DATA 6250
Machine Learning for Data Science

Using Machine Learning to Predict Resource and Performance Metrics in AI Model Configurations

Leveraging regression techniques to forecast hardware usage, latency, and other system-level outcomes.

Done By: Rohan Pratap Reddy Ravula

14 April, 2024

Introduction

This project aims to develop machine learning regression models to predict key deployment and performance metrics of state-of-the-art AI models. By learning from existing models' architectural, hardware, and training features, we estimate outcomes like latency, memory usage, and energy consumption

Why This Matters:

With the rapid proliferation of AI systems, understanding model efficiency and deployment costs is vital. This project supports:

- Optimized model deployment
- Informed hardware-resource decisions
- Energy and inference trade-off analysis

Background Project

Current scenario:

Explosive Growth of AI Models

Over the last few years, we've seen rapid scaling of AI models in both size and complexity. Models such as GPT-4, Gemini, and Claude operate with hundreds of billions of parameters, demanding massive computational resources.

Challenges in AI Deployment

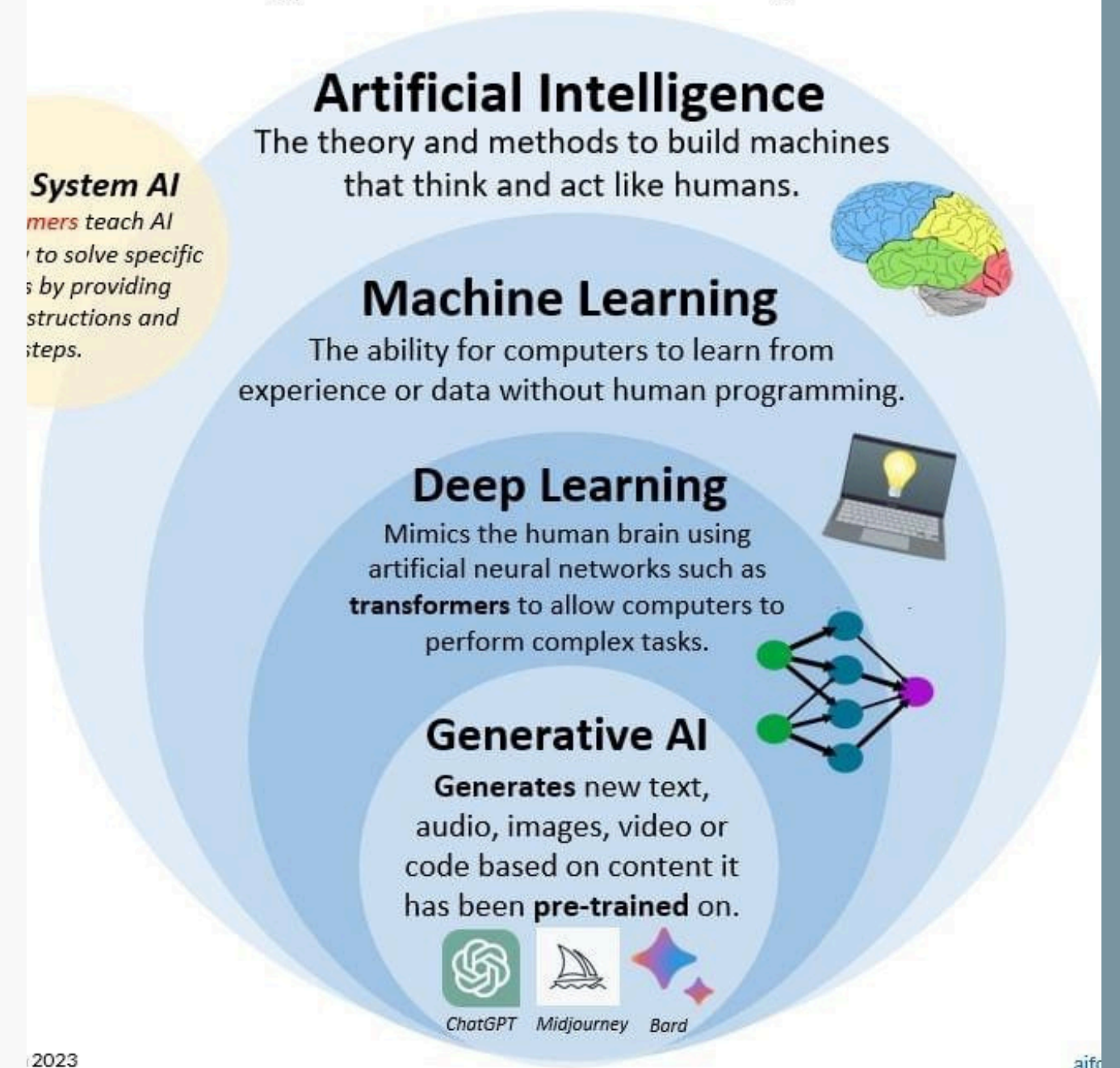With increased scale comes a growing concern for:

- Inference latency
- Memory and energy efficiency
- Hardware compatibility and cost trade-offs

Organizations now face critical decisions in choosing the right model–hardware combination that balances performance and resource usage.

Defining Generative AI

understand generative artificial intelligence (GenAI), we first need to understand how the technology builds from each of the AI subcategories listed below.

System AI
mers teach AI
to solve specific
s by providing
structions and
steps.

Artificial Intelligence
The theory and methods to build machines that think and act like humans.

Machine Learning
The ability for computers to learn from experience or data without human programming.

Deep Learning
Mimics the human brain using artificial neural networks such as **transformers** to allow computers to perform complex tasks.

Generative AI
Generates new text, audio, images, video or code based on content it has been **pre-trained** on.

ChatGPT   Midjourney   Bard

2023

Background Project

Current scenario:

Why Predictive Modeling Matters
By analyzing real-world data from the Epoch AI repository, this project addresses:
- How model features (e.g., training FLOPs, architecture) relate to deployment costs
- Which configurations yield better efficiency

How to proactively predict deployment behavior for unseen models
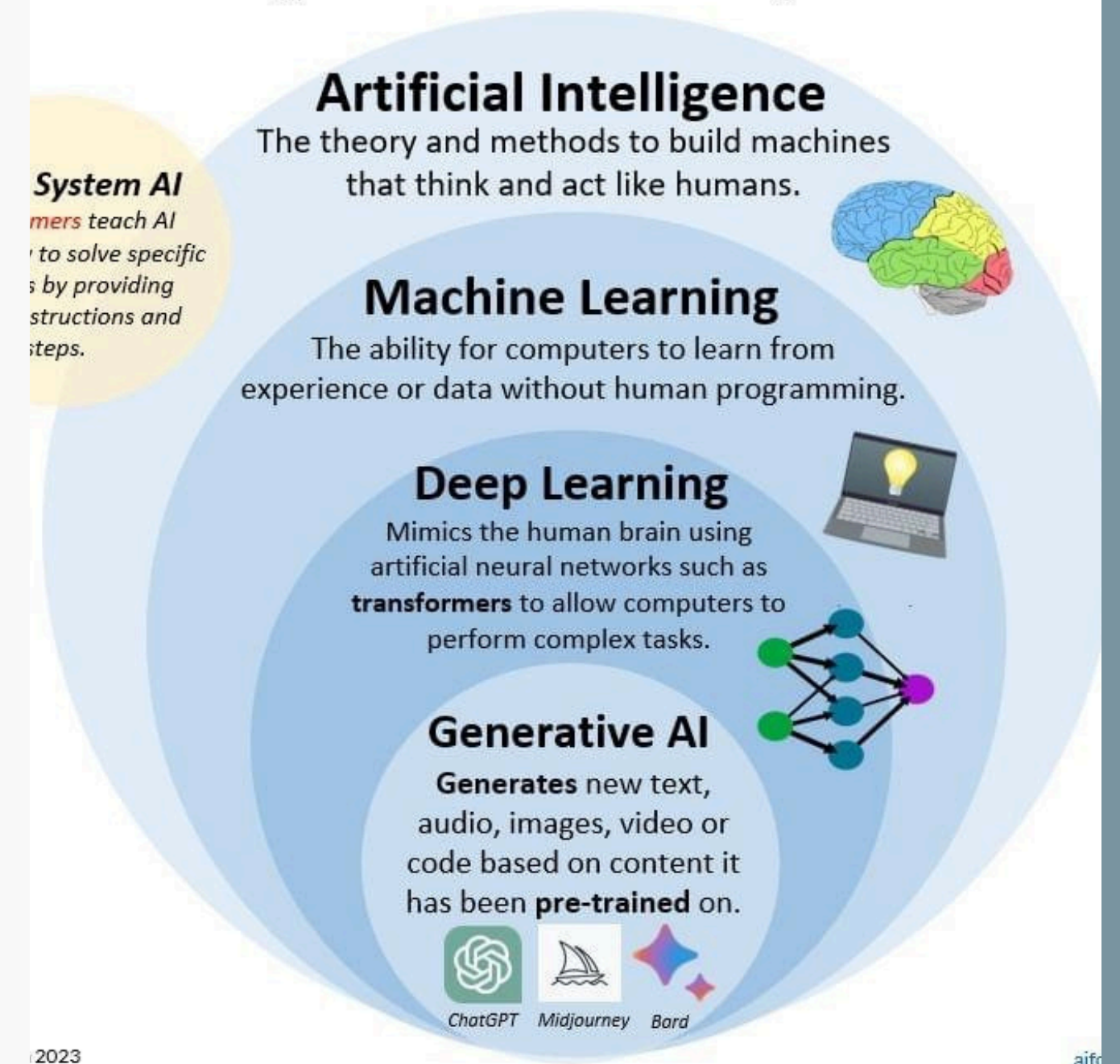
Data Landscape
This analysis draws from:
- Notable AI Models – Architectures, datasets, FLOPs
- Large-scale Training Stats – Training duration, compute usage
- ML Hardware – Devices used for deployment and training

