Projects' Portfolio

by Oleksandra Ivanova

Table Of Contents

Deep Learning

I. Generative Adversarial Network for MNIST Dataset

II. Convolutional Neural Network for MNIST Dataset

Project I

Generative Adversarial Network for MNIST Dataset

Generative Adversarial Networks (GANs) are a class of deep learning models that consist of two neural networks, a generator, and a discriminator, which are trained simultaneously through adversarial training. For the MNIST dataset, GANs can be used to generate realistic hand-written digit images.

Step 1. Data Preparation

Firstly, I set up the future conversion of images to PyTorch tensors and normalize pixel values. The MNIST dataset is then loaded from datasets of torchvision package. The dataset is transformed using the previously defined transforms. A data loader is then set up, using a specified batch size and shuffling the data.

Step 2. Generator and Discriminator

```python
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = x.view(x.size(0), 784)
        output = self.model(x)
        return output
```

```python
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 784),
            nn.Tanh(),
        )

    def forward(self, x):
        output = self.model(x)
        output = output.view(x.size(0), 1, 28, 28)
        return output
```

The Discriminator consists of a fully connected neural network with multiple hidden layers, each followed by a ReLU activation function and dropout layers. The final layer utilizes the Sigmoid activation function to output a probability indicating the authenticity of the input. The Generator has a similar architecture, but uses the Tanh activation function in the final layer to produce values within the range of [-1, 1].

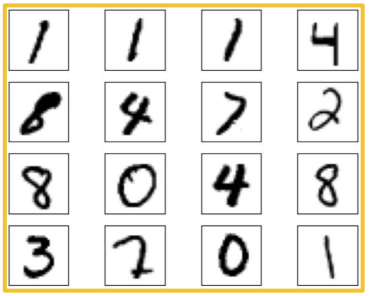


Image 1. Input Samples

Image 2. Generated Samples

The model was trained through 50 epochs, on the final evaluation presenting such results:

- **Generator loss:** 1.068444848060608
- **Discriminator loss:** 0.5816411375999451

Project 2
Convolutional Neural Network for MNIST Dataset

Convolutional Neural Networks (CNNs) are a type of deep learning algorithm designed for processing structured grid data, such as images. They are particularly effective in tasks like image recognition and classification. MNIST dataset, which consists of handwritten digits (0 through 9), CNNs are used for digit recognition.

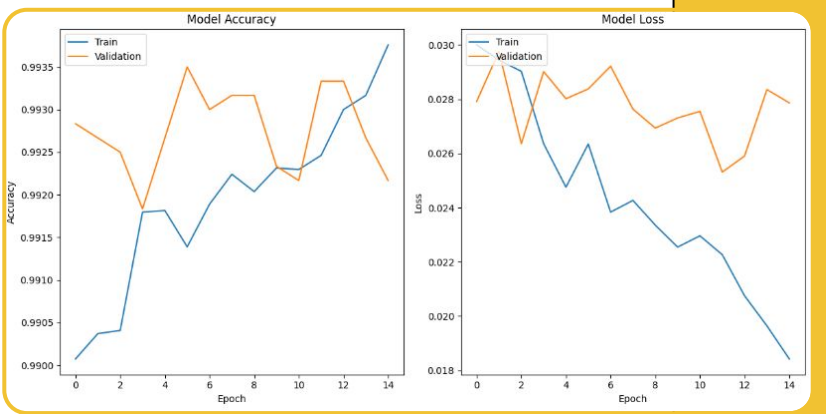
Step 1. Data Preparation

After loading the dataset I specified the number of classes for the classification and the input shapes of images for training. Using Keras potential I split data to training and testing datasets (X for images, y for labels). The tricky part is to not overlook scaling images to [0;1] and to expand their dimensions, ensuring initial shape of input. With one-hot encoding also done, I started on my model training.

