K-fold Cross-Validation in Machine Learning with Python Implementation

helloml.org/k-fold-cross-validation-in-machine-learning-with-python-implementation/

June 20, 2021

In this article, we will discuss K-fold cross-validation in Machine Learning. We will also briefly discuss many associated topics such as Bias vs Variance Trade-off, Train-Test Split and Leave-one-out cross-validation (Loocv).

Train-Test Split

Train-Test split is a technique used to evaluate the performance of a machine learning algorithm. It is used for **classification** and **regression** problems when we have **pre-defined labels** for our data (i.e. **supervised learning** methods).

It generally involves splitting our existing dataset randomly into two subsets, a training set and a test set and is done so that we can know how our model will perform in the real world (by validating our model on new data, i.e. data that the model hasn't been trained upon).

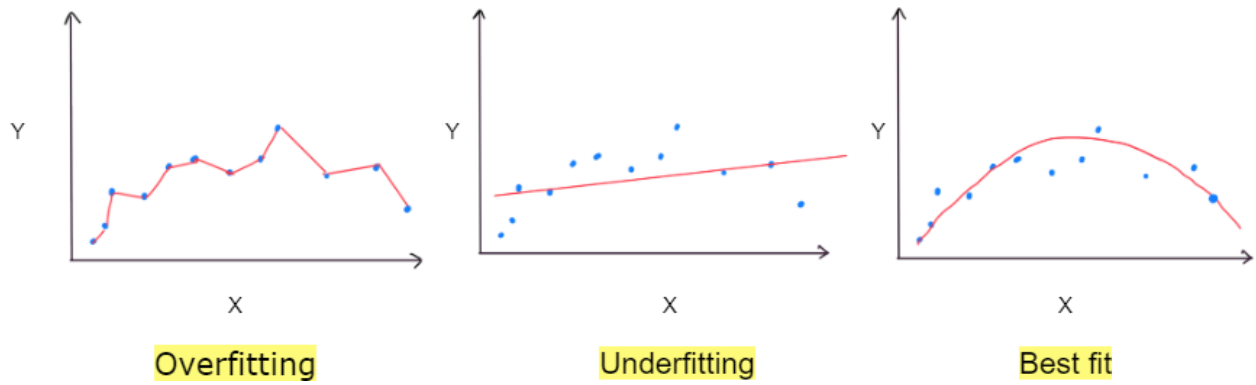
In summary, the machine learning model is fit (trained) on the training set and the test set is then used to evaluate the fit of the model.

The easiest way to perform this split in Python is to use **train_test_split** from **sklearn.model_selection** which accepts input arguments **X** (features), **y** (labels) and the **test_size**, which indicates the fraction of the dataset which should be divided into a test set. It splits data randomly into train and test subsets based on the seed we assign to it.

Problems in normal Train-Test Split – The Bias vs Variance Problem and Selection of Random State

Why not select a model so that it fits **perfectly** on our data? In this case, the model will reach **100%** training accuracy. However, such a model will be highly inaccurate when deployed in production as we don't know whether real-world data will fit our model or not. In short, this leads to **overfitting**.

Overfit, Underfit or the Best fit models are a part of the **Bias vs Variance trade-off** discussion in Machine Learning. However, we will go through it rather quickly.



