Applied Data Science

Session 31_32:
Model Selection -- Model Validation,
Hyper Parameter Tuning.
Pipeline.
Model Persistence

Dr. Soharab Hossain Shaikh

1

Model Selection

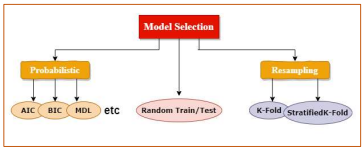
2

Model Selection

- **Model Selection** is the final step of selecting the right model for production. We can apply various algorithms of regression as well as classification to an input data.
- All data scientists may also use the same algorithms.
- Then why some data scientists produce better models with higher accuracy than others?
- The answer lies in **Cross Validation** and **Hyperparameter tuning.**
- This step will separate an expert data scientist from a novice.

3

Model Selection



- **Model Selection** is like choosing either a model with different hyper-parameters or best among different candidate models. Normally, the selection of any model should not rely only on its performance but instead, also on its complexity.
- Usually, we categorize the techniques of model selection as follows:
- Random Train/Test Split
- Resampling Techniques
- Probabilistic Model Selection Techniques

4

Random Train/Test Split

- Data to be passed in model is divided into train and test, ratio wise. This can be achieved using ***train_test_split*** function of scikit-learn python library.
- On re-running of train_test_split data code, results come out to be different on each run of code.
- So, you aren't sure how exactly your model will perform on unseen data. **The uncertainty of model performance in this technique introduces the following techniques.**

```
from sklearn.model_selection import train_test_split

X = df.drop(['target'],axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X , y, test_size=0.2, random_state=0)
```

5

Random Sampling

- In resampling technique of model selection, for a set of iterations, **data is resampled** into train/test followed by training on train and evaluation on test set.
- Model chosen from this technique is **assessed based on performance,** not the model complexity.
- **Performance** is computed **on out-of-sample** data. Resampling techniques estimate the error by evaluating out-of-sample data aka unseen data.
  - Strategies of resampling are such as K-Fold, StratifiedK-Fold etc.

6

Probabilistic Model Selection

- Probabilistic model selection is statistical methods whose quality can be measured by ***Information Criterion*** (IC).
- Its techniques involve **scoring method** that uses a probability framework of **log-likelihood** of Maximum Likelihood Estimation (MLE) to chose the best among candidate models.
- A very useful way of selecting your model considering both performance and complexity, unlikely Resampling techniques.

Factors considered to select model	Resampling	Probabilistic
Model Performance	✓	✓
Model Complexity	✗	✓

7

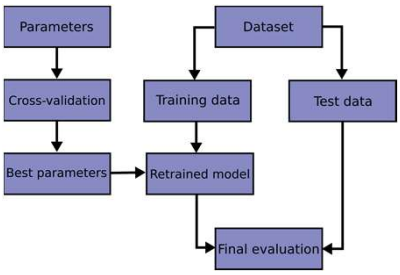
Probabilistic Model Selection

- IC is a measure of statistics which **leads to some score**. Model with the **lowest score** means that **less information is lost** and considered as the **best model**. A single score is useless until you draw some comparison of scores of multiple models.
- Model is chosen by scoring method whose scores are based on:
  - **Performance** on train data is evaluated using **log-likelihood** which comes from the concept of MLE so as to optimize model parameters. It says about how well your model is fitted with your data.
  - **Model Complexity** is evaluated using number of parameters (or degrees of freedom) in model.
- Performance is computed on in-sample data which means test set is not required and the score is computed on whole train data directly.
- Less complexity means a simple model with less parameters, easy to understand, and maintain but it can't catch variations impacting the performance of a model.

- ***More will be discussed shortly……..***

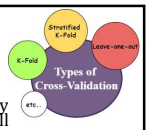
8

Model Validation



9

Cross Validation



- Assumption is that some data is Independent and Identically Distributed (i.i.d.). IID is making the assumption that all samples stem from the same generative process and that the generative process is assumed to have no memory of past generated samples.
- The following cross-validators can be used in such cases (next slides).
- **NOTE:** While i.i.d. data is a common assumption in machine learning theory, it rarely holds in practice. If one knows that the samples have been generated using a time-dependent process, it is safer to use a time-series aware cross-validation scheme. Similarly, if we know that the generative process has a group structure (samples collected from different subjects, experiments, measurement devices), it is safer to use group-wise cross-validation.

