ADAPT Pro – Topic 2:
Data Analysis and Modeling

Seminar for

J.P.Morgan

2023 ADAPT Training

Athena Data & Analytics Python Training

July 27-28, 2022

The Marquee Group

Leaders in Financial Modeling Since 2002

- We believe that spreadsheet-based financial models are the most important decision-making tools in modern finance
- We have developed a framework and discipline for model design, structure and development that leads to best-in-class, user-friendly financial models
- We help finance professionals use this framework to turn their models into powerful communication tools that lead to better, more effective decisions

The Marquee Group Offering

Instructor Led Training	Online Self Study Training	Modeling Solutions
✓ Instructors have real-world experience and a passion for teaching	✓ Industry leading self paced learning management system	✓ Services include: – Model Development – Model Re-builds – Model Reviews – Model Audits
✓ Topics include: Modeling, Valuation, Excel, Python	✓ Broad library with targeted job learning paths	✓ Clients include a wide range of companies in various industries
✓ Courses are interactive	✓ Over 80 hours of video-based instruction through: – Marquee Group Self Study – Training The Street Academy	
✓ Clients include banks, corporations, business schools and societies		

J.P. Morgan July 2023

THE MARQUEE GROUP
A TRAINING THE STREET COMPANY

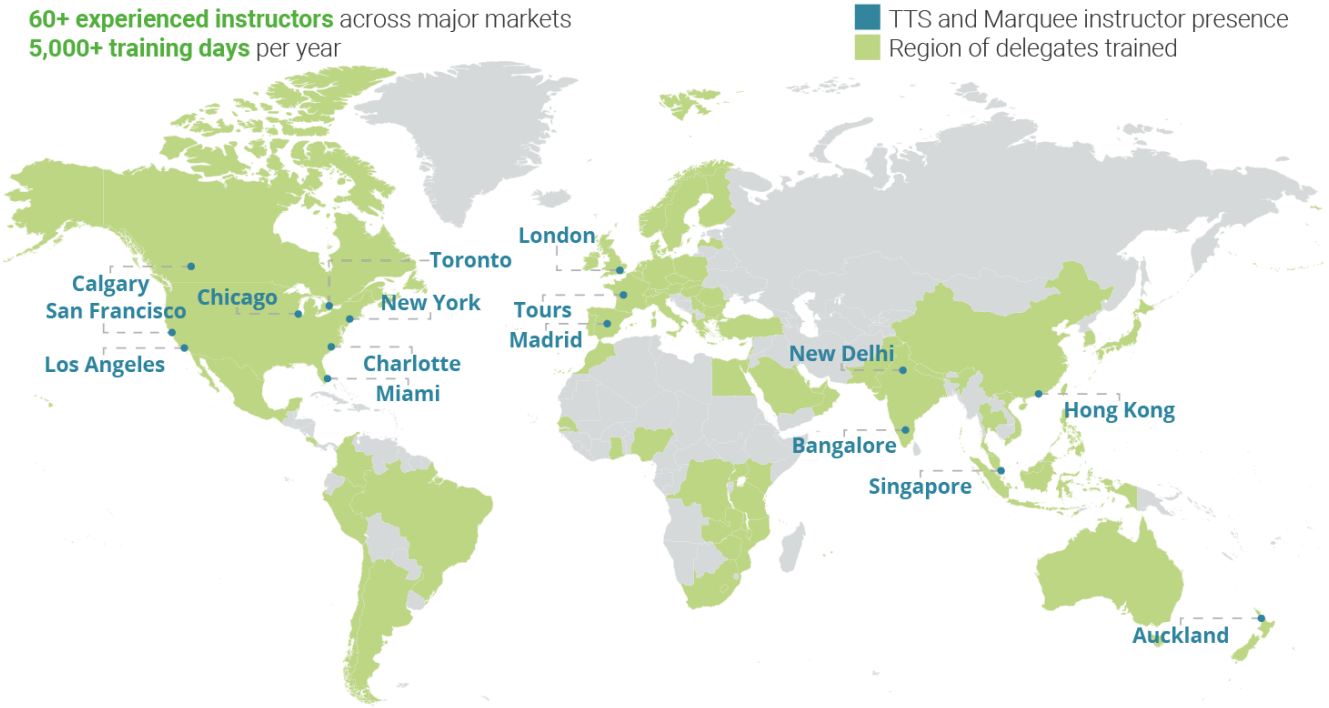
Training The Street

Acquired The Marquee Group in April 2023

- Training The Street ("TTS") is a leading professional development training provider headquartered in New York
- Paired with Marquee's world-class instructor team and content, the TTS team adds a wealth of learning assets, trainer capabilities and creates a unique global training provider

Enhanced Global Presence, Local Resources

60+ experienced instructors across major markets
5,000+ training days per year



Global Leader in Content and Capabilities

TRAINING THE STREET℠

Our Combined Advantage

Customer Focused
Full spectrum of service both inside and outside the classroom

Industry Experience
Proven track record since our founding which dates back to 1997

Instructor Quality
Practitioners with a passion for teaching and adjunct credentials

Expanded Content Capabilities
Content customized for finance professionals across several business areas

Cutting Edge Technology
Interactive, engaging content for practical, hands-on training

Unequalled Breadth of Content Expertise

Fundamental Content	Sector Capabilities Training	Data Sciences	Specialist/Functional Capabilities
Accounting	Commercial Real-Estate	Applied Excel	Corporate Credit Analysis
Corporate Valuation	FIG Analysis *(Banks & Insurance)*	Python	Cash Mgt & Treasury Services
Financial Modeling	Financial & Corporate Restructuring	SQL	ESG
M&A Modeling	Oil & Gas	Power BI	Investment Banking Overview
LBO Modeling	Project Finance	VBA	Investment Authorities
Capital Markets *(DCM | ECM Origination)*	Private Equity Investing & Analysis	Google Sheets	Private Co. Analysis
Financial Products / Global Markets (S&T)	Infrastructure		PowerPoint
Portfolio / Investments Analysis	Investment and Wealth Management		Data Storytelling
Applied Excel	Software *(SaaS)* Analysis		FP&A
	Venture Capital		Model Building Solutions
	Renewables		

J.P. Morgan July 2023

THE MARQUEE GROUP
A TRAINING THE STREET COMPANY

Table of Contents

Using *statsmodels*

Using *statsmodels*

- Statsmodels is a python package that provides functions to perform linear and time series analysis
  - Similar to R for linear regression and time series
- This lesson will focus on
  - Linear Regression (CAPM)
  - Time Series (ARMA)
  - Visualizing Analysis

```
# %% Packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import statsmodels.api as sm
import statsmodels.tsa as tsa
```

Using *statsmodels*

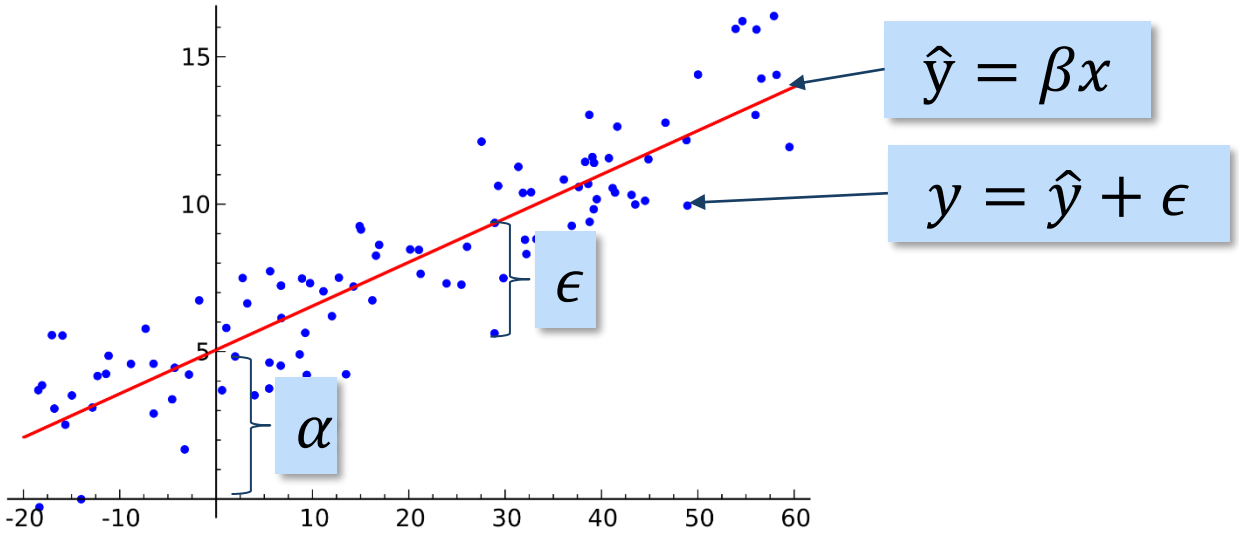
- Organizing data
  - It is best to organize the data into a single DataFrame
  - Verify the index is aligned, then handle missing data accordingly
- After the data is cleaned, copy the DataFrame into an array and if necessary scale the data and check for outliers
  - For numerical stability, try to keep numbers close to one
  - There are several techniques to handle outliers
    - Normalization: Transform the data to be zero mean and unit variance
    - Winsorizing: replace the outliers with the value contained at 90th, 95th or 99th percentile
    - Log Transform: take the log of the data

```
>>> sp500 = pd.read_csv('StockData/SP500.csv', header=0, index_col=0, parse_dates=True)
>>> aapl = pd.read_csv('StockData/aapl.csv', header=0, index_col=0, parse_dates=True)
# Need to cast the series as a DataFrame to merge
>>> data = pd.DataFrame(sp500['Adj Close']).merge(aapl['Adj Close’],
                                  left_index=True, right_index=True)
>>> data.columns = ['sp500', 'aapl']
>>> data[['sp500_rtns', 'aapl_rtns']] = np.log(data[['sp500’, 'aapl']]).diff()
>>> data[['sp500_rtns', 'aapl_rtns']] *= 100
>>> data.dropna(inplace=True)
```

