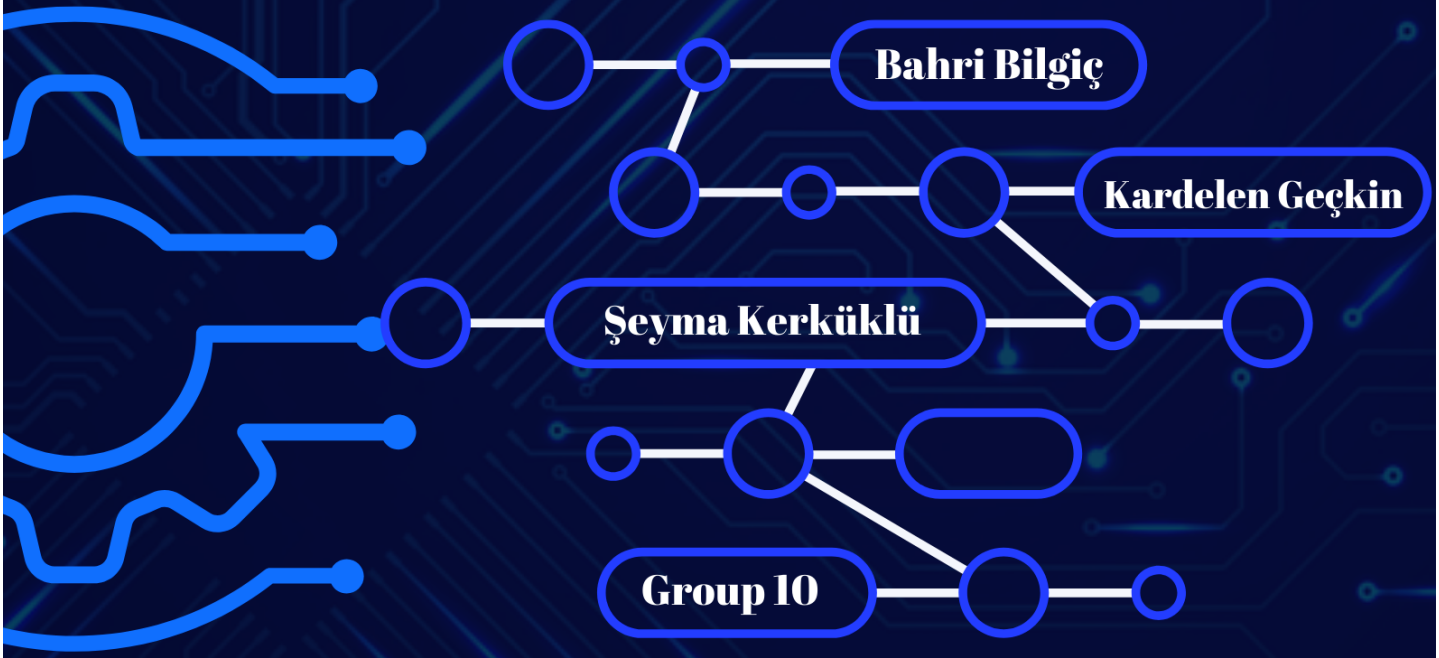
STATISTICAL LEARNING FROM DATA: APPLICATIONS IN PHYSICS

Project on 2D Ising Dataset



1.The 2D Ising Dataset

1.1 Explanation of Dataset

In this chapter we would like to illustrate a path from reading the provided data to giving a brief explanation of our dataset by developing further ideas about the Ising Model.

1.1.1 Reading Data

The website provided in the project leads us to Ising datasets from Prof. Pankaj Mehta's research. This dataset consists of samples generated from the two-dimensional nearest-neighbor coupled Ising model at a range of temperatures above and below the critical point.

In total, there are 21 files consisting of 16 different files of samples, temperatures differing from 0.25 to 4.0 K, and each specific temperature file consisting of 10,000x1600 spin samples, two other files containing all temperatures, 160,000x1600 spin samples, and their labels which is an array of 1600 elements. Lastly, a zip file for all temperatures given above and as these files are in .pkl format It is given two codes for reading the sample files.

1.1.2 Ising Model

The Ising model is a mathematical model in statistical mechanics that is used to study the behavior of spins in a system. The Ising model is commonly used to describe magnetic properties of materials, particularly ferromagnetic and antiferromagnetic materials. In the model, a system is represented by a lattice of discrete spins, which can take two possible values, typically +1 or -1, representing the orientation of magnetic moments.The basic form of the Ising model Hamiltonian (energy function) is given by:

$$H = -J \sum_{<i,j>} S_i S_j - h \sum_i S_i$$

- J is the coupling constant that represents the strength of interaction between neighboring spins.
- S_i is the spin at site i, which can take values of +1 or -1.

- $\langle i,j \rangle$ denotes that the summation is over neighboring pairs of spins.
- h represents an external magnetic field.

The first term in the Hamiltonian accounts for the interaction between neighboring spins. When J is positive, it favors aligned spins, leading to ferromagnetic behavior. When J is negative, it favors anti-aligned spins, resulting in antiferromagnetic behavior. The second term represents the interaction of spins with an external magnetic field.

The Ising model has been widely studied and has applications beyond magnetism, including in the study of phase transitions and critical phenomena. It serves as a fundamental model in statistical mechanics and has been used in various fields of physics, mathematics, and computer science.

1.1.3 The 2D Ising Dataset

The 2D Ising dataset for different temperatures ranging between 0.25 to 4.0 K can be illustrated as an array of ones and zeros which is depicting the magnetic momentums (spins)

```
data = pickle.load(urlopen(url_main + data_file_name)) # pickle reads the file and returns the Python object (1D array, compressed bits)
data = np.unpackbits(data).reshape(-1, 1600) # Decompress array and reshape for convenience
data
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1],
       ...,
       [1, 1, 1, ..., 1, 1, 0],
       [1, 1, 1, ..., 1, 0, 0],
       [1, 1, 1, ..., 1, 0, 0]], dtype=uint8)
```

We will change the zeros to -1 so we can represent the spins as +1 and -1 as counter spins.

```
data=data.astype('int')
data[np.where(data==0)]=-1
data
```

```
array([[-1, -1, -1, ..., -1, -1, -1],
       [ 1,  1,  1, ...,  1,  1,  1],
```

```
 [ 1,  1,  1, ...,  1,  1,  1],
 ...,
 [ 1,  1,  1, ...,  1,  1, -1],
 [ 1,  1,  1, ...,  1, -1, -1],
 [ 1,  1,  1, ...,  1, -1, -1]])
```

1.2 Deep Dive into Dataset

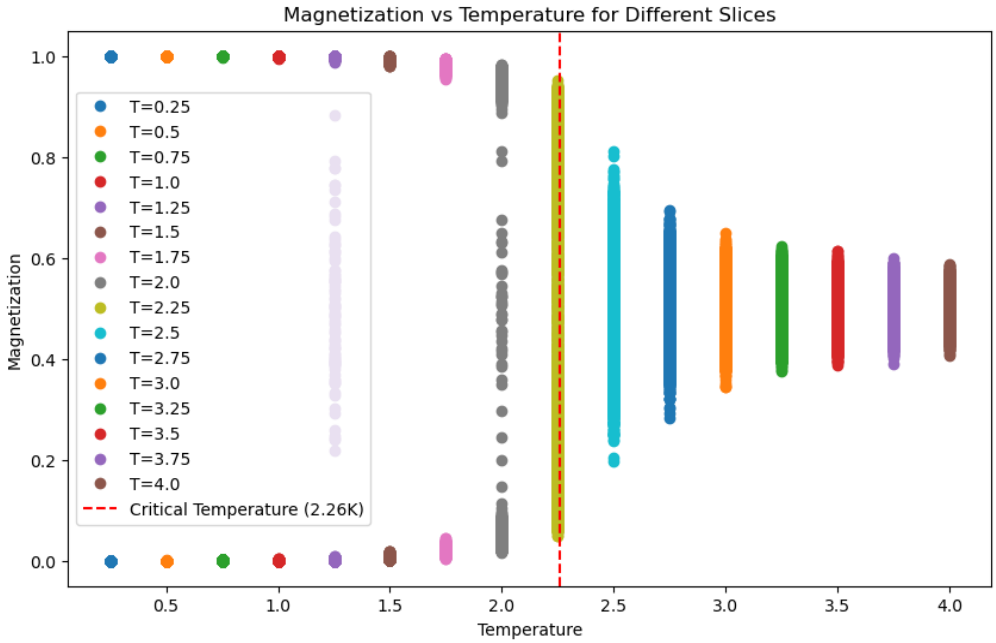
1.2.1 Depiction of Dataset

The Ising model describes a system of magnetic spins arranged on a lattice. These spins can be in an "up" or "down" state. At low temperatures, the system tends to align in a parallel manner, leading to a ferromagnetic phase where neighboring spins have the same orientation. At high temperatures, thermal fluctuations become significant, causing spins to be randomly oriented, leading to a disordered paramagnetic phase.

