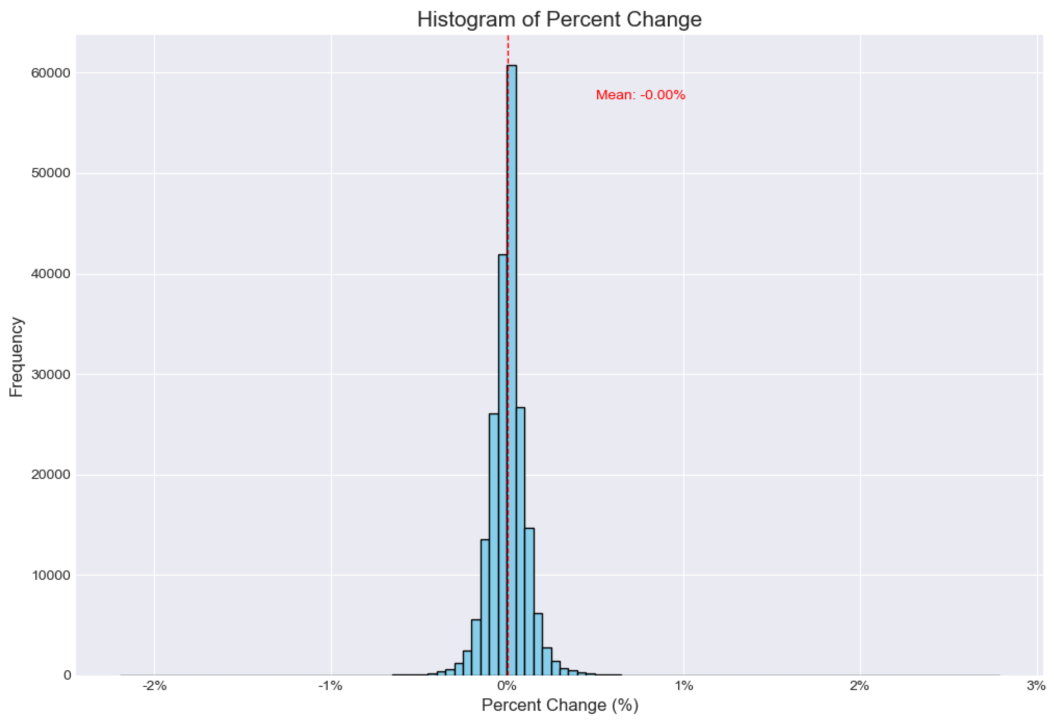
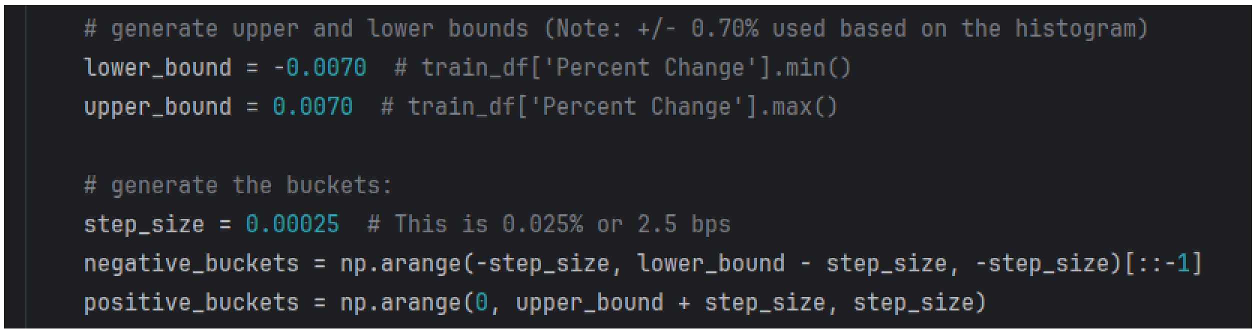


Here is the distribution of minute percent changes in the training set (first 80% of data).