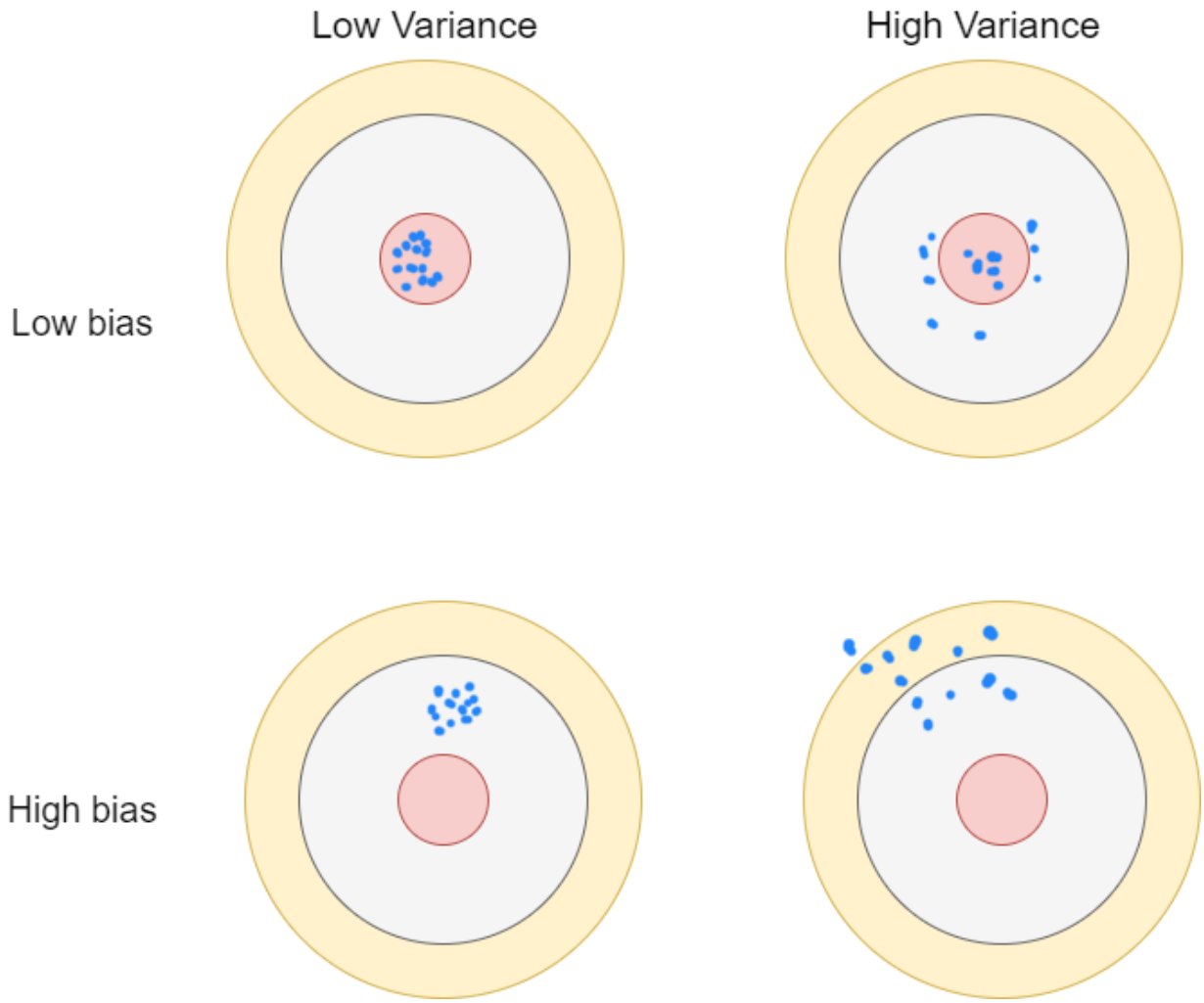
3 different training outcomes possible for our model – overfitting, underfitting and best fit.

In the diagram shown above, the blue points refer to the training data points and the red line is our trained model. In the first case, we fit our model perfectly on the training data and achieve a training accuracy of 100%.

However, this is just like memorizing the questions of a mock test before giving the exam. If the same questions come again in the exam, we are bound to get full marks. In the real world, that doesn't happen since there is variability in the kind of questions asked. In a similar manner, this model doesn't account at all for **variance** in real-world data. However, the linear error (**bias**) is the least (zero) among all three models. **Hence, an overfit model is said to have low bias, high variance.**

