E-Commerce Sales Forecasting

We aim to analyze the sales data of a UK-based online retailer. Given the potential costliness of storage space and the crucial importance of timely delivery to outperform competitors, we seek to assist the retailer by predicting the daily quantities of products sold.

1. Prepare to start

In [81]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set()

from catboost import CatBoostRegressor, Pool, cv
from catboost import MetricVisualizer

from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

from scipy.stats import boxcox
from os import listdir

import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=RuntimeWarning)
warnings.filterwarnings("ignore", category=FutureWarning)

import shap
shap.initjs()
```

In [82]:
```
print(listdir("../input"))
```

```
['ecommerce-data', 'glove-global-vectors-for-word-representation']
```

In [83]:
```
data = pd.read_csv("../input/ecommerce-data/data.csv", encoding="ISO-8859-1", dtype={'
data.shape
```

Out[83]:
```
(541909, 8)
```

The dataset has 541909 entries and 8 variables.

2. The data

In [84]:
```
data.head()
```

Out[84]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/2010 8:26	2.55	17850	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/2010 8:26	2.75	17850	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/2010 8:26	3.39	17850	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/2010 8:26	3.39	17850	United Kingdom

We can see that the data file contains information for each individual transaction. Take a look at the InvoiceNo and the CustomerID of the first entries. Here, we observe that a single customer with the ID 17850 from the United Kingdom placed an order with the InvoiceNo 536365. The customer ordered several products with different stock codes, descriptions, unit prices, and quantities. Additionally, we notice that the InvoiceDate was the same for these products.

3. Data exploration

Missing values

What is the percentage of missing values for each feature?

In [85]:
```
missing_percentage = data.isnull().sum() / data.shape[0] * 100
missing_percentage
```

Out[85]:
```
InvoiceNo       0.000000
StockCode       0.000000
Description     0.268311
Quantity        0.000000
InvoiceDate     0.000000
UnitPrice       0.000000
CustomerID     24.926694
Country         0.000000
dtype: float64
```

Almost 25% of the customers are labeled as 'unknown'—quite an unusual figure. Additionally, there is a 0.2% rate of missing descriptions, indicating potentially incomplete or unclean data.

Missing descriptions

In [86]:
```
data[data.Description.isnull()].head()
```

Out[86]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
622	536414	22139	NaN	56	12/1/2010 11:52	0.0	NaN	United Kingdom
1970	536545	21134	NaN	1	12/1/2010 14:32	0.0	NaN	United Kingdom
1971	536546	22145	NaN	1	12/1/2010 14:33	0.0	NaN	United Kingdom
1972	536547	37509	NaN	1	12/1/2010 14:33	0.0	NaN	United Kingdom
1987	536549	85226A	NaN	1	12/1/2010 14:34	0.0	NaN	United Kingdom

How frequently do we encounter missing customer information?

In [87]:
```
data[data.Description.isnull()].CustomerID.isnull().value_counts()
```

Out[87]:
```
True    1454
Name: CustomerID, dtype: int64
```

What is the unit price?

In [88]:
```
data[data.Description.isnull()].UnitPrice.value_counts()
```

Out[88]:
```
0.0    1454
Name: UnitPrice, dtype: int64
```

In instances where descriptions are missing, we consistently lack both customer and unit price information. Why does the retailer record entries without adequate descriptions? It appears that there is no established protocol for managing and documenting such transactions. This lack of procedure serves as an indication that our data might contain irregular entries, making their detection and handling quite challenging.

Missing Customer IDs

In [89]:
```
data[data.CustomerID.isnull()].head()
```

Out[89]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
622	536414	22139	NaN	56	12/1/2010 11:52	0.00	NaN	United Kingdom
1443	536544	21773	DECORATIVE ROSE BATHROOM BOTTLE	1	12/1/2010 14:32	2.51	NaN	United Kingdom
1444	536544	21774	DECORATIVE CATS BATHROOM BOTTLE	2	12/1/2010 14:32	2.51	NaN	United Kingdom
1445	536544	21786	POLKADOT RAIN HAT	4	12/1/2010 14:32	0.85	NaN	United Kingdom
1446	536544	21787	RAIN PONCHO RETROSPOT	2	12/1/2010 14:32	1.66	NaN	United Kingdom

In [90]:
```
data.loc[data.CustomerID.isnull(), ["UnitPrice", "Quantity"]].describe()
```

Out[90]:

	UnitPrice	Quantity
count	135080.000000	135080.000000
mean	8.076577	1.995573
std	151.900816	66.696153
min	-11062.060000	-9600.000000
25%	1.630000	1.000000
50%	3.290000	1.000000
75%	5.450000	3.000000
max	17836.460000	5568.000000

The prices and quantities of entries without a customer ID can exhibit extreme outliers. As we might intend to create features later based on historical prices and sold quantities, this could be highly disruptive. Our primary recommendation for the retailer is to establish strategies for handling transactions that are potentially flawed or unique.

The lingering question is: Why is it possible for a transaction to exist without a customer ID? It is conceivable that a purchase might occur without an associated customer account, yet it would be more appropriate to assign a special ID to indicate a guest transaction.

Moving on: Are there hidden 'NaN' (null) values in the Descriptions? To investigate, let's create a new feature that stores descriptions in lowercase:

Hidden missing descriptions

In [91]:
```
data.loc[data.Description.isnull()==False, "lowercase_descriptions"] = data.loc[
    data.Description.isnull()==False,"Description"
].apply(lambda l: l.lower())

data.lowercase_descriptions.dropna().apply(
    lambda l: np.where("nan" in l, True, False)
).value_counts()
```

Out[91]:
```
False    539724
True        731
Name: lowercase_descriptions, dtype: int64
```

empty ""-strings

In [92]:
```
data.lowercase_descriptions.dropna().apply(
    lambda l: np.where("" == l, True, False)
).value_counts()
```

Out[92]:
```
False    540455
Name: lowercase_descriptions, dtype: int64
```

We've discovered additional hidden NaN (null) values represented as the string 'nan' instead of the appropriate NaN value. Let's proceed to transform these instances into NaN.

In [93]:
```
data.loc[data.lowercase_descriptions.isnull()==False, "lowercase_descriptions"] = data
    data.lowercase_descriptions.isnull()==False, "lowercase_descriptions"
].apply(lambda l: np.where("nan" in l, None, l))
```

As the reasons for missing customers or descriptions are unknown and given the presence of unusual outliers in quantities, prices, and zero-pricing, it's prudent to err on the side of caution and remove all such occurrences.

In [94]:
```
data = data.loc[(data.CustomerID.isnull()==False) & (data.lowercase_descriptions.isnul
```

Just to confirm: Are there any remaining missing values?

In [95]:
```
data.isnull().sum().sum()
```

Out[95]:
```
0
```

The Time period

How long is the period in days?

In [96]:
```
data["InvoiceDate"] = pd.to_datetime(data.InvoiceDate, cache=True)

data.InvoiceDate.max() - data.InvoiceDate.min()
```

Out[96]:
```
Timedelta('373 days 04:24:00')
```

In [97]:
```
print("Datafile starts with timepoint {}".format(data.InvoiceDate.min()))
print("Datafile ends with timepoint {}".format(data.InvoiceDate.max()))
```

