Multi-class classification on gene expression data

Simone Daniotti[1] and Riccardo Castelli[2]

[1]*Physics Department, University of Milan*
[2]*Informatics Department, University of Milan*

September 26, 2019

Abstract

Many recent studies focussed their attention to the relationship between gene expressions and cancer types, for example [1]. The goal of our work is to build learning algorithms that can classify cancer types starting from gene expression levels of patients. In doing so, we analyzed efficiency and computational performances of the most famous python libraries that enable users to build such models.

Contents

1 Dataset description

The dataset for this work is taken from UCI Machine Learning Repo (available at https://archive.ics.uci.edu/ml/datasets/gene+expression+cancer+RNA-Seq) [2], and it is part of the RNA-Seq (HiSeq, a tool for measuring gene expression levels) PANCAN dataset. The dataset is a collection of gene expression levels of patients having different types of tumor: BRCA (breast), KIRC (kidney), COAD (colon), LUAD (lung) and PRAD (prostate). These data represent the quantity of gene information used in the synthesis of a functional gene product.

2 Dataset Manipulation

2.1 Preliminary manipulation

Both dataset and labels can be downloaded in a .csv ('comma separated value') format, that then can be easily imported into a Pandas Dataframe (https://pandas.pydata.org/). The first column, which represents the patient's ID has been removed as it is useless for our purposes.

2.2 Label handling

The labels are strings representing the five types of cancer. Learning models can be created using raw features, as in the case of trees and forests, Label Encoding or One-Hot Encoding. Label Encoding creates a map between the strings representing the categories and an ordered sequence of natural numbers, in our case from 0 to 4. One-Hot Encoding creates a single binary label for each class, where the only 1 present in the vector represents the index of the class the example is associated with.

2.3 Normalization

We normalized our dataset using the *normalize* method present in Scikit-learn library *preprocessing*: this method scales vectors individually to unit norm.

2.4 Train and Test Set

For training the models and evaluate them, we split the dataset into training and test set using the Scikit-learn method *train_test_split* (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) from the *model_selection* library. Here we set the seed of the random split (for reproducibility purposes) and the proportions between the two sets: test set is the 15% of our entire examples.

2.5 Class Weighting

Since our dataset is unbalanced, we chose to handle this problem using *class_weight* parameter (which is present in almost every classifier). This parameter can be defined as a dictionary that maps each class in the dataset with its respective weight; weights can be computed using Scikit-learn method *compute_class_weight* in the *class_weight* library, which assigns greater values to the less represented classes. Through this parameter, the classifier is aware of the fact that some classes are more represented than others and gives the examples different importance during the training step.

3 Models

We thought it could be a useful idea to approach our problem with models with a relative simple high-level implementation (such as the Decision Trees, Random Forests and Support Vector Machines) and after their evaluation deepen in more complex ones (such as the Neural Networks).

3.1 Decision Tree Classifier and Random Forests

Decision Trees are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multi-output tasks. They are particularly useful for treating with complex data, such as a dataset that can hardly be represented by a vector in a multi-dimensional space. Tree Classifiers have the structure of an ordered and rooted tree. It is ordered because the children of any internal node are numbered consecutively, and rooted because the splitting starts from only one node. From that node, the model is built following the attribute selection measure. Attribute selection

measure is a heuristic for selecting the splitting criterion that partitions data in the best possible manner.

