bsub_notebook3-report

May 14, 2025

1 Test a naive linear regression model on the dataset

[12]:
```python
import polars as pl
import os
import json
from loguru import logger
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
```

Load the dataset (very large!)

[2]:
```python
output_dir = '/lustre/scratch123/hgi/mdt2/teams/hgi/eh19/work-data/bsub-memory/'
combined_embeddings_file = os.path.join(output_dir, 'combined_embeddings.
↪parquet')
final_dataset_file = os.path.join(output_dir, 'final_dataset.parquet')

# Load the original commands
logger.info(f"Loading commands from {os.path.join(output_dir,␣
↪'filtered_under_5GB.jsonl')}")
df = pl.scan_ndjson(os.path.join(output_dir, 'filtered_under_5GB.jsonl'))
df_collected = df.collect()

logger.info(f"Loading indices from {os.path.join(output_dir,␣
↪'df_without_downsampling.json')}")
with open(os.path.join(output_dir, 'df_without_downsampling.json'), 'r') as f:
    indices = json.load(f)

# Filter commands using indices
logger.info("Filtering commands using indices")
df_commands = df_collected.filter(pl.arange(0, df_collected.height).
↪is_in(indices))
logger.info(f"Filtered to {df_commands.height} commands")

# Load embeddings
logger.info(f"Loading embeddings from {combined_embeddings_file}")
```

```
df_embeddings = pl.read_parquet(combined_embeddings_file)
logger.info(f"Loaded {df_embeddings.height} embeddings with {df_embeddings.
 ↪width} dimensions")

# Verify dimensions match
if df_commands.height != df_embeddings.height:
    logger.error(f"Number of commands ({df_commands.height}) does not match␣
 ↪number of embeddings ({df_embeddings.height})")
```

```
2025-05-14 10:40:27.036 | INFO     |
__main__:<module>:6 - Loading commands from

/lustre/scratch123/hgi/mdt2/teams/hgi/eh19/work-data/bsub-

memory/filtered_under_5GB.jsonl
2025-05-14 10:43:21.984 | INFO     |
__main__:<module>:10 - Loading indices from

/lustre/scratch123/hgi/mdt2/teams/hgi/eh19/work-data/bsub-

memory/df_without_downsampling.json
2025-05-14 10:43:22.423 | INFO     |
__main__:<module>:15 - Filtering commands using

indices
2025-05-14 10:43:22.991 | INFO     |
__main__:<module>:17 - Filtered to 1949712

commands
2025-05-14 10:43:22.993 | INFO     |
__main__:<module>:20 - Loading embeddings from

/lustre/scratch123/hgi/mdt2/teams/hgi/eh19/work-data/bsub-

memory/combined_embeddings.parquet
2025-05-14 10:46:04.693 | INFO     |
__main__:<module>:22 - Loaded 1949712 embeddings

with 768 dimensions
```

[4]:
```
# Check memory usage of df_commands
memory_usage_bytes = df_commands.estimated_size()
memory_usage_mb = memory_usage_bytes / (1024 * 1024)
logger.info(f"Memory usage of df_commands: {memory_usage_mb:.2f} MB")

memory_usage_bytes = df_embeddings.estimated_size()
memory_usage_mb = memory_usage_bytes / (1024 * 1024)
logger.info(f"Memory usage of df_embeddings: {memory_usage_mb:.2f} MB")
```

```
2025-05-14 10:48:16.208 | INFO     |
```

```
__main__:<module>:4 - Memory usage of

df_commands: 4777.85 MB
2025-05-14 10:48:16.214 | INFO     |
__main__:<module>:8 - Memory usage of

df_embeddings: 5712.05 MB
```

