S4: Time Series Analysis, Time Series Preprocessing and Modeling with ML

Phanuphat Srisukhawasu (Oad)

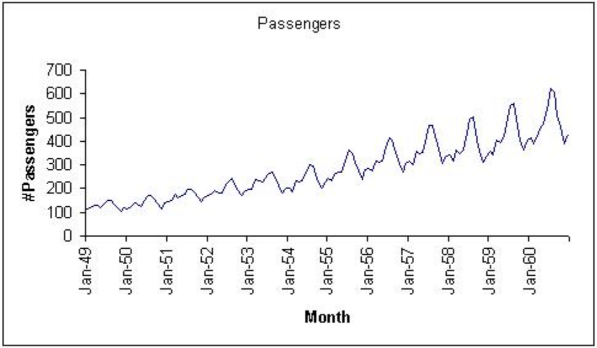
Machine Learning Department
NUS Fintech Society

Week 6 (16 Sep 2024 - 20 Sep 2024)

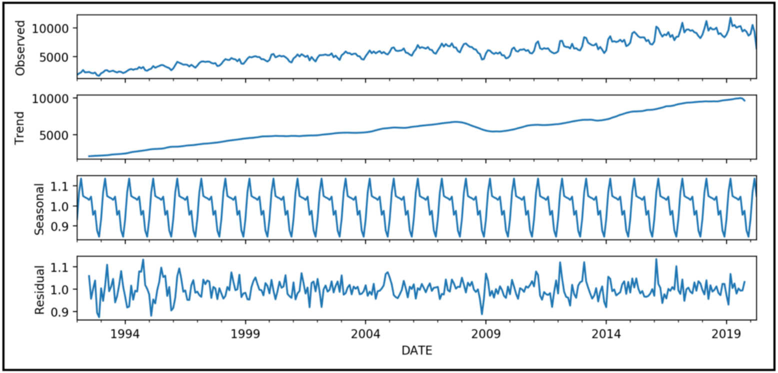
Session Outline

Time Series Data in Simple Words

- **Time series** are data points indexed in an order of time.
- It could be sampled as frequent as we want: every second, every minute, hourly, daily, monthly, yearly, etc.



Trend, Seasonality, and Noise



Note: **Residual** (a superset of noises) is what we are left with, after analyzing the other components.

Loading Time Series Data

- As we have a CSV file containing time series data, we could use `pd.read_csv()` to import your data as usual.
- But when we inspect the data, we generally see that the data type of the data and time column is **a string (object.)**

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1226 entries, 0 to 1225
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Date           1226 non-null   object
 1   Open           1226 non-null   int64
 2   High           1226 non-null   int64
 3   Low            1226 non-null   int64
 4   Close          1226 non-null   int64
 5   Volume         1226 non-null   int64
 6   Stock Trading  1226 non-null   int64
dtypes: int64(6), object(1)
memory usage: 67.2+ KB
```

Converting Date Column to `Datetime` Object

- In general, we will process the dataframe in such way that the `Date` column is a `Datetime` object instead of a string.
- To accomplish this, we usually use `df['Date'] = pd.to_datetime(df['Date'])`.
- We also usually do `df = df.set_index('Date')`. This will make the processing more convenient.

Date	Open	High	Low	Close	Volume	Stock Trading
2016-12-30	42120	42330	41700	41830	610000	25628028000
2016-12-29	43000	43220	42540	42660	448400	19188227000
2016-12-28	43940	43970	43270	43270	339900	14780670000
2016-12-27	43140	43700	43140	43620	400100	17427993000
2016-12-26	43310	43660	43090	43340	358200	15547803000

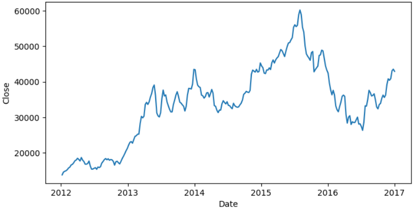
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1226 entries, 2016-12-30 to 2012-01-04
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Open           1226 non-null   int64
 1   High           1226 non-null   int64
 2   Low            1226 non-null   int64
 3   Close          1226 non-null   int64
 4   Volume         1226 non-null   int64
 5   Stock Trading  1226 non-null   int64
dtypes: int64(6)
memory usage: 67.0 KB
```

Filling the Missing Data with Forward Fill

- For time series, a simple method to fill missing values at time t_0 is to use the value from the closest previous time $t < t_0$.
- For this example dataset, we already know that we don't have any missing data. But if we do, we could use `df = df.ffill()`. After that, we could use `sns.lineplot()` from Seaborn to visualize the data.

