Reading the data

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt

base_dir = "/kaggle/input/jane-street-real-time-market-data-forecasting/train.parquet"
df = pd.read_parquet(f'{base_dir}/partition_id=0/part-0.parquet')
```

Exploring the data

Dataset Info

- **79 features and 9 responders**
  - *Anonymized but representing real market data. The training set contains historical data and returns.*
- **date_id** and **time_id**
  - *Integer values that are ordinally sorted, providing a chronological structure to the data, although the actual time intervals between time_id values may vary.*
- **symbol_id**
  - *Identifies a unique financial instrument.*
- **weight**
  - *The weighting used for calculating the scoring function.*
- **feature_{00...78}**
  - *Anonymized market data.*
- **responder_{0...8}**
  - *Anonymized responders clipped between -5 and 5. The `responder_6` field is what we are trying to predict.*

Goal

- Predict `responder_6`.

responder_6

In [2]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1944210 entries, 0 to 1944209
Data columns (total 92 columns):
 #   Column       Dtype
---  ------       -----
 0   date_id      int16
 1   time_id      int16
 2   symbol_id    int8
 3   weight       float32
 4   feature_00   float32
 5   feature_01   float32
 6   feature_02   float32
 7   feature_03   float32
 8   feature_04   float32
 9   feature_05   float32
 10  feature_06   float32
 11  feature_07   float32
 12  feature_08   float32
 13  feature_09   int8
 14  feature_10   int8
 15  feature_11   int16
 16  feature_12   float32
 17  feature_13   float32
 18  feature_14   float32
 19  feature_15   float32
 20  feature_16   float32
 21  feature_17   float32
 22  feature_18   float32
 23  feature_19   float32
 24  feature_20   float32
 25  feature_21   float32
 26  feature_22   float32
 27  feature_23   float32
 28  feature_24   float32
 29  feature_25   float32
 30  feature_26   float32
 31  feature_27   float32
 32  feature_28   float32
 33  feature_29   float32
 34  feature_30   float32
 35  feature_31   float32
 36  feature_32   float32
 37  feature_33   float32
 38  feature_34   float32
 39  feature_35   float32
 40  feature_36   float32
 41  feature_37   float32
 42  feature_38   float32
 43  feature_39   float32
 44  feature_40   float32
 45  feature_41   float32
 46  feature_42   float32
 47  feature_43   float32
 48  feature_44   float32
 49  feature_45   float32
 50  feature_46   float32
 51  feature_47   float32
 52  feature_48   float32
 53  feature_49   float32
 54  feature_50   float32
 55  feature_51   float32
 56  feature_52   float32
 57  feature_53   float32
 58  feature_54   float32
 59  feature_55   float32
 60  feature_56   float32
 61  feature_57   float32
 62  feature_58   float32
 63  feature_59   float32
 64  feature_60   float32
```

```
 65  feature_61   float32
 66  feature_62   float32
 67  feature_63   float32
 68  feature_64   float32
 69  feature_65   float32
 70  feature_66   float32
 71  feature_67   float32
 72  feature_68   float32
 73  feature_69   float32
 74  feature_70   float32
 75  feature_71   float32
 76  feature_72   float32
 77  feature_73   float32
 78  feature_74   float32
 79  feature_75   float32
 80  feature_76   float32
 81  feature_77   float32
 82  feature_78   float32
 83  responder_0  float32
 84  responder_1  float32
 85  responder_2  float32
 86  responder_3  float32
 87  responder_4  float32
 88  responder_5  float32
 89  responder_6  float32
 90  responder_7  float32
 91  responder_8  float32
dtypes: float32(86), int16(3), int8(3)
memory usage: 654.5 MB
```

In [ ]:

```python
def print_empty_cols(df):
    empty_columns = []
    fully_filled_columns = []
    partially_empty_columns = []

    for feature in df.columns:
        # Count empty and non-empty rows
        empty_rows = df[feature].isnull().sum()
        nonempty_rows = len(df[feature]) - empty_rows


        # Classify the columns based on the counts
        if nonempty_rows == 0:
            empty_columns.append(feature)
        elif empty_rows == 0:
            fully_filled_columns.append(feature)
        else:
            partially_empty_columns.append(feature)

        # Print feature statistics
        print(f'{feature} : total - {len(df[feature])} - empty - {empty_rows} - nonempty - {nonempty_rows}')

print_empty_cols(df)
```

```
date_id : total - 1944210 - empty - 0 - nonempty - 1944210
time_id : total - 1944210 - empty - 0 - nonempty - 1944210
symbol_id : total - 1944210 - empty - 0 - nonempty - 1944210
weight : total - 1944210 - empty - 0 - nonempty - 1944210
feature_00 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_01 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_02 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_03 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_04 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_05 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_06 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_07 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_08 : total - 1944210 - empty - 16980 - nonempty - 1927230
feature_09 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_10 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_11 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_12 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_13 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_14 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_15 : total - 1944210 - empty - 54992 - nonempty - 1889218
feature_16 : total - 1944210 - empty - 63 - nonempty - 1944147
feature_17 : total - 1944210 - empty - 9232 - nonempty - 1934978
feature_18 : total - 1944210 - empty - 59 - nonempty - 1944151
feature_19 : total - 1944210 - empty - 59 - nonempty - 1944151
feature_20 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_21 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_22 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_23 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_24 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_25 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_26 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_27 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_28 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_29 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_30 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_31 : total - 1944210 - empty - 1944210 - nonempty - 0
feature_32 : total - 1944210 - empty - 21737 - nonempty - 1922473
feature_33 : total - 1944210 - empty - 21737 - nonempty - 1922473
feature_34 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_35 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_36 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_37 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_38 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_39 : total - 1944210 - empty - 324732 - nonempty - 1619478
feature_40 : total - 1944210 - empty - 38328 - nonempty - 1905882
feature_41 : total - 1944210 - empty - 97113 - nonempty - 1847097
feature_42 : total - 1944210 - empty - 324732 - nonempty - 1619478
feature_43 : total - 1944210 - empty - 38328 - nonempty - 1905882
feature_44 : total - 1944210 - empty - 97113 - nonempty - 1847097
feature_45 : total - 1944210 - empty - 166374 - nonempty - 1777836
feature_46 : total - 1944210 - empty - 166374 - nonempty - 1777836
feature_47 : total - 1944210 - empty - 87 - nonempty - 1944123
feature_48 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_49 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_50 : total - 1944210 - empty - 293120 - nonempty - 1651090
feature_51 : total - 1944210 - empty - 2290 - nonempty - 1941920
feature_52 : total - 1944210 - empty - 64120 - nonempty - 1880090
feature_53 : total - 1944210 - empty - 293120 - nonempty - 1651090
feature_54 : total - 1944210 - empty - 2290 - nonempty - 1941920
feature_55 : total - 1944210 - empty - 64120 - nonempty - 1880090
feature_56 : total - 1944210 - empty - 59 - nonempty - 1944151
feature_57 : total - 1944210 - empty - 59 - nonempty - 1944151
feature_58 : total - 1944210 - empty - 21732 - nonempty - 1922478
feature_59 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_60 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_61 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_62 : total - 1944210 - empty - 153999 - nonempty - 1790211
feature_63 : total - 1944210 - empty - 133274 - nonempty - 1810936
feature_64 : total - 1944210 - empty - 136458 - nonempty - 1807752
feature_65 : total - 1944210 - empty - 166374 - nonempty - 1777836
```

```
feature_66 : total - 1944210 - empty - 166374 - nonempty - 1777836
feature_67 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_68 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_69 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_70 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_71 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_72 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_73 : total - 1944210 - empty - 21732 - nonempty - 1922478
feature_74 : total - 1944210 - empty - 21732 - nonempty - 1922478
feature_75 : total - 1944210 - empty - 16 - nonempty - 1944194
feature_76 : total - 1944210 - empty - 16 - nonempty - 1944194
feature_77 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_78 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_0 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_1 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_2 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_3 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_4 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_5 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_6 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_7 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_8 : total - 1944210 - empty - 0 - nonempty - 1944210
```

In [4]:
```
# from IPython.display import display

# with pd.option_context('display.max_rows', None, 'display.max_columns', None):  # more options can be spec

#     display(df.describe().drop(['date_id', 'time_id', 'symbol_id', 'weight'], axis = 1).T.style.background
```

In [5]:
```
print("Unique Symbol Id in Day 0 :",df[df.date_id==0].symbol_id.unique())
print("Unique Symbol Id in Day 0 :",df[df.date_id==1].symbol_id.unique())

print("Difference between the days we have for symbol id 1 and days we have for symbol_id 7")
print(set(df[df.symbol_id == 1].date_id.unique()) - set(df[df.symbol_id == 7].date_id.unique()))
```

```
Unique Symbol Id in Day 0 : [ 1  7  9 10 14 16 19 33]
Unique Symbol Id in Day 0 : [ 0  1  2  7  9 10 13 15 16 19 33 38]
Difference between the days we have for symbol id 1 and days we have for symbol_id 7
{97, 53, 71}
```

In [6]:
```
mask1 = (df.symbol_id == 1) & (df.date_id == 0)

mask2 = (df.symbol_id == 7) & (df.date_id == 0)

list(set(df[mask1].time_id.unique()) - set(df[mask2].date_id.unique()))[:10]
```

Out[6]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Data Processing

In [7]:
```
df = df.drop(columns = ['responder_0','responder_1','responder_2','responder_3', 'responder_4', 'responder_5
df.head()
```

Out[7]:

	date_id	time_id	symbol_id	weight	feature_00	feature_01	feature_02	feature_03	feature_04	feature_05	...	fe
0	0	0	1	3.889038	NaN	NaN	NaN	NaN	NaN	0.851033	...	
1	0	0	7	1.370613	NaN	NaN	NaN	NaN	NaN	0.676961	...	
2	0	0	9	2.285698	NaN	NaN	NaN	NaN	NaN	1.056285	...	
3	0	0	10	0.690606	NaN	NaN	NaN	NaN	NaN	1.139366	...	
4	0	0	14	0.440570	NaN	NaN	NaN	NaN	NaN	0.955200	...	

5 rows × 84 columns

Percentage Based Column Filtering

In [8]:
```
# Sort the DataFrame
# The symbol_id refers to the individual stock. Since this data is chronological.
# It must be sorted into groups: by stock then day, then time.

df = df.sort_values(by=['symbol_id', 'date_id', 'time_id'])
df = df.reset_index(drop=True)
df.head(100)
```

Out[8]:

	date_id	time_id	symbol_id	weight	feature_00	feature_01	feature_02	feature_03	feature_04	feature_05	...
0	1	0	0	1.749479	NaN	NaN	NaN	NaN	NaN	0.053447	...
1	1	1	0	1.749479	NaN	NaN	NaN	NaN	NaN	-0.029047	...
2	1	2	0	1.749479	NaN	NaN	NaN	NaN	NaN	0.017732	...
3	1	3	0	1.749479	NaN	NaN	NaN	NaN	NaN	0.247528	...
4	1	4	0	1.749479	NaN	NaN	NaN	NaN	NaN	-0.024495	...
...	...	...	...	...	...	...	...	...	...	...	...
95	1	95	0	1.749479	NaN	NaN	NaN	NaN	NaN	2.692226	...
96	1	96	0	1.749479	NaN	NaN	NaN	NaN	NaN	1.865496	...
97	1	97	0	1.749479	NaN	NaN	NaN	NaN	NaN	1.897475	...
98	1	98	0	1.749479	NaN	NaN	NaN	NaN	NaN	1.237614	...
99	1	99	0	1.749479	NaN	NaN	NaN	NaN	NaN	1.700521	...

100 rows × 84 columns

From the data exploration we saw that each feature column is not completely full. There are random points at which there are NANs or empty datapoints.

Thus, need to be accounted for in some way.

Our method is to process the data frame by stock (symbol_id). Each group contains all the time data (features) for a given stock. Our goal is to check the percentage of missing rows (NANs) for each group and individual feature.

- If a feature for a stock is completely missing we drop the feature for the whole dataframe not just the group/stock.
- If the number of missing rows for a feature of a given stock exceeds a given percentage threshold, then the feature is also dropped for the whole dataframe. In this code the percentage threshold can be easily altered and is arbitrarily

set to 10%.

- If there are any empty rows for the feature in that group then we forward and back fill those rows. Since the data is chronologically sorted in the previous step, ffill() and bfill() take the previous or forward time step data. They do not take random data from some other place in time.
- If there are no empty rows nothing happens

The code uses lists: dropped columns, filled columns, and percent filled. These are used to track the changes to the dataframe and print out the changes made in the data preprocessing for each group. The processed_groups dataframe is used to store the changes of all the groups.

There is still an issue though with using ffill and bfill. If a symbol_id group starts with a significant portion of NaNs at the boundaries of the group (start and end), in feature_40 for example, ffill and bfill actually add more Nans. In these cases there are no valid non-NaN values to propagate using ffill or bfill. The percentage threshold, here set to 5%, allows feature_40 to remain despite having significant missing values in specific groups. Even after ffill and bfill residual NaNs remain due to boundary conditions, causing the incomplete filling.