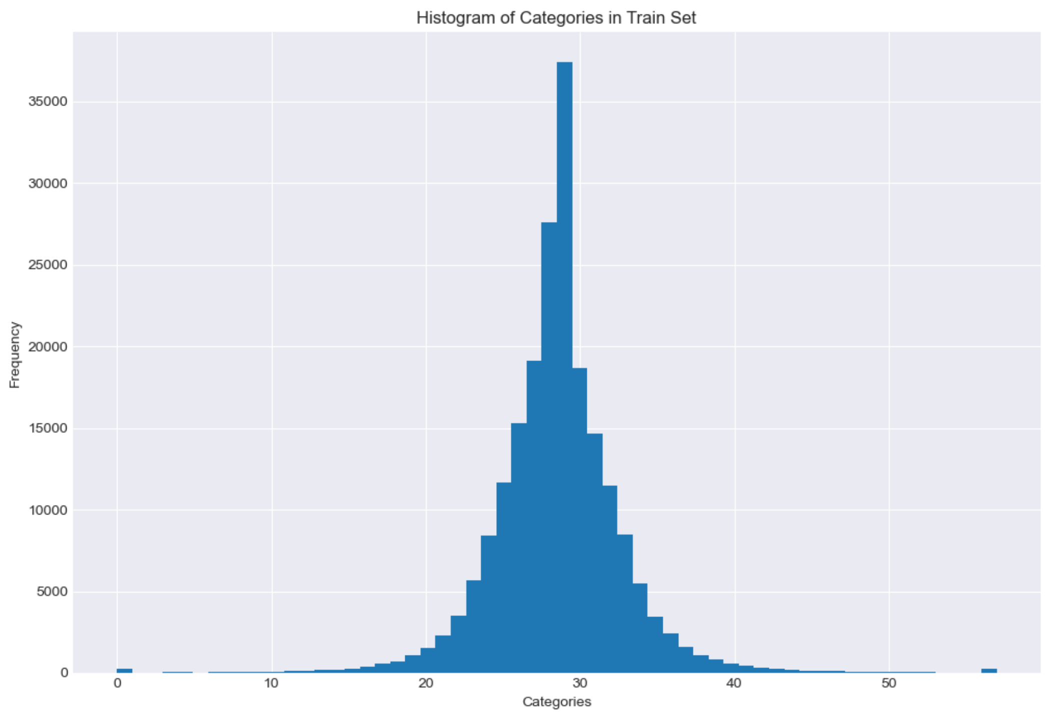
I decided to use +/- 0.7% for the min and max bucket ranges with a step size of 2.5bps, so basically it will create even buckets to categorize each percent change with a break at 0, so that we can distinguish between positive and negative returns.

Anything less than -0.70% is put at category 0 and anything greater than 0.70% is put into category 58.



