//

Machine Learning V

maXbox Starter 67 - Data Science with Max

There are two kinds of data scientists:
 1) Those who can extrapolate from incomplete data.

This tutor makes a second step with our classifiers in Scikit-Learn of the preceding tutorial 66 with predicting test sets and histograms.

Lets start with a prediction of one sample:

```
svm = LinearSVC(random_state=100, C=2)  #C for regularization a scale
svm.fit(X,y)
print('predict single sample ',svm.predict([[7,8,9,8]]))
>>> array([1.])
```

As you can see the sample is unseen data and was not in the dataset before:

```
   A    B    C    D
[[ 1.   2.   3.   4.   0.]
 [ 3.   4.   5.   6.   0.]
 [ 5.   6.   7.   8.   1.]
 [ 7.   8.   9. 10.   1.]
 [10.   8.   6.   4.   0.]
 [ 9.   7.   5.   3.   1.]]
```

Next topic is the data split. Normally the data we use is split into training data and test data. The training set contains a known output and the model learns on this data alone in order to be generalized to other data later on. We have a test dataset (or subset) in order to test our model's prediction on this subset. We'll do this by Scikit-Learn library and specifically:

```
from sklearn.model_selection import train_test_split
```

But our dataset is very small so we have to split it by 0.5. A test_size=0.2 inside the function indicates a percentage of the data that should be held over for testing. It's usually around 80/20 or 70/30.

```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.5,random_state=0)
```

```
X_train      [[7. 8. 9.10.]
              [ 1. 2. 3. 4.]
              [10. 8. 6. 4.]]

X_test       [[9. 7. 5. 3.]
              [ 5. 6. 7. 8.]
              [ 3. 4. 5. 6.]]
```

Now we fit the model on the training data:

```
svm.fit(X_train,y_train)
y_pred = svm.predict(X_test)
print('class test report:
             \n',classification_report(y_test,y_pred,target_names=targets))
```

As you can see, we are fitting the model on the training data and trying to predict the test data.

```
class test report:
              precision    recall  f1-score   support

    class 0       0.50      1.00      0.67         1
    class 1       1.00      0.50      0.67         2

avg / total       0.83      0.67      0.67         3
```

There are two possible predicted classes: 1 as "yes" and 0 as "no". If we were predicting for example the presence of a disease, "yes" would mean they have the disease, and "no" would mean they don't have the disease.

The class test Score is 0.66 and the train score is 1.0; its usual that the train score is higher than the test score to avoid over-fitting.
What about the confusion matrix:

Confusion Matrix		Predicted	
		Negative	Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

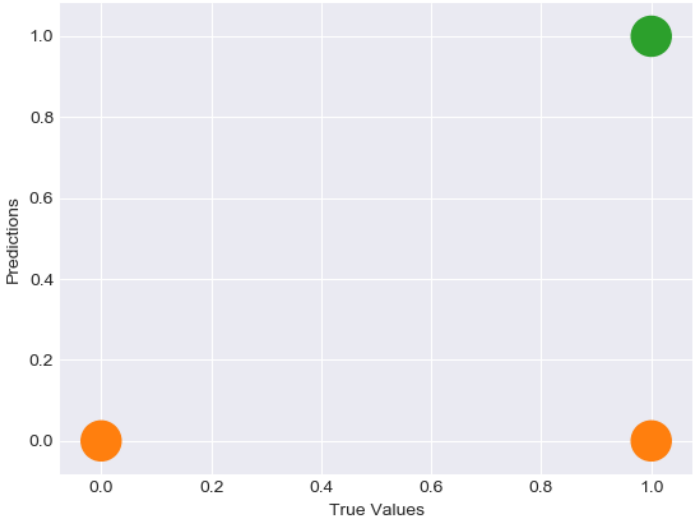
```
print('Class test Score:', svm.score(X_test, y_test))
print(confusion_matrix(y_test, y_pred))
[[1 0]
 [1 1]]
```

There's a false negative in the 0-Class so the precision is only 0.5!
Best fit would be:

```
[[1 0]
 [0 2]]
```

Now we plot the model (500 is just the marker bubble size):

```
plt.scatter(y_test, y_pred, 500)
plt.xlabel('True Values')
plt.ylabel('Predictions')
plt.show()
```



