APPLIED MACHINE LEARNING

LAB ACTIVITIES (LAB 5) (WITH SOLUTIONS)

AUTOMATING THE PROCESS AND SAVING THE MODEL

This workbook is designed to guide you through the activities proposed for today's lab. As you will be working independently, feel free to proceed through the text at your own pace, spending more time on the parts that are less familiar to you. The workbook contains both hands-on tasks and links to learning materials such as tutorials, articles and videos. When you are unsure about something, feel free to ask our teaching assistants or use Internet resources to look for a solution. At the end of each section, there will be questions and exercises to verify your understanding of the presented information. You may need to do some research to answer the questions.

1. Pipelining

There are standard workflows in applied machine learning. Standard because they overcome common problems like data leakage in your test harness. Python scikit-learn provides a Pipeline utility to help automate machine learning workflows. Pipelines work by allowing for a linear sequence of data transforms to be chained together culminating in a modelling process that can be evaluated.

The goal is to ensure that all of the steps in the pipeline are constrained to the data available for the evaluation, such as the training dataset or each fold of the cross-validation procedure. You can learn more about *Pipelines* in scikit-learn by reading the Pipeline section of the user guide. See below.

https://scikit-learn.org/stable/modules/compose.html

You can also review the API documentation for the *Pipeline* and *FeatureUnion* classes and the pipeline module.

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.pipeline

An easy trap to fall into in applied machine learning is leaking data from your training dataset to your test dataset. To avoid this trap you need a robust test harness with strong separation of training and testing. This includes data preparation. Data preparation is one easy way to leak knowledge of the whole training dataset to the algorithm. For example, preparing your data using normalisation or standardisation on the entire training dataset before learning would not be a valid test because the training dataset would have been influenced by the scale of the data in the test set.

Pipelines help you prevent data leakage in your test harness by ensuring that data preparation like standardisation is constrained to each fold of your cross-validation procedure. The example below demonstrates this important data preparation and model evaluation workflow on the Pima Indians onset of diabetes dataset. The pipeline is defined with two steps:

1. Standardise the data.
2. Learn a Linear Discriminant Analysis model.

The pipeline is then evaluated using 10-fold cross-validation.

```
# Create a pipeline that standardises the data then creates a model
from pandas import read_csv
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

```
# load data
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]

# create pipeline
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('lda', LinearDiscriminantAnalysis()))
model = Pipeline(estimators)

# evaluate pipeline
kfold = KFold(n_splits=10, random_state=7)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Verify your understanding:

(a) *Analyse the code block and see how we create a Python list of steps that are provided to the Pipeline for process the data and how the Pipeline itself is treated like an estimator and is evaluated in its entirety by the k-fold cross-validation procedure.*

(b) *Run the code block above to see a summary of accuracy of the setup on the dataset.*

Feature extraction is another procedure that is susceptible to data leakage. Like data preparation, feature extraction procedures must be restricted to the data in your training dataset. The pipeline provides a handy tool called the *FeatureUnion* which allows the results of multiple feature selection and extraction procedures to be combined into a larger dataset on which a model can be trained. Importantly, all the feature extraction and the feature union occurs within each fold of the cross-validation procedure. The example below demonstrates the pipeline defined with four steps:

1. Feature Extraction with Principal Component Analysis (3 features).
2. Feature Extraction with Statistical Selection (6 features).
3. Feature Union.
4. Learn a Logistic Regression Model.

