# AIMS BALITU - Week 1 Assignment

This assignment covers the fundamental concepts of active learning and information theory introduced in the first week's lectures. It is designed to take approximately 6-10 hours to complete.

**Points:** Only 75/100 points will be needed to obtain 100% on this assignment. (You can choose to do more if you want extra practice.)

**Due Date:** 2024-11-30

**Submission Instructions:** Please use the normal process at AIMS to submit your assignment. For non-code questions, submit handwritten answers to the assignment questions as they are quite math heavy, and there are some diagrams to be drawn.

## Part 1: Information Theory Foundations

**1. (12p) Entropy and Information Content:**

**(a) (2p)** Calculate the entropy (in bits) of a four-sided die $X$ with $\mathrm{p}(X = i) = (0.4, 0.3, 0.2, 0.1)$.

**(b) (6p)** Huffman Codes:

1. A random variable $Y$ takes on three values $\{A, B, C\}$ with probabilities $(0.25, 0.5, 0.25)$. Design a Huffman code for this random variable and draw the corresponding tree. What is its expected code length? Compare this to the theoretical lower bound.
2. A random variable $Z$ takes on eight values $\{A, B, C, D, E, F, G, H\}$ with probabilities $(0.25, 0.25, 0.125, 0.125, 0.0625, 0.0625, 0.0625, 0.0625)$. Design a Huffman code for this random variable and draw the corresponding tree. What is the expected length of the code?
3. A random variable $W$ takes on five values $\{A, B, C, D, E\}$ with probabilities $(0.35, 0.17, 0.17, 0.16, 0.15)$. Design a Huffman code for this distribution and draw the corresponding tree. How does the expected code length compare to the entropy? What makes this case different from the previous examples?

**(c) (2p)** Explain in your own words the relationship between entropy, information content, and code length. Why is entropy a lower bound on expected code length? What are some reasons why, in practice, we might not be able to achieve this lower bound?

**(d) (1p)** When is entropy maximized? When is it minimized? Give examples for both cases. Consider only discrete random variables for now.

**(e) (1p)** A source emits symbols from an alphabet with probability distribution $(0.1, 0.2, 0.3, 0.4)$. What codeword lengths might this code have? Calculate the average number of bits needed using a near-optimal code and a Huffman code. Give some possible prefix-free codewords.

**2. (16p) KL Divergence and Cross-Entropy:**

**(a) (8p)** Compute the KL divergence between the following pairs of distributions:

1. $\mathrm{p}(x) \sim \mathrm{Bernoulli}(0.8), \mathrm{q}(x) \sim \mathrm{Bernoulli}(0.2)$
2. $\mathrm{p}(x) \sim \mathrm{Uniform}(0.5, 1), \mathrm{q}(x) \sim \mathrm{Uniform}(0, 1)$
3. $\mathrm{p}(x) \sim \mathrm{Uniform}(0.5, 1.5), \mathrm{q}(x) \sim \mathrm{Uniform}(0, 1)$

(Draw diagrams of both distributions and the overlapping region.)

**(b) (4p)** Explain intuitively what the KL divergence measures. Why does the KL divergence between a distribution and itself equal zero? Provide both an intuitive and a formal proof.

**(c) (2p)** What is the relationship between cross-entropy and KL divergence?

**(d) (2p)** Give two (real-world) examples where minimizing cross-entropy is a practical objective. One should be from machine learning.

**3. (16p) Mutual Information and Interaction Information:**

**(a) (6p)** Draw I-diagrams and discuss what each region represents for the following scenarios:

1. Two independent variables $X$ and $Y$.
2. Two perfectly correlated variables $X$ and $Y$.
3. Three variables $X$, $Y$, and $Z$, where $X$ and $Y$ are independent, and $Z = X \oplus Y$ (where $\oplus$ denotes XOR).
4. A Markov chain $X \to Y \to Z$.

> **Note**
>
> **Markov Chain** A sequence of random variables $X_1 \to X_2 \to X_3 \to \cdots$ where each variable depends only on the previous one. Formally, $\mathrm{p}(X_n \mid X_1, \ldots, X_{n-1}) = \mathrm{p}(X_n \mid X_{n-1})$. The "future" is conditionally independent ($\perp$) of the "past" given the "present".

**(b) (6p)** Calculate $\mathrm{I}[X;Y]$ for the XOR example above, assuming $X$ and $Y$ are independent Bernoulli(0.8) random variables. What is $\mathrm{I}[X;Z]$? What is $\mathrm{I}[Y;Z]$? What is $\mathrm{I}[X;Y;Z]$? Relate these calculated quantities to the I-diagram. Explain what each value means.

