loaded correctly by print it out.

Lets plot a quick histogram to understand the distribution of the data more (assuming pyplot has been imported):

$ pyplot.figure()
$ nbaDf.hist()
$ pyplot.show()

If we tried to run this with a classifier, we will find it will complain about NaN values. What NaN values generally mean is that there was missing values in the loaded data. Lets examine them:

$ pandas.isna(nbaDf)

That outputs the whole dataframe and entries with True means the value is NaN or None. Given the size of the dataframe, it is hard to visualise it. Please check up the reference for isna() also, at https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.isna.html .Lets do a count instead, using the following:

$ pandas.isna(nbaDf).sum()

You should find that 3P% column as 11 such values. Lets see which instances/rows/players this corresponds to:

$ nbaDf[pandas.isna(nbaDf).any(axis=1)]

Ah, we find these players all have 0 3PA (3 pointer attempts), hence can't have a 3 pointer percentage (3P%). There are several ways to deal with this, but in this case if there are no 3 pointer attempts, then we can set their 3P% to 0.

$ nbaDf.fillna(0)

This essentially fills all NaN entries with 0 (remember to check documentation for details of method). There is another useful function to deal with NaN and missing values called interpolate, that tries to infer values – again check documentation for details. Another option is to drop the row/instance if it appears the instance might be erroneous or there is no good way to fill or infer.

Final task in this section is to setup the feature/attribute data and the column we are predicting 'TARGET_5yrs':

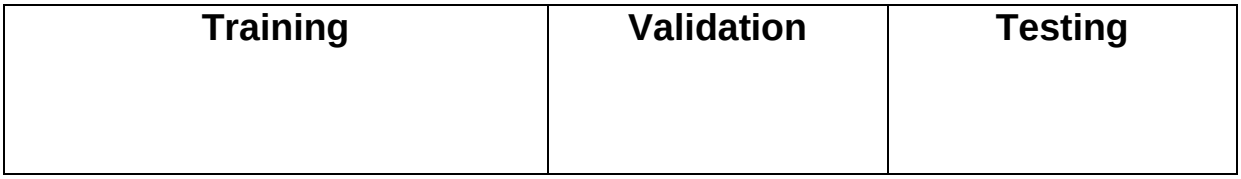
$ nbaFeatDf = nbaDf.drop(['Name, 'TARGET _5Yrs'], axis=1)
$ nbaLabelsDf = nbaDf[['TARGET_5Yrs']]

What do these commands do? Why do you think we are also removing the name attribute? (hint: consider whether name is a useful feature to learn decision boundaries, considering all players have a unique name). Ask your demonstrator if you are not sure.

Setting up training and testing Data

Similar to last week (and we'll discuss in lectures about evaluation), we will divide our data into a number of testing datasets.

Dissimilar from last week, we will also have an addition partition called validation dataset. Consider the diagram below.



Whole Dataset

What we want to do is to use the training (data)set to construct the model, then use the validation set to tune the parameters of the model. Then once the parameters + model are tuned, we evaluate it on the testing set. This reduces the risk that we overfit if we use the testing set to tune the parameters (something we will talk about in lectures).

Scikit-learn doesn't have a function to split the data into the three sets. Instead, we can call it twice! First lets split into training and testing dataset, as per last week (remember to import the relevant packages):

```
$ nbaTrainFeat, nbaTestFeat, nbaTrainLabels, nbaTestLabels =
sklearn.metrics.train_test_split(nbaFeats, nbaLabels, test_size=0.2)
$ print(nbaTrainFeat.shape)
$ print(nbaTestFeat.shape)
$ print(nbaTrainLabels.shape)
$ print(nbaTestLabels.shape)
```

This will split data into training set consisting of 80% of the data, and testing the remaining 20%. To generate the validation set, we further split the data into 60% new training set and 20% validation:

```
$nbaTrainFeat, nbaValFeat, nbaTrainLabels, nbaValLabels = =
sklearn.metrics.train_test_split(nbaTrainFeat, nbaTrainLabels, test_size=0.2)
```

COSC 2673 Machine Learning

Lab 04

Now we are almost ready to perform some classification via logistic regression.