The critical temperature is the temperature at which a phase transition occurs in the Ising model. Below T_c, the system exhibits long-range order, and the material is in a ferromagnetic phase. Above T_c, thermal effects dominate, and the material transitions to a paramagnetic phase with disordered spins.

$$T_c = \frac{2}{ln(1+\sqrt{2}} \approx 2.269$$

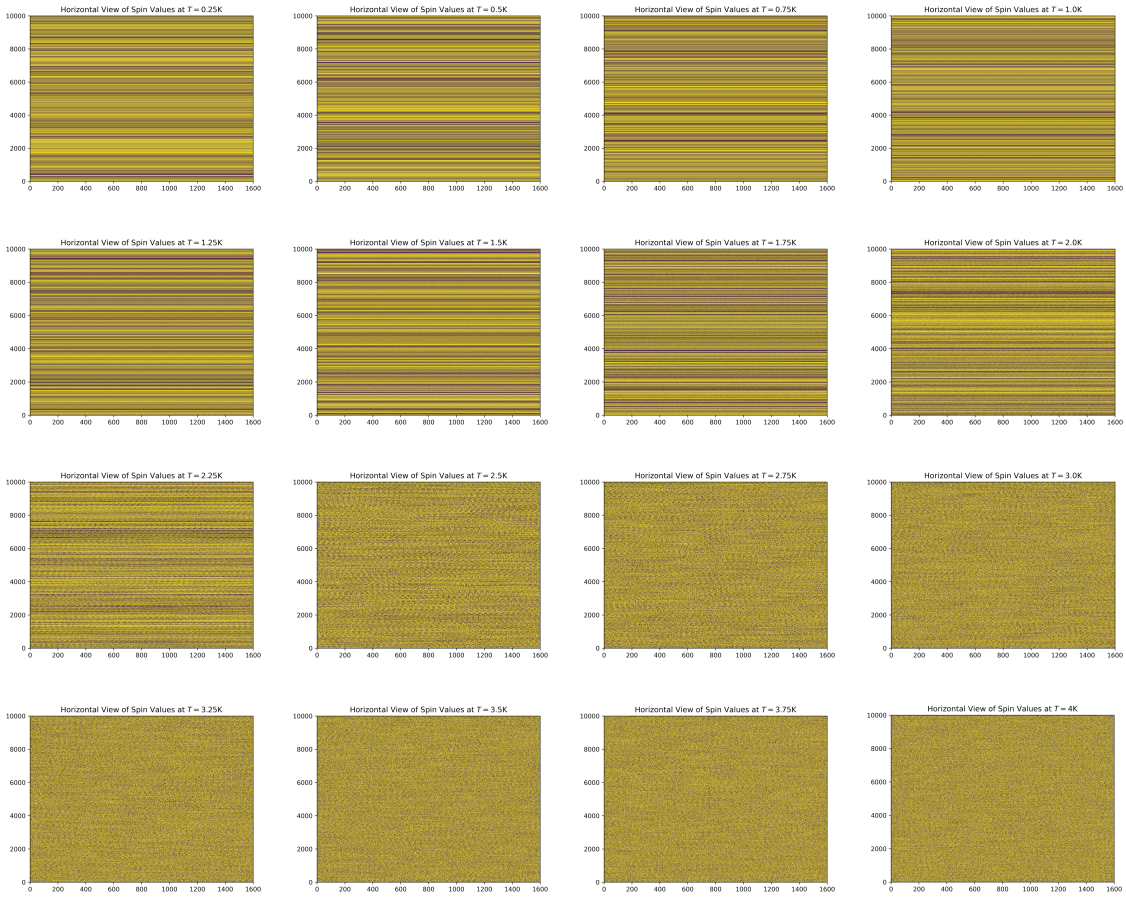
In our 2D Ising Dataset, the system of temperatures below T_c (0.25-2.25 K), can be considered to be in the ordered phase, exhibiting long-range order and possibly magnetization and the system of temperatures above T_c (2.5-4.0 K) where thermal effects become dominant, the system transitions to disordered phase; losing long-range order and magnetization.



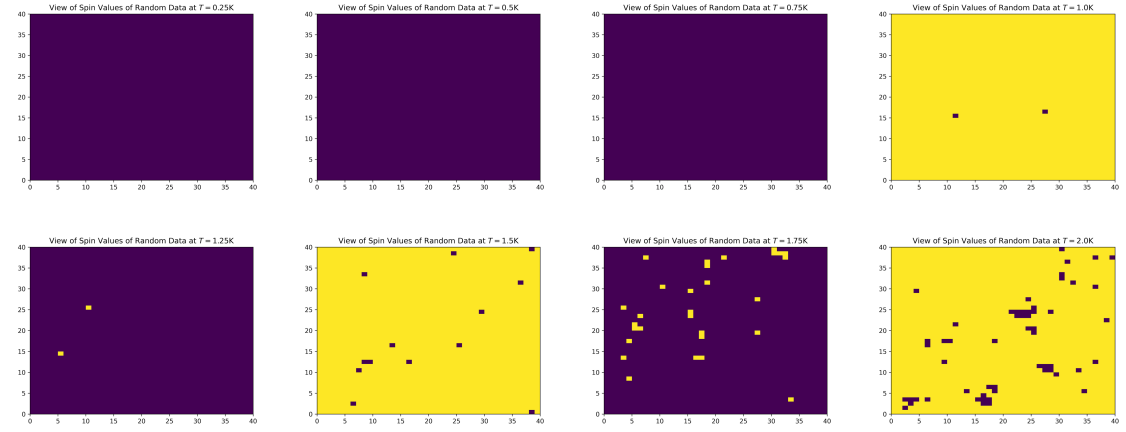
The scatter plot illustrates the magnetization behavior of a 2D Ising model as temperature varies. Different colors correspond to different temperatures, ranging from T=0.25 to T=4.0. The plot shows that at low temperatures (T=0.25 to T=2.25), the system is in the ordered phase, exhibiting high magnetization due to spins aligning parallel to each other - a characteristic of the ferromagnetic phase. As the temperature increases towards the critical temperature ($T_c \approx 2.26$, marked by the dashed red line), magnetization decreases, indicating the phase transition from ferromagnetic to paramagnetic. Beyond T_c, in the high-temperature range (T=2.5 to T=4.0), the plot depicts a significant drop in magnetization, aligning with the disordered paramagnetic phase where thermal fluctuations disrupt the spin alignment, leading to randomness and a lack of long-range order.

