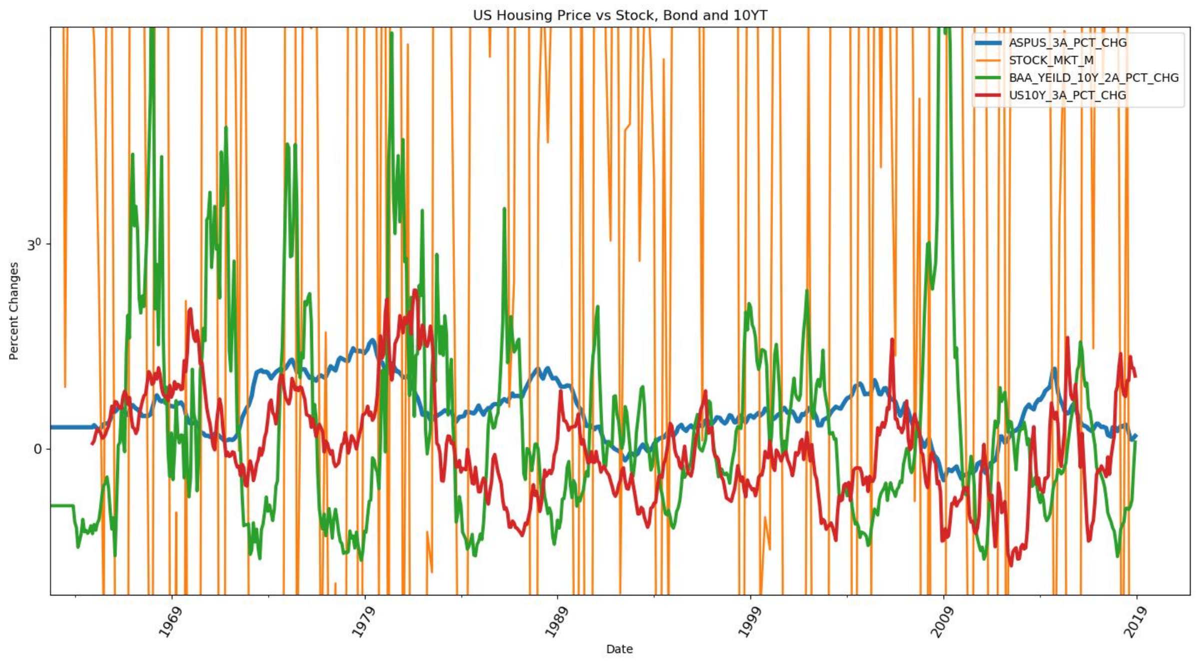
Predicting Housing Market

Table of Contents

Create Data sets and Data Exploration (Part 1)

- **During Housing market crash, average housing price drops 10% or more in 12 months**
- **This graph clearly indicate that during housing market crash (1981, 1992 & 2007) Personal Consumer Expenditures drops rapidly**
- **We can see positive growth in bond market during housing market crash (1981 & 2007)**
- **We can see that strong unemployment growth 1981 & 2007 or US Unemployment Growth is over 10% during housing market crash**
- **We can see that GDP growth is negative during 1981 & 2007 or GDP Growth is below -5% during housing market crash**
- **Higher interest rate is one of the major factor for housing market crash**
- **Normally stock market goes down sharply along with housing market**
- **We can see that housing supply ratio jump quickly right before the housing market crash. In another words, rapid housing supply growth eventually brings the housing market down.**
- **During the Housing market (1981, 1992 & 2007), Number of new 1F house sold in US sharply goes down.**
- **During the Housing market crash (1981, 1992 & 2007), Number of US Construction PERMIT in US sharply goes down.**

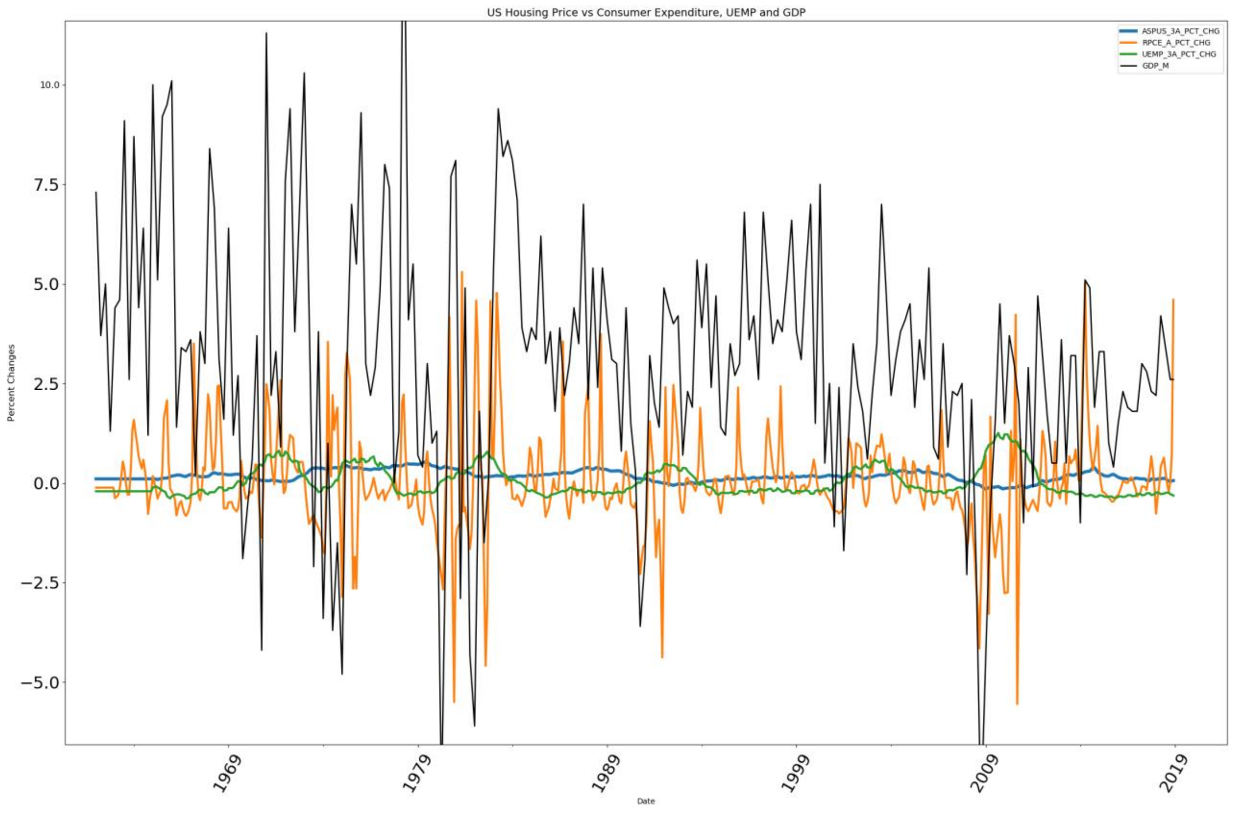
Compare US average housing price with stock market, bonds and interest rates



Before the housing market crash, generally interest rates are higher and bond markets are lower. During Market Crash (1981, 1992 & 2007), stock market goes down, bonds goes up and interest rates goes down.

Recent Data (2018) shows strong housing market, because strong stock market, low interest rate and healthy bond market.

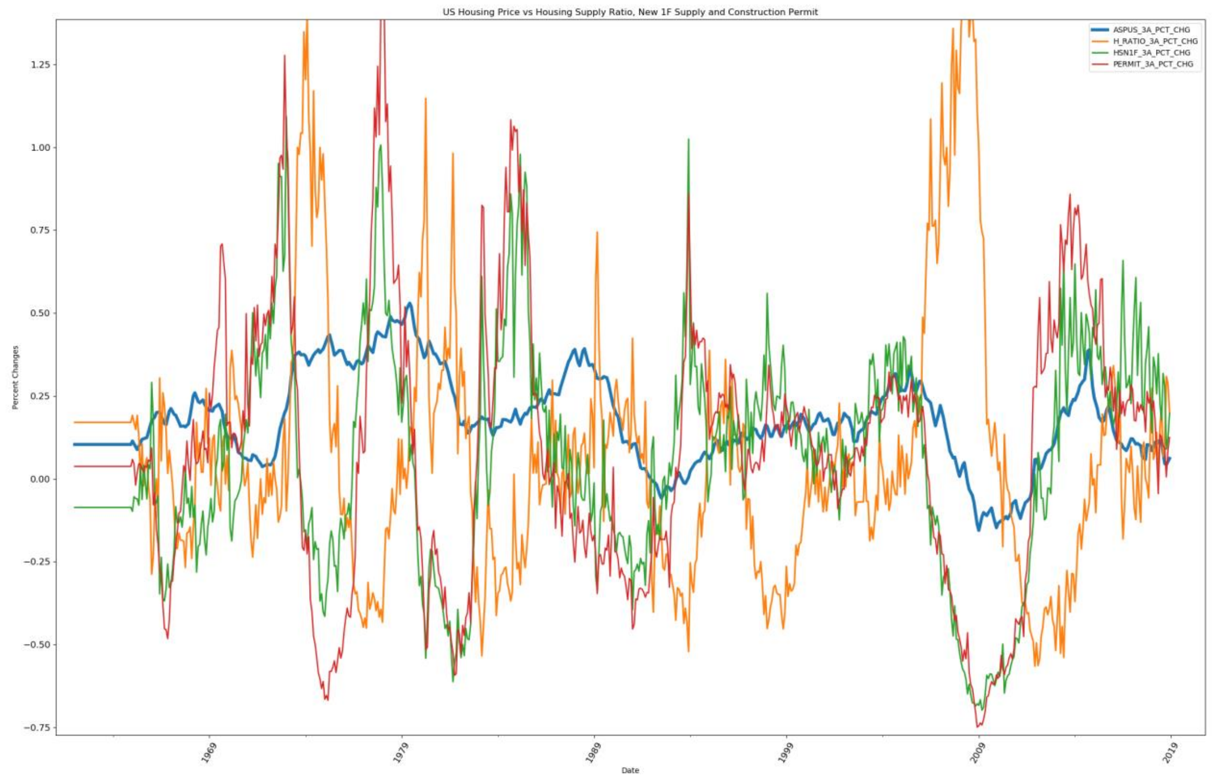
Compare US average housing price with Consumer expenditures, Unemployment market and US GDP Growth



During Market Crash (1981, 1992 & 2007), stock US GDP Growth drop sharply, US Unemployment Rate jump quickly and Personal Consumer Expenditures goes down.

If we look at the recent data, GDP is positive, Unemployment rate is very low, Personal Consumer expenditures is positive. Based on these data housing market still remain positive.

Compare US average housing price with Housing Supply Ratio, New 1F house sold in US and US Construction Permit



During Market Crash (1981, 1992 & 2007), Housing Supply Ratio jump sharply, Both number of new 1F houses sold & number of construction permits drop rapidly.

If we closely look at the graph for 2018 data, which shows Housing Supply Ratio is increasing slowly. Both number of new 1F houses sold & number of construction permits dropping slowly, which indicates downtrend market.

Machine Learning (Part 2)

Importing Housing Data

housing_df=pd.read_csv('C:/scripts/capstone2/housing_df2.csv', index_col=0)

ASPUS_3A_PCT_CHG US Average housing price movement (3 years % change)
H_RATIO_3A_PCT_CHG Housing Ratio: Number of house available vs. sold. (3 years % change)
HSN1F_3A_PCT_CHG New 1F House sold (3 years % change)
PERMIT_3A_PCT_CHG Current construction permits across USA (3 years % change)
STOCK_MKT_3A_PCT_CHG Stock market movement (3 years % change)
BAA_YEILD_10Y_2A_PCT_CHG Bond Market movement (2 years % change)

US10Y_3A_PCT_CHG US 10 Years Treasury Rate (3 years % change)
RPCE_A_PCT_CHG Personal consumer expenditures (1 years % change)
UEMP_3A_PCT_CHG Long Term Unemployment Rate (3 years % change)
RGDP_M_PCT_CHG Current US GDP (3 years % change)

```
from sklearn.model_selection import train_test_split # for train and test set split
from sklearn.model_selection import cross_val_score #Sklearn.model_seletion is used
import numpy as np
# Construct data for the model
type(housing_df)
#type(model)
housing_df.info()
X = np.array(housing_df.drop(['ASPUS_3A_PCT_CHG'],1))
y = np.array(housing_df['ASPUS_3A_PCT_CHG'])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("size of the training feature set is",X_train.shape)
print("size of the test feature set is",X_test.shape)
print("size of the training Target set is",y_train.shape)
print("size of the test Target set is",y_test.shape)
```

Linear Regression

```
The train root mean squarred error is : 0.11189417013121745
The test root mean squarred error is : 0.11556063834363067
The Linear Regression coefficient parameters are : [ 0.04182125  0.2144663
7 -0.07073799  0.00134644 -0.00772195  0.21773385
  0.00599973 -0.0770209   0.00099034]
The Linear Regression intercept value is : 0.170767518777298
The R squared metric is : 0.3324038459224431
```

The R^2 in scikit learn is the coefficient of determination. It is 1 - residual sum of square / total sum of squares.

RMSE of the test data is closer to the training RMSE (and lower) if you have a well trained model. It will be higher if we have an overfitted model.

