Time Series Retail Sales Forecasting with DeepAR

Forecasting is a central problem in many businesses. In the retail industry probabilistic forecasts are important for inventory management to ensure that there is enough product on-hand to meet the seasonal spikes in sales for differnet categories of products.

Most forecasting methods have been developed in the setting of forecasting individual time series where model parameters are independently estimated from past observations for each given time series. Today retailers are faced with forecasting demand on potentially millions of time series for different products across their catalog. Retailers also face cold start problems where they need to forecast for a new item that has no existing time series data.

In this notebook we will see how the DeepAR forecasting algorithm (https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html) can help retailers solve these business problems. Using a generated dataset of daily sales for a set of clothing products we will see how DeepAR can learn jointly across the related time series to capture complex group dependent behavior at a categorical level. Finally we will see how this learned categorical behavior can be used to forecast for products with existing time series as well as new "cold" products with no existing data.

For a more rigorous explanation of the DeepAR algorithm check out the DeepAR white paper (https://pdfs.semanticscholar.org/4eeb/e0d12aefeedf3ca85256bc8aa3b4292d47d9.pdf).

Importing modules and defining helper functions

The modules and helper functions below will be used throughout this lab to generate the dataset and convert data between formats. You do not need to read over and understand each function to proceed with the lab but comments have been added for the inquisitive to reference. Run the below cell before proceeding.