1.0.1 First let's generate a downsampled training set using the log binning method

[5]:
```python
df_filtered = df_commands.with_columns(
    pl.col("MAX_MEM_USAGE_MB").log10().alias("log_max_usage")
).with_row_index("row_id")

# Create bins and sample
logger.info("Creating bins and sampling data")
# Create bins using numpy
bins = np.linspace(
    df_filtered["log_max_usage"].min(),
    df_filtered["log_max_usage"].max(),
    101  # 100 bins means 101 edges
)

# Add bin labels
df_filtered = df_filtered.with_columns(
    pl.col("log_max_usage").cut(bins, labels=[f"bin_{i}" for i in range(102)]).
  ↪alias("bin")
)

# Group by bin and sample
df_train = (
    df_filtered
        .group_by("bin")
        .map_groups(lambda x: x.sample(min(len(x), 1000), seed=42))
        .drop(["bin"])
)

# Get the row IDs that are in df_train
train_row_ids = df_train.select("row_id").to_series()

# Create df_test with rows that are not in df_train
df_test = df_filtered.filter(~pl.col("row_id").is_in(train_row_ids))

# Verify the split
print(f"Original df size: {df_filtered.height}")
print(f"Training set size: {df_train.height}")
print(f"Test set size: {df_test.height}")
print(f"Sum of splits: {df_train.height + df_test.height}")
```

```
2025-05-14 10:50:29.593 | INFO     |
__main__:<module>:6 - Creating bins and sampling
data
```

```
Original df size: 1949712
Training set size: 77649
Test set size: 1872063
Sum of splits: 1949712
```

```
/tmp/ipykernel_1770326/3998462455.py:21: CategoricalRemappingWarning: Local
categoricals have different encodings, expensive re-encoding is done to perform
this merge operation. Consider using a StringCache or an Enum type if the
categories are known in advance
  df_filtered
/tmp/ipykernel_1770326/3998462455.py:31: DeprecationWarning: `is_in` with a
collection of the same datatype is ambiguous and deprecated.
Please use `implode` to return to previous behavior.

See https://github.com/pola-rs/polars/issues/22149 for more information.
  df_test = df_filtered.filter(~pl.col("row_id").is_in(train_row_ids))
```

1.0.2 Then plot a histogram to show the memory use distribution in the training set...

[6]:
```python
import plotly.express as px

def plot_histogram(df, title="Distribution of Max Memory Usage in Training␣
↪Set"):
    # Get log values of memory usage
    log_values = df["MAX_MEM_USAGE_MB"].log10().to_numpy()

    # Calculate histogram bins and counts using NumPy
    min_log = log_values.min()
    max_log = log_values.max()
    bins = np.linspace(min_log, max_log, 101)  # 100 bins needs 101 edges
    counts, bin_edges = np.histogram(log_values, bins=bins)
    bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])

    # Create bar plot with pre-calculated data
    fig = px.bar(
        x=bin_centers,
        y=counts,
        title=title
    )

    # Create tick values that span the log range
    log_ticks = np.linspace(min_log, max_log, 8)
```

```
    # Convert log values back to actual memory values
    actual_memory_mb = 10 ** log_ticks

    # Format tick labels to show MB or GB as appropriate
    tick_labels = []
    for val in actual_memory_mb:
        if val < 1000:
            tick_labels.append(f"{val:.1f} MB")
        else:
            tick_labels.append(f"{val/1000:.1f} GB")

    # Update x-axis to show actual memory values at tick marks
    fig.update_xaxes(
        title="Memory Usage",
        tickvals=log_ticks,
        ticktext=tick_labels
    )

    # Update other layout properties
    fig.update_layout(
        yaxis_title="Count",
        bargap=0.1
    )

    # Display the plot
    fig.show()
plot_histogram(df_train, "Distribution of Max Memory Usage in Training Set")
```

1.0.3 And the test set...

[7]:
```
plot_histogram(df_test, "Distribution of Max Memory Usage in Test Set")
```