K fold cross validation

```
# cross validation score
cv_score= cross_val_score(LinearRegression(),X,y,scoring='neg_mean_squared_error', cv=10) # k =10
print('cv_score is :', cv_score)

# mean squared error
print('cv_score is :', cv_score.mean())

# Root mean squared error
rmse_cv= np.sqrt(cv_score.mean() * -1)
print('The cross validation root mean squarred error is :', rmse_cv)
```

```
cv_score is : [-0.00846139 -0.02480723 -0.05623692 -0.02185939 -0.02657064
-0.0169021 -0.00375804 -0.00803476 -0.02712047 -0.02857531] cv_score is :
-0.022232625171923696 The cross validation root mean squared error is :
0.14910608697140335
```

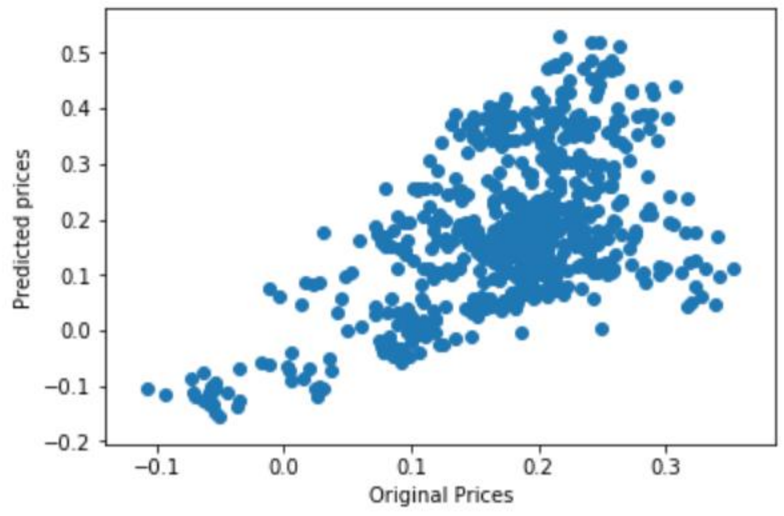
with Linear regressor we are able to predict the model with .114 RMSE and r squared 0.33 and cross validation root mean squarred error is : 0.1507

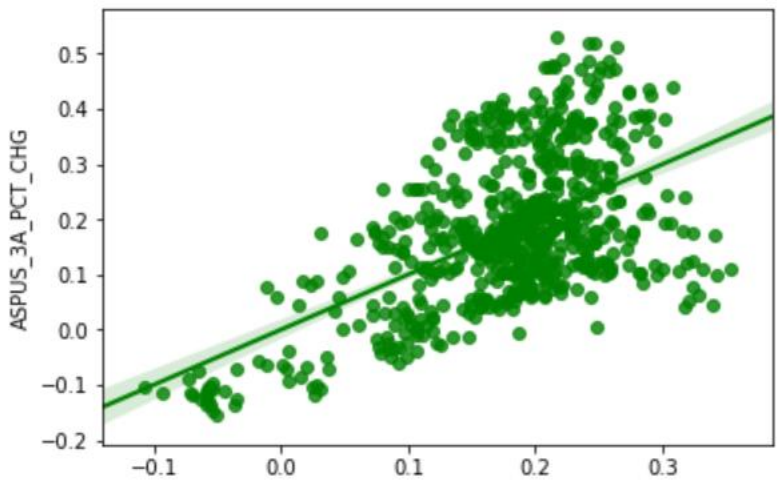
Fitting Linear Regression using statsmodels¶

Statsmodels is a great Python library for a lot of basic and inferential statistics. It also provides basic regression functions using an R-like syntax, so it's commonly used by statisticians. While we don't cover statsmodels officially in the Data Science Intensive workshop, it's a good library to have in your toolbox. Here's a quick example of what you could do with it. The version of least-squares we will use in statsmodels is called ordinary least-squares (OLS). There are many other versions of least-squares such as partial least squares (PLS) and weighted least squares (WLS).

```
m_rcpi = ols('y ~ H_RATIO_3A_PCT_CHG + HSN1F_3A_PCT_CHG + PERMIT_3A_PCT_CHG +
STOCK_MKT_3A_PCT_CHG + BAA_YEILD_10Y_2A_PCT_CHG + US10Y_3A_PCT_CHG +
RPCE_A_PCT_CHG + UEMP_3A_PCT_CHG + RGDP_M_PCT_CHG',housing_df).fit()
```

```
fdval = m_rcpi.fittedvalues
yplt.scatter(fdval, y)
plt.ylabel('Predicted prices')
plt.xlabel('Original Prices')
plt.show()
sns.regplot(x=fdval, y="ASPUS_3A_PCT_CHG", data=housing_df, fit_reg = True, color='g')
plt.show()
```



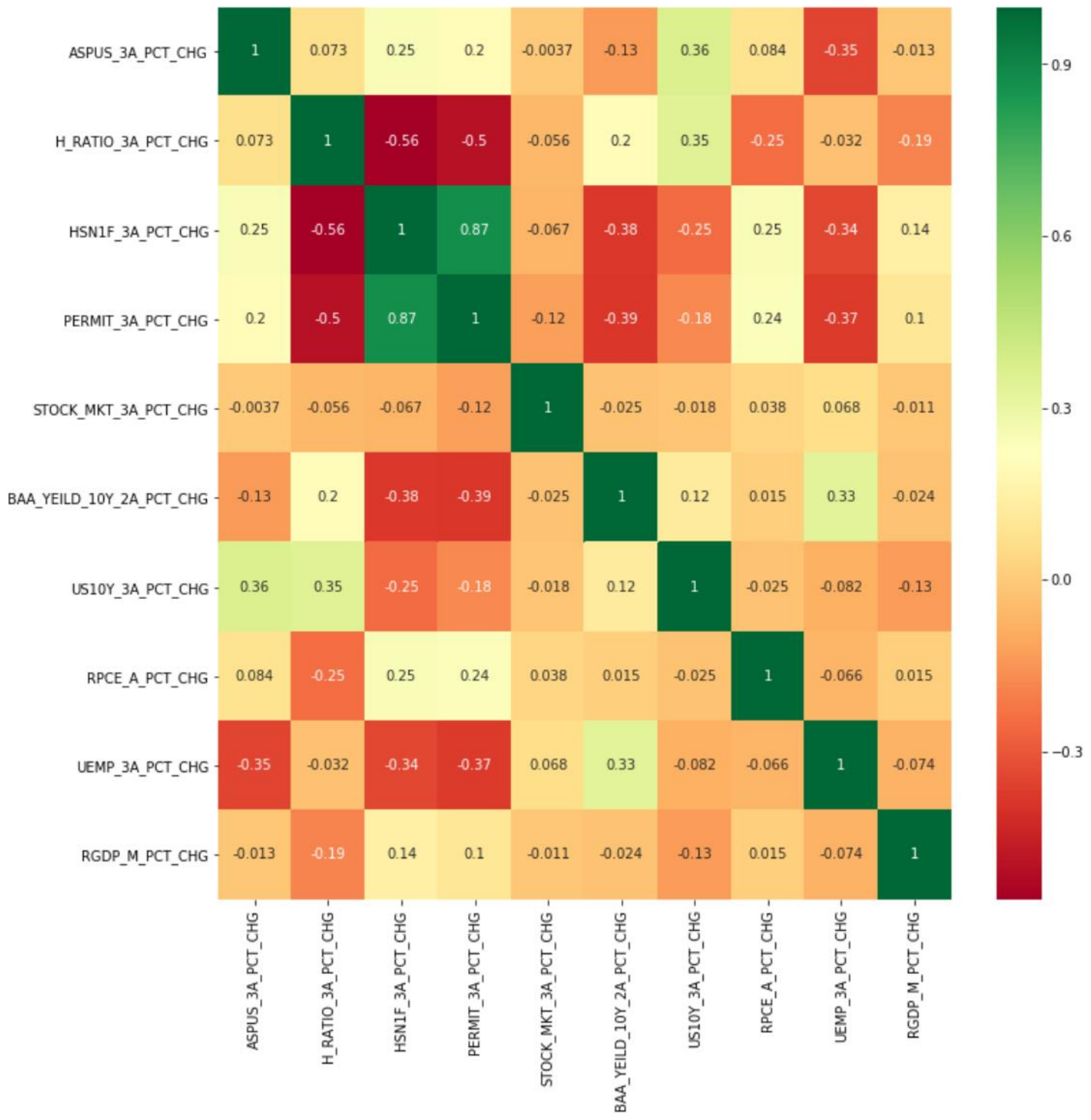


Correlation Matrix with Heatmap

Correlation states how the features are related to each other or the target variable.

Correlation can be positive (increase in one value of feature increases the value of the target variable) or negative (increase in one value of feature decreases the value of the target variable)

Heatmap makes it easy to identify which features are most related to the target variable, we will plot heatmap of correlated features using the seaborn library.



Number of new 1F house sold is positively (0.87) correllated with construction permits
Number of new 1F house sold is negatively (-0.56) correllated with Housing supply ratio
US Ave Houe price is negatively (-0.35) correllated with US Unemployment rate
Number of new 1F house sold is positively (0.87) correllated with construction permit

Monthly housing DATA ERROR TESTING

Each time series in the h_m_df DataFrame have very different seasonality patterns!

	ASPUS_M	H_RATIO_M	HSN1F_M	PERMIT_M	STOCK_MKT_M	BAA10YM	US10Y_M	RPCE_M	LRUN_UEMP	GDP_M
DATE										

	ASPUS_M	H_RATIO_M	HSN1F_M	PERMIT_M	STOCK_MKT_M	BAA10YM	US10Y_M	RPCE_M	LRUN_UEMP	GDP_M
DATE										
1962-01-01	19300.0	4.7	591	1122	-1.194624	1.00	4.08	4.30	6.536096	7.300000
1962-02-01	19300.0	4.7	591	1194	-4.306072	1.03	4.04	4.53	6.202647	6.100000
1962-03-01	19300.0	4.7	591	1134	-7.417520	1.11	3.93	4.77	5.869198	4.900000
1962-04-01	19300.0	4.7	591	1235	-10.528967	1.18	3.84	5.00	5.535749	3.700000
1962-05-01	19300.0	4.7	591	1142	-9.215257	1.13	3.87	4.40	5.437348	4.133333

'Correlations between multiple time series Earlier, we have extracted the seasonal component of each time series in the h_m_df DataFrame and stored those results in new DataFrame called seasonality_df. In the context of h_m_df data, it can be interesting to compare seasonality behavior, as this may help uncover which h_m_df indicators are the most similar or the most different.

This can be achieved by using the seasonality_df DataFrame and computing the correlation between each time series in the dataset. Here, we will compute and create a clustermap visualization of the correlations between time series in the seasonality_df DataFrame.

```
from sklearn.model_selection import train_test_split # for train and test set split
from sklearn.model_selection import cross_val_score #Sklearn.model_seletion is used

# Construct data for the model
type(housing_df)
#type(model)
housing_df.info()
X = np.array(h_m_df.drop(['ASPUS_M'],1))
y = np.array(h_m_df['ASPUS_M'])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("size of the training feature set is",X_train.shape)
print("size of the test feature set is",X_test.shape)
print("size of the training Target set is",y_train.shape)
print("size of the test Target set is",y_test.shape)

#Linear regression
from sklearn.linear_model import LinearRegression #import from sklearn
linear_reg= LinearRegression() # instantiated linreg
linear_reg.fit(X_train,y_train) #fit the model

#predict using X_test
predicted_train= linear_reg.predict(X_train)
predicted_test= linear_reg.predict(X_test)

from sklearn.metrics import mean_squared_error # import mse from sklearn
#calculate root mean squarred error
rmse_train=np.sqrt(mean_squared_error(y_train, predicted_train))
rmse_test=np.sqrt(mean_squared_error(y_test, predicted_test))

print('The train root mean squarred error is :', rmse_train)
print('The test root mean squarred error is :', rmse_test)

print('The Linear Regression coefficient parameters are :', linear_reg.coef_ )
print('The Linear Regression intercept value is :', linear_reg.intercept_)
```

```
size of the training feature set is (547, 9)
size of the test feature set is (137, 9)
size of the training Target set is (547,)
size of the test Target set is (137,)
The train root mean squarred error is : 70267.9274598809
The test root mean squarred error is : 77818.89478743549
The Linear Regression coefficient parameters are : [ 1788.79238781 216.56884311 -75.96446774
1773.71494551 56562.90186394 -16389.03294286 -9061.49679158 -5763.22028744 2896.18391227]
The Linear Regression intercept value is : 147549.50571956355
```

Apply time series decomposition to our dataset We will now perform time series decomposition on multiple time series. We can achieve this by leveraging the Python dictionary to store the results of each time series decomposition.

Here, we will initialize an empty dictionary with a set of curly braces, {}, use a for loop to iterate through the columns of the DataFrame and apply time series decomposition to each time series. After each time series decomposition, we will place the results in the dictionary by using the command my_dict[key] = value, where my_dict is our dictionary, key is the name of the column/time series, and value is the decomposition object of that time series.

Initialize an empty dictionary called h_m_df_decomp. Extract the column names of the housing DataFrame and place the results in a list called h_m_df_names. Iterate through each column in h_m_df_names and apply time series decomposition to that time series. Place the results in the h_m_df_decomp dictionary, where the column name is the key, and the value is the decomposition of the time series just performed.