Linear Regression with *statsmodels*

Linear Regression

- The simplest form of linear regression is finding the line of “best fit” between two variables, the dependent and independent variable
  - Traditionally the dependent variable is called endogenous and labelled as y
  - The independent variables are called exogenous and labelled as x
  - The error term is what is left over, not explained by the exogenous variable, and is labelled as ϵ
  - The general form equation is $y = \alpha + \beta x + \epsilon$
  - The slope, beta, is the sensitivity of the dependent variable to the independent variable
    - Typically thought of as a unit change in x results in a beta change of y
  - Best fit is found by minimizing the least squares objective function $\min_{\beta} \sum(y_i - \beta x_i)^2$



Linear Regression

- This simple two variable model can be expanded into a multivariate model
  - The equation doesn't change, $y = X\beta + \epsilon$, but now X is a matrix with multiple variables used to explain y including a column of ones for the intercept
- The following assumptions are made for the model
  - **The model is correctly specified:** a linear model is specified, thus the relationship in reality should be linear (not exponential, square etc.)
  - **Exogeneity**: $E[\epsilon|X] = 0$ the conditional mean of the error term is zero based on the exogenous variables. Be aware for this assumption that the objective function will make this true but its not necessarily true
  - **Linearly independent**: Full rank system, no perfect multicollinearity between variables
  - **Errors are uncorrelated and homoscedastic** : $E\left[\epsilon_i \epsilon_j \middle| X\right] = 0 \; for \; i \neq j$ and $E\left[\epsilon_i^2 \middle| X\right] = \sigma^2$, meaning the error terms are not changing with x and the variance is constant throughout the data (there is no value of x that has a smaller error variance)
  - **Error terms are normally distributed**: $\epsilon|X \sim N(0, \sigma^2)$, used to impose the asymptotic properties to determine the standard errors and confidence intervals on the betas

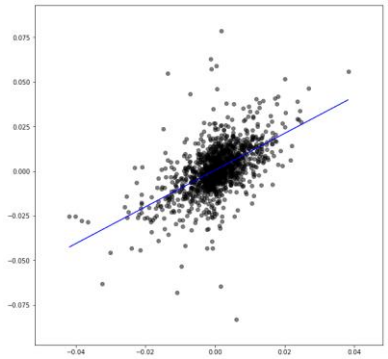
Linear Regression

- The CAPM can be estimated using linear regression
  - $E(r_i) = r_f + \beta(E(r_m) - r_f)$
  - Rearrange to make it easier to estimate: $E(r_i) - r_f = \beta(E(r_m) - r_f)$
- Syntax: sm.OLS(endo, exog)
  - **Endo**: the dependent variable, y or left hand side
  - **Exog**: the independent variable(s), X or right hand side
- An intercept is not included in this model
  - To add one use the sm.add_constant() function

```
# Performing a simple/modified CAPM, not going to subtract rf
>>> capm = sm.OLS(data['aapl_rtns'], data['sp500_rtns'])
# y, X -> apple, sp500 (endogenous, exogenous)
>>> results = capm.fit()

>>> print(results.params)
[1.03475198]
# This means that Apple has a beta of 1.0347 to the market (sp500)
```

Linear Regression

- To plot the linear regression, generate predicted values for Y, which in this example are the returns for Apple
  - Calculating $\hat{y} = \beta x$, or more specifically $\widehat{r_{aapl}} = \beta r_{mrkt}$



```
>>> data['aapl_hat'] = results.predict(data['sp500_rtns'])
# Plot the scatter plot and the linear regression
>>> plt.subplots(figsize=(10, 10))
>>> plt.scatter(data['sp500_rtns’],
                  data['aapl_rtns'], color='black', alpha=0.5)
>>> plt.plot(data['sp500_rtns’],
                  data['aapl_hat'], color='blue', linewidth=1)
>>> plt.show()
```

Linear Regression

- When a linear regression is performed the coefficients are a point estimate; the confidence interval must be reviewed to learn if they are significant
  - The statsmodels package will automatically provide summary statistics, using the .summary() method for the results object

```
>>> print(results.summary())
OLS Regression Results
==============================================================================
Dep. Variable:              aapl_rtns   R-squared:                       0.322
Model:                            OLS   Adj. R-squared:                  0.322
Method:                 Least Squares   F-statistic:                     598.1
Date:                Thu, 02 May 2019   Prob (F-statistic):          2.28e-108
Time:                        17:47:34   Log-Likelihood:                -1979.4
No. Observations:                1258   AIC:                             3961.
Df Residuals:                    1257   BIC:                             3966.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
sp500_rtns     1.0348      0.042     24.456      0.000       0.952       1.118
==============================================================================
Omnibus:                      209.489   Durbin-Watson:                   1.933
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             3680.503
Skew:                           0.080   Prob(JB):                         0.00
Kurtosis:                      11.378   Cond. No.                         1.00
==============================================================================
```

Linear Regression

- Bootstrapping is a method to generate a distribution for coefficients using the data set (non-parametric or empirical)
  - The data set is permutated (with replacement) ‘n’ times. The total number of observations do not change, and the pairs are maintained
  - For each permutation iteration, the coefficients are estimated using OLS.
    - This will generate ‘n’ estimates for the coefficients
    - Store all the coefficient estimates to create an empirical distribution
  - The intuition is that the data set is a representative sample of the population, so draws from the data set (samples) are equivalent to draws from the population.

Dataset	
y	x
5	3
3	5
4	12
12	9
6	3
2	4
4	4
3	2
2	10
1	2

Bootstrap (Iteration 1)	
y	x
5	3
4	12
4	12
2	4
6	3
3	2
2	10
1	2
2	10
3	2

Bootstrap (Iteration 2)	
y	x
3	5
3	5
2	4
1	2
3	2
2	4
6	3
1	2
2	10
12	9

Linear Regression

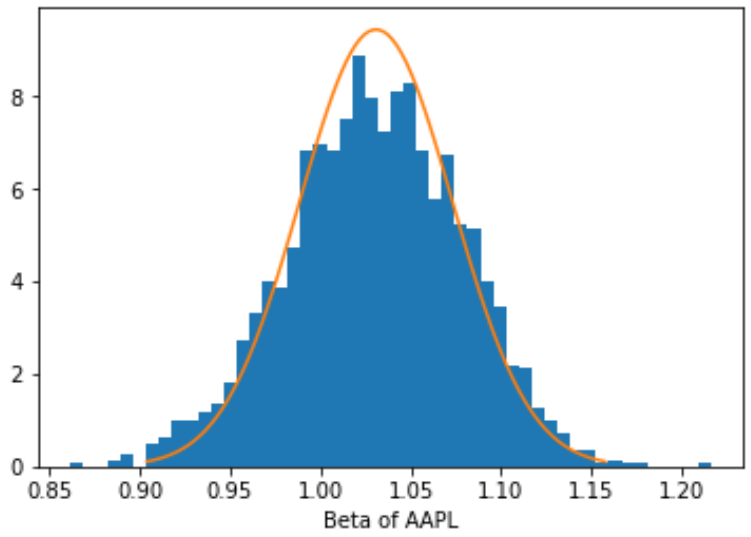
- To automate this process, create a function that takes the data as x and y, along with the number of boot strap iterations
  - **X**: for CAPM will be the market returns
  - **Y**: for CAPM will be the stock returns
  - **Nboot**: note this should always end in 9 if the original estimate is added back in. In the example below nboot=999, and the original estimate is also calculated which will return a total of 1000 estimates for alpha and beta
    - Original estimate refers to the regression coefficient using the sample

```
>>> def bootcoef(X, y, nboot=999):
#Range of sample index, this allows for pairs sampling
...         inds = np.arange(len(X))
...         temp_reg = sm.OLS(y, X).fit()
...         bootresults = temp_reg.params.tolist()
...         for i in range(nboot):
...         # generate a random set of integers to select a bootstrap sample from data
...                     bootint = np.random.choice(inds, len(inds))
...                     Xboot = X[bootint]
...                     yboot = y[bootint]
...                     temp_reg = sm.OLS(yboot, Xboot).fit()
...                     bootresults.extend(temp_reg.params.tolist())
...         return(bootresults)

>>> np.random.seed(42)
>>> bootCAPM = bootcoef(data['sp500_rtns'].values, data['aapl_rtns'].values, nboot = 1999)
```

Linear Regression

- The boot strapped results can be plotted to see the distribution, along with overlaying the parametric distribution, which in this case is the normal distribution.
  - The mean and variance of the coefficient is required to plot the parametric distribution, in this case our beta
  - These values can be pulled from the model
    - .params returns a list of parameters
    - .cov_params returns a list of var/cov of the parameters
    - .conf_int() provides 95 percentile confidence interval

```
#Display the Histogram and Normal Distribution
(parametric)
>>> beta_mean = results.params.values[0]
>>> beta_var = results.cov_params().values[0]
>>> beta_std = np.sqrt(beta_var)

>>> x_numbers = np.linspace(beta_mean - 3*beta_std,
                        beta_mean + 3*beta_std, 100)
>>> plt.hist(bootCAPM, bins=51, density=True)
>>> plt.plot(x_numbers, norm.pdf(
             x_numbers, beta_mean, np.sqrt(beta_var)))
>>> plt.xlabel('Beta of AAPL')
>>> plt.show()
```