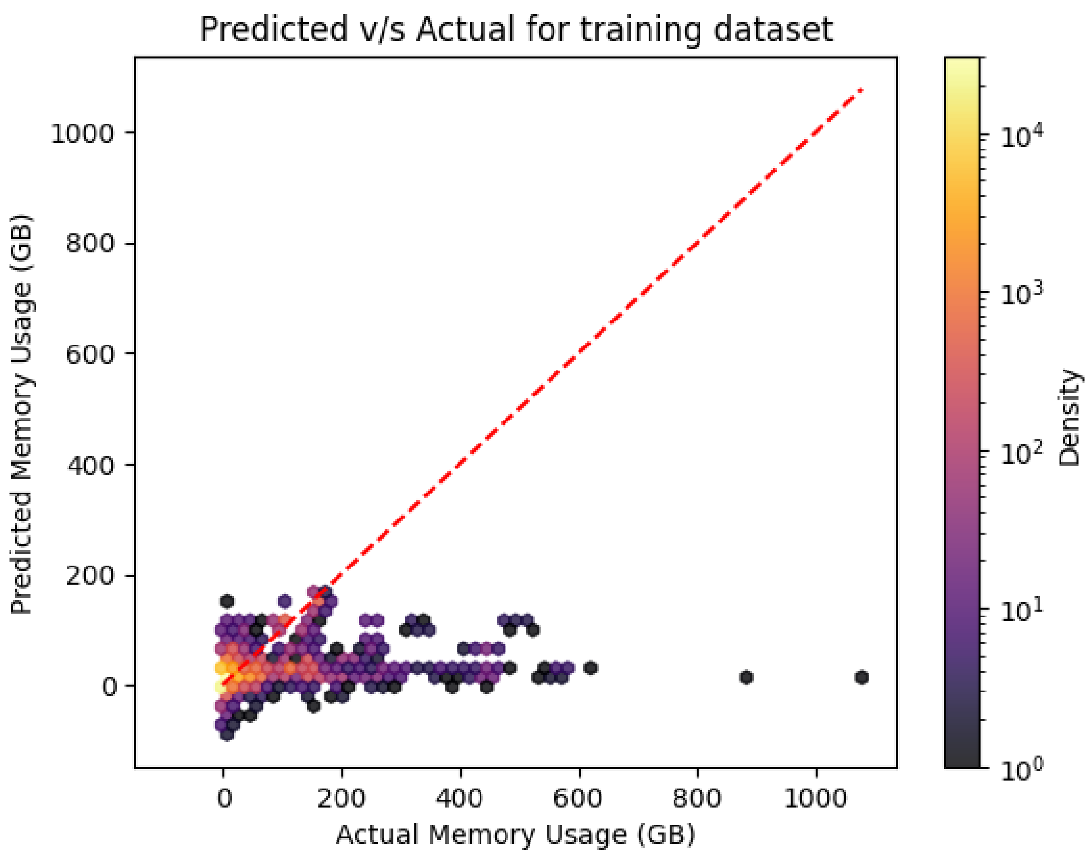
So we can see the downsampling flatten the distribution significantly

1.0.4 Run a linear regression with a random downsampling

[14]:
```
def plot_pred_vs_act(y_pred, y, title="Predicted v/s Actual for training␣
↪dataset"):
    y_actual = y / 1024 if max(y) > 100 else y
    y_predict = y_pred / 1024 if max(y_pred) > 100 else y_pred

    x_extent = [min(y_actual.min(), y_predict.min()), max(y_actual.max(),␣
↪y_predict.max())]
    y_extent = [min(y_actual.min(), y_predict.min()), max(y_actual.max(),␣
↪y_predict.max())]
    fig = plt.figure()
```

```python
    plt.hexbin(y_actual, y_predict, gridsize=60, cmap='inferno', alpha=0.8,␣
↪mincnt=1, bins='log',extent=(x_extent + y_extent))
    plt.colorbar(label='Density')
    plt.xlabel("Actual Memory Usage (GB)")
    plt.ylabel("Predicted Memory Usage (GB)")
    plt.plot([min(y_actual), max(y_actual)], [min(y_actual), max(y_actual)],␣
↪'r--', label='Ideal')
    plt.title(title)
    return fig

def run_linear_regression(df_train, df_embeddings):
    X = (df_embeddings.filter(pl.arange(0, df_embeddings.height)
                              .is_in(df_train["row_id"].implode()))
                    .to_numpy()
    )
    y = df_train["MAX_MEM_USAGE_MB"].to_numpy()

    # Extract the target variable y (use log scale for better modeling)
    model = LinearRegression()
    model.fit(X, y)

    y_pred = model.predict(X)
    r2 = r2_score(y, y_pred)
    logger.info(f"R2 score: {r2}")

    fig = plot_pred_vs_act(y_pred, y)
    fig.show()
    return model

model = run_linear_regression(df_train, df_embeddings)
```