Step 2. CNN model training

```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model.summary()
```

In the model used there are 8 layers. 2 convolutional layers detect patterns and features in input, number of filters increases to catch firstly basic features and then more complicated ones. After each such layer max pooling is used to reduce spatial dimensions. In the end the result is flattened to a 1-dimensional vector. With softmax activation function we connect all layers and get the end result.



The model was trained through 15 epochs, in testing evaluation presenting such results:

- **Test loss:** 0.023984204977750778
- **Test accuracy:** 0.9922000169754028

Data Mining

I. Regression for Motorbike Ambulance Calls Dataset

II. Classification for Credit Card Fraud Detection

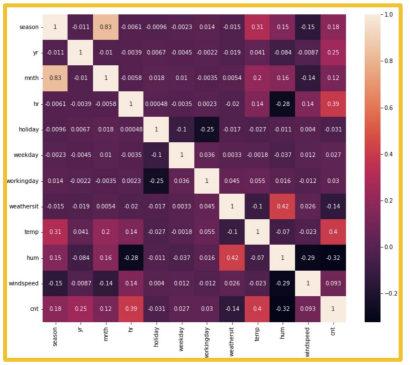
III. Extensive Data Analysis for Titanic Dataset

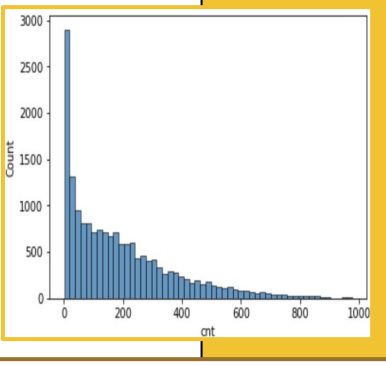
Project 1

Regression for Motorbike Ambulance Calls Dataset

Motorcycle accidents and related emergency calls are highly dependent on environmental and seasonal conditions such as weather conditions, precipitation, day of the week, season, time of day, etc. The data was collected for two years every hour and then correlated with the relevant weather and seasonality for detailed analysis.

Step 1. Data Preparation



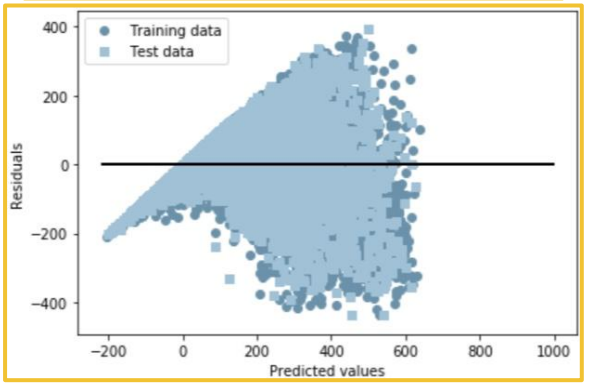


The dataset went through a lot of transformations, included in Analysis of categorical and numerical variables and their relationships. Two of the main visualizations were correlation matrices and histogram of the distribution of the target variable.

Step 2. Regression model training

```
def get_train_data(df: pd.DataFrame, target:str, test_size:float):
    X = df.drop(target, axis = 1)
    y = df[target]
    X_train, X_test, y_train, y_test =  train_test_split(X, y, test_size=test_size, random_state=4
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = get_train_data(features_lr, 'cnt', 0.3)

def train_linear_model(X_train, y_train):
    model = LinearRegression()
    model.fit(X_train, y_train)
    return model

model = train_linear_model(X_train, y_train)

# predict
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)
# scores
print('MSE train: {:.3f}, test: {:.3f}'.format(
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
print('R^2 train: {:.3f}, test: {:.3f}'.format(
        r2_score(y_train, y_train_pred),
        r2_score(y_test, y_test_pred)))
```

```
MSE train: 10470.041, test: 10112.308
R^2 train: 0.687, test: 0.680
```



Dummy variables were created, one from each category, with removal of specific attributes to prevent multicollinearity. The model is trained, and evaluations such as MSE and R^2 scores are presented, showing good predictive performance. To assess the model's quality, a plot of residuals against predicted values is generated, providing insights into model performance. Dummy variables are introduced to handle categorical data appropriately during training.