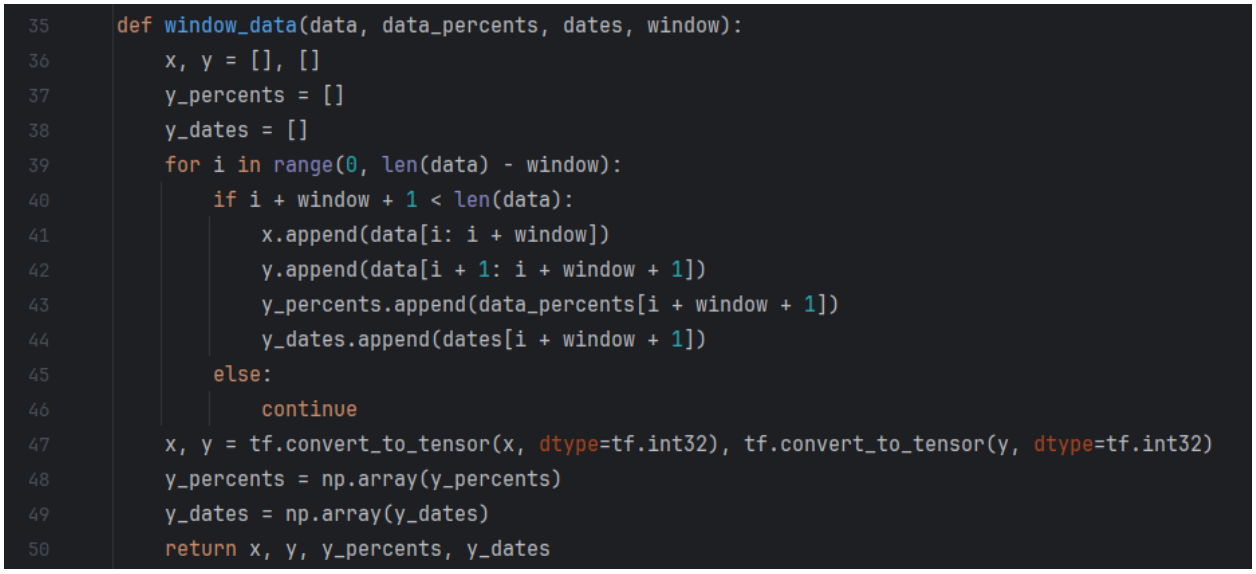
```
# generate upper and lower bounds (Note: +/- 0.70% used based on the histogram)
lower_bound = -0.0070  # train_df['Percent Change'].min()
upper_bound = 0.0070  # train_df['Percent Change'].max()

# generate the buckets:
step_size = 0.00025  # This is 0.025% or 2.5 bps
negative_buckets = np.arange(-step_size, lower_bound - step_size, -step_size)[::-1]
positive_buckets = np.arange(0, upper_bound + step_size, step_size)
```

This turns percent changes into categories (tokenization) that can later have their own associated embeddings as in language modeling, while maintaining the distribution.



Preprocessing:

Data is windowed so that each x is a length of 10 categories and then the y is the categories shifted right into the future where the last category of y is what we are trying to predict (new data):

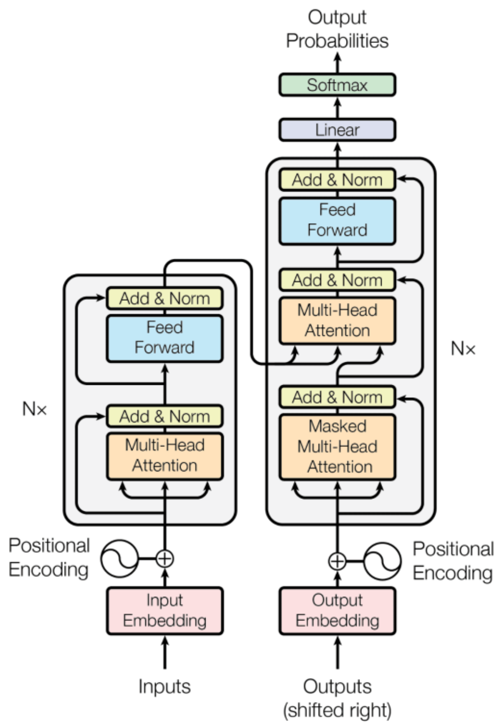


```
def window_data(data, data_percents, dates, window):
    x, y = [], []
    y_percents = []
    y_dates = []
    for i in range(0, len(data) - window):
        if i + window + 1 < len(data):
            x.append(data[i: i + window])
            y.append(data[i + 1: i + window + 1])
            y_percents.append(data_percents[i + window + 1])
            y_dates.append(dates[i + window + 1])
        else:
            continue
    x, y = tf.convert_to_tensor(x, dtype=tf.int32), tf.convert_to_tensor(y, dtype=tf.int32)
    y_percents = np.array(y_percents)
    y_dates = np.array(y_dates)
    return x, y, y_percents, y_dates
```