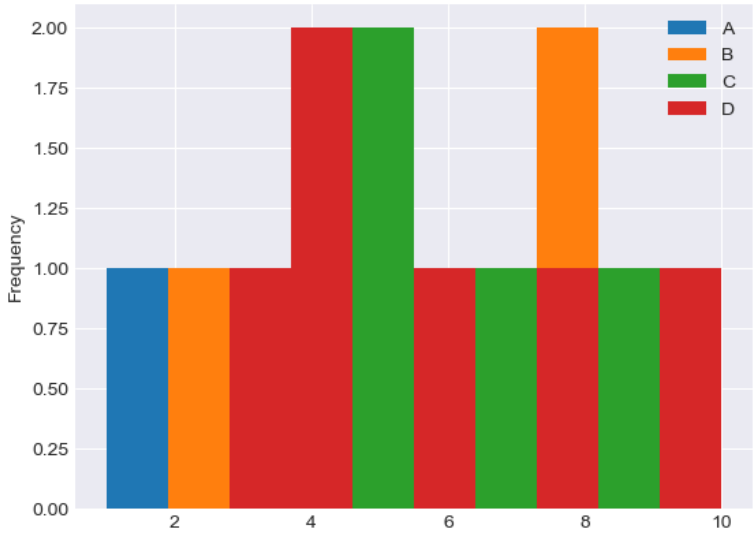
Now we want to see how to build a histogram in Python from the ground up.

```
df= pd.DataFrame.from_records(arr2, columns=features+['Class'])
df.iloc[0:,0:4].plot.hist(density=False)
plt.show()
```

We want to see how many times our features X[0:4] occur

```
    A    B    C    D
[[ 1.   2.   3.   4.]
 [ 3.   4.   5.   6.]
 [ 5.   6.   7.   8.]
 [ 7.   8.   9.  10.]
 [10.   8.   6.   4.]
 [ 9.   7.   5.   3.]]
```

Lets take the feature D the 4 is count 2 times, the 3,6,8,10 counts one time. Mathematically, a histogram is a mapping of bins (intervals or numbers) to frequencies. More technically, it can be used to approximate a probability density function (PDF) of the underlying variable that we see later on.



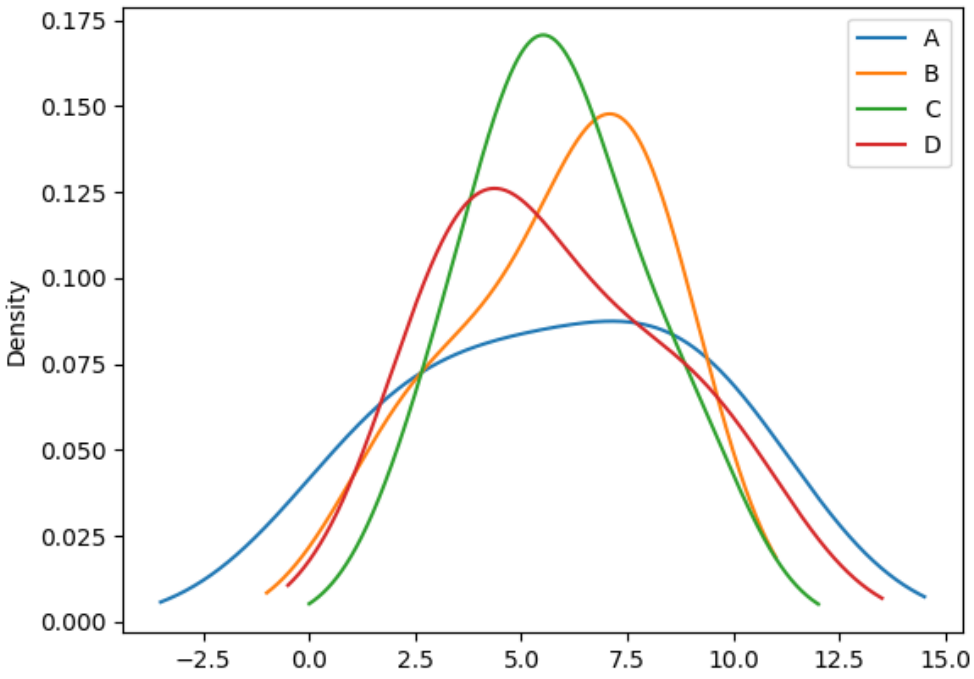
Moving on from a frequency table above (density=**False** counts at y-axis), a true histogram first <bins> the range of values and then counts the number of values that fall into each bin or interval. Our bins are so small that we can count each number as one bin. A plot of a histogram uses its bin edges on the x-axis and the corresponding frequencies on the y-axis.

Now a kernel density estimation (KDE) is a way to estimate the probability density function (PDF) of the random variable that "underlies" our sample. KDE is a means of data smoothing and studying density.