In [9]:
```
# The % threshold of missing values for a given symbol_ID group at which to delete a feature from the whole
percentage_threshold = 5

# The processed_groups dataframe is used to store the changes for a given symbol ID.
processed_groups = pd.DataFrame(columns = ['Symbol ID', 'Dropped Columns', 'Filled Columns', '% Filled'])


# Iterates on the df into groups by symbol ID. These are already sorted by symbol, time, and date in the pre
for symbol_id, group in df.groupby('symbol_id'):

    # lists to store the changes made to each group
    dropped_columns = []
    filled_columns = []
    percent_filled = []

    for feature in group.columns:
        # calculates percentage of missing rows for a feature in the group
        total_rows = len(group)
        empty_rows = group[feature].isnull().sum()
        empty_rows_percentage = (empty_rows / total_rows) * 100
        nonempty_rows_percentage = 100 - empty_rows_percentage

        # if the feature is empty or the % threshold of missing rows is exceeded --> delete the feature from
        if empty_rows == total_rows or empty_rows_percentage >= percentage_threshold:
            if feature in df.columns:
                df = df.drop(columns=feature)
                dropped_columns.append(feature)


        elif empty_rows > 0:      # Forward-fill and backward-fill missing values if lower than threshold an
            df.loc[group.index,feature] = group[feature].ffill().bfill()
            filled_columns.append(feature)
            percent_filled.append(empty_rows_percentage)

    # Check if there is data to add to the df for a given symbol_id group
    if dropped_columns or filled_columns or percent_filled:
        new_row = pd.DataFrame({
            'Symbol ID': [symbol_id],
            'Dropped Columns': [dropped_columns],
            'Filled Columns': [filled_columns],
            '% Filled': [percent_filled]
        })
    processed_groups = pd.concat([processed_groups, new_row], ignore_index=True)

remaining_null_columns = df.columns[df.isnull().any()]
print("Columns with remaining NaNs after ffill and bfill:", remaining_null_columns)
df = df.drop(columns=remaining_null_columns)
```

```
# print(processed_groups)
```

```
Columns with remaining NaNs after ffill and bfill: Index(['feature_40', 'feature_41', 'feature_43', 'feature_
44', 'feature_63'], dtype='object')
```

In [10]:
```
# This is just a double check if there any missing rows left

null_columns = df.columns[df.isnull().any()]
print(null_columns)
```

```
Index([], dtype='object')
```

In [11]:
```
print_empty_cols(df)
```

```
date_id : total - 1944210 - empty - 0 - nonempty - 1944210
time_id : total - 1944210 - empty - 0 - nonempty - 1944210
symbol_id : total - 1944210 - empty - 0 - nonempty - 1944210
weight : total - 1944210 - empty - 0 - nonempty - 1944210
feature_05 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_06 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_07 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_08 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_09 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_10 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_11 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_12 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_13 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_14 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_15 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_16 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_17 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_18 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_19 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_20 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_22 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_23 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_24 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_25 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_28 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_29 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_30 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_32 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_33 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_34 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_35 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_36 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_37 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_38 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_47 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_48 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_49 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_51 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_52 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_54 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_55 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_56 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_57 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_58 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_59 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_60 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_61 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_67 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_68 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_69 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_70 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_71 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_72 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_73 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_74 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_75 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_76 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_77 : total - 1944210 - empty - 0 - nonempty - 1944210
feature_78 : total - 1944210 - empty - 0 - nonempty - 1944210
responder_6 : total - 1944210 - empty - 0 - nonempty - 1944210
```

Here we can see how there are no more NaNs in the df.

Principal Component Analysis - PCA

One way to get rid of features and to reduce the size of our large data set is through Principal Component Analysis.

PCA transforms the feature set into a smaller and new set of uncorrelated variables called principal components (PCs). Our original feature set has 79 un-named features that we know nothing about. They could have linear relationships between them (they could be inter-correlated) that makes using all 79 features redundant and computationally expensive. It could also lead to overfitting. PCA helps to solve this by reducing the dimensionality of the data into a set of uncorrelated PCs. Each PC is essentially a distillation/reduction of the feature set. The PCs capture the most significant variations in the data as possible. These are new variables created by linear combinations of the original variables, where the first principal component (PC #1) explains the most variance in the data. The subsequent components progressively explain less and less variance (PC #2, PC #3, etc.). PCA inherently removes multicollinearity between the features by transforming the features into orthogonal PCs.

How the algorithm works:

- calculate the covariance matrix of the data to determines the relationships between variables.
- The covariance matrix is then decomposed into eigenvalues and eigenvector. The eigenvectors represent the directions of maximum variance and the eigenvalues represent the amount of variance explained by each component
- A subset of principal components with the highest variance are chosen to represent the feature set. These are specified when running the model.

However the data must be normalized first before running principle component analysis.

- PCA works by identifying the directions (principal components) that maximize the variance in the data. If the features have different scales (e.g., one feature ranges from 0 to 1, while another ranges from 0 to 1000), the variance of the features with larger ranges will dominate. This would cause PCA to incorrectly weight those features in its calculations.

Thus we must use Standard Scaler before before implementingre PCA.

Standard Scaler ensures:

- All features are centered around 0 with unit variance (mean = 0, variance = 1).
- The variance of each feature contributes equally when PCA calculates the covariance matrix of the feature set

However standard scaler must be performed on the individual symbol_id groups (the full set of date and time id's for a given symbol id). Normalizing each group independently is necessary as each symbol id represents distinct entities (individual stocks) with no shared scale or distribution. At least we have no way on knowing since the data is unlabeled. Thus to be safe, we decided to apply standard scaler on individual symbol_id groups. Furthermore, by applying standard scalar, PCA will now identify meaningful principal components rather than being skewed by unscaled feature magnitudes.

For the PCA calculations however, this must be applied globally. Using the sklearn PCA model, we chose the model give us the the top 'n' PCs that represent a 90% of the variance of the feature set. It this threshold was applied to each symbol id_group, we might get three PC columns needed to reach the threshold for the 1st symbol_id and then 5 needed for the 2nd symbol_id group. This means that the df of the final feature set could not merge the symbol_id PC columns (different dimensions). Additionally, if we were to use the largest number of PC columns needed to reach the threshold (e.g. 6 PC columns due to symbol_id 2) and share this with the rest of the symbol_id groups, this would cause, for example, symbol_id to explain more than 90% of the variance. Thus we decided to apply PCA globally on the feature set.

Normalizing the data

- We must normalize only on the remaining features. Not on time,data,symbol,etc.

In [12]:
```
# Normalize just the features

from sklearn.preprocessing import StandardScaler

df_feature_cols = []

# All the possible feature columns in the original data file
possible_feature_cols = [f'feature_{i:02}' for i in range(0,79)]
```

```
# Check which features are remaining after the preprocessing
for feature in df.columns:
    if feature in possible_feature_cols:
        df_feature_cols.append(feature)


# Standardize the individual groups and only on the remaining features
for symbol_id, group in df.groupby('symbol_id'):
    std_scaler = StandardScaler()
    df.loc[group.index, df_feature_cols] = std_scaler.fit_transform(group[df_feature_cols])
```

- Lets make sure that the standard scaler is applied correctly. Standard scaler ensures that each feature has a mean of 0 and a standard deviation of 1.
- If the standard deviation if 1, the variance becomes: Variance=(Standard Deviation)^2 = 1^2 =1
- This is why most if all features now have a variance of 1.

In [13]:
```
variances = df.drop(['date_id', 'time_id', 'symbol_id', 'weight'], axis=1).var()
summary = df.describe().drop(['date_id', 'time_id', 'symbol_id', 'weight'], axis=1).T
summary['variance'] = variances

with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    display(summary.style.background_gradient(cmap='coolwarm'))
```

	count	mean	std	min	25%	50%	75%	max	variance
feature_05	1944210.000000	-0.000000	1.000000	-13.166103	-0.468021	-0.014746	0.426808	15.371867	1.000000
feature_06	1944210.000000	0.000000	1.000000	-23.328115	-0.366534	-0.006040	0.339358	24.230495	1.000000
feature_07	1944210.000000	-0.000000	1.000000	-18.487604	-0.420731	-0.014323	0.382832	25.195225	1.000000
feature_08	1944210.000000	-0.000000	1.000000	-7.610195	-0.535809	-0.034787	0.469149	15.806594	1.000000
feature_09	1944210.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
feature_10	1944210.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
feature_11	1944210.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
feature_12	1944210.000000	-0.000000	1.000000	-2.351544	-0.533084	-0.232653	0.222173	23.696554	1.000000
feature_13	1944210.000000	-0.000000	1.000000	-1.172081	-0.403503	-0.260976	-0.003155	63.608902	1.000000
feature_14	1944210.000000	-0.000000	1.000000	-2.019938	-0.495114	-0.259361	0.158226	35.363472	1.000000
feature_15	1944210.000000	-0.000000	1.000000	-5.197333	-0.593919	-0.081250	0.466415	50.338348	1.000000
feature_16	1944210.000000	0.000000	1.000000	-4.129604	-0.631197	-0.052489	0.538546	119.160233	1.000000
feature_17	1944210.000000	-0.000000	1.000000	-6.187441	-0.582245	-0.074060	0.462158	92.464577	1.000000
feature_18	1944210.000000	-0.000000	1.000000	-5.891507	-0.689698	-0.069746	0.656459	5.539991	1.000000
feature_19	1944210.000000	-0.000000	1.000000	-4.454686	-0.664753	-0.021526	0.670414	5.350324	1.000000
feature_20	1944210.000000	-0.000000	1.000000	-3.561182	-0.669112	-0.099683	0.561899	4.296779	1.000000
feature_22	1944210.000000	-0.000000	1.000000	-3.343972	-0.720133	-0.087744	0.647601	4.030527	1.000000
feature_23	1944210.000000	-0.000000	1.000000	-3.263179	-0.711540	-0.062431	0.630285	3.483932	1.000000
feature_24	1944210.000000	0.000000	1.000000	-3.313749	-0.679841	-0.048756	0.650203	4.484638	1.000000
feature_25	1944210.000000	0.000000	1.000000	-2.421045	-0.740209	-0.078972	0.685843	3.507091	1.000000
feature_28	1944210.000000	0.000000	1.000000	-3.776496	-0.765847	-0.058145	0.772709	3.234708	1.000000
feature_29	1944210.000000	-0.000000	1.000000	-3.518775	-0.720044	-0.057837	0.687878	3.193416	1.000000
feature_30	1944210.000000	0.000000	1.000000	-3.798231	-0.693908	-0.063695	0.634146	5.601021	1.000000
feature_32	1944210.000000	0.000000	1.000000	-4.766582	-0.642212	0.055264	0.661374	5.555742	1.000000
feature_33	1944210.000000	-0.000000	1.000000	-3.894441	-0.733991	-0.004069	0.749473	4.111681	1.000000
feature_34	1944210.000000	0.000000	1.000000	-5.944628	-0.659473	0.062765	0.713081	6.675881	1.000000
feature_35	1944210.000000	-0.000000	1.000000	-5.192869	-0.652447	0.065446	0.702835	6.461654	1.000000
feature_36	1944210.000000	0.000000	1.000000	-4.108497	-0.702369	0.003905	0.704638	4.106891	1.000000
feature_37	1944210.000000	0.000000	1.000000	-14.334377	-0.496780	-0.039826	0.408453	17.190060	1.000000
feature_38	1944210.000000	-0.000000	1.000000	-11.228268	-0.497587	-0.042660	0.431190	13.351240	1.000000
feature_47	1944210.000000	-0.000000	1.000000	-29.518755	-0.263653	0.015229	0.290999	40.947750	1.000000
feature_48	1944210.000000	0.000000	1.000000	-138.192307	-0.016447	0.001483	0.019764	150.786041	1.000000
feature_49	1944210.000000	-0.000000	1.000000	-57.132011	-0.193486	0.001804	0.201982	88.222725	1.000000
feature_51	1944210.000000	-0.000000	1.000000	-3.878741	-0.697721	0.233949	0.705527	7.974266	1.000000
feature_52	1944210.000000	-0.000000	1.000000	-5.835448	-0.580099	-0.006558	0.635203	6.106771	1.000000
feature_54	1944210.000000	-0.000000	1.000000	-6.520976	-0.756055	-0.222538	0.891157	3.758827	1.000000
feature_55	1944210.000000	-0.000000	1.000000	-6.155972	-0.635994	-0.014445	0.566907	5.711173	1.000000
feature_56	1944210.000000	0.000000	1.000000	-4.711965	-0.692926	-0.005648	0.685411	4.283638	1.000000