In [ ]:
```
from collections import defaultdict
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import sagemaker
import datetime
import tempfile
import random
import boto3
import copy
import uuid
import json
import os

%matplotlib inline


def make_product(categories, num_years):
    """ Creates a random product with a unique product id from the list of pot
ential category and subcategory pairings

    Args:
        categories (dict): Dictionary containing mappings of categories to lis
ts of subcategories.
        num_years (int): Range of years from today's date to pick start date f
or time series data from.
    Returns:
        (dict)
        Dictionary containing product information for the generated product.
    """
    product_id = str(uuid.uuid4().fields[-1])
    cat, subcats = random.choice(list(categories.items()))
    subcat = random.choice(subcats)
    start_date = datetime.date.today() - datetime.timedelta(random.randint(num
_years/2, num_years)*365)
    return {'product_id': product_id,
            'category': cat,
            'subcategory': subcat,
            'start_date': start_date}

def make_weights(categories, num_years):
    """ Creates normalized weights in the interval 0-1 for each category for e
ach month of each year in the input range of years.

    Args:
        categories (dict): Dictionary containing mappings of categories to lis
ts of subcategories.
        num_years (int): Range of years from today's date to pick start date f
or time series data from.
    Returns:
        (dict)
        Dictionary containing weights for each category for each month of each
 year in the input range of years.
    """
    end_year = datetime.date.today().year
    results = defaultdict(lambda: defaultdict(dict))
    for year in range(end_year-num_years, end_year+1):
        for month in range(1, 13):
            rands = np.random.random(size=len(categories))
            weights = rands / rands.sum()
            weight_index = 0
            for category in categories:
                results[category][year][month] = weights[weight_index]
                weight_index += 1
    return results

def make_category_weights(categories, num_years):
    """ Creates random weights for each category and subcategory to further au
gment seasonality within these groupings
    Weights are randomly created for every category and subcategory for each y
ear and month
    At the category and subcategory levels the weights are normalized to sum t
o 1

    Args:
        categories (dict): Dictionary containing mappings of categories to lis
ts of subcategories.
        num_years (int): Range of years from today's date to pick start date f
or time series data from.
    Returns:
        (dict)
        Dictionary containing weights for each category for each month of each
 year in the input range of years.
    """
    category_weights = make_weights(list(categories.keys()), num_years) # Crea
te weights for the top level categories
    for category, subcategories in categories.items(): # Add weights for the c
orresponding subcategories
        subcat_weights = make_weights(subcategories, num_years)
        for subcat, weights in subcat_weights.items():
            category_weights[category][subcat] = weights
    return category_weights

def make_time_point_value(date, cat, subcat, seasonality, weights):
    """ Creates daily time point sales value for a product based on baseline s
easonality and product weights

    Args:
        date (datetime): Date to generate time point value for.
        cat (string): Category of product.
        subcat (string): Subcategory of product.
        seasonality (dict): Dictionary containing details about seasonality of
 sales for products.
        weights (dict): Dictionary containing weights for each category and su
bcategory for each month of each year.
    Returns:
        (int)
        Time point sales value for a given product on a given day.
    """
    month = date.month
    year = date.year
    baseline = seasonality[month]
    noise = np.random.normal(scale=0.1)
    cat_weight = weights[cat][year][month]
    subcat_weight = weights[cat][subcat][year][month]
    value = (baseline + noise*baseline)*cat_weight*subcat_weight
    return int(value)

def make_data_for_product(product, seasonality, weights):
    """ Creates DataFrame containing time point sales values for input product
 based on baseline seasonality and product weights.

    Args:
        product (dict): Dictionary containing product information for a single
 product.
        seasonality (dict): Dictionary containing details about seasonality of
 sales for products.
        weights (dict): Dictionary containing weights for each category and su
bcategory for each month of each year.
    Returns:
        (DataFrame)
        DataFrame containing time point sales values for a given product.
    """
    cat = product['category']
    subcat = product['subcategory']
    today = datetime.date.today()
    start_date = product['start_date']
    delta = today - start_date
    data = []
    for i in range(delta.days + 1):
        local_product = product.copy()
        local_product.pop('start_date', None)
        date = start_date + datetime.timedelta(days=i)
        local_product['date'] = date