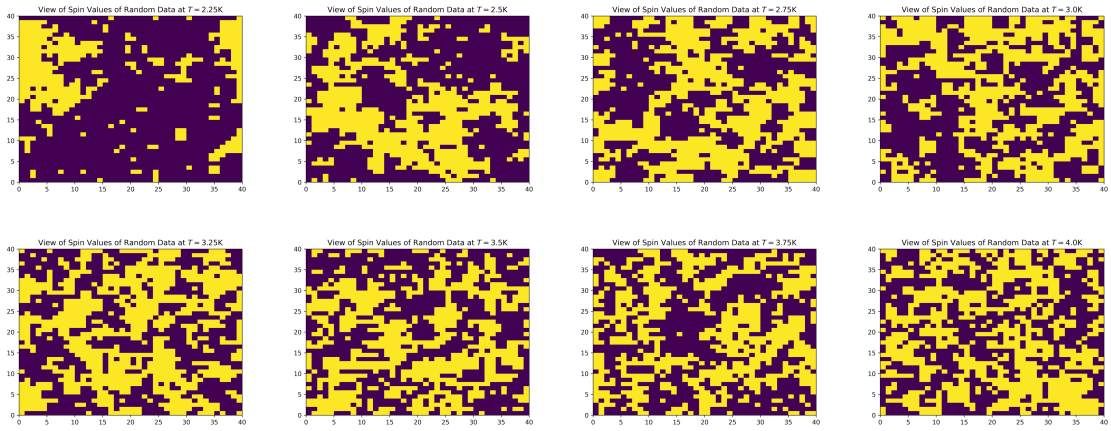
1.2.2 Visualization of Dataset

The 2D Ising Dataset consists of 16 different files of spin configurations depending on temperatures which are ranging 0.25 to 4.0 K by 0.25 increments. Each data file has a 10,000x1,600 size of spin configurations (-1,+1). To illustrate each data sample we can plot the horizontal view of spin values to see the pattern line by line and have a better understanding of the dataset.

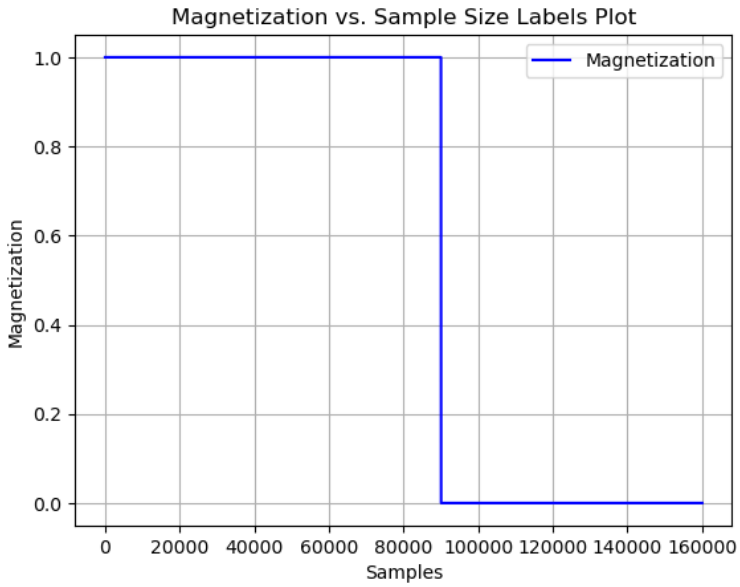


We can also show the spin configurations for each of the 2D Ising Dataset as we chose one random line from each temperature file size 1600 to create a 2 dimensional 40x40 grid to illustrate heatmap view.





If we consider all of the data, we can also separate it by its magnetization values (1,0). It is illustrated in the figure below:



2. Problem Selection

2.1 Problem Decision

Given the 2D Ising dataset, which is solely based on the spin configurations, common tasks might include predicting the phase of the material (ferromagnetic or paramagnetic) or estimating thermodynamic properties such as magnetization at different temperatures. In our machine learning models, the question we asked is:

“Can we train a machine learning model to classify different phases of the Ising model based on the spin configurations? ”

2.2 Problem Implementation Technique

To train a machine learning model to classify different phases of the Ising model based on spin configurations, we followed these general steps:

1. Data Preparation
2. Data Splitting
3. Model Selection
4. Feature Scaling
5. Model Training
6. Model Evaluation

3. Application

In this chapter, we select 4 different machine learning methods: logistic regression, random forest, MLP and CNN to train the dataset. Firstly, we give a brief insight about the methods. Secondly, implementation of our code. Finally, the prediction and evaluation of the results.

3.1 Algorithms

3.1.1 Logistic Regression

3.1.1.1 Logistic Regression Method

Logistic Regression is a machine learning algorithm used for binary classification. It models the probability of an instance belonging to a particular class using the logistic (sigmoid) function. The algorithm learns a decision boundary that separates the two classes in the feature space. During training, it adjusts its parameters to minimize the log loss, effectively optimizing the model for accurate classification. Logistic Regression is computationally efficient, interpretable, and serves as a foundational method in classification tasks.

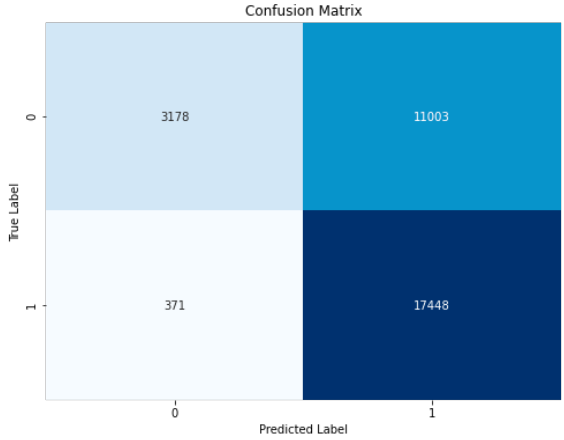
3.1.1.2 Implementation of Logistic Regression on the 2D Ising Dataset

```
import pickle
from urllib.request import urlopen
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score

def load_ising_data():
    url = 'https://physics.bu.edu/~pankajm/ML-Review-Datasets/isingMC/'
    file_name = "Ising2DFM_reSample_L40_T=All.pkl"
    label_file_name = "Ising2DFM_reSample_L40_T=All_labels.pkl"

    data = pickle.load(urlopen(url + file_name))
    data = np.unpackbits(data).reshape(-1, 1600).astype('int')
    data[data == 0] = -1
    labels = pickle.load(urlopen(url + label_file_name))
    return data, labels

def train_logistic_regression(X_train, y_train):
    model = LogisticRegression()
    model.fit(X_train, y_train)
    return model

def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    conf_matrix = confusion_matrix(y_test, y_pred)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    return conf_matrix, accuracy, precision, recall, f1

def plot_confusion_matrix(conf_matrix):
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()
```

```
def plot_evaluation_metrics(metrics_values, metrics_names):
    plt.figure(figsize=(10, 6))
    sns.barplot(x=metrics_values, y=metrics_names, palette='viridis')
    plt.title('Model Evaluation Metrics')
    plt.xlabel('Metric Value')
    plt.show()

data, labels = load_ising_data()

X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)

model = train_logistic_regression(X_train, y_train)

conf_matrix, accuracy, precision, recall, f1 = evaluate_model(model, X_test, y_test)

plot_confusion_matrix(conf_matrix)
```

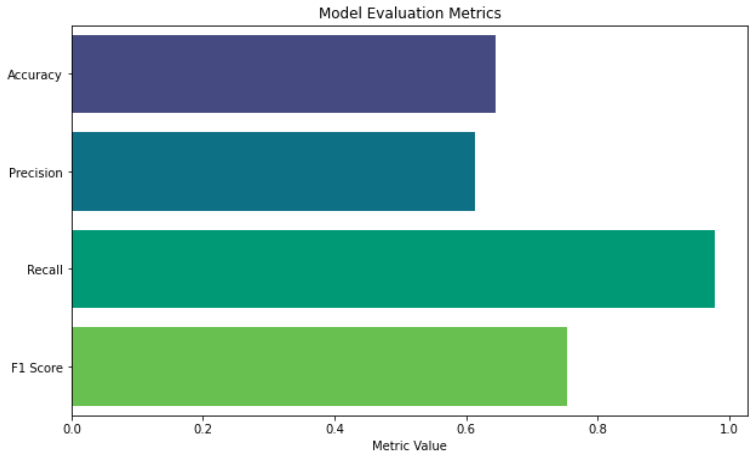
3.1.1.3 Prediction and Evaluation of Logistic Regression