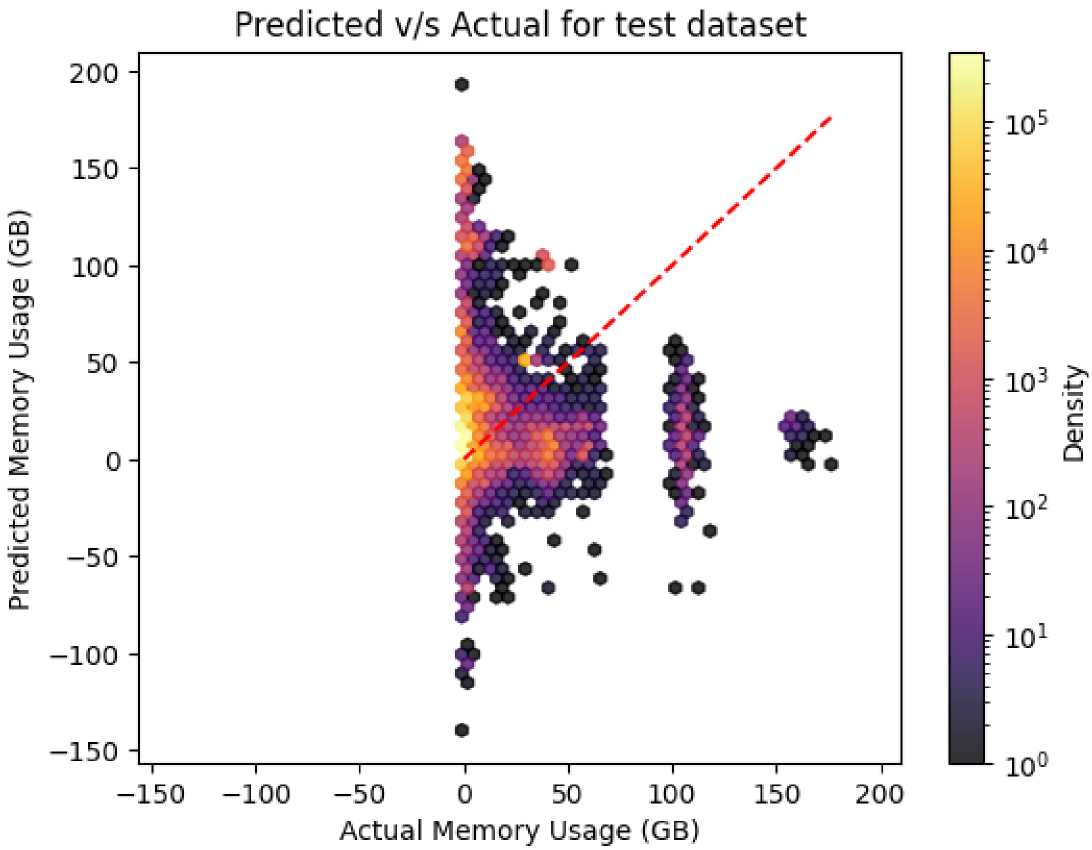
```
2025-05-14 11:01:51.914 | INFO     |
__main__:run_linear_regression:29 - R2 score:
0.24884217977523804
```



The r2 is very low, suggesting a bad fit. Let's look at it for the test set

[15]:
```
X_test = (df_embeddings.filter(pl.arange(0, df_embeddings.height)
                            .is_in(df_test["row_id"].implode()))
                    .to_numpy()
    )
y_test = df_test["MAX_MEM_USAGE_MB"].to_numpy()
y_pred = model.predict(X_test)
logger.info(f"R2 score: {r2_score(y_test, y_pred)}")
fig = plot_pred_vs_act(y_pred, y_test, title="Predicted v/s Actual for test␣
 ↪dataset")
fig.show()
```

```
2025-05-14 11:02:15.242 | INFO     |
__main__:<module>:7 - R2 score:
-7.353732109069824
```