Time Series with *statsmodels*

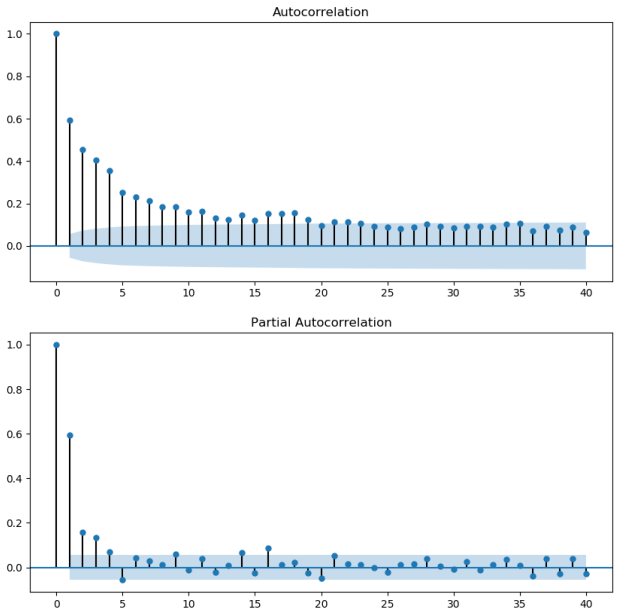
Time Series

- Time Series Analysis is used to find a relationship through time, and forecast (predict) the future based on historical data
- Time Series can be thought of as a linear regression on lagged variables
  - **AR**: auto regression, how many lags.
  - **I**: Number of differences (y(t) – y(t+1))
  - **MA**: Moving average window
- Benefit of not using a packaged ARIMA model
  - Can control the specification better
  - Built in drift handling
  - Easier to construct hierarchical models
- General AR Model Specification:
  - $y_{t+1} = \delta + \phi y_t + \epsilon_t$
  - Long-run average: $\frac{\delta}{1-\phi}$
  - The model is considered stationary if $\phi < 1$, otherwise the process can 'blow-up'. Some more advanced modeling techniques can relax this condition for a few periods.

Time Series

- Typically, the ACF (autocorrelation function) and the PACF (partial autocorrelation function) are reviewed before modeling to get an idea of how many lags (AR and MA) to include in the model
- A simplified explanation is that
  - ACF uses all the data, it is useful to determine how many MA lags to include
  - PACF uses data incrementally, it is useful to determine how many AR lags to add

```
>>> fig = plt.figure(figsize=(10,10))
>>> ax1 = fig.add_subplot(2,1,1)
>>> fig = sm.graphics.tsa.plot_acf(
          ts500['vol'].values, lags=40, ax=ax1)

>>> ax2 = fig.add_subplot(2,1,2)
>>> fig = sm.graphics.tsa.plot_pacf(
          ts500['vol'].values, lags=40, ax=ax2)
>>> fig.show()
```



Time Series

- .tsa.ARMA(data, (p,q)).fit(disp=False, trend='c')
  - **Data**: the time series data being modelled
  - **(p,q):** p is the number of AR lags, and q is the number of MA lags to include in the model
  - **Disp**: if True will display the convergence information
  - **Trend**: 'c' will include a constant (trend), 'nc' will remove the constant (no trend)
- Python will flag an error if the model is not stationary
  - It is best to code a try/except to handle the error if running a production script

```
>>> arma = sm.tsa.ARMA(ts500['vol'], (1,5)).fit(disp=False)
>>> print(arma.summary())


>>> arma = sm.tsa.ARMA(ts500['vol'], (1,22)).fit(disp=False)
ValueError: The computed initial AR coefficients are not stationary
You should induce stationarity, choose a different model order, or
you can pass your own start_params.
```

Time Series

- .forecast(steps=1, exog=None, alpha=0.05)
  - **Steps**: number of out-of-sample steps to forecast
  - **Exog**: used to pass exogenous variables if the model is an ARMAX model, where the X is for additional non-lagged variables
  - **Alpha**: the level for which to produce the confidence intervals
- This function will return three results:
  - An array containing the forecast
  - An array containing the standard error (stderr) for the forecast
  - An array containing the confidence intervals for the forecast

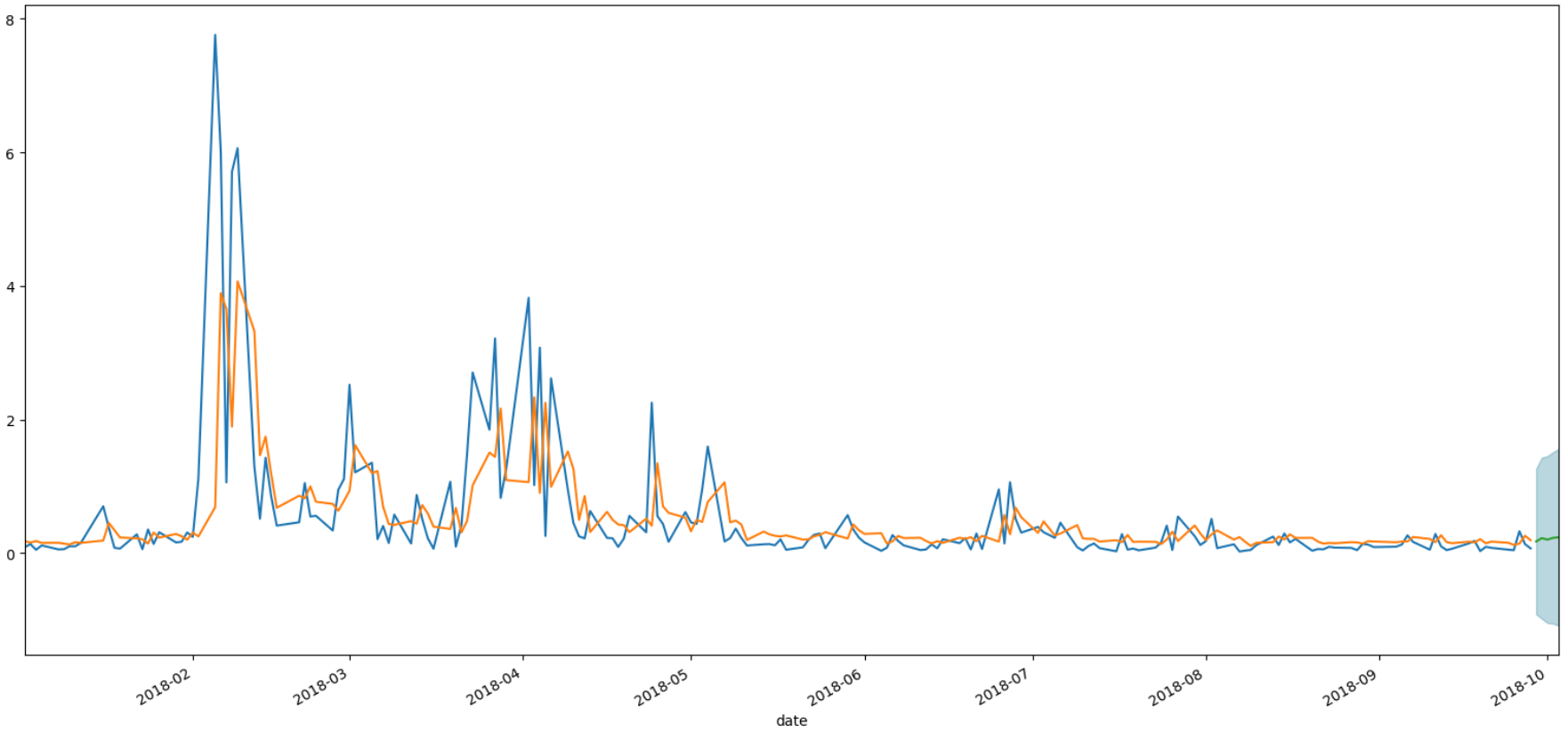
```
>>> y_T, y_T_stderr, y_T_conf_int = arma.forecast(steps=5)


# The one step ahead forecast to see the structure
>>> arma.forecast(steps=1)
(array([0.17562662]), array([0.55567033]), array([[-0.91346723,
1.26472046]]))
```

Time Series

- Plotting the data, estimation ($\hat{y}$) and forecast
  - Extract the fitted values ($\hat{y}$) from the model using .fittedvalues
    - The index passed from the DataFrame will be carried through
  - Create a forecast and combines the results into a DataFrame
    - Merge the forecast and confidence intervals into a DataFrame
    - The index will need to be extended before combining
  - Plot the data, model and forecast
    - Use the .fill_between() method to create the shaded confidence band on the plot

```
# Let's plot the 2018 data with the ARMA model using the built in functions
>>> y_hat = arma.fittedvalues
>>> y_T, y_T_stderr, y_T_conf_int = arma.forecast(steps=5)

>>> tsfore = pd.DataFrame(y_T, columns=['y_T']).join(pd.DataFrame(y_T_conf_int,
                      columns=['y_T_low', 'y_T_upper']))

# Need to set index
>>> last_day = max(ts500.index)
>>> tsfore['date'] = [pd.to_datetime(last_day + pd.offsets.BDay(i+1)) for i in range(5)]
>>> tsfore.set_index('date', inplace=True, drop=True)

# Now plot the data plus the forecast
>>> ts500.loc['2018':, 'vol'].plot()
>>> y_hat.loc['2018':].plot()
>>> tsfore['y_T'].plot()
>>> plt.fill_between(tsfore.index, tsfore['y_T_low'], tsfore['y_T_upper’],
                      color = '#539caf', alpha = 0.4)
>>> plt.show()
```

Time Series



Using *SciPy* - Optimization

Using *SciPy* – Optimization

- The SciPy library is a core python package providing numerical functions and is the parent package for Numpy
- Includes many common numerical functions, including Integration, Optimization, Interpolation and many others
- Many of these functions are commonly used in Finance applications, focus will be placed on the optimization function
  - Optimization is a common problem in finance, especially in portfolio selection
  - The minimization function will be used to find the weights of the optimal portfolio

```
>>> from scipy.optimize import minimize
>>> from scipy.optimize import LinearConstraint
```

Portfolio Optimization

Portfolio Optimization – Overview

- Brief overview of portfolio theory
  - A collection of investment assets is called a portfolio. The weights of the portfolio are calculated by their respective values ($\sum w_i = 1$).
    - For example, consider two assets with $3000 in asset 1 and $2000 in asset 2. The weight of asset 1 is 0.6 and the weight of asset 2 is 0.4.
  - The expected return of the portfolio is the weighted average of the expected return of the individual assets.
    - For example, if asset 1 has an expected return of 5% and asset 2 has an expected return of 10% the portfolio expected return would be 7% using the weights above.
  - Why form a portfolio? If asset 2 is expected to make more than asset 1, why not invest fully in asset 2?
    - The purpose is to diversify risk, think of the phrase “don’t put all your eggs in one basket”
    - To illustrate this, consider asset 1 has a standard deviation 10% and asset 2 has a standard deviation of 20%, with a correlation of 0.3 between the two, the portfolio variance is

$$\sigma_p^2 = w_1^2\sigma_1^2 + w_2^2\sigma_2^2 + 2w_1w_2\rho_{1,2}\sigma_1\sigma_2$$
$$\sigma_p^2 = 1.288\%$$
$$\sigma_p = 11.349\%$$

  - The investor is able to increase their portfolio return by 2% over just investing in asset 1 while only adding 1.4% to the risk of the portfolio.