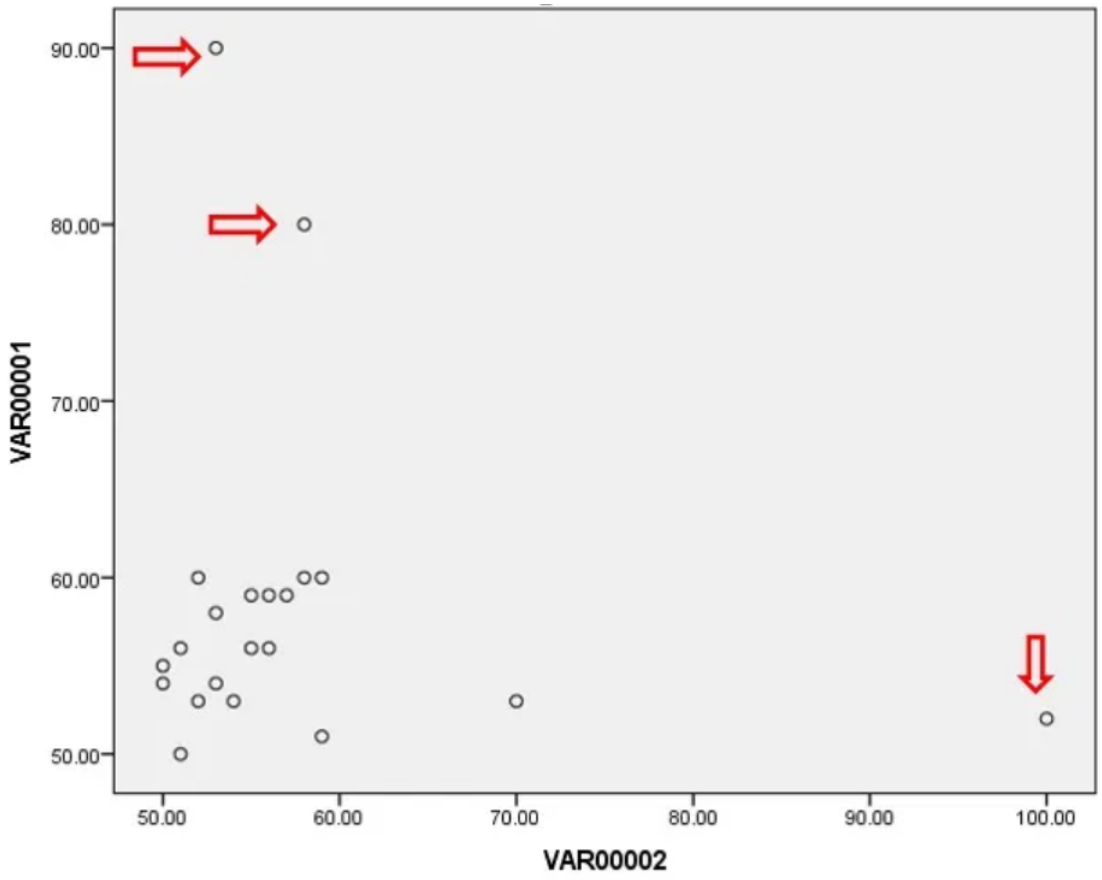
In the second case, the model proves to be inaccurate for a large amount of training data i.e. it is underfitting. However, it leaves a lot of room for variability. **Hence, it is said to have high bias, low variance.**

What we actually desire is a good balance of the aforementioned mix i.e. **Low bias, low variance.** Such models are said to be the "best fit" for deployment. The below figure is a good way to summarize this discussion. We need to (figuratively as well as in real life) hit the bull's eye on unseen data!



Depiction of various permutation-combinations that can be reached by varying levels of bias and variance i.e.
The bias-variance trade-off.

What if we have a dataset like the one shown below?



Dataset with insufficient data and that too with sharp outliers.

Will train-test split account for providing the best fit for our algorithm?

In the case of small datasets/insufficient data, the **isolated data points/outliers** affect the model's accuracy. Whether the outliers are present in the training set or not will lead to the model differing in outcomes on the test set, since the model will be trained in accordance with the points in the training set.

Hence, differing subsets, in this case, will lead to largely different accuracy levels and we won't have a clear answer to how accurate our model really is. In short, accuracy will vary depending on the **random_state** which we provided as an argument to the **train_test_split**.

For the uninitiated, just know that different random states lead to different data points being chosen for the training and testing subsets. Hence, changing the random state changes the data in the train and test sets. And in case of insufficient data, outlier points will cause large fluctuations in the outcome of our model as the random state changes.

This begs the question, is there a way to prevent all of this?

The answer is **Cross-Validation**.

What is Cross-Validation?

Simply put, cross-validation involves **partitioning** our original dataset into **complementary datasets** (complementary means that the resultant datasets after partitioning will be mutually exclusive and the sum (union) of all such datasets forms our original dataset).

We fit the model into one subset. We then validate the analysis on another subset. Next, we repeat this procedure on all the complementary datasets formed. **Cross-validation combines (averages) measures of prediction to derive a more accurate estimate of model prediction performance.** Therefore, our model sees all of the data and is tested on every subset once. Hence, such a model truly accounts for all the outliers and all types of data that we have.

There are different types of cross-validation techniques:

- **Exhaustive cross-validation:**
  - Leave-p-out cross-validation
  - Leave-one-out cross-validation
- **Non-exhaustive cross-validation**
  - K-fold cross-validation
  - Holdout method
  - Repeated random sub-sampling validation

We'll go through one type from each of these two techniques.

Leave one-out cross-validation – LOOCV

The process for LOOCV can be summarized as:

1. Split data set into two parts:
   - The validation set comprises one data point.
   - The train set comprises (n-1) data points.
2. Fit the model using the train set.
3. Measure error on the prediction
4. Repeat for n times taking different data point as the validation set.
5. Take the average of the error as an overall error.