Time Series Downsampling

- In this example, we have a data recorded in a **daily basis.** In some cases, we may want to analyze the data in a weekly, monthly, or yearly basis.
- In Pandas, we could resample the data as frequent as we want: `df = df.resample(<FREQ>).mean()`.



Note: The above plot is generated from sampling the data using `'W'` as `<FREQ>` and calculate the average. This will resample the data **weekly.** The plot is more smoothed out compared to the previous. For other frequencies, you may refer to this link.

Lag Features (1)

- To use machine learning, we need inputs or features. What are the features for time series?
- It does not make any senses to use the time index as the features, as the date should not be correlated to the values.
- In time series forecasting, we want to use values from the past to predict the future values.
- Using this principle, we can create features from the past values known as **lag features**.
- For example, we can use `df['lag1'] = df['Close'].shift(1)`. This command creates a new column `'lag1'` that shows the closing price from one time step earlier.

Lag Features (2)

- In general, we can pass in different values of steps other than `1` (i.e. `2,3,...`) to create lagging from multiple time steps.
- This allow us to have multiple features to train the model.

Date	Open	High	Low	Close	Volume	Stock Trading	lag1	lag2	lag3
2012-01-08	13920.0	13973.333333	13696.666667	13790.0	6.120333e+05	8.462075e+09	NaN	NaN	NaN
2012-01-15	14377.5	14630.000000	14230.000000	14567.5	1.019975e+06	1.480215e+10	13790.0	NaN	NaN
2012-01-22	14800.0	14908.000000	14702.000000	14794.0	5.475800e+05	8.120263e+09	14567.5	13790.0	NaN
2012-01-29	14832.0	15010.000000	14772.000000	14948.0	5.902400e+05	8.795512e+09	14794.0	14567.5	13790.0
2012-02-05	15212.0	15330.000000	15090.000000	15256.0	5.479400e+05	8.341651e+09	14948.0	14794.0	14567.5
2012-02-12	15742.0	15846.000000	15586.000000	15722.0	6.961400e+05	1.095340e+10	15256.0	14948.0	14794.0
2012-02-19	15826.0	16064.000000	15718.000000	15974.0	7.001000e+05	1.121111e+10	15722.0	15256.0	14948.0

Note: The numbers of lagging features to use could be one of the **hyperparameters** to tune.

ARIMA: AR (Autoregressive) + I (Integrated) + MA (Moving Average)

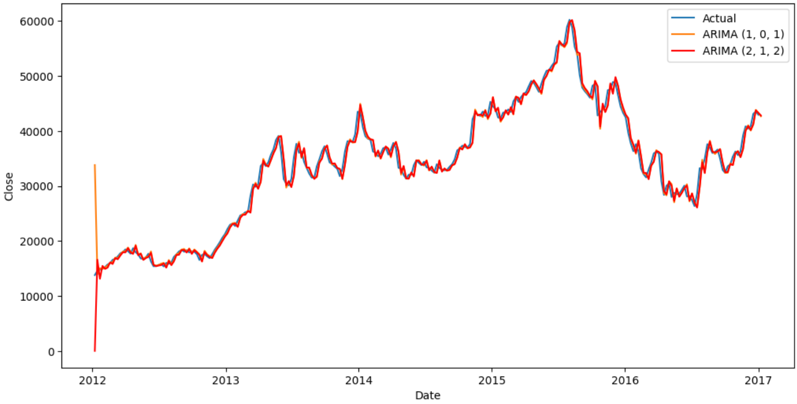
- **ARIMA** is a statistical model used to forecast the time series based on the past values.
- Sometimes, we could solve the simpler problems using just ARIMA and without any machine learning!
- In simple words, each part of ARIMA play the following roles:
  - **AR** = Previous values (lag features,)
  - **I** = Adjustments to make data stable (removing trends,)
  - **MA** = Adjustments to smooth out the noises.
- If you want to explore the mathematics behind ARIMA, feel free to visit this site (click on the formula XD):