We split the data as test and train data. The test data consists of 128,000 and the train data consists of 32,000 elements of spin configurations. After training and evaluating our data we can summarizes the performance of Logistic Regression Algorithm by confusion matrix depicted below:



To understand the performance of our training we used accuracy, precision, recall and f1-score values and illustrated them:

- Overall correctness (accuracy): 65 %
- The accuracy of positive predictions (precision): 61 %
- The ability of the model to capture all positive instances(recall): 98 %
- Harmonic mean of precision and recall (f1-score): 75 %



3.1.2 Random Forest

3.1.2.1 Random Forest Method

Random Forest is an ensemble learning method that constructs multiple decision trees during training. Each tree is trained on a random subset of the data, and their predictions are combined through voting or averaging. This approach reduces overfitting, improves generalization, and provides insights into feature importance. Random Forest is widely used for classification and regression tasks due to its robustness, efficiency with large datasets, and high predictive accuracy.

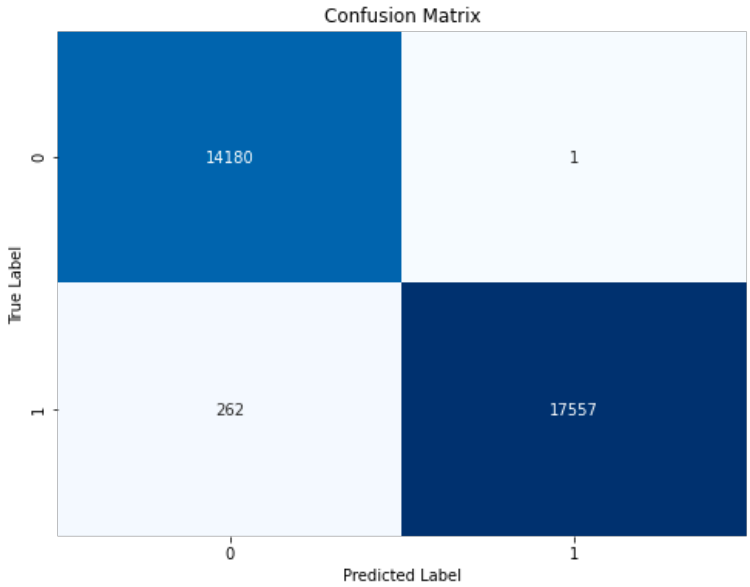
3.1.2.2 Implementation of Random Forest on the 2D Ising Dataset

```
import pickle, os
from urllib.request import urlopen
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler

def load_ising_data():
  url = 'https://physics.bu.edu/~pankajm/ML-Review-Datasets/isingMC/'
  file_name = "Ising2DFM_reSample_L40_T=All.pkl"
  label_file_name = "Ising2DFM_reSample_L40_T=All_labels.pkl"

  data = pickle.load(urlopen(url + file_name))
```

```
    data = np.unpackbits(data).reshape(-1, 1600).astype('int')
    data[data == 0] = -1
    labels = pickle.load(urlopen(url + label_file_name))

    return data, labels
def train_random_forest(X_train, y_train):
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    return model

def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    conf_matrix = confusion_matrix(y_test, y_pred)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    return conf_matrix, accuracy, precision, recall, f1

def plot_confusion_matrix(conf_matrix):
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

data, labels = load_ising_data()
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = train_random_forest(X_train, y_train)

conf_matrix, accuracy, precision, recall, f1 = evaluate_model(model, X_test, y_test)
plot_confusion_matrix(conf_matrix)

metrics_values = [accuracy, precision, recall, f1]
metrics_names = ['Accuracy', 'Precision', 'Recall', 'F1 Score']
plot_evaluation_metrics(metrics_values, metrics_names)
```

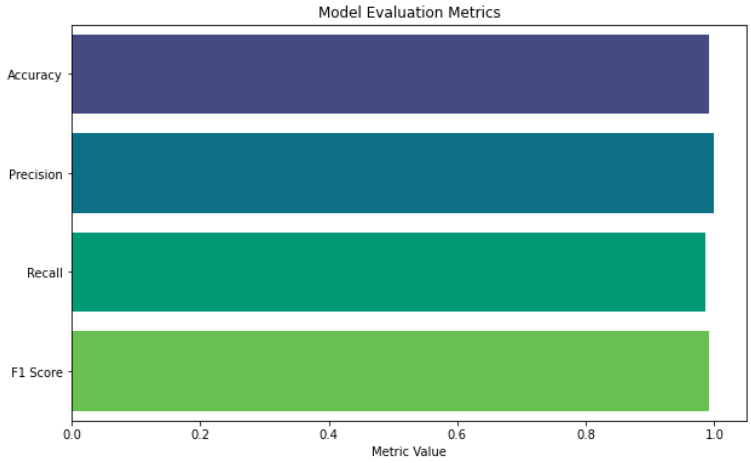
3.1.2.3 Prediction and Evaluation of Random Forest Model

We split the data as test and train data. The test data consists of 128,000 and the train data consists of 32,000 elements of spin configurations. After training and evaluating our data we can summarizes the performance of Random Forest Algorithm by confusion matrix depicted below:



To understand the performance of our training we used accuracy, precision, recall and f1-score values and illustrated them:

- Overall correctness (accuracy): 99.2 %
- The accuracy of positive predictions (precision): 99.9 %
- The ability of the model to capture all positive instances(recall): 98.5 %
- Harmonic mean of precision and recall (f1-score): 99.3 %



3.1.3 Multilayer Perceptron

3.1.3.1 Multilayer Perceptron (MLP) Method

The Multilayer Perceptron (MLP) is a commonly preferred classification algorithm within deep learning models. Belonging to the family of artificial neural networks, MLP generally consists of at least three layers: an input layer, one or more hidden layers, and an output layer. Nodes in each layer are connected to each other, and these connections are facilitated through weights. During the training process, the model is fed with examples from the dataset and strives to produce the desired output by adjusting the weights.

Hidden layers enable the learning ability to handle the complexity in the dataset and allow the network to learn non-linear relationships. The backpropagation algorithm is employed during the training process to enhance the performance of the network. MLP classifiers are often effective in multiclass problems, and they can achieve successful results, especially in large datasets. However, due to the network's tendency to overfit, appropriate regularization techniques need to be applied to improve the generalization ability of the network.