The model in train and test evaluations presented such results:

- **MSE Train:** 10470.041, **Test:** 10112.308
- **R^2 Train:** 0.687, **Test:** 0.680

Project 2

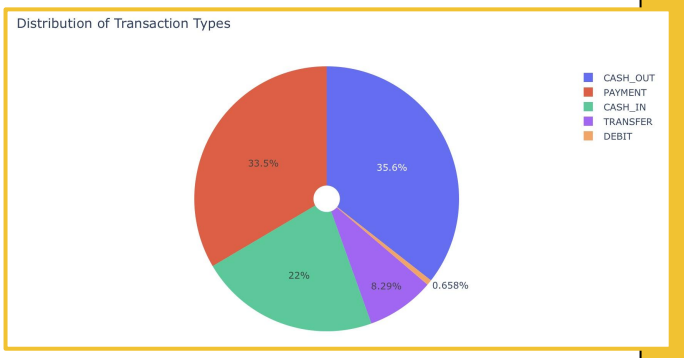
Classification for Credit Card Fraud Detection

Credit card fraud detection using classification is crucial due to the rising sophistication of fraudulent activities. Such models can identify patterns and anomalies in transaction data, distinguishing between legitimate and fraudulent transactions. This is essential for timely detection and prevention of unauthorized transactions, safeguarding both financial institutions and cardholders.

Step 1. Data Preparation

Firstly, I look into key statistics, such as mean, standard deviation, etc., aiding in understanding the distribution of numerical variables. Data cleaning is performed next, checking for missing values and handling any null entries. After that, the dataset is visualized to understand the distribution of transaction types, using a pie chart. Then I replace categorical values with numerical counterparts for model compatibility.

Step 2. Classification model training




```
X = data[['type', 'amount', 'oldbalanceOrg', 'newbalanceOrig']]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state= 42)

from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

```
DecisionTreeClassifier()
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.
```

```
print("Accuracy of classification: ", model.score(X_test, y_test))
```

```
ccuracy of classification:  0.9996499081008765
```

```
model.predict([[2,9800,170136,160296]])
```

```
usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning:

: does not have valid feature names, but DecisionTreeClassifier was fitted with feature names
```