        local_product['sales'] = make_time_point_value(date, cat, subcat, seas
onality, weights)
        data.append(local_product)
    return pd.DataFrame(data)

def make_data_for_products(products, seasonality, weights):
    """ Creates DataFrame containing time point sales values for all input pro
ducts based on baseline return seasonality and product weights.

    Args:
        products (list): List of dictionaries containing product information f
or all products.
        seasonality (dict): Dictionary containing details about seasonality of
 sales for products.
        weights (dict): Dictionary containing weights for each category and su
bcategory for each month of each year.
    Returns:
        (DataFrame)
        DataFrame containing time series sales values for a given product.
    """
    df = pd.concat([make_data_for_product(product, seasonality, weights) for p
roduct in products])
    df = df[['product_id', 'date', 'category', 'subcategory', 'sales']]
    df['product_id'] = df['product_id'].apply(str)
    df['date'] = df['date'].apply(lambda t: pd.to_datetime(t, format='%Y-%m-%d
'))
    return df

def plot_dataset_ts(df, condition):
    """ Plots summed time point sales values of data at monthly granularity gr
ouped by condition.

    Args:
        df (DataFrame): DataFrame containing time point sales data for product
s.
        condition (str): Condition to group DataFrame by ('category' or 'subca
tegory').
    Returns:
        (None)
    """
    data = df.copy()
    data = data.groupby(['date', condition]).sum().unstack().fillna(0.0)
    data = data.groupby([(data.index.year.rename('year')),(data.index.month.re
name('month'))]).sum()
    fig, ax = plt.subplots(figsize=(15,15))
    data.plot(ax=ax)

def ts_to_json_obj(series, cat):
    """ Converts input time point DataFrame into a JSON object in DeepAR forma
t

    Args:
        series (DataFrame): DataFrame containing time point sales data for a p
roduct.
        cat (str): Label encoded category for product.
    Returns:
        (dict)
        Dictionary containing time point sales data converted to DeepAR forma
t.
    """
    ts = series.copy() # Make a local copy of series so as not to modify origi
nal DataFrame
    ts = ts.rename('sales') # Rename series
    ts = ts.reset_index() # Input series is indexed by date, re-index to make
 df with date as column
    start_date_index = ts['sales'].nonzero()[0][0] # Pull out index of first n
on-zero day of return values
    ts = ts.iloc[start_date_index:].reset_index(drop=True) # Truncate leading
 0 return rows from time series and reset index to 0
    json_obj = {"start": str(ts['date'][0]), "cat": int(cat), "target": ts['sa
les'].astype(int).tolist()}
    return json_obj

def transform_to_json_objs(df, le):
    """ Converts input DataFrame of all time point sales values for all produc
ts into list of JSON objects in DeepAR format for each product.

    Args:
        df (DataFrame): DataFrame containing time point sales data for all pro
ducts.
        le (LabelEncoder): SciKit LabelEncoder fit on product categories and s
ubcategories to label encode these values with.
```

```
    Returns:
        (dict)
        Dictionary containing time point sales data converted to DeepAR format
 for all products.
    """
    cats_df = df[['product_id', 'category', 'subcategory']].drop_duplicates().
set_index('product_id')
    ts_df = df.groupby(['date', 'product_id']).sum().unstack().fillna(0.0)
    ts_df.columns = ts_df.columns.droplevel() # Drop unneeded multi-index leve
l
    json_objs = []
    for column in list(ts_df.columns.values):
        cat = cats_df.loc[column, 'category']
        subcat = cats_df.loc[column, 'subcategory']
        num_cat = le.transform([cat])[0]
        num_subcat = le.transform([subcat])[0]
        json_objs.append(ts_to_json_obj(ts_df.loc[:, column], num_cat))
        json_objs.append(ts_to_json_obj(ts_df.loc[:, column], num_subcat))
    return json_objs

def make_train_set(json_objs, prediction_length):
    """ Creates a training set from an input dataset by removing prediction_le
ngth number of time points from each time series.

    Args:
        json_objs (list): List of JSON objects in DeepAR format.
        prediction_length (int): Number of time points to remove from each tim
e series.
    Returns:
        (list)
        List of JSON objects in DeepAR format where each object has had predic
tion_length number of time points removed.
    """
    objs = copy.deepcopy(json_objs)
    for obj in objs:
        obj['target'] = obj['target'][:-prediction_length]
    return objs

def write_json_to_file(json_objs, channel):
    """ Writes list of JSON objects to file in JSON Lines format encoded as ut
f-8.
    Output file is named after channel.

    Args:
        json_objs (list): List of JSON objects in DeepAR format.
        channel (str): Channel that JSON Lines data will be used for ('train'
 or 'test').
    Returns:
        (str)
        Path of file data was written to.
    """
    file_name = '{}.json'.format(channel)
    file_path = os.path.join(os.getcwd(), file_name)
    with open(file_path, 'wb') as f:
        for obj in json_objs:
            line = json.dumps(obj) + '\n'
            line = line.encode('utf-8')
            f.write(line)
        f.seek(0)
        f.flush()
    return file_path
```