The columns of a DataFrame can be accessed by using the .columns attribute. The basic structure of a for loop is: for item in list: perform command

```
h_m_df_decomp = {}

# Get the names of each time series in the DataFrame
h_m_df_names = h_m_df.columns

# Run time series decomposition on each time series of the DataFrame
for ts in h_m_df_names:
   ts_decomposition = sm.tsa.seasonal_decompose(h_m_df[ts])
   h_m_df_decomp[ts] = ts_decomposition
```

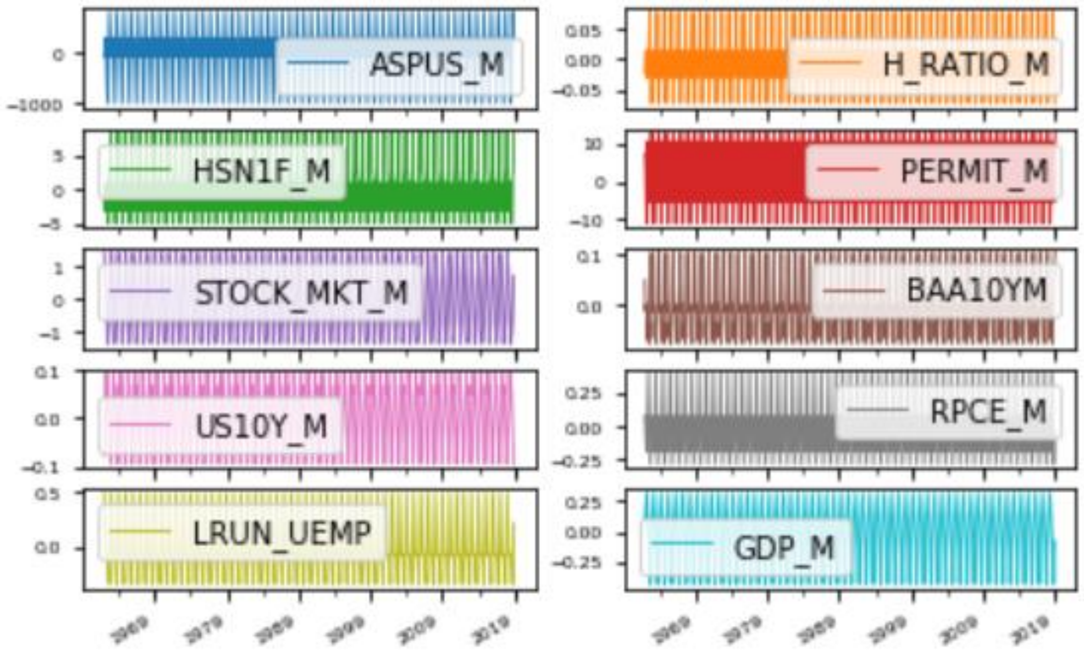
Visualize the seasonality of multiple time series We will now extract the seasonality component of h_m_df_decomp to visualize the seasonality in these time series. Note that before plotting, you will have to convert the dictionary of seasonality components into a DataFrame using the pd.DataFrame.from_dict() function.

An empty dictionary h_m_df_seasonal and the time series decompisiton object h_m_df_decomp created.

Iterate through each column name in jobs_names and extract the corresponding seasonal component from h_m_df_decomp. Place the results in the jobs_seasonal, where the column name is the name of the time series, and the value is the seasonal component of the time series. Convert h_m_df_seasonal to a DataFrame and call it seasonality_df. Create a facetted plot of all 10 columns in seasonality_df. Ensure that the subgraphs do not share y-axis.

The seasonal component can be extracted using the .seasonal attribute. Use the pd.DataFrame.from_dict() to convert a dictionary to a DataFrame. Faceted plots of DataFrame df can be generated by setting the subplots argument to True

```
# Extract the seasonal values for the decomposition of each time series
h_m_df_seasonal = {}

for ts in h_m_df_names:
   h_m_df_seasonal[ts] = h_m_df_decomp[ts].seasonal

# Create a DataFrame from the housing_seasonal dictionnary
seasonality_df = pd.DataFrame.from_dict(h_m_df_seasonal)
# Remove the label for the index
seasonality_df.index.name = None
# Create a faceted plot of the seasonality_df DataFrame
seasonality_df.plot(subplots=True,
            layout=(5, 2),  sharey=False,  fontsize=6,   linewidth=0.9,  legend=True)
plt.show()
```



Each time series in the h_m_df DataFrame have very different seasonality patterns!

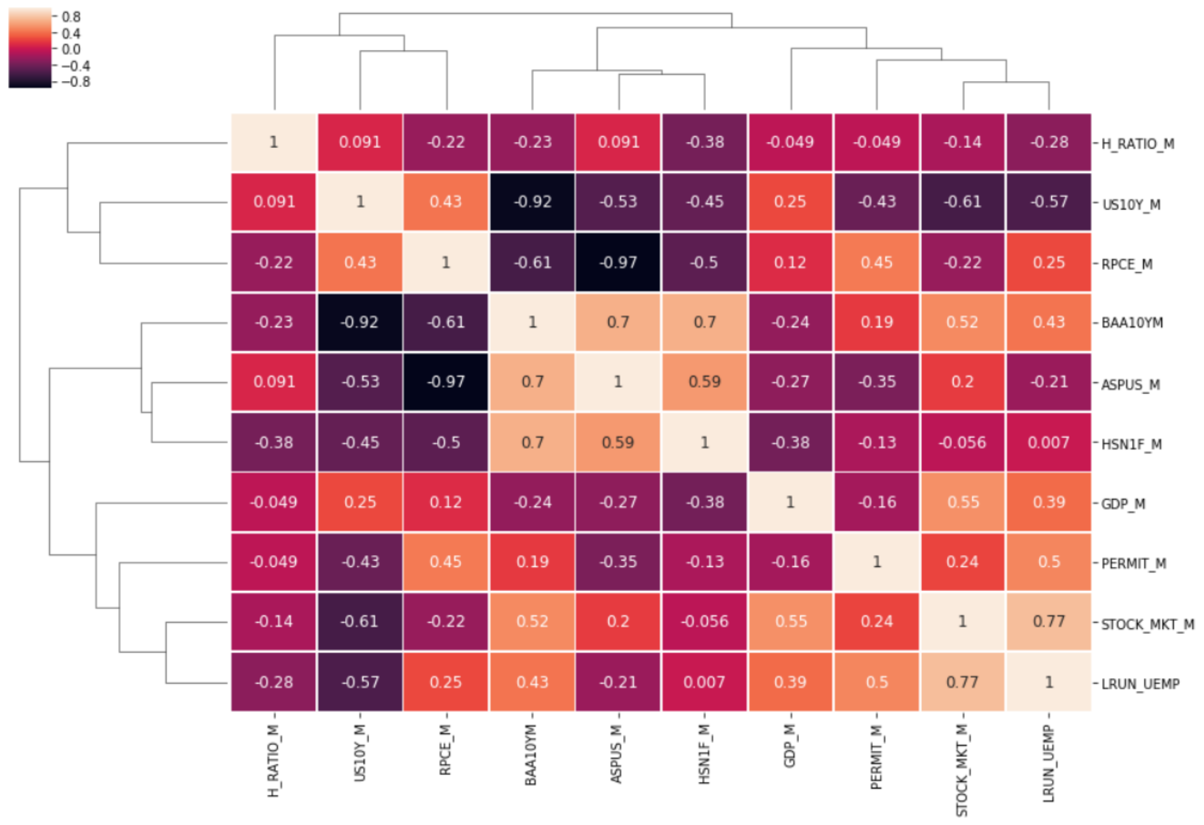
Correlations between multiple time series Earlier, we have extracted the seasonal component of each time series in the h_m_df DataFrame and stored those results in new DataFrame called seasonality_df. In the context of h_m_df data, it can be interesting to compare seasonality behavior, as this may help uncover which h_m_df indicators are the most similar or the most different.

This can be achieved by using the seasonality_df DataFrame and computing the correlation between each time series in the dataset. Here, we will compute and create a clustermap visualization of the correlations between time series in the seasonality_df DataFrame.

Compute the correlation between all columns in the seasonality_df DataFrame using the spearman method and assign the results to seasonality_corr. Create a new clustermap of your correlation matrix.

Use the .corr() method along with the method argument to create a correlation matrix. To plot a clustermap, use the sns.clustermap() function.

```
# Get correlation matrix of the seasonality_df DataFrame
seasonality_corr = seasonality_df.corr(method='spearman')

# Customize the clustermap of the seasonality_corr correlation matrix
fig = sns.clustermap(seasonality_corr, annot=True, annot_kws={"size": 12}, linewidths=.8, figsize=(15,
10))
plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=0)
plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(), rotation=90)
plt.show()
```



	H_RATIO_M	US10Y_M	RPCE_M	BAA10YM	ASPUS_M	HSN1F_M	GDP_M	PERMIT_M	STOCK_MKT_M	LRUN_UEMP
H_RATIO_M	1	0.091	-0.22	-0.23	0.091	-0.38	-0.049	-0.049	-0.14	-0.28
US10Y_M	0.091	1	0.43	-0.92	-0.53	-0.45	0.25	-0.43	-0.61	-0.57
RPCE_M	-0.22	0.43	1	-0.61	-0.97	-0.5	0.12	0.45	-0.22	0.25
BAA10YM	-0.23	-0.92	-0.61	1	0.7	0.7	-0.24	0.19	0.52	0.43
ASPUS_M	0.091	-0.53	-0.97	0.7	1	0.59	-0.27	-0.35	0.2	-0.21
HSN1F_M	-0.38	-0.45	-0.5	0.7	0.59	1	-0.38	-0.13	-0.056	0.007
GDP_M	-0.049	0.25	0.12	-0.24	-0.27	-0.38	1	-0.16	0.55	0.39
PERMIT_M	-0.049	-0.43	0.45	0.19	-0.35	-0.13	-0.16	1	0.24	0.5
STOCK_MKT_M	-0.14	-0.61	-0.22	0.52	0.2	-0.056	0.55	0.24	1	0.77
LRUN_UEMP	-0.28	-0.57	0.25	0.43	-0.21	0.007	0.39	0.5	0.77	1

Visualizing predicted values¶

When dealing with time series data, it's useful to visualize model predictions on top of the "actual" values that are used to test the model.

In this exercise, after splitting the data (stored in the variables X and y) into training and test sets, you'll build a model and then visualize the model's predictions on top of the testing data in order to estimate the model's performance.

Split the data (X and y) into training and test sets. Use the training data to train the regression model. Then use the testing data to generate predictions for the model.

You should be splitting up the arrays X and y into training and test sets with train_test_split().

Coefficient of Determination (R^2) The value of R is bounded on the top by 1, and can be infinitely low
Values closer to 1 mean the model does a better housing price of predicting outputs

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

X = np.array(housing_df.drop(['ASPUS_3A_PCT_CHG'],1))
y = np.array(housing_df['ASPUS_3A_PCT_CHG'])
```

```
# Split our data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.295, random_state=42)
# Fit our model and generate predictions
model = Ridge()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
score = r2_score(y_test, predictions)
print(score)
```

```
0.278077940417225
```

Using Ridge regression

```
print('r2_score for h_m_df (Transformed RAW Hosing Data) :')
X = np.array(h_m_df.drop(['ASPUS_M'],1))
y = np.array(h_m_df['ASPUS_M'])

# Split our data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.297, random_state=42)
#
# Fit our model and generate predictions
model = Ridge()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
score = r2_score(y_test, predictions)
print(score)
```

```
r2_score for h_m_df (Transformed RAW Hosing Data) : 0.5977151633811058
```

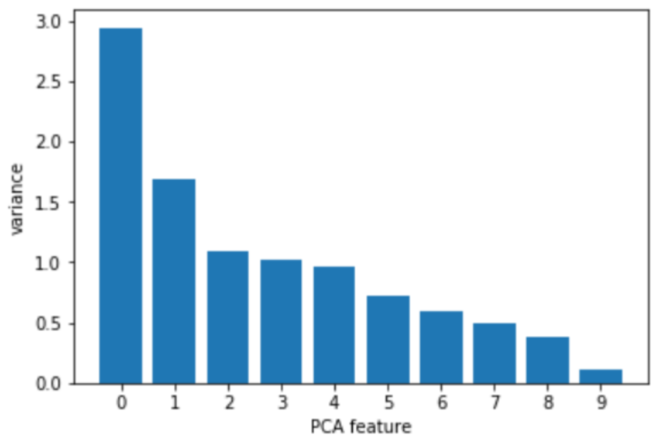
Variance of the PCA features

The dataset is 10-dimensional. But what is its intrinsic dimension? Make a plot of the variances of the PCA features to find out. As before, samples is a 2D array, where each row represents a fish. You'll need to standardize the features first.

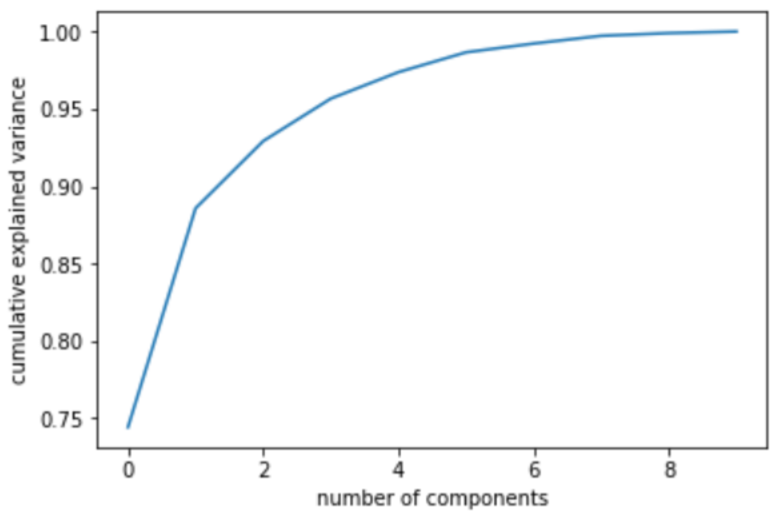
the use of principal component analysis for dimensionality reduction, for visualization of high-dimensional data, for noise filtering, and for feature selection within high-dimensional data. Because of the versatility and interpretability of PCA, it has been shown to be effective in a wide variety of contexts and disciplines. Given any high-dimensional dataset, I tend to start with PCA in order to visualize the relationship between points), to understand the main variance in the data and to understand the intrinsic dimensionality (by plotting the explained variance ratio).

We want to know how many principal components we can choose for our new feature subspace? A useful measure is the so-called "explained variance ratio". The explained variance ratio tells us how much information (variance) can be attributed to each of the principal components. We can plot bar graph between no. of features on X axis and variance ratio on Y axis