The pipeline is then evaluated using 10-fold cross-validation.

```
# Create a pipeline that extracts features from the data then creates a model
from pandas import read_csv
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest

# load data
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]

# create feature union
features = []
features.append(('pca', PCA(n_components=3)))
features.append(('select_best', SelectKBest(k=6)))
feature_union = FeatureUnion(features)

# create pipeline
estimators = []
estimators.append(('feature_union', feature_union))
estimators.append(('logistic', LogisticRegression()))
model = Pipeline(estimators)

# evaluate pipeline
kfold = KFold(n_splits=10, random_state=7)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Verify your understanding:

(c) *Analyse the code block to see how the FeatureUnion is its own Pipeline that in turn is a single step in the final Pipeline used to feed Logistic Regression. This might get you thinking about how you can start embedding pipelines within pipelines.*

(d) *Run the above code block to see a summary of accuracy of the setup on the dataset.*

(e) Compare the result with its previous step.

```
Soln:
0.773462064252 Vs 0.776042378674
```

2. Ensembles

The three most popular methods for combining the predictions from different models are:

- **Bagging.** Building multiple models (typically of the same type) from different subsamples of the training dataset.
- **Boosting.** Building multiple models (typically of the same type) each of which learns to fix the prediction errors of a prior model in the sequence of models.
- **Voting.** Building multiple models (typically of differing types) and simple statistics (like calculating the mean) are used to combine predictions.

This assumes you are generally familiar with machine learning algorithms and ensemble methods and will not go into the details of how the algorithms work or their parameters. The Pima Indians onset of Diabetes dataset is used to demonstrate each algorithm. Each ensemble algorithm is demonstrated using 10-fold cross-validation and the classification accuracy performance metric.

Bagging Algorithms

Bootstrap Aggregation (or Bagging) involves taking multiple samples from your training dataset (with replacement) and training a model for each sample. The final output prediction is averaged across the predictions of all of the sub-models. The two bagging models covered in this section are as follows:

- Bagged Decision Trees.
- Random Forest.

Bagging performs best with algorithms that have high variance. A popular example are decision trees, often constructed without pruning. In the example below is an example of using the `BaggingClassifier` with the Classification and Regression Trees algorithm (*DecisionTreeClassifier*). A total of 100 trees are created. See below.

Bagging
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html

DecisionTree
https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

```
# Bagged Decision Trees for Classification
from pandas import read_csv
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
seed = 7
kfold = KFold(n_splits=10, random_state=seed)
cart = DecisionTreeClassifier()
num_trees = 100
model = BaggingClassifier(base_estimator=cart, n_estimators=num_trees, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Random Forests is an extension of bagged decision trees. Samples of the training dataset are taken with replacement, but the trees are constructed in a way that reduces the correlation between individual classifiers. Specifically, rather than greedily choosing the best split point in the construction of each tree, only a random subset of features are considered for each split. You can construct a Random Forest model for classification using the $RandomForestClassifier$ class. The example below demonstrates using Random Forest for classification with 100 trees and split points chosen from a random selection of 3 features. See below.

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Verify your understanding:

(f) Referring the information on the above link, modify the above code to build a Random Forest Classifier with 100 trees and 3 max features.

(g) What is the accuracy?

```
Soln:
# Random Forest Classification
from pandas import read_csv
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_trees = 100
max_features = 3
kfold = KFold(n_splits=10, random_state=7)
model = RandomForestClassifier(n_estimators=num_trees,
max_features=max_features)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Boosting Algorithms

AdaBoost was perhaps the first successful boosting ensemble algorithm. It generally works by weighting instances in the dataset by how easy or difficult they are to classify, allowing the algorithm to pay less attention to them in the construction of subsequent models. You can construct an AdaBoost model for classification using the $AdaBoostClassifier$ class. The example below demonstrates the construction of 30 decision trees in sequence using the AdaBoost algorithm. See below.

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

Verify your understanding:

(h) Referring the information on the above link, modify the above code to build a AdaBoost classifier with 30 trees and a seed of 7.

(i) What is the accuracy?

