**Week 15 lab2: Logistic Regression and k-fold Validation**

This lab: the aim is to build and evaluate linear classification models, and to understand and implement a k-fold cross validation mechanism to tune the model hyperparameters.

During this lab it is highly recommended to check scikit-learn's documentation on Logistic Regression **(**[**https://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.LogisticRegression.html**](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)**)** classifiers.

**Logistic Regression (LR)**

**Loading Data**

1. Use pandas.read\_csv to load the *titanic.csv* as a pandas dataframe.  
     
   1a. From this dataframe extract a binary vector, *y*, that contains labels/outcomes indicating whether a passenger survived or not.  
     
   1b. Create a feature matrix, *X*, which contains the following columns as features: Pclass, Sex, Age, Siblings\_SpousesAboard, Parents\_ChildrenAboard, and Fare.  
     
   Note: You will need to convert the *Sex* variable (currently a string) into a numerical variable, in order to include it in the training data.

**Train models**

1. Use scikit-learn to instantiate and subsequently train (using .fit(X, y)):

A Logistic Regression classifier

**Note:** don't forget to split your data into train/test e.g. using sklearn's train\_test\_split !

1. Use the .predict attribute of the trained models to classify the train and test data.

3a. Import sklearn.metricsfunction confusion\_matrix to compute the confusion matrix of your classifier evaluated on the train and test data.

3b. Use information from the confusion matrix to compute and print the classification accuracy evaluated on the train and test data for these classifiers.

**Evaluating classification performance**

**(recap) What is a Confusion Matrix?** A confusion matrix is a table that summarizes the performance of a classification model by showing the actual versus predicted classifications. It is primarily used in binary classification (two possible outcomes), but it can also be extended to multi-class problems. For a binary classification problem with Positive (e.g. survived) vs. Negative (e.g. not-survived) labels, the confusion matrix looks like this:

|  | **Predicted Positive** | **Predicted Negative** |
| --- | --- | --- |
| **Actual Positive** | True Positive (TP) | False Negative (FN) |
| **Actual Negative** | False Positive (FP) | True Negative (TN) |
|  |  |  |

* **True Positive**: The model correctly predicts a positive class.
* **True Negative**: The model correctly predicts a negative class.
* **False Positive**: The model incorrectly predicts a positive class.
* **False Negative**: The model incorrectly predicts a negative class.

Refer to slide on Decision Trees for how to calculate ACC, Precision Recall, and Specificity.

**Hyperparameter tuning**

A classification model (linear or non-linear) usually has hyperparameters that control its expressivity or complexity.

* too complex models: highly expressive, but can lead to overfitting.
* too simple models: may result in underfitting if they cannot capture the complexity of the data.

Symptoms of over- and underfitting can be oberved from the validation curves (we will practice this below) similar to a previous lesson on polynomial fitting example:

* **Overfitting**: you obtain good performance (high classification accuracy) on the training data, but poor performance on the unseen test data i.e., poor generalisation.
* **Underfitting**: poor performance on both train and test data.

Tuning the hyperparams to obtain a balance between overfitting and underfitting is therefore critical for good generalisation. Different models may have different hyperparams, for example, the regularisation type and the regularisation strength are among the hyperparams of Logistic Regression models. Model hyperparameters need to be tuned by cross-validation.

**Cross-Validation**

Cross-validation is a technique used to assess how well the model generalizes to unseen data by splitting the available training dataset into multiple subsets for training and validation. Below we outline two approaches to cross-validation (for more details see e.g. <https://scikit-learn.org/stable/modules/cross_validation.html>):

* **Held-out CV** (simple approach):
  + Randomly split the training data into a training subset and a validation subset.
  + Train the model on the training data and evaluate it on the validation set.
  + Repeat for different hyperparameter values (i.e. a grid-search) and choose the one that performs best on the validation set.
  + **pros/cons**: simple to implement, quicker to run, but it only uses part (not all) of the training data for evaluation, which may not be desirable.
* **K-Fold CV** (comprehensive approach):
  + Split the training data into K equal partitions (folds).
  + For each fold:
    - Train the model on K-1 partitions as training subset.
    - Validate the model on the remaining fold as validation subset.
  + Repeat this process for each fold.
  + The performance metric (e.g., accuracy) is averaged across all K folds.
  + **pros/cons**: efficient use of the entire training dataset for both training and validation, can be more robust than the held-out CV, but can be computationally expensive as the model is trained and validated K times.