	count	mean	std	min	25%	50%	75%	max	variance
feature_57	1944210.000000	0.000000	1.000000	-4.213375	-0.649796	0.014693	0.670767	4.740458	1.000000
feature_58	1944210.000000	-0.000000	1.000000	-35.479267	-0.405970	0.010344	0.413654	55.390072	1.000000
feature_59	1944210.000000	-0.000000	1.000000	-147.035217	-0.119770	0.001685	0.114837	144.772568	1.000000
feature_60	1944210.000000	0.000000	1.000000	-63.914101	-0.312314	0.005454	0.313467	85.623611	1.000000
feature_61	1944210.000000	-0.000000	1.000000	-4.688773	-0.699467	0.002340	0.595550	5.083373	1.000000
feature_67	1944210.000000	-0.000000	1.000000	-2.388613	-0.518969	-0.246463	0.191484	20.348713	1.000000
feature_68	1944210.000000	-0.000000	1.000000	-0.949288	-0.346585	-0.240594	-0.066656	89.502502	1.000000
feature_69	1944210.000000	-0.000000	1.000000	-1.677922	-0.463345	-0.265142	0.111516	36.619137	1.000000
feature_70	1944210.000000	0.000000	1.000000	-2.341202	-0.540018	-0.225924	0.234372	28.002213	1.000000
feature_71	1944210.000000	0.000000	1.000000	-0.940122	-0.370391	-0.248438	-0.054228	74.819794	1.000000
feature_72	1944210.000000	0.000000	1.000000	-1.741317	-0.501631	-0.260576	0.161499	40.134617	1.000000
feature_73	1944210.000000	-0.000000	1.000000	-4.267814	-0.523735	-0.170183	0.324260	74.236702	1.000000
feature_74	1944210.000000	-0.000000	1.000000	-5.775111	-0.525577	-0.147598	0.351097	56.778591	1.000000
feature_75	1944210.000000	-0.000000	1.000000	-3.905784	-0.399647	-0.225630	0.065168	54.933083	1.000000
feature_76	1944210.000000	-0.000000	1.000000	-4.945653	-0.413956	-0.211217	0.105107	49.145046	1.000000
feature_77	1944210.000000	-0.000000	1.000000	-4.258845	-0.474893	-0.200676	0.227788	80.408401	1.000000
feature_78	1944210.000000	0.000000	1.000000	-6.548354	-0.482000	-0.179732	0.263211	73.618484	1.000000
responder_6	1944210.000000	0.001488	0.870577	-5.000000	-0.355871	-0.009597	0.336100	5.000000	0.757904

- We can also see that features 9, 10, and 11 do not have a variance of 1. This is because their min and max values are all very close to the mean of 0. Thus there is no variance in the data.
- Thus these features are not significant in predicting responder 6 if they do not change at all with time.

Running PCA

In [14]:
```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np

# Threshold of number of PC Columns that explain 90% of the data
threshold = 0.9

pca = PCA()
pca.fit(df[df_feature_cols])

explained_variance = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)


# +1 since it is 0-indexed. So result can give actual number of components
threshold_components = np.argmax(cumulative_variance >= threshold) + 1

pca_optimal = PCA(n_components = threshold_components)
pca_data_cols = pca_optimal.fit_transform(df[df_feature_cols])

for i in range(1,threshold_components + 1):
    df[f'PCA_{i}'] = pca_data_cols[:,i-1]

df = df.drop(columns = df_feature_cols)

df.head()
```

Out[14]:

	date_id	time_id	symbol_id	weight	responder_6	PCA_1	PCA_2	PCA_3	PCA_4	PCA_5	...	PCA_
0	1	0	0	1.749479	2.337418	-1.659468	3.545253	-2.143758	4.254129	1.005976	...	-1.1907
1	1	1	0	1.749479	2.492198	0.727005	3.062852	-1.060689	2.383633	-1.791936	...	-0.6579
2	1	2	0	1.749479	1.993902	0.637612	2.558855	-1.077409	2.478476	-1.495102	...	-1.1036
3	1	3	0	1.749479	1.864082	0.062889	1.868430	-0.807554	2.518893	-1.250505	...	-0.6790
4	1	4	0	1.749479	2.604931	-0.405855	1.063370	-0.499962	2.391226	-1.914477	...	-0.6302

5 rows × 34 columns

Thus, 29 PCs are necessary to reach the 90% variance threshold for this feature set.

In [15]:
```
# # Number of principal components to initially calculate from a symbol_id's feature set
# n = 3

# for symbol_id, group in df.groupby('symbol_id'):
#     pca = PCA(n_components = n)
#     pca_data_cols = pca.fit_transform(group[df_feature_cols])
#     df.loc[group.index, [f'PCA_{i}' for i in range(1,n+1)]] = pca_data_cols

# df = df.drop(columns = df_feature_cols)

# df.head()
```

One Hot Encoding

In our data set the numeric representation of Symbol_id (e.g. '1' or '2') doesn't imply order:

- Symbol_id is a categorical variable. Thus the numeric value of symbol_id does not have a meaningful order (e.g., 1, 2, 3 aren't inherently "larger" or "smaller").

One Hot Encoding avoids misinterpretation by the models we will use:

- Using raw numeric values could mislead the model into interpreting the column as continuous (e.g., implying a distance between symbol_id = 1 and symbol_id = 2). One-Hot Encoding converts categorical variables into binary indicators that can be safely used in our machine learning models.

In [16]:
```
encoded = pd.get_dummies(df['symbol_id'], prefix='symbol_id')
max_symbol_id = df['symbol_id'].max()
print(f'This parquet has {max_symbol_id} distinct symbol_ids')
encoded.head()
```

```
This parquet has 38 distinct symbol_ids
```

Out[16]:

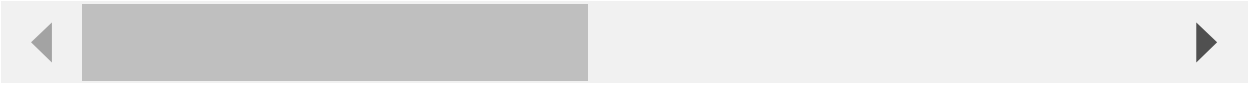
	symbol_id_0	symbol_id_1	symbol_id_2	symbol_id_3	symbol_id_7	symbol_id_8	symbol_id_9	symbol_id_10	symbol
0	True	False	False	False	False	False	False	False	
1	True	False	False	False	False	False	False	False	
2	True	False	False	False	False	False	False	False	
3	True	False	False	False	False	False	False	False	
4	True	False	False	False	False	False	False	False	

In [17]:
```
df = pd.concat([df, encoded], axis = 1)
df = df.drop(['symbol_id'], axis = 1)
df.head()
```

Out[17]:

	date_id	time_id	weight	responder_6	PCA_1	PCA_2	PCA_3	PCA_4	PCA_5	PCA_6	...	symbol_
0	1	0	1.749479	2.337418	-1.659468	3.545253	-2.143758	4.254129	1.005976	-3.453507	...	
1	1	1	1.749479	2.492198	0.727005	3.062852	-1.060689	2.383633	-1.791936	-1.978490	...	
2	1	2	1.749479	1.993902	0.637612	2.558855	-1.077409	2.478476	-1.495102	-1.656684	...	
3	1	3	1.749479	1.864082	0.062889	1.868430	-0.807554	2.518893	-1.250505	-1.905784	...	
4	1	4	1.749479	2.604931	-0.405855	1.063370	-0.499962	2.391226	-1.914477	-1.422862	...	

5 rows × 53 columns



Temporal Splitting

In [18]:
```
df = df.sort_values(['date_id', 'time_id'])
date_counts = df.date_id.value_counts()
```

In [19]:
```
date_counts = pd.DataFrame(date_counts.sort_index())
date_counts['cumulative_sum'] = date_counts['count'].cumsum()
date_counts.head()
```

Out[19]:

	count	cumulative_sum
date_id		
0	6792	6792
1	10188	16980
2	9339	26319
3	10188	36507
4	10188	46695

In [20]:
```
total = len(df)
train_percentage = 0.6
val_percentage = 0.2
test_percentage = 0.2

apprx_train_len = int(total*train_percentage)
apprx_val_len = int(total*val_percentage)
apprx_test_len = total - apprx_train_len - apprx_val_len



def split_func(row):
    s = row['cumulative_sum']

    if s <= apprx_train_len:
        return 'Train'

    elif (s > apprx_train_len) and (s <= apprx_train_len + apprx_val_len):
        return 'Val'

    elif (s > apprx_train_len + apprx_val_len):
        return 'Test'

    else:
```

```
        raise ValueError



date_counts['Split'] = date_counts.apply(split_func, axis = 1)
print(date_counts.Split.value_counts())
date_counts.head()
```

```
Split
Train    113
Val       30
Test      27
Name: count, dtype: int64
```

Out[20]:

date_id	count	cumulative_sum	Split
0	6792	6792	Train
1	10188	16980	Train
2	9339	26319	Train
3	10188	36507	Train
4	10188	46695	Train

In [21]:
```
last_train_data = date_counts[date_counts.Split == 'Train'].tail(1)

first_test_data = date_counts[date_counts.Split == 'Test'].head(1)
```

In [22]:
```
last_train_index = last_train_data.index[0]
first_test_index = first_test_data.index[0]


df['Split'] = 'Test'
df.loc[df['date_id'] <= last_train_index, 'Split'] = 'Train'
df.loc[(df['date_id'] > last_train_index) & (df['date_id'] < first_test_index), 'Split'] = 'Val'

print(df['Split'].value_counts())
```

```
Split
Train    1158036
Test      398181
Val       387993
Name: count, dtype: int64
```

In [23]:
```
train_df = df[df.Split == 'Train']
val_df = df[df.Split == 'Val']
test_df = df[df.Split == 'Test']
```

Setting the features

Our final features are:

- Date_id
- Time_id
- PCA columns 1 -> n
- Boolean Symbol_id columns

In [24]:
```
TEMPORAL_FEATURES = ['date_id', 'time_id']

MARKET_FEATURES = [f'PCA_{i}' for i in range(1,i+1)]

SYMBOL_FEATURES = [f'symbol_id_{i}' for i in range(max_symbol_id) if f'symbol_id_{i}' in df.columns]
```

In [25]:
```
ALL_FEATURES = TEMPORAL_FEATURES + MARKET_FEATURES + SYMBOL_FEATURES
```

In [26]:
```
train_x = train_df[ALL_FEATURES]
train_y = train_df[['responder_6']]

val_x = val_df[ALL_FEATURES]
val_y = val_df[['responder_6']]

test_x = test_df[ALL_FEATURES]
test_y = test_df[['responder_6']]
```

Regression Models

In [27]:
```
from sklearn.feature_selection import chi2
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import Ridge, Lasso, ElasticNet
```

Ridge Regression

Penalty: Squares the magnitude of the coefficients in loss function

$$Cost Function:J(β) = RSS + α ∑βᵢ^2$$
Where:

- RSS is the Residual Sum of Squares (error term).
- β_k are the model's coefficients.
- α controls the strength of the penalty. Larger α forces coefficients closer to zero.

Behavior:

- Penalizes large coefficients by squaring them.
- Reduces multicollinearity (correlation between predictors).
- Keeps all features but shrinks their coefficients towards zero.
- Best for situations where many predictors are useful.

Thus we would want to use ridge regression when feature reduction is not possible and multicollinearity is present. However, we have PCA features which are linearly uncorrelated by design. PCA inherently removes multicollinearity by transforming the features into orthogonal principal components. Fortunately, ridge regression can still be beneficial when using PCA features with high-dimensionality that could cause overfitting.

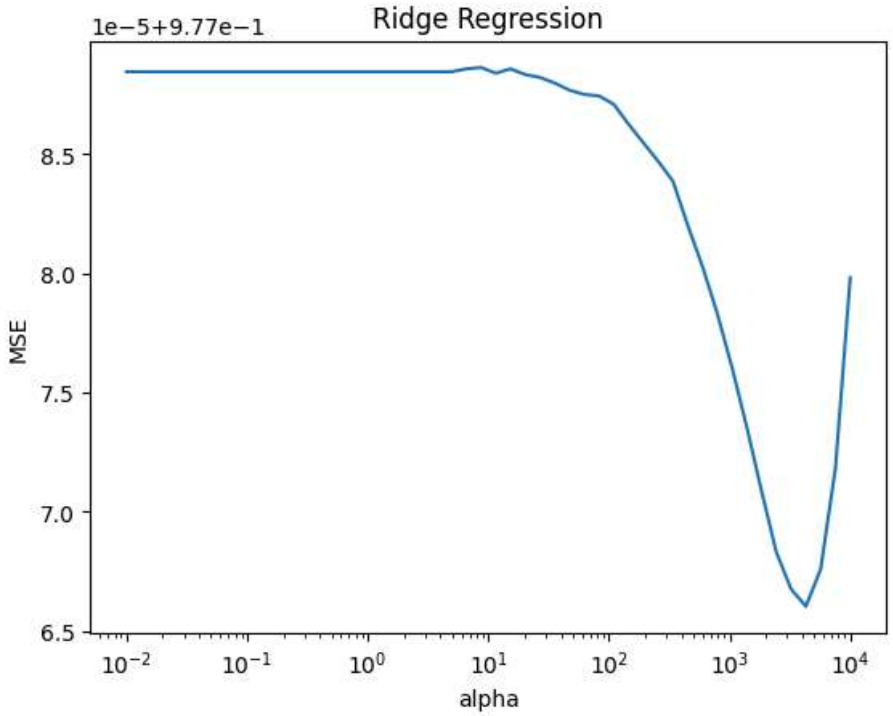
Ridge regression introduces a penalty that shrinks the coefficients of less important components, helping to stabilize the model and improve generalization to unseen data. This combination ensures that the PCA captures the most variance while ridge regression handles potential overfitting and ensures robust predictions.