```
Datafile starts with timepoint 2010-12-01 08:26:00
Datafile ends with timepoint 2011-12-09 12:50:00
```

The invoice number

How many different invoice numbers do we have?

In [98]:
```
data.InvoiceNo.nunique()
```

Out[98]:
```
22186
```

In the data description we can find that a cancelled transactions starts with a "C" in front of it. Let's create a feature to easily filter out these cases:

In [99]:
```
data["IsCancelled"]=np.where(data.InvoiceNo.apply(lambda l: l[0]=="C"), True, False)
data.IsCancelled.value_counts() / data.shape[0] * 100
```

Out[99]:
```
False    97.81007
True      2.18993
Name: IsCancelled, dtype: float64
```

2,2 % of all entries are cancellations.

In [100…
```
data.loc[data.IsCancelled==True].describe()
```

Out[100]:

	Quantity	UnitPrice
count	8896.000000	8896.000000
mean	-30.882981	18.862815
std	1170.746458	444.590459
min	-80995.000000	0.010000
25%	-6.000000	1.450000
50%	-2.000000	2.950000
75%	-1.000000	4.950000
max	-1.000000	38970.000000

Every cancellation exhibits negative quantities but positive, non-zero unit prices. With this data, it's challenging to discern why a customer initiated a return, making it very difficult to predict such cases. Let's proceed by removing these cancellations.

In [101…
```
data = data.loc[data.IsCancelled==False].copy()
data = data.drop("IsCancelled", axis=1)
```

Stockcodes

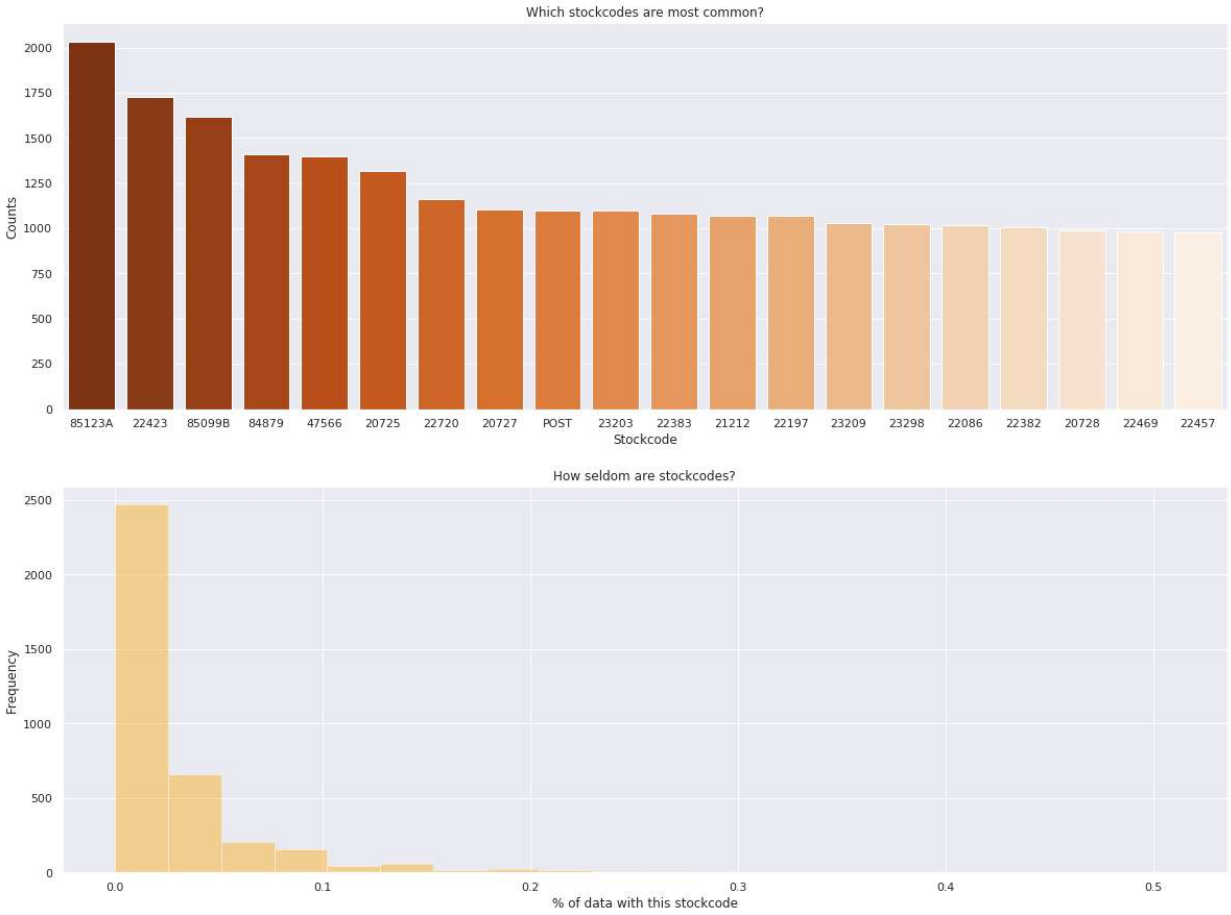
How many unique stockcodes do we have?

In [102…
```
data.StockCode.nunique()
```

Out[102]:
```
3663
```

Which codes are most common?

In [103…
```
stockcode_counts = data.StockCode.value_counts().sort_values(ascending=False)
fig, ax = plt.subplots(2,1,figsize=(20,15))
sns.barplot(stockcode_counts.iloc[0:20].index,
            stockcode_counts.iloc[0:20].values,
            ax = ax[0], palette="Oranges_r")
ax[0].set_ylabel("Counts")
ax[0].set_xlabel("Stockcode")
ax[0].set_title("Which stockcodes are most common?");
sns.distplot(np.round(stockcode_counts/data.shape[0]*100,2),
             kde=False,
             bins=20,
             ax=ax[1], color="Orange")
ax[1].set_title("How seldom are stockcodes?")
ax[1].set_xlabel("% of data with this stockcode")
ax[1].set_ylabel("Frequency");
```



Do you notice that the 'POST' stockcode appears frequently? That's quite unusual! Therefore, we might expect peculiar occurrences not only in the descriptions and customerIDs but also in the stockcode. Oh, and its code is shorter than the others and non-numeric.

Most stockcodes are infrequent, suggesting that the retailer sells a wide variety of products without strong specialization in a specific stockcode. However, we must proceed cautiously as

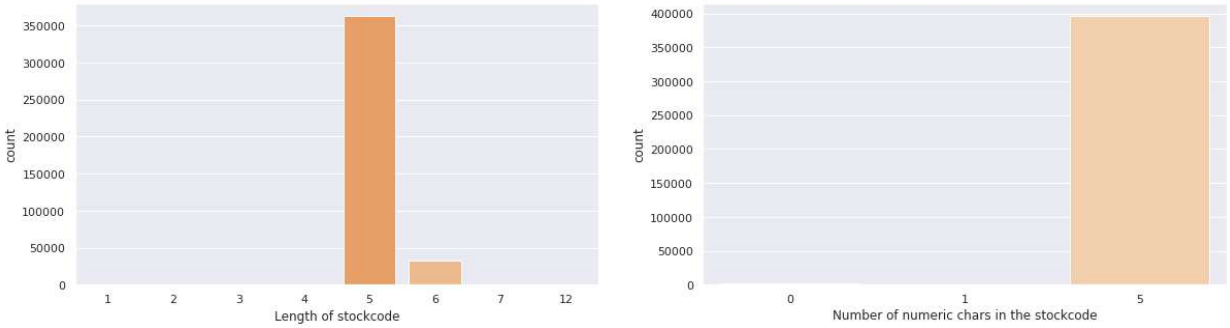
this doesn't necessarily mean the retailer lacks specialization in specific product types. A stockcode could serve as a highly detailed identifier that doesn't inherently specify the type. For instance, 'water bottles' may have various versions in color, name, and shape, but they are all categorized under 'water bottles'.

Let's count the number of numeric chars in and the length of the stockcode:

In [104…
```
def count_numeric_chars(l):
    return sum(1 for c in l if c.isdigit())

data["StockCodeLength"] = data.StockCode.apply(lambda l: len(l))
data["nNumericStockCode"] = data.StockCode.apply(lambda l: count_numeric_chars(l))
```

In [105…
```
fig, ax = plt.subplots(1,2,figsize=(20,5))
sns.countplot(data["StockCodeLength"], palette="Oranges_r", ax=ax[0])
sns.countplot(data["nNumericStockCode"], palette="Oranges_r", ax=ax[1])
ax[0].set_xlabel("Length of stockcode")
ax[1].set_xlabel("Number of numeric chars in the stockcode");
```



While the majority of samples feature a stockcode consisting of 5 numeric characters, there are various other occurrences. The length of stockcodes ranges from 1 to 12 characters, and interestingly, there are stockcodes that contain no numeric characters at all.

In [106…
```
data.loc[data.nNumericStockCode < 5].lowercase_descriptions.value_counts()
```

Out[106]:
```
postage                        1099
manual                          290
carriage                        133
dotcom postage                   16
bank charges                     12
pads to match all cushions        4
Name: lowercase_descriptions, dtype: int64
```

Once more, this is another aspect we do not wish to predict. It suggests that the retailer does not distinguish well between special types of transactions and regular customer-retailer transactions. To address this, let's remove all such occurrences.

In [107…
```
data = data.loc[(data.nNumericStockCode == 5) & (data.StockCodeLength==5)].copy()
data.StockCode.nunique()
```

Out[107]:
```
2783
```

In [108…
```
data = data.drop(["nNumericStockCode", "StockCodeLength"], axis=1)
```

Descriptions

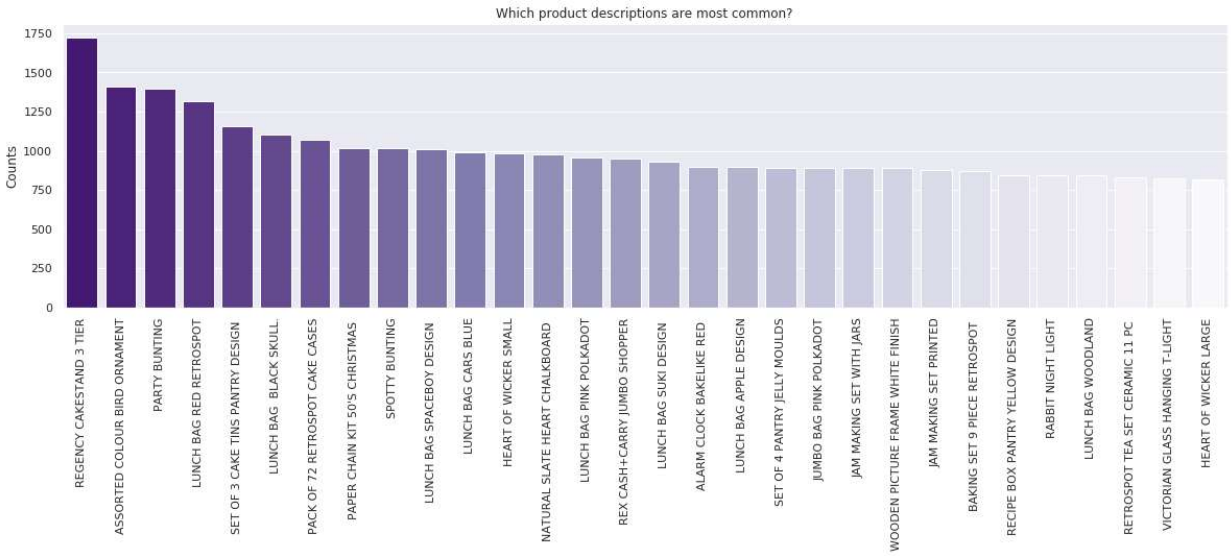
How many unique descriptions do we have?

In [109…
```
data.Description.nunique()
```

Out[109]:
```
2983
```

And which are most common?

In [110…
```
description_counts = data.Description.value_counts().sort_values(ascending=False).iloc
plt.figure(figsize=(20,5))
sns.barplot(description_counts.index, description_counts.values, palette="Purples_r")
plt.ylabel("Counts")
plt.title("Which product descriptions are most common?");
plt.xticks(rotation=90);
```

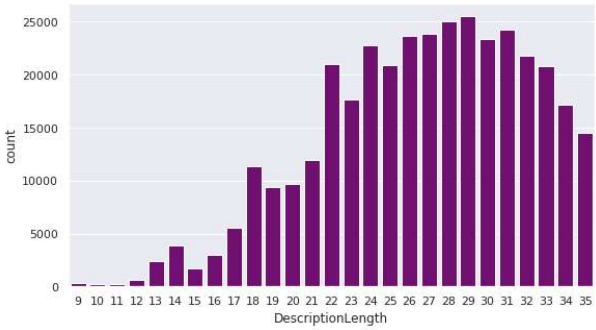


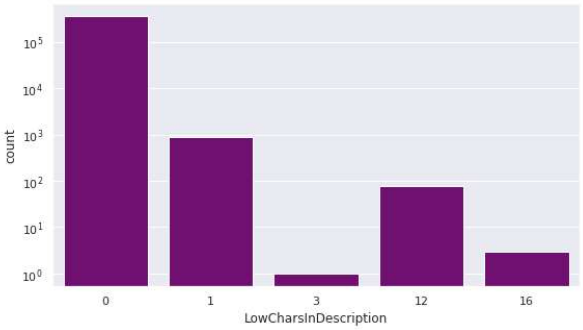
We've noticed that some descriptions correspond to a similar product type, notably the multiple occurrences of 'lunch bags'. These descriptions often contain color information about the product. Moreover, the prevalence of the most common descriptions suggests that the retailer offers a wide range of diverse products. It's worth noting that all descriptions appear to be composed of uppercase characters. Now, let's conduct additional analysis on the descriptions by counting the length and the number of lowercase characters.

In [111…
```
def count_lower_chars(l):
    return sum(1 for c in l if c.islower())
```

In [112…
```
data["DescriptionLength"] = data.Description.apply(lambda l: len(l))
data["LowCharsInDescription"] = data.Description.apply(lambda l: count_lower_chars(l))
```

In [113…
```
fig, ax = plt.subplots(1,2,figsize=(20,5))
sns.countplot(data.DescriptionLength, ax=ax[0], color="Purple")
sns.countplot(data.LowCharsInDescription, ax=ax[1], color="Purple")
ax[1].set_yscale("log")
```

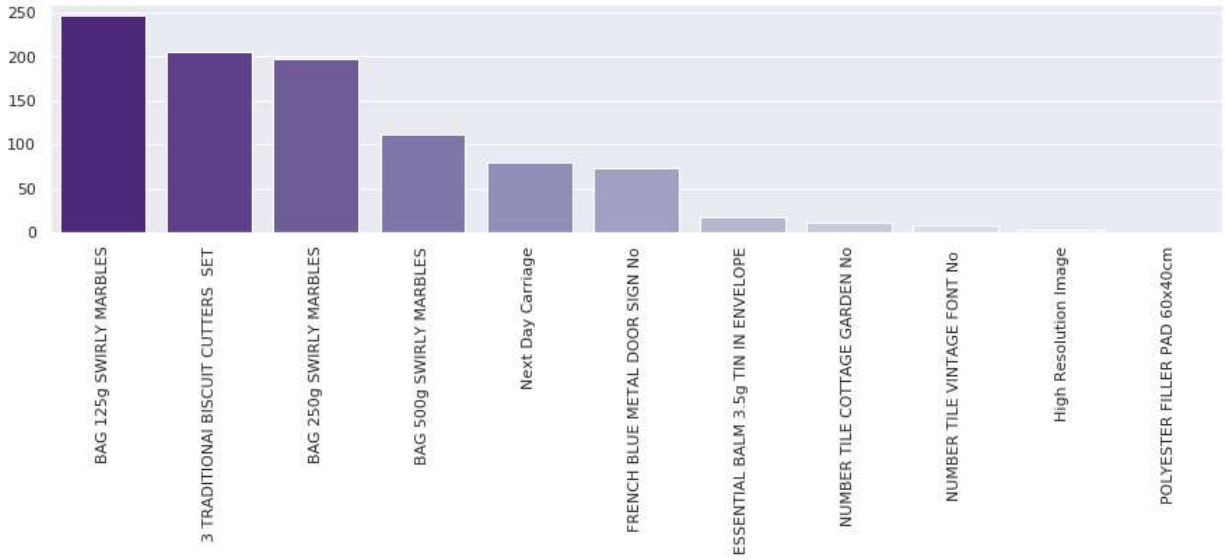




The majority of descriptions do not contain any lowercase characters; however, we have discovered exceptional cases that deviate from this pattern.

In [114…
```
lowchar_counts = data.loc[data.LowCharsInDescription > 0].Description.value_counts()