**Let's practice**

scikit-learn's *Logistic Regression* command gives you options to set regularisation type and strength for the LR classifiers. Please read scikit-learn's documentation to familiarise yourself with these options (links in first cell of this notebook).

For *Logistic Regression*, the default regularisation implements a penalty on the L2 norm of the model's coefficients during training to avoid them becoming too large (overfitting). The strength of the regularisation is inversely proportional to a positive scalar "C" i.e., the classifier hyperparameter.

In this part we will learn how to implement a "K-fold cross validation" mechanism to tune this hyperparameter "C". Let's begin with K=5 fold cross validation.

1. Here we use sklearn's KFold command (<https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html>) to randomly split the train data into K=5 disjoint and (approximately) equally sized partitions: In each fold you will have 4/5 data available for training, and 1/5 data for validation. Also, let's print the shapes of validation and train data in each fold.
2. Select one of the 5 folds. This should give 4/5 of the entire train data as a subset for training your model, and the remaining 1/5 of data held-out as a validation set. On the selected fold, train your classifiers and evaluate the training and validation classification accuracies for different values of "C". Choose "C" values within a logarithmically spaced range i.e., using numpy's logspace command. Finally, plot the corresponding "Validation Curves" i.e., two curves displaying the classification accuracies on the train & validation data against the choices of the regularisation hyperparameter "C".

**Note:** Since choices of "C" have logarithmic scales, you can use matplotlib.pyplot's semilogx function to display the validation curves (semilogx is similar to plot, but the x-axis values will be displayed in a logarithmic (not linear) spacing).

1. Repeat this procedure 5 times, where each time a different fold is used for train and validations. Plot the validation curves averaged across these 5 runs. Observe on which choices of hyperparameter "C" your model will encouter over- and under-fitting.
2. For the LR classifier, find the best hyperparameter "C" with the highest averaged validation accuracy.
3. Finally, use the optimal hyperparameter "C" to train and test the LR classifier and print the corresponding classification test accuracies. NB: For the trained classifier, you can access the corresponding regularisation parameters using model.C (here "model" is the output of, for example, svm.SVC(), it can be named arbitrarily).
4. You have now tuned your models using K=5 fold cross validation scheme and a grid search! The number K could vary in different applications. The K-fold cross validation approach enables to train and evaluate your model over all your available training data, hence is more efficient (but also more time-consuming) compared to the simple held-out approach which only keeps one partition for validation and the other for training.

**Note:** scikit-learn's machine learning tools give you some built-in options for hyperparameter tuning and cross-validation, without needing you to implement it by yourself. One of the tools is the sklearn's GridSearchCV command (<https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html>). Input to GridSearchCV include a dictionary with your choice of hyperparameters as keys, and the lists of hyperparameter settings to try as values. For the rest of the lab you may use these available options.

Use scikit-learn's GridSearchCV command to tune a Logistic Regression classifer for data. Let's jointly tune (i.e., grid search) the two hyperparams of this classifier, namely "penalty" and "C". "penalty" determines the type of regularisation norm used, which you will chose from either L2 (ridge) vs L1 (lasso). "C" inversely controls the regularisation stregth for which, as above, you will define a list of values to search. Compute and print the train and test accuracies for the tuned model.