Generating the dataset

For this lab we will be working with a dataset composed of category + subcategory pairings. The cell below generates a catalog of products using possible category + subcategory pairings from the `categories` dictionary of categories and subcategories. Each product is given a unique product ID to be identified by.

Next each product category and subcategory is given a random normalized weight for each month of each year in the time series range. These weights are used to simulate seasonality in different product categories and subcategories, such as boots being more popular in winter than in summer.

Lastly, time series sales data is generated for each product using the the product catalog, the normalized weights, and the `seasonality` dictionary, which is used to augment the sales values roughly around what a typical retailers yearly sales trends might look like. To simulate a store adding products over time each product is randomly assigned a "start" date in the time series interval in which it begins to have non-zero sales values.

In [ ]:
```
# Categories and subcategories to generate products with
# Each product id is generated with a random category and subcategory from tha
t category's options
categories = {
    'shoe': ['sneaker', 'boot', 'slipper'],
    'outerwear': ['coat', 'jacket', 'shell'],
    'top': ['shirt', 't-shirt', 'sweater', 'knit'],
    'bottom': ['skirt', 'pant', 'short', 'leggings'],
    'accessories': ['belt', 'tie', 'scarf', 'hat', 'brooche']
}

# Seasonality for time series data to be generated around
seasonality = {
    1: 500,
    2: 400,
    3: 200,
    4: 100,
    5: 40,
    6: 80,
    7: 150,
    8: 180,
    9: 140,
    10: 240,
    11: 100,
    12: 400
}

n_products = 100 # Number of products to generate. Increase this to generate m
ore individual product data

n_years = 4 # Number of years in past from current date to generate date for.
 Increase this to generate longer time series for each product

products = [make_product(categories, n_years) for i in range(n_products)]

category_weights = make_category_weights(categories, n_years)

product_data = make_data_for_products(products, seasonality, category_weights)

print(product_data.head())
```

Visualizing the data

Now that we have generated our dataset let's take a look at the time series sales values over time at the category and subcategory levels across products. This can be done by running the code in the cells below.

In [ ]:
```
# Visualize monthly sales of products at category level
plot_dataset_ts(product_data, 'category')
```

```
In [ ]: # Visualize monthly sales of products at subcategory level
        plot_dataset_ts(product_data, 'subcategory')
```

Preparing the dataset for DeepAR

Now that we've generated our raw dataset we need to wrangle it into the format and encoding expected by DeepAR.

DeepAR expects data (for training or inference) in JSON Lines (http://jsonlines.org/) or Parquet (https://parquet.apache.org/) format. For this lab we'll be working with JSON Lines.

In JSON Lines format each line is a seperate JSON object representing a time series for a single product. DeepAR expects each JSON object to have a `start` key, a string or datetime object representing the time the time series data starts at, and a `target` key, whose value is an array of floats (or integers) that represent the time series variable’s values. Additionally each JSON object can include a `cat` key, which is an integer that encodes the categorical grouping that record’s time series is a member of. This allows the model to learn typical behavior for that group and can increase accuracy.

Currently our product sales data is in a pandas DataFrame and is not yet converted into JSON objects with time series grouped by product ID as the DeepAR algorithm expects for training. In the following cells we'll wrangle the data into the required format.

To start, our categorical grouping values (product category and subcategory) are currently strings but DeepAR expects integers for these values. Let's encode our category and subcategory values to integers to use for our `cat` values.

```
In [ ]: uniq_cats = product_data.category.unique().tolist()
        uniq_subcats = product_data.subcategory.unique().tolist()

        le = LabelEncoder().fit((uniq_cats + uniq_subcats))
        print(le.classes_)
```

As you can see we've encoded class labels for both the product subcategories and the categories. Later we will see how this allows us to make forecasting predictions for existing product subcategories at a granular level while also allowing us to generalize our predictions to new subcategories that we might wish to introduce by predicting at the higher category level.

Next we set the epochs, frequency, prediction length, context length, cardinality, and embedding dimension hyperparameters we wish to train our DeepAR model on.

Epochs specifies the maximum number of times to pass over the data when training. For this lab we set the value to '50' to minimize training job runtime due to time constraints.

Frequency specifies the granularity of the time series in the dataset. In this case out time series values correspond to daily sales results for each product so our frequency is 'D' for daily. Other possible values are 'min' (every minute), 'H' (hourly), 'W' (weekly), and 'M' (monthly).

Prediction length controls the number of time steps (based off the unit of frequency) that the model is trained to predict, also called the forecast horizon. Our prediction length is set to '28' to predict roughly a month's worth of days into the future for forecast requests submitted to the trained DeepAR model.

Context length controls the the number of time points that the model gets to see before making a prediction. The value for this parameter should be about the same as the prediction_length. The model also receives lagged inputs from the target, so context_length can be much smaller than typical seasonalities. For example, a daily time series can have yearly seasonality. The model automatically includes a lag of one year, so the context length can be shorter than a year. The lag values that the model picks depend on the frequency of the time series. For example, lag values for daily frequency are the previous week, 2 weeks, 3 weeks, 4 weeks, and year. Here we set it to '28' to predict based off roughly the last month's worth of time points.

Cardinality is only required when specifying categorical level groupings in your data, and controls the number of unique categories found in your dataset. The label encoder we trained already has a set of the unique classes found in our dataset so we can use the length of this set as our cardinality.