plt.figure(figsize=(15,3))
sns.barplot(lowchar_counts.index, lowchar_counts.values, palette="Purples_r")
plt.xticks(rotation=90);
```



Terms like 'Next day carriage' and 'High resolution image' stand out as unusual. Let's calculate the fraction of lowercase letters in comparison to uppercase letters.

In [115…
```
def count_upper_chars(l):
    return sum(1 for c in l if c.isupper())

data["UpCharsInDescription"] = data.Description.apply(lambda l: count_upper_chars(l))
```

In [116…
```
data.UpCharsInDescription.describe()
```

Out[116]:
```
count    362522.000000
mean         22.572291
std           4.354845
min           3.000000
25%          20.000000
50%          23.000000
75%          26.000000
max          32.000000
Name: UpCharsInDescription, dtype: float64
```

In [117…
```
data.loc[data.UpCharsInDescription <=5].Description.value_counts()
```

Out[117]:
```
Next Day Carriage        79
High Resolution Image     3
Name: Description, dtype: int64
```

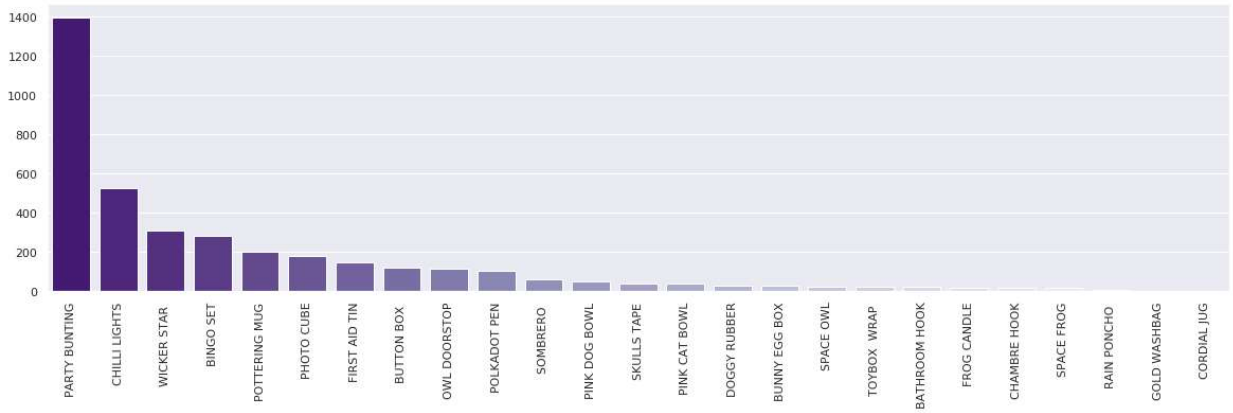
It's strange that they differ from the others. Let's drop them:

In [118…
```
data = data.loc[data.UpCharsInDescription > 5].copy()
```

And what about the descriptions with a length below 14?

In [119…
```
dlength_counts = data.loc[data.DescriptionLength < 14].Description.value_counts()