```
features = range(pca.n_components_)
feature_names = features = range(pca.n_components_)
plt.bar(features, pca.explained_variance_)

plt.xlabel('PCA feature')
plt.ylabel('variance')
plt.xticks(feature_names)
plt.show()

plt.plot([1, 9])
#plt.plot(['ASPUS_3A_PCT_CHG', 'H_RATIO_3A_PCT_CHG', 'HSN1F_3A_PCT_CHG',
'PERMIT_3A_PCT_CHG', 'STOCK_MKT_3A_PCT_CHG', 'BAA_YEILD_10Y_2A_PCT_CHG',
'US10Y_3A_PCT_CHG', 'RPCE_A_PCT_CHG', 'UEMP_3A_PCT_CHG', 'GDP_M'])
ax = plt.gca()
labels = ax.get_xticklabels()
for label in labels:
   print(label)

pca.fit_transform(X)
print(pca.mean_)
print(pca.components_)
print(pca.explained_variance_)
print(pca.explained_variance_ratio_)
print(pca.singular_values_)
print(pca.n_components_)
print(pca.noise_variance_)
```



```
#Let us load the basic packages needed for the PCA analysis

pca = PCA().fit(housing_df)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
plt.show()
```



Density plots¶

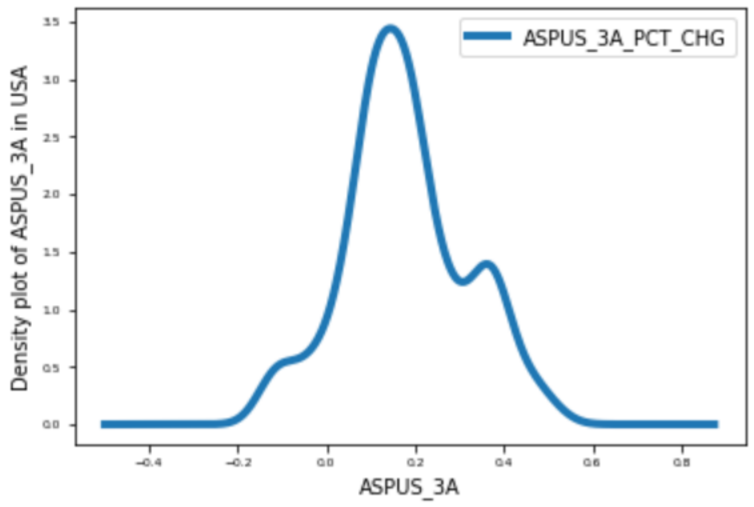
In practice, histograms can be a substandard method for assessing the distribution of data because they can be strongly affected by the number of bins that have been specified. Instead, kernel density plots represent a more effective way to view the distribution of your data. An example of how to generate a density plot of is shown below:

ax = df.plot(kind='density', linewidth=2) The standard .plot() method is specified with the kind argument set to 'density'. We also specified an additional parameter linewidth, which controls the width of the line to be plotted.

Using the ASPUS_3A DataFrame, produce a density plot of the ASPUS_3A data with line width parameter of 4. Annotate the x-axis labels of your boxplot with the string 'ASPUS_3A'. Annotate the y-axis labels of your boxplot with the string 'Density plot of ASPUS_3A levels in USA'.

Use the .plot() method with kind = 'density' along with the linewidth argument. The x and y labels can be set using the .set_xlabel() and .set_ylabel() methods.

```
ax = ASPUS_3A.plot(kind='density', linewidth=4, fontsize=6)
# Annotate x-axis labels
ax.set_xlabel('ASPUS_3A', fontsize=10)
# Annotate y-axis labels
ax.set_ylabel('Density plot of ASPUS_3A in USA', fontsize=10)
plt.show()
```



Cross-validation with shuffling¶

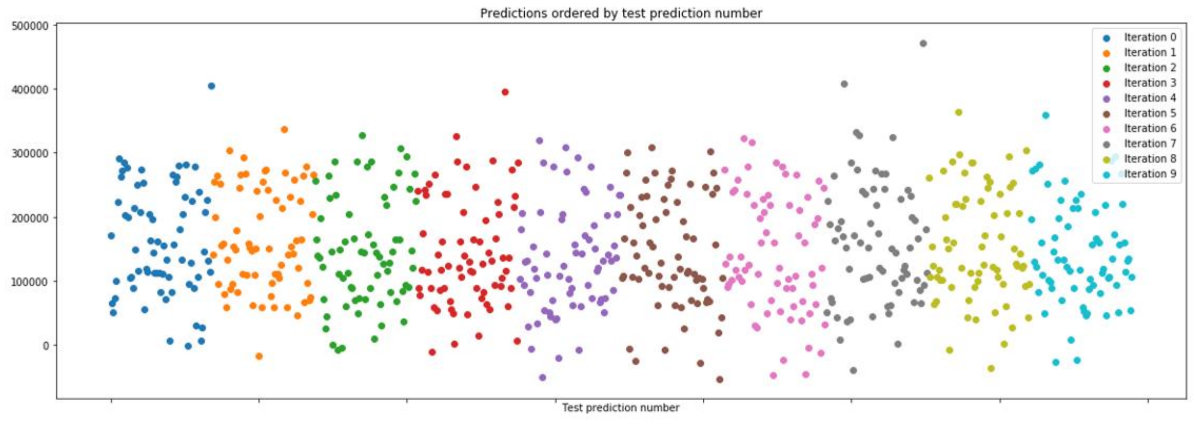
As you'll recall, cross-validation is the process of splitting your data into training and test sets multiple times. Each time you do this, you choose a different training and test set. In this exercise, you'll perform a traditional ShuffleSplit cross-validation on the company value data from earlier. Later we'll cover what changes need to be made for time series data. The data we'll use is the same historical price data for several large companies.

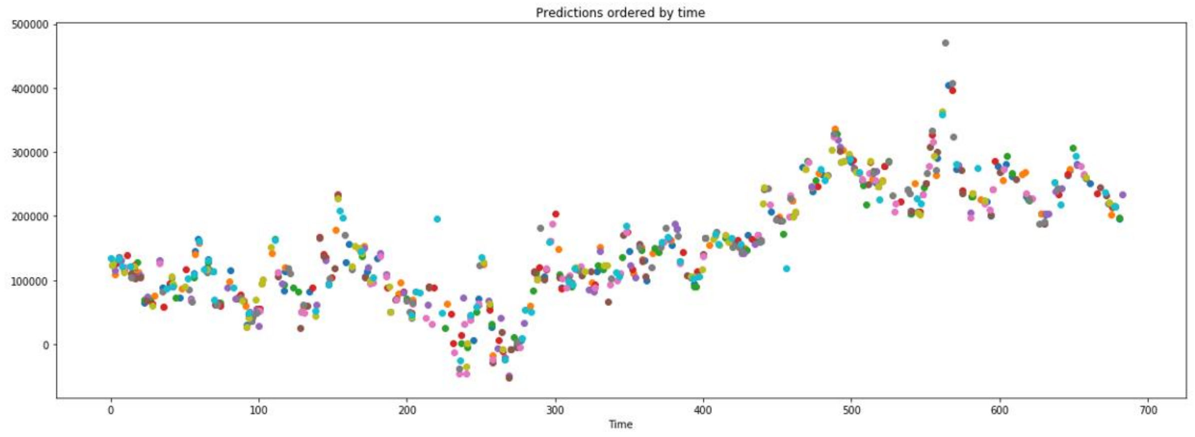
An instance of the Linear regression object (model) is available in your workspace along with the function r2_score() for scoring. Also, the data is stored in arrays X and y. We've also provided a helper function (visualize_predictions()) to help visualize the results.

Initialize a ShuffleSplit cross-validation object with 10 splits. Iterate through CV splits using this object. On each iteration: Fit a model using the training indices. Generate predictions using the test indices, score the model (R^2) using the predictions, and collect the results.

Use the ShuffleSplit() function with n_splits argument. Use the .split() method of the cross-validation object to yield training indices (for fitting the model) and test indices (for scoring the model). The r2_score function should take the actual and the predicted values as inputs, and returns the score.

```
r2_score for h_m_s_df (Transformed RAW Hosing Data) :
0.576676633395743
0.6454216130137627
0.6399803883503089
0.5782451294300545
0.5581412884464076
0.5435636150165827
0.642314533047976l
0.5803362625210808
0.5667778422298352
0.4593433908848251
```





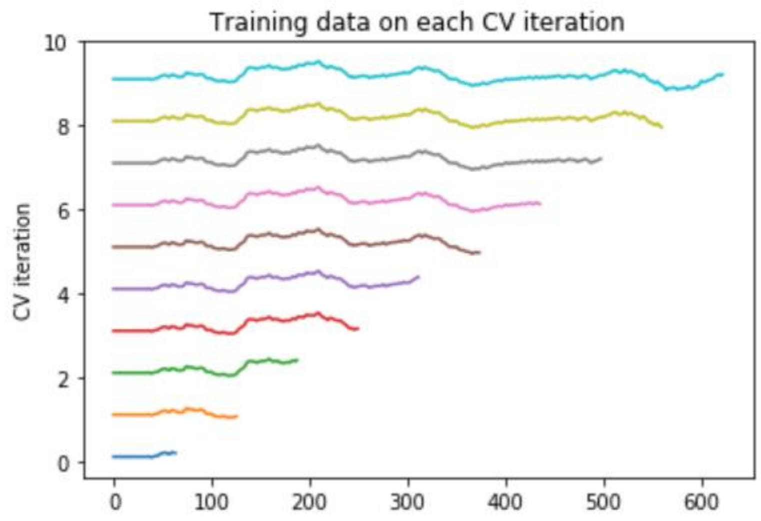
Time-based cross-validation¶

Finally, let's visualize the behavior of the time series cross-validation iterator in scikit-learn. Use this object to iterate through your data one last time, visualizing the training data used to fit the model on each iteration.

An instance of the Linear regression model object is available in your workpsace. Also, the arrays X and y (training data) are available too.

Import TimeSeriesSplit from sklearn.model_selection. Instantiate a time series cross-validation iterator with 10 splits. Iterate through CV splits. On each iteration, visualize the values of the input data that would be used to train the model for that iteration.

```
# Import TimeSeriesSplit
from sklearn.model_selection import TimeSeriesSplit

# Create time-series cross-validation object
cv = TimeSeriesSplit(n_splits=10)

# Iterate through CV splits
fig, ax = plt.subplots()
for ii, (tr, tt) in enumerate(cv.split(X, y)):
    X_train, X_test = X[tr], X[tt]
    y_train, y_test = y[tr], y[tt]
    # Plot the training data on each iteration, to see the behavior of the CV
    ax.plot(tr, ii + y_train)