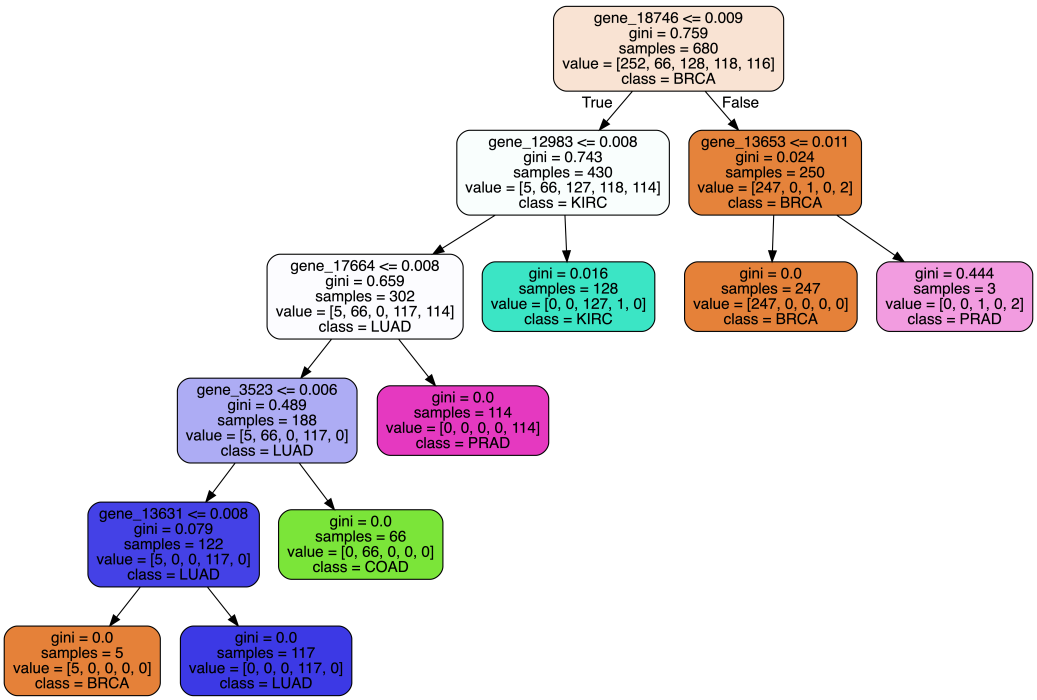


Figure 1: Shape of a tree classifier with tuned hyperparameters

Scikit-learn uses the Classification And Regression Tree (CART) algorithm to train Decision Trees. It works as follows: it firstly splits the training set in two subsets using a single feature k and a threshold t_k ; to choose k and t_k, it searches for the pair (k, t_k) that produces the purest subsets. The cost functions that the algorithm tries to minimize are different: the most used are Gini Impurity and Entropy, which are tunable in Scikit-learn by the *criterion* hyperparameter; the *entropy* hyperparameter measures Shannon's Entropy, a concept taken from the Information Theory. After the first split, the algorithm goes on until a stopping criterion is satisfied (for example the error is minimized). Each leaf of the tree corresponds to a possible classification label.

The aggregation of a group of predictors (regulated by certain rules, such as a majority rule) will often get better results than the best individual one. A group of predictors is called an ensemble, thus, this technique is

called Ensemble Learning and an ensemble learning algorithm is called an ensemble method. Training a group of Decision Tree Classifiers and gathering them into one single predictor is called a Random Forest. The Random Forest algorithm introduces extra randomness when growing trees: instead of searching for the very best feature when splitting a node (as in the tree case), it searches for the best feature among a random subset of features, resulting in a greater tree diversity.

3.2 Support Vector Machines (SVM)

Another approach that we used to create a model that well classifies our dataset is the Support Vector Machines (SVM in short). Both a binary and a multi-class classifier, SVM tries to find the hyperplane (called decision boundary) that best separates the data. The optimal hyperplane is chosen by the use of the so called *supportvectors*, which are the samples of each class closest to the hyperplane. The distance between the support vectors is called *margin*, and the goal of the SVM is to maximise this margin in order to get a good separation of the classes in a way that also guarantees a good generalisation. SVM are particularly effective in high dimensional spaces, even when the number of features is greater than the number of samples: as this fact fits very well with the properties of our dataset, we decided to experiment the classification with this kind of model to see the results.
For the implementation, we utilised Scikit-learn SVC class from the svm library: for multi-class classification, it uses a "one-against-one" approach, creating $n\_class * (n\_class - 1)/2$ (where n_class is the number of classes) classifiers and train each of them with two classes.
We considered three parameters for our model: C, $kernel$ and $class\_weights$. C is called the 'soft margin constant': we can describe it as a measure of tolerance regarding the misclassification of the data, the more the C increases, the less amount of misclassified points the model will tolerate, giving them a huge penalty. During the training process, the model will try find a trade-off between the maximisation of the margin and the minimisation of the misclassification. $Kernel$ is used by the SVM to apply transformations to the data in order to project them in a higher dimensional space where they are more likely to be separable; this method, called the "kernel trick" is used to find non-linear decision boundaries.
The parameter $class\_weights$ is used to make the SVM aware that the dataset is unbalanced.

3.3 Deep Learning

Moving to more complex models, we decided to implement a Deep Neural Network using Keras and Pytorch. In this section we will discuss about the peculiarity of these libraries and in the following we will provide the characteristics of our model, presenting his architecture and the losses and the optimizers we used.

