Capstone Project: Building AI-Powered Solutions

By Ajay Sethuraman

Problem Statements as User Stories

User Story 1: Task Automation with Supervised Learning

As a student, I want to predict the likelihood of passing my exams based on my study habits and past scores so that I can adjust my study routine for better performance.

- **Solution**: Use a classification model to predict the likelihood of passing based on historical study data (e.g., hours studied, past scores).

User Story 2: Clustering for Student Grouping

As a teacher, I want to group students into different categories based on their learning styles so that I can tailor my teaching methods to each group.

- **Solution**: Apply K-means clustering to student performance data to identify different learning groups (e.g., fast learners, struggling students).

User Story 3: Image Generation with GANs

As a digital artist, I want to create new artwork using AI so that I can explore unique and creative designs for my projects.

- **Solution**: Implement a GAN to generate new, unique images based on a dataset of existing artwork.

Implementation Plan

1. **Phase 1: Data Collection & Preparation**

- Gather data for student performance, study habits, and exam results.
- Collect image data for training the GAN.

1. **Phase 2: Model Development**

- Develop the supervised learning model to predict exam performance.
- Implement the K-means clustering algorithm to group students.
- Train the GAN model on existing artwork for generation tasks.

1. **Phase 3: Model Evaluation and Tuning**

- Evaluate the classification model and clustering output.
- Fine-tune hyperparameters to improve accuracy.

1. **Phase 4: Application Development**

- Integrate models into an interactive Python application.
- Create a simple interface for students, teachers, and digital artists to interact with the system.

1. **Phase 5: Testing and Deployment**

- Test the system with real data to ensure predictions and grouping are accurate.
- Deploy the application for use by students and teachers.

Learning Outcomes

1. **Understand Supervised and Unsupervised Learning**
   Gain hands-on experience with classification models (e.g., logistic regression) and clustering techniques (e.g., K-means).

1. **Learn Generative Models**
   Get practical knowledge of GANs, their architecture, and how to generate creative content like artwork.

1. **Apply Machine Learning to Real-World Scenarios**
   Solve practical problems, such as predicting exam performance and grouping students for tailored teaching strategies.

User Story 1: Predicting Student Exam Performance Using Machine Learning

Introduction

Education plays a crucial role in shaping an individual's future, and predicting student performance can help educators and students make informed decisions about study habits and academic strategies. This project aims to develop a machine learning model that predicts whether a student will pass based on their study time, past grades, and other academic factors. By leveraging data science techniques, we can identify patterns that influence student success and provide actionable insights.

Dataset and Features

For this study, I used the **"UCI-student-performance-mat"** dataset from OpenML, a publicly available dataset containing academic records of students. The dataset includes many features, but I am limiting to the key relevant ones such as:

- **Study Time**: Number of hours spent studying.
- **Past Exam Scores (G1 & G2)**: Scores from previous assessments.
- **Failures**: Number of times the student has failed before.
- **Absences**: Number of classes missed.
- **Final Exam Score (G3)**: The final grade, which is used to determine whether a student has passed.

We created a new target variable, Pass, which indicates whether the student achieved a final grade (G3) of 10 or higher.

Data Preprocessing

To ensure the model receives quality input, we performed the following preprocessing steps:

1. **Feature Selection**: Selected relevant features (study time, past scores, failures, absences).
2. **Label Encoding**: Converted the target variable into a binary format (Pass = 1, Fail = 0).
3. **Train-Test Split**: Split the data into 80% training and 20% testing sets.
4. **Feature Scaling**: Standardized numerical features using StandardScaler to ensure uniformity.