Portfolio Optimization – Overview

- The classical portfolio selection problem is to minimize risk (variance/standard deviation) while maximizing return
  - One constraint occurs by formulation of the problem, the desired return
  - The simplest constraint is that the portfolio weights have to sum to one not to break equilibrium rules
  - Other constraints can be added, such as no short selling ($w \geq 0$), or not have more than x% in a security ($w \leq 0.05$)

- Our problem is as follows
  - $\min_{w} w^T V w \;\; s.t. \; 1 = \sum w_i \; and \; E[r] = \sum (w_i r_i)$

- The objective function, portfolio variance, is the same as in the example but now in matrix notation.
  - Remember that $\rho_{1,2}\sigma_1\sigma_2 = \sigma_{1,2}$ where $\sigma_{1,2}$ is the covariance between assets 1 and 2

Portfolio Optimization – Minimize

- Minimize(func, x0=none, args=(), constraints=(), bounds=())
  - **func**: the objective function, which should be minimized. Can be a defined function or a lambda function
  - **x0**: the starting values for x. For complicated functions the starting conditions can have an effect on the result
  - **args**: passed as a tuple to the func
  - **constraints**: a tuple of the defined constraints for the problem
  - **bounds**: a sequence or tuple of (min, max) tuple pairs for each element in x

```
 #Function returns the variance of the portfolio given weights and
covariance matrix
>>> def optw(w, *args):
...    return(np.matmul(np.matmul(w,V),w))

>>> weight = minimize(optw, w, (V),
        constraints=(constraint_1,constraint_2),
        options={'maxiter':1000})
```

Portfolio Optimization – Constraints

- LinearConstraint(A, lb, ub)
  - **A**: is a matrix/array which has the coefficients for the constraint equation
    - Remember that no coefficient is equivalent to having a 1
  - **Lb**: the lower limit of the constraint
  - **Ub**: the upper limit of the constraint
    - Set Lb equal to Ub for equality constraint
- The constraints are specified in the form $lb \leq A \cdot X \leq ub$
- In older versions of SciPy, the constraints need to be set up as a dictionary

```
# constraint for the weights
>>> constraint_1 = LinearConstraint(A, 1, 1)

# for the return constraint, the asset returns are the coefficients
>>> constraint_2 = LinearConstraint(mu, expect_return, np.inf)

# for older versions of scipy use:
>>> constraint_1 = {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}
>>> constraint_2 = {'type': 'ineq', 'fun': lambda x: np.sum(x * mu ) -
expect_return}
```

Portfolio Optimization – Constraint 1

- How to go from $1 = \sum w_i$ to python code
  - The constraints are specified in the form $lb \leq A \cdot x \leq ub$
  - The dot operator takes two matrices, performs element wise multiplication and then sums them. In notation,

$$\mathrm{A} \cdot X = \sum a_i x_i = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n$$

  - Lining up the two on top of each other

$$1 = \sum w_i$$
$$A \cdot X = \sum a_i x_i$$

  - We can see that $A \cdot X = 1$, X is the matrix of unknown weights.
  - If a variable, such as w_i, does not have a written coefficient in front of it, it actually has a 1. This means that in this case the A matrix is comprised of all ones
  - Since the weights must add up to 1, the lower and upper bounds are specified to be 1, resulting in $1 \leq 1 \cdot w \leq 1$

```
# constraint for the weights, n = # of stocks to be analyzed
>>> A = np.matrix([1]*n)
>>> constraint_1 = LinearConstraint(A, 1, 1)
```

Portfolio Optimization – Constraint 2

- How to go from $E[r] = \sum(w_i r_i)$ to code
  - The constraints are specified in the form $lb \leq A \cdot x \leq ub$
  - Following the same logic as above, stacking the formulas

$$E[r] = r^T w = \sum(r_i w_i)$$
$$A \cdot X = \sum(a_i x_i)$$

  - The difference now is that there is a coefficient in front of the unknown weight variable, the return of the asset. This means that the A matrix is the matrix of expected asset returns.
  - The left hand side, $A \cdot X$, is the specified return the investor wishes to achieve on the portfolio. This is considered to be the lower bound, the minimum the investor wishes to achieve. The upper bound is left unconstrained, infinity, because if a higher return of the same variance is possible it should be attained.
  - This gives $E[r] \leq r \cdot w \leq \infty$

```
# series of average returns of each stock (expected returns)
>>> mu = np.mean(rtns, axis=0) * 100
# constraint for returns
>>> constraint_2 = LinearConstraint(mu, expect_return, np.inf)
```

Portfolio Optimization – Constraints as Dictionaries

- In previous versions of SciPy (packages with versions <1.0), the LinearConstraint function did not exist
- Instead, each constraint is set up as a dictionary with the following keys and values:
  - **type:** 'eq' for equality and 'ineq' for inequality
    - Equality constraint means that the constraint function result needs to equate to zero, while inequality means the function result needs to be non-negative
  - **fun**: the function defining the constraint
    - Typically set up as a lambda function
    - The inputs to the function are the values of x in the optimization's minimize function
- Specifying the constraints in the older format:
  - Constraint 1: $1 = \sum w_i$ needs to be re-arranged as an equality of 0, as $\sum w_i - 1 = 0$
  - Constraint 2: $E[r] = \sum(w_i r_i)$ needs to be re-arranged as an inequality, $\sum(w_i r_i) - E[r] >= 0$

```
# series of average returns of each stock (expected returns)
>>> mu = np.mean(rtns, axis=0) * 100
>>> constraint_1 = {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}
>>> constraint_2 = {'type': 'ineq', 'fun': lambda x: np.sum(x * mu )
- expect_return}
```

Portfolio Optimization – Bounds

- bounds = tuple of (min, max) pairs
- The bounds argument of the minimize function provides minimum and maximum limits for the x variables being sensitized in the optimization
- For a portfolio optimization, bounds of (0,1) are usually provided to restrict for short selling and ensure none of the asset weightings exceeds 100%
  - The max is also sometimes restricted to a maximum exposure in the portfolio (e.g. max 25% weighting)

```
# bounds for the weights to restrict short-selling
>>> n = mu.shape[0] #number of assets in portfolio
>>> bound = (0.0,1.0) # (min, max) boundaries for any one of the assets
>>> bounds = tuple(bound for asset in range(n)) #tuple for all assets

# updating the minimize function:
>>> weight = minimize(optw, w, (V),
        constraints=(constraint_1,constraint_2),
        bounds=bounds,
        options={'maxiter':1000})
```

Portfolio Optimization - Results

- The results are stored in a result object
  - **.x**: stores the solution vector in a NumPy array
  - **.success**: True/False if the solver was able to find a solution
  - **.nit**: The number of iterations required to find a solution
  - **.message**: The cause of termination

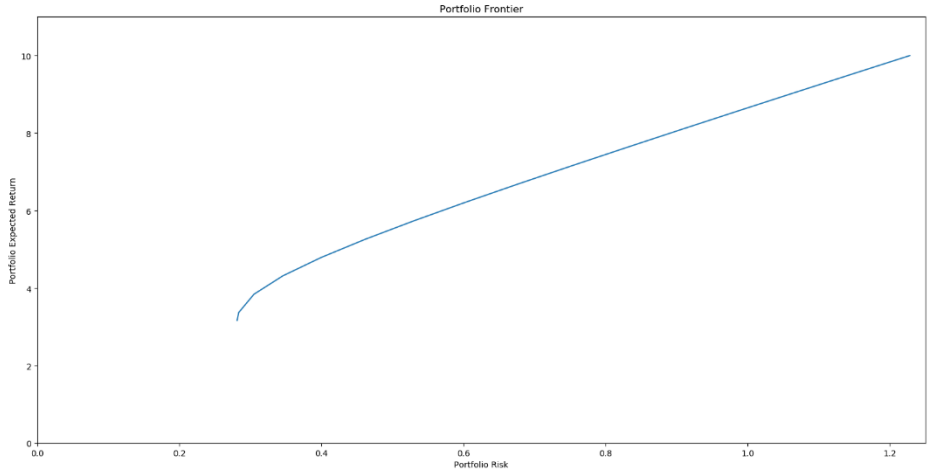
```
>>> weight
     fun: 0.18184203971590196
     jac: array([-0.10255267, -0.08567294, -0.05414635, -0.06006911, -0.09526219,
        0.03190438,  0.05650454,  0.02412239, -0.09328883,  0.02637638,
        0.15002657,  0.11958694,  0.13675068,  0.17859602,  0.31597819,
        0.26219905,  0.27370915,  0.27453952,  0.37605105,  0.22642007,
        0.36682575,  0.37303495,  0.31686172,  0.47696703,  0.51525565])
 message: 'Optimization terminated successfully.'
    nfev: 781
     nit: 28
    njev: 28
  status: 0
 success: True
       x: array([-0.11165125,  0.00122382, -0.01348225,  0.00306191, -0.03130665,
        0.06896753, -0.01649814, -0.0042416 ,  0.0132445 ,  0.03334851,
       -0.13659692,  0.04308279,  0.00444776,  0.04963469,  0.00505731,
        0.32991776,  0.15612027,  0.03729035, -0.00815732,  0.04175576,
        0.32737215,  0.17916443, -0.02932554,  0.02308353,  0.03448659])
```