```
array([0.])
```

The dataset is split from the target variable into X and y, further being split into training and testing sets. A Decision Tree Classifier (DTC) is chosen as the classification model and is fitted to the training data. DTC is a machine learning algorithm that makes decisions based on recursively splitting the dataset into subsets, using the most informative features at each node, ultimately forming a tree-like structure that aids in classification tasks.

The accuracy of the model is assessed on the test set, yielding an impressive accuracy of approximately **99.97%**.

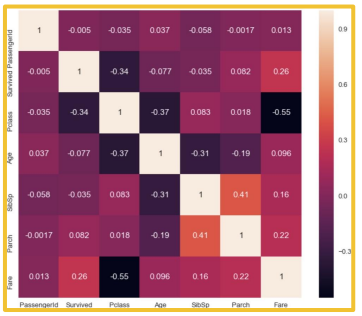
Finally, a prediction is made on a sample input representing a transaction ('type': 2, 'amount': 9800, 'oldbalanceOrg': 170136, 'newbalanceOrig': 160296). The model predicts the output class as 0, indicating a non-fraudulent transaction.

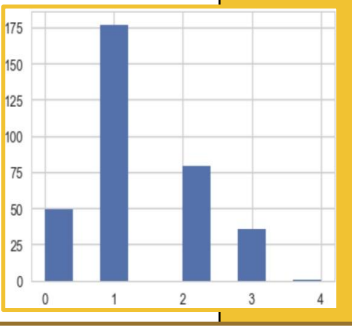
Project 3

Extensive Data Analysis for Titanic Dataset

This project was one of my first ones, and it focuses more on data pre-processing, finding patterns not by machine learning yet, but by my own observations. This project contains many expansive conclusions on every part of the pre-processing processes. One of the goals in this work was to understand what attributes influenced the chance of survival on Titanic.

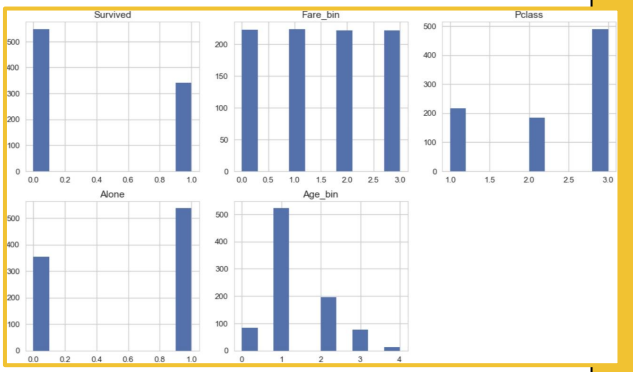
Step 1. Data Analysis

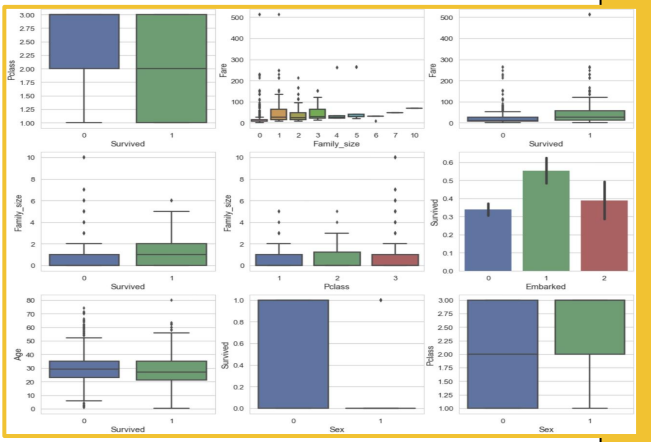




One of the most valuable changes to data was the categorization of the age variable. With that I saw that most of the passengers survived from class 1, aged 16-31 years. The fewest survivors are in class 4. Also the Pearson's correlation heatmap helped to notice hidden bonds between attributes.

Step 2. Major Conclusions





The analysis revealed that passenger class significantly influenced survival, with higher-class passengers having better chances, due to their priority in accessing lifeboats.

The link of family size to cruise cost explains the high mortality for those with 4 or more family members. Outliers in family size highlight exceptions, suggesting an impact on survival rates.

There is also an outlier in survival for those who paid the maximum amount.

Landing site affects survival as well.

Age-related survival patterns hold surprising outliers in older age groups.

A notable gender survival difference, attributed to the higher number of women, particularly in the first class, obviously. However, the discrepancy in numbers raises questions.

Overall, the analysis provided valuable insights into the factors influencing Titanic passengers' survival, giving proofs of obvious assumptions that can be made concerning the case.

Reinforcement Learning

I. Twin Delayed DDPG Algorithm for Pendulum Task

II. Soft Actor-Critic Algorithm for Mountain Car Continuous Task

Project 1

Twin Delayed DDPG Algorithm For Pendulum Task

The TD3 (Twin Delayed DDPG) algorithm is a modification of the DDPG (Deep Deterministic Policy Gradient) algorithm aimed at improving its stability and performance. TD3 uses two Q-functions (critics) to evaluate the value of actions. This approach helps to avoid too abrupt a change in strategy, which can lead to learning failure. To combat the discontinuities in strategy that can occur due to periodic updates, TD3 uses artificially added noise to selected actions to help smooth out strategic changes.

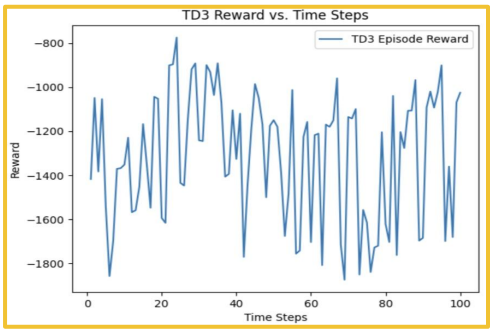
Step 1. Environment Preparation

The environment used is "Pendulum-v1" from OpenAI. This is a classic reinforcement learning environment designed to test and develop algorithms that can effectively learn control policies. In the Pendulum environment, the agent has to control the movement of a pendulum, attempting to keep it upright by applying torque at its pivot point.

Step 2. Actor and Critic models training

