COMP.4220 - Machine Learning

Tutorial: Python Basics

1 Python modules

Import the required modules for this tutorial

```python
import numpy as np
import scipy as sp
from scipy.stats import norm
```

2 Distributions

Drawing samples from a binomial distribution

```python
# result of flipping a coin 10 times, tested 100 times.
n = 10      # number of trials,
p = 0.5     # probability of each trial
s = np.random.binomial(n, p, 100)
```

Using binomial PMF and CDF

```python
p = sp.stats.binom.pmf(5.0, n=10, p=0.5)
c = sp.stats.binom.cdf(0.5, n=10, p=0.5)
```

Drawing samples from a multinomial distribution The multinomial distribution is a multivariate generalization of the binomial distribution.

```python
# Sample throwing a dice 20 times
n = 20  # number of experiments
pval = [1/6.]*6 # probabilities of each of p outcomes
s = np.random.multinomial(n, pval, size=1) # array([[4, 1, 7, 5, 2,
    1]]) It landed 4 times on 1, once on 2, etc
```

Drawing samples from a 1-D Gaussian distribution This function accepts the mean and the standard deviation (not the variance) of the Gaussian distribution and returns a vector of samples.

```python
mean = 0.0
std = 1.0
s = np.random.normal(mean, std, size=10)
```

Using Gaussian PDF and CDF Note that by default all the functions use a standard Gaussian, $\mathcal{N}(x|0,1)$.

```
x = norm.ppf(0.5) # find ppf in the middle of the range
p = norm.pdf(x)
c = norm.cdf(0.5)
```

Drawing samples from a Multivariate Gaussian distribution

```
mean = [0, 0]
cov = [[1, 0], [0, 100]]  # diagonal covariance
S = random.multivariate_normal(mean, cov, size=20)
```

Drawing samples from a Gamma distribution

```
shape = 2.  # mean=4
scale = 2.  # std=2*sqrt(2)
s = np.random.gamma(shape, scale, 100)
```

Note: The `numpy` and `scipy` modules include required functions for all distributions we might need in this course. For more details, refer to the respective documentations.

3 Matrix Inversion

Inverse of a square matrix To compute the inverse of a square matrix, you can use the `inv` function. The solution satisfies $AA^{-1} = A^{-1}A = I$.

```
A = np.array([[1., 2.], [3., 4.]])
Ainv = np.linalg.inv(A)
np.allclose(A @ Ainv, np.eye(2)) # verification
```

Pseudo-inverse of a matrix To compute the (Moore-Penrose) pseudo-inverse of a matrix, you can use the `pinv` function which calculates the generalized inverse of a matrix using its singular-value decomposition (SVD). Note that the pseudo-inverse of a matrix A, denoted $A^\dagger$, is defined as: "the matrix that 'solves' [the least-squares problem] $Ax = b$," i.e., if $\bar{x}$ is said solution, then $A^\dagger$ is that matrix such that $\bar{x} = A^\dagger b$.

```
rng = np.random.default_rng()
A = rng.normal(size=(9, 6))
Ainv = np.linalg.pinv(A)
```

4 $Ax = b$

To compute the "exact" solution, x, of the well-determined, i.e., full rank, linear matrix equation $Ax = b$, you can use the `solve` function.

```
A = np.array([[1, 2], [3, 5]])
b = np.array([1, 2])
x = np.linalg.solve(a, b)
```

Note: When implementing various algorithms, in many cases it would be easier to solve a system of linear matrix equations (i.e., $Ax = b$) instead of calculating the inverse of a matrix, A and then multiplying it with the vector b to find x.