**(c) (2p)** Give two real-world examples of three variables where you would expect negative interaction information. One example should be from a scientific domain of your choice. Explain why negative interaction information exists in these cases.

> Note
>
> **Interaction Information** A measure that quantifies the synergistic or redundant information shared among three or more variables. For three variables $X$, $Y$, and $Z$, it is defined as $\mathrm{I}[X;Y;Z] = \mathrm{I}[X;Y] - \mathrm{I}[X;Y \mid Z]$. (This is the mutual information between three or more variables.)

**(d) (4p)** Consider a sequence of $n$ fair coin flips $X_1, \dots, X_n$. Let $Y = \sum_i X_i$, where $n$ is sufficiently large. What is $\mathrm{I}[X_i;X_j]$ for $i \neq j$? What is $\mathrm{I}[Y;X_i]$? Draw an I-diagram.

**4. (25p + 16p) Coding Exercise:**

> Note
>
> A Colab notebook is available here.

Consider a system of binary random variables that models basic logic operations. This problem explores the information-theoretic properties of XOR and AND gates, providing insight into how information flows through simple computational units.

> Note
>
> **XOR Gate** The exclusive OR operation outputs 1 only when inputs differ. This creates a form of "perfect synergy" where neither input alone provides information about the output.
>
> **Synergistic Information** Information about a target variable that can only be obtained by jointly observing multiple source variables. XOR is a canonical example.

**(a) (3p)** Let $A$ and $B$ be independent, uniformly distributed binary random variables. Define two derived variables:

- $C = A \oplus B$ (XOR gate)
- $D = A \cdot B$ (AND gate)

(1) **(2p)** Implement Python functions that compute this distribution:

```python
def compute_joint_distribution() -> dict[tuple, float]:
    """
    Returns:
        dict mapping (a,b,c,d) outcome tuples to probabilities
    """
```

**Note**

For a single Bernoulli random variable $X$ with parameter $p$, the probability distribution is $\mathrm{p}(X = 0) = 1 - p$ and $\mathrm{p}(X = 1) = p$, and the distribution `dict` above would be:

```python
{(0,): 1-p, (1,): p}
```

**Note**

Note the code file contains several helper functions to compute this joint distribution. We describe them in the appendix. You can use them or roll your own.

(2) **(1p)** Output the joint probability distribution $\mathrm{p}(A, B, C, D)$ in a table (using Pandas or similar—whatever works) and using `repr()`.

**(b)** **(8p)** Information Flow Analysis:

(3) **(2p)** Calculate the pairwise mutual information quantities:

- $\mathrm{I}[A; B]$
- $\mathrm{I}[A; C]$ and $\mathrm{I}[B; C]$
- $\mathrm{I}[A; D]$ and $\mathrm{I}[B; D]$
- $\mathrm{I}[C; D]$

**Note**

You can do this by hand or do 2. first and then use that. However, it can be useful to compute some of these quantities to check your work.

(4) **(6p)** Implement these calculations numerically:

```python
def compute_mutual_information(joint_dist: dict[tuple, float]) -> dict[tuple, float]:
    """
    Args:
        joint_dist: Output from compute_joint_distribution()
    Returns:
        dict mapping variable pairs, e.g. ('A', 'B'), to MI values
    """
```

**(c) (8p)** Higher-Order Dependencies:

(5) **(6p)** Compute the other possible mutual information terms:

- $\mathrm{I}[A; B; C]$, $\mathrm{I}[A; B; D]$, etc.
- $\mathrm{I}[A; B; C; D]$

(6) **(2p)** Draw an I-diagram that contains all the variables $A, B, C, D$.

**(d) (6p)** Analysis:

(7) Why does the XOR operation lead to different mutual information patterns compared to AND?
(8) Explain the presence of negative interaction information (if any) in terms of synergistic information.
(9) How do these results relate to the concept of redundancy in information theory?

**(e) (16p)** Atomic Quantities (Optional):

Implement a Python class that computes all atomic information quantities for N variables given their joint distribution. The class should provide methods to compute arbitrary information measures from these atomic quantities.

Note

**Atomic Quantity** Using the signed measure: any intersection of the variable or a variable's complement that references all variables. E.g. $\mu^*[A \cap B \cap \bar{C} \cap \bar{D}] \triangleq \mathrm{I}[A; B \mid C, D]$.