```
Soln:
# AdaBoost Classification
from pandas import read_csv
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import AdaBoostClassifier
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_trees = 30
seed=7
kfold = KFold(n_splits=10, random_state=seed)
model = AdaBoostClassifier(n_estimators=num_trees, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Voting Ensemble

Voting is one of the simplest ways of combining the predictions from multiple machine learning algorithms. It works by first creating two or more standalone models from your training dataset. A Voting Classifier can then be used to wrap your models and average the predictions of the sub-models when asked to make predictions for new data. The predictions of the sub-models can be weighted, but specifying the weights for classifiers manually or even heuristically is difficult. More advanced methods can learn how to best weight the predictions from sub-models, but this is called stacking (stacked aggregation) and is currently not provided in scikit-learn.

You can create a voting ensemble model for classification using the $VotingClassifier$ class. The code below provides an example of combining the predictions of logistic regression, classification and regression trees and support vector machines together for a classification problem. See below.

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

Verify your understanding:

(j) Referring the information on the above link, build a Voting classifier with LogisticRegression(), DecisionTreeClassifier() and SVC().

(k) *Run your code block and compare the results.*

(l) What are the ensemble algorithms perhaps worthy of further study on this problem? Explain why.

```
Soln:
# Voting Ensemble for Classification
from pandas import read_csv
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
kfold = KFold(n_splits=10, random_state=7)
# create the sub models
estimators = []
model1 = LogisticRegression()
estimators.append(('logistic', model1))
model2 = DecisionTreeClassifier()
estimators.append(('cart', model2))
model3 = SVC()
estimators.append(('svm', model3))
# create the ensemble model
ensemble = VotingClassifier(estimators)
results = cross_val_score(ensemble, X, Y, cv=kfold)
print(results.mean())
```

3. Tuning

Algorithm tuning is a final step in the process of applied machine learning before finalising your model. It is sometimes called hyperparameter optimisation where the algorithm parameters are referred to as hyperparameters, whereas the coefficients found by the machine learning algorithm itself are referred to as parameters. Optimisation suggests the search-nature of the problem. Phrased as a search problem, you can use different search strategies to find a good and robust parameter or set of parameters for an algorithm on a given problem. Python scikit-learn provides two simple methods for algorithm parameter tuning:

1. Grid Search Parameter Tuning.
2. Random Search Parameter Tuning.

Grid search is an approach to parameter tuning that will methodically build and evaluate a model for each combination of algorithm parameters specified in a grid. You can perform a grid search using the *GridSearchCV* class. See below.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

The example below evaluates different alpha values for the Ridge Regression algorithm on the standard diabetes dataset. This is a one-dimensional grid search.

```
# Grid Search for Algorithm Tuning
import numpy
from pandas import read_csv
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
alphas = numpy.array([1,0.1,0.01,0.001,0.0001,0])
param_grid = dict(alpha=alphas)
model = Ridge()
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid.fit(X, Y)
print(grid.best_score_)
print(grid.best_estimator_.alpha)
```

Verify your understanding:

(m) *Run the above code block and interpret the results.*

(n) What is the recommended alpha value?

```
Soln:
0.279617559313
1.0
```

Random search is an approach to parameter tuning that will sample algorithm parameters from a random distribution (i.e. uniform) for a fixed number of iterations. A model is constructed and evaluated for each combination of parameters chosen. You can perform a random search for algorithm parameters using the *RandomizedSearchCV* class. See below.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

The example below evaluates different random alpha values between 0 and 1 for the Ridge Regression algorithm on the standard diabetes dataset. A total of 100 iterations are performed with uniformly random alpha values selected in the range between 0 and 1 (the range that alpha values can take).

```python
# Randomized for Algorithm Tuning
from pandas import read_csv
from scipy.stats import uniform
from sklearn.linear_model import Ridge
from sklearn.model_selection import RandomizedSearchCV
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
param_grid = {'alpha': uniform()}
model = Ridge()
rsearch = RandomizedSearchCV(estimator=model, param_distributions=param_grid, n_iter=100,
    random_state=7)
rsearch.fit(X, Y)
print(rsearch.best_score_)
print(rsearch.best_estimator_.alpha)
```