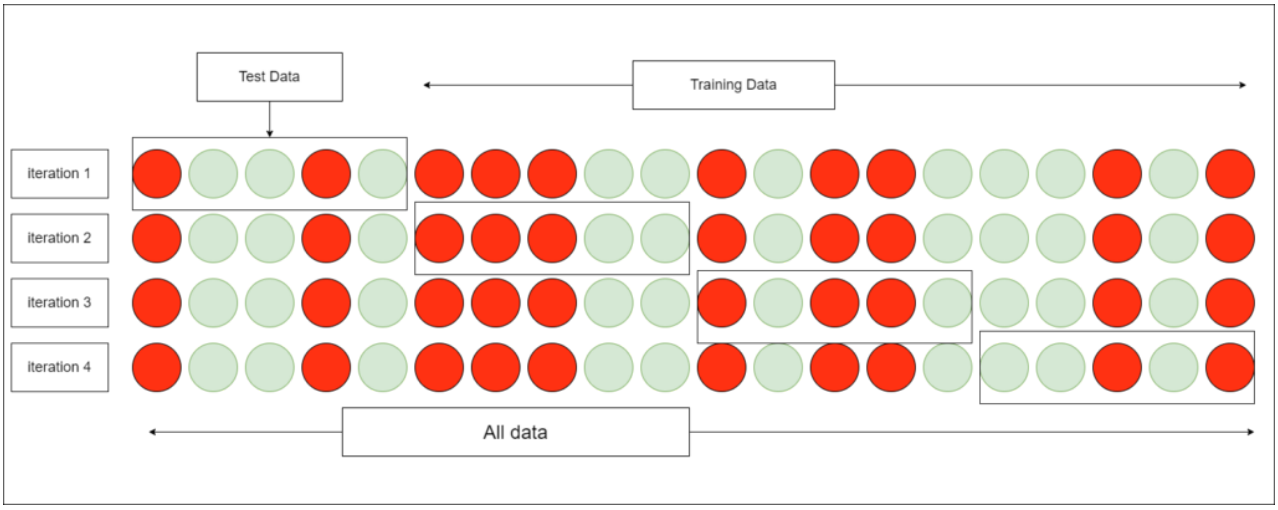
This actually is **K-fold cross validation** taken to the extreme, which we'll see next.

K-fold cross-validation

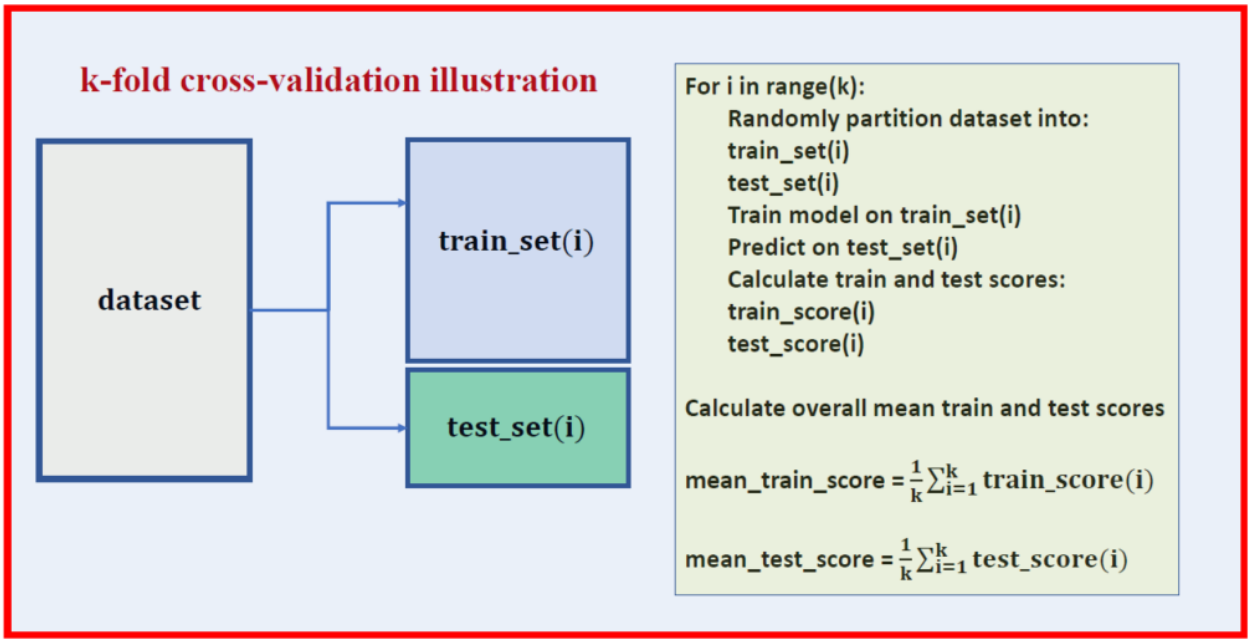
The process for k-fold cross validation can be summarized as:

1. Randomly partition data into k equally sized subsamples:
   - Retain a single subsample as the validation set.
   - The remaining k-1 subsamples are used as the training set.
2. Fit the model.
3. Measure error on the prediction.
4. Repeat for k times taking different subsamples as the validation set.