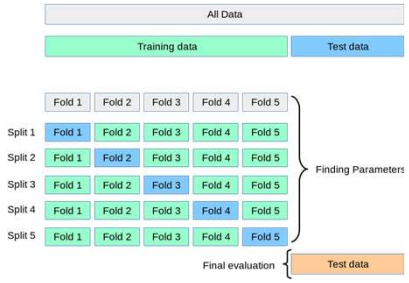
10

Cross Validation

- A resampling technique to make model sure about its efficiency and accuracy on the unseen data.
- In short, Model Validation technique, up for other applications.
- **Bunch of train/test splits — testing accuracy for each split — average them**
- Steps are as follows:
  1: Divide data into K partitions. These partitions will be of equal size.
  2: Treat Fold-1 as test fold while K-1 as train folds.
  3: Compute score of test-fold.
  4: Repeat step 3 for all folds taking another fold as test while remaining as train.
  5: Take average of scores of all the folds.
- It **partitions data** into **train/test** in such a way that the previous set will not repeat and for each set training and testing is performed once.
- **Each partition** (*aka fold*) **is of equal size**. **Can say, every datapoint plays as train and test in its life journey.**
- This is basic functionality of cross-validation which **can be useful while tuning hyper-parameters or selecting ML model** such as logistic or decision trees for classification problem.
- It **prevents over-fitting and under-fitting** by choosing optimal value of K. Also, **accurate model estimate** than *train_test_split* method.

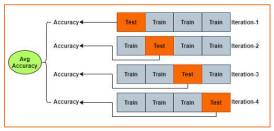
11

K-Fold Cross Validation



12

K-Fold Cross Validation



- In 1st iteration, model is trained on the 4 blocks and tested on 1st block resulted an accuracy.
- In next iteration, another block as test set while remaining as train, resulting into another accuracy.
- This process of dividing and evaluating is done for all 5 folds.
- Average of all sets are computed and thrown as a final accuracy which is treated as a model accuracy.

- **Merits:**
- Overcome the problem of computational power like in LOO to some extent.
- Since not every record is treated as test set like LOOCV hence, hence model may not be affected much if any outlier is present in data. Overcomes problem of variability.

- **Limitations:**
- In any iteration, there is a possibility that test set may have records of just one class. This will make data imbalance and impact our model.

13

Implement Manually

```
X = df.drop(['target'],axis=1)
y = df['target']

model = LogisticRegression()

def return_score(model,X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    score = model.score(X_test, y_test)
    return score

scores = []
#Try StratifiedK-Fold (Stratified5-Fold)
#cv = StratifiedKFold(n_splits=5, random_state=0, shuffle=False)
#Try K-Fold (5-Fold)
cv = KFold(n_splits=5, random_state=seed, shuffle=False)
for train_index, test_index in cv.split(X,y):
    X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2)
    score = return_score(model,X_train, X_test, y_train, y_test)
    scores.append(score)

print("Accuracy score in each iteration: {}".format(scores))
print("K-Fold Score: {}".format(np.mean(scores)))
```

14

Using Sklearn

```
from sklearn.model_selection import train_test_split,KFold,cross_val_score,cross_val_predict,StratifiedKFold,

X = df.drop(['target'],axis=1)
y = df['target']

model = LogisticRegression()

#By default, cv parameters assumes splits for StratifiedK-Fold
cv_scores_5_folds = cross_val_score(model, X, y, cv=5)
#Specify K-Fold in cv parameter
#cv_scores_5_folds = cross_val_score(model, X, y, cv=KFold(n_splits=5))

cv_predicts_5_folds = cross_val_predict(model,X,y,cv=5)

print("Accuracy score in each iteration: {}".format(cv_scores_5_folds))
print("Predicted class for each record: {}".format(cv_predicts_5_folds))
print("K-Fold Score: {}".format(np.mean(cv_scores_5_folds)))
print("Total records: {}, Total predicted values: {}".format(df.shape[0],len(cv_predicts_5_folds)))
```

15

Using K-Fold - Sklearn

KFold divides all the samples in **k** groups of samples, called folds (if **n=k** , this is equivalent to the Leave-One-Out strategy), of equal sizes (if possible).