```python
class ActorTD3(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(ActorTD3, self).__init__()
        self.fc1 = nn.Linear(state_dim, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, action_dim)
        self.max_action = max_action

    def forward(self, state):
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        x = torch.tanh(self.fc3(x)) * self.max_action
        return x
```

```python
class CriticTD3(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(CriticTD3, self).__init__()
        self.fc1 = nn.Linear(state_dim + action_dim, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, 1)

    def forward(self, state, action):
        x = torch.cat([state, action], dim=1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```



The actor is responsible for selecting actions based on the current state of the environment. The critic evaluates the actions taken by the actor in a given state. They are both represented as a Multilayered perceptrons with three fully connected layers. The TD3 model was trained on 100 episodes and a final plot visualizes the progression of episode rewards over time steps. Unfortunately, we see that 100 episodes is not enough for this environment. It is worth mentioning that Reinforcement learning doesn't fit every task as well

The model was trained through 100 episodes, in final evaluation presenting such results:

- **Reward:** -1025.401785847083

Project 2

Soft Actor-Critic Algorithm For Mountain Car Continuous Task

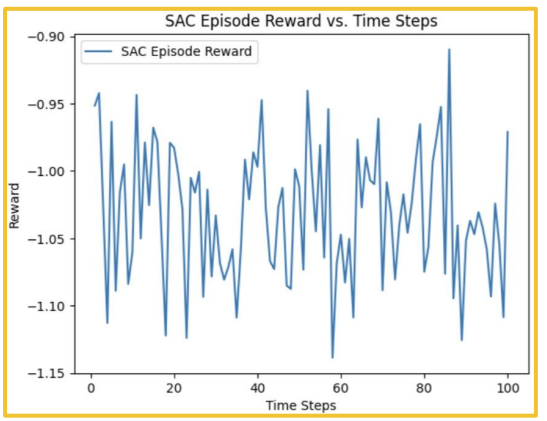
The Soft Actor-Critic (SAC) algorithm is an advanced deep reinforcement learning algorithm designed for continuous action spaces. It can learn from a replay buffer, which stores past experiences, and is not limited to using only the most recent data. In addition to optimizing for expected return, SAC maximizes the entropy of the policy, which encourages exploration. Maximizing entropy helps the agent to balance between exploration and exploitation effectively.

Step 1. Environment Preparation

MountainCarContinuous is a variation of the classic MountainCar problem. A car is situated between two hills, with the goal of the agent being to drive the car up the right hill to reach the flag at the top. The car has insufficient power to make it directly up the hill, requiring the agent to learn how to rock back and forth to build enough momentum to reach the goal. There is a continuous action space, so the agent can apply a continuous range of actions to the car, allowing for more nuanced control over the acceleration and direction of the car.