Portfolio Optimization – Frontier

- The portfolio frontier can be plotted (asset or efficient frontier if talking about the market)
  - Loop through a range of returns to calculate then find the minimum portfolio variances. Some returns are not achievable, so use the .success attribute to filter
  - Find the portfolio expected return and variances
  - Plot the return versus standard deviation (risk)

```
>>> weights = []

>>> for port_return in np.linspace(1,10, 20):
...       constraint_2 = LinearConstraint(mu, port_return, np.inf)
...       weight = minimize(optw, w, (V),
...                         constraints=(constraint_1,constraint_2),
...                         options={'maxiter':1000})
...       if weight.success == True:
...                 weights.append(weight.x)
```

Portfolio Optimization – Frontier




```
# Find the portfolio returns and
variences
>>> portmu = []
>>> portvar = []

>>> for w in weights:
        portmu.append(w @ mu)
        portvar.append(w @ V @ w)

>>> portstd = np.sqrt(portvar)
```

```
>>> plt.plot(portstd, portmu)
>>> plt.title('Portfolio Frontier')
>>> plt.xlim([0,1.25])
>>> plt.ylim([0,11])
>>> plt.xlabel('Portfolio Risk')
>>> plt.ylabel('Portfolio Expected
Return')
>>> plt.show()
```

Using *scikit-learn (sklearn)*

Using scikit-learn (sklearn)

- Scikit-learn (sklearn) is a large data analysis packaged used for applying machine learning algorithms
  - Typically, only the necessary modules are imported in order to save memory and time
  - The package contains functions for
    - Classification, regression, cluster detection, dimensionality reduction, data preprocessing and model selection.
- Focus will be on K-Means Clustering to group data sets into clusters

```
>>> from sklearn.cluster import KMeans
```

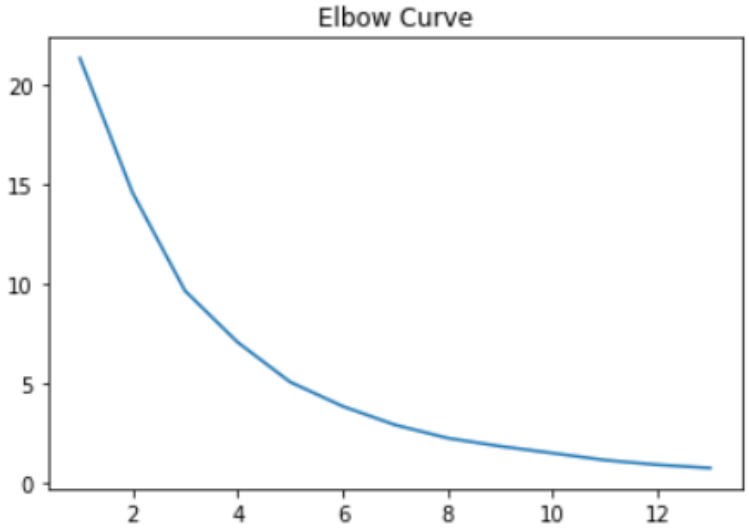
Machine Learning Algorithms

- Python has become very popular in the data science community due to the large amount of Machine Learning and AI algorithms available through third party packages
- **Scikit-learn** is the most commonly used package for Machine Learning and has algorithms for the following applications:
  - **Classification**: identifying which category an object belongs to
    - E.g. after training a model what is spam and what is not, the classifier model will “classify” new emails
  - **Regression**: predicting continuous-valued attributes associated with independent variables
    - E.g. predicting returns of portfolio based on certain factors (market risk premium, size premium, etc.)
  - **Clustering:** automatic grouping of similar objects into sets
    - E.g. allocating customers into different categories based on spending habits and other characteristics
- Clustering is a type of Unsupervised Machine Learning algorithm
  - Unsupervised Learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables
  - We can derive this structure by clustering the data based on relationships among the variables in the data
  - Example: provide a data set of thousands of trading customers and find a way to automatically group the clients into groups that are similar or related by different variables, such as the # of trades, commissions generated, $ volume traded, etc.

K-Means Clustering

K-Means Clustering

- There are many clustering algorithms available in the *sklearn* package and a good overview and comparison can be found at:
  - https://scikit-learn.org/stable/modules/clustering.html
- The K-Means algorithm is one of the most common cluster algorithms
  - The algorithm clusters data by trying to separate samples in “n” groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares.
  - When using the algorithm in python, the number of clusters needs to be specified.
  - For further reading on how the algorithm categorizes data into cluster, please see:
    - https://scikit-learn.org/stable/modules/clustering.html#k-means
- One of the major assumptions when using K-Means is picking the optimal number of clusters
  - The “Elbow Method” is one method of finding the number of clusters and consists of looping through different scenarios and looking at the SSE (Sum of Squared Errors) then plotting the SSE's on a graph for each # of clusters picked.



K-Means with sklearn

- Documentation from version 0.20:
  - https://scikit-learn.org/0.20/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans
- KMeans(n_clusters=8)
  - If n_clusters is not set, the default is 8 clusters
- KMeans().fit(X)
  - Returns a fitted model object
  - X: data to be decomposed
- The result object has several attributes that can be accessed
  - .cluster_centers_
    - This provides an array of the x, y coordinates of the centroids of each cluster found
  - .labels_
    - An array of the cluster number for each data point that was analyzed
  - .inertia_
    - Sum of squared distances of samples to their closest cluster center

```
>>> X =  mu_var_rtns.values #Converting std and returns into nummpy array for the model
>>> kmeans = KMeans(n_clusters = 5).fit(X)
>>> centroids = kmeans.cluster_centers_
>>> princomp = PCA().fit(simrtns)
```

Cluster of Securities Based on Risk/Return Profile

- In this example, securities are grouped into clusters based on their expected returns and variance
- Given a DataFrame of daily returns of multiple securities called *rtns*, the following code creates the X variable using the mean returns and standard deviations of each security.

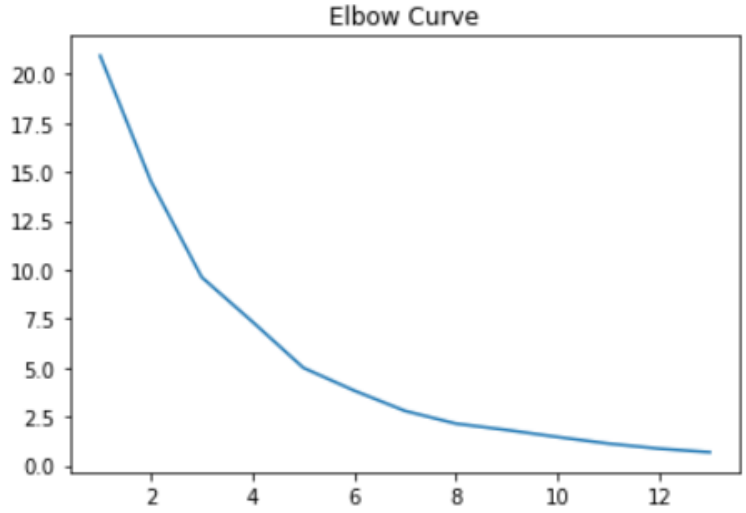
Date	AAPL	AXP	BA	CAT	CSCO	CVX	DIS	DWDP	GE	GS	...	NKE
2013-11-30	0.070053	0.048900	0.032485	0.014875	-0.058067	0.029244	0.028430	-0.010388	0.019893	0.053679	...	0.044614
2013-12-31	0.008902	0.057459	0.016685	0.073404	0.055530	0.020173	0.096445	0.144879	0.059895	0.049248	...	-0.000072
2014-01-31	-0.107697	-0.060516	-0.082277	0.047728	-0.015723	-0.106317	-0.049607	0.025000	-0.103461	-0.074128	...	-0.073627
2014-02-28	0.057511	0.073630	0.035024	0.032584	-0.005021	0.042322	0.112932	0.070314	0.022385	0.017620	...	0.081430
2014-03-31	0.019953	-0.013694	-0.026606	0.024750	0.028441	0.031041	-0.009157	0.005060	0.016490	-0.015620	...	-0.056705

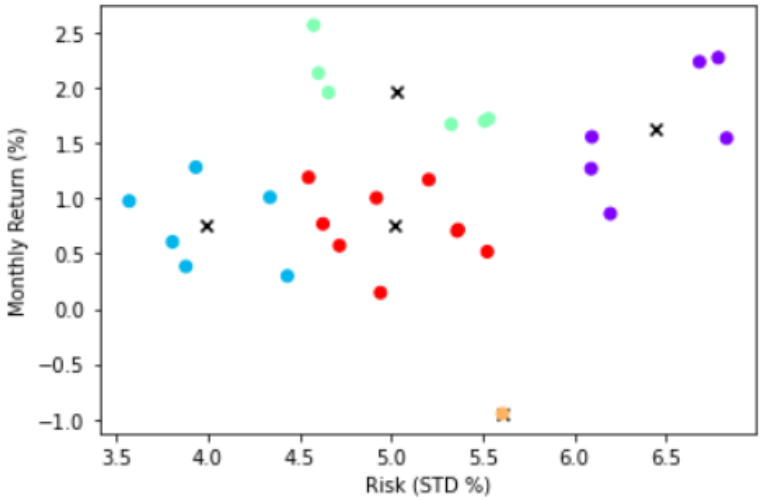
	risk	return
AAPL	6.787209	2.276223
AXP	5.366346	0.718297
BA	6.684699	2.238272
CAT	6.832630	1.546707
CSCO	5.533057	1.723833
CVX	5.524930	0.518233
DIS	5.205335	1.170476
DWDP	6.092664	1.270462
GE	5.608769	-0.950579
GS	6.197503	0.862902

```
# create five factor portfolios
mu_rtns = rtns.mean() * 100
var_rtns = rtns.var() * 100
std_rtns = rtns.std() * 100

#Storing the Monthly Average Returns and STD in a new
dataframe
mu_var_rtns = pd.concat([std_rtns, mu_rtns], axis = 1)
mu_var_rtns.columns = ['risk','return']
mu_var_rtns
```

```
# Running the k-means model
X =  mu_var_rtns.values
kmeans = KMeans(n_clusters = 5).fit(X)
centroids = kmeans.cluster_centers_
categories = kmeans.labels_
mu_var_rtns['Cluster'] = kmeans.labels_
```