3.3.1 Keras

Keras is a Python high-level machine learning library capable of running on top of multiple back-ends (such as TensorFlow, Theano, CNTK and many more), taking advantage of their power but putting aside their complexity. In fact, Keras allows the user to build deep learning models through a long list of modules that can be selected and combined in a very easy to use manner, which does not however comes at the expenses of the flexibility of the back-end frameworks and does not limit the user during the development and implementation process. The main Keras model is the Sequential, which is a linear stack of layers. The construction of the model requires three main phases: adding layers, compile the model and train it. In the first phase we can add an arbitrary number of layers using the *add* function. There is a consistent number of type of layers that can be selected, for example, in our model, we chose to use 3 Dense (that is fully-connected) layers and 3 Dropout layers for regularization purposes. The second phase is where the model is compiled through the *compile* function; this function needs the optimizer, the loss function and the metrics of evaluation to be specified. The third and last phase is the training of the model: Keras uses the fit function for this purpose, to which we have to pass for example (other than the training examples and labels) the batch size, the epochs and the class weights. After the model has been built and trained, it can be evaluated on the test set using Keras' method *evaluate*.

3.3.2 PyTorch

PyTorch is a Python machine learning library, built to create learning algorithms based on graph models. It starts at low-level, but it has a lot of high-level APIs that allow to create even complex neural networks with a very few lines of code. Its core element are Pytorch tensors, in the same way arrays are for Numpy, but built to take advantage of GPU computing.

The transition between low-level programming and APIs is truly continuous, which is a very important feature when it comes to build and tune a deep neural network. In depth, PyTorch has some useful built-in methods to handle training and evaluation, for example *autograd* to compute gradients and *DataLoader* to handle training and batch loading.

3.4 Neural network architecture, losses and optimizers

3.4.1 Architecture

Our model is a fully connected linear deep neural network, composed by:

- an input layer of the size of the training set features;
- 1_{st} hidden layer with 50 nodes;
- 2_{nd} hidden layer with 25 nodes;
- 3_{rd} hidden layer with 10 nodes;
- an output layer with 5 nodes.

Between each layer has been applied a dropout of 20% (during the weight update in the training step each neuron can be ignored with a probability of 0.2) and a ReLU activation function. Moreover, the net has been trained for 300 epochs, with a batch size of 16.

3.4.2 Losses

Here we list the losses used during our work:

BCEWithLogitsLoss: This loss combines in one single class both a Sigmoid layer and the BCELoss. This version is more numerically stable than using a plain Sigmoid followed by a BCELoss as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability. The equation of the loss is the following:

$$loss(x, y) = \mathbb{E}\left[\{l_1, ..., l_N\}^\top\right], \tag{1}$$

where

$$l_n = -w_n \left[y_n \log[\sigma(x_n) + (1 - y_n) \log(1 - \sigma(x_n))\right] \tag{2}$$

CrossEntropyLoss: This criterion combines a $LogSoftmax$ and $NLLLoss$ in one single class. This criterion expects as the target of each value of the minibatch-sized 1D tensor a class index in the range [0,C-1]. The equation of the loss is the following:

$$loss(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) \tag{3}$$

3.4.3 Optimizers

Optimization algorithms are used to minimise the loss function, in order to find optimum values for our learnable model's parameter. During our neural network design and implementation we focused particularly on two main optimization techniques: Gradient Descent and Adam.

Gradient Descent, one of the most famous and used optimization algorithms, works as follows: on each iteration, the algorithm computes the gradient of the loss function for the current weights values and updates them by an arbitrary percent (that we are able to choose by a parameter called learning rate); this procedure goes on util a stopping criterion is satisfied. The traditional algorithm needs to see the whole dataset in order to perform a parameter update, and this can lead to very slow computational times, so from this original idea it has been developed the Stochastic Gradient Descent, that updates the parameter for each training example, but since this approach can lead to fluctuations and difficulties in the convergence, an additional improvement named Mini Batch Gradient Descent was created, which solves a lot of issues related to GD and SGD performing a parameter update for every batch (a subset of training examples that we can decide the size of). For our model we implemented Mini Batch Gradient Descent, that takes as parameters the learning rate, which is variable through the decay rate, the Momentum (which accelerates the movement towards relevant directions) and the Nesterov (which makes the SGD perform the updates based on the previous momentum, in order to try to prevent to jump over minima due to the extreme acceleration of the descent).
The formula is the following:

For every mini batch of size n, repeat until convergence:

$$\nu_t \leftarrow \gamma * \nu_{t-1} - \alpha \nabla_w L(w - \gamma \nu_{t-1}),$$
$$w \leftarrow w - \nu_t,$$

where ν_t is the Momentum update calculated at time t, γ is the Momentum term, η is the learning rate and ∇_w is the gradient of L.