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q} + \epsilon_t$$

Implementing ARIMA in Python

- **ARIMA** is in a library called `statsmodels`. We can import it using `from statsmodels.tsa.arima.model import ARIMA`.
- We can **fit the model** to the data using: `model = ARIMA(df['Close'], order=(p, d, q)).fit()`.
  - `p` is the number of lag features to use (i.e. `p = 2` will use only the values at step $t-1$ and $t-2$.)
  - `d` is the number of times we differenced the time series $(x(t) - x(t-1),)$ to make it stationary. We start with $d = 0$.
  - `q` is the number lagged forecast errors to use.
- After fitting the model, we could **forecast** the time series values by assigning `predictions = model.predict(start=0, end=len(df)-1)`.

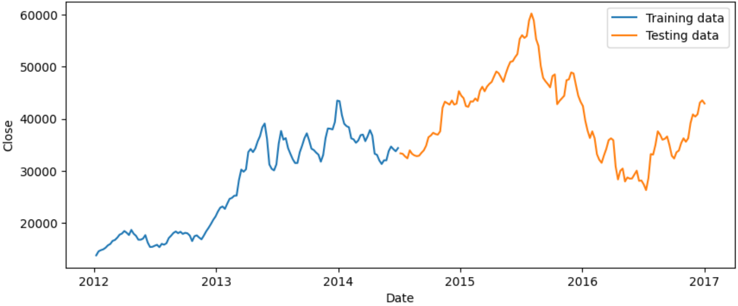
Example Results of ARIMA



Note: Sometimes, it is difficult to judge the **performance** by eyes. In the next section, we will introduce numerical metrics to evaluate the models.

Train-Test Split on Time Series

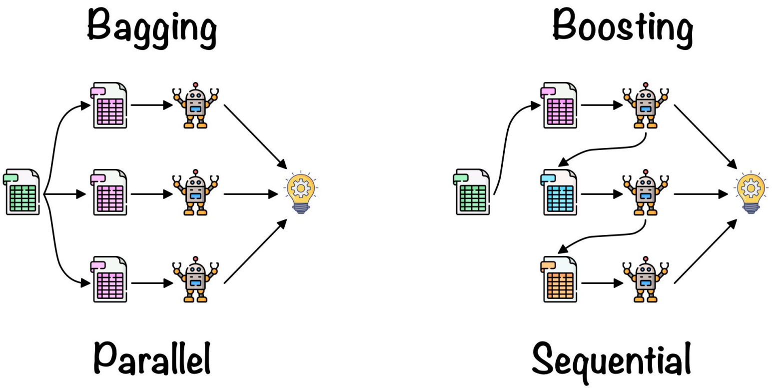
- When it comes to time series, we **cannot use random subsets** of data as the training and testing sets.
- Obviously, we want to train the model on the past data to forecast the future. Therefore, this is what we usually do:



More Advanced Machine Learning Models

- Last week, you have already learned about **Decision Trees.** In this session, we are introducing more advanced tree-based models which are constructed by ensembling multiple trees.
- With that being said, we shall introduce two primary methods for ensembling multiple trees:
  - **Bagging**: We construct multiple decision trees using random subsets of data and features. The final model decision is the **voting** (**average prediction**) from all of the trees. The model that use this ensembling method is **Random Forest.**
  - **Boosting**: We construct new trees from the previous trees. The early trees are called the **weak learner.** The errors made by them are taken into account as the weights of importance to train the subsequent trees, creating **strong learner** at the end. The models that use this ensembling method are **XGBoost, LightGBM**, and their variations.

Bagging and Boosting Illustration



Note: you can read more about the mathematical details of each model here.

Implementing Random Forests & XGBoost in Python

- Like most models, Random Forests can also be accessed through `Scikit-learn` library. We can import it using `from sklearn.ensemble import RandomForestClassifier` or `RandomForestRegressor`.
- However, XGBoost **is not** inside `Scikit-learn`. We can import it from the library `xgboost` where we could use `XGBClassifier` and `XGBRegressor`. You may need to install it first if you haven't (`pip install xgboost`.)
- After having the models, you could use `model.fit()` on the training data as well as the other methods you have learned.
- Now, how do we evaluate the models using metrics?