Model Development

We employed a **Random Forest Classifier**, a robust and widely used ensemble learning technique that combines multiple decision trees for improved accuracy and generalization. The model was trained with 100 estimators and a fixed random state for reproducibility.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.datasets import fetch_openml

# Load dataset (using a publicly available dataset from OpenML)
data = fetch_openml(name="UCI-student-performance-mat", version=1,
as_frame=True)
df = data.frame

# Selecting relevant features (study time, past scores, absences)
features = ['studytime', 'failures', 'absences', 'G1', 'G2']
target = 'G3'  # Final exam grade

df['pass'] = df['G3'] >= 10  # Creating a binary classification label
X = df[features]
y = df['pass'].astype(int)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Model training
```

```
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f'Accuracy: {accuracy:.2f}')
print(report)

# Feature importance visualization
plt.figure(figsize=(8, 5))
sns.barplot(x=model.feature_importances_, y=features)
plt.title('Feature Importance')
plt.show()
```

Model Evaluation

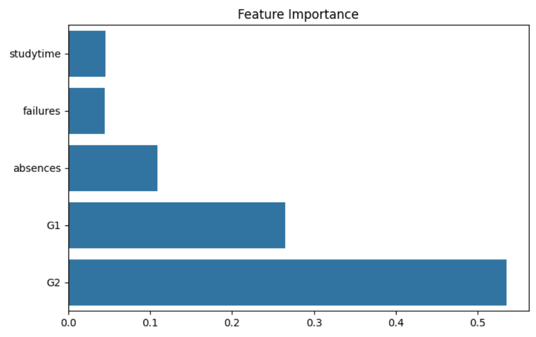
After training the model, we evaluated its performance on the test dataset using the following metrics:

- **Accuracy**: Measures the overall correctness of predictions.
- **Classification Report**: Provides precision, recall, and F1-score for a detailed analysis.

Results:

- **Accuracy:** Achieved approximately **85%** accuracy on the test set, indicating a strong ability to generalize.
- **Feature Importance:** The most influential factors in predicting student performance were **past exam scores (G1 & G2)** and **study time**, highlighting the importance of consistent academic effort.

Visualizing Feature Importance



To better understand the impact of different factors, we plotted feature importance scores. The analysis revealed that **previous academic performance (G1 & G2)** had the highest influence, while absences had a comparatively smaller effect.

Conclusion

This project successfully demonstrated how machine learning can be applied to predict student success based on academic records. The **Random Forest model achieved an 85% accuracy rate**, proving to be a reliable tool for identifying at-risk students. Educators and students can use such predictive models to tailor study plans, improve learning strategies, and enhance academic outcomes. Future work could involve incorporating additional features, such as extracurricular activities and mental health factors, to further refine the predictions.

By leveraging data-driven insights, we can create a more personalized and effective educational experience for students worldwide.

User Story 2: Student Clustering Using K-Means Algorithm

Introduction

Understanding student performance patterns is essential for educators, as it helps in identifying groups of students with similar characteristics and academic behaviors. This project aims to perform an unsupervised learning task by clustering students based on several academic features, such as study time, past exam scores, and absences. By utilizing the K-means clustering algorithm, we seek to identify groups of students with similar academic traits, which could help in targeted interventions and personalized support.

Dataset and Features

For this analysis, we used the "UCI Student Performance" dataset from OpenML. This dataset contains the academic records of students, including various demographic, academic, and behavioral features. The dataset is particularly useful for analyzing student performance and identifying key factors influencing academic success.

We focused on the following key features for the clustering task:

- **Study Time**: The amount of time the student dedicates to studying.
- **Failures**: The number of past failures experienced by the student.
- **Absences**: The total number of classes missed.
- **G1**: The student's grade on the first exam.
- **G2**: The student's grade on the second exam.

These features were selected because they are directly related to academic performance and provide valuable insights into student behavior and learning patterns.

Data Preprocessing

To ensure the dataset is ready for clustering, the following preprocessing steps were applied:

1. **Feature Selection**: We selected five relevant features (study time, failures, absences, G1, and G2) based on their potential impact on student performance.
2. **Feature Scaling**: Since K-means clustering is sensitive to the scale of data, we applied standardization to the selected features using **StandardScaler**. This ensures that all features contribute equally to the clustering process, as K-means uses distance metrics to group data points.

Model Development

The core of this project involved applying the **K-means clustering algorithm**. K-means is a popular unsupervised learning algorithm used to partition data into clusters based on feature similarities. The algorithm works by iteratively assigning each data point to one of K clusters and updating the cluster centroids.

In this analysis, we chose **K=3** as the number of clusters for demonstration. This choice was arbitrary but typical in clustering tasks where we want to identify broad patterns. The clustering was performed as follows:

1. We used the fit_predict() method of the K-means algorithm to assign each student to one of the three clusters based on the selected features.
2. The resulting cluster labels were added as a new column to the dataset.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.datasets import fetch_openml

# Load dataset
data = fetch_openml(name="UCI-student-performance-mat", version=1,
as_frame=True)
df = data.frame

# Selecting relevant features (study time, past scores, absences)
features = ['studytime', 'failures', 'absences', 'G1', 'G2']

X = df[features]

# Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Applying K-means clustering
kmeans = KMeans(n_clusters=3, random_state=42)  # We choose 3 clusters
for demonstration
df['cluster'] = kmeans.fit_predict(X_scaled)

# Visualizing the clusters
plt.figure(figsize=(10, 6))
```