ax.set(title='Training data on each CV iteration', ylabel='CV iteration')
plt.show()
```



Note that the size of the training set grew each time when you used the time series cross-validation object. This way, the time points you predict are always after the timepoints we train on.

Time Series (Part 3)

Autocorrelation plots can be used to quickly discover patterns into your time series, so let's delve a little bit deeper into that!

Partial autocorrelation in time series data Like autocorrelation, the partial autocorrelation function (PACF) measures the correlation coefficient between a time-series and lagged versions of itself. However, it extends upon this idea by also removing the effect of previous time points. For example, a partial autocorrelation function of order 3 returns the correlation between our time series (t_1, t_2, t_3, ...) and its own values lagged by 3 time points (t_4, t_5, t_6, ...), but only after removing all effects attributable to lags 1 and 2.

The plot_pacf() function in the statsmodels library can be used to measure and plot the partial autocorrelation of a time series.

Import tsaplots from statsmodels.graphics.

Use the plot_pacf() function from tsaplots to plot the partial autocorrelation of the

Time series decomposition

When visualizing time series data, we should look out for some distinguishable patterns:

seasonality: does the data display a clear periodic pattern?

trend: does the data follow a consistent upwards or downward slope?

noise: are there any outlier points or missing values that are not consistent with the rest of the data?

You can rely on a method known as time-series decomposition to automatically extract and quantify the structure of time-series data. The statsmodels library provides the seasonal_decompose() function to perform time series decomposition out of the box.

To print the seasonality component, access the seasonal component of the decomposition object.

FORECASTING ARIMA

Applying statistical modeling and machine learning to perform time-series forecasting.

What techniques may help answer these questions?

Statistical models

Ignore the time-series aspect completely and model using traditional statistical modeling toolbox.

Examples. Regression-based models.

Univariate statistical time-series modeling.

Examples. Averaging and smoothing models, ARIMA models.

Slight modifications to univariate statistical time-series modeling.

Examples. External regressors, multi-variate models.

Additive or component models.

Examples. Facebook Prophet package.

Structural time series modeling.

Examples. Bayesian structural time series modeling, hierarchical time series modeling.

Machine learning models

Examples. Support Vector Machines (SVMs), Random Forest Regression, Gradient-Boosted Decision Trees (GBDTs).

Hidden markov models (HMMs).

Other sequence-based models.

Gaussian processes (GPs).

Recurrent neural networks (RNNs).

Additional data considerations before choosing a model

Whether or not to incorporate external data

Whether or not to keep as univariate or multivariate (i.e., which features and number of features)

Outlier detection and removal

Missing value imputation

Let us model some time-series data! Finally! ARIMA models.

We will be doing an example here! We can use ARIMA models when we know there is dependence between values and we can leverage that information to forecast.

ARIMA = Auto-Regressive Integrated Moving Average.

Assumptions. The time-series is stationary.

Depends on:

1. Number of AR (Auto-Regressive) terms (p).
2. Number of I (Integrated or Difference) terms (d).
3. Number of MA (Moving Average) terms (q).

ACF and PACF Plots

How do we determine p, d, and q? For p and q, we can use ACF and PACF plots (below).

Autocorrelation Function (ACF). Correlation between the time series with a lagged version of itself (e.g., correlation of Y(t) with Y(t-1)).

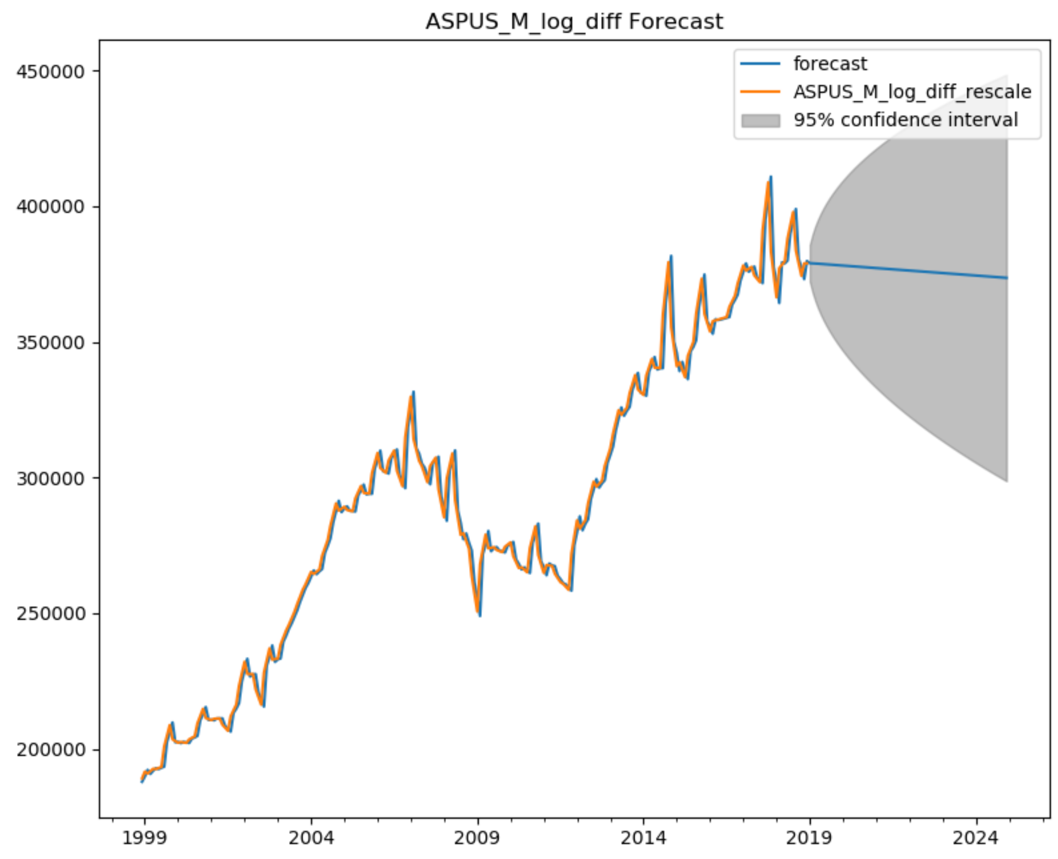
Partial Autocorrelation Function (PACF). Additional correlation explained by each successive lagged term.

How do we interpret ACF and PACF plots?

p – Lag value where the PACF chart crosses the upper confidence interval for the first time.

q – Lag value where the ACF chart crosses the upper confidence interval for the first time.

```
import statsmodels.api as sm
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error, r2_score

ASPUS_M_log_diff=h_m_df_transform['ASPUS_M_log_diff']
ASPUS_M_log_diff = ARMA(ASPUS_M_log_diff, order=(1,0, 1))

ASPUS_M_log_diff = ASPUS_M_log_diff.fit()
print("The AIC for an AR(1) is: ", ASPUS_M_log_diff.aic)
print("The BIC for an AR(1) is: ", ASPUS_M_log_diff.bic)

h_m_df['ASPUS_M_log_diff_rescale'] = np.exp(h_m_df['ASPUS_M_log_diff'] +
h_m_df['ASPUS_M_log'])
h_m_df['ASPUS_M_log_diff_rescale']=h_m_df['ASPUS_M_log_diff_rescale'].fillna(method='bfill')
#viz_df['yhat_rescaled'] = np.exp(viz_df['yhat'])

ASPUS_M_log_diff.plot_predict(start='12-01-98', end='12-01-2024')
h_m_df[['ASPUS_M', 'ASPUS_M_log_diff_rescale']].plot()
#ASPUS_M_log_diff_rescale.plot_predict(start='12-01-01', end='12-01-2024')
X3= h_m_df['ASPUS_M_log_diff_rescale']
X3 = ARMA(X3, order=(3,0,6))  #Good

X3=X3.fit()
X3.plot_predict(start='12-01-98', end='12-01-2024')
plt.legend(fontsize=10)
plt.title('ASPUS_M_log_diff Forecast')
plt.show()
```



Forecasting Housing Market using FBProphet (Part 4)

Let us model some time-series data! using Facebook Prophet package.

We will be doing an example here! Installing the necessary packages might take a couple of minutes. In the meantime, I can talk a bit about Facebook Prophet, a tool that allows folks to forecast using additive or component models relatively easily. It can also include things like:

Day of week effects
Day of year effects
Holiday effects
Trend trajectory
Can do MCMC sampling

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

Accurate and fast.

Prophet is used in many applications across Facebook for producing reliable forecasts for planning and goal setting. We've found it to perform better than any other approach in the majority of cases. We fit models in Stan so that you get forecasts in just a few seconds.

Get a reasonable forecast on messy data with no manual effort. Prophet is robust to outliers, missing data, and dramatic changes in your time series.

Tunable forecasts.

The Prophet procedure includes many possibilities for users to tweak and adjust forecasts. You can use human-interpretable parameters to improve your forecast by adding your domain knowledge.

Prophet follows the sklearn model API. We create an instance of the Prophet class and then call its fit and predict methods.

The input to Prophet is always a dataframe with two columns: ds and y. The ds (datestamp) column should be of a format expected by Pandas, ideally YYYY-MM-DD for a date or YYYY-MM-DD HH:MM:SS for a timestamp. The y column must be numeric, and represents the measurement we wish to forecast.

https://facebook.github.io/prophet/docs/quick_start.html#python-api

We fit the model by instantiating a new Prophet object. Any settings to the forecasting procedure are passed into the constructor. Then you call its fit method and pass in the historical dataframe. Fitting should take 1-5 seconds.

```
# Python
m = Prophet()
m.fit(df)
```

Predictions are then made on a dataframe with a column ds containing the dates for which a prediction is to be made. You can get a suitable dataframe that extends into the future a specified number of days using the helper method Prophet.make_future_dataframe. By default it will also include the dates from the history, so we will see the model fit as well.

Using Prophet is extremely straightforward. You import it, load some data into a pandas dataframe, set the data up into the proper format and then start modeling /

```
forecasting.h_m_df = pd.read_csv('C:/scripts/capstone2/h_m_df.csv', index_col=0)
from fbprophet import Prophet
from datetime import datetime
plt.rcParams['figure.figsize']=(20,10)
plt.style.use('ggplot')
#conda install pystan
#!pip install fbprophet
from fbprophet import Prophet
```

```
import datetime
from datetime import datetime

plt.rcParams['figure.figsize']=(20,10)
plt.style.use('ggplot')

X1 = np.array(housing_df.drop(['ASPUS_3A_PCT_CHG'],1))
y1 = np.array(housing_df['ASPUS_3A_PCT_CHG'])

X = np.array(h_m_df.drop(['ASPUS_M'],1))
y = np.array(h_m_df['ASPUS_M'])

df=df.rename(columns={'DATE':'ds', 'ASPUS_M':'y'})
```

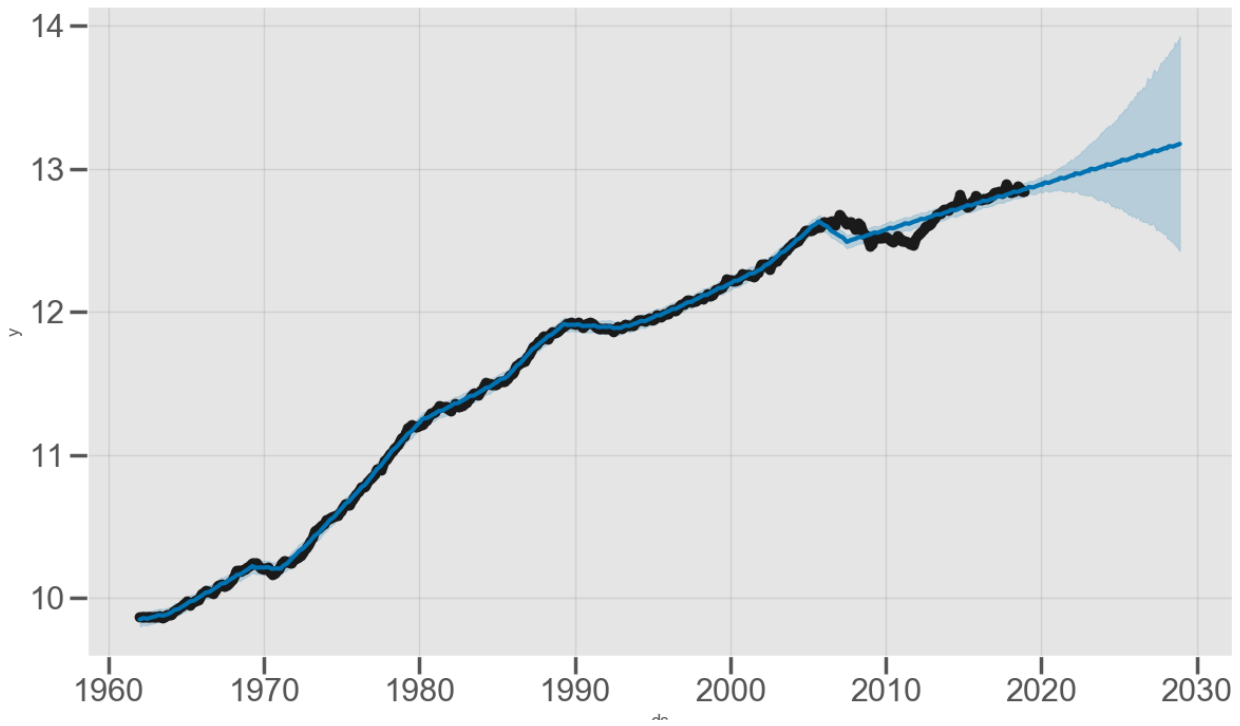
The format of the dataframe. This is the format that Prophet expects to see. There needs to be a 'ds' column that contains the datetime field and and a 'y' column that contains the value we are wanting to model/forecast.

```
df=df[['ds', 'y']]
df.tail()

df['y_orig'] = df['y'] # to save a copy of the original data..you'll see why shortly.
# log-transform y
df['y'] = np.log(df['y'])
df.tail()

model = Prophet() #instantiate Prophet
model.fit(df) #fit the model with your dataframe

'''In this line of code, we are creating a pandas dataframe with 12 (periods = 12) future data points with a monthly frequency (freq = 'm'). If you're working with daily data, you wouldn't want include freq='m'.'''
```

Forecast using the 'predict' command:

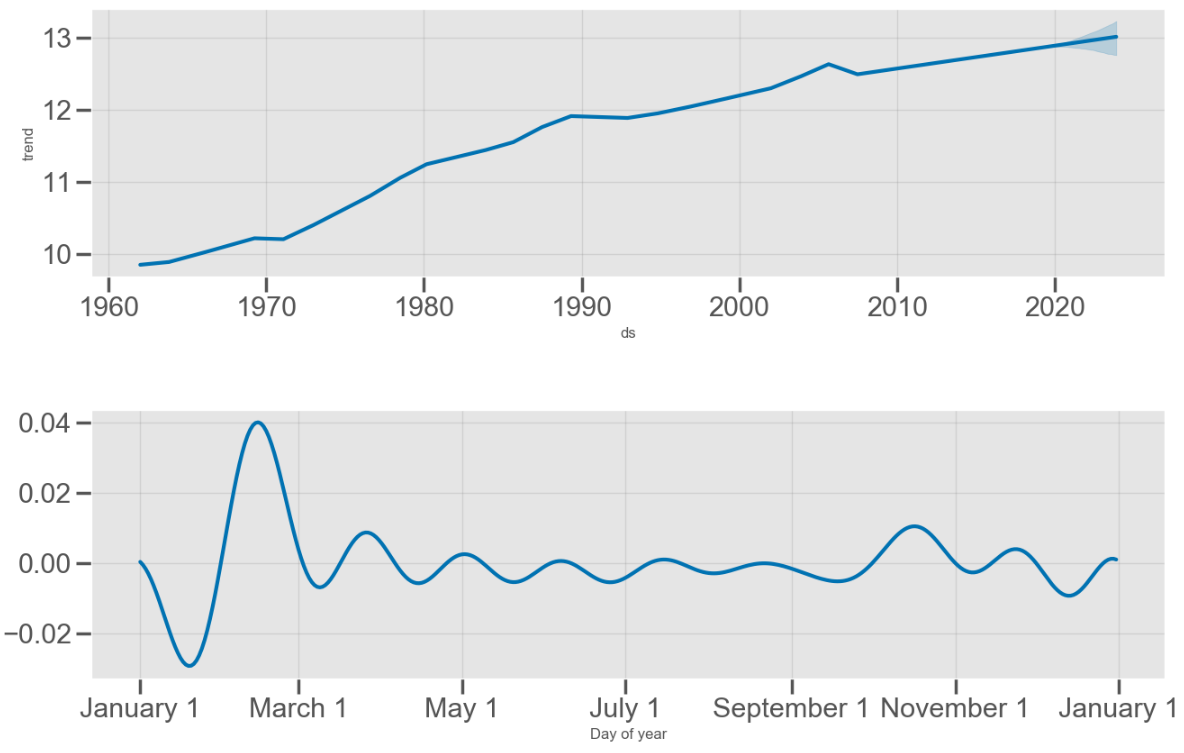
If you take a quick look at the data using .head() or .tail(), you'll notice there are a lot of columns in the forecast_data dataframe. The important ones (for now) are 'ds' (datetime), 'yhat' (forecast), 'yhat_lower' and 'yhat_upper' (uncertainty levels).'''