Verify your understanding:

(o) *Run the above code block and interpret the results.*

(p) What is the recommended alpha value? Compare with the value from Grid search.

```
Soln:
0.279617354112
0.989527376274
```

4. Saving and Loading Models

Pickle is the standard way of serialising objects in Python. You can use the *pickle* operation to serialise your machine learning algorithms and save the serialised format to a file. See below.

https://docs.python.org/2/library/pickle.html

Later you can load this file to deserialise your model and use it to make new predictions. The example below demonstrates how you can train a logistic regression model on the Pima Indians onset of diabetes dataset, save the model to file and load it to make predictions on the unseen test set.

```
# Save Model Using Pickle
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from pickle import dump
from pickle import load
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.33, random_state=7)

# Fit the model on 33%
model = LogisticRegression()
model.fit(X_train, Y_train)

# save the model to disk
filename = 'finalized_model.sav'
dump(model, open(filename, 'wb'))

# some time later...

# load the model from disk
loaded_model = load(open(filename, 'rb'))
result = loaded_model.score(X_test, Y_test) print(result)
```

Verify your understanding:

(q) *Run the above code block to save the model to finalized model.sav in your local working directory. Load the saved model and evaluating it provides an estimate of accuracy of the model on unseen data.*

The *Joblib* library is part of the SciPy ecosystem and provides utilities for pipelining Python jobs. See below.

https://pypi.org/project/joblib/

It provides utilities for saving and loading Python objects that make use of NumPy data structures, efficiently. This can be useful for some machine learning algorithms that require a lot of parameters or store the entire dataset (e.g. k-Nearest Neighbors). The example below demonstrates how you can train a logistic regression model on the Pima Indians onset of diabetes dataset, save the model to file using Joblib and load it to make predictions on the unseen test set.

```
# Save Model Using joblib
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.externals.joblib import dump
from sklearn.externals.joblib import load
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(filename, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.33, random_state=7)

# Fit the model on 33%
model = LogisticRegression()
model.fit(X_train, Y_train)

 # save the model to disk
filename = 'finalized_model1.sav'
```

```
dump(model, filename)


# some time later...


# load the model from disk
loaded_model = load(filename)
result = loaded_model.score(X_test, Y_test)
print(result)
```

Verify your understanding:

(r) Run the above code block and save the model to file as finalized model1.sav and also create one file for each NumPy array in the model. After the model is loaded see how an estimate of the accuracy of the model on unseen data is reported.

(s) Below are some important considerations when finalising your machine learning models.

Python Version. *Take note of the Python version. You almost certainly require the same major (and maybe minor) version of Python used to serialise the model when you later load it and deserialise it.*

Library Versions. *The version of all major libraries used in your machine learning project almost certainly need to be the same when deserialising a saved model. This is not limited to the version of NumPy and the version of scikit-learn.*

Manual Serialisation. *You might like to manually output the parameters of your learned model so that you can use them directly in scikit-learn or another platform in the future. Often the techniques used internally by machine learning algorithms to make predictions are a lot simpler than those used to learn the parameters and can be easy to implement in custom code that you have control over.*

Take note of the version so that you can re-create the environment if for some reason you cannot reload your model on another machine or another platform at a later time.

5. References

[1]. Géron, A., 2017. Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc.".

[2]. Raschka, S., 2015. Python machine learning. Packt Publishing Ltd.

[3]. Grus, J., 2019. Data science from scratch: first principles with python. O'Reilly Media.

[4]. Müller, A.C. and Guido, S., 2016. Introduction to machine learning with Python: a guide for data scientists. " O'Reilly Media, Inc.".

[5]. Brownlee, J., 2014. Machine learning mastery.

[6]. Raschka, S., 2015. Python machine learning. Packt Publishing Ltd.

[7]. SAS, 2018, Advanced Predictive Modelling using SAS

[8]. Géron, A., 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media.