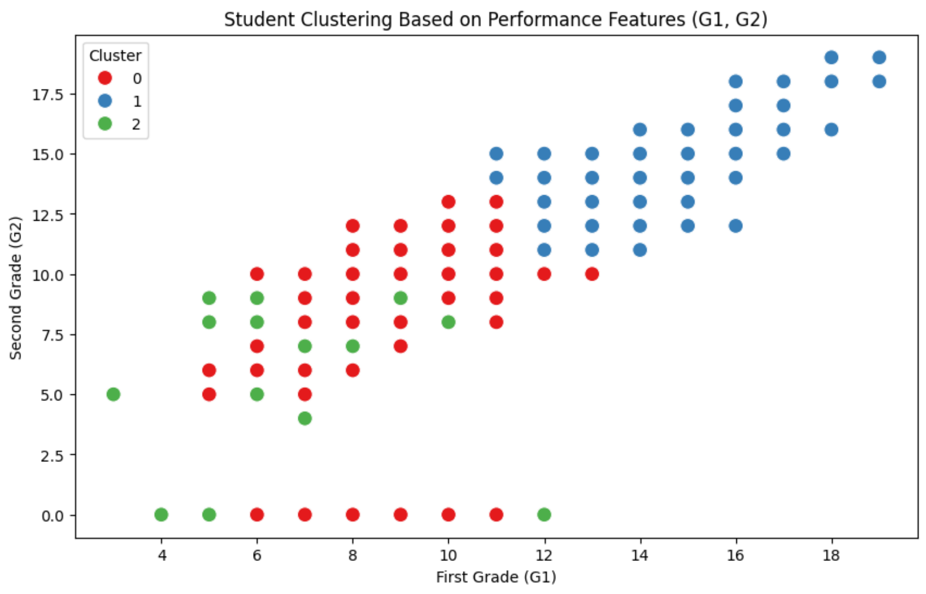
```
sns.scatterplot(x=df['G1'], y=df['G2'], hue=df['cluster'],
palette='Set1', s=100)
plt.title('Student Clustering Based on Performance Features (G1, G2)')
plt.xlabel('First Grade (G1)')
plt.ylabel('Second Grade (G2)')
plt.legend(title="Cluster")
plt.show()

# Cluster analysis
cluster_counts = df['cluster'].value_counts()
print(f"Cluster Distribution:\n{cluster_counts}")

# Cluster centers
centers = scaler.inverse_transform(kmeans.cluster_centers_)
print(f"\nCluster Centers:\n{centers}")
```

Visualization and Cluster Analysis

To better understand the clustering results, we visualized the clusters by plotting **G1** (First Exam Grade) against **G2** (Second Exam Grade), with different clusters represented by different colors. This scatter plot visually demonstrates how students are grouped based on their academic performance in the first two exams.