3.1.3.2 Implementation of MLP on the 2D Ising Dataset

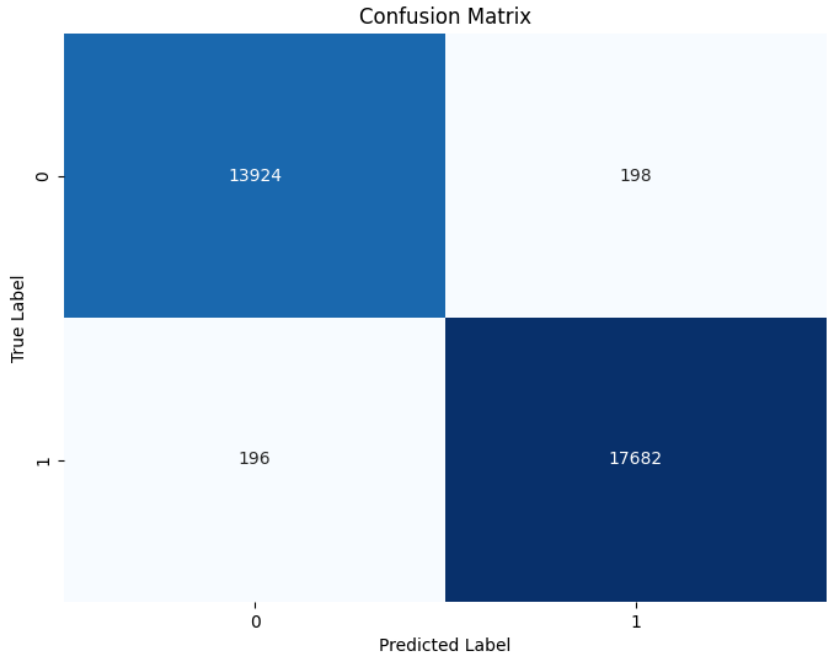
```
import numpy as np
import seaborn as sns
import pickle
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
from sklearn.neural_network import MLPClassifier


def read_t_data(t=0.25,root="../data/"):
  if t > 0.:
     data = pickle.load(open(root+'Ising2DFM_reSample_L40_T=%.2f.pkl'%t,'rb'))
  else:
     data = pickle.load(open(root+'Ising2DFM_reSample_L40_T=All.pkl','rb'))
  return np.unpackbits(data).astype(int).reshape(-1,1600)
def read_all_data(root="../data/"):
  data = pickle.load(open(root+'Ising2DFM_reSample_L40_T=All.pkl','rb'))
  label = pickle.load(open(root+'Ising2DFM_reSample_L40_T=All_labels.pkl','rb'))
  return np.unpackbits(data).astype(int).reshape(-1,1600), label

data, label = read_all_data()
idx = np.random.permutation(len(data))
data, label = data[idx], label[idx]
data[data == 0] = -1
X_train, X_test, y_train, y_test = train_test_split(data, label, test_size=0.2, random_state=0)

def train_MLPClassifier(X_train, y_train):
```

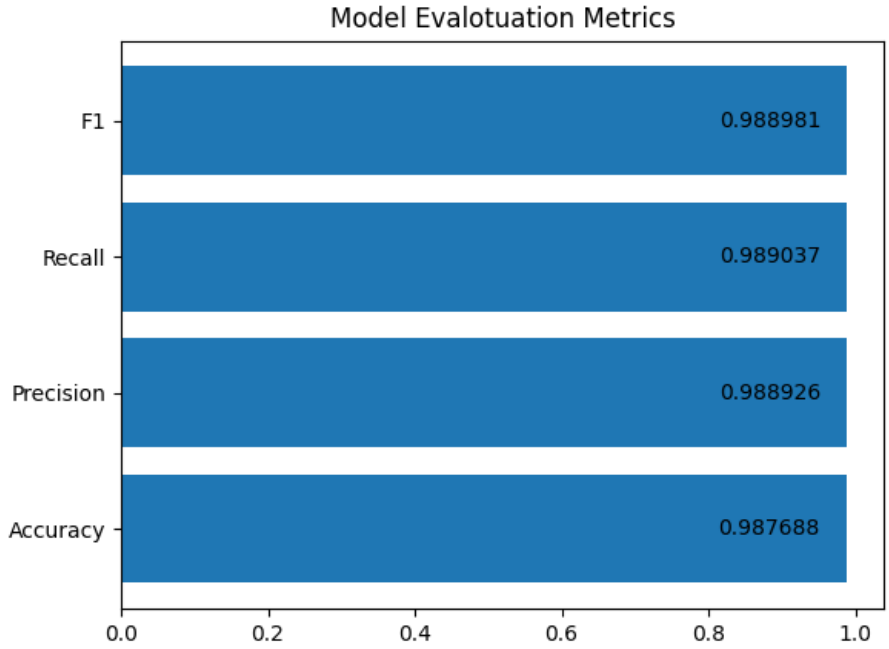
```
  model = MLPClassifier(solver='sgd', alpha=1e-3, hidden_layer_sizes=(5, 2), random_state=1773,
                      early_stopping=True)
  model.fit(X_train, y_train)
  return model


def evaluate_model(model, X_test, y_test):
  y_pred = model.predict(X_test)
  conf_matrix = confusion_matrix(y_test, y_pred)
  accuracy = accuracy_score(y_test, y_pred)
  precision = precision_score(y_test, y_pred)
  recall = recall_score(y_test, y_pred)
  f1 = f1_score(y_test, y_pred)
   return conf_matrix, accuracy, precision, recall, f1


def plot_confusion_matrix(conf_matrix):
  plt.figure(figsize=(8, 6))
  sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
  plt.title('Confusion Matrix')
  plt.xlabel('Predicted Label')
  plt.ylabel('True Label')
  plt.show()


def plot_evaluation_metrics(metrics_values, metrics_names):
  plt.figure(figsize=(10, 6))
  sns.barplot(x=metrics_values, y=metrics_names, palette='viridis')
  plt.title('Model Evaluation Metrics')
  plt.xlabel('Metric Value')
  plt.show()

model = train_MLPClassifier(X_train, y_train)
conf_matrix, accuracy, precision, recall, f1 = evaluate_model(model, X_test, y_test)
plot_confusion_matrix(conf_matrix)

plt.plot(model.validation_scores_)
plt.title("Accuracy Over Epochs")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.show()

fig, ax = plt.subplots()
ax.set_title("Model Evalotuation Metrics")
bars = ax.barh(("Accuracy", "Precision", "Recall", "F1"), (accuracy, precision, recall, f1))
ax.bar_label(bars,padding=-60);
plt.show()
```

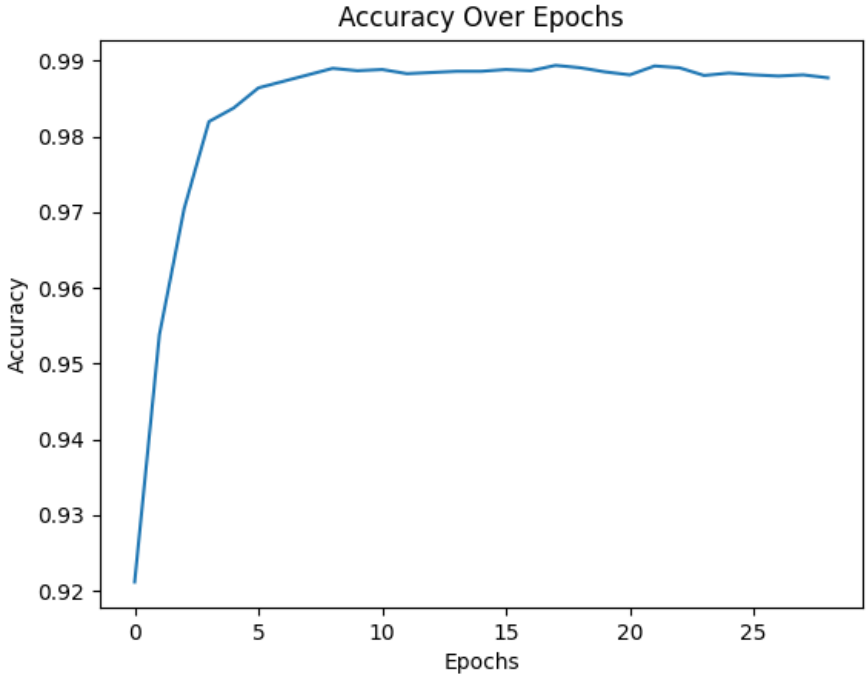
3.1.3.3 Prediction and Evaluation of MLP Model

We divided the data into training and test sets with a ratio of 2/10. As a result, we obtained a test dataset of 32,000 rows and a training dataset of 128,000 rows. We trained an MLP model from the scikit library using the training data. During this training, we specified various parameters for MLP. We used Stochastic Gradient Descent (SGD) as the solver.