The prediction function is learned using **k-1** folds, and the fold left out is used for test.

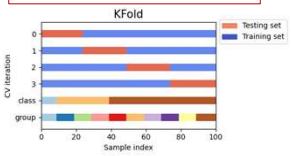
Here is a visualization of the cross-validation behavior.

Note that KFold is not affected by classes or groups.

Example of 2-fold cross-validation on a dataset with 4 samples.

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold

>>> X = ["a", "b", "c", "d"]
>>> kf = KFold(n_splits=2)
>>> for train, test in kf.split(X):
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```



16

Folds

- Each fold is constituted by two arrays: the first one is related to the *training set*, and the second one to the *test set*.
- Thus, one can create the training/test sets using numpy indexing:

```
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> y = np.array([0, 1, 0, 1])
>>> X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
```

17

Repeated K-Fold

- **RepeatedKFold** repeats K-Fold n times. It can be used when one requires to run KFold n times, producing different splits in each repetition.
- Example of 2-fold K-Fold repeated 2 times:

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> random_state = 12883823
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=random_state)
>>> for train, test in rkf.split(X):
...     print("%s %s" % (train, test))
...
[2 3] [0 1]
[0 1] [2 3]
[0 2] [1 3]
[1 3] [0 2]
```

Similarly, **RepeatedStratifiedKFold** repeats Stratified K-Fold n times with different randomization in each repetition.

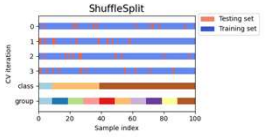
18

Random permutations cross-validation a.k.a. Shuffle & Split

The **ShuffleSplit** iterator will generate a user defined number of independent train / test dataset splits. Samples are first shuffled and then split into a pair of train and test sets.

It is possible to control the randomness for reproducibility of the results by explicitly seeding the *random_state* pseudo random number generator.
Here is a visualization of the cross-validation behavior.
Note that **ShuffleSplit** is not affected by classes or groups.

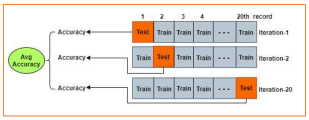
```
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.arange(10)
>>> ss = ShuffleSplit(n_splits=5, test_size=0.25, random_state=0)
>>> for train_index, test_index in ss.split(X):
...     print("%s %s" % (train_index, test_index))
[9 1 6 7 3 0 5] [2 8 4]
[2 9 8 0 6 7 4] [3 5 1]
[4 5 1 0 6 9 7] [2 3 8]
[2 7 5 8 0 3 4] [6 1 9]
[4 1 0 6 8 9 3] [5 2 7]
```



ShuffleSplit is thus a good alternative to KFold cross validation that allows a finer control on the number of iterations and the proportion of samples on each side of the train / test split.

19

LeaveOneOut (LOO) Cross-validation



- In 1st iteration, model is trained the 19 folds and tested on the 1st block (fold) giving out an accuracy.
- In the next iteration, another block having next record is selected as the test set while remaining as train, gives another accuracy.
- LOOCV makes testing and training on all 20 records in 20 iterations, hence **expensive computational power**.
- Average of all sets are then computed and thrown as a final accuracy aka k-fold score.

- **Merits:**
- It gives sort of certainty of model performance when tested on unseen data.

- **Limitations:**
- As compared to *train_test_split*, it takes more computational power .
- As everything is tested and trained, leads to higher variance and testing unseen data in production leads to poor results.
- Results can lead to higher variation as when an outlier datapoint is tested.

20

LeaveOneOut (LOO) Cross-validation

- **LeaveOneOut (or LOO)** is a simple cross-validation. Each learning set is created by taking all the samples except one, the test set being the sample left out.
- Thus, for samples, we have different training sets and different tests set.
- This cross-validation procedure does not waste much data as only one sample is removed from the training set.

```
>>> from sklearn.model_selection import LeaveOneOut

>>> X = [1, 2, 3, 4]
>>> loo = LeaveOneOut()
>>> for train, test in loo.split(X):
...     print("%s %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

21

LeaveOneOut (LOO) Cross-validation

Potential users of LOO for model selection should weigh a few known caveats. When compared with k-fold cross validation, one builds **n** models from n samples instead of **k** models, where **n>k**. Moreover, each is trained on **n−1** samples rather than **(k−1)n/k**.

