# Classification with Python

# Import Tools and Download Data

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import  train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score

# load iris data
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
attributes = ['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Class']
iris = pd.read_csv(url, sep=',', header=None, names=attributes, index_col=None, )
```

# Prepare Data and Create Model

```
# Preparing data
X = iris.drop('Class', axis=1).values
# input data
y = iris['Class'].values                                    # target data
validation_size = 0.20                                              # validateion ratio
seed = 7                                                    # random seed
X_train, X_validation, y_train, y_validation = train_test_split(X, y,
    test_size=validation_size, random_state=seed)
# CART: Make predictions on validation dataset
cart = DecisionTreeClassifier()
cart.fit(X_train, y_train)
predicted = cart.predict(X_validation)
print(accuracy_score(y_validation, predicted))
print(confusion_matrix(y_validation, predicted))
print(classification_report(y_validation, predicted))
```

# More Machine Learning

```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
```

# Prepare data

```
# load iris data
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
attributes = ['SepalLength','SepalWidth','PetalLength','PetalWidth','Class']
iris = pd.read_csv(url, sep=',', header=None, names=attributes, index_col=None, )

# Preparing data
X = iris.drop('Class', axis=1).values
# input data
y = iris['Class'].values                    # target data

validation_size = 0.20                              # validateion ratio
seed = 7                                    # random seed
X_train, X_validation, y_train, y_validation = train_test_split(X, y,
    test_size=validation_size, random_state=seed)
```

# Prepare models

```
# Test options and evaluation metric
scoring = 'accuracy'
# Spot Check Algorithms
models = []
models.append(('CART', DecisionTreeClassifier()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
models.append(('QDA', QuadraticDiscriminantAnalysis()))
models.append(('GPC', GaussianProcessClassifier()))
models.append(('MLP', MLPClassifier()))
models.append(('ADC', AdaBoostClassifier()))
models.append(('RFC', RandomForestClassifier()))
```

# Evaluate Models

```
# evaluate each model in turn
results = []
names = []
seed = random.seed(42)
for name, model in models:
    kfold = model_selection.KFold(n_splits=10, random_state=seed)

    cv_results = model_selection.cross_val_score(model, X_train, y_train,
                                cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

# Plot model comparisons

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

# Linear Regression with Python

# Import tools

- import pandas as pd
- import numpy as np
- import matplotlib.pyplot as plt
- import seaborn as sns
- %matplotlib inline

# Explore data

```
USAhousing = pd.read_csv('USA_Housing.csv')

USAhousing.head()

USAhousing.info()

USAhousing.describe()

USAhousing.columns

sns.pairplot(USAhousing)

sns.distplot(USAhousing['Price'])

sns.heatmap(USAhousing.corr())

X = USAhousing[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms', 'Avg. Area Number of Bedrooms', 'Area Population']]

y = USAhousing['Price']
```

# Train model

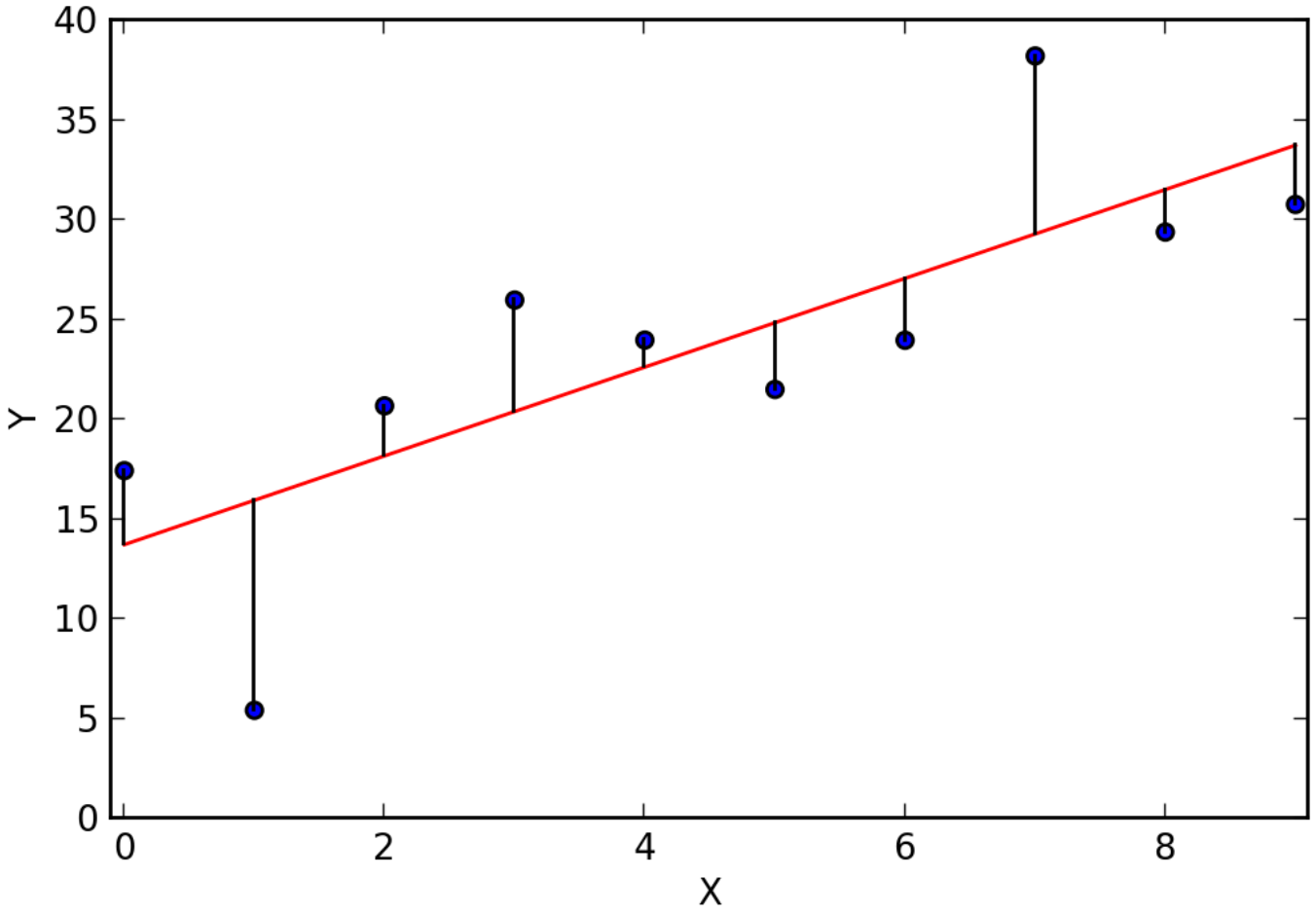
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=101)