In [28]:
```
## for progress bar in notebook
from tqdm import tqdm
```

In [29]:
```
ridge_mse = []

# Ridge Regression
# Typical values of alpha are 0.01 to 10,000
# 20 alpha values from 0.01 to 10000 in logarithmic scaling
alpha_counter = np.logspace(-2, 4, 50)

import warnings

with warnings.catch_warnings():
    ## to ignore the LinAlgWarning
```

```
    warnings.simplefilter('ignore')
    for a in tqdm(alpha_counter):
        ridge = Ridge(alpha=a, random_state=42)
        ridge.fit(train_x, train_y)
        ridge_y_pred = ridge.predict(test_x)
        ridge_mse.append(mean_squared_error(test_y, ridge_y_pred))
        del ridge



plt.plot(alpha_counter,ridge_mse)
plt.xscale('log')
plt.title('Ridge Regression')
plt.xlabel('alpha')
plt.ylabel('MSE')
plt.show()



# argmin() finds index of minimnum value
min_mse_index = np.argmin(ridge_mse)
print(f'Minimum MSE = {ridge_mse[min_mse_index]} at alpha = {alpha_counter[min_mse_index]}')
ra = alpha_counter[min_mse_index]
```

```
100%|██████████| 50/50 [00:28<00:00,  1.76it/s]
```




```
Minimum MSE = 0.9770660400390625 at alpha = 4291.934260128778
```

Lasso Regression

Penalty: Takes the absolute value of the coefficients in loss function

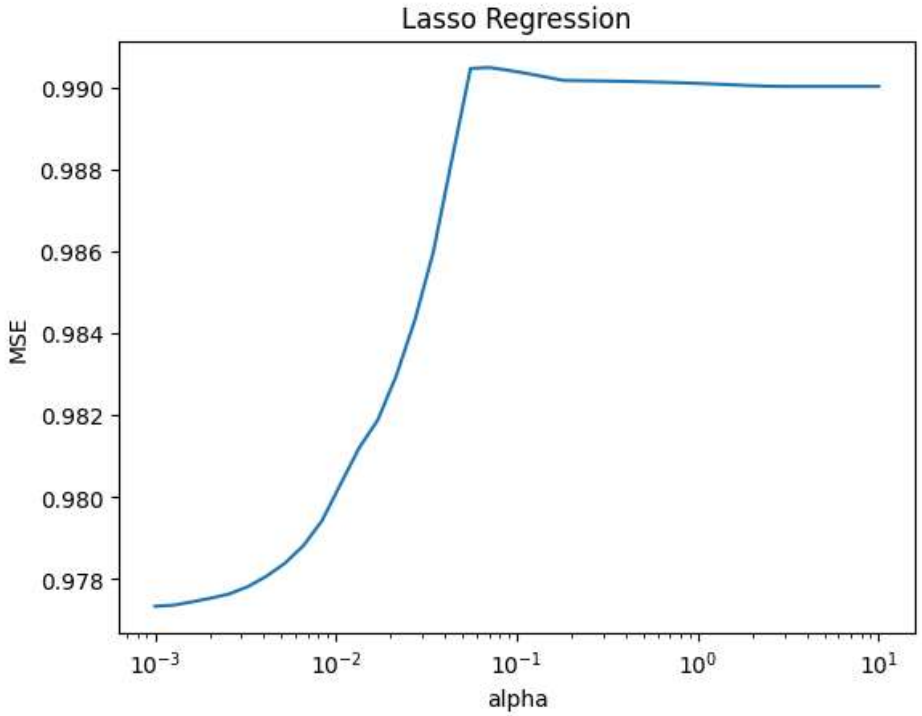
$$Cost Function:J(β) = RSS + α ∑ |βᵢ|$$
Behavior:

- Penalizes large coefficients by their absolute values.
- Performs feature selection: can set some coefficients to exactly 0, effectively removing irrelevant features.
- Best for situations where only a subset of predictors is relevant

In [30]:
```
lasso_mse = []
Lalpha_counter = np.logspace(-3, 1, 40)
```

```
for a in tqdm(Lalpha_counter):
    lasso = Lasso(alpha=a, random_state=42)
    lasso.fit(train_x, train_y)
    lasso_y_pred = lasso.predict(test_x)
    lasso_mse.append(mean_squared_error(test_y, lasso_y_pred))
    del lasso


plt.plot(Lalpha_counter,lasso_mse)
plt.xscale('log')
plt.title('Lasso Regression')
plt.xlabel('alpha')
plt.ylabel('MSE')
plt.show()



# argmin() finds index of minimnum value
min_mse_index = np.argmin(lasso_mse)
print(f'Minimum MSE = {lasso_mse[min_mse_index]} at alpha = {Lalpha_counter[min_mse_index]}')
la = Lalpha_counter[min_mse_index]
```

```
100%|██████████| 40/40 [02:10<00:00,  3.27s/it]
```




```
Minimum MSE = 0.9773330092430115 at alpha = 0.001
```

Checking Elastic Results

Elastic Regression combines both Ridge and Lasso Regression

$$Cost Function:J(β) = RSS + α1 ∑βᵢ^2 + α2 ∑ |βᵢ|$$
Behavior:

- Hybrid approach: Balances feature selection (Lasso) and coefficient shrinkage (Ridge).
- Useful when there are multiple correlated features, and Lasso struggles to choose one.

In [31]:
```
l1_alpha_values = np.logspace(-3, 2, 10)  # 10 values from 0.001 to 100
l1_ratios = np.linspace(0.1, 1.0, 10)  # 10 values from 0.1 to 1.0
```

```
elastic_mse_results = np.zeros((len(l1_alpha_values), len(l1_ratios)))

with tqdm(total=len(l1_alpha_values)*len(l1_ratios)) as progress_bar:
    for i, a in enumerate(l1_alpha_values):
        for j, l1 in enumerate(l1_ratios):
            elastic_net = ElasticNet(alpha=a, l1_ratio=l1, random_state=42)
            elastic_net.fit(train_x, train_y)
            elastic_net_y_pred = elastic_net.predict(test_x)
            elastic_mse_results[i, j] = mean_squared_error(test_y, elastic_net_y_pred)
            progress_bar.update(1)

min_mse_index = np.unravel_index(np.argmin(elastic_mse_results, axis=None), elastic_mse_results.shape)
optimal_alpha = l1_alpha_values[min_mse_index[0]]
optimal_l1_ratio = l1_ratios[min_mse_index[1]]



print(f"Optimal Alpha: {optimal_alpha}")
print(f"Optimal L1 Ratio: {optimal_l1_ratio}")
print(f"Minimum MSE: {elastic_mse_results[min_mse_index]}")
```

```
100%|██████████| 100/100 [05:28<00:00,  3.28s/it]
Optimal Alpha: 0.001
Optimal L1 Ratio: 0.1
Minimum MSE: 0.9770938754081726
```

Decision Trees and XGBoost

In [32]:
```
# Import libraries

from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression
import xgboost as xgb
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, explained_variance_score



# Models for regression
regressors = {
    "Ridge Regression" : Ridge(alpha= ra, random_state=42),
    "Lasso Regression" : Lasso(alpha = la, random_state=42),
    "Elastic Regression" : ElasticNet(alpha=optimal_alpha, l1_ratio=optimal_l1_ratio, random_state=42),
    "Decision Tree": DecisionTreeRegressor(max_depth=5, random_state=42),
    "Linear Regression": LinearRegression(),
    "XGBoost": xgb.XGBRegressor(tree_method="hist")
}



# Performance metrics for regression
metrics = {
    "Mean Squared Error": mean_squared_error,
    "Mean Absolute Error": mean_absolute_error,
    "R^2 Score": r2_score,
}
```

In [33]:
```
import time

results_train = {}
results_val = {}

models = list(regressors.keys())
elap_time = []


mse_train = []
mse_test = []
mse_val = []
```

```
mae_train = []
mae_test = []
mae_val = []

r2_train = []
r2_test = []
r2_val = []



for model in regressors:
    start_time = time.time()

    regressors[model].fit(train_x, train_y)

    end_time = time.time()
    elapsed_time = end_time - start_time
    elap_time.append(elapsed_time)

    pred_y_train = regressors[model].predict(train_x)
    pred_y_test = regressors[model].predict(test_x)
    pred_y_val = regressors[model].predict(val_x)

    train_list = []
    val_list = []
    for metric in metrics:
        score_train = metrics[metric](train_y, pred_y_train)
        score_train = round(score_train, 4)

        score_test = metrics[metric](test_y, pred_y_test)
        score_test = round(score_test, 4)

        score_val = metrics[metric](val_y, pred_y_val)
        score_val = round(score_val, 4)

        if metric == "Mean Squared Error":
            mse_train.append(score_train)
            mse_test.append(score_test)
            mse_val.append(score_val)

        elif metric == "Mean Absolute Error":
            mae_train.append(score_train)
            mae_val.append(score_val)
            mae_test.append(score_test)

        elif metric == "R^2 Score":
            r2_train.append(score_train)
            r2_val.append(score_val)
            r2_test.append(score_test)



regression_results = pd.DataFrame({
    'Model': models,
    'Elapsed Time': elap_time,
    'MSE Train': mse_train,
    'MSE Test': mse_test,
    'MSE Val': mse_val,
    'MAE Train': mae_train,
    'MAE Test': mae_test,
    'MAE Val': mae_val,
    'R² Train': r2_train,
    'R² Test': r2_test,
    'R² Val': r2_val
})

regression_results
```

Out[33]:

	Model	Elapsed Time	MSE Train	MSE Test	MSE Val	MAE Train	MAE Test	MAE Val	R^2 Train	R^2 Test	R^2 Val
0	Ridge Regression	0.420308	0.6596	0.9771	0.7785	0.5187	0.6576	0.5721	0.0135	0.0130	0.0100
1	Lasso Regression	3.515715	0.6598	0.9773	0.7781	0.5185	0.6575	0.5719	0.0131	0.0127	0.0104
2	Elastic Regression	4.307851	0.6596	0.9771	0.7785	0.5187	0.6576	0.5721	0.0135	0.0129	0.0100
3	Decision Tree	26.765968	0.6622	0.9833	0.7836	0.5199	0.6601	0.5738	0.0095	0.0066	0.0035
4	Linear Regression	2.544657	0.6595	0.9771	0.7786	0.5187	0.6576	0.5722	0.0135	0.0129	0.0098
5	XGBoost	14.053773	0.5533	1.0511	0.8350	0.4909	0.6906	0.6013	0.1724	-0.0618	-0.0618

Results

1. Ridge, Lasso, Elastic

   - The MSE values for the MSE train are all nearly identical for these three models. This pattern occurs again with the test and validation data. The Train MSE is the lowest at around 0.659 while the MSE test data is the highest at 0.977. This indicates that the model is slightly overfitting. However, we have no way of knowing how large of a difference this is, as we do not have the units of our target variable Responder_6. All we know is lower is better. Knowing the units helps in understanding how significant the errors are in real-world terms.
   - MAE shows the same pattern: the train, test, and validation columns are the same. However, this time the MAE train and test are closer together as compared to the MSE (MAE test of 0.51 and MAE train of 0.65). Furthermore, similar MAE test and MAE val results (0.65 for test and 0.57 for val) demonstrate consistent generalization performance
   - Again, R2 shows repetitive performance across the three models. Since the results are similar, at around 0.01, this means that the model only explains a small portion of the variance in Responder_6. R2 measures how well the model explains the variance in the target variable. It ranges from -infinity to 1, where 1 means that the model explains all the variance in the target and 0 means the model explains none of the variance. Less than 0 means the model is worse than guessing. Although the R2 is similar for train, test, and val it does slightly decrease in test and val. This could mean that the dataset is not suitable for these linear regression models. It suggests that the dataframe is either inherently noisy or lacks strong linear relationships between the features and responder_6.

2. Decision Tree

   - This model had slightly worse performance than the ridge, lasso, and elastic models. The MSE train, test, and val results where almost exactly the same but slightly higher (0.66 compared with 0.65, 0.98 compared with 0.97, and 0.78 compared with 0.77). Again, the trend between the train and test suggests the model is overfitting. This trend is also present for MAE.
   - The R2 values are all significantly lower than the previous models. Even R2 val is 0.0032. This means that the model essentially does not explain any of the variance in responder_6. In addition, to the poor results, the model also took over 38 seconds to compile. Thus this model is not worth to use.

3. Linear Regression

   - The results are pretty much exactly the same as those for ridge regression surprisingly. However, Ridge regression does take less to compile and has slighly slower values (by 0.001 for example). Thus, ridge regression slightly edges out linear regression.