5. Take the average of the error as an overall error.



Splitting of data in each iteration. Green and red represent different classes of data. (Here N=20, each subsample for the validation set is of length 5. Hence k = 20/5 = 4). Hence, it is said that we are performing CV in 4 folds.



Pseudo Code for k-fold cross-validation.

How does it help in model evaluation?

Rather than performing model diagnosis just by taking training and test scores to decide overfitting/underfitting, now we introduce a third parameter, the **cross-validation score**.

The following table depicts all the scenarios:

Train Score	Test Score	Cross Validation Score	Diagnosis
High	High	High	Good model
High	Low	High	Bad sample in train-test split
High	Low	Low	High variance model
Low	Low	Low	High bias model

Model diagnosis based on the train, test and cross-validation scores

Now, a question arises, about choosing the value of k.

How do we choose the value of K?

That depends on a variety of factors, such as:

- Number of instances in the dataset
- Impact of K on the variance and bias. Generally,
  Lower K implies more bias.
  Higher K implies more variance.
- Computational cost: If the model itself is expensive to train, then the entire process of k-fold cross-validation can be computationally infeasible/draining.

Also, always remember to **shuffle the data** before performing k-fold cross-validation to prevent any sorts of sample bias from being induced into the process.

K-fold cross-validation helps in **reducing overfitting**, helps in **hyperparameter tuning** (finding optimal value of the model hyperparameters so as to improve the efficiency of the chosen machine learning algorithm) as well as knowing which model (among multiple given models) will **fit best** on our data.

There are 2 main limitations of k-fold cross-validation as well, one we discussed already, i.e. if the model itself is expensive to train, then the entire operation will be computationally expensive. Secondly, k-fold cv doesn't work well with time-series data. Hence, it is problematic for time series models.

K-fold cross-validation in action using Python

First, we'll import the essential libraries and required models from scikit-learn. Next, we import the digits dataset included in the scikit-learn package.

```
import numpy as np
import matplotlib.pyplot as plt


%matplotlib inline

from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import load_digits
digits = load_digits()
```

Split into training and testing dataset

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(digits.data,digits.target,test_size=0.3)
```

Implement Logistic Regression on the dataset

```
lr = LogisticRegression(solver='liblinear',multi_class='ovr')
lr.fit(X_train, y_train)
a = lr.score(X_test, y_test)
```

Implement an SVM Classifier

```
svm = SVC(gamma='auto')
svm.fit(X_train, y_train)
b = svm.score(X_test, y_test)
```

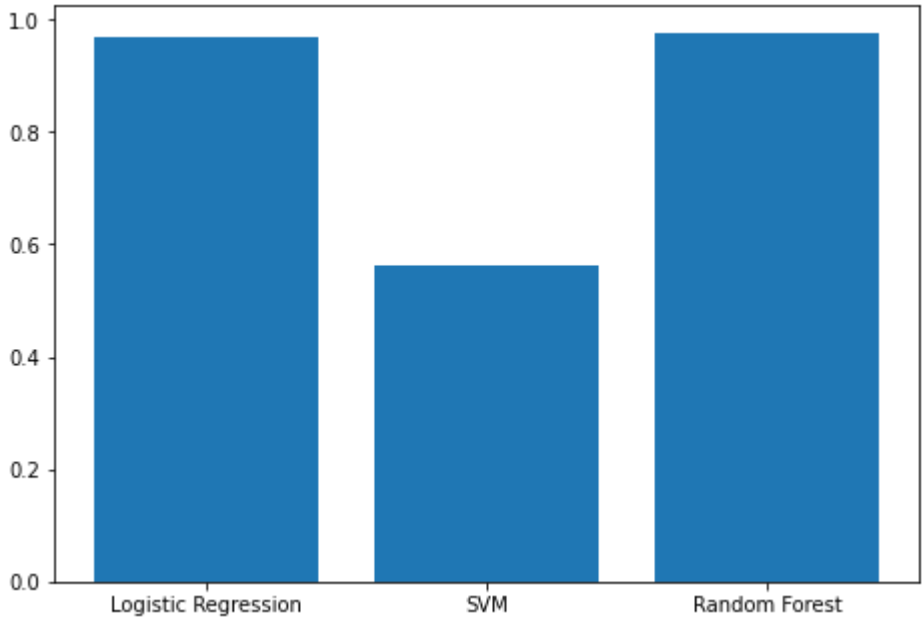
Implement a Random Forest Classifier

```
rf = RandomForestClassifier(n_estimators=40)
rf.fit(X_train, y_train)
c = rf.score(X_test, y_test)
```

Plot the accuracies achieved by each of the classifiers

```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
langs = ['Logistic Regression', 'SVM', 'Random Forest']
scores = [a,b,c]
ax.bar(langs,scores)
plt.show()

print("Accuracy of Logistic Regression ", a)
print("Accuracy of SVM ", b)
print("Accuracy of Random Forest ", c)
```

Output



plt.show() comparing the accuracies of all the models.