Step 2. Actor and Critic models training

```python
class SACAgent:
    def __init__(self, state_dim, action_dim, max_action_sac, alpha_lr_sac=0.0003):
        self.actor = ActorSAC(state_dim, action_dim, max_action_sac)
        self.actor_optimizer = optim.Adam(self.actor.parameters(), lr=actor_lr_sac)

        self.critic1 = CriticSAC(state_dim, action_dim)
        self.critic2 = CriticSAC(state_dim, action_dim)
        self.critic1_optimizer = optim.Adam(self.critic1.parameters(), lr=critic_lr_sac
        self.critic2_optimizer = optim.Adam(self.critic2.parameters(), lr=critic_lr_sac

        self.log_alpha = torch.zeros(1, requires_grad=True)
        self.log_alpha_optimizer = optim.Adam([self.log_alpha], lr=alpha_lr_sac)

        self.replay_buffer = []
        self.pointer = 0
        self.buffer = []

        self.max_action = max_action_sac

    def select_action(self, state):
        state = torch.FloatTensor(state.reshape(1, -1))
        mean = self.actor(state)
        action_dist = Normal(mean, torch.ones_like(mean) * 0.1)
        action = action_dist.sample()
        return action.numpy().flatten()
```



SAC uses a soft update mechanism for both the actor and the critics, which is responsible for smoothly blending the target networks towards the current networks. This should help stabilize the learning process, but even with that we can see in plot that the model hasn't been able to stabilize in span of 100 episodes of training. It may be so that like for Pendulum task this algorithm of Reinforcement Learning isn't good enough for this particular task or that there could be some improvements on a bigger number of episodes.

The model was trained through 100 episodes, in final evaluation presenting such results:

- **Reward:** -0.9710508524927299

Evolutionary Algorithms

I. IBEA Ie+-indicator algorithm

II. Binary and Real-Coded Genetic Algorithms

Project 1

IBEA Ie+-indicator algorithm

In this project I tried to replicate the IBEA Ie+-indicator algorithm. I tested my IBEA on ZDT6 and DTLZ2 problems, so as to perform the Wilcoxon rank test for comparisons of how my algorithm is working on different problems. I used the NSGA-II algorithm from the pymoo Python package to do a Wilcoxon test between my algorithm and NSGA-II on such problems as ZDT6, DTLZ2 and DTLZ6. I visualized final Pareto fronts done with my algorithm to compare them to the true Pareto fronts of the problems. I researched the difference of results between my algorithm and the one from the completed jmetalpy package as well.

Results Analysis

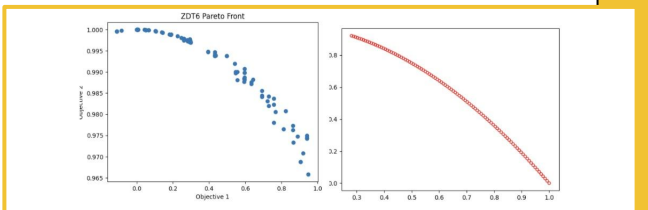


Figure 1. ZDT6: 1 — My Adaptive IBEA (Ie+) (second try); 2 — True ZDT6 Pareto front

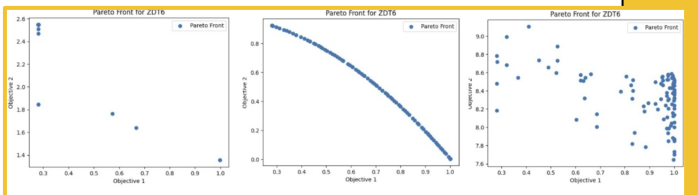


Figure 4. jmetalpy IBEA Ie+: 1 — Iterations = 25000, pc = 0.8, pm = 0.04, k = 0.05; 2 — iterations = 25000, pc = 1, pm = 1, k = 0.05; 3 — iterations = 200, pc = 1, pm = 1, k = 0.05

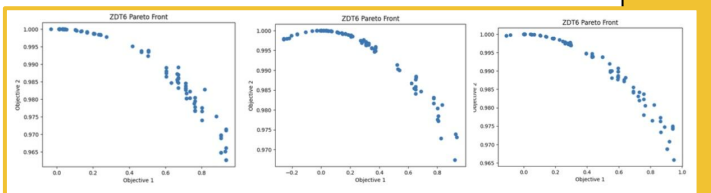


Figure 5. My IBEA Ie+: 1 — iterations = 100, k = 0.5; 2 and 3 — it = 60, k = 0.5, num_variables = 100