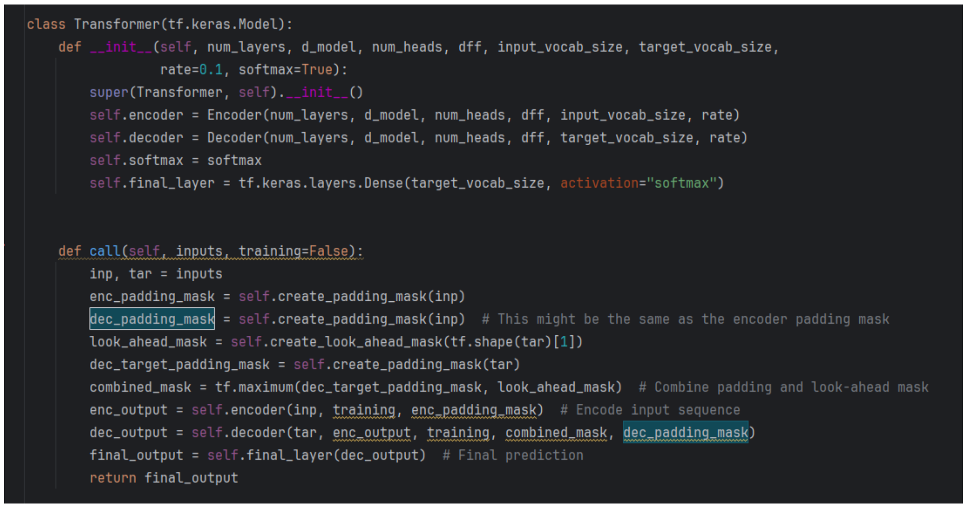
During Training, we feed the x and the y basically using teacher forcing with a look ahead mask.

We use positional encoding along with input embedding to keep track of the position of the categories and the meaning behind the categories and their sequences.

The x is fed through the decoder with positional encoding and input embedding. The y (shifted with output) is fed through the decoder with positional encoding, input embedding and a look ahead mask.