Defining Generative AI

understand generative artificial intelligence (GenAI), we first need to understand how the technology builds from each of the AI subcategories listed below.

System AI
ers teach AI
to solve specific
s by providing
structions and
steps.

Artificial Intelligence
The theory and methods to build machines that think and act like humans.

Machine Learning
The ability for computers to learn from experience or data without human programming.

Deep Learning
Mimics the human brain using artificial neural networks such as **transformers** to allow computers to perform complex tasks.

Generative AI
Generates new text, audio, images, video or code based on content it has been **pre-trained** on.

ChatGPT   Midjourney   Bard

2023

Datasets Overview

Data Source: Epoch AI

The datasets used in this project are obtained from <u>Epoch AI</u>, a research initiative focused on documenting and analyzing trends in machine learning systems. Their curated databases enable scalable and transparent analysis of AI model development and deployment.

Links to Datasets:

- https://epoch.ai/data/notable_ai_models.csv
- Contains detailed metadata for 900+ widely recognized AI models (e.g., GPT-4, Gemini, Claude).
  - https://epoch.ai/data/large_scale_ai_models.csv
- Offers training specs, compute data, and publication details for large foundational models.
  - https://epoch.ai/data/ml_hardware.csv
- Provides hardware specifications used for model training and inference (e.g., A100, H100, TPU v4).

📋 **Dataset Summary:**

Dataset	Rows	Columns	Highlights
Notable AI Models	914	33	Model size, compute, training time, dataset size, power usage
Large-Scale AI Models	284	28	Task domain, model org, compute FLOPs, training infra
ML Hardware	159	29	FLOP/s, memory, bandwidth, TDP, release date, transistors

PreProcessing: Normalization

1. Redundant Data Cleaning

- Removed unnecessary columns like URLs, references, and notes to streamline the datasets and reduce noise during modeling.

2. Categorical Explosion and Normalization

Many features (e.g., Model, Organization, Training hardware, Country, ML models) contained multiple comma-separated values. These were:

- Split into individual values
- Exploded into separate rows, normalizing one-hot associations
- Country names standardized (e.g., UK, USA, South Korea)

3. Duplicate Removal

- After normalization, duplicate rows were removed to ensure data integrity — e.g., reducing from ~59,000 to 5,700 unique rows in the notable_ai_models dataset.