```
Accuracy of Logistic Regression  0.9685185185185186
Accuracy of SVM  0.5611111111111111
Accuracy of Random Forest  0.9777777777777777
```

As you can see, the approximate accuracies for Logistic Regression, SVM and Random Forest classifiers are 96.85%, 56.11% and 97.77% respectively.

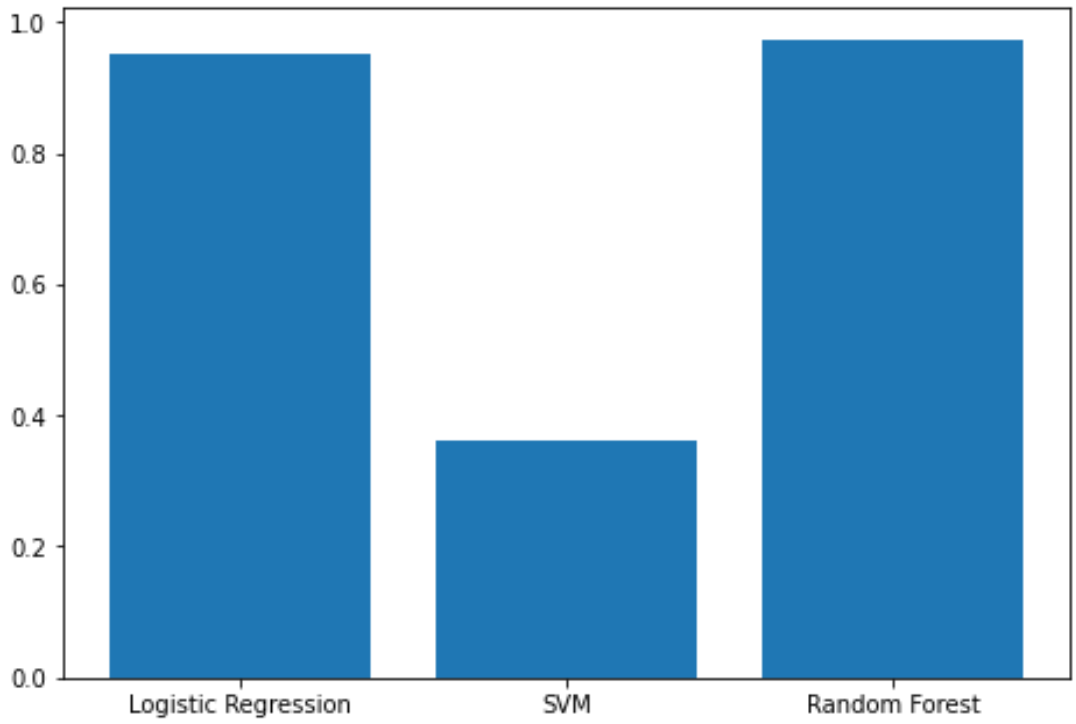
Now, we shall see why cross-validation is important. To show that, first we'll perform the entire process from the start.

Since the train-test split chooses subsets randomly, different data points in the training and testing sets will affect the model accuracy. Hence, we will not have a clear picture of the true accuracies achieved and it is not possible to simply run train-test split all over again in the case of computationally expensive models. (Looping for x number of times and then getting an average score also doesn't help since there is **no** guarantee of outlier selection/rejection in any of the subsets any time you run train-test split.

What we want is a **guarantee** that all the data points have been accounted for in training and we shall see how cross-validation does exactly that.)

Same thing performed all over again (this time train-test split will chose different data for the subsets since it always chooses randomly. This will affect model accuracy)

```
% perform imports

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(digits.data,digits.target,test_size=0.3)

% define and fit data to the models

lr = LogisticRegression(solver='liblinear',multi_class='ovr')
lr.fit(X_train, y_train)
a = lr.score(X_test, y_test)
print("Accuracy of Logistic Regression ", a)

svm = SVC(gamma='auto')
svm.fit(X_train, y_train)
b = svm.score(X_test, y_test)
print("Accuracy of SVM ", b)

rf = RandomForestClassifier(n_estimators=40)
rf.fit(X_train, y_train)
c = rf.score(X_test, y_test)
print("Accuracy of Random Forest ", c)