For our model, we chose a learning rate of 0.01, a decay rate of 0 and a Momentum term of 0.9.

Although we obtained very good result using this version of the SGD, we also decided to use another optimizer to compare the results. We chose to use **Adam**, a technique derived both form the idea of Momentum and another optimization algorithm called RMSProp.
While RMSProp tries to dampen oscillations by updating each parameter separately and choosing for each one a different learning rate computed using the exponential average of squares of gradients, Adam uses in addition the exponential average of gradients, keeping also two constants named β_1 and β_2 to control the decay rates of these two moving averages.
The formula is the following:

For each parameter w^j:

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$
$$\Delta w_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$w_{t+1} = w_t + \Delta w_t,$$

where η is the initial learning rate, g_t is the gradient at time t along w^j, ν_t is the exponential average of gradients along w_j, s_t is the exponential average of squares of gradients along w_j and β_1, β_2 are hyperparameters.

For our model, we chose an initial learning rate of 0.01, $beta_1$=0.9 and $beta_2$=0.999 (as suggested from the Adam paper) and an epsilon value (to prevent divisions by zero) of 10^{-8}.

4 Parameter optimization and Validation

Once having built a model, a developer must ensure that the model has a good generalising capacity. Often, a too fitted net fails to predict in test set: this behaviour is called overfitting. In the sections below, we will review some of the techniques used to avoid this problem and some characteristics of the models we considered.

4.1 Cross-Validation

One may desire to have a technique to evaluate the generalisation performances of the model before testing it. An idea could be to split again the training set, and using a part of it to test the performances: this procedure is called Cross-Validation. It works as follows: it randomly splits the training set into k distinct subsets called folds, then it trains and evaluates the model k times, each time picking a different fold as the test set and using the other $k-1$ for training. This procedure is automatically implemented in a lot of libraries inside both the methods used to train the classifiers as well as in the evaluation ones; for example, in Keras it is sufficient to enable the shuffling of data and specify the percentage of the training set that we want to use as validation in order to implement this technique. In our work we also utilised the $StratifiedKFold$ method form the *model_selection* library, which has the property to preserve the percentage of samples for each class present in the training set.

4.2 Grid Search and Random Search

During the process of tuning hyperparameters for each algorithm, one can see that some of them influence overfitting more than others: selecting the right parameters is a tedious job if one does it by hand, and it can become almost impossible as the number of parameters increases. Instead, it is possible to use for example Scikit-learn's GridSearchCV, that will calculate all the possible combinations of the hyperparameters given in input and return the best of all evaluated through cross-validation. The grid search approach is fine when we are exploring relatively few combinations, but when the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead. This class is similar to the one we have just discussed, but it evaluates random values (taken from a probability distribution) for every parameter

passed. It is useful when we want to explore larger parts of the hyperparameter search space and also lighten the computation. Both functions were used in our work, giving similar results in terms of performances.

4.3 Architecture settings

In general, when we build a model, we must determine which parameters mostly influence the performances and try to tune them in a way that we obtain the best predictions. By using the techniques considered above, we obtained some optimal parameters for every model we designed.

4.3.1 Decision Tree

- Criterion for the splitting = gini
- Maximum depth for a tree = 5
- Maximum number of leaf nodes =7

4.3.2 Random Forest

- Criterion for the splitting = entropy.
  (Some of them are inherited from Trees)
- Number of trees in the forest = 60
- Maximum number of leaf nodes = 50
- Bootstrap samples are used when building trees.
  (Otherwise, the whole datset is used to build each tree).