Evaluating Time Series Forecasting Model

- In time series forecasting, we are trying to predict the numerical outputs based on inputs. It's just **regression!**
- To evaluate regression models, we usually look at the average errors they made (lower is better.)
  - **Root Mean Squared Error (RMSE):**

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}}$$

  - **Mean Absolute Error (MAE):**

$$MAE = \frac{\sum_{i=1}^{N} |y_i - \hat{y}_i|}{N}$$

Note: Sometimes, we may also want to look at the coefficient of determination (`r2_score`.) This value range from $0 - 1$, **higher is better.**

Data Preprocessing Summary

Here are the **general steps** to process time series data.

1. Convert the date and time column from string to Pandas' `Datetime` object using `df[‘Date’] = pd.to_datetime(df[‘Date’], format=<...>)`.
2. Set the dataframe index to be that date and time column using `df = df.set_index(‘Date’)`.
3. Deal with the missing data with the appropriate methods. You could try a forward fill using `df = df.ffill()`.
4. Resample the data to be indexed at the frequency you need (i.e. weekly, daily, hourly, etc.) using `df = df.resample(<FREQ>).mean()`
5. Generate multiple lag features and assigning them to multiple columns using `df[‘<LAG>’] = df[‘Close’].shift(<STEP>)`.

Model Training Summary

1. Split the data into **training and testing** by simply slicing the dataframe. If we have enough data, we could split the data into half training and half testing (i.e. `df_train = df.loc[:len(df)//2, :]` and `df_test = df.loc[len(df)//2:, :]`.)
2. Create **features and target** (for each training and testing): `X_train = df_train.loc[:, ['lag1', 'lag2', 'lag3', ...]]` and `y_train = df_train.loc[:, 'Close']`.
3. Train the model on the training data. For example:
   - Using **Random Forests**: `model = RandomForestRegressor()` and then `model.fit(X_train, y_train)`.
   - Using **XGBoost**: `model = XGBRegressor()` and then `model.fit(X_train, y_train)`.

Model Evaluation Summary

1. View the predicted time series by using `y_train_pred = model.predict(X_train)` and/or `y_test_pred = model.predict(X_test)`, then plot the results.
2. Evaluate the numerical metrics (**RMSE, MAE, and R-Squared**) using functions from `sklearn.metrics`. For example:
   - `mae = mean_absolute_error(y_test, y_pred_test)`
   - `rmse = root_mean_squared_error(y_test, y_pred_test)`
   - `r2 = r2_score(y_test, y_pred_test)`

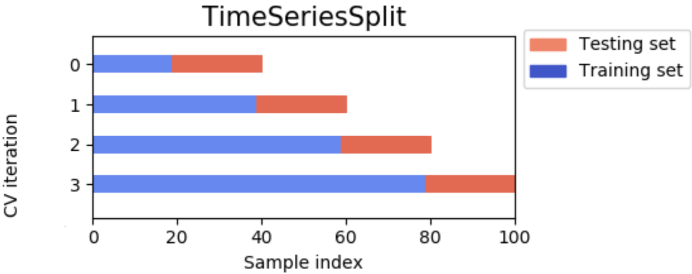
   Recap: Generally, we want to evaluate the model performance on the **testing set**, which reflects how well can the model generalize to unseen data.
3. Using the performance metrics, we can iteratively go back and tune the models until getting the desired performance.

Extra: Time Series Cross-Validation (1)

1. We know from the previous sessions that we use **cross-validation** to properly tune the hyperparameters while having a separated test set as the actual unseen data.
2. For time series, the appropriate way for cross validation is to use `sklearn.model_selection.TimeSeriesSplit()`
3. You may also pass an argument `gap` which determines how many samples you want to **exclude** between the training and validation set.
4. After assigning `cv = TimeSeriesSplit(n=5, gap=1)`, we can pass `cv` to classes like `GridSearchCV(cv=cv)` or `RandomizedSearchCV(cv=cv)` and then tune the hyperparameters as you wished.

Extra: Time Series Cross-Validation (2)

- This is how the cross-validation looks like when using `TimeSeriesSplit(n_splits=4)`.



- Note: More complex models like **Random Forests** and **XGBoost** have lots of hyperparameters to tune. You may consider looking at a lightweight version of XGBoost, **LightGBM** as well.