4. XGBoost

   - This model is the by far the model which overfits the most. The the MSE train is the lowest at 0.55 but then has a MSE test of 1.05 which is the highest.
   - The MAE however is similar to the previous models.

- The R2 values for the test and val data are all below 0. This means that Xboost is worse than using the mean of responder_6 as a constant prediction for every input. This is again a sign that the relationship between features and responder_6 might be non-linear.

5. Overall

- Due to the poor r2 results for all the models, it can be conlcluded that these regression models are not perfectly suitable for this dataset.
- It is important to note that polynomial regression was attempted on the dataset but was too computationally expensive for kaggle to run (even for a 2nd order polynomial). We suspect however that the data does not have a linear relationship and must be model using higher order polynomials or other types of models.
- **However, the units of MSE are the square of the target's (responder_6) units. So a val mse of 0.77 (in the best case) equates to a RMSE of 0.877. This means that on average these regression models deviate by about 0.877 responder_6 units. In the data-preprocessing section we can see that the range of responder_6 is -5 to 5 with the mean at 0. Thus, this represents 8.77% of responder_6's range. However since we, again, don't know what responder 6 is, we don't know if this is good or bad. Here, "good performance" depends on the context of the problem. For financial modeling 8.66% could be a very large innaccuracy.**

Question From Presentation: Why is the test MSE higher than the validation MSE?

- Our temporal splitting is based on date_id, meaning that the splits are made chronologically
  - The train set covers the earliest date_id values
  - The Validation set covers the intermediate/middle date_id values
  - The test data consists of the latest/last date_id values
- Thus when the model is run with the validation data, this dataset is closer chronologically to the train set. This means that the distance in time is shorter and thus the model is predicting short into the future. With the test set data, however, this is the farthest distance in the future and thus harder to accurately predict. The model, most likely, has a hard time accurately predicting Responder_6 the farther away the input data is from the train set. To reiterate, since the validation set is closer in time to the train set, they likely share more similar patterns/trends, as compared to the test set and train set. This closer distance leads to better generalization by the model and lower MSE.
- This inability to properly generalize would indicate that the model overfit on the temporal data around training set. Though some of it worked well on the validation set, it did not work well on testing set that is too far temporally and required better generalization capability.

Neural Nets

In [34]:
```
import tensorflow as tf
from tensorflow.keras.callbacks import EarlyStopping

tf.keras.backend.clear_session()
tf.random.set_seed(42)
```

In [35]:
```
input_shape=train_x.shape[1:]
print(input_shape)
```

```
(50,)
```

From the regression results the main goal is to prevent overfitting our data between the train and test sets.

L2 Regularization:

- We added kernel_regularizer=tf.keras.regularizers.l2(0.001) to dense layers to penalize large weights and reduce overfitting.
- It prevents the model from assigning excessively large weights to certain features. Thus it should prevent overfitting by reducing the model's reliance on any single feature. This shold create a simpler model that s better on unseen data (the validation set).
- This should help model performance on validation set.

Dropout:

- Dropout is a regularization technique that chooses a random subset of neurons and temporarily "drops" it (set to zero) during each training batch. This prevents co-dependency among neurons, which forces the network of nuerons to learn its own reduntant/overfitting behavior.
- We added tf.keras.layers.Dropout(0.3) after each activation layer to randomly disable 30% of neurons during training.
- This also helps prevent overfitting.

Reduced Learning Rate:

- We set the learning_rate = 0.001 in the Adam optimizer for smaller weight updates during training.
- A lower learning rate takes longer but reduces the risk of the model overshooting the optimal neuron weights.

Early Stopping:

- We added EarlyStopping to halt training if validation loss does not improve for 5 consecutive epochs. Here the loss function is the validation set MSE. By added EarlingStopping this should prevent overfitting and also make a faster runnning model.

In [36]:
```
regressors['Neural Network'] = tf.keras.Sequential([

    tf.keras.layers.Flatten(input_shape=input_shape),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal",kernel_regularizer=tf.keras.regularizers.l2(0.
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
     # add a drop out layer to prevent overfitting
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(100, kernel_initializer="he_normal",kernel_regularizer=tf.keras.regularizers.l2(0.
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),


    tf.keras.layers.Dropout(0.3),

    # Single output neuron for regression. We want to predict just Responder_6
    tf.keras.layers.Dense(1, activation=None)

])



# tf.keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True)
```

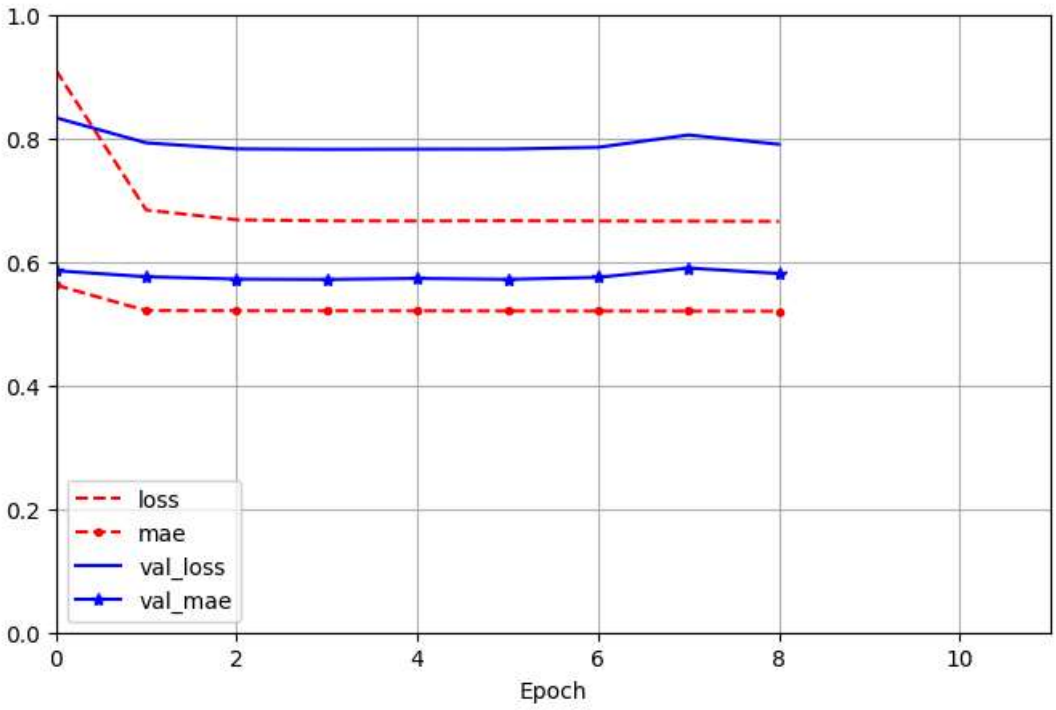
```
/opt/conda/lib/python3.10/site-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [38]:
```
# Early stopping callback --> stops model from running more epochs without improvement
# It monitors validation loss and stops training after 5 epochs of no improvement
# Restores model weights from the best epoch
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)



regressors['Neural Network'].compile(loss="mse",
              # reduced learning rate
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              metrics=["mae"])

history = regressors['Neural Network'].fit(train_x.values.astype('float32'),
                   train_y.values.astype('float32'),
```

```
                    batch_size = 1024,
                    validation_data = (val_x.values.astype('float32'), val_y.values.astype('float32')),
                    epochs=30,
                    callbacks=[early_stopping]
                   )
```

```
Epoch 1/30
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1733372444.036008      96 service.cc:145] XLA service 0x7ab55c006040 initialized for platform CUD
A (this does not guarantee that XLA will be used). Devices:
I0000 00:00:1733372444.036050      96 service.cc:153]   StreamExecutor device (0): Tesla P100-PCIE-16GB, Comp
ute Capability 6.0
  92/1131 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 2.0496 - mae: 0.8355
I0000 00:00:1733372447.208810      96 device_compiler.h:188] Compiled cluster using XLA!  This line is logged
at most once for the lifetime of the process.
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - loss: 1.2032 - mae: 0.6287 - val_loss: 0.8331 - val_mae: 0.5856
Epoch 2/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6928 - mae: 0.5211 - val_loss: 0.7925 - val_mae: 0.5758
Epoch 3/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6687 - mae: 0.5206 - val_loss: 0.7827 - val_mae: 0.5719
Epoch 4/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6660 - mae: 0.5205 - val_loss: 0.7820 - val_mae: 0.5714
Epoch 5/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6656 - mae: 0.5203 - val_loss: 0.7823 - val_mae: 0.5730
Epoch 6/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6662 - mae: 0.5203 - val_loss: 0.7825 - val_mae: 0.5715
Epoch 7/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6656 - mae: 0.5201 - val_loss: 0.7853 - val_mae: 0.5748
Epoch 8/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6652 - mae: 0.5200 - val_loss: 0.8052 - val_mae: 0.5898
Epoch 9/30
1131/1131 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6652 - mae: 0.5200 - val_loss: 0.7901 - val_mae: 0.5810
```

In [39]:
```
pd.DataFrame(history.history).plot(

    figsize=(8, 5), xlim=[0, 11], ylim=[0, 1], grid=True, xlabel="Epoch",

    style=["r--", "r--.", "b-", "b-*"])

plt.legend(loc="lower left")  # extra code

plt.show()
```



Results

1. We can see that the EarlyStopping function stopped the model training after 9 epochs.
2. Training Metrics:
   - Loss: 0.66
   - MAE: 0.52
   - The Mean Absolute Error is 0.52, which represents the average absolute difference between the predicted and actual values in the training set. Since the target range is this error corresponds to 5.19% of the range on average.
3. Validation Metrics:
   - val_loss: 0.79
   - val_mae: 0.58
4. Overall
   - Since the validation loss is higher than the training loss, this means that model performs slightly worse on the unseen validation set. Again however, we don't know if this signifies a lot of overfitting or slight overfitting. We don't know the units or context of responder_6.
   - The best results of the normal regression models was a train MSE of 0.65 and a validation MSE of 0.77. The results of the neural net was a train MSE of 0.66 and a val MSE of 0.79. This is essentially the same output as the normal regression models.
   - For MAE, the best results of the normal regression models was a train MAE of 0.49 and a validation MAE of 0.59. For the neural net it was 0.52 and 0.58 respectively. This is a smaller gap and smaller MAE validation value too. Thus the neural net performed better than the regression models for MAE.
   - Since the results are almost the same as the normal regressino models, we would recommend to just use elastic, ridge, or lasso regression. If the results are the same, normal regression models don't have the same computational expense as nueral nets do. Plus they are faster too!

In [40]:
```
test_x = test_df[ALL_FEATURES]

test_y = test_df[['responder_6']]
```

In [41]:
```
regressors['Neural Network'].evaluate(test_x.values.astype('float32'),
                                      test_y.values.astype('float32'),
                                      batch_size = 1024)
```

```
389/389 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.0281 - mae: 0.6688
```

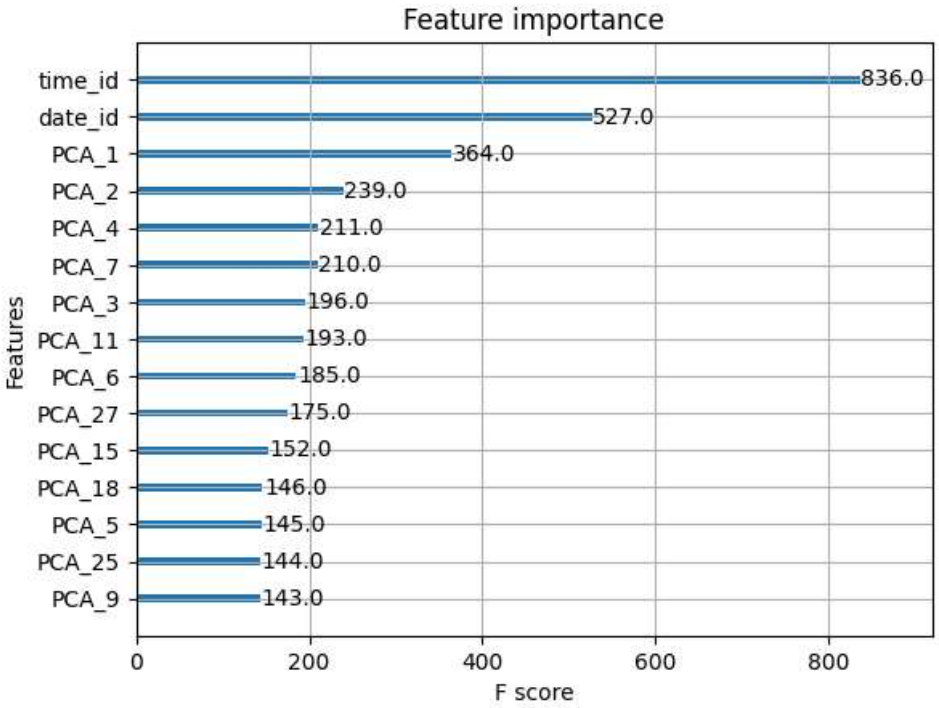
Out[41]:
```
[0.9834357500076294, 0.6580089926719666]
```

Feature Importance

In [42]:
```
from xgboost import plot_importance
plt.figure(figsize=(10, 6))

plot_importance(regressors["XGBoost"], max_num_features=15)
```

Out[42]:
```
<Axes: title={'center': 'Feature importance'}, xlabel='F score', ylabel='Features'>
```