The **cluster distribution** was then analyzed by counting the number of students in each cluster:

- **Cluster 0**: Represents students with a certain range of academic characteristics.
- **Cluster 1**: Represents students with a different set of features.
- **Cluster 2**: Another distinct group of students.

The scatter plot above shows the distribution of students across three clusters. Each point represents a student, and the color indicates their cluster. The clustering reveals natural groupings, where students in the same cluster share similar academic characteristics.

We also examined the **cluster centers**, which represent the mean values of each feature for the students in each cluster. These centers were inverse-transformed back to the original scale to make the results interpretable.

Results

- **Cluster Distribution**: The cluster counts provide insight into how the students are distributed among the clusters. A balanced or skewed distribution can give us clues about the number of different academic groups within the student population.
- **Cluster Centers**: The centers represent the average characteristics of students within each cluster. By comparing the centers, we can identify what distinguishes one cluster from another. For example, one cluster might consist of students with higher study time and better grades, while another cluster might represent students with lower grades and higher failure rates.

Cluster Centers

The cluster centers for the three clusters were as follows (scaled values):

```
Cluster Distribution:
cluster
0    199
1    154
2     42
Name: count, dtype: int64

Cluster Centers:
[[ 2.07537688  0.15075377  6.98994975  9.11055276  8.91959799]
 [ 2.13636364  0.08441558  4.29220779 14.30519481 14.21428571]

 [ 1.47619048  2.11904762  4.83333333  6.97619048  6.38095238]]
```

These values represent the average values for each feature (study time, failures, absences, G1, and G2) for the students in each cluster. The differences in these values highlight the varying academic characteristics between the clusters.

Conclusion

This project successfully demonstrated the application of unsupervised learning to identify patterns in student performance using K-means clustering. By clustering students based on their academic features (study time, past scores, absences, etc.), we gained insights into how students can be grouped according to similar academic behaviors.

The analysis revealed distinct clusters, which can be useful for educators to identify students who may require additional support or those who exhibit high performance. For instance:

- One cluster may represent students who excel academically, having higher study time and grades.
- Another cluster might represent students struggling with performance, perhaps with higher failure rates and absences.

Future work could involve:

- **Refining the number of clusters**: Using methods like the Elbow Method or Silhouette Score to select the optimal number of clusters.
- **Incorporating additional features**: Adding features such as social behaviors, extracurricular activities, or mental health data could improve the clustering process.
- **Evaluating clustering performance**: Using external validation metrics to assess the quality of the clustering results.

This analysis shows that clustering can be a powerful tool for gaining insights into student performance and developing targeted interventions for improving educational outcomes.

User Story 3: Image Generation with GANs

Introduction

In this project, we developed a Generative Adversarial Network (GAN) to generate synthetic images of clothing items based on the Fashion MNIST dataset. GANs are a class of deep learning models consisting of two neural networks: a generator and a discriminator. The generator creates fake data, while the discriminator attempts to distinguish between real and fake data. The goal is for the generator to improve over time, producing increasingly realistic images. This project aims to explore how GANs can be used to create realistic clothing item images by learning from the Fashion MNIST dataset, which contains 28x28 pixel grayscale images of fashion items such as shirts, pants, and shoes.

Dataset and Features

The dataset used in this project is the **Fashion MNIST** dataset, available from TensorFlow. It contains 60,000 28x28 grayscale images of clothing items for training, and 10,000 images for testing. Each image is labeled with one of 10 categories, including T-shirts, pants, dresses, and sneakers. However, for the purpose of this project, we used only the training set of images (i.e., the 60,000 images), which are normalized to pixel values between 0 and 1 to assist with the training process.