PreProcessing

Missing Value Handling

- Key hardware metrics like TDP (W), transistors, and Process size (nm) had missing values.
- Two imputation strategies were applied:
- RMS-based imputation for rough estimates.
- Machine Learning–based imputation using:
- RandomForestRegressor + IterativeImputer for numeric fill-ins.
- ExtraTreesRegressor for estimating Release price (USD) and secondary performance metrics.

Filling Missing Value Using RAG search and sentence transformers

- From the data column "Notes" we will use RAG based LLM Model:

"Dragon-qwen-7b-ov" to search for missing values related to "model",

for sentence - transformers we use "all-mpnet-base-v2" to fill based on cosine similarity from the models columns.

And also we use filling strategy by using other datasets "ml_hardware", "notable_ai_models" data as well to fill in

"large_scale_ai_models

```python
features = ['Parameters','data size','Training time (hours)',
            'Training compute (FLOP)', 'Finetune compute (FLOP)']
for feature in tqdm(features, desc="Processing features"):
    # Trying to get values by RAG
    mask = df_new[df_new[feature].isna()].copy()
    unmask = df_new[~df_new[feature].isna()].copy()

    mask[feature] = mask.progress_apply(lambda row: extract_feature_from_notes(row,feature),axis=1)

    df_new.loc[mask.index,feature] = mask[feature]
    df_new[feature] = pd.to_numeric(df_new[feature],errors='coerce')
    # Trying to get values by nearest known values  with confidence of 0.5
    mask = df_new[df_new[feature].isna()].copy()
    unmask = df_new[~df_new[feature].isna()].copy()

    mask[feature] = mask.progress_apply(lambda row: assign_values_to_features(row,unmask,feature),axis=1)

    df_new.loc[mask.index,feature] = mask[feature]
    df_new[feature] = pd.to_numeric(df_new[feature],errors='coerce')
    # Trying to get values by rms filling of known values
    mask = df_new[df_new[feature].isna()].copy()
    unmask = df_new[~df_new[feature].isna()].copy()

    mask[feature].fillna(np.sqrt((unmask[feature] ** 2).mean()),inplace=True)

    df_new.loc[mask.index,feature] = mask[feature]
print('Given Features are filled')
```

```python
for idx in tqdm(df.index, desc="Similarity matching on Base Model"):
    # Get the embeddings for the current model in 'df'
    model_emb = df_model_emb[idx]

    # Calculate cosine similarity with all base models in 'df_new'
    sim_scores = util.cos_sim(model_emb, df_new_base_model_emb).cpu().numpy().flatten()

    # Find the best match (highest similarity score)
    best_match_idx = np.argmax(sim_scores)

    # Check if the similarity score is above the threshold (0.7)
    if sim_scores[best_match_idx] >= 0.7:
        # Get the matching row from 'df_new'
        best_match_row = df_new.iloc[best_match_idx]

        # Fill NaN values in 'df' using the matching row
        for col in merge_cols:
            if pd.isna(df.at[idx, col]):
                df.at[idx, col] = best_match_row[col]
```

ML Processing

Models

- ◆ Linear Regression (baseline)
- ◆ Ridge & Lasso Regression
- ◆ Support Vector Regressor (SVR)
- ◆ Decision Tree Regressor
- ◆ Random Forest Regressor
- ◆ Gradient Boosting Regressor

Experiments:

- Original : Applied Standard Scaler and applied Algorithms
- Experiment 1: Applied Minmax Scaler and applied Algorithms'
- Experiment 2: Encoded categorical variables sentence tranformer "the npler/gte-small" and applied clustering hdbscan. and then applied another sentence transformer on text column "all-Mini-LM-v6" to get feature and then applied umap reduction to reduce features
- Experiment 3: Applied pcf on original data to reduce to 15 columns
- Experiment 4: Applied one categorical and continuous noise.

```python
def generate_feature_encoding(data,feature,comp=100,n_cluster=50):
    data[feature] = data[feature].astype(str)
    embeddings = model.encode(data[feature].tolist(),
                                  convert_to_tensor=True,
                                  show_progress_bar=True)
    umap_model = UMAP(n_components=comp,
                      min_dist=0.0,
                      metric='cosine',
                      random_state=42)
    red_embeddings = umap_model.fit_transform(embeddings)
    cluster = HDBSCAN(min_cluster_size=n_cluster,
                      metric='euclidean',
                      cluster_selection_method='eom')
    cluster.fit(red_embeddings)
    data[f'{feature}_encoded'] = cluster.labels_
    return data
```

```python
def gen_features(data,feature,n_features=30):
    data[feature] = data[feature].astype(str)
    embeddings = sentence_model.encode(data[feature].tolist(),
                                  convert_to_tensor=True,
                                  show_progress_bar=True)
    umap_model = UMAP(n_components=n_features,
                      min_dist=0.0,
                      metric='cosine',
                      random_state=42)
    red_embeddings = umap_model.fit_transform(embeddings)
    for i in range(n_features):
        data[f'feature_{feature} {i}'] = red_embeddings[:,i]
    return data
```