K-Means Cluster Visualizations






```
sse = []
for k in range(2,15): #number of clusters
    kmeans = KMeans(n_clusters = k)
    kmeans.fit(X)
    sse.append(kmeans.inertia_)

#Quick plot of Elbow Curve
plt.plot(range(1,len(sse)+1), sse)
plt.title("Elbow Curve")
plt.show()
```

```
kmeans = KMeans(n_clusters = 5).fit(X)
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:,0], centroids[:,1], c
= 'black', marker='x')
plt.scatter(X[:,0],X[:,1],
            c = kmeans.labels_,
            cmap ="rainbow")
plt.ylabel("Monthly Return (%)")
plt.xlabel("Risk (STD %)")
plt.show()
```

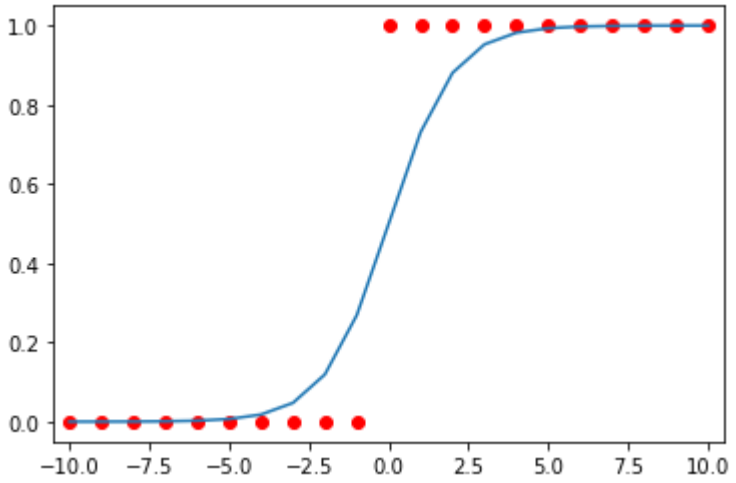
Logistic Regression

Logistic Regression – Overview

- Logistic Regression is used for classification type problems which have a discrete outcome
- Examples
  - Pass/Fail
  - Tall/Short
  - Default/No-Default
  - Approved/Denied

- Logistic regression works by taking inputs from predictive variables and transforming them to a probability between 0 & 1
- This is done using the following equation:

$$p(x) = \frac{1}{1 + \exp -(\beta_0 + \beta_1 x)}$$



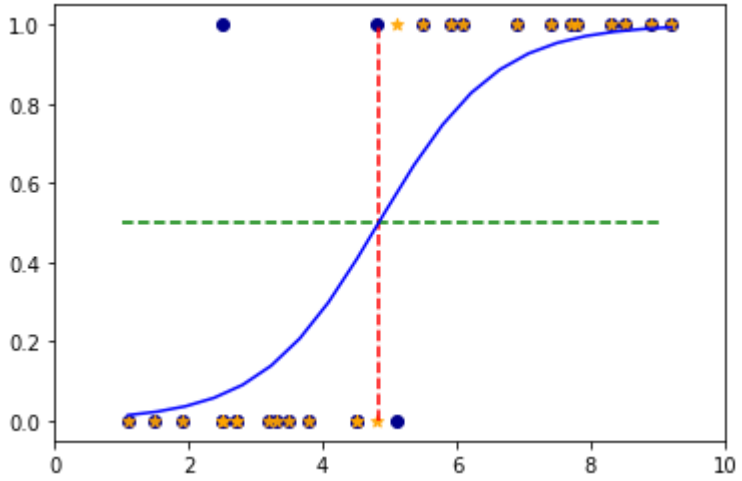
Logistic Regression – Cost Function

- The goal is to find an algorithm that is able to predict the class
  - The parameters (β) are determined through estimation (in ML this is known as training)
  - The estimation is done using maximum likelihood algorithm to optimize

$$\min_{\beta} \sum_{i=1}^{n} \left(-y_i \log\left(p(X_i)\right) - (1 - y_i) \log\left(1 - p(X_i)\right)\right)$$

  - If y_i is 0, and $p(X_i)$ is far from 0 (i.e. wrong prediction), the likelihood algorithm will produce a very large positive number (log of 0 is negative infinity); inversely if $p(X_i)$ is close to 0 (i.e. correct prediction), the likelihood algorithm will produce a value close to 0 (log of 1 is 0)
  - If y_i is 1, and $p(X_i)$ is close to 0 (i.e. wrong prediction), the likelihood algorithm will produce a very large positive number (log of 0 is negative infinity); inversely if $p(X_i)$ is close to 1 (i.e. correct prediction), the likelihood algorithm will produce a value close to 0 (log of 1 is 0)

- Once the model is trained, it can be used for prediction
  - The function is determined by the parameters calibrated during training
  - The threshold probability between categories is 0.5



Logistic Regression – Model Function

- LogisticRegression(penalty='l2', random_state=None, solver='lbfgs')
  - **Penalty**
    - 'none': no penalty is added
    - 'l2': default, l2 penalty is added to cost function
    - 'l1': l1 penalty is added to cost function
    - 'elasticnet': l1 and l2 penalty are added to cost function
  - **Random_state**
    - Seed to shuffle data for 'sag', 'saga' or 'liblinear' solvers
  - **Solver**
    - Algorithm used to optimize the cost function. Note that not all penalties work with all solvers
    - 'newton-cg' - ['l2', 'none']
    - 'lbfgs' - ['l2', 'none']
    - 'sag' - ['l2', 'none']
    - 'saga' - ['elasticnet', 'l1', 'l2', 'none']
    - 'liblinear' - ['l1', 'l2']
      - Limited to one vs rest type problems (not multiclassification)

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0)
clf.fit(X, y)
```

Logistic Regression – Function Penalties

- Penalty Functions

$$\min_{\beta} \sum_{i=1}^{n} \left(-y_i \log\left(p(X_i)\right) - (1 - y_i) \log\left(1 - p(X_i)\right)\right) + r(\beta)$$

Penalty Type	$r(\boldsymbol{\beta})$
'none'	0
'l1'	$\left\|\|\beta\|\right\|_1$
'l2'	$\left\|\|\beta\|\right\|_2^2 = \frac{1}{2}(\beta^T \beta)$
'elasticnet'	$\rho\left\|\|\beta\|\right\|_1 + \frac{1-\rho}{2}\left\|\|\beta\|\right\|_2^2$

Logistic Regression – Confusion Matrix

- confusion_matrix(y_true, y_pred, …)
  - **Y_true**: Array of actual outcomes
  - **Y_pred**: Array of predicted outcomes from model
- Notes
  - Provides the number of observations that are classified correctly and incorrectly
  - Observations on the diagonal are correctly classified

		Predicted	
		0	**1**
Actual	**0**	True Negative (TN)	False Positive (FP)
	1	False Negative (FN)	True Positive (TP)

Logistic Regression – Classification Report

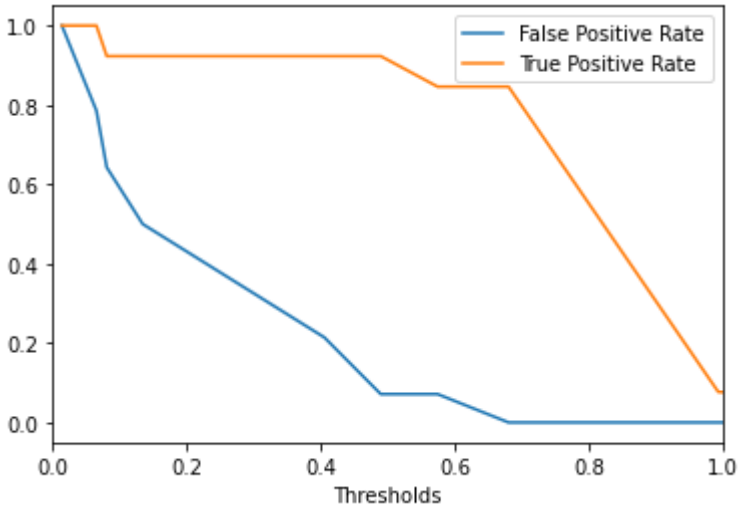
- classification_report(y_true, y_pred, …)
  - **Y_true**: the actual target values
  - **Y_pred**: the predicted target values from the classifier using .predict()

	Precision	Recall	F1-Score	Support
0	$\frac{TN}{TN+FN}$	$\frac{TN}{TN+FP}$	$2*\frac{Precision*Recall}{Precision+Recall}$	# of obs in class
1	$\frac{TP}{TP+FP}$	$\frac{TP}{TP+FN}$	$2*\frac{Precision*Recall}{Precision+Recall}$	# of obs in class
Accuracy			$\frac{TP+TN}{TP+TN+FP+FN}$	# of obs
Macro avg	$\frac{Precision_0+Precision_1}{2}$	$\frac{Recall_0+Recall_1}{2}$	$\frac{F1_0+F1_1}{2}$	# of obs
Weighted avg	$w_0Precision_0+w_1Precision_1$	$w_0Recall_0+w_1Recall_1$	$w_1F1_0+w_2F1_1$	# of obs

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
print(confusion_matrix(y, clf.predict(X)))
print(classification_report(y, clf.predict(X)))
```