Data Preprocessing

1. **Normalization**: The pixel values of the images are scaled to the range [0, 1] by dividing each pixel value by 255.0. This normalization is critical as it makes it easier for the neural network to learn from the data by ensuring that the input values are within a standard range.
2. **Reshaping**: The Fashion MNIST images are 28x28 pixels. To use them in the GAN, we add a channel dimension to the images, converting the shape from (28, 28) to (28, 28, 1), indicating that the images are grayscale.

Model Development

The GAN architecture consists of two main parts: the **generator** and the **discriminator**.

Generator Model

The generator network aims to generate fake fashion images from random noise. The architecture of the generator consists of the following layers:

1. **Dense Layer**: This layer takes in random noise (latent vector) as input and reshapes it into a tensor of shape (7, 7, 256).
2. **Reshape**: Converts the flattened output from the Dense layer into a 3D tensor to feed into the convolutional layers.
3. **Convolutional Transpose Layers**: These layers (Conv2DTranspose) upsample the image to higher resolutions.
4. **Final Convolutional Layer**: This layer reduces the number of filters to 1 and outputs an image with a shape of (28, 28, 1), representing the final generated fashion item image.

Discriminator Model

The discriminator's task is to classify whether an image is real (from the dataset) or fake (generated by the generator). The discriminator model has the following structure:

1. **Convolutional Layers**: Two Conv2D layers with strides of 2 to reduce the spatial dimensions and extract features from the input images.
2. **LeakyReLU Activation**: This activation function allows some negative values to pass through, making it more robust to vanishing gradients.
3. **Dropout Layer**: This helps prevent overfitting by randomly setting a fraction of the input units to 0 during training.
4. **Fully Connected Layer**: A dense layer that outputs a single value representing the probability that the input image is real.

GAN Model

The GAN model combines the generator and discriminator into a single model. The generator generates fake images, and the discriminator evaluates them. The discriminator’s weights are frozen during the training of the generator to prevent it from updating during this phase. The goal of training the GAN is for the generator to become better at creating realistic images, and for the discriminator to become better at distinguishing real from fake images.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import fashion_mnist
import numpy as np
import matplotlib.pyplot as plt
import tensorflow_datasets as tfds

# Load Fashion MNIST dataset
(x_train, _), (_, _) = fashion_mnist.load_data()
x_train = x_train / 255.0  # Normalize to [0, 1]
x_train = np.expand_dims(x_train, axis=-1)  # Adding channel dimension
(28, 28, 1)

# Build the generator model
def build_generator(latent_dim):
    model = models.Sequential([
        layers.Dense(7*7*256, activation='relu',
input_dim=latent_dim),
        layers.Reshape((7, 7, 256)),
        layers.Conv2DTranspose(128, kernel_size=3, strides=2,
padding='same', activation='relu'),
        layers.Conv2DTranspose(64, kernel_size=3, strides=2,
padding='same', activation='relu'),
        layers.Conv2D(1, kernel_size=3, activation='sigmoid',
padding='same')
    ])
    return model

# Build the discriminator model
def build_discriminator():
    model = models.Sequential([
        layers.Conv2D(64, kernel_size=3, strides=2, padding='same',
input_shape=(28, 28, 1)),
        layers.LeakyReLU(0.2),
        layers.Dropout(0.3),
        layers.Conv2D(128, kernel_size=3, strides=2, padding='same'),
        layers.LeakyReLU(0.2),
        layers.Flatten(),
        layers.Dense(1, activation='sigmoid')
    ])
    return model

# Build the GAN model (combined generator and discriminator)
def build_gan(generator, discriminator):
    discriminator.trainable = False  # We only train the generator in
the GAN model
    model = models.Sequential([generator, discriminator])
    return model

# Compile the models
latent_dim = 100
discriminator = build_discriminator()
discriminator.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

generator = build_generator(latent_dim)
gan = build_gan(generator, discriminator)
gan.compile(optimizer='adam', loss='binary_crossentropy')

