On the Same Wavelength: Clustering Product Demand with Dynamic Time Warping

Problem:

My project used data from the American subsidiary of a European manufacturing company of high-end consumer products (the company wishes to remain anonymous). Demand for this product is highly seasonal with 45-48 percent of all sales occurring in the fourth quarter of the year. All forecasting is done on a yearly basis since long lead times (120 days) provide little time to adjust to peak season market changes.

Luckily, carrying costs are low (the items take up little space in the warehouse, do not expire, or go out of fashion) but the cost of stock-outs are high (with 5 major retailers accounting for 59 percentage of all orders, it is in the company's interest to keep retailers happy by fulfilling orders). To ensure high customer service levels, it is essential to have the right inventory (mix and quantities) in the warehouse. My task was to find forecasting methods that improve the company's ability to predict seasonal demand for these products.

In [13]:
```
# imports
from functions import load_data, ts_train_test_split, make_copy_df, plot
_time_series
from functions import moving_average, RMSE, rename_columns, create_fb_fo
recast
# functions for data import, processing, forecasting and plotting in the
 functions.py file
from product_segmentation_functions import identify_non_active, identify
_new_product
from product_segmentation_functions import identify_intermittent_product
from product_segmentation_functions import identify_minute_demand, ident
ify_repackage_product
from product_segmentation_functions import make_remainder_dataframe
# functions for product segmentation in segmentation_functions.py file
from clustering_functions import prep_dataframe_for_warping, assign_prod
ucts
# functions for clustering
from ts_cluster import ts_cluster
# this class was written by Alex Minnaar, minor modifications added by m
e
import numpy as np
```

In [15]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
%load_ext autoreload
%autoreload 2
import autoreload
```

1. Load data

- Data: 1833 products from 2012-04-08 to 2017-11-05
- Received Excel file with over 922K instances
- Describe prepping data (CH)

In [3]:
```
data_df = load_data('data/time_series.xlsx')
data_df.head()
```

Out[3]:

	012	017	03008944ST-1	03008944ST-3	0300ST1550-1	0300ST15X9-1	0300ST15X9-2
EntDate							
2012-04-08	0	0	0	0	0	0	0
2012-04-15	0	0	0	0	0	0	0
2012-04-22	0	0	0	0	0	0	0
2012-04-29	0	0	0	0	0	0	0
2012-05-06	0	0	0	0	0	0	0

5 rows × 1833 columns

2. Train-Test Split

Time series data cannot be evaluated using traditional cross validation methods. That leaves two options:

- Splitting the data manually using a certain point in time as our division line between 'past' observations (the training set) and 'future' values (the testing set we can measure our forecasts against).
- Splitting data into multiple training/testing folds using TimeSeriesSplit from the sklearn library.

Since forecasting is done on a yearly basis and my dataset only covers the time period through November 2017, I decided to split the data manually, using 5 years' worth for training, 1 year for testing, and then producing a third forecast for the time period of November 2017-May 2018 to test against completely unseen data.

In [4]:
```
# splitting into training and testing sets setting aside last year for t
esting
train_df, test_df = ts_train_test_split(data_df, 52)
# test set has been set aside until models are trained...
```