1. **(8p)** Implement the following interface:

```python
import numpy as np

class AtomicInformation:
    def __init__(self, n_vars: int, joint_dist: dict[tuple, float]):
        """Initialize with joint probability distribution.

        Args:
            n_vars: Number of variables
            joint_dist: Dictionary mapping tuples of outcomes to probabilities.
            E.g., {(0,0,1): 0.25, (0,1,1): 0.25, ...} for 3 binary variables
        """
        self.joint_dist = joint_dist
        self.n_vars = n_vars
        self._compute_atomic_quantities()

    def _compute_atomic_quantities(self):
        """Compute and store all atomic information quantities."""
        # Your implementation here
        pass

    def entropy(self, variables: list[int]) -> float:
        """Compute joint entropy H(X_i, X_j, ...) for given variable indices."""
        pass

    def conditional_entropy(self, target: list[int], given: list[int]) -> float:
        """Compute conditional entropy H(X_i, X_j | X_k, X_l, ...)."""
        pass

    def mutual_information(self, vars_a: list[int], vars_b: list[int],
                           given: list[int] = None) -> float:
        """Compute (conditional) mutual information I(X_i, X_j ; X_k | X_l, ...)."""
        pass
```

2. **(4p)** Test your implementation on the XOR example from Part 1, Question 4. Verify that:
   - All conditional entropies are non-negative
   - The chain rule of mutual information holds
   - The data processing inequality holds for any Markov chain
3. **(4p)** Use your implementation to:
   1. Find all atomic quantities for the XOR distribution
   2. Verify that $\mathrm{I}[X;Y] = 0$ but $\mathrm{I}[X;Y \mid Z] > 0$
   3. Compute the interaction information $\mu^*[X;Y;Z]$
4. **(Bonus)** Extend your implementation to:
   1. Generate I-diagrams using matplotlib or networkx
   2. Handle continuous variables through discretization (quantization)

Implementation Tips

- Leverage symmetries to reduce computation (e.g. $\mathrm{I}[A; B] = \mathrm{I}[B; A]$)
- Use logarithms with appropriate base (e.g., base 2 for bits)
- Handle numerical stability issues with small probabilities (e.g. add a very small epsilon to probabilities or mask NaNs)
- When testing consider using small values of $N$. (What is the problem with using larger values of $N$?)

## Part 2: Active Learning Core Concepts

### (15p) Active Learning vs. Passive Learning:

Note

A Colab notebook is available here.

**(a) (0p)** Execute the notebook cells:

- look at the 2D PCA visualization of the Forest Covertype dataset (`sklearn.datasets.fetch_covtype`),
- read the description of the dataset (https://scikit-learn.org/stable/datasets/real_world.html#covtype-dataset),
- consider what the performance gap tells us about the opportunity of active learning for this dataset.