Evaluation metrics

As stressed in lectures evaluation metric is an important decision choice and can lead to different conclusions about the best model. We haven't covered metrics for classification yet (bit unfortunate with the scheduling of the labs), but we will use two metrics that we will discuss more in lectures.

The first one is accuracy, which as its name suggest, is the percentage of instances where the predicted and tested class labels are the same (over total number of instances). The other measure is called F1-score, something we will discuss in class, but for now consider it as another measure of how good a classifier model + parameters are – higher F1-scores are more desirable, and it ranges from 0 to 1.

Logistic Regression

We now ready to do some classification! Similar to last week, lets first construct a logistic regressor (remember importing relevant packages):

```
$ logRegNba = sklearn.linear_models.LogisticRegression(C=10000, max_iter=100)
```

A quick explanation of the parameters. C specifies the amount of weighting placed on minimising the error. The standard implementation of logistic regression in Scikit learn includes regularisation, something to prevent overfitting and (again) discussed in coming lecture. But as we haven't learnt it yet, we don't use regularisation by setting C to a large value, which effectively negates the regularisation. The max_iter is the maximum number of iterations to fit parameters to the model. Although typically not a parameter that we tune too much, to demonstrate how parameter tuning works we will try various settings of max_iter to find the best setting (for the data).

First, lets fit the parameters and perform the prediction on the trained model. Look up how we did this for linear regression, the functions are actually the same name. If unsure, also look up documentation of LogisticRegression (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). Remember we want to train on the training data (nbaTrainFeat and nbaTrainLabels) and evaluate on the test data (nbaTestFeat and nbaTestLabels).

Evaluate the results as follows, assuming the predicted values are in 'pred':

```
$ from sklearn.metrics import f1_score, accuracy_score
$ print(f1_score(nbaTestLabels, pred))
$ print(accuracy_score (nbaTestLabels, pred))
```

What was the F1-score and accuracy?

Parameter Tuning
To evaluate how to set max_iter, we will try different values and see their performance on the validation set.

We first set up the parameters of our experiments, where we range max_iter from 2 to 1000 and setup lists to store the F1 and accuracy scores.

```
$ lMaxIter = [2,5,10,50,100,250, 1000]
$ lF1Score = []
$ lAccuracyScore = []
```

We then train a logistic regression model for each max_iter value, using the training set to train. We don't use the test set to evaluate max_iter, but use the validation set instead. The following code loops through each value of max_iter, trains model, evaluates then score the scores in LF1Score and lAccuracyScore.

```
$ for maxIter in lMaxIter:
$       currLogRegNba = sklearn.linear_models.LogisticRegression(C=1,
        max_iter=maxIter)
$       currLogRegNba.fit(nbaTrainFeat, nbaTrainLabels)
$       currPred = currLogRegNba.predict(nbaValFeat)
$       lF1Score.append(f1_score(nbaValLabels, currPred))
$        lAccuracyScore.append(accuracy_score(nbaValLabels, currPred))
```

Lets have a look at the scores and also plot the trend as we change max_iter:

```
$ print(lF1Score)
$ print(lAccuracyScore)

$ pp.figure()
$ pp.plot(lMaxIter, lF1Score)
$ pp.plot(lMaxIter, lAccuracyScore)
$ pp.show()
```

What do the results tell you? What would be a good value to set max_iter, assuming we want a setting that runs long enough for model to converge but not have wasted cycles. If this behaviour is repeated for other datasets, what does that suggest about max_iter (hint: consider the results and whether a good enough value is acceptable).

Select what you consider as the best parameter setting for max_iter, and evaluate your model (with this setting) on the test dataset.

Try constructing a few new train-validation-test splits/sets again. Did the best parameter setting changed. Does that change your mind about your initial answer? (we will discuss this in lectures)