```
<Figure size 1000x600 with 0 Axes>
```

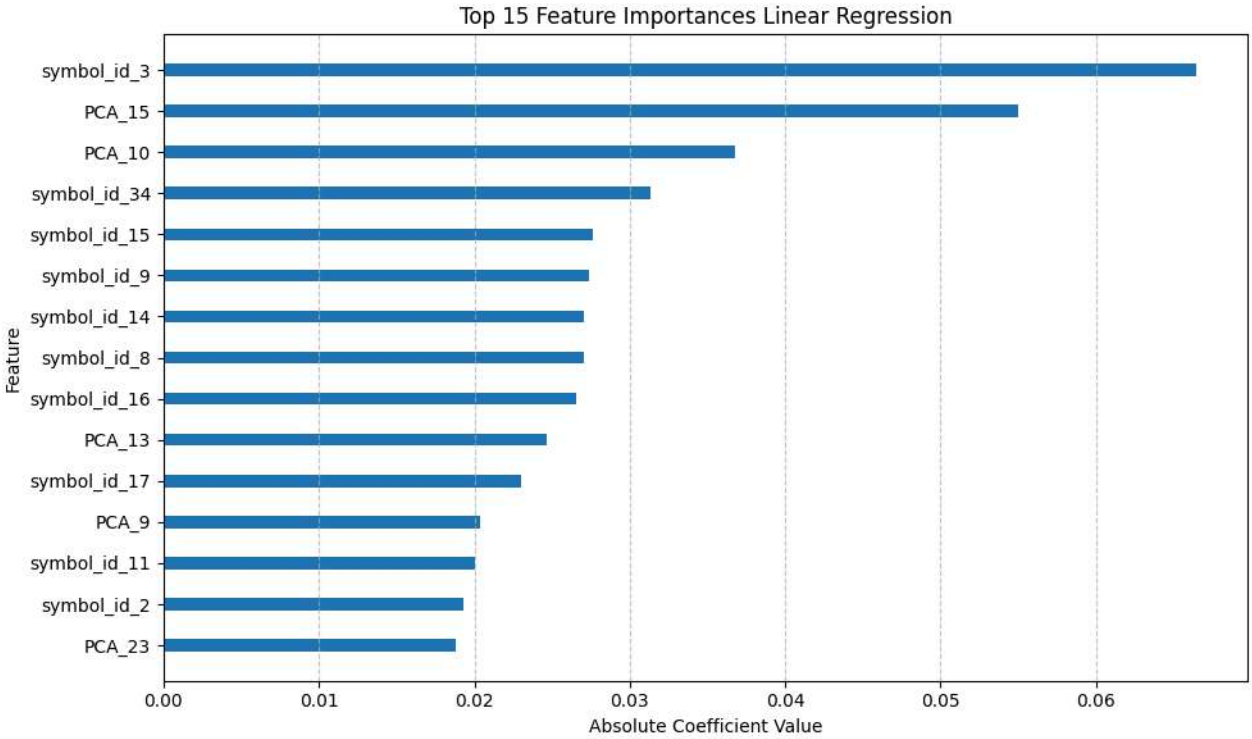


From the plots, it can be observed that for XGBoost, `time_id` and `date_id` is important. One explanation for this might be that the temporal information is an useful indicator of the stock market.

Other than that, XGBoost put emphasis on the `PCA_1`, `PCA_2`, `PCA_7` features.

In [43]:
```
def get_top_features_plot(model, feature_names, n=15, model_name = None):
    # Get coefficients and their absolute values
    if 'Regression' in model_name:
        coefs = model.coef_[0]

    else:
        coefs = model.feature_importances_

    abs_coefs = np.abs(coefs)
    # Create a DataFrame for better visualization

    coef_df = pd.DataFrame({"Feature": feature_names, "Coefficient": coefs, "Abs Coefficient": abs_coefs})

    # Sort by absolute value of coefficients
    top_features = coef_df.nlargest(n, "Abs Coefficient")
    #print(top_features)

    plt.figure(figsize=(10, 6))
    y_pos = np.arange(len(top_features))

    plt.barh(y_pos, top_features["Abs Coefficient"], height = 0.3)

    plt.xlabel("Absolute Coefficient Value")
    plt.ylabel("Feature")
    plt.yticks(y_pos, top_features["Feature"])
    plt.title(f"Top {n} Feature Importances {model_name}")