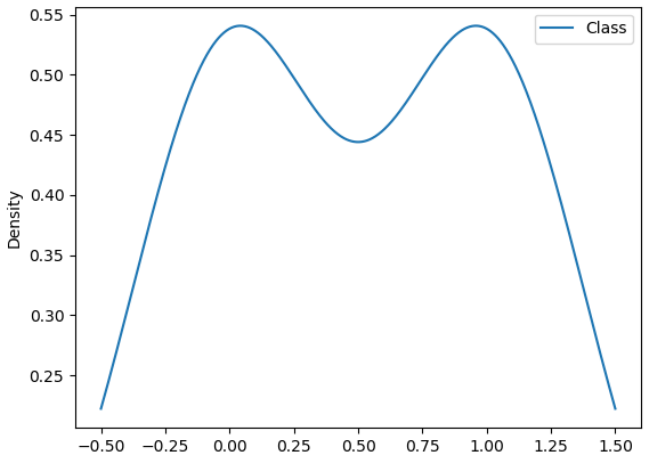
```
print(y,'\n',X,'\n')
[0. 0. 1. 1. 0. 1.]
[[ 1.   2.   3.   4.]
 [ 3.   4.   5.   6.]
 [ 5.   6.   7.   8.]
 [ 7.   8.   9.  10.]
 [10.   8.   6.   4.]
 [ 9.   7.   5.   3.]]
```

```
df.iloc[0:,0:4].plot.kde()
```

Sticking with the Pandas library, you can create and overlay density plots using plot.kde(), which is available for both [Series] and [DataFrame] objects.
`df.iloc[0:,0:4].plot.kde()`

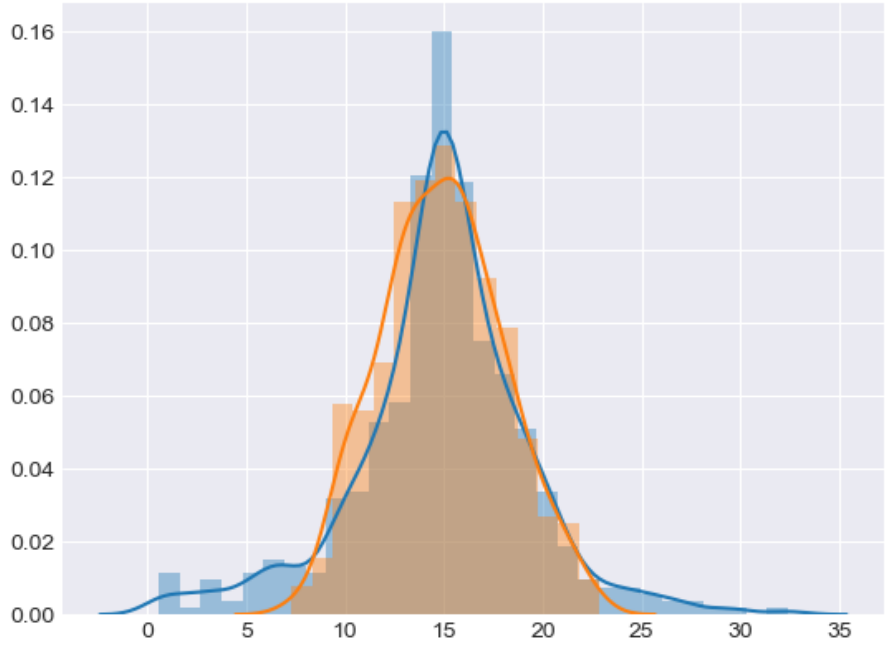


This is also possible for our binary targets to see a probabilistic distribution of the target class values (labels in supervised learning): **[0. 0. 1. 1. 0. 1.]**



Consider at last a sample of floats drawn from the Laplace and Normal distribution together. This distribution graph has fatter tails than a normal distribution and has two descriptive parameters (location and scale):

```
>>> d = np.random.laplace(loc=15, scale=3, size=500)
>>> d = np.random.normal(loc=15, scale=3, size=500)
```




```
svm = LinearSVC(random_state=100)
y_pred = svm.fit(X,y).predict(X)    # fit and predict in one line
print('linear svm score: ',accuracy_score(y, y_pred))
```

"Classification with Cluster & PCA Example"

Now we will use linear SVC to partition our graph into clusters **and** split the data into a training set **and** a test set **for** further predictions.

```
 X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# Run classifier, using a model that is too regularized (C too low) to see
# the impact on the results

 classifier = svm.SVC(kernel='linear', C=0.01)
 y_pred = classifier.fit(X_train, y_train).predict(X_test)
```