While training the model with our data, we set the "early_stopping" parameter to True, which stops the training of the model when the accuracy remains constant for a certain period. This helps prevent unnecessary prolongation of the training and avoids the issue of the model memorizing instead of learning to make predictions. After training, we created the confusion matrix below to demonstrate the accuracy of the resulting model. Additionally, we plotted a graph showing the accuracy per epoch.







3.1.4 Convolutional Neural Network

3.1.4.1 Convolutional Neural Network Method (CNN)

Convolutional Neural Networks (CNNs) are a specialized class of deep learning models primarily employed in image processing tasks. The network comprises several sequential

layers, starting with the input layer where images are received, each consisting of pixels with specific values. Convolutional layers play a crucial role in feature detection, utilizing filters to identify patterns like edges or textures and generating feature maps. Activation functions, often ReLU, enhance the network's ability to interpret complex patterns. Pooling layers follow, reducing the spatial dimension of feature maps for increased processing efficiency. Fully connected layers integrate information for decision-making, particularly in image classification tasks, and the output layer presents the network's findings, often using a softmax function for classification probabilities. During training, CNNs optimize their filters and neurons through backpropagation and gradient descent, minimizing discrepancies between predictions and actual labels to improve predictive accuracy. Known for their effectiveness in image-related applications, CNNs excel at discerning and interpreting diverse features within images, making them powerful tools for tasks like object recognition.