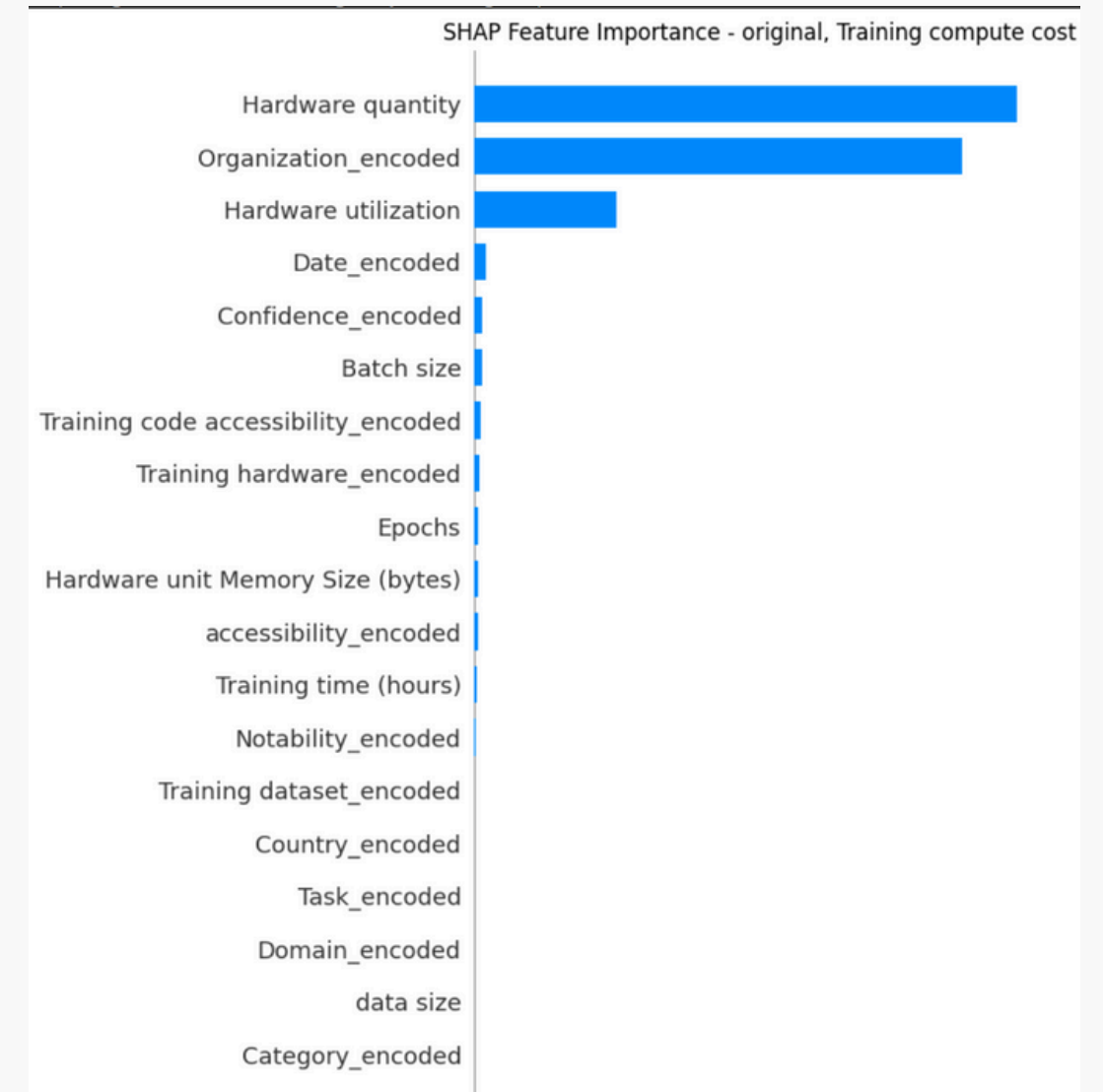# *Interpretability*

Goal

- To understand which features most influence the prediction of Training Compute Cost in AI models.

Method Used:

- ✅ SHAP (SHapley Additive exPlanations)
- A model-agnostic method that explains the contribution of each feature to the model's output.

Lesser Influential Features:

- Temporal factors like Date_encoded, Training time (hours)
- Metadata and descriptive encodings (e.g., Notability, Category, Task)
- Dataset size and accessibility details had minimal impact on cost prediction.
- Inference:
- ➡ The model learns that infrastructure and resource allocation dominate training cost prediction, far more than metadata or content size. This validates the importance of quantitative hardware features in modeling operational efficiency.

SHAP Feature Importance - original, Training compute cost

Hardware quantity
Organization_encoded
Hardware utilization
Date_encoded
Confidence_encoded
Batch size
Training code accessibility_encoded
Training hardware_encoded
Epochs
Hardware unit Memory Size (bytes)
accessibility_encoded
Training time (hours)
Notability_encoded
Training dataset_encoded
Country_encoded
Task_encoded
Domain_encoded
data size
Category_encoded

Time and Memory Usage

Goal:

Evaluate model performance not just in terms of accuracy but also training/inference efficiency and memory footprint.

Memory Usage (MB):

All reported as 0.0 MB, likely due to measurement limitations in the notebook environment.

Suggestion: Future runs can integrate precise memory profilers like memory_profiler for accurate tracking.

Key Takeaways:

- Random Forest has the fastest training time, but higher inference time.
- Gradient Boosting models offer faster inference but show variance in training time based on experiment complexity.
- Overall, models are lightweight in terms of runtime, making them suitable for deployment.

Experiment	Model	Training Time	Inference Time
original	Random Forest	0.1162	0.0040
experiment-1	Gradient Boosting	0.2501	0.0019
experiment-2	Gradient Boosting	1.9353	0.0024
experiment-3	Gradient Boosting	0.2863	0.0017
experiment-4	Gradient Boosting	0.9618	0.0026

Final Results

Key Takeaways:
- Gradient Boosting proved to be the most consistent and reliable model across all experimental setups, despite Random Forest scoring highest in one instance.
- Model robustness and adaptability across varied preprocessing steps are more important than isolated peak accuracy.
- Tools like SHAP and LIME were instrumental in interpreting model behavior and enhancing trust in the predictions.

Experiment	Best Model	R-squared Score
original	Random Forest	0.9482
experiment-1	Gradient Boosting	0.9258
experiment-2	Gradient Boosting	0.9226
experiment-3	Gradient Boosting	0.8858
experiment-4	Gradient Boosting	0.9477

Future Scope:

Opportunities for Further Improvement:

- 🔁 Model Ensembling: Combine strengths of Random Forest + Gradient Boosting for performance synergy.
- 🧪 Automated Tuning: Use Optuna or Bayesian optimization to fine-tune model hyperparameters.
- 📉 Dimensionality Reduction: Apply SHAP-driven feature selection to reduce model complexity and overfitting.
- 🔄 Robust Validation: Use multi-split cross-validation to ensure model generalizability.

Overall Learning Outcome:

This project deepened practical skills in:

- Machine learning workflows
- Model interpretability
- Hyperparameter tuning
- Trade-offs between accuracy, efficiency, and transparency