```
forecast_data_y[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()

'''Let's take a look at a graph of this data to get an understanding of how well our model is working.'''

model.plot(forecast_data_10y)
```



'''That looks pretty good. Now, let's take a look at the seasonality and trend components of our /data/model/forecast.'''

```
model.plot_components(forecast_data_5y)
```



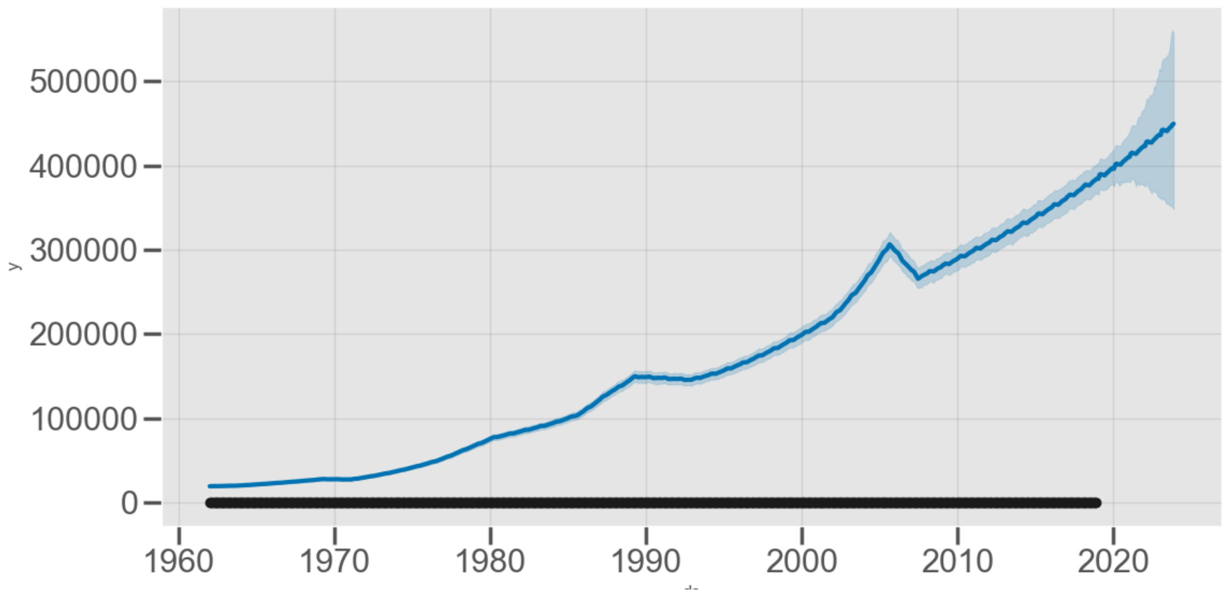
From the trend and seasonality, we can see that the trend is a playing a large part in the underlying time series and seasonality comes into play more toward the beginning of the year.

So far so good. With the above info, we've been able to quickly model and forecast some data to get a feel for what might be coming our way in the future from this particular data set.

Forecast plot to display our 'original' data so you can see the forecast in 'context' and in the original scale rather than the log-transformed data. We can do this by using np.exp() to get our original data back.

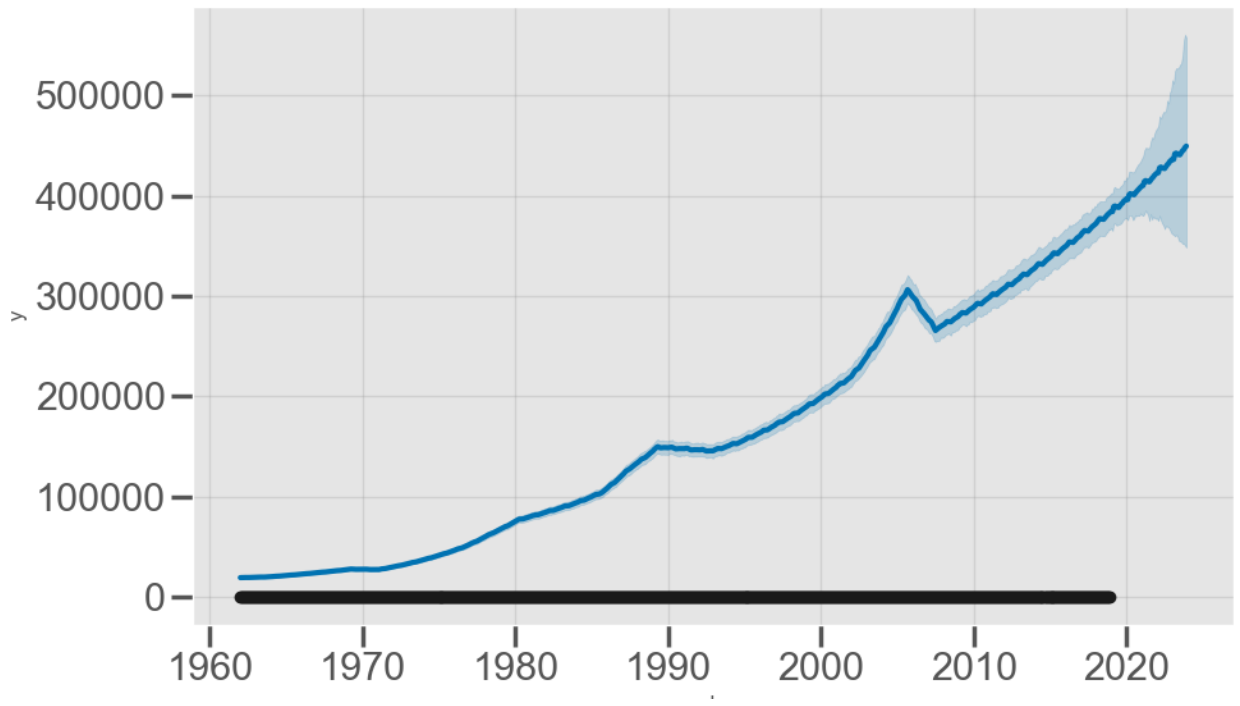
Let's take a look at the forecast with the original data:

model.plot(forecast_data_5y_orig)



In 5 years average US Housing range will be between $360000 and $560000

Something looks wrong (and it is)! Our original data is drawn on the forecast but the black dots (the dark line at the bottom of the chart) is our log-transform original 'y' data. For this to make any sense, we need to get our original 'y' data points plotted on this chart. To do this, we just need to rename our 'y_orig' column in the df dataframe to 'y' to have the right data plotted. Be careful here…you want to make sure you don't continue analyzing data with the non-log-transformed data.

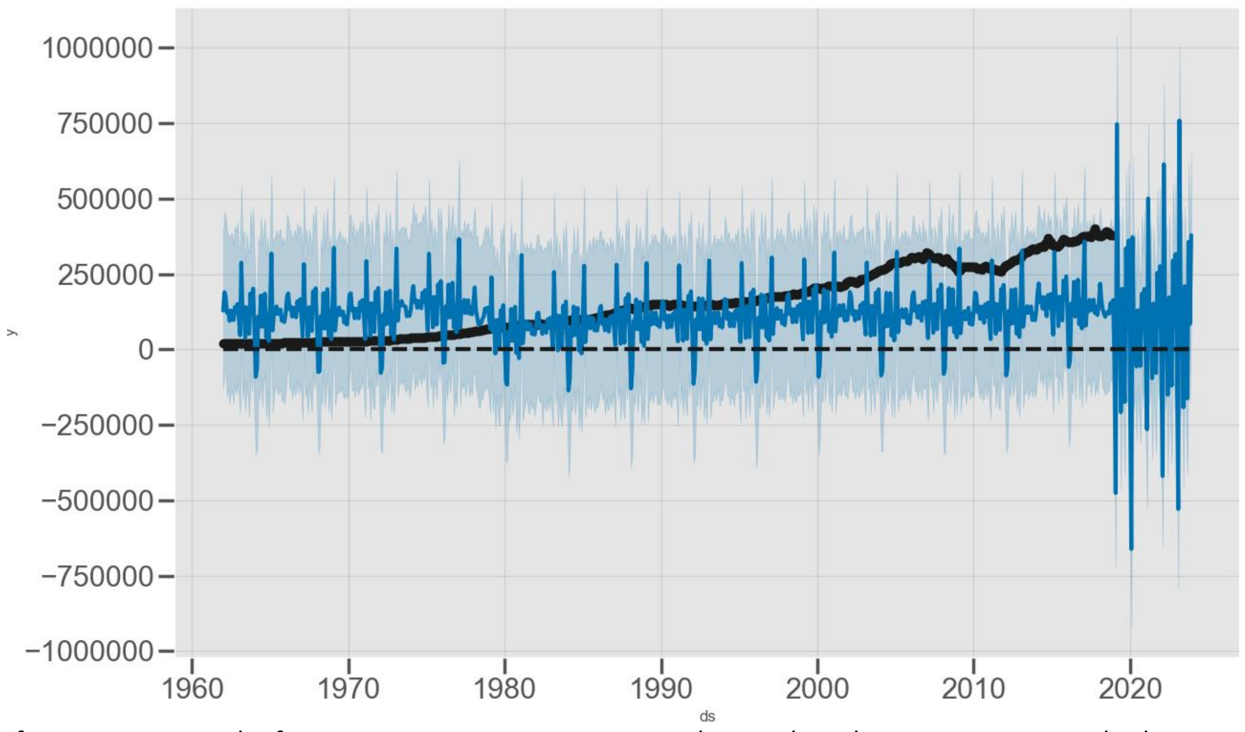


There we got the forecast for Housing Price 60 months into the future (you have to look closely at the very far right-hand side for the forecast). It looks like the next sixths months will see sales between 360K and 570K.

We make a dataframe for future predictions as before, except we must also specify the capacity in the future. Here we keep capacity constant at the same value as in the history, and forecast 3 years into the future:

```
cap = df['cap']
flr = df['floor']
df['cap'] = cap
df['floor'] = flr

model=Prophet(changepoint_range=0.8, changepoint_prior_scale=0.12, growth='logistic',
              seasonality_mode='additive', interval_width=0.95, yearly_seasonality=True,
weekly_seasonality=True)
model.add_seasonality(name="monthly", period=30.5, fourier_order=12, prior_scale=0.02)
model.fit(df);
#create 12 months of future data
future_data = model.make_future_dataframe(periods=60, freq = 'm')
type(future_data)
future_data.tail()
future_data['cap']=30
future_data['floor']=1
forecast_data = model.predict(future_data)
#forecast_data['cap'] = cap
```