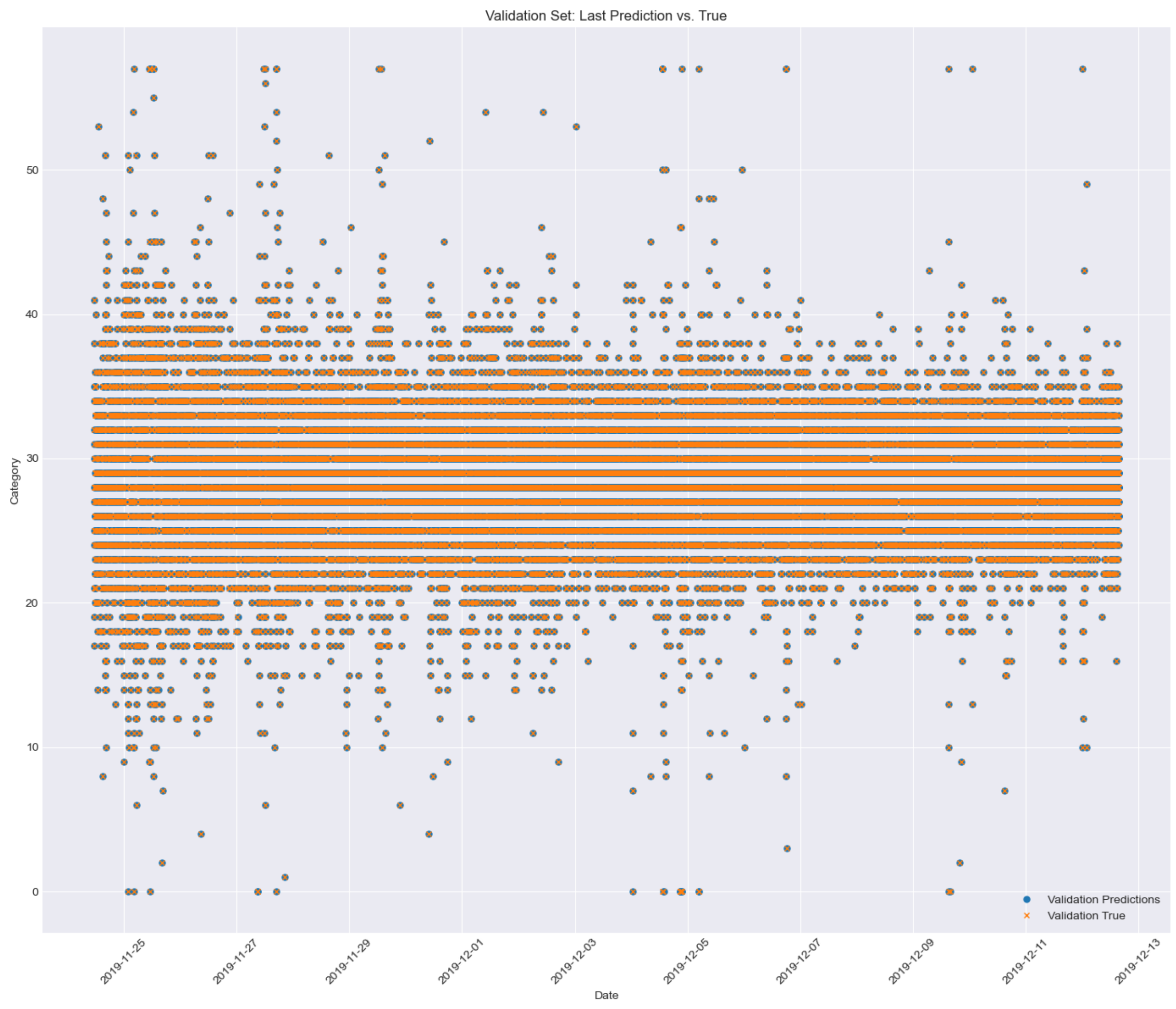
Original Transformer Architecture



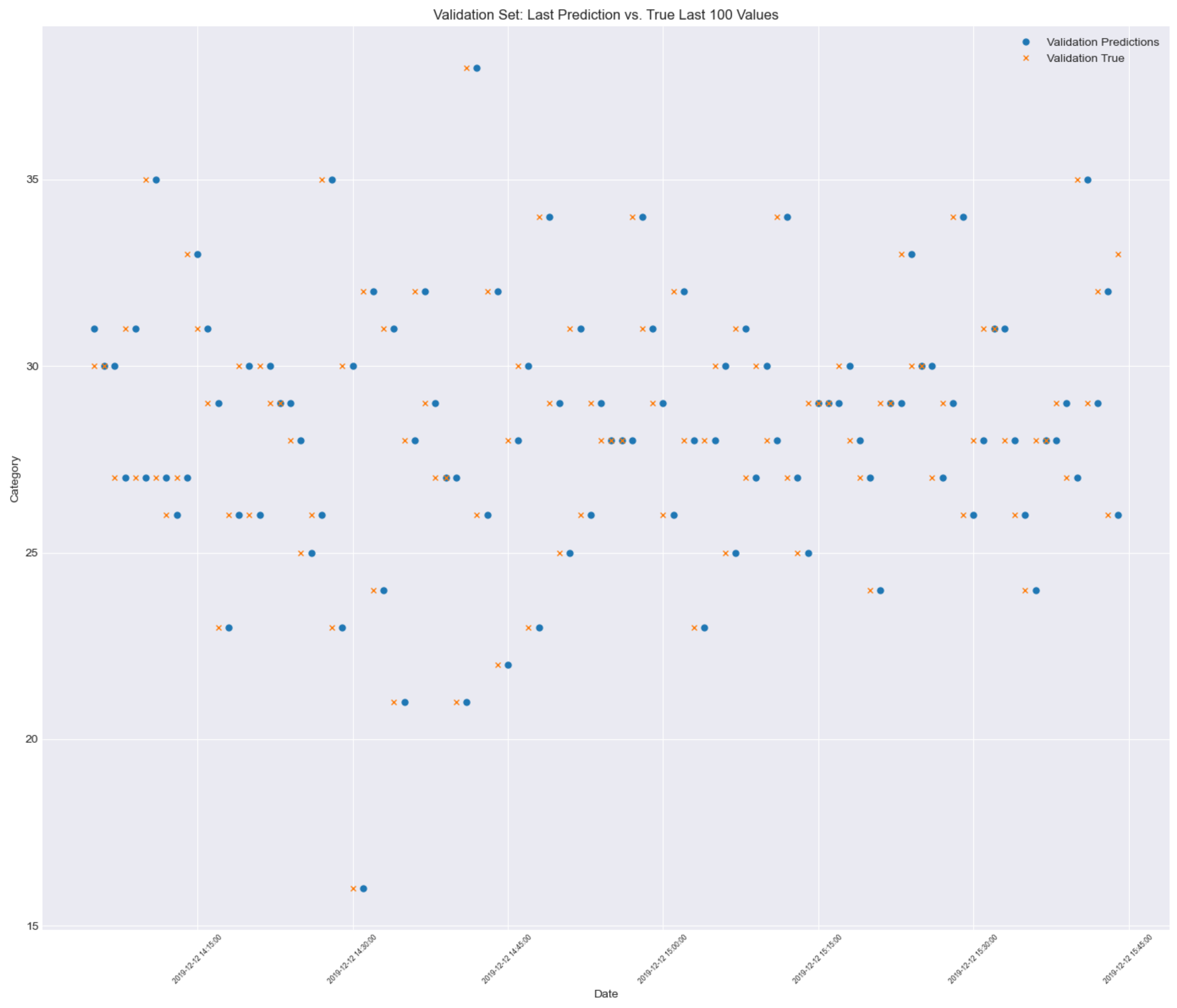
```
class Transformer(tf.keras.Model):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size, target_vocab_size,
                 rate=0.1, softmax=True):
        super(Transformer, self).__init__()
        self.encoder = Encoder(num_layers, d_model, num_heads, dff, input_vocab_size, rate)
        self.decoder = Decoder(num_layers, d_model, num_heads, dff, target_vocab_size, rate)
        self.softmax = softmax
        self.final_layer = tf.keras.layers.Dense(target_vocab_size, activation="softmax")


    def call(self, inputs, training=False):
        inp, tar = inputs
        enc_padding_mask = self.create_padding_mask(inp)
        dec_padding_mask = self.create_padding_mask(inp)  # This might be the same as the encoder padding mask
        look_ahead_mask = self.create_look_ahead_mask(tf.shape(tar)[1])
        dec_target_padding_mask = self.create_padding_mask(tar)
        combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)  # Combine padding and look-ahead mask
        enc_output = self.encoder(inp, training, enc_padding_mask)  # Encode input sequence
        dec_output = self.decoder(tar, enc_output, training, combined_mask, dec_padding_mask)
        final_output = self.final_layer(dec_output)  # Final prediction
        return final_output
```

Our Transformer (needs to be made into an graphic as above)

Results look almost perfect:

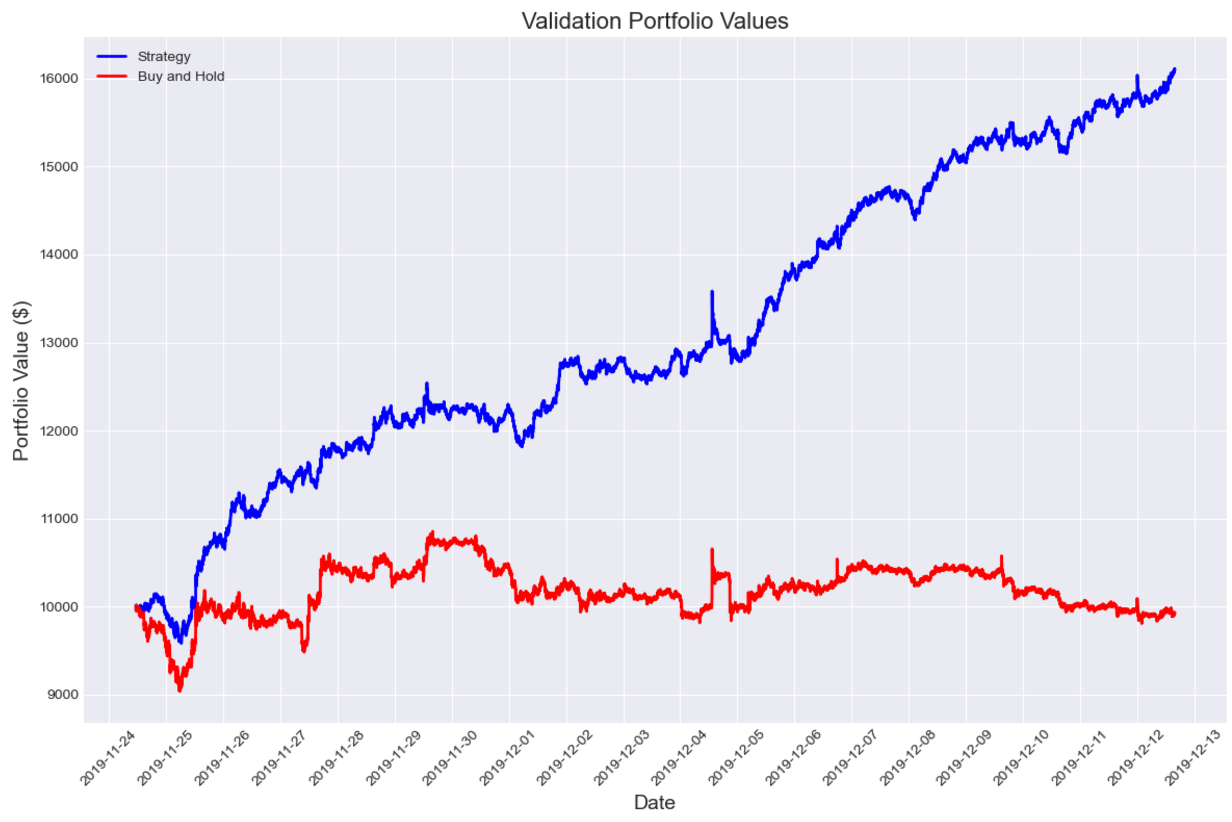


But, model just sort of ends up predicting previous category:



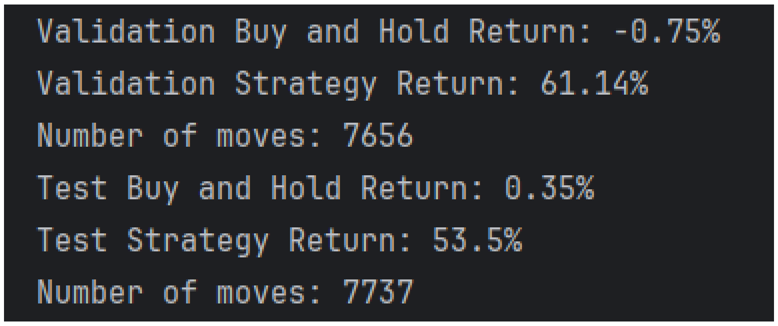
For creating a trading strategy, we buy when the category prediction is above or equal to the "category_at_zero" or the category where the positive buckets start.