% print and plot accuracies

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
langs = ['Logistic Regression', 'SVM', 'Random Forest']
scores = [a,b,c]
ax.bar(langs,scores)
plt.show()
```



plt.show() output

```
Accuracy of Logistic Regression  0.95
Accuracy of SVM  0.362962962962963
Accuracy of Random Forest  0.9740740740740741
```

This time, the accuracies achieved are different for all the 3 classifiers.

And if you observe, it is radically different for the SVM classifier. A difference of almost 20%.

This means that depending on the kind of data in training/testing sets, the model accuracy fluctuates by 20%. This is surely not a good sign for evaluating our model. We clearly cannot infer anything from this, since we can't even approximate how well/badly our model is performing. As a result, we'll perform k-fold cross-validation instead of the traditional train-test split, account for all the kinds of data and create a final accuracy score based on the average of accuracies received in each of the k-folds (k-iterations).

K-Fold Cross Validation

Basic Example

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=3)
kf

KFold(n_splits=3, random_state=None, shuffle=False)

for train_index, test_index in kf.split([1,2,3,4,5,6,7,8,9]):
  print(train_index, test_index)

[3 4 5 6 7 8] [0 1 2]
[0 1 2 6 7 8] [3 4 5]
[0 1 2 3 4 5] [6 7 8]

from sklearn.model_selection import KFold
kf = KFold(n_splits=9)
for train_index, test_index in kf.split([1,2,3,4,5,6,7,8,9]):
  print(train_index, test_index)

[1 2 3 4 5 6 7 8] [0]
[0 2 3 4 5 6 7 8] [1]
[0 1 3 4 5 6 7 8] [2]
[0 1 2 4 5 6 7 8] [3]
[0 1 2 3 5 6 7 8] [4]
[0 1 2 3 4 6 7 8] [5]
[0 1 2 3 4 5 7 8] [6]
[0 1 2 3 4 5 6 8] [7]
[0 1 2 3 4 5 6 7] [8]
```

Using K-fold (k = 3) for our digits dataset

```
def get_score(model, X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    return model.score(X_test, y_test)
```

```
from sklearn.model_selection import StratifiedKFold
folds = StratifiedKFold(n_splits=3)
scores_logistic = []
scores_svm = []
scores_rf = []

for train_index, test_index in folds.split(digits.data,digits.target):
    X_train, X_test, y_train, y_test = digits.data[train_index],
digits.data[test_index], \
                                       digits.target[train_index],
digits.target[test_index]

scores_logistic.append(get_score(LogisticRegression(solver='liblinear',multi_class=
 X_train, X_test, y_train, y_test))
    scores_svm.append(get_score(SVC(gamma='auto'), X_train, X_test, y_train,
y_test))
    scores_rf.append(get_score(RandomForestClassifier(n_estimators=40), X_train,
X_test, y_train, y_test))
```

Scores obtained by Logistic Regression

```
scores_logistic

[0.8948247078464107, 0.9532554257095158, 0.9098497495826378]
```

Scores obtained by SVM

```
scores_svm

[0.3806343906510851, 0.41068447412353926, 0.5125208681135225]
```

Scores obtained by Random Forest

```
scores_rf

[0.9298831385642737, 0.9432387312186978, 0.9298831385642737]
```

Average Scores

```
print("Average Score of Logistic Regression: ", np.average(scores_logistic))
print("Average Score of SVM: ", np.average(scores_svm))
print("Average Score of Random Forest: ", np.average(scores_rf))

Average Score of Logistic Regression:  0.9193099610461881
Average Score of SVM:  0.4346132442960489
Average Score of Random Forest:  0.9343350027824151
```

Importing the cross_val score function

```
from sklearn.model_selection import cross_val_score
```

Logistic regression model performance using cross_val_score

```
a = cross_val_score(LogisticRegression(solver='liblinear',multi_class='ovr'),
digits.data, digits.target,cv=3)
print("Linear Regression scores ", a)
np.average(a)

Linear Regression scores  [0.89482471 0.95325543 0.90984975]
0.9193099610461881
```