plt.figure(figsize=(20,5))
sns.barplot(dlength_counts.index, dlength_counts.values, palette="Purples_r")
plt.xticks(rotation=90);
```



Short-length descriptions appear to be valid, so we should retain them. Now, let's determine the count of unique stock codes and unique descriptions in our dataset.

In [120…
```
data.StockCode.nunique()
```

Out[120]:
```
2781
```

In [121…
```
data.Description.nunique()
```

Out[121]:
```
2981
```

We still have more descriptions than stockcodes and we should continue to find out why they differ.

In [122…
```
data.groupby("StockCode").Description.nunique().sort_values(ascending=False).iloc[0:10
```

Out[122]:
```
StockCode
23196    4
23236    4
23244    3
23231    3
23131    3
22937    3
23126    3
23370    3
23366    3
23413    3
Name: Description, dtype: int64
```

we still have stockcodes with multiple descriptions. Let's look at an example:

In [123…
```
data.loc[data.StockCode == "23244"].Description.value_counts()
```

Out[123]:
```
ROUND STORAGE TIN VINTAGE LEAF    96
STORAGE TIN VINTAGE LEAF           7
CANNISTER VINTAGE LEAF DESIGN      2
Name: Description, dtype: int64
```

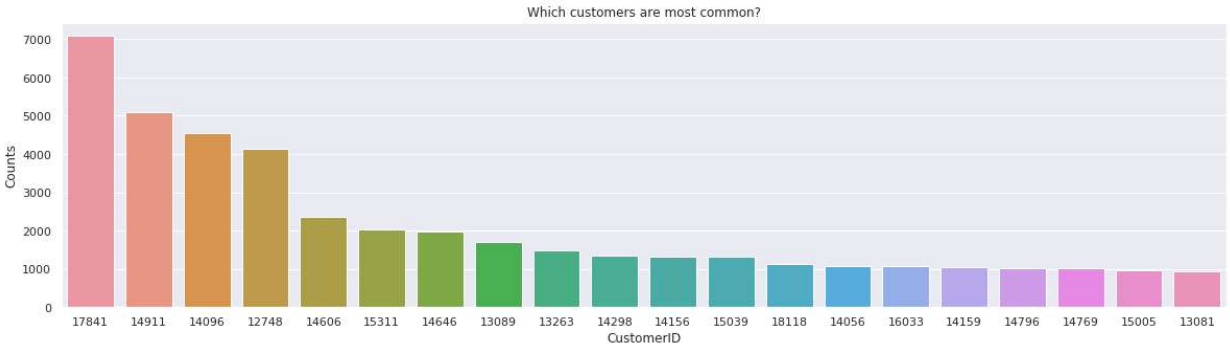
As we browse through the cases, we notice that stock codes are occasionally named slightly differently, perhaps due to missing or altered words or typographical errors. Despite these variations, they appear acceptable, and we can proceed with our analysis."

Customers

In [124…
```
data.CustomerID.nunique()
```

Out[124]:
```
4315
```

In [125…
```
customer_counts = data.CustomerID.value_counts().sort_values(ascending=False).iloc[0:2
plt.figure(figsize=(20,5))
sns.barplot(customer_counts.index, customer_counts.values, order=customer_counts.index
plt.ylabel("Counts")
plt.xlabel("CustomerID")
plt.title("Which customers are most common?");
#plt.xticks(rotation=90);
```



Countries

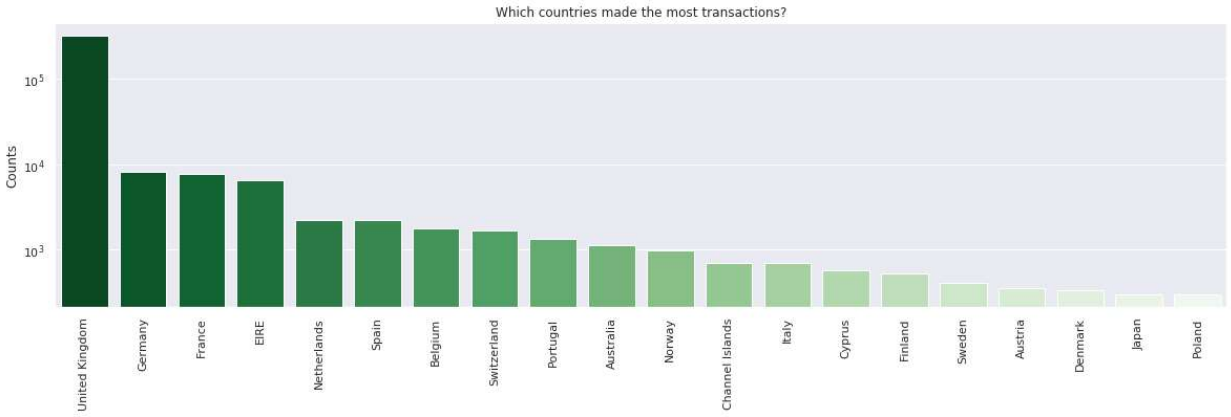
How many unique countries are delivered by the retailer?

In [126…
```
data.Country.nunique()
```

Out[126]:
```
37
```

And which ones are most common?

In [127…
```
country_counts = data.Country.value_counts().sort_values(ascending=False).iloc[0:20]
plt.figure(figsize=(20,5))
sns.barplot(country_counts.index, country_counts.values, palette="Greens_r")
plt.ylabel("Counts")
plt.title("Which countries made the most transactions?");
plt.xticks(rotation=90);
plt.yscale("log")
```



We can observe that the retailer predominantly sells its products in the UK, followed by numerous European countries. What percentage of entries are within the UK?

In [128…
```
data.loc[data.Country=="United Kingdom"].shape[0] / data.shape[0] * 100
```

Out[128]:
```
89.10192031784572
```

Let's create a feature to indicate inside or outside of the UK:

In [129…
```
data["UK"] = np.where(data.Country == "United Kingdom", 1, 0)
```

Unit Price

In [130…
```
data.UnitPrice.describe()
```

Out[130]:
```
count    362440.000000
mean          2.885355
std           4.361812
min           0.000000
25%           1.250000
50%           1.790000
75%           3.750000
max         649.500000
Name: UnitPrice, dtype: float64
```

we have strange occurences: zero unit prices.

In [131…
```
data.loc[data.UnitPrice == 0].sort_values(by="Quantity", ascending=False).head()
```

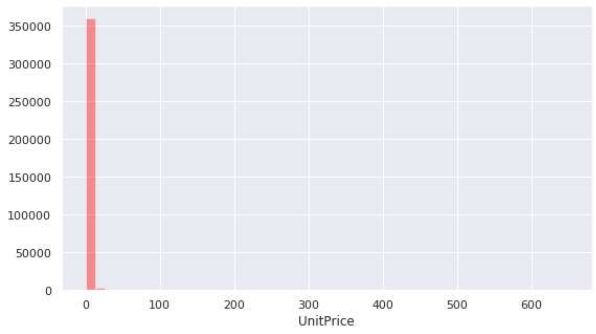
Out[131]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Countr
502122	578841	84826	ASSTD DESIGN 3D PAPER STICKERS	12540	2011-11-25 15:57:00	0.0	13256	Unite Kingdor
298054	562973	23157	SET OF 6 NATIVITY MAGNETS	240	2011-08-11 11:42:00	0.0	14911	EIR
436428	574138	23234	BISCUIT TIN VINTAGE CHRISTMAS	216	2011-11-03 11:26:00	0.0	12415	Australi
314746	564651	23268	SET OF 2 CERAMIC CHRISTMAS REINDEER	192	2011-08-26 14:19:00	0.0	14646	Netherland
314748	564651	21786	POLKADOT RAIN HAT	144	2011-08-26 14:19:00	0.0	14646	Netherland

It's not obvious if they are gifts to customers or not :Let's drop them:

In [132…
```
data = data.loc[data.UnitPrice > 0].copy()
```

In [133…
```
fig, ax = plt.subplots(1,2,figsize=(20,5))
sns.distplot(data.UnitPrice, ax=ax[0], kde=False, color="red")
sns.distplot(np.log(data.UnitPrice), ax=ax[1], bins=20, color="tomato", kde=False)
ax[1].set_xlabel("Log-Unit-Price");
```



In [134…
```
np.exp(-2)
```

Out[134]:
```
0.1353352832366127
```

In [135…
```
np.exp(3)
```

Out[135]:
```
20.085536923187668
```

In [136…
```
np.quantile(data.UnitPrice, 0.95)
```

Out[136]:
```
8.5
```

Let's narrow our focus to transactions with prices within this range, as we aim to avoid making predictions for very rare products with high prices. Beginning with simpler criteria is often a good approach.

In [137…
```
data = data.loc[(data.UnitPrice > 0.1) & (data.UnitPrice < 20)].copy()
```

Quantities

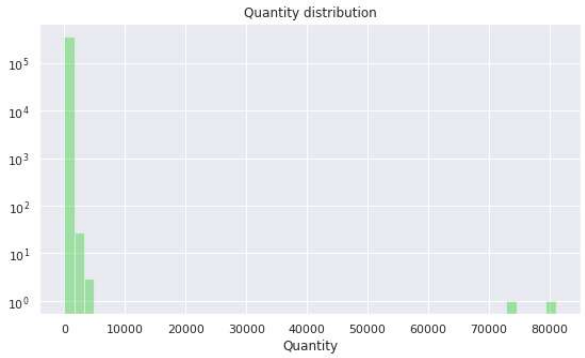
the most important one - the target. Let's take a look at its distribution:

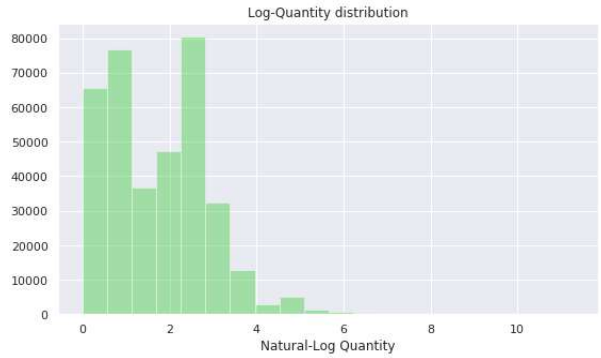
In [138…
```
data.Quantity.describe()
```

Out[138]:
```
count    361608.000000
mean         13.024112
std         187.566510
min           1.000000
25%           2.000000
50%           6.000000
75%          12.000000
max       80995.000000
Name: Quantity, dtype: float64
```

The majority of products are typically sold in quantities ranging from 1 to 12. However, we once again encounter extreme, unrealistic outliers:

In [139…
```
fig, ax = plt.subplots(1,2,figsize=(20,5))
sns.distplot(data.Quantity, ax=ax[0], kde=False, color="limegreen");
sns.distplot(np.log(data.Quantity), ax=ax[1], bins=20, kde=False, color="limegreen");
ax[0].set_title("Quantity distribution")
ax[0].set_yscale("log")
ax[1].set_title("Log-Quantity distribution")
ax[1].set_xlabel("Natural-Log Quantity");
```





As indicated by the log-transformed distribution, it would be reasonable to set a threshold or make a cut at a specific value:

In [140…
```
np.exp(4)
```

Out[140]:
```
54.598150033144236
```

In [141…
```
np.quantile(data.Quantity, 0.95)
```

Out[141]:
```
36.0
```

In this case we would still cover more than 95 % of the data.

In [142…
```
data = data.loc[data.Quantity < 55].copy()
```

Focus on daily product sales

Since our goal is to predict the daily product sales, we need to aggregate this data on a daily basis. To achieve this, we'll extract temporal features from the InvoiceDate. Additionally, we can calculate the revenue generated by each transaction using the unit price and the quantity sold.

In [143…
```
data["Revenue"] = data.Quantity * data.UnitPrice