Logistic Regression – ROC Curve

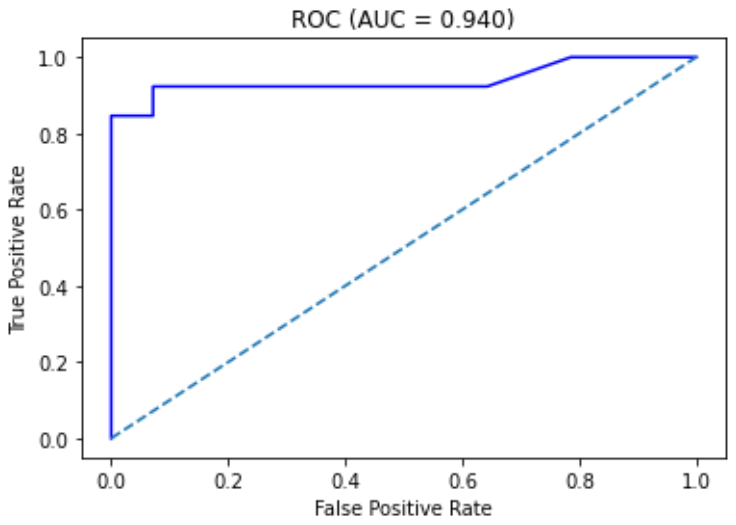
- roc_curve(y_true, y_score, …)
  - **Y_true**: actual results
  - **Y_prob**: probability of being True (1) class
  - Returns: false positive rate, true positive rate, thresholds




```
from sklearn.metrics import roc_curve, auc
y_prob = clf.predict_proba(df['AnnualSalaryStd'].values.reshape(-1,1))[:, 1]
fpr, tpr, thresholds = roc_curve(df['Purchased'].values, y_prob)
plt.plot(thresholds, fpr)
plt.plot(thresholds, tpr)
plt.xlabel("Thresholds")
plt.legend(['False Positive Rate', 'True Positive Rate'])
plt.xlim(0,1)
```

Logistic Regression - AUC

- auc(x, y)
  - **X**: False positive rate under different thresholds
  - **Y**: True positive rate under different thresholds
- Notes:
  - Computes the Area Under the Curve




```
clf_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, color='b')
plt.plot(np.linspace(0,1,10), np.linspace(0,1,10), linestyle="--")
plt.title(r"ROC (AUC = {:.3f})".format(clf_auc))
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
```

Decision Trees

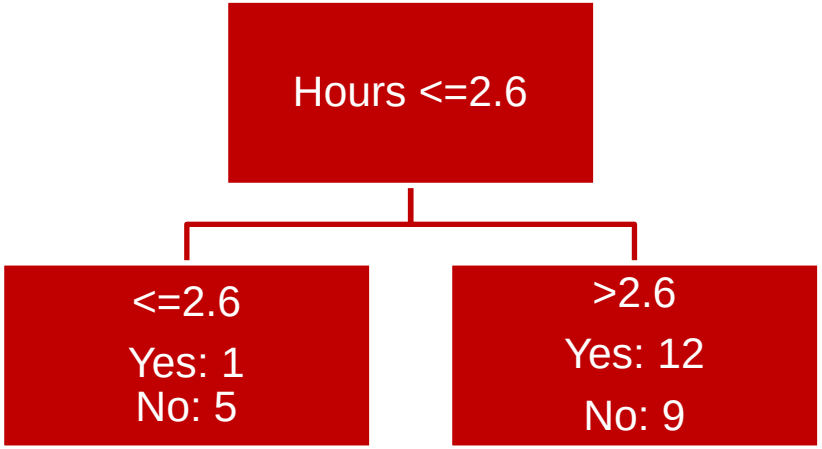
DecisionTreeClassifier

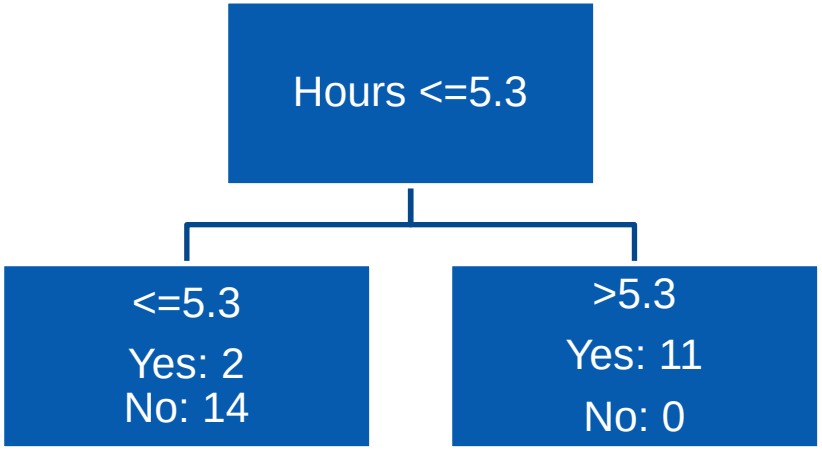
- tree.DecisionTreeClassifier(max_depth=None, random_state=None, ccp_alpha=0.0)
  - **Max_depth**: maximum depth of the tree
  - **Random_state**: Features are randomly permuted at each split, setting value creates deterministic behaviour
  - **Ccp_alpha**: Complixty parameter used for Minimal Cost-Complexity Pruning. Default no pruning is performed. The subtree with the largest cost complexity that is smaller than specified value will be chosen.


```
from sklearn import tree
clf = tree.DecisionTreeClassifier(random_state=0)  # clf for classifier
clf.fit(X, y)  # X=predictors, Y=target
```

Impurity – Example

Hours	Scores	Outcome
1.1	17	0
1.5	20	0
1.9	24	0
2.5	21	0
2.5	30	0
2.5	75	1
2.7	25	0
2.7	30	0
3.2	27	0
3.3	42	0
3.5	30	0
3.8	35	0
4.5	41	0
4.5	49	0
4.8	54	1
5.1	47	0
5.5	60	1
5.9	62	1
6.1	67	1
6.9	76	1
7.4	69	1
7.7	85	1
7.8	86	1
8.3	81	1
8.5	75	1
8.9	95	1
9.2	88	1





Impurity – Calculation

- Impurity represents how many misclassified observations inside the leaf
- Gini Impurity of leaf

$$1 - (\text{probability of } 1)^2 - (\text{the probability of } 0)^2$$

- For Hours <=2.6 leaf

$$1 - \left(\frac{1}{1+5}\right)^2 - \left(\frac{5}{1+5}\right)^2 = 0.2778$$

- For Hours >2.6 leaf

$$1 - \left(\frac{12}{12+9}\right)^2 - \left(\frac{9}{12+9}\right)^2 = 0.4898$$

- Total Gini Impurity

$$\frac{6}{27}(0.2778) + \frac{21}{27}(0.4898) = 0.44269$$

- For Hours <=5.3 leaf

$$1 - \left(\frac{2}{2+14}\right)^2 - \left(\frac{14}{2+14}\right)^2 = 0.2188$$

- For Hours >5.3 leaf

$$1 - \left(\frac{11}{11+0}\right)^2 - \left(\frac{0}{11+0}\right)^2 = 0.0000$$

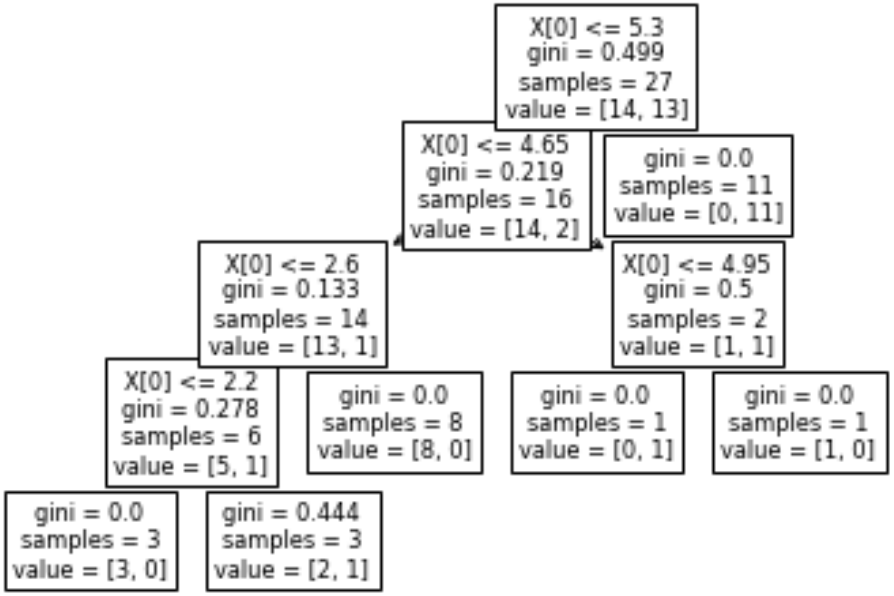
- Total Gini Impurity

$$\frac{16}{27}(0.2198) + \frac{11}{27}(0.000) = 0.1296$$

- Need to take weighted average since leaves are not same size
- This process is repeated for each pair average until the lowest Gini Impurity is found.
- Once the lower Impurity is found, the next feature is selected to split along.
- Each layer is optimized in isolation
  - Not peeking ahead in a “what if” scenario
- For terminal leaf, the output is the highest category