    plt.grid(axis="x", linestyle="--", alpha=0.7)
    plt.tight_layout()
    plt.gca().invert_yaxis()  # Ensure the highest value is at the top
    plt.show()
```

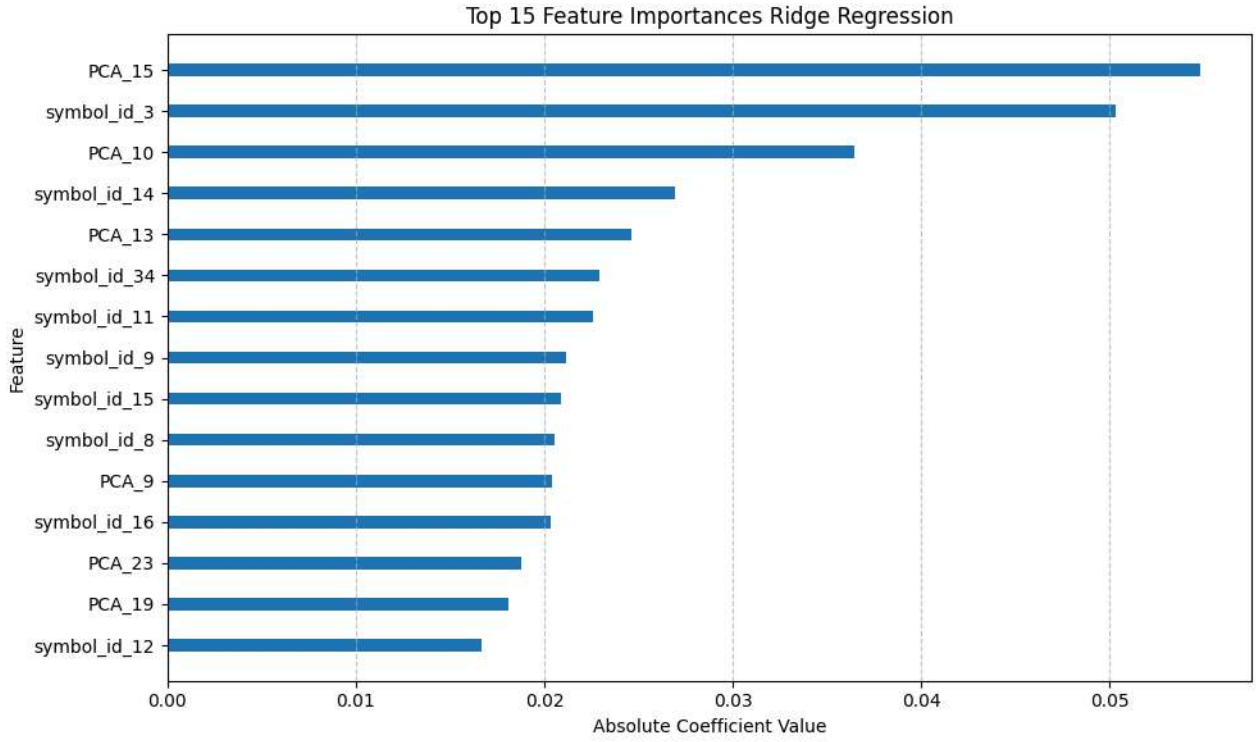
In [44]:
```
model_name = 'Linear Regression'
get_top_features_plot(regressors[model_name], ALL_FEATURES, model_name = model_name)
```



The simple Linear Regression model put emphasis on some `symbol_id` and `PCA` features. The symbol ids are one-hot encoded, so this would mean that the regression model is attaching specific weights to each symbol_ids, which is similar to a bias term corresponding to prediction for a symbol_id.

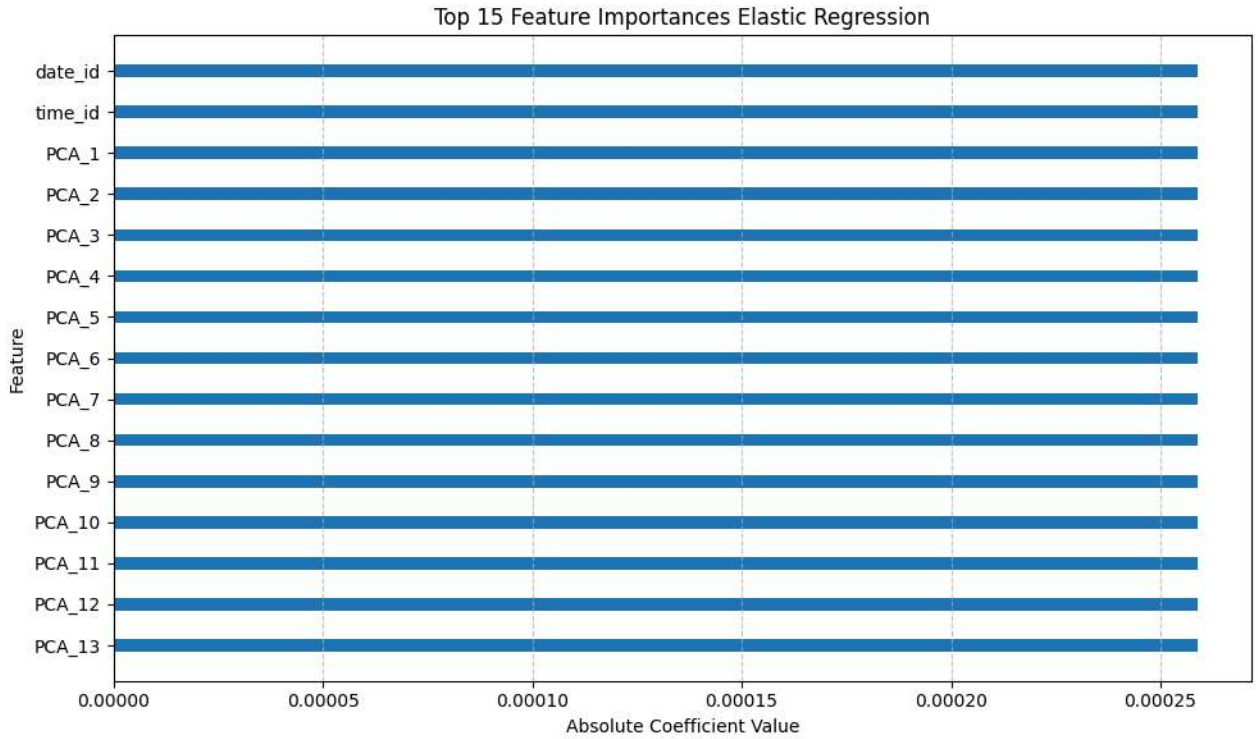
Ignoring the symbol_ids, the Linear Regression model is putting emphasis on `PCA` features different from the XGBoost model

In [45]:
```
model_name = 'Ridge Regression'
get_top_features_plot(regressors[model_name], ALL_FEATURES, model_name = model_name)
```



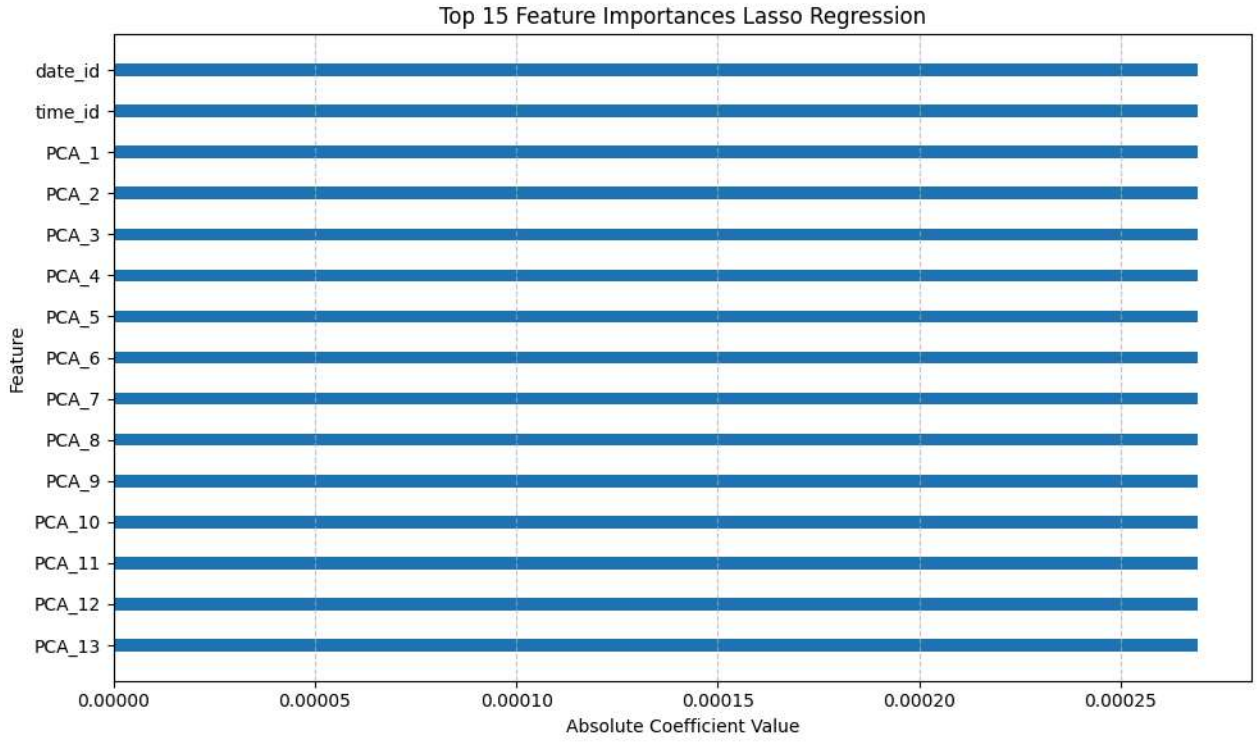
The Ridge Regression model is putting emphasis on `PCA_15` and `PCA_10` features, similar to the Linear Regression model.

In [46]:
```
model_name = 'Elastic Regression'
get_top_features_plot(regressors[model_name], ALL_FEATURES, model_name = model_name)
```

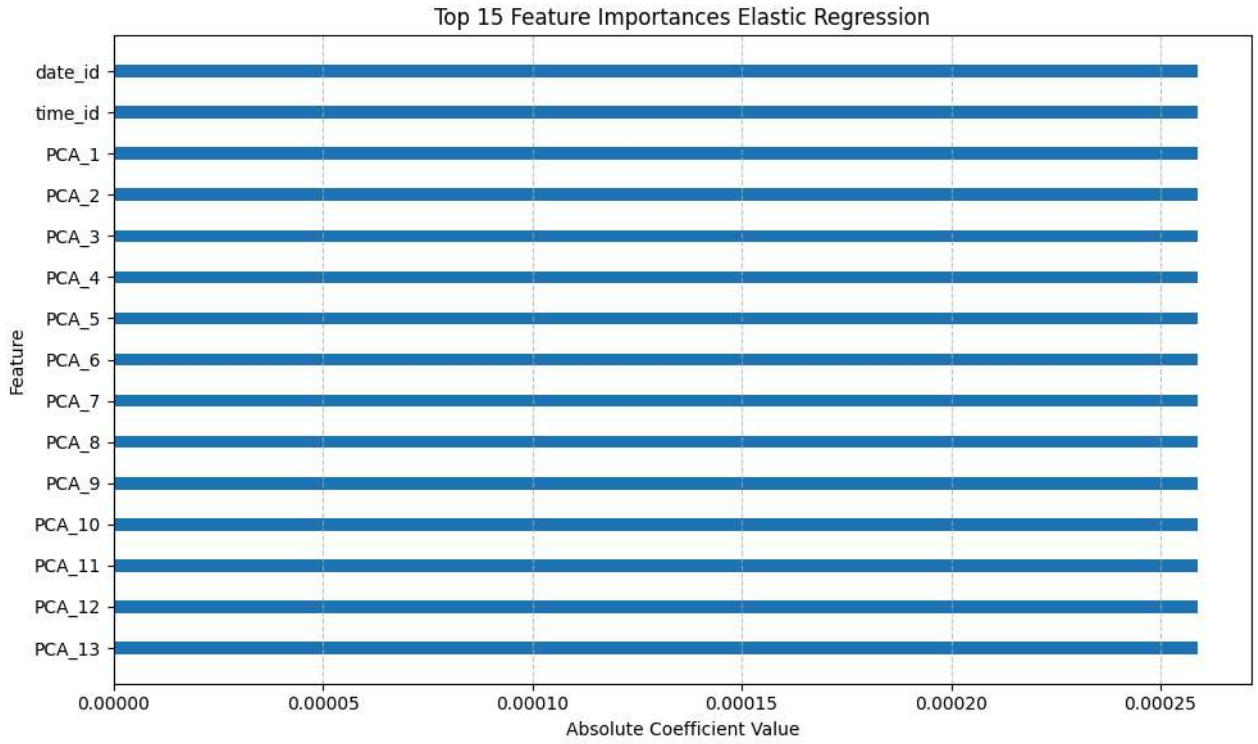


The regularization term is making the weights of all the features as small as possible, which would explain the relative lack of difference between the features

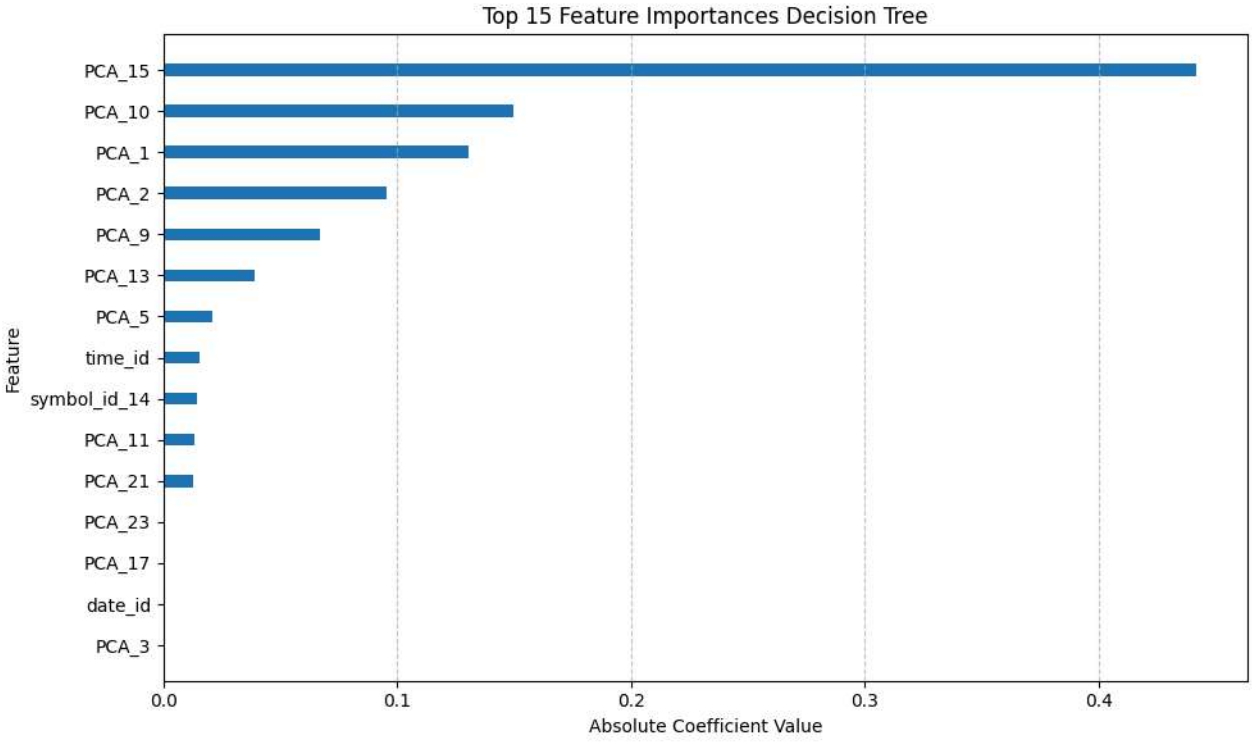
In [47]:
```
model_name = 'Lasso Regression'
get_top_features_plot(regressors[model_name], ALL_FEATURES, model_name = model_name)
```



In [48]:
```
model_name = 'Elastic Regression'
get_top_features_plot(regressors[model_name], ALL_FEATURES, model_name = model_name)
```



In [49]:
```
model_name = 'Decision Tree'
get_top_features_plot(regressors[model_name], ALL_FEATURES, model_name = model_name)
```



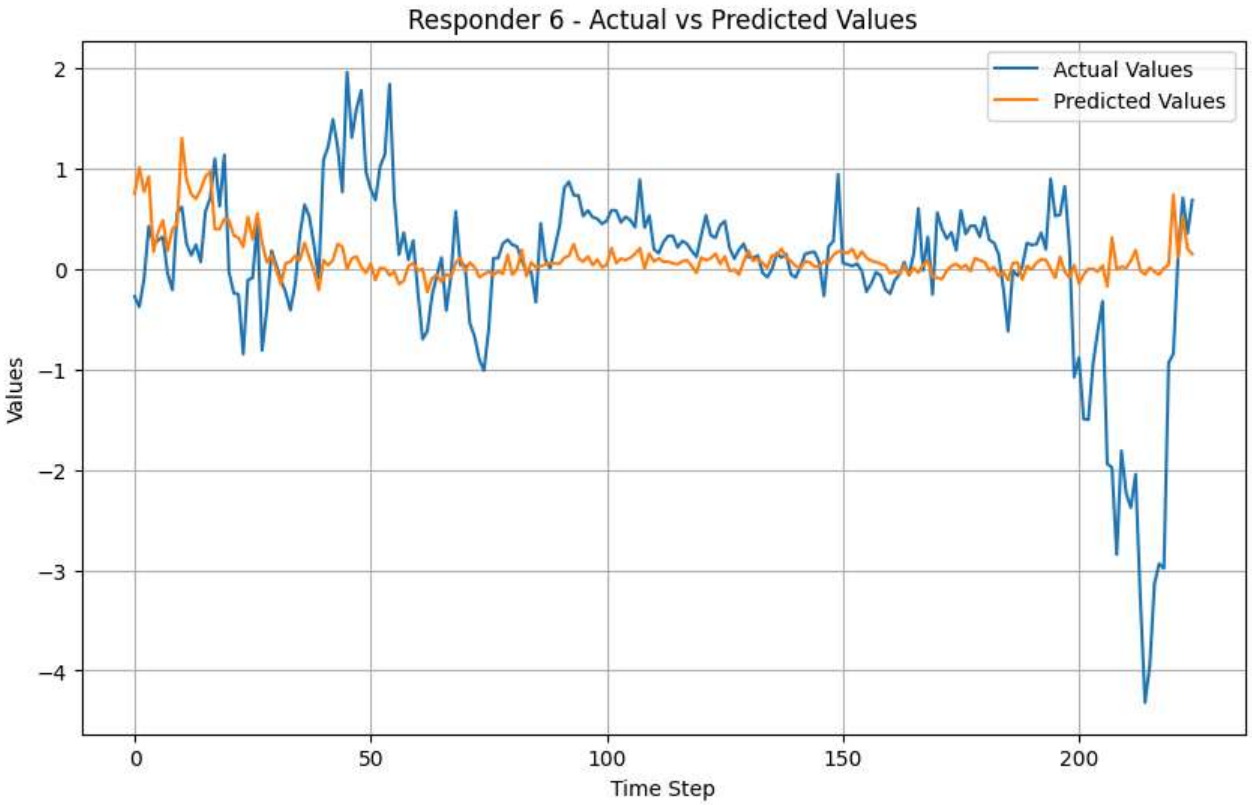
It is interesting to note that the Decision Tree also put emphasis on `PCA_15` and `PCA_10` features

Plotting on a few samples of the test_data

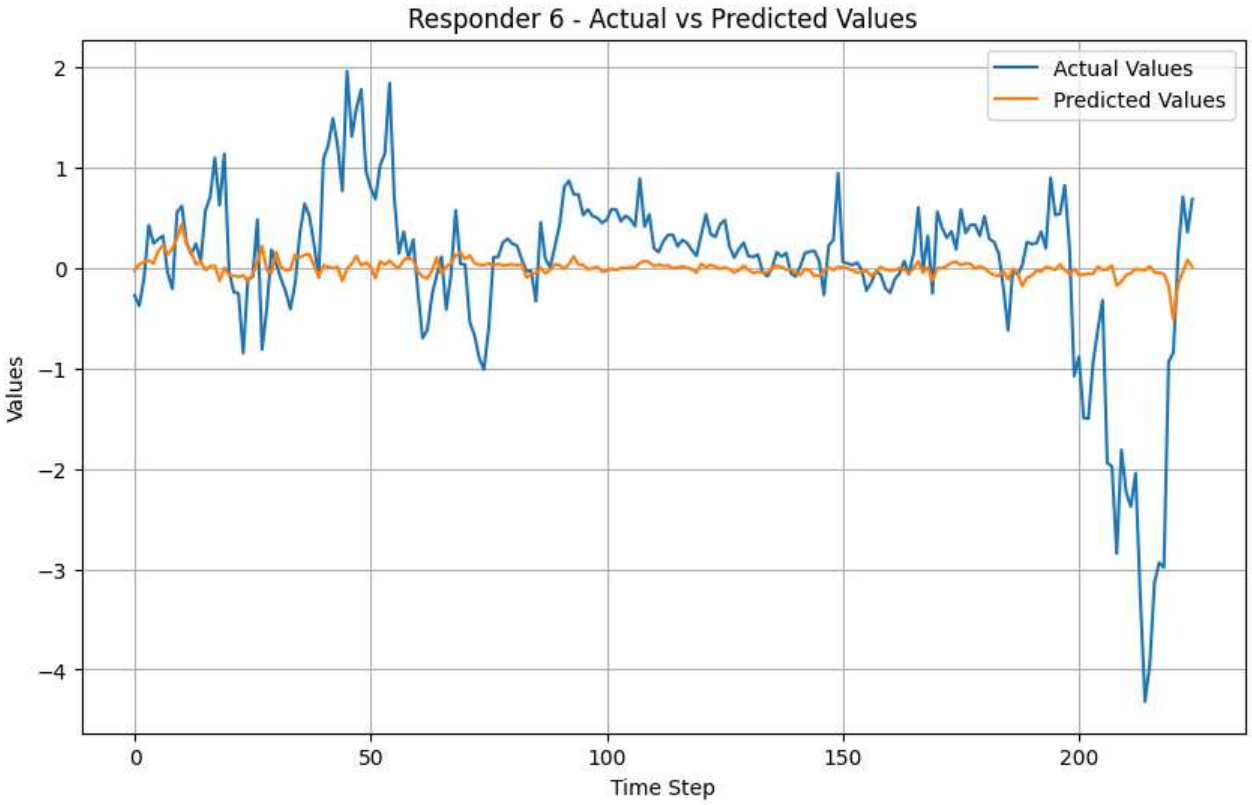
In [50]:
```
s_0 = test_df[test_df.symbol_id_0 == 1]
print(len(s_0))
sample_df = s_0.iloc[:225]
print(len(sample_df))
sample_data = sample_df[ALL_FEATURES]
y_s = sample_df['responder_6']
```