data["Year"] = data.InvoiceDate.dt.year
data["Quarter"] = data.InvoiceDate.dt.quarter
data["Month"] = data.InvoiceDate.dt.month
data["Week"] = data.InvoiceDate.dt.week
data["Weekday"] = data.InvoiceDate.dt.weekday
data["Day"] = data.InvoiceDate.dt.day
data["Dayofyear"] = data.InvoiceDate.dt.dayofyear
data["Date"] = pd.to_datetime(data[['Year', 'Month', 'Day']])
```

Since the primary objective is to forecast the daily product sales, we can aggregate the daily quantities per product stock code:

In [144…
```
grouped_features = ["Date", "Year", "Quarter","Month", "Week", "Weekday", "Dayofyear",
                    "StockCode"]
```

In this process, we lose information about customers, countries, and pricing details. However, we will recover this information later in this analysis. In addition to aggregating the quantities, let's also aggregate the revenues.

In [145…
```
daily_data = pd.DataFrame(data.groupby(grouped_features).Quantity.sum(),
                          columns=["Quantity"])
daily_data["Revenue"] = data.groupby(grouped_features).Revenue.sum()
daily_data = daily_data.reset_index()
daily_data.head(5)
```

Out[145]:

	Date	Year	Quarter	Month	Week	Weekday	Dayofyear	Day	StockCode	Quantity	Revenue
0	2010-12-01	2010	4	12	48	2	335	1	10002	60	51.00
1	2010-12-01	2010	4	12	48	2	335	1	10125	2	1.70
2	2010-12-01	2010	4	12	48	2	335	1	10133	5	4.25
3	2010-12-01	2010	4	12	48	2	335	1	16014	10	4.20
4	2010-12-01	2010	4	12	48	2	335	1	16016	10	8.50

How are the quantities and revenues distributed?

In [146…
```
daily_data.loc[:, ["Quantity", "Revenue"]].describe()
```

Out[146]:

	Quantity	Revenue
count	195853.000000	195853.000000
mean	14.964244	28.181114
std	18.809496	43.938183
min	1.000000	0.120000
25%	3.000000	6.950000
50%	9.000000	15.300000
75%	20.000000	30.600000
max	411.000000	1266.300000

Observing the minimum and maximum values, it's apparent that the target variable contains extreme outliers. If we intend to use it as the target, excluding these outliers becomes necessary to prevent misleading our validation process. Utilizing early stopping will directly impact the training of predictive models as well.

In [147…
```
low_quantity = daily_data.Quantity.quantile(0.01)
high_quantity = daily_data.Quantity.quantile(0.99)
print((low_quantity, high_quantity))
```

```
(1.0, 88.48000000001048)
```

In [148…
```
low_revenue = daily_data.Revenue.quantile(0.01)
high_revenue = daily_data.Revenue.quantile(0.99)
print((low_revenue, high_revenue))
```

```
(0.78, 204.0)
```

Let's limit our use of the target variable to the data within ranges occupied by 90% of the entries. This initial strategy allows us to exclude heavy outliers. However, it's important to

acknowledge that by excluding a portion of the data, we lose some information. Understanding and analyzing the causes behind these outliers could be beneficial and valuable in gaining general insights.

In [149…
```
samples = daily_data.shape[0]
```

In [150…
```
daily_data = daily_data.loc[
    (daily_data.Quantity >= low_quantity) & (daily_data.Quantity <= high_quantity)]
daily_data = daily_data.loc[
    (daily_data.Revenue >= low_revenue) & (daily_data.Revenue <= high_revenue)]
```

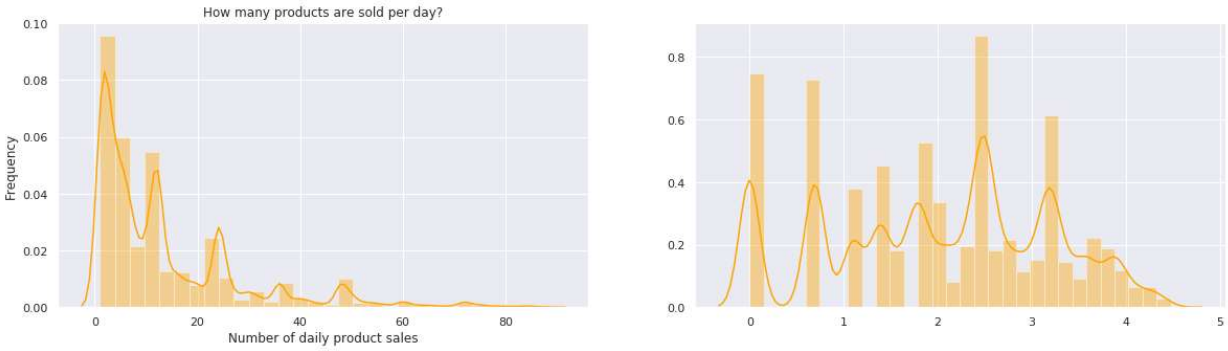
How much entries have we lost?

In [151…
```
samples - daily_data.shape[0]
```

Out[151]:
```
5258
```

Let's take a look at the remaining distributions of daily quantities:

In [152…
```
fig, ax = plt.subplots(1,2,figsize=(20,5))
sns.distplot(daily_data.Quantity.values, kde=True, ax=ax[0], color="Orange", bins=30);
sns.distplot(np.log(daily_data.Quantity.values), kde=True, ax=ax[1], color="Orange", b
ax[0].set_xlabel("Number of daily product sales");
ax[0].set_ylabel("Frequency");
ax[0].set_title("How many products are sold per day?");
```



The distributions appear right-skewed, indicating that lower values are more prevalent. Additionally, the daily sales quantities exhibit a multimodal pattern. It's noteworthy that a daily sale of 1, as well as quantities of 12 and 24, are common. This observation suggests an interesting pattern where quantities are frequently divisible by 2 or 3. In summary, specific products are often purchased either as single units or in small bunches.

How to predict daily product sales?

In this task, I intend to utilize the CatBoost model for predictive tasks. Given that the prediction of daily quantities and revenues both involve regression, I will employ the CatBoost Regressor. The loss and metric I prefer to use is the root mean square error (RMSE).

it's important to note that this metric is heavily influenced by outliers. If there are predictions that significantly deviate from the targets, they will skew the mean towards higher values. As a

result, even if the majority of the predictions are accurate, the RMSE might still be high due to the high errors for a minority of samples.

Validation strategy

Since the data spans only one year and exhibits a substantial surge in product sales during the pre-Christmas period, selecting validation data requires careful consideration. I plan to commence by using validation data that spans at least 8 full weeks (plus the remaining days). After creating new features through data exploration, I'll implement a sliding window time series validation approach. This strategy should assist in determining if the model effectively handles the prediction task during both the pre-Christmas and non-Christmas periods.

Hyperparameter Class

This class encapsulates all crucial hyperparameters essential for setting before training, including the loss function, evaluation metric, maximum depth of trees, the number of maximum trees (iterations), and the l2_leaf_reg used for regularization to prevent overfitting.

In [153…

```
class CatHyperparameter:

    def __init__(self,
                 loss="RMSE",
                 metric="RMSE",
                 iterations=1000,
                 max_depth=4,
                 l2_leaf_reg=3,
                 #learning_rate=0.5,
                 seed=0):
        self.loss = loss,
        self.metric = metric,
        self.max_depth = max_depth,
        self.l2_leaf_reg = l2_leaf_reg,
        #self.learning_rate = learning_rate,
        self.iterations=iterations
        self.seed = seed
```

Catmodel class

The model receives train and validation pools as data, or pandas dataframes, for features X and targets y along with a specific week. It starts with the first week of our validation data and utilizes all subsequent weeks above it. The model not only trains the model but also displays the learning process, feature importances, and provides various figures for result analysis. It's the quickest option to begin exploring and experimenting.

In [154…

```
class Catmodel:

    def __init__(self, name, params):
        self.name = name
        self.params = params

    def set_data_pool(self, train_pool, val_pool):
```

```
        self.train_pool = train_pool
        self.val_pool = val_pool

    def set_data(self, X, y, week):
        cat_features_idx = np.where(X.dtypes != np.float)[0]
        x_train, self.x_val = X.loc[X.Week < week], X.loc[X.Week >= week]
        y_train, self.y_val = y.loc[X.Week < week], y.loc[X.Week >= week]
        self.train_pool = Pool(x_train, y_train, cat_features=cat_features_idx)
        self.val_pool = Pool(self.x_val, self.y_val, cat_features=cat_features_idx)

    def prepare_model(self):
        self.model = CatBoostRegressor(
                loss_function = self.params.loss[0],
                random_seed = self.params.seed,
                logging_level = 'Silent',
                iterations = self.params.iterations,
                max_depth = self.params.max_depth[0],
                #learning_rate = self.params.learning_rate[0],
                l2_leaf_reg = self.params.l2_leaf_reg[0],
                od_type='Iter',
                od_wait=40,
                train_dir=self.name,
                has_time=True
            )

    def learn(self, plot=False):
        self.prepare_model()
        self.model.fit(self.train_pool, eval_set=self.val_pool, plot=plot);
        print("{}, early-stopped model tree count {}".format(
            self.name, self.model.tree_count_
        ))

    def score(self):
        return self.model.score(self.val_pool)

    def show_importances(self, kind="bar"):
        explainer = shap.TreeExplainer(self.model)
        shap_values = explainer.shap_values(self.val_pool)
        if kind=="bar":
            return shap.summary_plot(shap_values, self.x_val, plot_type="bar")
        return shap.summary_plot(shap_values, self.x_val)

    def get_val_results(self):
        self.results = pd.DataFrame(self.y_val)
        self.results["prediction"] = self.predict(self.x_val)
        self.results["error"] = np.abs(
            self.results[self.results.columns.values[0]].values - self.results.predict
        self.results["Month"] = self.x_val.Month
        self.results["SquaredError"] = self.results.error.apply(lambda l: np.power(l,

    def show_val_results(self):
        self.get_val_results()
        fig, ax = plt.subplots(1,2,figsize=(20,5))
        sns.distplot(self.results.error, ax=ax[0])
        ax[0].set_xlabel("Single absolute error")
        ax[0].set_ylabel("Density")
        self.median_absolute_error = np.median(self.results.error)
        print("Median absolute error: {}".format(self.median_absolute_error))
        ax[0].axvline(self.median_absolute_error, c="black")
        ax[1].scatter(self.results.prediction.values,
```

```
                    self.results[self.results.columns[0]].values,
                    c=self.results.error, cmap="RdYlBu_r", s=1)
        ax[1].set_xlabel("Prediction")
        ax[1].set_ylabel("Target")
        return ax

    def get_monthly_RMSE(self):
        return self.results.groupby("Month").SquaredError.mean().apply(lambda l: np.sc

    def predict(self, x):
        return self.model.predict(x)

    def get_dependence_plot(self, feature1, feature2=None):
        explainer = shap.TreeExplainer(self.model)
        shap_values = explainer.shap_values(self.val_pool)
        if feature2 is None:
            return shap.dependence_plot(
                feature1,
                shap_values,
                self.x_val,
            )
        else:
            return shap.dependence_plot(
                feature1,
                shap_values,
                self.x_val,
                interaction_index=feature2
            )
```

Time series validation Catfamily

This model manages the data splitting into validation chunks and orchestrates training using sliding window validation. Additionally, it can compute a score by averaging the RMSE scores from all its models.

In [155…