4.3.3 Support Vector Machine

- C = 10;
- Kernel= linear

5 Models Evaluation

5.1 Feature Selection and Dimensionality Reduction

Our dataset is really interesting because while it is formed by a low number of patients (800) it has a really high number of features (more than 20000). To improve computation times, especially for training times, we tried out some dimensionality reduction techniques. Dimensionality reduction assumes an intrinsic dimension of the data significantly lower than the data space and tries to delete all the redundant or less useful information in order to obtain a vector with a lower number of components that maintain a considerably amount of information. In our work we used the Principal Component Analysis (PCA in short) and the Permutation Importance.
The idea of the **Principal Component Analysis** algorithm is to find among all the possible directions in the space of data the one that maximise the variance once that the data are projected in that direction. We implemented PCA using Scikit-learn PCA method from the decomposition library, setting the number of components to 700: the resulting features still retained a high percentage of information (98%), but with a remarkable compression rate that reduced our data to slightly less than 4%.
Another procedure was to adopt the *eli*5 library for computing **Permutation Importance**, that works as follows: one at a time a feature is removed from the test part of the dataset and replaced it with random noise; then an evaluation score (depending on the model we developed) is computed. This method works if noise is drawn from the same distribution as the original feature values (as otherwise the estimator may fail) and the simplest way to get such noise is to shuffle values for a feature (i.e. use other examples' feature values).
We applied the permutation importance method to our tree classifier and got the results shown in Figure 2.

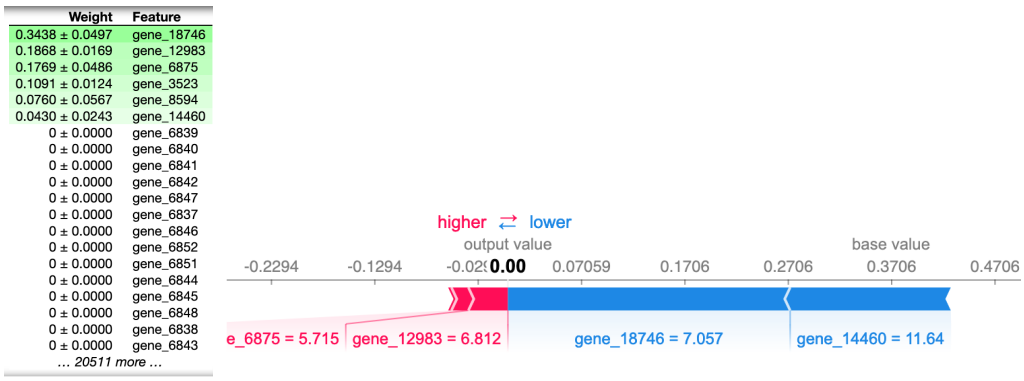
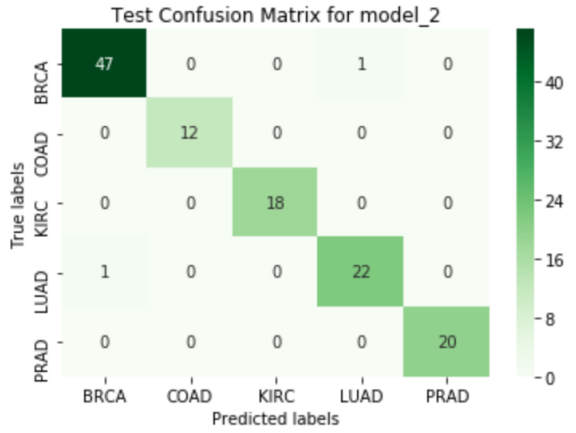


Figure 2: On the left: permutation importance for the parameter tuned tree; it clearly shows that only a low number of genes is correlated with the prediction. On the right: a visual description using Shape library of how these features influence the data prediction.

From these results we assumed that only highlighted genes were responsible for the predictions, so we tried to train a Decision Tree Classifier keeping only those, reducing the features dimensionality from 20530 to 6.
The results are the shown in Figure 3.




```
Accuracy for best Tree Classifier Model: 0.9834710743801653
Hamming Loss for best Tree Classifier Model: 0.01652892561983471
```

Figure 3: Confusion matrix and evaluation metrics for the tree classifier trained with only relevant genes.

5.2 Metrics

During our analysis we used a variety of losses and metrics:

- Accuracy: in multi-class problems, this is a discrete measure of predictions, in the sense that results scores only when predition is exact.
- Hamming Loss: the fraction of labels that are incorrectly predicted.

Talking about these metrics, we believe that they are not a truly good measures for the evaluation of the performances of models in the case of unbalanced classes. A more precise measure, and the metric we used the most during our work, is the Confusion Matrix. Having the true labels as rows and predictions as columns, the confusion matrix presents the number of right classifications on the principal diagonal and the sum of misclassifications off-diagonal. In the following section we will present our results.