In both ways, assuming **k** is not too large and **k<n**, LOO is more computationally expensive than k-fold cross validation.

In terms of accuracy, LOO often results in high variance as an estimator for the test error. Intuitively, since **n−1** of the n samples are used to build each model, models constructed from folds are virtually identical to each other and to the model built from the entire training set.

However, if the learning curve is steep for the training size in question, then 5- or 10- fold cross validation can overestimate the generalization error.
As a general rule, most authors, and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to LOO.

22

Leave-P-Out Cross-validation

- Leave-P-Out is very similar to Leave-One-Out as it creates all the possible training/test sets by removing samples from the complete set.
- For **n** samples, this produces ${}^{n}C_{p}$ train-test pairs.
- Unlike Leave-One-Out and KFold, the test sets will overlap for **p>1**.

Example of Leave-2-Out on a dataset with 4 samples:

```
>>> from sklearn.model_selection import LeavePOut

>>> X = np.ones(4)
>>> lpo = LeavePOut(p=2)
>>> for train, test in lpo.split(X):
...     print("%s %s" % (train, test))
[2 3] [0 1]
[1 3] [0 2]
[1 2] [0 3]
[0 3] [1 2]
[0 2] [1 3]
[0 1] [2 3]
```

23

Stratified K-Fold

- Some classification problems can exhibit a large imbalance in the distribution of the target classes: for instance there could be several times more negative samples than positive samples.
- In such cases it is recommended to use stratified sampling as implemented in ***StratifiedKFold*** and ***StratifiedShuffleSplit*** to ensure that relative class frequencies is approximately preserved in each train and validation fold.
- **StratifiedKFold** is a variation of k-fold which returns stratified folds: each set contains approximately the same percentage of samples of each target class as the complete set.

24

Stratified K-Fold

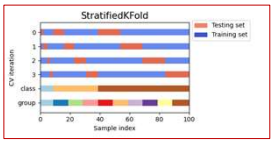
- Here is an example of stratified 3-fold cross-validation on a dataset with 50 samples from two unbalanced classes. We show the number of samples in each class and compare with KFold.

```
>>> from sklearn.model_selection import StratifiedKFold, KFold
>>> import numpy as np
>>> X, y = np.ones((50, 1)), np.hstack(([0] * 45, [1] * 5))
>>> skf = StratifiedKFold(n_splits=3)
>>> for train, test in skf.split(X, y):
...     print('train -  {}   |   test -  {}'.format(
...         np.bincount(y[train]), np.bincount(y[test])))
train -  [30  3]   |   test -  [15  2]
train -  [30  3]   |   test -  [15  2]
train -  [30  4]   |   test -  [15  1]
>>> kf = KFold(n_splits=3)
>>> for train, test in kf.split(X, y):
...     print('train -  {}   |   test -  {}'.format(
...         np.bincount(y[train]), np.bincount(y[test])))
train -  [28  5]   |   test -  [17]
train -  [28  5]   |   test -  [17]
train -  [34]   |   test -  [11  5]
```

25

Stratified K-Fold

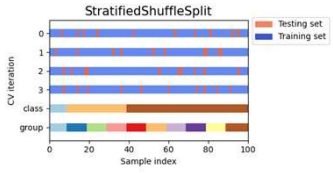
- Here is an example of stratified 3-fold cross-validation on a dataset with 50 samples from two unbalanced classes.
- We show the number of samples in each class and compare with KFold.
- Here is a visualization of the cross-validation behavior.
- **RepeatedStratifiedKFold** can be used to repeat Stratified K-Fold n times with different randomization in each repetition.



26

Stratified Shuffle Split

- **StratifiedShuffleSplit** is a variation of ShuffleSplit, which returns stratified splits, i.e which creates splits by preserving the same percentage for each target class as in the complete set.
- Here is a visualization of the cross-validation behavior.