Table 1. Comparison of algorithms (Wilcoxon rank test of my Adaptive IBEA Ie+ and NSGA-II)

	P-value	
ZDT6 (second try IBEA)	0.017197235805579216	< 5% (significant difference)
DTLZ2 (second try IBEA)	1.0	> 5% (no significant difference)
DTLZ6 (first try IBEA)	1.0	>5% (no significant difference)

Table 2. Comparison of problems(Wilcoxon rank test of my Adaptive IBEA Ie+ on problems)

	DTLZ2 (second try IBEA)	
	P-value	
ZDT6 (second try IBEA)	0.017197235805579216	< 5% (significant difference)

As we can see in table 1, based on indicator values acquired, IBEA seems to do better on the ZDT6 problem than NSGA-II, but in DTLZ2 and 6 there is no significant difference. While performing IBEA on ZDT6 and DTLZ6 we certainly see that results from ZDT6 are much more valuable than those of DTLZ2.

jmetalpy IBEA in Figure 4 heavily relies on probability of mutation and crossover, because the best of its performances are when these variables are equal to 1, so we definitely always get crossover and mutation to the children that will be added to the new population. In 200 iterations even with all the right parameters the algorithm still can't present the true Pareto front we're searching for.

Unlike jmetalpy IBEA, my IBEA in Figure 5 relies much more on the k value, which represents the fitness scaling factor. Interestingly, but 0.5 appeared to be the best value for this parameter — if we set k bigger values, the function won't converge, with lesser levels the function starts to spread instead of narrow down. My IBEA performs best on 60 iterations. I would say this is a good result so far.

Project 2

Binary and Real-Coded Genetic Algorithms

The goal of this project was to discover the differences in implementing binary and real-coded genetic algorithms, exploring one-point, SBX- and BLX- crossovers, and studying the constraint handling using Death, Kuri's Static and Joines and Houck's Dynamic Penalties.

Step 1. BLX-alpha VS SBX-Crossover

Whether to choose the BLX-alpha or the SBX and which of them is better, depends on the specific problem being solved and the characteristics of the solution space. There is no single answer for all situations, and the choice between these two crossover operators should be based on empirical testing and the nature of the optimization problem. Problem characteristics are that BLX-alpha tends to produce offsprings that are closer to the parents, providing more exploitation. If our problem has well-defined local optima and we want to converge quickly, BLX-alpha may be a better choice. SBX also generates more diverse offsprings, promoting exploration, but our problem has multiple, widely separated optima, SBX may help discover them more effectively.

Step 2. Constraint Handling Methods' Comparison

Genotype		Phenotype		f(x)
x	y	x	y	
00100000	10000000	1.5	3.01	-4.535
10110011	10010010	3.81	3.29	-20.221
11101010	11000010	4.67	4.04	-32.363
00111101	10100001	1.96	3.53	-7.735
11010001	11001011	4.28	4.18	-27.759
10100111	10010000	3.62	3.26	-18.155
01011110	01101101	2.47	2.71	-7.672
01001001	00110011	2.15	1.8	-4.665
00000100	01011001	1.06	2.4	-2.003
11110000	10000000	4.76	3.01	-32.278

Table 1. Evaluation of the fitness function

	Death Penalty	Kuri's Static penalty	Joines and Houck's Dynamic penalty
Final result	x1: 8.227 x2: 14.088 Objective Value: 22.315	x1: 8.223 x2: 14.083 Objective Value: 22.305	x1: 8.223 x2: 14.083 Objective Value: 22.305

Table 3. Comparing penalties' results

Somehow, Kuri's static penalty and Joines and Houck's dynamic penalty were able to find the exact same answer, which puts them as equals in this particular exploration.