```
#forecast_data['floor'] =flr
##forecast the data for future data
#forecast_data = model.predict(future_data)
fig = model.plot(forecast_data, uncertainty=True)
```



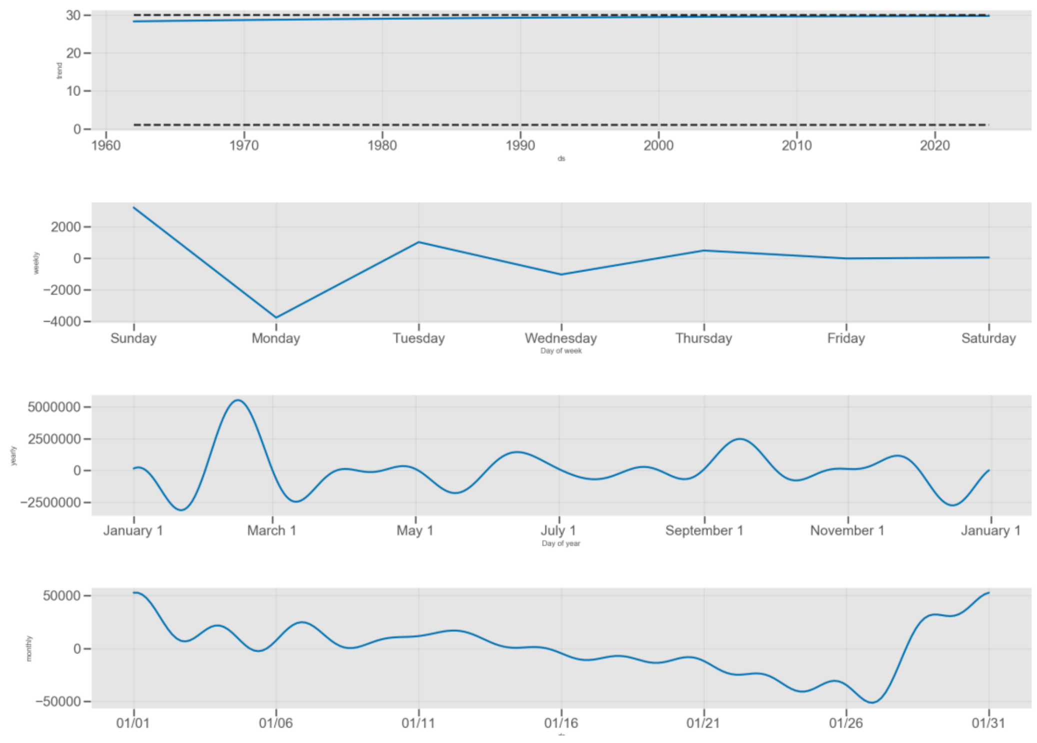
If you want to see the forecast components, you can use the Prophet.plot_components method. By default you'll see the trend, yearly seasonality, and weekly seasonality of the time series. If you include holidays, you'll see those here, too.

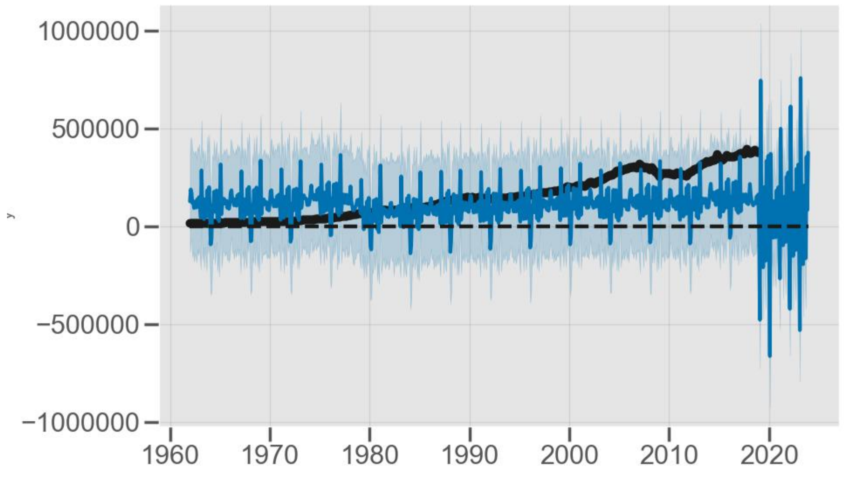
A variation in values from the output presented above is to be expected as Prophet relies on Markov chain Monte Carlo (MCMC) methods to generate its forecasts. MCMC is a stochastic process, so values will be slightly different each time.

Prophet also provides a convenient function to quickly plot the results of our forecasts:

```
fig2 = model.plot_components(forecast_data, uncertainty=True)
```

```
model.plot(forecast_data)
```





While this is a nice chart, it is kind of 'busy' for me. Additionally, I like to view my forecasts with original data first and forecasts appended to the end (this 'might' make sense in a minute).

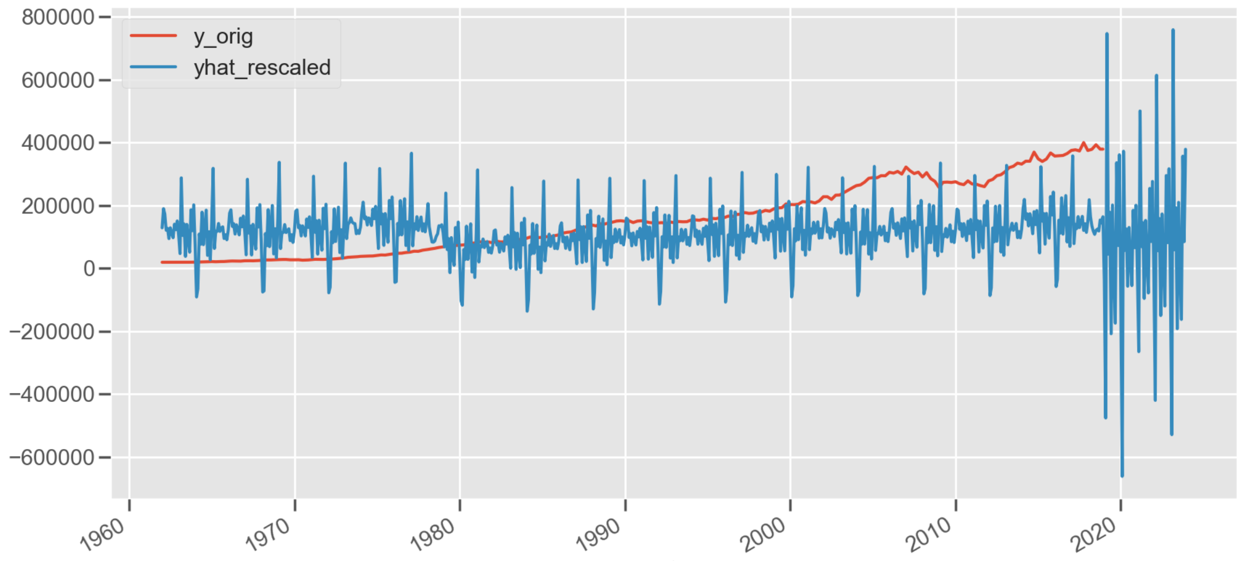
First, we need to get our data combined and indexed appropriately to start plotting. We are only interested (at least for the purposes of this article) in the 'yhat', 'yhat_lower' and 'yhat_upper' columns from the Prophet forecasted dataset. Note: There are much more pythonic ways to these steps, but I'm breaking them out for each of understanding.

```
df.set_index('ds', inplace=True)
forecast_data.set_index('ds', inplace=True)
viz_df = df.join(forecast_data[['yhat', 'yhat_lower','yhat_upper']], how = 'outer')
```

```
del viz_df['y']
#del viz_df['index']

viz_df=viz_df[['y_orig', 'yhat', 'yhat_lower', 'yhat_upper']]
viz_df.tail()
viz_df.head()
viz_df.describe()

viz_df['yhat_rescaled'] = viz_df['yhat']
viz_df.tail()
```

Let's take a look at the Housing Price and yhat_rescaled data together in a chart.

```
viz_df[['y_orig', 'yhat_rescaled']].plot()
```



To plot our forecasted data, we'll set up a function (for re-usability). This function imports a couple of extra libraries for subtracting dates (timedelta) and then sets up the function.

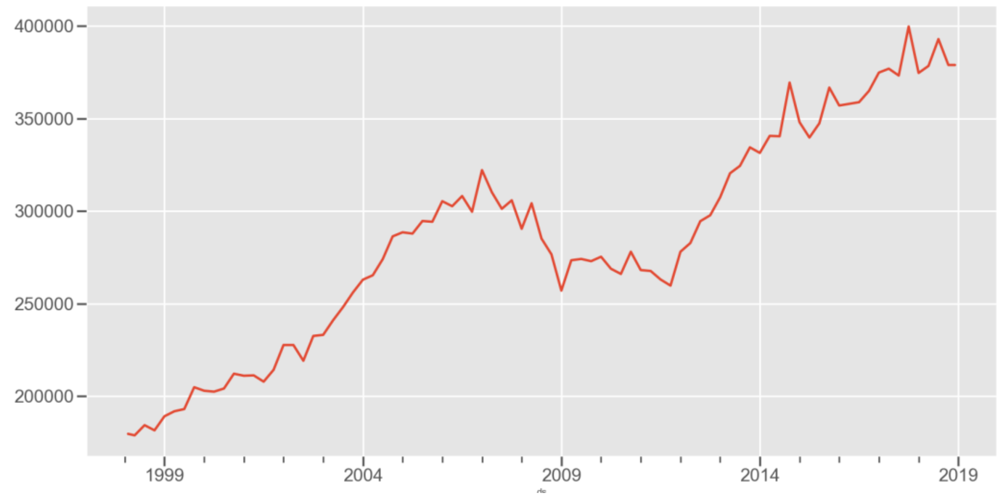
```
from datetime import date,timedelta

def plot_data(func_df, end_date):
   end_date = end_date - timedelta(weeks=4) # find the 2nd to last row in the data. We don't take the
last row because we want the charted lines to connect
   mask = (func_df.index > end_date) # set up a mask to pull out the predicted rows of data.
   predict_df = func_df.loc[mask] # using the mask, we create a new dataframe with just the predicted
data.

# Now...plot everything
   fig, ax1 = plt.subplots()
   ax1.plot(sales_df.y_orig)
   ax1.plot((np.exp(predict_df.yhat)), color='black', linestyle=':')
   ax1.fill_between(predict_df.index, np.exp(predict_df['yhat_upper']), np.exp(predict_df['yhat_lower']),
alpha=0.5, color='darkgray')
   ax1.set_title('Housing Price Orig (Orange) vs Housing Price Forecast (Black)')
   ax1.set_ylabel('Dollar Sales')
   ax1.set_xlabel('Date')

# change the legend text
   L=ax1.legend() #get the legend
   L.get_texts()[0].set_text('Actual Price') #change the legend text for 1st plot
   L.get_texts()[1].set_text('Forecasted Price') #change the legend text for 2nd plo
```

Import monthly housing price data

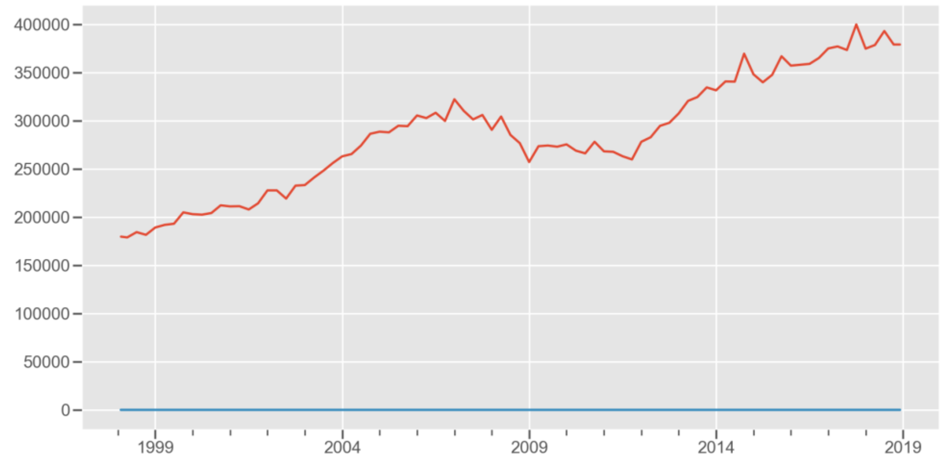
```
import pandas as pd
import numpy as np
h_m_df2 = pd.read_csv('C:/scripts/capstone2/h_m_df2.csv', index_col='DATE', parse_dates=True)
h_m_df2.head()

from fbprophet import Prophet
import datetime
from datetime import datetime

plt.rcParams['figure.figsize']=(20,10)
plt.style.use('ggplot')

#Looking at the recent data
df4 = h_m_df2.loc['19980102':'20181201']
df2 = df4.reset_index()
df3 = df4.reset_index()  #df3_orig
df2=df2.rename(columns={'DATE':'ds', 'ASPUS_M':'y'})
```

Let's rename the columns as required by fbprophet. Additioinally, fbprophet doesn't like the index to be a datetime...it wants to see 'ds' as a non-index column, so we won't set an index differnetly than the integer index.