```
Observations: 292
Training Observations: 240
Testing Observations: 52
```

3. Pick Forecasting Metric and Models

Forecasting Metric:

I selected Root Mean Squared Error as a forecasting metric -- since there is only one dataset, scale-dependent errors can be used to compare different forecasting methods.

- RMSE (root mean squared error): $\sqrt{\frac{(A_t - F_t)^2}{n}}$

Forecasting Methods:

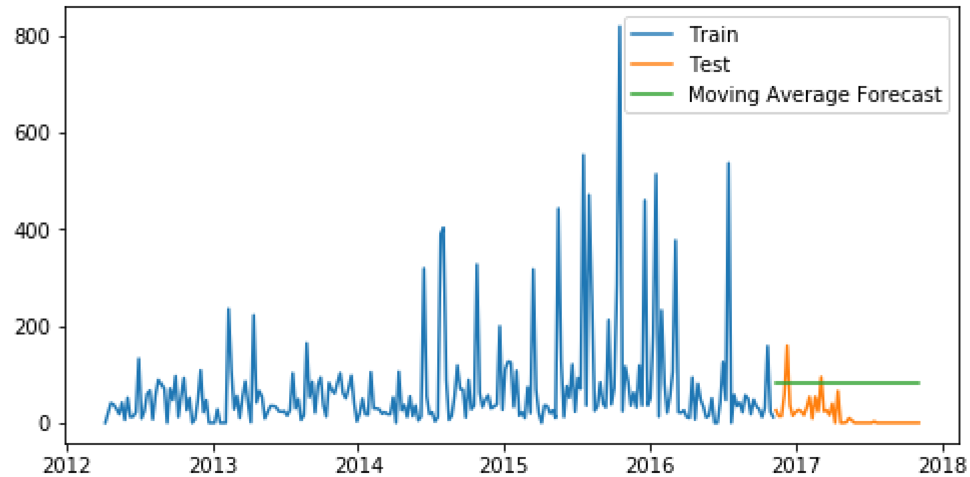
I compared three methods for forecasting for seasonal demand:

- Box-Jenkins (seasonal ARIMA), which is a 5-step process, including optimizing 7 parameters;
- Holt-Winters (Triple Exponential Smoothing), which requires adjusting 7 parameters, some of them to 3-point decimals; and
- FB Prophet, which was built to with the goal "to make it easier for experts and non-experts to make high quality forecasts that keep up with demand." Rather than requiring substantial experience in tuning parameters, "Prophet's default settings to produce forecasts that are often accurate as those produced by skilled forecasters, with much less effort." (see blog post: https://research.fb.com/prophet-forecasting-at-scale/ (https://research.fb.com/prophet-forecasting-at-scale/))

That sounds great, why bother with the other methods? Let's look at item # 9920-2:

In [5]:
```
# make a df to store all our predictions
y_hat = make_copy_df(test_df, '9920-2')
```

In [6]:
```
# baseline: Moving Average with 52
y_hat['moving_avg'] = moving_average(train_df['9920-2'], m=52)

plot_time_series(train_df, test_df, '9920-2', y_hat, 'moving_avg', 'Movi
ng Average Forecast')

RMSE(test_df,'9920-2', y_hat, 'moving_avg')
```



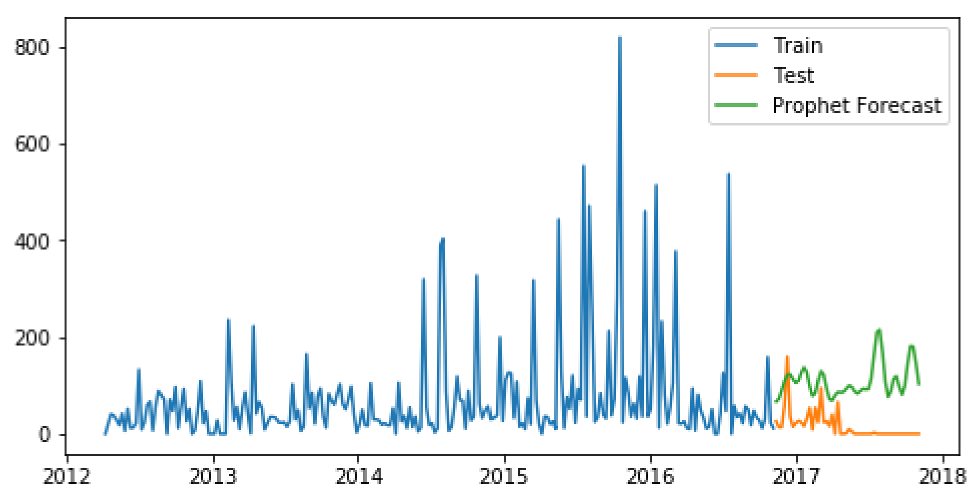
Out[6]: 70.59469489094461

Okay, 70.59, that's not great. Let's see what Prophet can do:

In [17]:

```
# FB Prophet
# imports
from fbprophet import Prophet

# make a copy of the dataframe for Prophet transformations
prophet_df = make_copy_df(train_df, '9920-2')

# rename variables (prophet requires the variable names in the time seri
es to be
# y for target and ds for Datetime)
rename_columns(prophet_df, '9920-2')

# instantiate model instance and set the uncertainty interval to 95%
# (the Prophet default is 80%)
my_model = Prophet(interval_width=0.95, weekly_seasonality=True)

# fit
my_model.fit(prophet_df)

# forecast for a year
future_dates = my_model.make_future_dataframe(periods=52, freq='W')
forecast = my_model.predict(future_dates)