5.2.1 Models results

- For the optimal Decision Tree Classifier, we have an Accuracy of 0.99174 and a Hamming Loss of 0.00826. The model was trained in 3.73 seconds. The resulting confusion matrix is shown in Figure 4.
- For the optimal Random Forest Classifier, we have and Accuracy of 1.0 and a Hamming Loss of 0.0. The model was trained in 2.37 seconds. The resulting confusion matrix is shown in Figure 4.
- For the optimal Support Vector Machine, we have an Accuracy of 0.99705. The model was trained in 0.1 seconds.
- For the optimal Keras Deep Neural Network with One-Hot Encoding and CrossEntropyLoss, the model was trained in 25.52 seconds, ending with a loss of the test set of $5,70688e-06$ and an Accuracy on the test set of 1.0.
- For the optimal PyTorch Deep Neural Network with Label Encoding and CrossEntropyLoss, the model was trained in 21.72 seconds ending with a loss of $6,407e-07$. The resulting confusion matrix is shown in Figure 5.

- For the optimal Pytorch Deep Neural Network with One-Hot Encoding and BCEWithLogitsLoss, the model was trained in 21.21 seconds ending with a loss of $0,00543$. The resulting confusion matrix is shown in Figure 5.

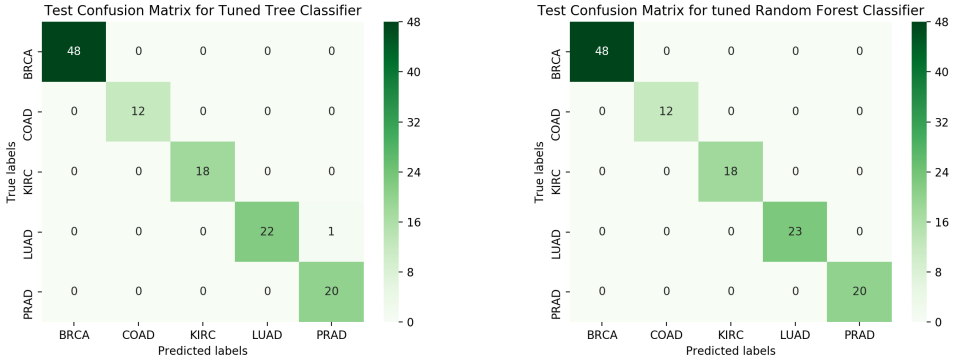


Figure 4: On the left: Confusion test matrix for the tuned decision tree. On the right: Confusion test matrix for the tuned random forest.

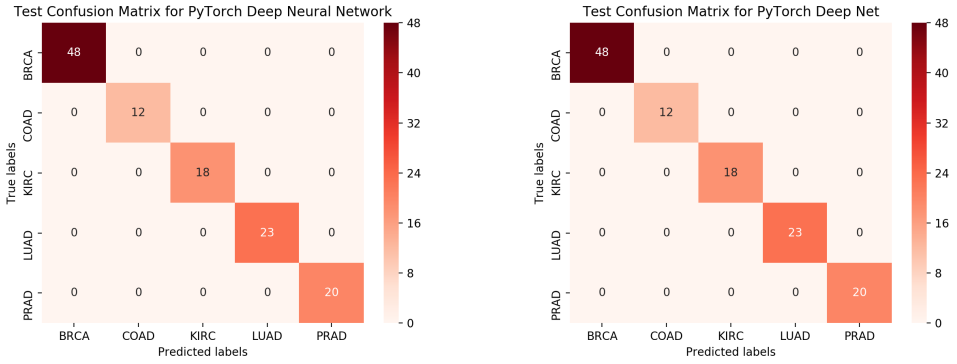


Figure 5: On the left: Confusion test matrix for the Pytorch Deep Neural Network with Label Encoding. On the right: Confusion test matrix for the Pytorch Deep Neural Network with One-Hot Encoding.

6 Conclusions and Outlook

References

[1] Jian Wang, Sana Sahengbieke, Xiaoping Xu, Lei Zhang, Xiaoming Xu, Lifeng Sun, Qun Deng, Da Wang, Dong Chen, Yuan Pan, et al. gene expression analyses identify a relationship between stanniocalcin 2 and the malignant behavior of colorectal cancer. *OncoTargets and therapy*, 11:7155, 2018.

[2] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.