3.1.4.2 Implementation of CNN on the 2D Ising Dataset

```
# %%
import numpy as np
import pickle
import pandas as pd
from urllib.request import Request, urlopen
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D
from keras import regularizers, optimizers
from keras.callbacks import ModelCheckpoint
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical



# %%
url_main = 'https://physics.bu.edu/~pankajm/ML-Review-Datasets/isingMC/'
# The data consists of 16*10000 samples taken in T=np.arange(0.25,4.0001,0.25):
data_file_name = "Ising2DFM_reSample_L40_T=All.pkl"
# The labels are obtained from the following file:
label_file_name = "Ising2DFM_reSample_L40_T=All_labels.pkl"

#DATA
# pickle reads the file and returns the Python object (1D array, compressed bits)
# Decompress array and reshape for convenience
# map 0 state to -1 (Ising variable can take values +/-1)
def load_data_from_url(url):
    request = Request(url, headers={'User-Agent': 'Mozilla/5.0'})
    with urlopen(request) as f:
        return pickle.load(f)
```

```
# Load data and labels
data = load_data_from_url(url_main + data_file_name)
data = np.unpackbits(data).reshape(-1, 1600)
data = data.astype('int')
#data[np.where(data == 0)] = -1 # map 0 state to -1

#LABELS (convention is 1 for ordered states and 0 for disordered states)
# pickle reads the file and returns the Python object (here just a 1D array with the binary labels)
labels = load_data_from_url(url_main + label_file_name)
#labels[labels == 0] = -1  # Map 0 state to -1

# %%
# Print the entire data array and its shape
print("Data Array:\n", data)
print("Shape of Data Array:", data.shape)

# Print the entire labels array and its shape
print("\nLabels Array:\n", labels)
print("Shape of Labels Array:", labels.shape)

# Print the first sample from the data array
print("\nFirst Sample in Data Array:\n", data[0])

# Print the last sample from the data array
print("\nLast Sample in Data Array:\n", data[-1])

# %%
# Calculate magnetization for each sample in the data by taking the mean along the rows
mag = np.mean(data, axis=1)
print(f"Shape of Magnetization Array: {mag.shape}")

# Create an array to represent sample sizes
sample_size = np.arange(len(data))
print(f"Shape of Sample Size Array: {sample_size.shape}")

# Create an array to represent temperature
temp_range = np.arange(0.25, 4.25, 0.25)
# Repeat the temperature values to match the length of the data (assuming each slice contains 10,000 samples)
tc = np.repeat(temp_range, 10000)
print(f"Temperature Array: {tc}")
print(f"Length of Temperature Array: {len(tc)}, Shape of Temperature Array: {tc.shape}")

# %%
# Plot the 'labels' data with a blue line
plt.plot(labels, c='blue', label='Magnetization')

# Set labels for the x and y axes
plt.xlabel('Samples')
plt.ylabel('Magnetization')
```

```
# Set the title of the plot
plt.title('Magnetization vs. Sample Size Labels Plot')

# Add a legend to the plot
plt.legend()

# Add a grid to the plot
plt.grid(True)

# Display the plot
plt.show()



# %%
# Create a DataFrame
df = pd.DataFrame({
    'Data': list(data),
    'Labels': labels,
    'Temperature': tc
})

# %%
# Define a function for slicing the data
def slice_data(df, num_slices):
    # Calculate the size of each slice
    slice_size = len(df) // num_slices
    sliced_data = []         # List to store sliced data
    sliced_labels = []       # List to store sliced labels
    sliced_temperatures = [] # List to store sliced temperatures

    # Loop through the specified number of slices
    for i in range(num_slices):
        start_idx = i * slice_size
        end_idx = (i + 1) * slice_size if (i < num_slices - 1) else len(df)

        # Slice the data, labels, and get the unique temperature value for this slice
        sliced_data.append(df['Data'][start_idx:end_idx].tolist())
        sliced_labels.append(df['Labels'][start_idx:end_idx].tolist())
        sliced_temperatures.append(df['Temperature'][start_idx:end_idx].unique()[0])

    # Return the sliced data, labels, and temperatures
    return sliced_data, sliced_labels, sliced_temperatures

# Define the number of slices
num_slices = 16

# Apply the slicing function to the DataFrame 'df'
sliced_data, sliced_labels, sliced_temperatures = slice_data(df, num_slices)

# Display information about each slice
```

```python
for i, data_slice in enumerate(sliced_data):
    print(f"Slice {i + 1}:")
    print(f"Data Shape: {np.array(data_slice).shape}")
    print(f"Labels Shape: {np.array(sliced_labels[i]).shape}")
    print(f"Temperature Value: {sliced_temperatures[i]}\n")


# %%
# Loop through the slices of data
for i in range(num_slices):
    slice_data = np.array(sliced_data[i])
    slice_labels = np.array(sliced_labels[i])
    slice_temperature = sliced_temperatures[i]

    # Create a DataFrame for the current slice
    df_slice = pd.DataFrame(slice_data)
    df_slice['Label'] = slice_labels  # Add a 'Label' column
    df_slice['Temperature'] = slice_temperature  # Add a 'Temperature' column

    # Display information about the DataFrame
    print(f"DataFrame for Slice {i+1}")

    # Print the first few rows of the DataFrame
    print(df_slice.head())

    # Print the shape (number of rows and columns) of the DataFrame
    print(f"Shape of Slice {i+1}: {df_slice.shape}\n")


# %%
import numpy as np
import matplotlib.pyplot as plt

# Define the size of the figure
plt.figure(figsize=(10, 6))

# Loop over the slices of data
for i in range(num_slices):
    slice_data = np.array(sliced_data[i])
    slice_labels = np.array(sliced_labels[i])
    slice_temperature = sliced_temperatures[i]

    # Calculate Magnetization
    mag = np.mean(slice_data, axis=1)

    # Plot the Magnetization vs Temperature
    plt.plot([slice_temperature] * len(mag), mag, 'o', label=f'T={slice_temperature}')

# Add a vertical line at the critical temperature (2.26K) for reference
plt.axvline(x=2.26, color='red', linestyle='--', label='Critical Temperature (2.26K)')
```

```
# Set labels for the x and y axes
plt.xlabel('Temperature')
plt.ylabel('Magnetization')

# Set the title of the plot
plt.title('Magnetization vs Temperature for Different Slices')

# Add a legend to the plot
plt.legend()

# Display the plot
plt.show()


# %%
# Create a figure with a specific size
plt.figure(figsize=(10, 6))

# Loop through the slices of data
for i in range(num_slices):
    slice_labels = np.array(sliced_labels[i])
    slice_temperature = sliced_temperatures[i]

    # Create a scatter plot of state (Ordered/Disordered) vs Temperature
    plt.scatter([slice_temperature] * len(slice_labels), slice_labels, alpha=0.5, label=f'T={slice_temperature}')

# Add a vertical dashed red line at the critical temperature (2.26K)
plt.axvline(x=2.26, color='red', linestyle='--', label='Critical Temperature (2.26K)')

# Set labels for the x and y axes
plt.xlabel('Temperature')
plt.ylabel('State (Ordered/Disordered)')

# Set the title of the plot
plt.title('Ordered and Disordered States at Different Temperatures')

# Add a legend to the plot
plt.legend()

# Display the plot
plt.show()


# %%
# Define the critical temperature
critical_temp = 2.26

# Define temperature ranges for the ordered and unordered regions
lower_bound_ordered = 0.25
```

```
upper_bound_ordered = 2.25
lower_bound_unordered = 2.5001
upper_bound_unordered = 4.001

# Separate the data into ordered and unordered regions
ordered_data = df[(df['Temperature'] >= lower_bound_ordered) & (df['Temperature'] <= upper_bound_ordered)]
unordered_data = df[(df['Temperature'] >= lower_bound_unordered) & (df['Temperature'] <= upper_bound_unordered)]

# Define the critical temperature region
critical_data = df[(df['Temperature'] > upper_bound_ordered) & (df['Temperature'] < lower_bound_unordered)]

# Print the temperature range and the number of data points in the critical temperature region
critical_temp_range = (upper_bound_ordered, lower_bound_unordered)
critical_data_count = len(critical_data)
print(f"Critical Temperature Region: {critical_temp_range}")
print(f"Number of Data Points in Critical Temperature Region: {critical_data_count}")



# %%
# Print the temperature range and the number of data points for the ordered data
ordered_temp_range = (lower_bound_ordered, upper_bound_ordered)
ordered_data_count = len(ordered_data)
print(f"Ordered Data Temperature Range: {ordered_temp_range}")
print(f"Number of Ordered Data Points: {ordered_data_count}")

# Print the temperature range and the number of data points for the unordered data
unordered_temp_range = (lower_bound_unordered, upper_bound_unordered)
unordered_data_count = len(unordered_data)
print(f"Unordered Data Temperature Range: {unordered_temp_range}")
print(f"Number of Unordered Data Points: {unordered_data_count}")



# %%
# After splitting the ordered data, check the dimensions
x_ordered_train, x_ordered_test, y_ordered_train, y_ordered_test = train_test_split(
    ordered_data['Data'].values, ordered_data['Labels'].values, test_size=test_size, random_state=42)

# Print the dimensions to ensure the data is split correctly
print("Ordered Data Set:")
print(f"x_ordered_train.shape: {x_ordered_train.shape}")
print(f"x_ordered_test.shape: {x_ordered_test.shape}")
print(f"y_ordered_train.shape: {y_ordered_train.shape}")
print(f"y_ordered_test.shape: {y_ordered_test.shape}")

# After splitting the unordered data, check the dimensions
x_unordered_train, x_unordered_test, y_unordered_train, y_unordered_test = train_test_split(
    unordered_data['Data'].values, unordered_data['Labels'].values, test_size=val_size, random_state=42)

# Print the dimensions to ensure the data is split correctly
print("\nUnordered Data Set:")
```

```
print(f"x_unordered_train.shape: {x_unordered_train.shape}")
print(f"x_unordered_test.shape: {x_unordered_test.shape}")
print(f"y_unordered_train.shape: {y_unordered_train.shape}")
print(f"y_unordered_test.shape: {y_unordered_test.shape}")



# %%
# Define the critical temperature
critical_temp = 2.26

# Define temperature ranges for the ordered and unordered regions
lower_bound_ordered = 0.25
upper_bound_ordered = 2.25
lower_bound_unordered = 2.5001
upper_bound_unordered = 4.001

# Separate the data into ordered and unordered regions
ordered_data = df[(df['Temperature'] >= lower_bound_ordered) & (df['Temperature'] <= upper_bound_ordered)]
unordered_data = df[(df['Temperature'] >= lower_bound_unordered) & (df['Temperature'] <= upper_bound_unordered)]

# Define the critical temperature region
critical_data = df[(df['Temperature'] > upper_bound_ordered) & (df['Temperature'] < lower_bound_unordered)]

# Set a flag to determine whether to split the data
split = True

# Split the data based on the 'split' flag
if split:
    x_critical_train, x_critical_test, y_critical_train, y_critical_test = train_test_split(
        critical_data['Data'].values, critical_data['Labels'].values, test_size=test_size, random_state=42)
else:
    x_critical_train, x_critical_test = critical_data['Data'].values, None
    y_critical_train, y_critical_test = critical_data['Labels'].values, None

# Split the ordered and unordered data into training and testing sets
x_ordered_train, x_ordered_test, y_ordered_train, y_ordered_test = train_test_split(
    ordered_data['Data'].values, ordered_data['Labels'].values, test_size=test_size, random_state=42)

x_unordered_train, x_unordered_test, y_unordered_train, y_unordered_test = train_test_split(
    unordered_data['Data'].values, unordered_data['Labels'].values, test_size=test_size, random_state=42)

# Print the results
print("Critical Region Data:")
print(f"x_critical_train.shape: {x_critical_train.shape}")
print(f"x_critical_test.shape: {x_critical_test.shape if split else 'Not Split'}")
print(f"y_critical_train.shape: {y_critical_train.shape}")
print(f"y_critical_test.shape: {y_critical_test.shape if split else 'Not Split'}")

print("\nOrdered Data:")
print(f"x_ordered_train.shape: {x_ordered_train.shape}")
```

```
print(f"x_ordered_test.shape: {x_ordered_test.shape}")
print(f"y_ordered_train.shape: {y_ordered_train.shape}")
print(f"y_ordered_test.shape: {y_ordered_test.shape}")

print("\nUnordered Data:")
print(f"x_unordered_train.shape: {x_unordered_train.shape}")
print(f"x_unordered_test.shape: {x_unordered_test.shape}")
print(f"y_unordered_train.shape: {y_unordered_train.shape}")
print(f"y_unordered_test.shape: {y_unordered_test.shape}")



# %%
# Concatenate the training sets from the ordered and unordered regions
x_train = np.concatenate((x_ordered_train, x_unordered_train))
y_train = np.concatenate((y_ordered_train, y_unordered_train))

# Concatenate the test sets from the ordered and unordered regions
x_test = np.concatenate((x_ordered_test, x_unordered_test))
y_test = np.concatenate((y_ordered_test, y_unordered_test))

# Print the shapes of the combined sets
print("Combined Training Set:")
print(f"x_train shape: {x_train.shape}")
print(f"y_train shape: {y_train.shape}")

print("\nCombined Test Set:")
print(f"x_test shape: {x_test.shape}")
print(f"y_test shape: {y_test.shape}")



# %%
# Specify the value of L
L = 40

# Initialize lists to store the reshaped images
x_train_images = []
x_test_images = []

# Reshape the combined training set
for i in range(x_train.shape[0]):
    image = x_train[i].reshape(L, L, 1)
    x_train_images.append(image)

# Reshape the combined test set
for i in range(x_test.shape[0]):
    image = x_test[i].reshape(L, L, 1)
    x_test_images.append(image)

# Convert the lists to numpy arrays
x_train_images = np.array(x_train_images)
```

```
x_test_images = np.array(x_test_images)

# Check the dimensions of the reshaped data
print("Combined Non-Critical Training Set:")
print(f"x_train_images.shape: {x_train_images.shape}")

print("Combined Non-Critical Test Set:")
print(f"x_test_images.shape: {x_test_images.shape}")


# %%
from tensorflow.keras import layers, models

# Define the CNN model
model = models.Sequential()

# Convolutional Layer 1
model.add(layers.Conv2D(3, (3, 3), activation='tanh', input_shape=(L, L, 1)))
# You can uncomment the following lines to add max-pooling and additional convolutional layers
# model.add(layers.MaxPooling2D((2, 2)))

# Convolutional Layer 2 (You can uncomment this section if needed)
# model.add(layers.Conv2D(64, (3, 3), activation='relu'))
# model.add(layers.MaxPooling2D((2, 2)))

# Convolutional Layer 3 (You can uncomment this section if needed)
# model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Flatten the output
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))  # Dropout layer to reduce overfitting

# Fully Connected Layers (You can uncomment this section if needed)
# model.add(layers.Dense(64, activation='relu'))

# Output layer for binary classification
model.add(layers.Dense(1, activation='sigmoid'))

# Compile the model with optimizer, loss function, and metrics
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Print model summary
model.summary()


# %%
# Train the model using training data and labels and obtain the history object
history = model.fit(x_train_images, y_train,
```

```
                epochs=3,  # Number of training epochs
                validation_split=0.1)  # 10% validation split


# %%
# Test the model on test data and labels
score = model.evaluate(x_test_images, y_test, verbose=0)

# Print the test loss and test accuracy
print(f"Test Loss: {score[0]:.4f}")
print(f"Test Accuracy: {score[1]:.4f}")


# %%
# Create a figure with two subplots side by side
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot training & validation loss values
ax1.plot(history.history['loss'], label='Train')
ax1.plot(history.history['val_loss'], label='Validation')
ax1.set_title('Model Loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.legend(loc='upper right')

# Plot training & validation accuracy values
ax2.plot(history.history['accuracy'], label='Train')
ax2.plot(history.history['val_accuracy'], label='Validation')
ax2.set_title('Model Accuracy')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy')
ax2.legend(loc='lower right')

# Adjust spacing between subplots
plt.tight_layout()

# Show the plot
plt.show()

from sklearn.metrics import precision_score, recall_score, f1_score

# Assuming 'y_test' contains the actual labels and 'y_pred' contains the predicted labels by your model
y_pred = model.predict(x_test_images)
y_pred_binary = (y_pred > 0.5).astype(int)  # Convert predicted probabilities to binary predictions (0 or 1)

# Calculate precision, recall, and f1-score
precision = precision_score(y_test, y_pred_binary)
recall = recall_score(y_test, y_pred_binary)
f1 = f1_score(y_test, y_pred_binary)
```

```
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
```