27

Data Splitting

28

Train, Validation and Test Sets

Train/Test Split

```
from sklearn.model_selection import train_test_split

X = df.drop(['target'],axis=1)

y = df['target']

X_train, X_test, y_train, y_test = train_test_split(X , y, test_size=0.2, random_state=0)
```

- It is a general practice to split our data into three sets
- **Train set**
  - The data that we use to train the model
- **Validation set**
  - The data that we use to 'validate' a model
  - Any hyper-parameter tuning that is done, is based on the performance of the model on the validation set
- **Test set**
  - The data that is used to simulate real unseen data

29

Train, Validation and Test Sets

```
from sklearn import datasets
# Split Data into 60:20:20 Train:Validation:Test split

from sklearn.model_selection import train_test_split

data = datasets.load_iris()

X=data.data
y=data.target

# 80:20 Train:Test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# Train:Validation split on 80% selected training set
# Validation size of 25% of the 80% Training data
# 0.25 x 0.8 = 0.2
# Note: Variables X_train and y_train are now overwritten
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=1)


print(X_train.shape)
print(X_val.shape)
print(X_test.shape)
```

```
(90, 4)
(30, 4)
(30, 4)
```

⇦ **Output**

30

Train, Validation and Test Sets

- Always tune the model based on the performance on the validation set, once the model is trained on the Train set.
- Never fine-tune a model based on its performance on the Test set.
- Test set is meant to aid in assessing a model's performance in production before the model hits production.

31

Spot-Checking

-select the best model

(comparing the performance of multiple models)

32

Spot Checking Models

- There are four parts to the framework that we need to develop; they are:
- Load Dataset
- Define Models
- Evaluate Models
- Summarize Results
- Select the best model based on performance.

33

Defining Models

- The models defined will be specific to the type predictive modeling problem, e.g. classification or regression.
- The defined models should be diverse, including a mixture of:
- Linear Models.
- Nonlinear Models.
- Ensemble Models.
- Each model should be a given a good chance to perform well on the problem.
- This might be mean providing a few variations of the model with different common or well-known configurations that perform well on average.

34

Standardize ML
Processing Steps with
Pipeline
(simplify the workflow)

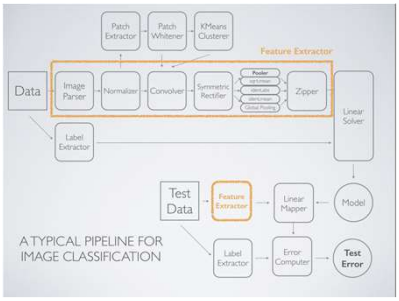
35

Need for a Pipeline

- Streamlines the process of transforming data, training an estimator and using it for prediction

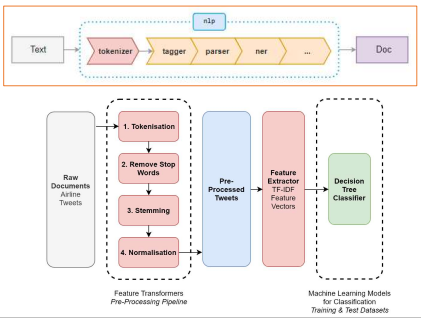
36

Pipeline – Tasks in Image Classification



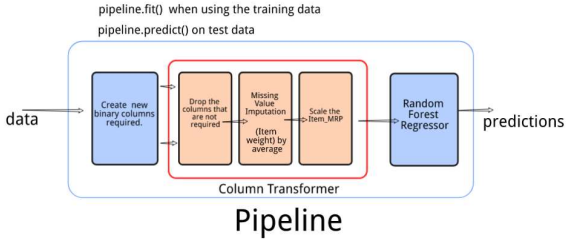
37

Pipeline – Tasks in Natural Language Processing



38

Pipeline



39

Sklearn Pipeline Example

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression

ohe = OneHotEncoder()
imp = SimpleImputer()
clf = LogisticRegression()

from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline

ct = make_column_transformer(
    (ohe, ['Embarked', 'Sex']),
    (imp, ['Age']),
    remainder='passthrough')

pipe = make_pipeline(ct, clf)