```
class CatFamily:

    def __init__(self, params, X, y, n_splits=2):
        self.family = {}
        self.cat_features_idx = np.where(X.dtypes != np.float)[0]
        self.X = X.values
        self.y = y.values
        self.n_splits = n_splits
        self.params = params

    def set_validation_strategy(self):
        self.cv = TimeSeriesSplit(max_train_size = None,
                                  n_splits = self.n_splits)
        self.gen = self.cv.split(self.X)

    def get_split(self):
        train_idx, val_idx = next(self.gen)
        x_train, x_val = self.X[train_idx], self.X[val_idx]
        y_train, y_val = self.y[train_idx], self.y[val_idx]
        train_pool = Pool(x_train, y_train, cat_features=self.cat_features_idx)
        val_pool = Pool(x_val, y_val, cat_features=self.cat_features_idx)
```

```
        return train_pool, val_pool

    def learn(self):
        self.set_validation_strategy()
        self.model_names = []
        self.model_scores = []
        for split in range(self.n_splits):
            name = 'Model_cv_' + str(split) + '/'
            train_pool, val_pool = self.get_split()
            self.model_names.append(name)
            self.family[name], score = self.fit_catmodel(name, train_pool, val_pool)
            self.model_scores.append(score)

    def fit_catmodel(self, name, train_pool, val_pool):
        cat = Catmodel(name, train_pool, val_pool, self.params)
        cat.prepare_model()
        cat.learn()
        score = cat.score()
        return cat, score

    def score(self):
        return np.mean(self.model_scores)

    def show_learning(self):
        widget = MetricVisualizer(self.model_names)
        widget.start()

    def show_importances(self):
        name = self.model_names[-1]
        cat = self.family[name]
        explainer = shap.TreeExplainer(cat.model)
        shap_values = explainer.shap_values(cat.val_pool)
        return shap.summary_plot(shap_values, X, plot_type="bar")
```

Baseline model & result analysis

Let's see how good this model performs without feature engineering and hyperparameter search:

In [156…

```
daily_data.head()
```

Out[156]:

	Date	Year	Quarter	Month	Week	Weekday	Dayofyear	Day	StockCode	Quantity	Revenue
0	2010-12-01	2010	4	12	48	2	335	1	10002	60	51.00
1	2010-12-01	2010	4	12	48	2	335	1	10125	2	1.70
2	2010-12-01	2010	4	12	48	2	335	1	10133	5	4.25
3	2010-12-01	2010	4	12	48	2	335	1	16014	10	4.20
4	2010-12-01	2010	4	12	48	2	335	1	16016	10	8.50

In [157…
```
week = daily_data.Week.max() - 2
print("Validation after week {}".format(week))
print("Validation starts at timepoint {}".format(
    daily_data[daily_data.Week==week].Date.min()
))
```

```
Validation after week 49
Validation starts at timepoint 2010-12-06 00:00:00
```

In [158…
```
X = daily_data.drop(["Quantity", "Revenue", "Date"], axis=1)
daily_data.Quantity = np.log(daily_data.Quantity)
y = daily_data.Quantity
params = CatHyperparameter()

model = Catmodel("baseline", params)
model.set_data(X,y, week)
model.learn(plot=True)
```

```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))
baseline, early-stopped model tree count 699
```

If you've forked this kernel and are in the interactive mode, you may observe that the model loss has converged. What is the evaluated root mean square error on the validation data?

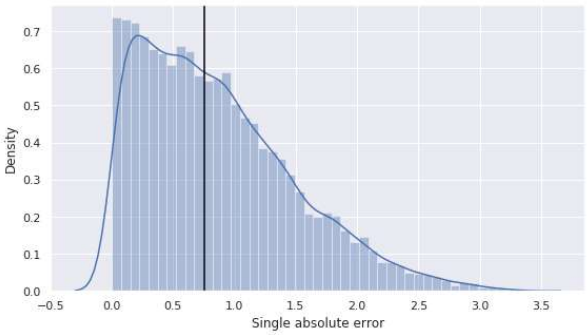
In [159…
```
model.score()
```

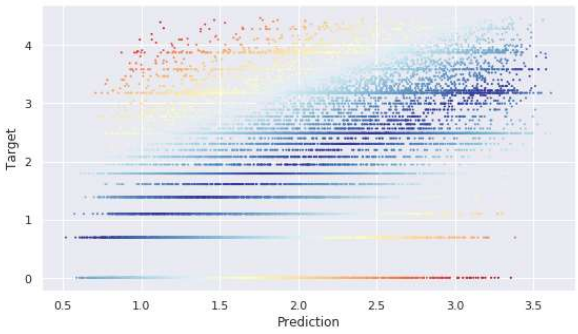
Out[159]:
```
1.065775039230805
```

This value is high, as previously noted, but it's important to remember that RSME is influenced by outliers. Let's take a look at the distribution of individual absolute errors.

In [160…
```
model.show_val_results();
```

```
Median absolute error: 0.7523879147418431
```

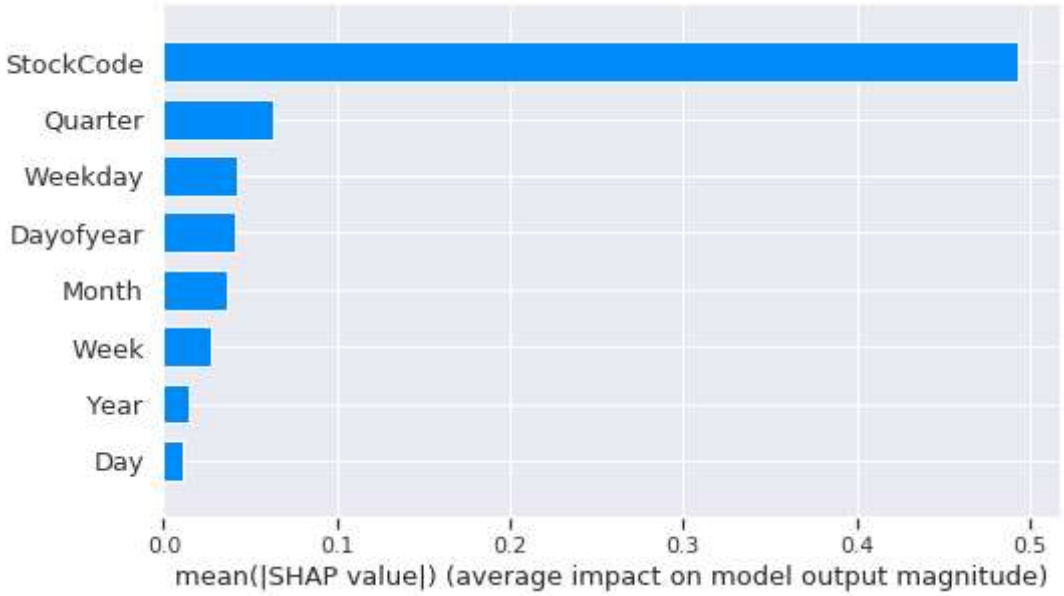




The distribution of absolute errors for individual predictions is right-skewed. The median single error (depicted in black) stands at half the RMSE score and is notably lower. By plotting the target against predictions, we observe higher errors for validation entries with true quantity values exceeding 30. The strong blue line represents the identity where predictions closely match target values. Improvement is needed in making better predictions for products with true high quantities during the validation period.

In [161…
```
model.show_importances()
```

```
The model has complex ctrs, so the SHAP values will be calculated approximately.
```

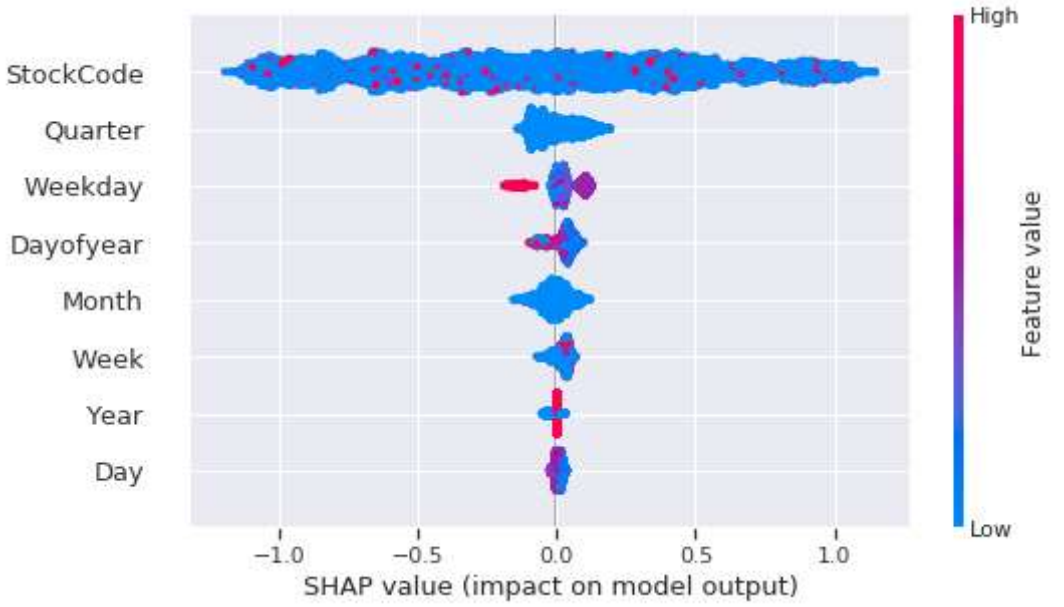


Both the stock code and product description play significant roles. These features lack numeric or value-based properties but demonstrate their importance in predicting sales.

The weekday is also a crucial feature, as earlier exploration of the data has shown. Lower values from Monday to Thursday represent the days with the highest sales for the retailer. Conversely, higher values (Friday to Sunday) indicate fewer sales.

In [162…
```
model.show_importances(kind=None)
```