Embedding dimension is only required when specifying categorical level groupings in your data, and specifies the size of the embedding vector the algorithm can learn to capture the common properties of all the time series within a categorical level grouping. Because of the small size of our dataset we leave set this to '2'. For larger datasets with more unique categories this value commonly ranges from 10-100.

Values for the epochs, frequency, prediction length, and context length hyperparameters are required when training a DeepAR model, but you can also configure other optional hyperparameters to further tune your model. For a more exhaustive list of all the different DeepAR hyperparameters you can tune check out the DeepAR documentation (https://docs.aws.amazon.com/sagemaker/latest/dg/deepar_hyperparameters.html).

In [ ]:
```
epochs = '50'
freq = 'D'
prediction_length = '28'
context_length = '28'
cardinality = str(len(le.classes_))
embedding_dimension = '2'

hyperparameters = {
    "epochs": epochs,
    "time_freq": freq,
    "context_length": context_length,
    "prediction_length": prediction_length,
    "cardinality": cardinality,
    "embedding_dimension": embedding_dimension
}
```

Next we transform our pandas DataFrame into JSON objects for each product ID as expected by the DeepAR algorithm. This transformation is implemented in the `transform_to_json_objs` helper function created at the start of this notebook for reference.

The DeepAR algorithm has an optional test channel for training that can be used to calculate accuracy metrics for the model after training, such as RMSE and quantile loss. For our test set we'll use the full time series for each product. For our training set we'll use the full time series minus our prediction length worth of time points. The loss for our model will then be calculated by how well our model predicts these missing time points in comparison to the ground truth values.

In [ ]:
```
json_objs = transform_to_json_objs(product_data, le)
train_set = make_train_set(json_objs, int(prediction_length))
test_set = json_objs
```

Lastly we upload our data to S3 so it can be accessed by the DeepAR algorithm during the training job. The SageMaker `Session` class has a convenient method, `upload_data` (http://sagemaker.readthedocs.io/en/latest/session.html#sagemaker.session.Session.upload_data), to help us upload our training and testing data to S3 while also returning S3 URIs to pass to our training job.

In [ ]:
```
your_username = '<YOUR_USERNAME_HERE>'

resource_prefix = 'deepar-retail-{}'.format(your_username)

sagemaker_session = sagemaker.Session()

train_file = write_json_to_file(train_set, 'train')
test_file = write_json_to_file(test_set, 'test')

train_location = sagemaker_session.upload_data(train_file, key_prefix=resource
_prefix)
test_location = sagemaker_session.upload_data(test_file, key_prefix=resource_p
refix)
```

Training and deploying a DeepAR model

DeepAR is one of SageMaker's built-in algorithms (https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html) so a container for training and hosting the algorithm is provided by the service. Here we select the container corresponding to the region we're running our notebook in and specify the output path in S3 for our SageMaker training job to output trained model artifacts.

In [ ]:
```
containers = {
    'us-east-1': '522234722520.dkr.ecr.us-east-1.amazonaws.com/forecasting-dee
par:latest',
    'us-east-2': '566113047672.dkr.ecr.us-east-2.amazonaws.com/forecasting-dee
par:latest',
    'us-west-2': '156387875391.dkr.ecr.us-west-2.amazonaws.com/forecasting-dee
par:latest',
    'eu-west-1': '224300973850.dkr.ecr.eu-west-1.amazonaws.com/forecasting-dee
par:latest'
}

role = sagemaker.get_execution_role()

image_name = containers[boto3.Session().region_name]

bucket = sagemaker_session.default_bucket()

s3_output_path = "{}/{}/output".format(bucket, resource_prefix)
```

Next we create an Estimator (http://sagemaker.readthedocs.io/en/latest/estimators.html) object for our DeepAR model, which is a high level interface for training and deploying SageMaker models programatically, and configure it with the hyperparameters we set earlier.