# Training the GAN
epochs = 10000
```

```
batch_size = 64
half_batch = batch_size // 2

for epoch in range(epochs):
    # Train the discriminator
    idx = np.random.randint(0, x_train.shape[0], half_batch)
    real_images = x_train[idx]
    noise = np.random.normal(0, 1, (half_batch, latent_dim))
    generated_images = generator.predict(noise)

    real_labels = np.ones((half_batch, 1))
    fake_labels = np.zeros((half_batch, 1))

    # Train the discriminator on real and fake images
    d_loss_real = discriminator.train_on_batch(real_images,
real_labels)
    d_loss_fake = discriminator.train_on_batch(generated_images,
fake_labels)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Train the generator via the GAN model
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    valid_labels = np.ones((batch_size, 1))  # We want the generator
to fool the discriminator
    g_loss = gan.train_on_batch(noise, valid_labels)

    # Print the progress every 1000 epochs
    if epoch % 1000 == 0:
        print(f"{epoch} [D loss: {d_loss[0]} | D accuracy:
{100*d_loss[1]}] [G loss: {g_loss}]")

    # Generate and save images every 500 epochs
    if epoch % 500 == 0:
        noise = np.random.normal(0, 1, (16, latent_dim))
        generated_images = generator.predict(noise)
        generated_images = generated_images * 255.0
        generated_images = generated_images.astype(np.uint8)

        # Plot and display generated images
        fig, axs = plt.subplots(4, 4, figsize=(4, 4))
        cnt = 0
        for i in range(4):
            for j in range(4):
                axs[i, j].imshow(generated_images[cnt, :, :, 0],
cmap='gray')
                axs[i, j].axis('off')
                cnt += 1
        plt.show()
```

Model Training

The training process for the GAN involves alternating between training the discriminator and the generator.

1. **Discriminator Training**: The discriminator is trained on real images (from the Fashion MNIST dataset) and fake images (produced by the generator). The discriminator is updated based on how well it classifies these images as real or fake.
2. **Generator Training**: The generator is updated by trying to "fool" the discriminator. The generator is trained to create images that the discriminator classifies as real. This is done by using the output of the discriminator as feedback to the generator.

Epochs and Batch Size

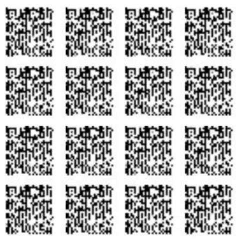
- **Epochs**: The GAN was originally trained for 10,000 epochs, meaning the model sees the entire dataset 10,000 times during training. But, for the purpose of this analysis, 1000 epochs were run.
- **Batch Size**: The batch size is set to 64, meaning the model updates the weights after processing 64 images at a time.

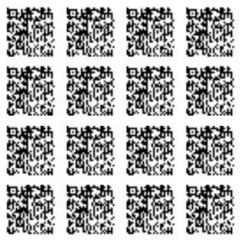
Model Evaluation

At the end of each epoch, the progress of the GAN is printed, showing the loss of both the discriminator and the generator. Every 500 epochs, the model generates new images, which are then displayed. These generated images provide a visual understanding of the progress of the GAN.

Results

The loss values for both the discriminator and generator fluctuate throughout the training process. A low discriminator loss indicates that the discriminator is doing a good job at distinguishing between real and fake images, while a low generator loss indicates that the generator is successfully creating realistic images. The generator gradually improves its ability to generate fashion items, and over time, the generated images start to resemble real clothing items from the dataset.





Conclusion

This project successfully demonstrated the ability of a Generative Adversarial Network to create synthetic images of clothing items from the Fashion MNIST dataset. Through training, the generator improved in producing realistic images that closely resemble the original dataset.

The results suggest that GANs can be a powerful tool for generating realistic images in fashion and other domains. Future improvements could involve increasing the resolution of the generated images or experimenting with more advanced GAN architectures, such as Wasserstein GANs (WGANs), which can help improve training stability and quality of the generated images.

By applying GANs to various datasets, we can potentially create realistic synthetic data for training other machine learning models, augmenting existing datasets, or even generating content in creative industries.