from sklearn.linear_model import LinearRegression

lm = LinearRegression()

lm.fit(X_train,y_train)

print(lm.intercept_)  # print the intercept

coeff_df = pd.DataFrame(lm.coef_,X.columns,columns=['Coefficient'])

predictions = lm.predict(X_test)

plt.scatter(y_test,predictions)

sns.distplot((y_test-predictions),bins=50);
```

# Metrics

```
from sklearn import metrics
print('MAE:', metrics.mean_absolute_error(y_test, predictions))
print('MSE:', metrics.mean_squared_error(y_test, predictions))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, predictions)))
```

# Pytorch Artificial Neural Network

- Linear Regression with PyTorch
- In this section we'll use PyTorch's machine learning model to progressively develop a best-fit line for a given set of data points. Like most linear regression algorithms, we're seeking to minimize the error between our model and the actual data, using a loss function like mean-squared-error.


40
35
30
25
20
15
10
5
0
Y
0
2
4
6
8
X

# Import tools

```
import torch
from torch import nn
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

# Create a column matrix of X values

We can create tensors right away rather than convert from NumPy arrays.

```
X = torch.linspace(1,50,50).reshape(-1,1)
# Equivalent to
# X = torch.unsqueeze(torch.linspace(1,50,50), dim=1)
X = torch.linspace(1,50,50).reshape(-1,1)
# Equivalent to
# X = torch.unsqueeze(torch.linspace(1,50,50), dim=1)
```

# Create a "random" array of error values

We want 50 random integer values that collectively cancel each other out.

```
torch.manual_seed(71) # to obtain reproducible results
e = torch.randint(-8,9,(50,1),dtype=torch.float)
print(e.sum())
torch.manual_seed(71) # to obtain reproducible results
e = torch.randint(-8,9,(50,1),dtype=torch.float)
print(e.sum())
```

# Create a column matrix of y values

Here we'll set our own parameters of  weight=2,bias=1 , plus the error amount.

y will have the same shape as X and e

y = 2*X + 1 + e

print(y.shape)

# Plot the results

We have to convert tensors to NumPy arrays just for plotting.

```
plt.scatter(X.numpy(), y.numpy())
plt.ylabel('y')
plt.xlabel('x');
```

Note that when we created tensor $X$ , we did not pass requires_grad=True. This means that $y$ doesn't have a gradient function, and y.backward() won't work. Since PyTorch is not tracking operations, it doesn't know the relationship between $X$ and $y$ .

# Simple linear model

As a quick demonstration we'll show how the built-in nn.Linear() model preselects weight and bias values at random.

```
torch.manual_seed(59)

model = nn.Linear(in_features=1, out_features=1)
print(model.weight)
print(model.bias)
```

# Model classes

PyTorch lets us define models as object classes that can store multiple model layers. This is a single layer.

```
class Model(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.linear = nn.Linear(in_features, out_features)

    def forward(self, x):
        y_pred = self.linear(x)
        return y_pred
```

When Model is instantiated, we need to pass in the size (dimensions) of the incoming and outgoing features. For our purposes we'll use (1,1).