In [ ]:
```
estimator = sagemaker.estimator.Estimator(
    sagemaker_session=sagemaker_session,
    image_name=image_name,
    role=role,
    train_instance_count=1,
    train_instance_type='ml.c5.2xlarge',
    base_job_name=resource_prefix,
    output_path="s3://" + s3_output_path
)

estimator.set_hyperparameters(**hyperparameters)
```

Next we specify the input data channels using the S3 URIs returned earlier when we uploaded our train and test datasets to S3 and fit the DeepAR model. Note that this cell may take 5-15 minutes to run to run to completion.

In [ ]:
```
data_channels = {
    "train": train_location,
    "test": test_location
}

estimator.fit(inputs=data_channels)
```

Lastly we deploy our trained model to a SageMaker model endpoint where we can leverage it for forecasting predictions.

In [ ]:
```
predictor = estimator.deploy(
    initial_instance_count=1,
    instance_type='ml.m4.xlarge',
    endpoint_name=resource_prefix,
    content_type="application/json"
)
```

Leveraging deployed DeepAR model endpoint for forecasting predictions

Now that we have a SageMaker model endpoint deployed let's use it to make some predictions. DeepAR model endpoints expect requests in JSON format with the following keys:

`instances` - A list of the time series that should be forecast by the model. Each entry in the list should be a JSON object in the same format that DeepAR expects for training.

`configuration` - Optional. A dictionary of configuration information for the type of response desired by the request.

Within `configuration` the following keys can be configured:

`num_samples` - An integer specifying the number of sample paths that the model generates when making a probabilistic prediction.

`output_types` - A list specifying the type of response. `mean` returns a single value for each time point which is the average of `num_samples` samples generated by the model. `quantiles` looks at the list of `num_samples` generated by the model and attempts to generate quantile estimates for each time point based on these values. `samples` returns the list of `num_samples` for each time point in the prediction length.

`quantiles` - If your specify `quantiles` as one of your desired output types then this list lets you control which quantiles estimates are generated and returned for in the response.

Below is an example of what a JSON query to a DeepAR model endpoint might look like.

```
{
 "instances": [
  { "start": "2009-11-01 00:00:00", "target": [4.0, 10.0, 50.0, 100.0, 113.0], "cat": 0},
  { "start": "2012-01-30", "target": [1.0], "cat": 2 },
  { "start": "1999-01-30", "target": [2.0, 1.0], "cat": 1 }
 ],
 "configuration": {
  "num_samples": 50,
  "output_types": ["mean", "quantiles", "samples"],
  "quantiles": ["0.5", "0.9"]
 }
}
```

In the cells below you can try predictions for a sample item with existing time series taken from our testing dataset as well as a prediction for a new "cold" item with no existing time series.

In [ ]:
```
# Prediction for item with existing time series sales history

start_date = datetime.date.today() - datetime.timedelta(days=int(prediction_le
ngth))

target = test_set[0]['target'][-int(prediction_length):]

cat = test_set[0]['cat']

request_json = {
 "instances": [
  { "start": str(start_date), "target": target, "cat": cat}
 ],
 "configuration": {
  "num_samples": 10,
  "output_types": ["quantiles"],
  "quantiles": ["0.9"]
 }
}

payload = json.dumps(request_json).encode('utf-8')

response = predictor.predict(payload)

print(response)
```

In [ ]:
```
# Cold-start prediction for new item that has no time series sales history

product_category = 'shoe'

cat = int(le.transform([product_category])[0])

request_json = {
 "instances": [
  {"start": "2019-06-12 00:00:00", "target": [], "cat": cat} # No target value
s because this is a new product with no existing time series sales data
 ],
 "configuration": {
  "num_samples": 10,
  "output_types": ["quantiles"],
  "quantiles": ["0.9"]
 }
}

payload = json.dumps(request_json).encode('utf-8')

response = predictor.predict(payload)

print(response)
```

Optional cleanup

When you're done with the endpoint, you'll want to clean it up.

In [ ]:
```
sess.delete_endpoint(predictor.endpoint)
```