Overfitting

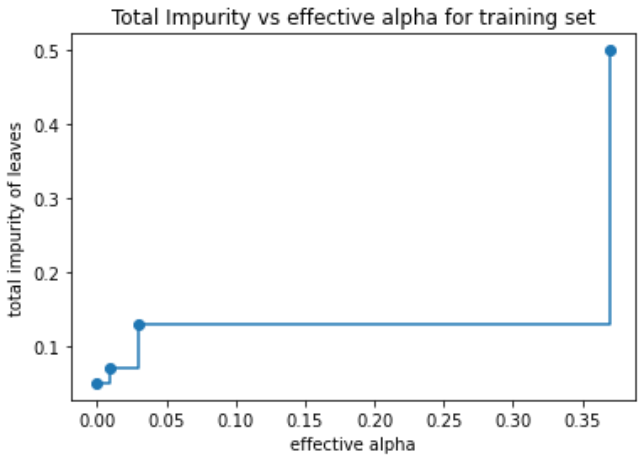
- Overfitting can occur when you have leaves with zero Impurity
- These do not perform well on unseen data
  - From the graph, you can see that everything is fully classified except for one terminal leaf




```
# after the classifier has been trained
tree.plot_tree(clf)
```

Overfitting – Pruning Trees

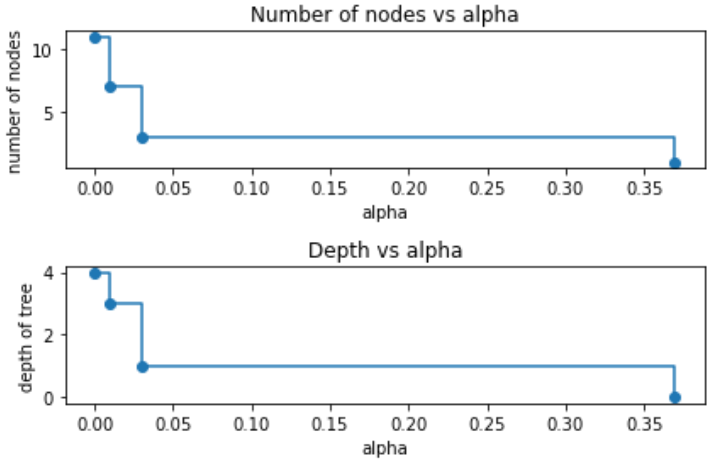
- To prevent against overfitting, you can limit the number of layers or features
  - This can be difficult on complex datasets understanding feature importance
  - You are imposing an arbitrary decision
- Cost_complexity_pruning_path(X, y)
  - **X**: predictors
  - **Y**: targets




```
path = clf.cost_complexity_pruning_path(X,y)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

#code to visualize
fig, ax = plt.subplots()
ax.plot(ccp_alphas, impurities, marker="o", drawstyle="steps-post")
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
```

Overfitting – CCP Alpha Values

- We can train several classifiers on different ccp_alpha values to see how the model performs as it is incrementally pruned




```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = tree.DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X, y)
    clfs.append(clf)
clfs = clfs
ccp_alphas = ccp_alphas
node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
```

Overfitting – Cost Complexity Alpha

- Cost Complexity Alpha (ccp_alpha) hyperparameter

$$R_\alpha(T) = R(T) + \alpha\left|\tilde{T}\right|$$

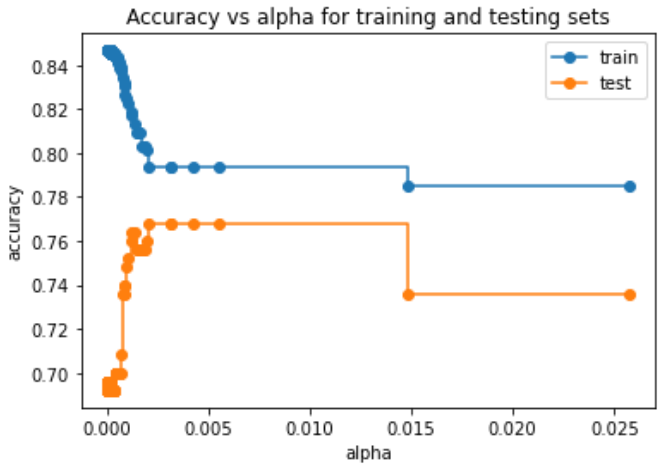
  - R(T) is the misclassification rate (impurity) of the tree, alpha is the cost complexity hyperparameter, and T is the number of leaves for the given tree
  - The effective alpha for pruning a terminal node is given by

$$\alpha_{eff}(t) = \frac{R(t) - R(T_t)}{|T| - 1}$$

  - $R(T_t)$ is the impurity for the root where $R(t)$ is the impurity of the terminal leaf to be pruned.
  - Pruning stops when the pruned trees minimal alpha is greater than the ccp_alpha hyperparameter. How to find the optimal value?

Overfitting – Cost Complexity Pruning

- The Cost Complexity Pruning Path function returns the effective alpha and total leaf impurity through a recursive algorithm
  - Basically works its way through pruning a terminal leaf one at a time to find the lowest effective alpha
- Want to compare the effective alphas from the training data to the accuracy of the test data
- This alpha is then used as the ccp_alpha parameter.




```
train_scores = [clf.score(X_train.values.reshape(-1,1), y_train) for clf in clfs]
test_scores = [clf.score(X_test.values.reshape(-1,1), y_test) for clf in clfs]
fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker="o", label="train", drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker="o", label="test", drawstyle="steps-post")
ax.legend()
plt.show()
```

Model Selection

Cross-Validation Function

- cross_val_score(estimator, X, y=None, cv)
  - **Estimator**: the model that you would use .fit() with to train.
  - **X**: predictors
  - **Y**: target
  - **Cv**: cross-validation generator
    - None: default 5-fold cross validation
    - Int, to specify the number of folds
    - An iterable that generates (train, test) splits as arrays of indices


```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import (KFold, StratifiedKFold, GroupKFold)
```

Cross-Validation – Overview

All Data					
Training Data					Test Data
Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	
Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	
Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	
Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	
Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	
					Final Eval
	Test	Train			

Stratified KFold

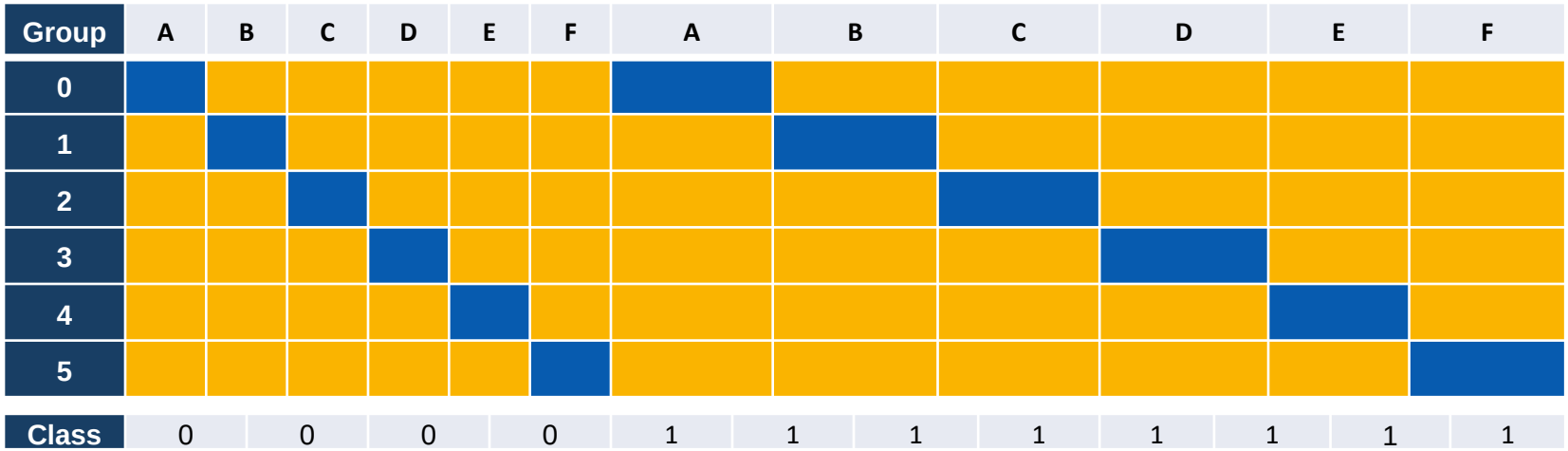
- Ensures proper representation of classes between folds (i.e. the target categories)
- StratifiedKFold(n_splits=5, shuffle=False, random_state=None)
  - **N_splits**: number of folds, must be at least 2
  - **Shuffle**: to mix up data before splitting
  - **Random_state**: controls the randomness of the shuffle, setting value provides deterministic result

Class	0	0	0	0	1	1	1	1	1	1	1	1
0	Test 1				Test 1	Test 1						
1		Test 2					Test 2	Test 2				
2			Test 3						Test 3	Test 3		
3				Test 4							Test 4	Test 4
Group	A	B	C	D	E	F	A	B	C	D	E	F

```
clf = tree.DecisionTreeClassifier(random_state=42)
cv = StratifiedKFold(n_splits=5)
scores = cross_val_score(clf, X_train, y_train, cv=cv)
scores
```

Group KFold

- Ensures groups are not split between folds (i.e. an additional category in data, other than the target classes)
- GroupKFold(n_splits=5)
  - **N_splits**: number of folds, must be at least 2.




```
cv = GroupKFold(n_splits=5)  # Must me less than or equal to number of groups
clf = LogisticRegression(random_state=42)
scores = cross_val_score(clf, X, y, groups=G, cv=cv)
scores
```

For more information on:

The Marquee Group

TheMarqueeGroup.com

info@TheMarqueeGroup.com

+1 416 583 1802

Training The Street

TrainingTheStreet.com

info@TrainingTheStreet.com

+1 866 931 5403 (Toll-free, US)

THE MARQUEE GROUP

A TRAINING THE STREET COMPANY