# plot
forecast_slice = create_fb_forecast(forecast, 240, 292)
plot_time_series(train_df, test_df, '9920-2', forecast_slice, 'yhat', 'P
rophet Forecast')

RMSE(test_df, '9920-2', forecast_slice, 'yhat')
```

```
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```

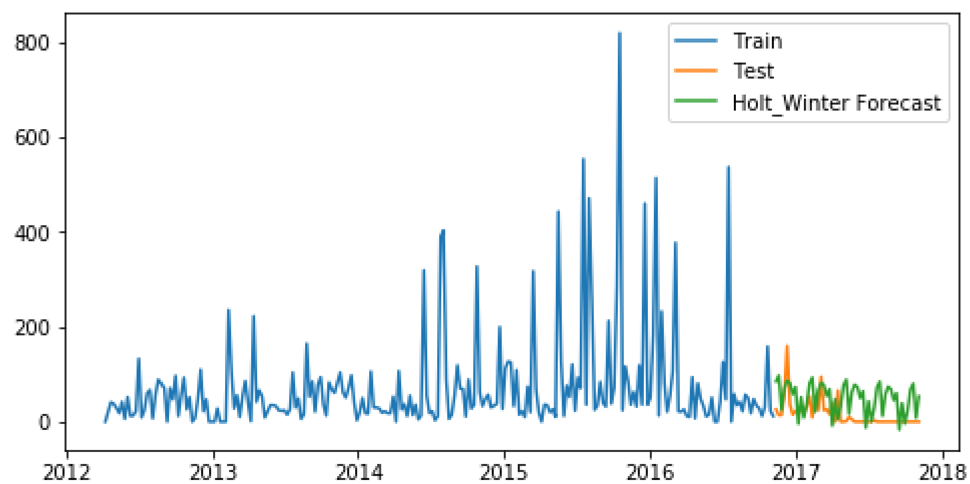


Out[17]: 104.04466571337738

104.04 is 50 percent higher than a simple averaging method! How about Holt-Winters?

In [18]:

```
# Holt-Winters with additive trend and seasonality, no trend damping, se
asonal periods=12
from statsmodels.tsa.api import ExponentialSmoothing

fit1 = ExponentialSmoothing(np.asarray(train_df['9920-2']), seasonal_per
iods=12,
                            trend='additive', seasonal='additive')
                            .fit(smoothing_level=0.51,smoothing_slope=0.
015,
                                 smoothing_seasonal=0.1)

y_hat['Holt_Winter'] = fit1.forecast(len(test_df))

# plot
plot_time_series(train_df, test_df, '9920-2', y_hat, 'Holt_Winter', 'Hol
t_Winter Forecast')

# calculate RMSE
RMSE(test_df,'9920-2', y_hat, 'Holt_Winter')
```



Out[18]: 51.35384040458961

At 51.35, Holt-Winters does much better (and this is only with some manual tuning based on very simple guidelines).

Given that:

- the accuracy of forecasts is highly dependent on the underlying distribution of data,
- 2 of the 3 forecasting methods require a great deal of parameter tuning, and
- there are 1,833 products to forecast altogether,
  it would be best to group together products that move similarly in time and can be forecasted using the same methods.

That's what k-means clustering with dynamic time warping can accomplish. However, before doing that, let's see if some of the products could be eliminated!

4. Segment Products

In [19]:
```
# make a list of products
product_SKUs = list(train_df.columns.values)
```

First I dropped products that are no longer active (defined as products that have not moved after a specified date).

In [20]:
```
# not active
non_active = identify_non_active(train_df, product_SKUs, 2015, 11, 4)
```

Then I dropped products that are new (, as they should be forecasted based on history

In [21]:
```
# new products
new_products = identify_new_product(train_df, product_SKUs, 2015, 11, 4)
```

Altogether, non active and new products accounted for 610 products. Next, I looked for products with intermittent demand (they should be forecasted using Croston's Method). Intermittent demand was defined as no demand in certain amount of weeks (here, I picked 4).

In [22]:
```
# intermittent demand
intermittent_demand = identify_intermittent_product(train_df, product_SK
Us, non_active, 2015, 11, 4, 4)
```

Interestingly, there were no products with intermittent demand. There were some products that are ordered in such small quantities, that spending a great deal of time on manually tuning their forecasting models may not be the best use of time. I set aside items that do not sell more than 30 units.