```
The model has complex ctrs, so the SHAP values will be calculated approximately.
```



Examining the weekday feature in this plot clarifies its impact: Lower values (0 to 3) corresponding to Monday, Tuesday, Wednesday, and Thursday, show a high volume of product sales, indicated by high quantity target values. These days are depicted in blue, leading to higher sharp values and consequently higher predicted quantity values. Conversely, higher

weekday values, representing Friday, Saturday, and Sunday, are displayed in red, guiding towards negative sharp values and lower predicted values. This aligns with our prior observations during the exploration of weekdays and the sum of daily quantities.

The StockCode and Description features are notably important, albeit highly complex. With close to 4000 different stock codes and even more descriptions, it's apparent that improving feature engineering by creating more generalized descriptors for the products could be beneficial.

In [163…
```
np.mean(np.abs(np.exp(model.results.prediction) - np.exp(model.results.Quantity)))
```

Out[163]:
```
9.383558683392673
```

In [164…
```
np.median(np.abs(np.exp(model.results.prediction) - np.exp(model.results.Quantity)))
```

Out[164]:
```
4.875429653223724
```

Bayesian Hyperparameter Search with GPyOpt

Now that we've familiarized ourselves with a single model, let's explore if we can achieve an improved score through hyperparameter search.

In [165…
```
#search = Hypertuner(model, max_depth=5, max_l2_leaf_reg=30)
#search.learn()
#model = search.retrain_catmodel()
#print(model.score())
#model.show_importances(kind=None)
```

Feature engineering

Creating product types

In [166…
```
products = pd.DataFrame(index=data.loc[data.Week < week].StockCode.unique(), columns =

products["MedianPrice"] = data.loc[data.Week < week].groupby("StockCode").UnitPrice.me
products["MedianQuantities"] = data.loc[data.Week < week].groupby("StockCode").Quantit
products["Customers"] = data.loc[data.Week < week].groupby("StockCode").CustomerID.nur
products["DescriptionLength"] = data.loc[data.Week < week].groupby("StockCode").Descri
#products["StockCode"] = products.index.values
org_cols = np.copy(products.columns.values)
products.head()
```

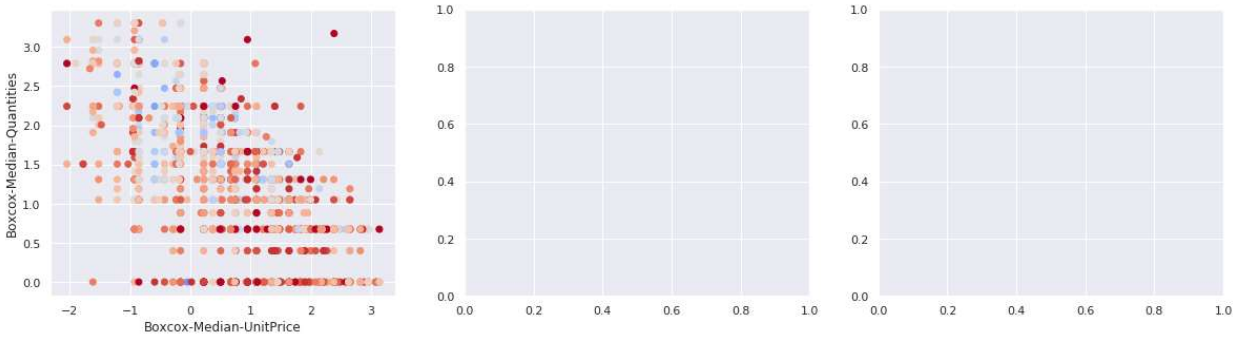
Out[166]:

	MedianPrice	MedianQuantities	Customers	DescriptionLength
71053	3.75	4.0	137	19
22752	8.50	2.0	163	28
21730	4.95	3.0	64	33
22633	2.10	4.0	263	22
22632	2.10	4.0	227	25

In [167…
```
for col in org_cols:
    if col != "StockCode":
        products[col] = boxcox(products[col])[0]
```

In [168…
```
fig, ax = plt.subplots(1,3,figsize=(20,5))
ax[0].scatter(products.MedianPrice.values, products.MedianQuantities.values,
           c=products.Customers.values, cmap="coolwarm_r")
ax[0].set_xlabel("Boxcox-Median-UnitPrice")
ax[0].set_ylabel("Boxcox-Median-Quantities")
```

Out[168]:
```
Text(0,0.5,'Boxcox-Median-Quantities')
```



In [169…
```
X = products.values
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

In [170…
```
km = KMeans(n_clusters=30)
products["cluster"] = km.fit_predict(X)

daily_data["ProductType"] = daily_data.StockCode.map(products.cluster)
daily_data.ProductType = daily_data.ProductType.astype("object")
daily_data.head()
```

Out[170]:

	Date	Year	Quarter	Month	Week	Weekday	Dayofyear	Day	StockCode	Quantity	Revenue	F
0	2010-12-01	2010	4	12	48	2	335	1	10002	4.094345	51.00	
1	2010-12-01	2010	4	12	48	2	335	1	10125	0.693147	1.70	
2	2010-12-01	2010	4	12	48	2	335	1	10133	1.609438	4.25	
3	2010-12-01	2010	4	12	48	2	335	1	16014	2.302585	4.20	
4	2010-12-01	2010	4	12	48	2	335	1	16016	2.302585	8.50	

Baseline for product types

In [171…

```
daily_data["KnownStockCodeUnitPriceMedian"] = daily_data.StockCode.map(
    data.groupby("StockCode").UnitPrice.median())

known_price_iqr = data.groupby("StockCode").UnitPrice.quantile(0.75)
known_price_iqr -= data.groupby("StockCode").UnitPrice.quantile(0.25)
daily_data["KnownStockCodeUnitPriceIQR"] = daily_data.StockCode.map(known_price_iqr)
```

In [172…

```
to_group = ["StockCode", "Year", "Month", "Week", "Weekday"]

daily_data = daily_data.set_index(to_group)
daily_data["KnownStockCodePrice_WW_median"] = daily_data.index.map(
    data.groupby(to_group).UnitPrice.median())
daily_data["KnownStockCodePrice_WW_mean"] = daily_data.index.map(
    data.groupby(to_group).UnitPrice.mean().apply(lambda l: np.round(l, 2)))
daily_data["KnownStockCodePrice_WW_std"] = daily_data.index.map(
    data.groupby(to_group).UnitPrice.std().apply(lambda l: np.round(l, 2)))

daily_data = daily_data.reset_index()
```

In [173…

```
daily_data.head()
```

Out[173]:

	StockCode	Year	Month	Week	Weekday	Date	Quarter	Dayofyear	Day	Quantity	Revenue	F
0	10002	2010	12	48	2	2010-12-01	4	335	1	4.094345	51.00	
1	10125	2010	12	48	2	2010-12-01	4	335	1	0.693147	1.70	
2	10133	2010	12	48	2	2010-12-01	4	335	1	1.609438	4.25	
3	16014	2010	12	48	2	2010-12-01	4	335	1	2.302585	4.20	
4	16016	2010	12	48	2	2010-12-01	4	335	1	2.302585	8.50	

Temporal patterns

In [174…

```
plt.figure(figsize=(20,5))
plt.plot(daily_data.groupby("Date").Quantity.sum(), marker='+', c="darkorange")
plt.plot(daily_data.groupby("Date").Quantity.sum().rolling(window=30, center=True).mea
        c="red")
plt.xticks(rotation=90);
plt.title("How many quantities are sold per day over the given time?");
```

In [175…

```
fig, ax = plt.subplots(1,2,figsize=(20,5))

weekdays = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunda
yearmonth = ["Dec-2010", "Jan-2011", "Feb-2011", "Mar-2011", "Apr-2011", "May-2011",
             "Jun-2011", "Jul-1011", "Aug-2011", "Sep-2011", "Oct-2011", "Nov-2011",
             "Dec-2011"]

daily_data.groupby("Weekday").Quantity.sum().plot(
    ax=ax[0], marker='o', label="Quantity", c="darkorange");
ax[0].legend();
ax[0].set_xticks(np.arange(0,7))
ax[0].set_xticklabels(weekdays);
ax[0].set_xlabel("")
ax[0].set_title("Total sales per weekday");

ax[1].plot(daily_data.groupby(["Year", "Month"]).Quantity.sum().values,
    marker='o', label="Quantities", c="darkorange");
ax[1].set_xticklabels(yearmonth, rotation=90)
ax[1].set_xticks(np.arange(0, len(yearmonth)))
ax[1].legend();
ax[1].set_title("Total sales per month");
```

Thursday appears to be the day with the highest product sales. Conversely, Fridays and Sundays show considerably fewer transactions, while Saturdays indicate no transactions at all. The pre-Christmas season seems to commence in September and peaks in November. Indeed, February and April depict months with notably low sales.

Given these insights, let's create new features for our daily aggregation, which may aid in making more accurate predictions.