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

ct = ColumnTransformer(
    [('encoder', ohe, ['Embarked', 'Sex']),
     ('imputer', imp, ['Age'])],
    remainder='passthrough')

pipe = Pipeline([('preprocessor', ct), ('classifier', clf)])
```

40

Sklearn Pipeline for NLP Tasks

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import (
        CountVectorizer, TfidfTransformer)
from xgboost import XGBClassifier

stopwords, lemmatizer = …

pipeline = Pipeline([
    ('preprocess', MessagePreprocessor(subject_weight=2)),
    ('text', TextProcessor(stopwords, lemmatizer)),
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', XGBClassifier(objective='multi:softmax')),
])
```

41

Hyperparameter Tuning

42

Hyper-Parameter Tuning

- As opposed to parameters (like the ones in linear regression slope and constant term) which change based on the data for a given parametric model, hyper-parameters are preset values even before a ML model gets trained on the data.
- Parameters change during the training process.
- Hyper-parameters are preset and do not change while training.
- The process of setting the right hyper-parameters to get maximum performance out of a given model, is called Hyper-parameter Tuning

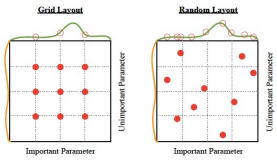
43

Grid Search and Random Search

- Both are the two most common methods of choosing the right hyper-parameters
- In Grid search, each and every combination of hyper-parameters exhaustively tested before selecting the 'best' combination of hyper-parameters
- In Random search, only a subset of combinations can be tested before selecting the 'best' combination of hyper-parameters
- We use Random Search when the parameter grid is fairly large, and we want to save on processing time.
- **GridSearchCV** and **RandomizdSearchCV** are included in the sklearn library to perform the same over a parameter grid, that is passed as an argument to the functions along with the estimator

44

Grid Search vs. Random Search



In the Grid Layout, it's easy to notice that, even if we have trained 9 models, we have used only 3 values per variable!

Whereas, with the Random Layout, it's extremely unlikely that we will select the same variables more than once.

It ends up that, with the second approach, we will have trained 9 models using 9 different values for each variable.

As you can tell from the space exploration at the top of each layout in the image, we have explored the hyperparameters space more widely with Random Search (especially for the more important variables).

This will help us to find the best configuration in fewer iterations.

45

Grid Search vs. Random Search

Grid Search
- Select values for each hyperparameter to test
- Try ALL combinations – exhaustive search through the all-possible combinations of the values for the hyper parameters – Bad for large search space.
- Time consuming & CPU intensive for many values of the hyper parameters

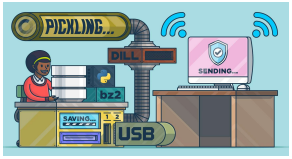
Random Search
- Varies important hyper-parameters more
- Try only a randomly selected subset of combinations – exhaustive search is not performed through the all-possible combinations of the values for the hyper parameters – good for large search space.
- More efficient at model tuning – less Time consuming & less CPU intensive as the search space is reduced due to random subset selection

46

Demo

- **Spot Checking** – select the best model
- **Pipeline** – automating ML workflow
- **Hyper-parameter Tuning** – improving model performance

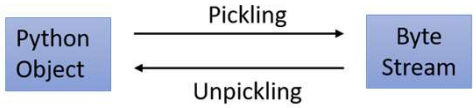
47



Model Persistence

48

Object Serialization with Pickle



49

Object Serialization with Pickle

```
# Import pickle Package
import pickle

# Save the Modle to file in the current working directory
Filename = "Knn.pkl"
with open(Filename, 'wb') as file:
    pickle.dump(knn, file)

# Load the Model back from file
with open(Pkl_Filename, 'rb') as file:
    loaded_knn = pickle.load(file)
```

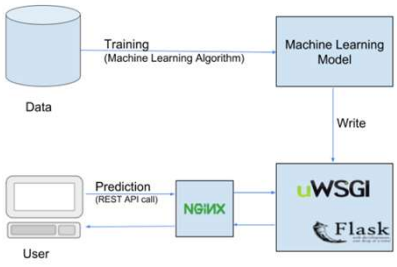
50

Comparing Libraries

Approach	PROS	CONS
Pickle	Quick Implementation Easy Readability	Python version issues. No stored results or data Only File Object
Joblib	Quick Implementation Easy Readability Accepts Objects and String filenames Different compression options.	Python version issues.
Proprietary	More Flexibility No Python version issues	Harder Readability Complexity

51

ML Model Deployment



52

53

To be continued in the next session…..

54