In [23]:
```
# products with very little demand
minute_demand = identify_minute_demand(train_df, product_SKUs, 30)
```

Finally, some products are ordered in large quantities (over 3,000 units) and then repackaged into sets.

In [24]:
```
# repackaged products
repackage_product = identify_repackage_product(train_df, product_SKUs, 3
000)
```

After dropping the above categories, I was left with 878 products, all ready for clustering with dynamic time warping!

```
In [25]: # create leftover dataset for dynamic time warping
products = make_remainder_dataframe(train_df, product_SKUs, non_active,
new_products, minute_demand, repackage_product)
```

5. Cluster Products

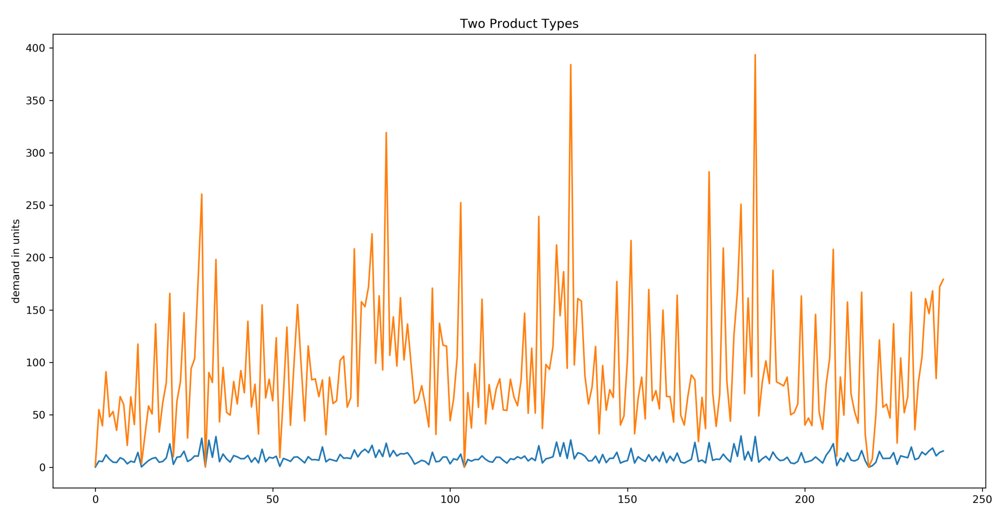
To cluster together similar time series, I used k-means clustering with dynamic time warping. Dynamic time warping is a measure that finds the best alignment between two time series. (For more on the method, see Alex Minnaar's excellent blog post: http://alexminnaar.com/time-series-classification-and-clustering-with-python.html (http://alexminnaar.com/time-series-classification-and-clustering-with-python.html))

```
In [ ]: # prepare products df for timewarping
data_arr = prep_dataframe_for_warping(products)

# k-means clustering with k=2
cl_obj=ts_cluster(2)
cl_obj.k_means_clust(data_arr,10,4, progress=False)
```

Why did I only make 2 clusters (k=2)? Based on the Calinski-Harabasz index, 2 clusters were more distinctive than 3, 4, 5, 6, 7 or 8.

```
In [31]: # plot two basic product types
#cl_obj.plot_centroids()

from IPython.display import Image
Image("product_clusters.png")
```

Out[31]:



The resulting plot shows two distinct time series types: one with modest fluctations and one with dramatic changes in demand.

6. Evaluate Clusters

Now I have 6 groups of products: not active, new products, minute demand, products for repackaging, and the two clusters (modest fluctuations and seasonal swings). To make sure these groups are truly meaningful,

In [ ]:
```
#
```

In [28]:
```
# save dictionary containing assignments
assigned_data = cl_obj.get_assignments()

# assign products to clusters
from product_segmentation_functions import assign_products

product_type0 = assign_products(products.T, assigned_data, 0)
product_type1 = assign_products(products.T, assigned_data, 1)
product_type2 = assign_products(products.T, assigned_data, 2)
product_type3 = assign_products(products.T, assigned_data, 3)
product_type4 = assign_products(products.T, assigned_data, 4)
product_type5 = assign_products(products.T, assigned_data, 5)
len(product_type0), len(product_type1), len(product_type2), len(product_
type3), len(product_type4), len(product_type5)
```

Out[28]:
```
(47, 309, 82, 70, 225, 139)
```

In [29]:
```
# save centroids
centroid0, centroid1, centroid2, centroid3, centroid4, centroid5 = cl_ob
j.get_centroids()
```