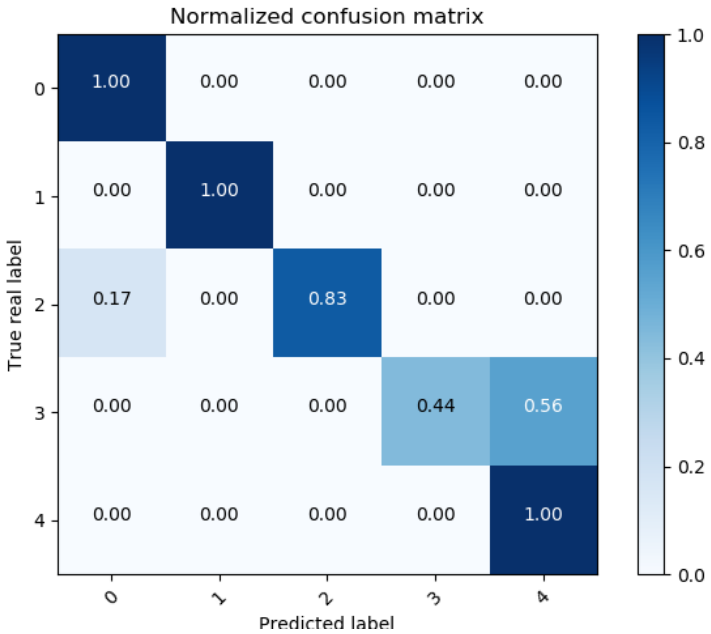
By setting up a dense mesh of points **in** the grid **and** classifying all of them, we can render the regions of each cluster **as** distinct colors:

```
def plotPredictions(clf):
        xx, yy = np.meshgrid(np.arange(0, 250000, 10),
                             np.arange(10, 70, 0.5))
        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
        plt.figure(figsize=(8, 6))
        Z = Z.reshape(xx.shape)
        plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)
        plt.scatter(X[:,0], X[:,1], c=y.astype(np.float))
        plt.show()
```

A simple CNN architecture was trained on MNIST dataset using TensorFlow with 1e-3 learning rate and cross-entropy loss using four different optimizers: SGD, Nesterov Momentum, RMSProp and Adam.

We compared different optimizers used in training neural networks and gained intuition for how they work. We found that SGD with Nesterov Momentum and Adam produce the best results when training a simple CNN on MNIST data in TensorFlow.

https://sourceforge.net/projects/maxbox/files/Docu/EKON_22_machinelearning_slides_scripts.zip/download



Last note concerning PCA and Data Reduction or Factor Analysis:

As PCA simply transforms the **input** data, it can be applied both to classification **and** regression problems. In this section, we will use a classification task to discuss the method.
The script can be found at:
  http://www.softwareschule.ch/examples/811_mXpcatest_dmath_datascience.pas
  ..\examples\811_mXpcatest_dmath_datascience.pas

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QA

clf = QA()
y_pred = clf.fit(X,y).predict(X)
print('\n QuadDiscriminantAnalysis score2: ',accuracy_score(y, y_pred))
print(confusion_matrix(y, y_pred))
```

  It may be seen that:
- High correlations exist between the original variables, which are therefore **not** independent
- The first principal factor **is** negatively correlated with the second **and** fourth variables, **and** positively correlated with the third variable
- The second principal factor **is** positively correlated with the first variable
- The table of principal factors show that the highest scores are usually associated with the first two principal factors, **in** agreement with the previous results

```
Const
  N    = 6;  { Number of observations }
  Nvar = 4;   { Number of variables }
```

Of course, its **not** always this **and** that simple. Often, we don't know what number of dimensions **is** advisable **in** upfront. In such a case, we leave n_components **or** Nvar parameter unspecified when initializing PCA to let it calculate the full transformation. After fitting the data, explained_variance_ratio_ contains an

```
array of ratios in decreasing order: The first value is the ratio of the basis
vector describing the direction of the highest variance, the second value is the
ratio of the direction of the second highest variance, and so on.

print('pearson correlation, coeff:, p-value:')
for i in range(3):
  print (pearsonr(X[:,i],X[:,i+1]))

corr = np.corrcoef(X, rowvar=0)  # correlation matrix
w, v = np.linalg.eig(corr)        # eigen values & eigen vectors
print('eigenvalues & eigenvector:')
print(w) print(v)

>>> eigenvalues & eigenvector:
[ 2.66227922e+00  1.33772078e+00 -4.33219131e-18  6.51846049e-17]

[[-0.46348627  0.56569851 -0.11586592  0.19557765]
 [-0.5799298   0.27966438  0.53986421  0.24189689]
 [-0.5724941  -0.30865195 -0.71438049 -0.75459654]
 [-0.34801208 -0.71169306  0.42986304  0.57777101]]
C:\maXbox\mX46210\DataScience\confusionlist
```

You can detect high-multi-collinearity by inspecting the *eigen values* of *correlation matrix*. A very low eigen value shows that the data are collinear, and the corresponding *eigen vector* shows which variables are collinear. If there is no collinearity in the data, you would expect that none of the eigen values are close to zero.

Being a linear method, PCA has, of course, its limitations when we are faced with strange data that has non-linear relationships. We wont go into much more details here, but its sufficient to say that there are extensions of PCA.

The script can be found:

http://www.softwareschule.ch/examples/classifier_compare2confusion2.py.txt

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
```

Ref:

http://www.softwareschule.ch/box.htm

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

https://realpython.com/python-histograms/

Doc:

https://maxbox4.wordpress.com