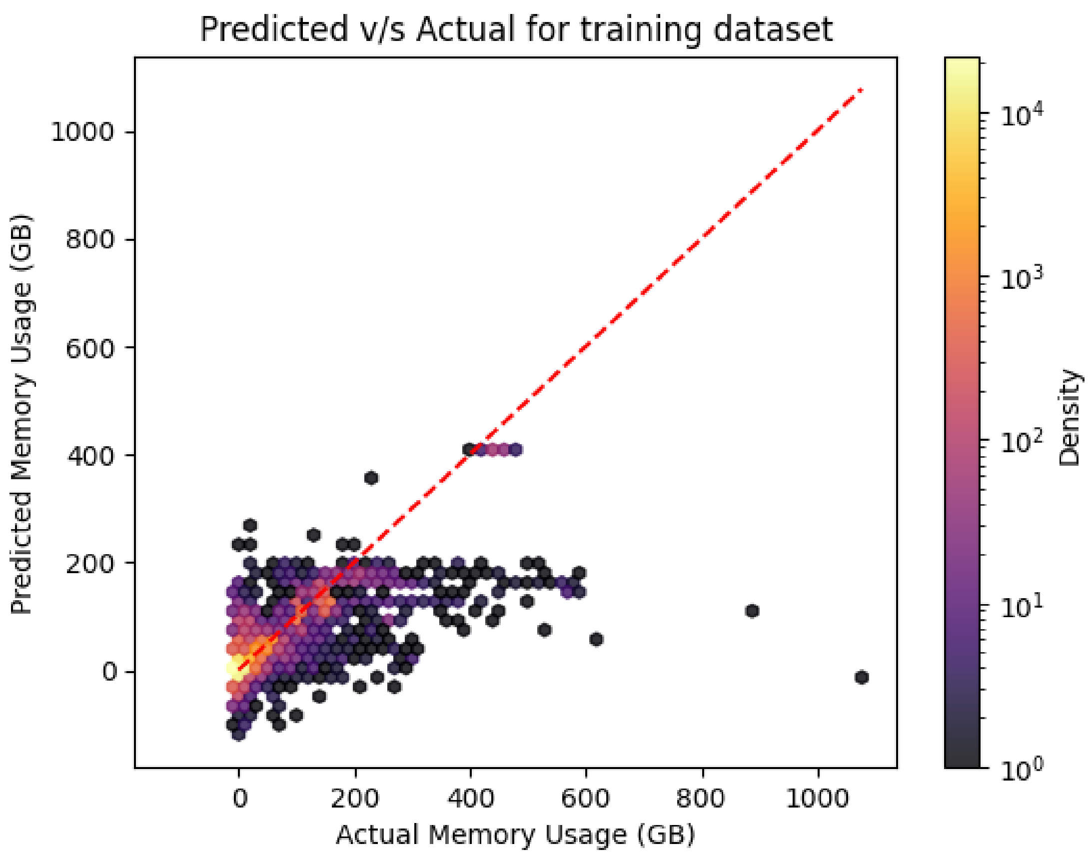
The fit is even worse for the test dataset

If we use the exact records that were used to fit the model from previous weeks…

[17]:
```
## Read the downsampled rows
df = pl.scan_ndjson(os.path.join(output_dir, 'df_with_downsampling.json'))
df_collected = df.collect()
df_train = df_commands.with_row_index("row_id").filter(pl.col("_id").
 ↪is_in(df_collected["_id"].implode()))
df_test = df_commands.with_row_index("row_id").filter(~pl.col("_id").
 ↪is_in(df_collected["_id"].implode()))

model = run_linear_regression(df_train, df_embeddings)
```

```
2025-05-14 11:04:08.620 | INFO     |
__main__:run_linear_regression:29 - R2 score:
```