In [176…
```
daily_data["PreChristmas"] = (daily_data.Dayofyear <= 358) & (daily_data.Dayofyear >=
```

In [177…
```
for col in ["Weekday", "Month", "Quarter"]:
    daily_data = daily_data.set_index(col)
    daily_data[col+"Quantity_mean"] = daily_data.loc[daily_data.Week < week].groupby(c
    daily_data[col+"Quantity_median"] = daily_data.loc[daily_data.Week < week].groupby
    daily_data[col+"Quantity_mean_median_diff"] = daily_data[col+"Quantity_mean"] - da
    daily_data[col+"Quantity_IQR"] = daily_data.loc[
        daily_data.Week < week].groupby(col).Quantity.quantile(0.75) - daily_data.loc[
        daily_data.Week < week].groupby(col).Quantity.quantile(0.25)
    daily_data = daily_data.reset_index()
daily_data.head()
```

Out[177]:

	Quarter	Month	Weekday	StockCode	Year	Week	Date	Dayofyear	Day	Quantity	Revenue	F
0	4	12	2	10002	2010	48	2010-12-01	335	1	4.094345	51.00	
1	4	12	2	10125	2010	48	2010-12-01	335	1	0.693147	1.70	
2	4	12	2	10133	2010	48	2010-12-01	335	1	1.609438	4.25	
3	4	12	2	16014	2010	48	2010-12-01	335	1	2.302585	4.20	
4	4	12	2	16016	2010	48	2010-12-01	335	1	2.302585	8.50	

In [178…
```
to_group = ["StockCode", "PreChristmas"]
daily_data = daily_data.set_index(to_group)
daily_data["PreChristmasMeanQuantity"] = daily_data.loc[
    daily_data.Week < week].groupby(to_group).Quantity.mean().apply(lambda l: np.round
daily_data["PreChristmasMedianQuantity"] = daily_data.loc[
    daily_data.Week < week].groupby(to_group).Quantity.median().apply(lambda l: np.rou
daily_data["PreChristmasStdQuantity"] = daily_data.loc[
    daily_data.Week < week].groupby(to_group).Quantity.std().apply(lambda l: np.round(
daily_data = daily_data.reset_index()
```

Lag-Features:

In [179…
```
for delta in range(1,4):
    to_group = ["Week","Weekday","ProductType"]
    daily_data = daily_data.set_index(to_group)

    daily_data["QuantityProducttypeWeekWeekdayLag_" + str(delta) + "_median"] = daily_
```

```
        to_group).Quantity.median().apply(lambda l: np.round(l,1)).shift(delta)

    daily_data = daily_data.reset_index()
    daily_data.loc[daily_data.Week >= (week+delta),
                   "QuantityProductTypeWeekWeekdayLag_" + str(delta) + "_median"] = np
```

In [180…
```
data["ProductType"] = data.StockCode.map(products.cluster)
```

In [181…
```
daily_data["TransactionsPerProductType"] = daily_data.ProductType.map(data.loc[data.We
```

About countries and customers

In [182…
```
delta = 1
to_group = ["Week", "Weekday", "ProductType"]
daily_data = daily_data.set_index(to_group)
daily_data["DummyWeekWeekdayAttraction"] = data.groupby(to_group).CustomerID.nunique()
daily_data["DummyWeekWeekdayMeanUnitPrice"] = data.groupby(to_group).UnitPrice.mean().

daily_data["WeekWeekdayAttraction_Lag1"] = daily_data["DummyWeekWeekdayAttraction"].sh
daily_data["WeekWeekdayMeanUnitPrice_Lag1"] = daily_data["DummyWeekWeekdayMeanUnitPric

daily_data = daily_data.reset_index()
daily_data.loc[daily_data.Week >= (week + delta), "WeekWeekdayAttraction_Lag1"] = np.r
daily_data.loc[daily_data.Week >= (week + delta), "WeekWeekdayMeanUnitPrice_Lag1"] = r
daily_data = daily_data.drop(["DummyWeekWeekdayAttraction", "DummyWeekWeekdayMeanUnitP
```

In [183…
```
daily_data["TransactionsPerStockCode"] = daily_data.StockCode.map(
    data.loc[data.Week < week].groupby("StockCode").InvoiceNo.nunique())
```

In [184…
```
daily_data.head()
```

Out[184]:

	Week	Weekday	ProductType	StockCode	PreChristmas	Quarter	Month	Year	Date	Dayofyear
0	48	2	0.0	10002	True	4	12	2010	2010-12-01	335
1	48	2	28.0	10125	True	4	12	2010	2010-12-01	335
2	48	2	6.0	10133	True	4	12	2010	2010-12-01	335
3	48	2	7.0	16014	True	4	12	2010	2010-12-01	335
4	48	2	0.0	16016	True	4	12	2010	2010-12-01	335

In [185…
```
daily_data["CustomersPerWeekday"] = daily_data.Month.map(
    data.loc[data.Week < week].groupby("Weekday").CustomerID.nunique())
```

In [186…
```
X = daily_data.drop(["Quantity", "Revenue", "Date", "Year"], axis=1)
y = daily_data.Quantity
params = CatHyperparameter()
```

```
model = Catmodel("new_features_1", params)
model.set_data(X,y, week)
model.learn(plot=True)
```

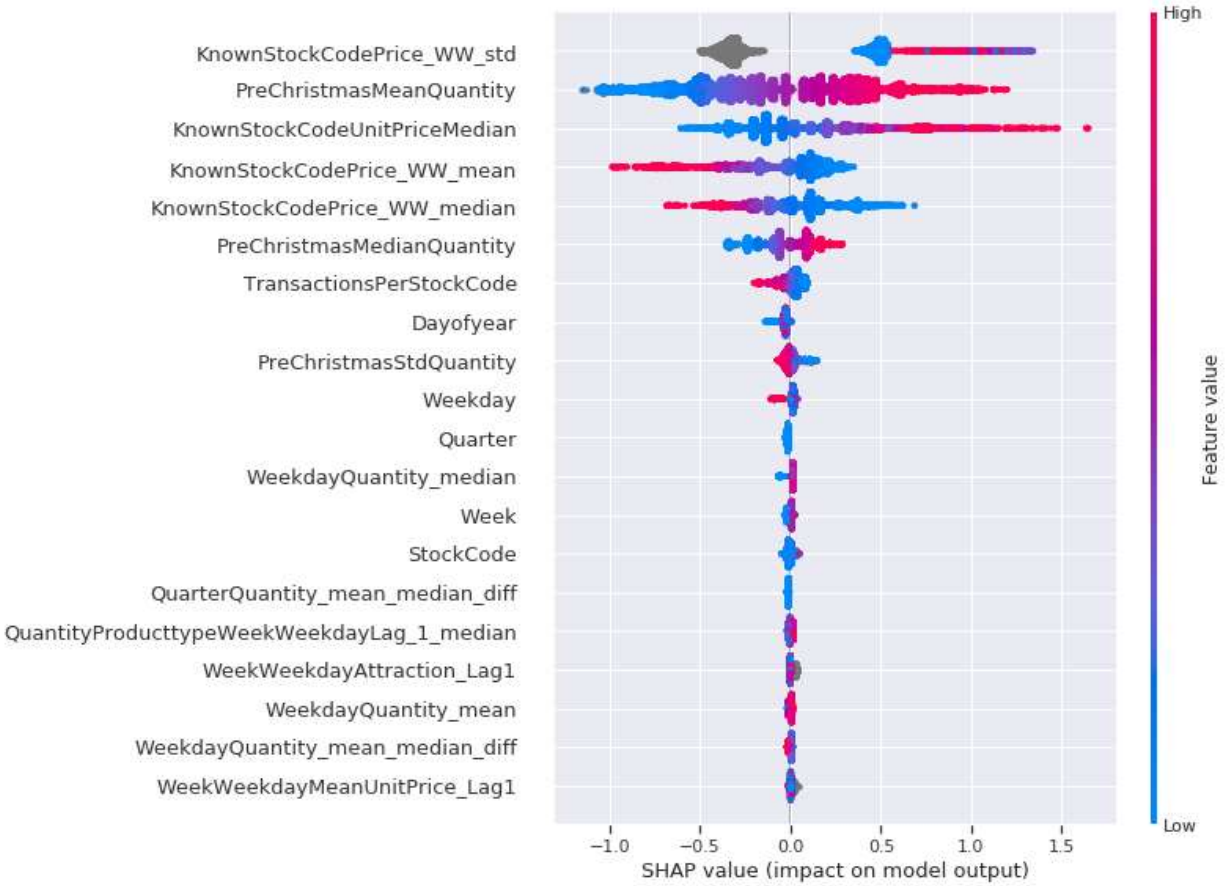
```
MetricVisualizer(layout=Layout(align_self='stretch', height='500px'))
new_features_1, early-stopped model tree count 650
```

In [187…
```
model.score()
```

Out[187]:
```
0.8846873222221254
```

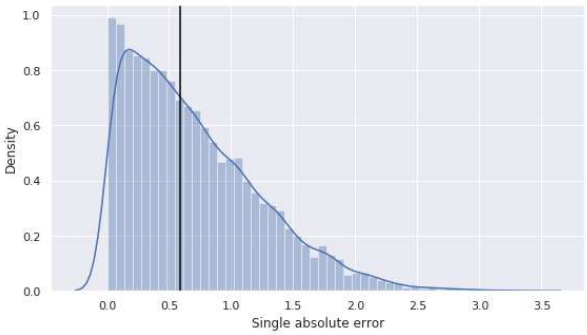
In [188…
```
model.show_importances(kind=None)
```

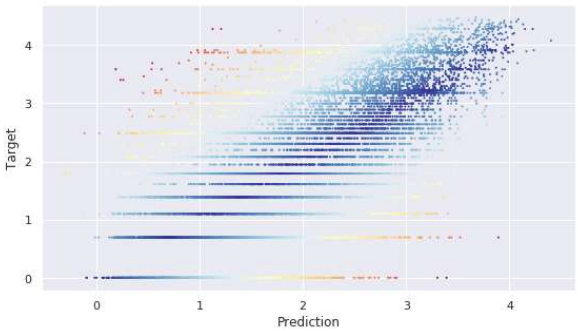
```
The model has complex ctrs, so the SHAP values will be calculated approximately.
```



In [189…
```
model.show_val_results();
```

```
Median absolute error: 0.5908334965501176
```





In [190…
```
np.mean(np.abs(np.exp(model.results.prediction) - np.exp(model.results.Quantity)))
```

Out[190]:
```
7.6246374267556885
```

In [191…
```
np.median(np.abs(np.exp(model.results.prediction) - np.exp(model.results.Quantity)))
```

Out[191]:
```
3.9136944324141005
```