3.1.4.3 Prediction and Evaluation of CNN Model

In this part of the project, we have undertaken the task of analyzing the phase transitions of the Ising model using machine learning techniques, specifically employing a Convolutional Neural Network (CNN) for the binary classification of states into ordered and disordered phases. The dataset, acquired from the online repository provided by Prof. Pankaj Mehta's research group, consists of 160,000 samples each with 1,600 binary features that represent spin configurations on a 40x40 lattice.

The initial data processing steps involved the extraction and decompression of binary files, followed by the reshaping of the dataset to fit the requirements of a 2D Ising model. Subsequently, these configurations were mapped to binary labels in accordance with the conventional interpretation of the Ising model's states, where '1' denotes an ordered state, and '0' signifies a disordered state. This critical preprocessing stage was accomplished using Python libraries such as numpy for array manipulation and pandas for managing structured data.

The exploratory data analysis (EDA) included visualizing the magnetization trends across different temperature slices, which provided a preliminary validation of the expected physical behavior; a high degree of magnetization at low temperatures (ordered phase) and a significant reduction as the temperature approached the critical threshold, beyond which the system transitions into the disordered phase.

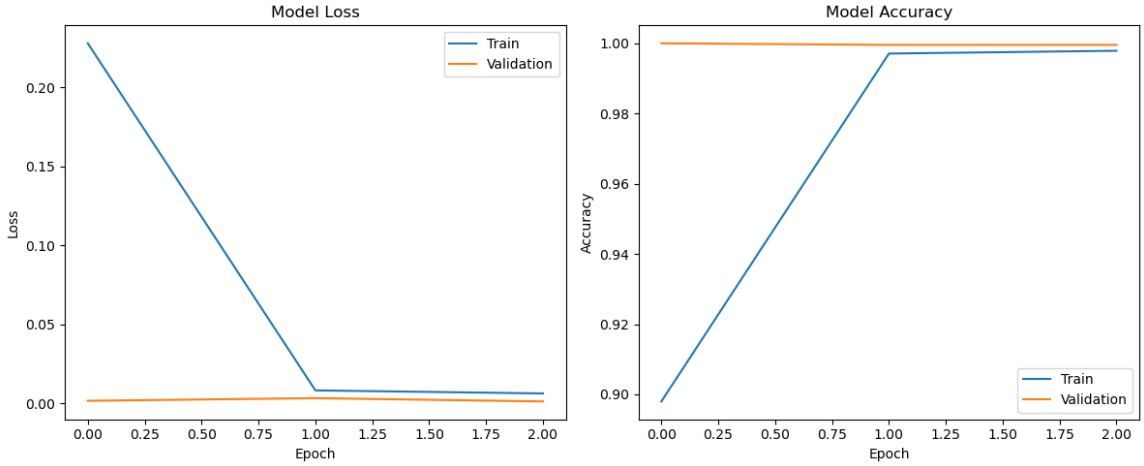
The machine learning model employed was a Sequential model from the Keras library, featuring a single convolutional layer with a tanh activation function. The choice of a relatively simple architecture was made to test the hypothesis that phase transitions in the Ising model can be effectively captured by CNNs without necessitating complex network structures. The output layer used a sigmoid function to facilitate binary classification. The

compilation of the model was done using the 'adam' optimizer and 'binary_crossentropy' as the loss function, which is typical for binary classification tasks.

Training was conducted over three epochs with a 10% validation split, allowing for quick iterative feedback on the model's performance. The model's accuracy was notably high, achieving near-perfect classification on the test dataset. This is a significant result, indicating the potential of CNNs to discern order from disordered phases with high reliability.

Test Loss - 0.0005
Test Accuracy - 0.99
Precision: 0.99
Recall: 0.99
F1-Score: 0.99

The evaluation of the model was not limited to accuracy metrics alone; loss metrics were also plotted over the training epochs, providing insight into the convergence behavior of the model and the effectiveness of the learning process.



The left graph displays a pronounced decline in both training and validation loss, indicating rapid learning and minimal overfitting. The right graph shows a corresponding increase in accuracy for both the training and validation sets, reaching near-perfect classification by the third epoch.

In conclusion, this project has demonstrated the efficacy of CNNs in identifying phase transitions within the Ising model. The high accuracy of our model, coupled with the

consistent performance on validation data, suggests that even simple CNN architectures are capable of capturing complex patterns in data.

4. Comparison of Algorithms

Based on the performance metrics provided for logistic regression, random forest, MLP (Multi-Layer Perceptron), and CNN (Convolutional Neural Network), it is evident that CNN achieved the highest overall correctness (accuracy) at 99.9%, closely followed by random forest at 99.2%, MLP at 98.9%, and logistic regression at 65%. The accuracy of positive predictions (precision) is consistently high for all models, with CNN, random forest, and MLP achieving scores above 98%. However, logistic regression lags significantly behind with a precision of 61%. In terms of the ability to capture all positive instances (recall), CNN, random forest, and MLP perform similarly well at around 99%, while logistic regression excels in recall at 98%. The harmonic mean of precision and recall (f1-score) is consistently high for CNN, random forest, and MLP, reflecting a balanced trade-off between precision and recall. Logistic regression, however, has a lower f1-score at 75%, indicating a less optimal balance between precision and recall. Overall, CNN appears to outperform the other models in terms of overall correctness and precision-recall balance, making it a strong choice for the given task.