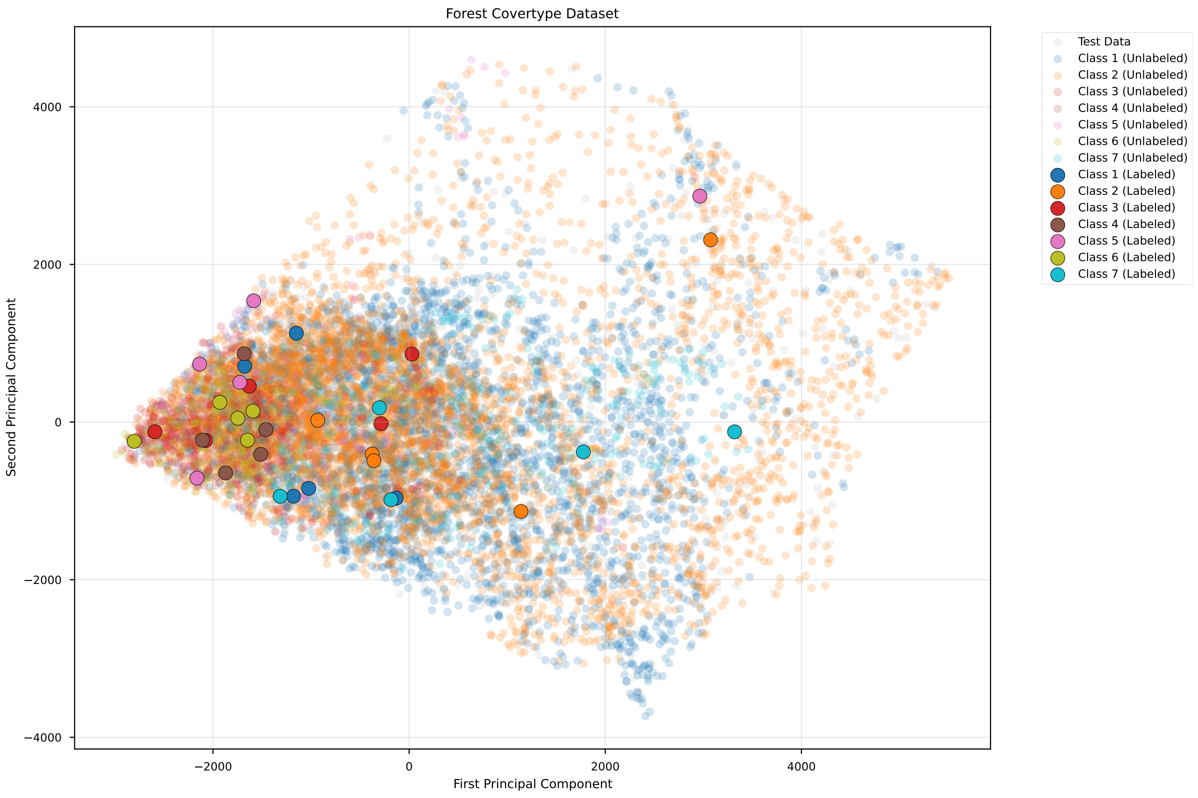

Figure 1: Dataset Visualization

**(b) (8p)** Implement a pool-based active learning loop for this AL dataset. Query the pool set one by one for up to 750 queries. Use the following query strategies, which we have discussed in the lecture:

1. **Uncertainty Sampling**
   - Least Confident: selects examples with lowest maximum predicted probability
   - Margin Sampling: uses difference between top two predicted probabilities
   - Entropy Sampling: uses information-theoretic entropy across all class probabilities
2. **Query-by-Committee (QBC)**
   - Vote Entropy: measures disagreement among ensemble members using each members class prediction.
3. **Random Sampling** (baseline)
   - Used as a comparison benchmark
   - Randomly selects examples from the unlabeled pool

4. **(Bonus) Expected Error Reduction**

   - Chooses examples that minimize expected future error
   - Formula: $x^* = \arg\min_x \mathbb{E}_{y \sim p(y|x)}[\sum_{x' \in \mathcal{X}_{\text{pool}}} \mathbb{E}_{y' \sim p(y'|x')} \mathcal{L}(\theta_{t|y,x}; y', x')]$

Note

For Query-by-Committee, train a bootstrapped ensemble of 3 random forest classifiers by creating bootstrap samples from the training set.

Note

For Expected Error Reduction (bonus only), use the `predict_proba` method of the classifier to compute the predicted class probabilities and the cross-entropy loss to compute the expected future error.
You will have to retrain the classifier for each possible outcome for each possible query sample.
If this is taking too long to score the whole pool set (it probably will), you can use a *different* subset of the pool set as a proxy for the full pool set.

**(c) (2p)** Collect and plot learning curves (accuracy on the test set vs. number of labeled examples). Discuss the observed behavior. When/where does active learning provide the largest gains? At what point do the gains diminish?

**(d) (4p)** Run each query strategy 5 times. Plot and report the confidence intervals (median, 68%, 95%). What can you say about the variance of the different query strategies?

**(e) (1p)** Plot the plots rotated by 90 degrees (so accuracy is on the y-axis *downwards* and number of labeled examples is on the x-axis). Plot and discuss which plotting strategy is better for highlighting the advantages of active learning.

**(f) (Bonus)** Visualize the data acquisition order at each step. Render an animation of the data acquisition process.

## Appendix: Code Helpers

Here's a clear documentation of the helper methods in a style suitable for a technical assignment sheet:

## Helper Functions for Probability Distributions

### Core Distribution Functions

**`bernoulli_variable(p: float) -> dict[tuple, float]`**

Creates a Bernoulli distribution with parameter `p`.

**Input**: Probability `p` of outcome 1

**Output**: Dictionary mapping singleton tuples to probabilities, i.e., `{(0,): 1-p, (1,): p}`

**Example**: `bernoulli_variable(0.5)` returns `{(0,): 0.5, (1,): 0.5}`

**`product_distribution(dists: tuple[dict[tuple, float]]) -> dict[tuple, float]`**

Computes the product distribution of multiple probability distributions.

**Input**: Tuple of probability distributions, each as a dict mapping outcome tuples to probabilities

**Output**: Joint distribution as dict mapping concatenated outcome tuples to joint probabilities

**Note**: Assumes independence between input distributions

**`repeated_distribution(dist: dict[tuple, float], n: int) -> dict[tuple, float]`**

Computes the n-fold product of a single distribution with itself.

**Input**: Base distribution and number of repetitions

**Output**: Joint distribution of n independent copies

**Example**: For n=2, creates distribution of (X ,X ) where X ,X  are independent and identically distributed

## Variable Computation

**`append_computed_variable(dist: dict[tuple, float], compute_fn: Callable[..., object]) -> dict[tuple, float]`**

Extends a distribution with a new variable computed from existing ones.

**Input**:

- Base distribution
- Function computing new variable from existing outcomes

**Output**: Extended distribution including computed variable

**Example**: Computing XOR of two variables: `append_computed_variable(dist, lambda x,y: x^y)`