```
22074
225
```

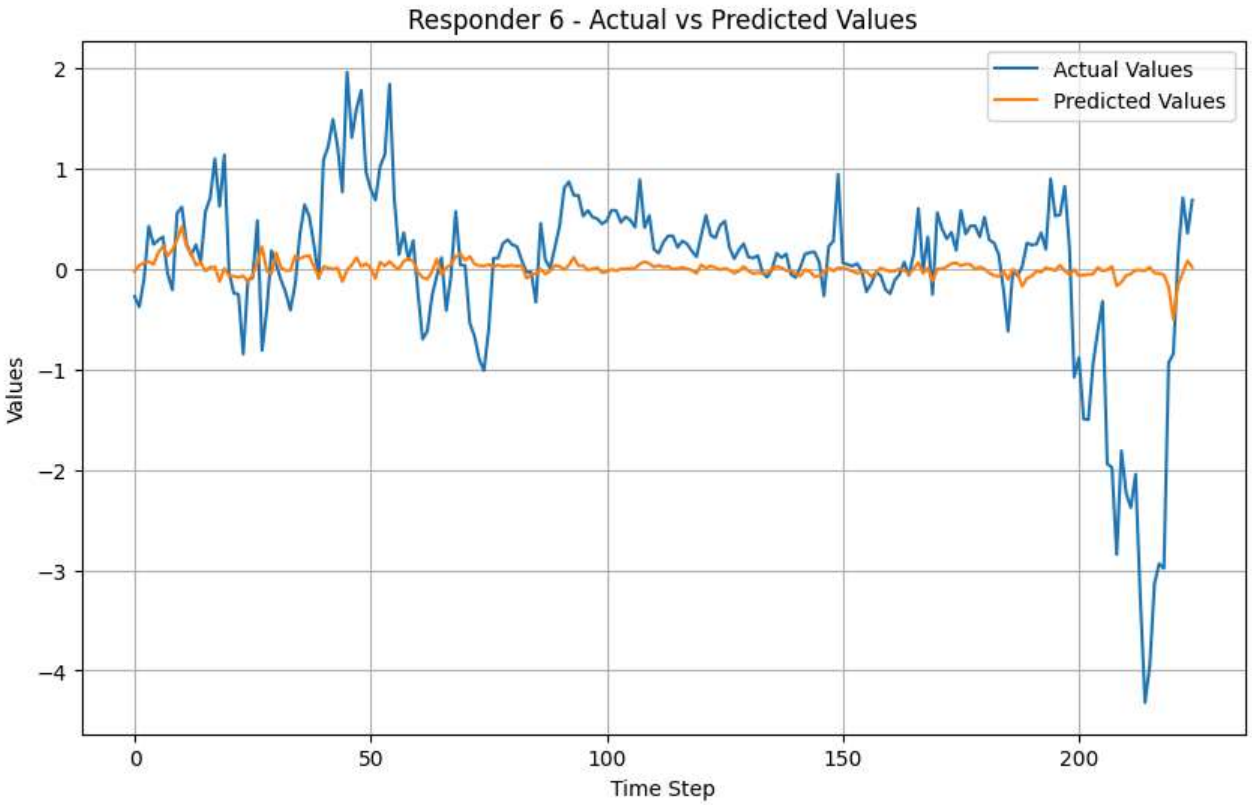
In [51]:
```
import matplotlib.pyplot as plt

# Get the data for responder_6 in the test set
responder_6_predictions = regressors['XGBoost'].predict(sample_data)
responder_6_actual_values = y_s
# Plot the actual vs predicted values
def plot(responder_6_predictions):
    plt.figure(figsize=(10, 6))  # Adjust figure size as needed
    plt.plot(range(len(responder_6_actual_values)), responder_6_actual_values, label='Actual Values')
    plt.plot(range(len(responder_6_predictions.squeeze())), responder_6_predictions.squeeze(), label='Predic
    plt.xlabel('Time Step')
    plt.ylabel('Values')
    plt.title(f'Responder 6 - Actual vs Predicted Values')
    plt.legend()
    plt.grid(True)
    plt.show()
plot(responder_6_predictions)
```



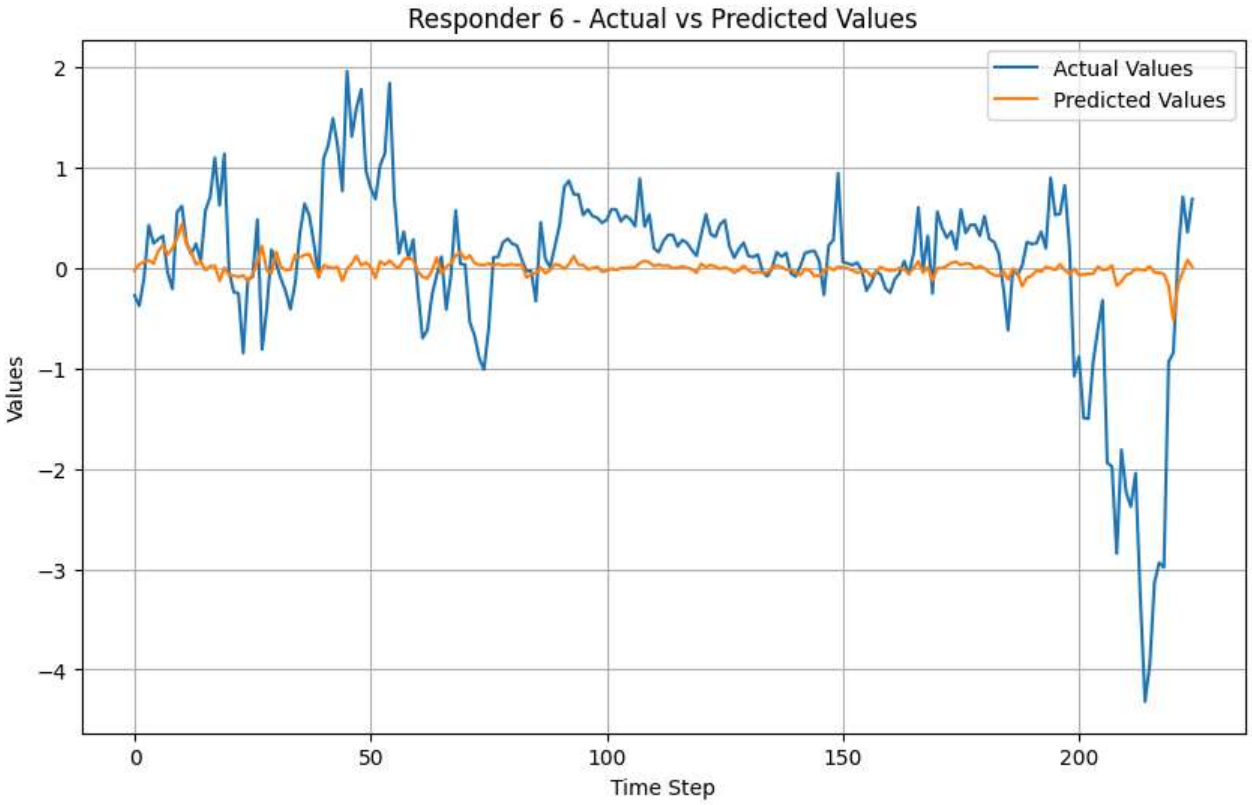
In [52]:
```
responder_6_predictions = regressors['Ridge Regression'].predict(sample_data)
plot(responder_6_predictions)
```



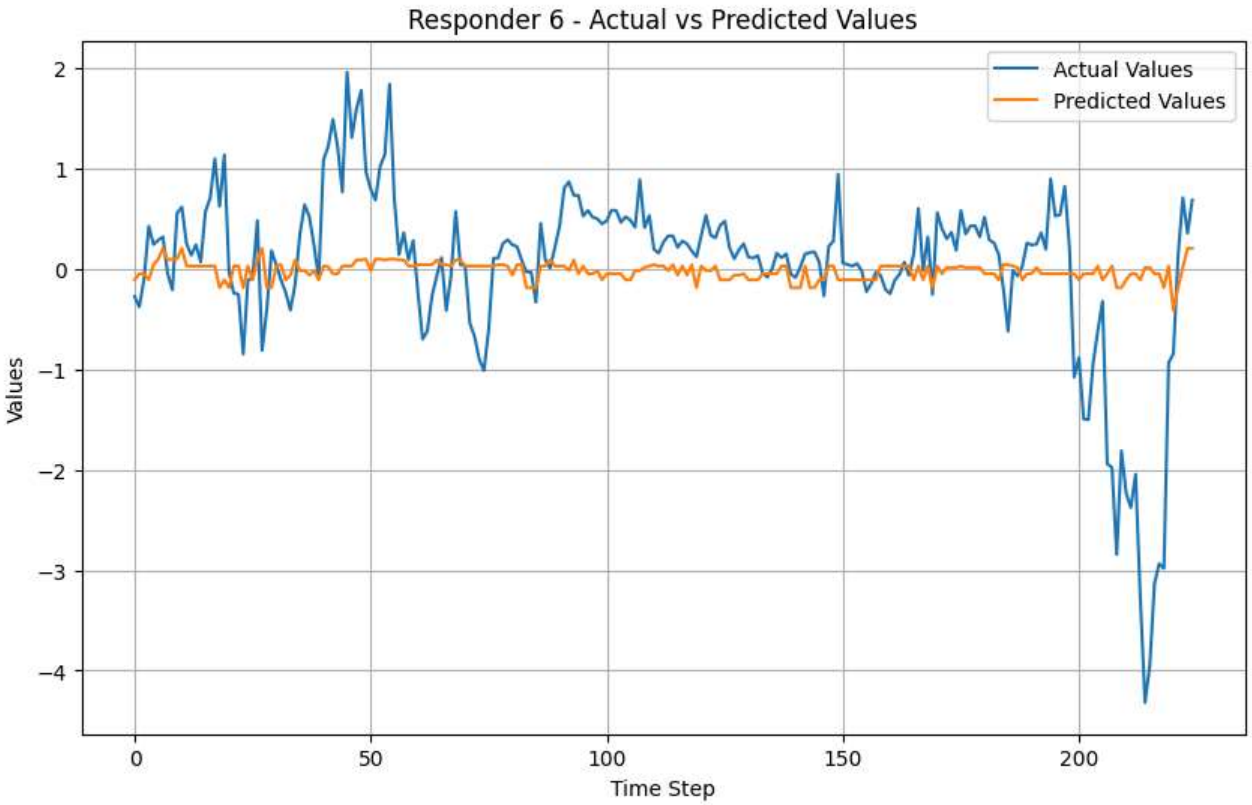
In [53]:
```
responder_6_predictions = regressors['Lasso Regression'].predict(sample_data)
plot(responder_6_predictions)
```



In [54]:
```
responder_6_predictions = regressors['Linear Regression'].predict(sample_data)
plot(responder_6_predictions)
```



In [55]:
```
responder_6_predictions = regressors['Decision Tree'].predict(sample_data)
plot(responder_6_predictions)
```



In [57]:
```
responder_6_predictions = regressors['Neural Network'].predict(sample_data.astype('float'))
plot(responder_6_predictions)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step
```