Death penalty was also really close, giving slightly bigger numbers than the other two methods.

Based on that, it is very easy to say that all three techniques are of the same calibre, giving similar enough results, but personally I give my preference to the death penalty technique, because it's very simple to understand and to code at the same time. Seeing that its results are on the same page as the more complex methods', I would choose the death penalty as the initial constraint technique.

Computer Vision

1. Haar Cascade Classification for Face Detection

Project 1

Haar Cascade Classification for Face Detection

Haar Cascade Classification is a machine learning-based object detection method widely used for detecting objects in images or video streams, particularly efficient for real-time applications due to their quick processing speed and reliable accuracy. This method has been extended to detect objects beyond faces, such as vehicles, pedestrians, and specific items. Its versatility, speed, and accuracy make it a popular choice when real-time object detection is crucial, such as in robotics, augmented reality, and autonomous vehicles.

Step 1. Data Preparation

Haar Cascade classifiers for face and eye detection are loaded using the CascadeClassifier class from OpenCV package. These pre-trained classifiers, stored as XML files, are essential for identifying patterns in images that correspond to faces and eyes. Then the source image is loaded.

Step 2. Face and Eyes Detection

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for(x, y, w, h) in faces:
   cv2.rectangle(image, (x,y), (x+w, y+h), (255,0,0), 2)
   roi_gray = gray[y:y+h, x:x+w]
   roi_color= image[y:y+h, x:x+w]
   eyes = eye_cascade.detectMultiScale(roi_gray)
   for(ex, ey, ew, eh)in eyes:
     cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh), (0, 255, 0), 2)
cv2_imshow(image)
cv2.waitKey(0)
```

The detectMultiScale function utilizes the Haar Cascade classifier to identify potential faces in the grayscale image. Detected faces are represented as rectangles, drawn onto the original image using cv2.rectangle. For each detected face, a region of interest (ROI) is extracted from the grayscale and color images. For each detected face, method proceeds to perform eye detection using a similar process. The region of interest within the detected face is processed by the detectMultiScale function for eyes.

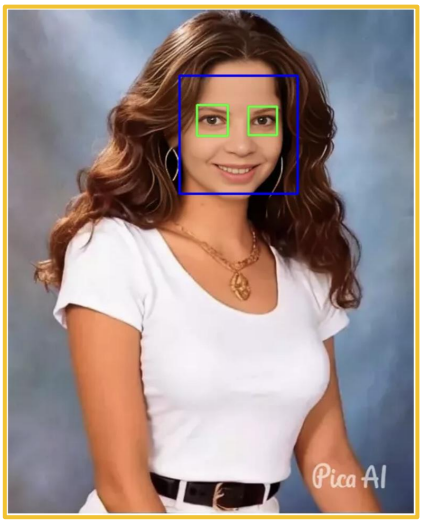


Image I. Successful Face and Eyes Detection

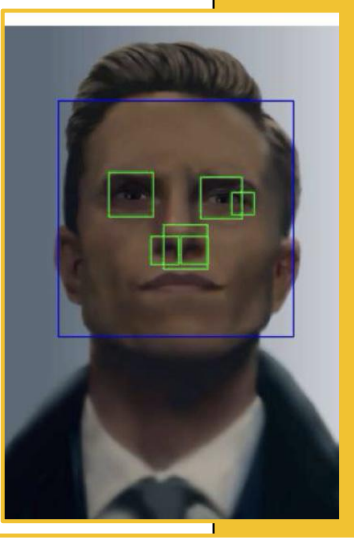


Image II. Unsuccessful Face and Eyes Detection

Though, Haar Cascade Classifier is a well performing algorithm (Image I), it can result in inaccurate results (Image II) or not release result at all, not being able to recognize the face.

Thank you for exploring this Portfolio

(You can find all implementations in my GitHub Profile)

My GitHub Profile

Best Regards,
Oleksandra Ivanova

Contact me:
aleksandra4115@gmail.com