**Tasks to complete for this worksheet:**

**Task 1**: have a look at the list of features above, and extract the bmi feature into a variable called 'bmi'.

It is always good to first visualise data. Given that it is numeric data, we can use the scatter visualisation within matplotlib for feature bmi versus target y.

OK, we can see somewhat of a trend here. Patients with a large value of bmi tend to have high level of diabetes progression. What about another feature? Let's look at the blood sugar level (glu).

**Task 2**: have a look at the list of features above, along with their index, and extract the blood sugar level into a variable. Once you do this, plot it as we have done for the bmi feature.

Before performing linear regression, we need to split train and test sets, with 80% of all patient data as training and the rest as test. Splitting train and test sets is a common practice in machine learning to ensure your model is trained and evaluated properly. Here, we call the function \*train\_test\_split\* in scikit-learn that splits arrays or matrices into train and test subsets. \*random\_state\* controls the shuffling applied to the data before applying the split for reproducible output across multiple function calls. Here, we set it to 5.

**Linear Regression:**

Now let's do some linear regression:

By looking at the plots, we can see that trend for the blood sugar level appears less clear. However, let's apply linear regression to each of these features, and see which is a better predictor of diabetes progression.

First, we want to import the LinearRegression model from scikit-learn that calculates gradient descent behind the scene. The scikit-learn documentation is pretty comprehensive, so you should try and get familiar with it: https://scikit-learn.org/0.22/modules/generated/sklearn.linear\_model.LinearRegression.html

*Importing the model*

Training a Linear Regression model:

The first step in training a model with scikit-learn is to import the '*instantiate*' of the model, that is, create an object that represents an instance of the model which you can use for training, predicting etc.

Instantiate the model

Once you have created your model, you can call the '*fit'* function on it. The fit function is available for all relevant models in scikit-learn, and what it does is to train your model on the supplied data. For a supervised model, you have to supply two arguments, the features and the labels associated with the features.

**Task 3**: In this case, we want to train our model on the *bmi* feature on the training set to predict diabetes progression. Use the fit function, by looking at the documentation, to fit your model with the *bmi* variable and the ground truth y on training samples.

**Making predictions**

Once you have fit the model to the data, you can use this model to make predictions. In this case, we will use our model to predict data that it was trained on. The purpose of doing this is to explicitly compare the *ground truth values* versus the *predicted values*. For example you can use the bmi value of -0.05 (or any value from the dataset) as input for the mode to make a prediction. Print out the prediction for you to have a look.

*Your code for prediction and print the outcome.*

**Task 4**: Use the \**model*\* to predict the diabetes progression given the corresponding bmi value for that patient in the training set. Print out both the predicted y-values and the ground truth y-values for the first 5 patients.

Plot the train\_dataset and the model (in different colours) to visualize how well your model fit the training data.

**Task 5**: Using the relevant code in the cells above, make predictions based on \*glu\* features on the training samples and produce a plot for it same as in **Task 4**.

**Gradient Descent**

**Task 6:** Next, we want to use gradient descent (GD) to find the 'best line' that fits all the features. GD minimizes the MSE loss function by iteratively updating the parameters (weights) of the model. (You can try both with\_bias and without\_bias versions).

Note: you will need to implement the *MSE* loss function.

Then compute the coefficients *W* and *MSE* values using gradient descent for defining a linear regression function. Call the defined linear regression function on the training samples to obtain its best coefficients and its loss.

Finally, plot the loss curve to observe the optimization process.

**Quantifying how good the model is**

Looking at visualisations is helpful, but often we want to quantify how well a model has fit our data. We can do that by calculating the mean squared error of our model – you can call it directly from SKlearn.

**Task 7**: Look at the documentation for the mean\_squared\_error function (https://scikit-learn.org/0.22/modules/generated/sklearn.metrics.mean\_squared\_error.html), and apply it below to calculate the mean squared error of your predictions.

**Task 8**: Produce the predictions for the bmi feature on the test set and report its MSE.

**Task 9**: What is the performance if you use \*all\* of the features?