SVM model performance using cross_val_score

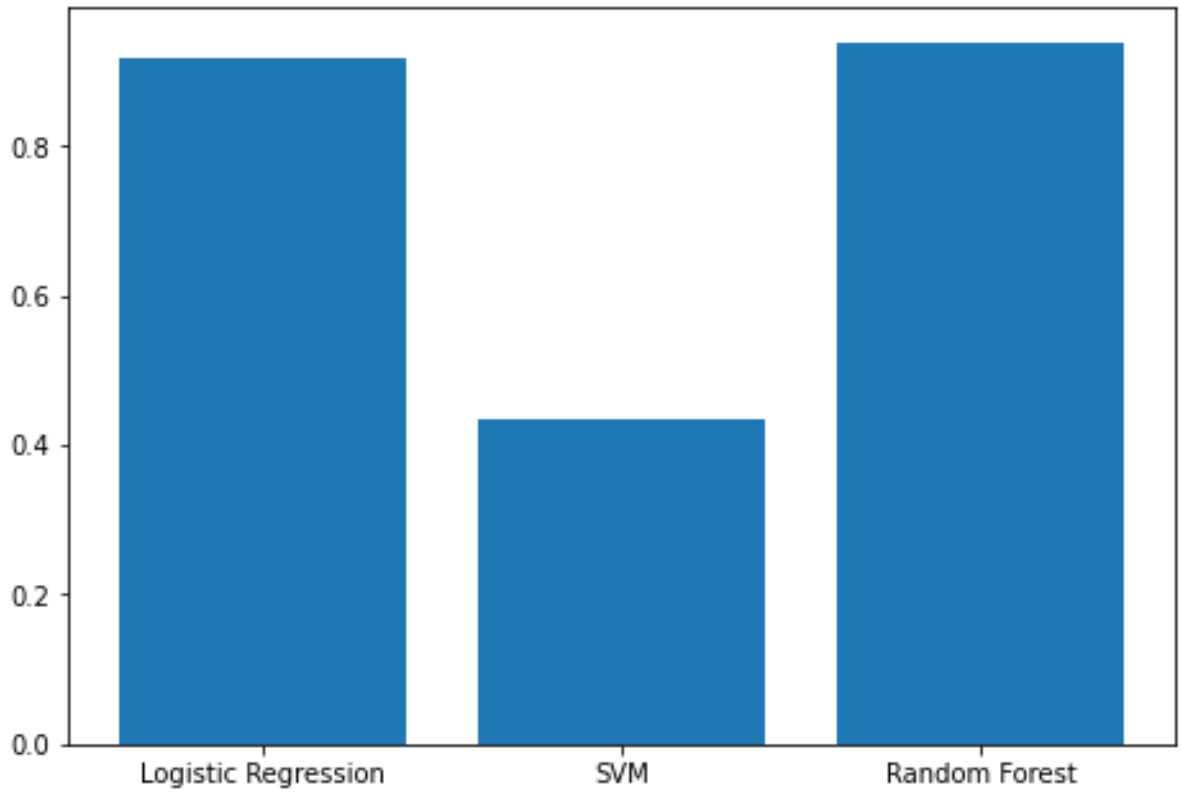
```
b = cross_val_score(SVC(gamma='auto'), digits.data, digits.target,cv=3)
print("SVM scores ", b)
print(np.average(b))

SVM scores  [0.38063439 0.41068447 0.51252087]
0.4346132442960489
```

Random forest model performance using cross_val_score

```
c = cross_val_score(RandomForestClassifier(n_estimators=40),digits.data,
digits.target,cv=3)
print("Random Forest scores ", c)
print(np.average(c))

Random Forest scores  [0.91652755 0.94824708 0.91819699]
0.9276572064552031
```

Plotting out the final achieved accuracies

```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
langs = ['Logistic Regression', 'SVM', 'Random Forest']
scores = [np.average(a),np.average(b),np.average(c)]
ax.bar(langs,scores)
plt.show()

print("Accuracy of Logistic Regression ", np.average(a))
print("Accuracy of SVM ", np.average(b))
print("Accuracy of Random Forest ", np.average(c))
```



plt.show() output

```
Accuracy of Logistic Regression  0.9193099610461881
Accuracy of SVM  0.4346132442960489
Accuracy of Random Forest  0.9276572064552031
```

The entire point of this exercise was truly evaluating our model performance. Now, since we have accounted for all of our data and can say that approximate accuracies for our LR, SVM and RF classifiers for the given data are 91.93%, 43.46% and 92.76% respectively since we have accounted for all the present data.

That's it! If you've come this far, surely you now know that the next time your model gives largely varying results upon choosing different random states for train_test_split, cross-validation may be what you're really looking for! If you want to fine-tune your model (perform hyper-parameter tuning), cross-validation once again may be your aide.

Thanks for reading this far and happy learning!

Topics for Further Reading

We have discussed about simple K-fold cross-validation in this article. There are many variations of it, namely:

- **Leave-one-out cross-validation**
- **Stratified k-fold cross-validation**
- **Repeated cross-validation**

iamajit.pythonanywhere.com