As above, we can see the initial hyperparameters.

```
torch.manual_seed(59)
model = Model(1, 1)
print(model)
print('Weight:', model.linear.weight.item())
print('Bias:  ', model.linear.bias.item())
```

As models become more complex, it may be better to iterate over all the model parameters:

```
for name, param in model.named_parameters():
    print(name, '\t', param.item())
```

NOTE: In the above example we had our Model class accept arguments for the number of input and output features.

For simplicity we can hardcode them into the Model:

```
class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(1,1)
model = Model()
```

Alternatively we can use default arguments:

```
class Model(torch.nn.Module):
    def __init__(self, in_dim=1, out_dim=1):
        super().__init__()
        self.linear = torch.nn.Linear(in_dim,out_dim)

model = Model()
# or
model = Model(i,o)
```

Now let's see the result when we pass a tensor into the model.

```
x = torch.tensor([2.0])
print(model.forward(x))
# equivalent to print(model(x))
```

# Plot the initial model

We can plot the untrained model against our dataset to get an idea of our starting point.

```
x1 = np.array([X.min(),X.max()])
print(x1)
```

```
w1,b1 = model.linear.weight.item(), model.linear.bias.item()
print(f'Initial weight: {w1:.8f}, Initial bias: {b1:.8f}')
print()

y1 = x1*w1 + b1
print(y1)
```

```
plt.scatter(X.numpy(), y.numpy())
plt.plot(x1,y1,'r')
plt.title('Initial Model')
plt.ylabel('y')
plt.xlabel('x');
```

# Set the loss function

We could write our own function to apply a Mean Squared Error (MSE) that follows

$$MSE=(1/n)\sum(y_i-\hat{y}_i)^2=(1/n)\sum(y_i-(wx_i+b))^2, i = 1 \dots, n$$

Fortunately PyTorch has it built in.

By convention, you'll see the variable name "criterion" used, but feel free to use something like "linear_loss_func" if that's clearer.

```
criterion = nn.MSELoss()
```

# Set the optimization

Here we'll use Stochastic Gradient Descent (SGD) with an applied learning rate (lr) of 0.001. Recall that the learning rate tells the optimizer how much to adjust each parameter on the next round of calculations. Too large a step and we run the risk of overshooting the minimum, causing the algorithm to diverge. Too small and it will take a long time to converge.

For more complicated (multivariate) data, you might also consider passing optional momentum and weight_decay arguments. Momentum allows the algorithm to "roll over" small bumps to avoid local minima that can cause convergence too soon. Weight decay (also called an L2 penalty) applies to biases.

```
optimizer = torch.optim.SGD(model.parameters(), lr = 0.001)

# You'll sometimes see this as
# optimizer = torch.optim.SGD(model.parameters(), lr = 1e-3)
```

# Train the model

- An epoch is a single pass through the entire dataset. We want to pick a sufficiently large number of epochs to reach a plateau close to our known parameters of  weight=2,bias=1

# Let's walk through the steps we're about to take:

**Let's walk through the steps we're about to take:**

1. Set a reasonably large number of passes
   ```
   epochs = 50
   ```
2. Create a list to store loss values. This will let us view our progress afterward.
   ```
   losses = []
   for i in range(epochs):
   ```
3. Bump "i" so that the printed report starts at 1
   ```
       i+=1
   ```
4. Create a prediction set by running "X" through the current model parameters
   ```
       y_pred = model.forward(X)
   ```
5. Calculate the loss
   ```
       loss = criterion(y_pred, y)
   ```
6. Add the loss value to our tracking list
   ```
       losses.append(loss)
   ```
7. Print the current line of results
   ```
       print(f'epoch: {i:2} loss: {loss.item():10.8f}')
   ```
8. Gradients accumulate with every backprop. To prevent compounding we need to reset the stored gradient for each new epoch.
   ```
       optimizer.zero_grad()
   ```
9. Now we can backprop
   ```
       loss.backward()
   ```
10. Finally, we can update the hyperparameters of our model
    ```
        optimizer.step()
    ```

```
epochs = 50
losses = []
for i in range(epochs):
    i+=1
    y_pred = model.forward(X)
    loss = criterion(y_pred, y)
    losses.append(loss)
    print(f'epoch: {i:2}  loss: {loss.item():10.8f}  weight: {model.linear.weight.item():10.8f}  \
bias: {model.linear.bias.item():10.8f}')
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

# Plot the result

Now we'll derive y1 from the new model to plot the most recent best-fit line.

```
w1,b1 = model.linear.weight.item(), model.linear.bias.item()
print(f'Current weight: {w1:.8f}, Current bias: {b1:.8f}')
print()

y1 = x1*w1 + b1
print(x1)
print(y1)
```

```
plt.scatter(X.numpy(), y.numpy())
plt.plot(x1,y1,'r')
plt.title('Current Model')
plt.ylabel('y')
plt.xlabel('x');
```