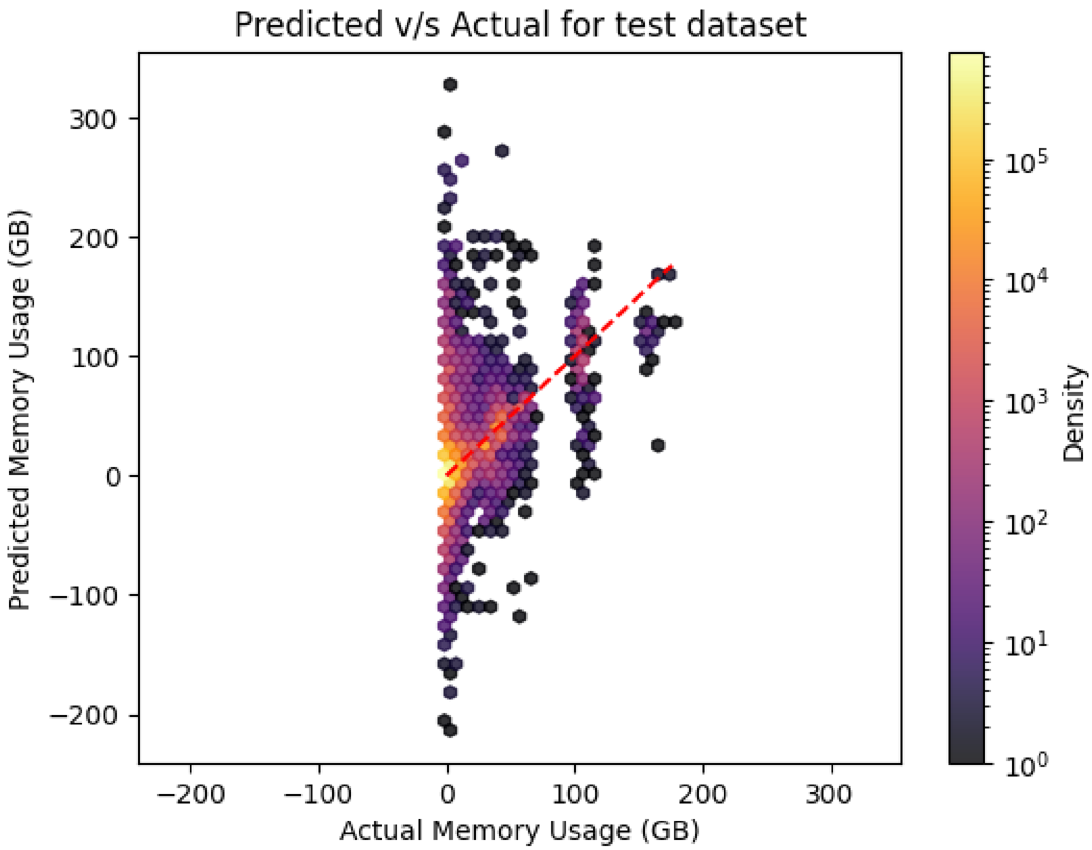
```
0.7481296062469482
```



[20]:
```
X_test = (df_embeddings.filter(pl.arange(0, df_embeddings.height)
                              .is_in(df_test["row_id"].implode()))
                    .to_numpy()
    )
y_test = df_test["MAX_MEM_USAGE_MB"].to_numpy()
y_pred = model.predict(X_test)
logger.info(f"R2 score: {r2_score(y_test, y_pred)}")
fig = plot_pred_vs_act(y_pred, y_test, title="Predicted v/s Actual for test␣
↪dataset")
fig.show()
```

```
2025-05-14 11:06:27.400 | INFO     |
__main__:<module>:7 - R2 score:
```

-1.2371060848236084



It is better than previous sampling, but still bad at predicting for the test set.

I have also conducted the sampling hundreds of times but the average R2 I got was around 0.2. I am not sure why this sampling gets a much better result. However the linear regression in this naive form is still a bad model.

From the prediction it seems the model has predicted quite a lot of negative memory usage. Using log transformation would by-pass this.

2 In conclusion:

- I recommend trying the log transformed values for training instead
- and next steps to try more complex models