However, this works very well for actually trading because we are following the trends of the market and hope to get lucky that previous negative minutes are indicative of future negative minutes when the market is trending (see the simulate_trading function in helper_functions.py)



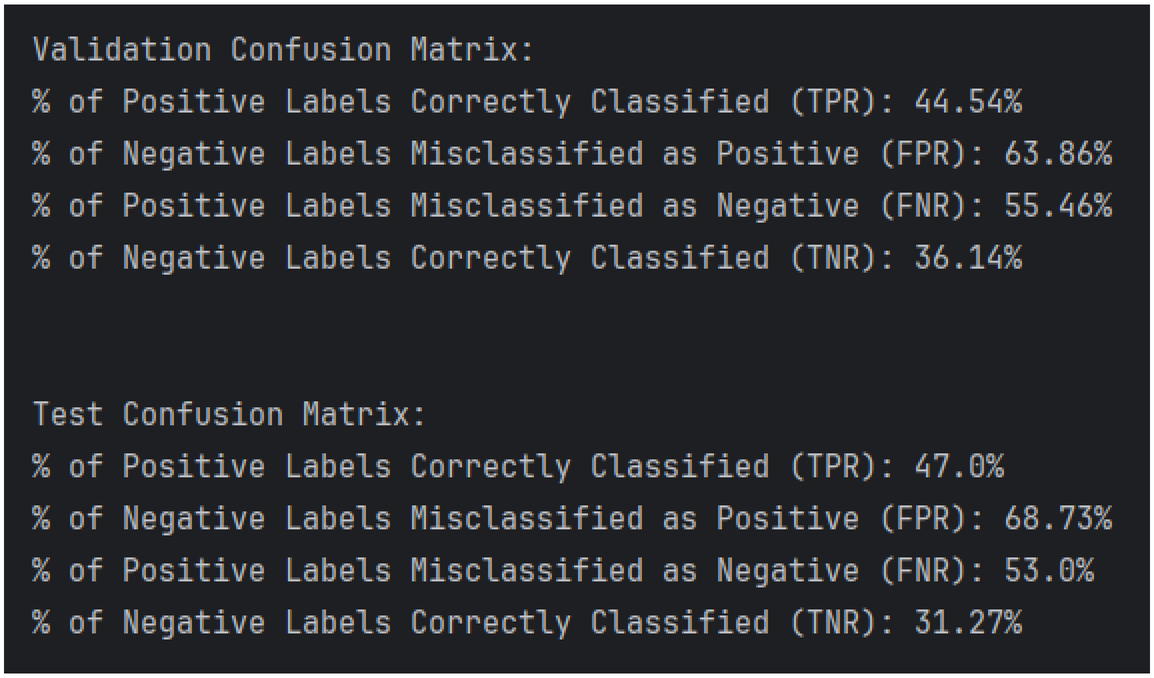
Results are similar for the Testing set.

Further Results: Did about 61% -> 10,000 to 16,000 over a month vs Bitcoin was flat.

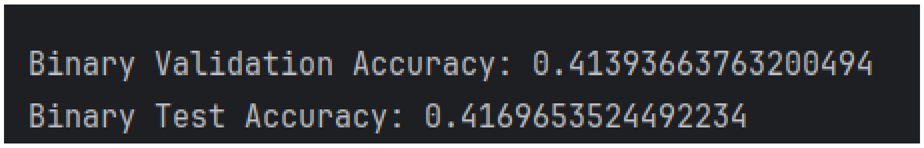


```
Validation Buy and Hold Return: -0.75%
Validation Strategy Return: 61.14%
Number of moves: 7656
Test Buy and Hold Return: 0.35%
Test Strategy Return: 53.5%
Number of moves: 7737
```

Confusion Matrix for Binary Classification:



```
Validation Confusion Matrix:
% of Positive Labels Correctly Classified (TPR): 44.54%
% of Negative Labels Misclassified as Positive (FPR): 63.86%
% of Positive Labels Misclassified as Negative (FNR): 55.46%
% of Negative Labels Correctly Classified (TNR): 36.14%


Test Confusion Matrix:
% of Positive Labels Correctly Classified (TPR): 47.0%
% of Negative Labels Misclassified as Positive (FPR): 68.73%
% of Positive Labels Misclassified as Negative (FNR): 53.0%
% of Negative Labels Correctly Classified (TNR): 31.27%
```



```
Binary Validation Accuracy: 0.41393663763200494
Binary Test Accuracy: 0.4169653524492234
```

Low accuracy, but just have to avoid big losses and follow trends to do well.