```
df2.set_index('ds').y.plot()
```



df2['y_orig'] = df2['y'] # to save a copy of the original data..you'll see why shortly.
df2['y'] = np.log(df2['y'])
df2.set_index('ds').y.plot()
df2.tail()



By default changepoints are only inferred for the first 80% of the time series in order to have plenty of runway for projecting the trend forward and to avoid overfitting fluctuations at the end of the time series. This default works in many situations but not all, and can be change using the changepoint_range argument. For example, m = Prophet(changepoint_range=0.9) in Python will place potential changepoints in the first 90% of the time series.

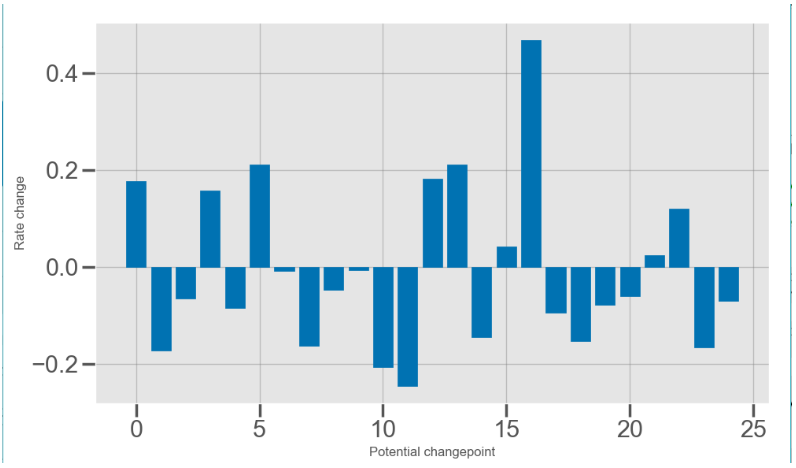
Adjusting trend flexibility

If the trend changes are being overfit (too much flexibility) or underfit (not enough flexibility), you can adjust the strength of the sparse prior using the input argument changepoint_prior_scale. By default, this parameter is set to 0.05. Increasing it will make the trend more flexible:'''
#95-96 .9-.95 -60-80

```
model=Prophet(changepoint_range=0.97, changepoint_prior_scale=0.96, growth='logistic',
              seasonality_mode='additive', interval_width=0.70, yearly_seasonality=True,
weekly_seasonality=True)
model.add_seasonality(name="monthly", period=12, fourier_order=120, prior_scale=0.5)
model.fit(df2);

#create 12 months of future data
future_data = model.make_future_dataframe(periods=60, freq = 'm')
type(future_data)
future_data.tail()
future_data['cap']=30
future_data['floor']=1
forecast_data = model.predict(future_data)
fig = model.plot(forecast_data, uncertainty=True)
```

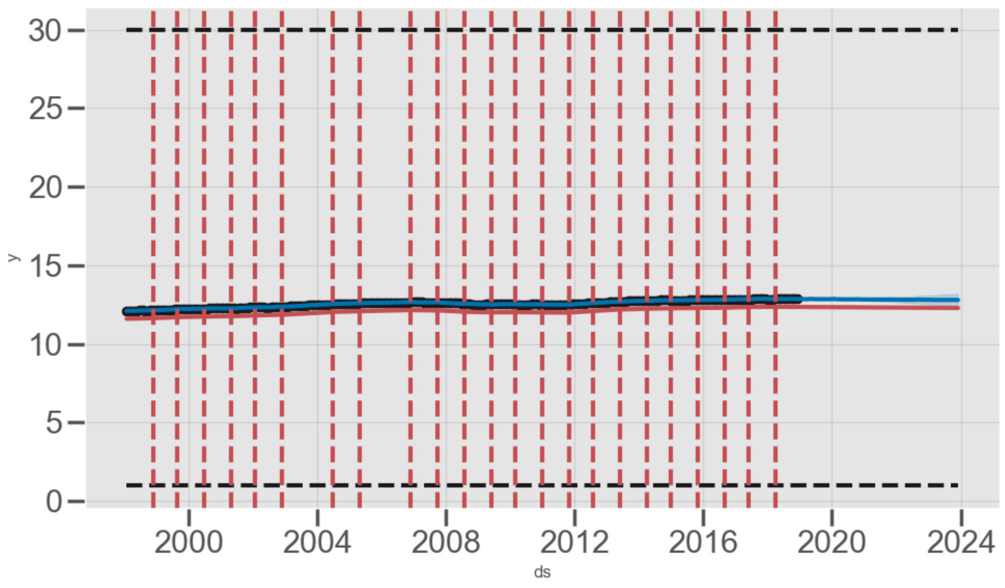
Even though we have a lot of places where the rate can possibly change, because of the sparse prior, most of these changepoints go unused. We can see this by plotting the magnitude of the rate change at each changepoint:

```
for cp in model.changepoints:
   plt.axvline(cp, c='gray', ls='--', lw=2)

deltas = model.params['delta'].mean(0)
fig = plt.figure(facecolor='w', figsize=(10, 6))
ax = fig.add_subplot(111)
ax.bar(range(len(deltas)), deltas, facecolor='#0072B2', edgecolor='#0072B2')
ax.grid(True, which='major', c='gray', ls='-', lw=1, alpha=0.4)
ax.set_ylabel('Rate change')
ax.set_xlabel('Potential changepoint')
fig.tight_layout()
```



The number of potential changepoints can be set using the argument n_changepoints, but this is better tuned by adjusting the regularization. The locations of the signification changepoints can be visualized with:

```
from fbprophet.plot import add_changepoints_to_plot
fig = model.plot(forecast_data)
a = add_changepoints_to_plot(fig.gca(), model, forecast_data)
```



```
forecast_data[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

```
            ds       yhat  yhat_lower  yhat_upper
306  2023-07-31  12.786866   12.435150   13.207405
307  2023-08-31  12.790330   12.420118   13.220739
308  2023-09-30  12.788353   12.407588   13.230428
309  2023-10-31  12.791838   12.401196   13.236067
310  2023-11-30  12.785865   12.386704   13.252775
```

Plotting Prophet results
Prophet has a plotting mechanism called plot. This plot functionality draws the original data (black dots), the model (blue line) and the error of the forecast (shaded blue area).

```
model.plot(forecast_data);
```

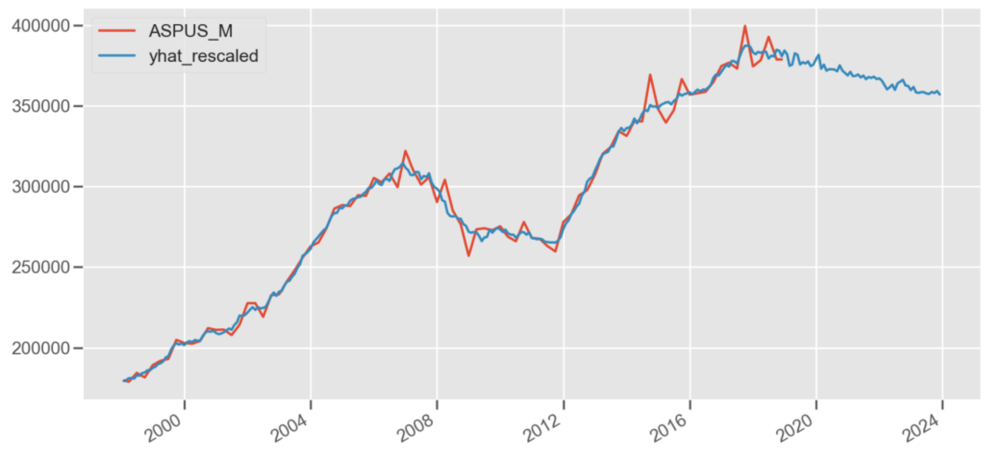
Visualizing Prophet models

In order to build a useful dataframe to visualize our model versus our original data, we need to combine the output of the Prophet model with our original data set, then we'll build a new chart manually using pandas and matplotlib. First, let's set our dataframes to have the same index of ds

```
df2.set_index('ds', inplace=True)
forecast_data.set_index('ds', inplace=True)
df3.set_index('DATE', inplace=True)
viz_df = df3.join(forecast_data[['yhat', 'yhat_lower','yhat_upper']], how = 'outer')
```

If we look at the head(), we see the data has been joined correctly but the scales of our original data (sales) and our model (yhat) are different. We need to rescale the yhat colums(s) to get the same scale, so we'll use numpy's exp function to do that.
viz_df['yhat_rescaled'] = np.exp(viz_df['yhat'])

Let's take a look at the sales and yhat_rescaled data together in a chart.'
viz_df[['ASPUS_M', 'yhat_rescaled']].plot()



We can see downward trends in 60 months for AVE US HOUSING PRICE

You can see from the chart that the model (blue) is pretty good when plotted against the actual signal (orange) but I like to make my vizualization's a little better to understand. To build my 'better' visualization, we'll need to go back to our original df3 and forecast dataframes.

First things first - we need to find the 2nd to last date of the original sales data in sales_df in order to ensure the original sales data and model data charts are connected.

df3.index = pd.to_datetime(df3.index) #make sure our index as a datetime object
connect_date = df3.index[-2] #select the 2nd to last date

Using the connect_date we can now grab only the model data that after that date. To do this, we'll mask the forecast data.

mask = (forecast_data.index > connect_date)
predict_df = forecast_data.loc[mask]

Let's build a dataframe to use in our new visualization. We'll follow the same steps we did before.

viz_df = df3.join(predict_df[['yhat', 'yhat_lower','yhat_upper']], how = 'outer')
viz_df['yhat_scaled']=np.exp(viz_df['yhat'])

If we take a look at the head() of viz_df we'll see 'NaN's everywhere except for our original data rows.

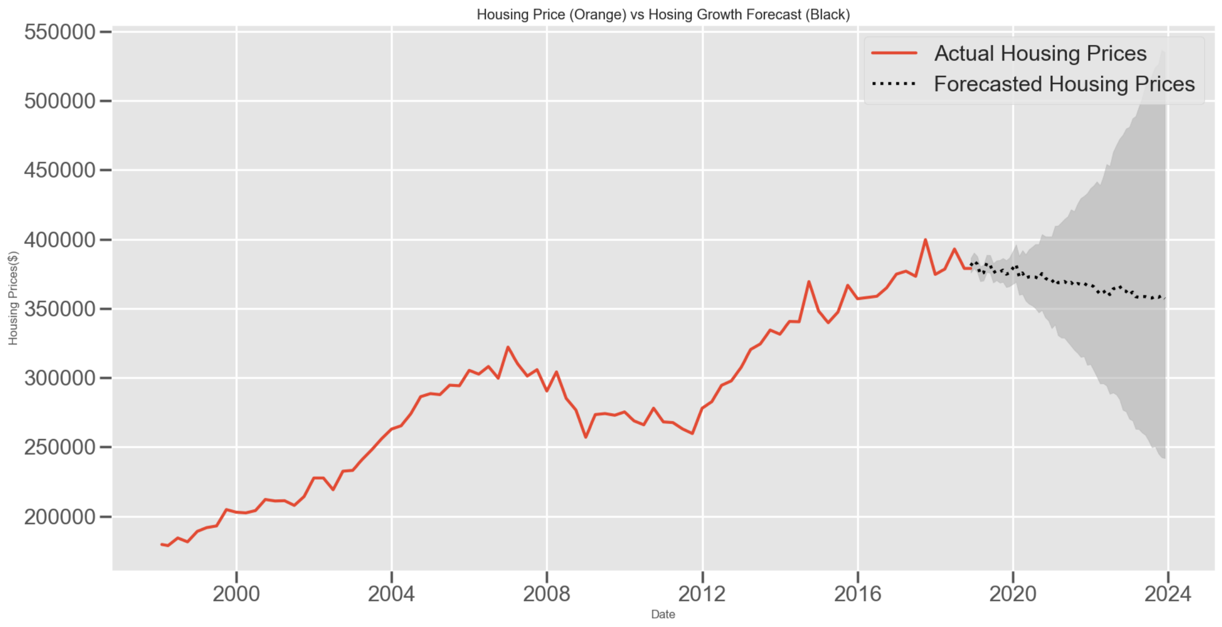
viz_df.head()
Out[492]:

```
            ASPUS_M  yhat  yhat_lower  yhat_upper  yhat_scaled
1998-02-01 179600.00000   nan       nan       nan        nan
1998-03-01 179200.00000   nan       nan       nan        nan
1998-04-01 178800.00000   nan       nan       nan        nan
1998-05-01 180633.33330   nan       nan       nan        nan
1998-06-01 182466.66670   nan       nan       nan        nan
```

```
viz_df.tail()
Out[493]:
          ASPUS_M     yhat  yhat_lower  yhat_upper  yhat_scaled
2023-07-31     nan 12.78687    12.42824    13.15839 357491.21288
2023-08-31     nan 12.79033    12.43049    13.16973 358731.64171
2023-09-30     nan 12.78835    12.41032    13.17335 358023.08677
2023-10-31     nan 12.79184    12.39891    13.19285 359273.03963
2023-11-30     nan 12.78586    12.39647    13.18974 357133.46469
```

Time to plot: let's plot everything to get the 'final' visualization of our monthly housing data and forecast with errors

```
fig, ax1 = plt.subplots()
ax1.plot(viz_df.ASPUS_M)
ax1.plot(viz_df.yhat_scaled, color='black', linestyle=':')
ax1.fill_between(viz_df.index, np.exp(viz_df['yhat_upper']), np.exp(viz_df['yhat_lower']), alpha=0.5,
color='darkgray')
ax1.set_title('Housing Price (Orange) vs Hosing Growth Forecast (Black)')
ax1.set_ylabel('Housing Prices($)')
ax1.set_xlabel('Date')

L=ax1.legend() #get the legend
L.get_texts()[0].set_text('Actual Housing Prices') #change the legend text for 1st plot
L.get_texts()[1].set_text('Forecasted Housing Prices') #change the legend text for 2nd plot
```



This visualization is much better (in my opinion) than the default fbprophet plot. It is much easier to quickly understand and describe what's happening. The orange line is actual sales data and the black dotted line is the forecast. The gray shaded area is the uncertaintity estimation of the forecast.

For next Five Years, we can see Ave Housing Price downward trend. Price range will remain between $248000 and $550000

Forecasting US housing market using Deep Learning (Part 5)

Time series forecasting is challenging, especially when working with long sequences, noisy data, multi-step forecasts and multiple input and output variables.

Deep learning methods offer a lot of promise for time series forecasting, such as the automatic learning of temporal dependence and the automatic handling of temporal structures like trends and seasonality.

Time series data can be phrased as supervised learning.
Given a sequence of numbers for a time series dataset.

- We will discover how to develop a Long Short-Term Memory Neural Network model or LSTM for univariate time series forecasting.
- We can define a simple univariate problem as a sequence of integers, fit the model on this sequence and have the model predict the next value in the sequence.
- The LSTM model expects three-dimensional input with the shape [samples, timesteps, features]. We will define the data in the form [samples, timesteps] and reshape it accordingly.

- We will use one LSTM layer to process each input sub-sequence of 3 time steps, followed by a Dense layer to interpret the summary of the input sequence. The model uses the efficient Adam version of stochastic gradient descent and optimizes the mean squared error (‘mse‘ or 'rmse') loss function.

- Once the model is defined, it can be fit on the training data and the fit model can be used to make a prediction.

```
import keras
import pandas as pd
import sklearn as sk
import tensorflow as tf
```

Import Monthly Housing Price Data

```
h_m_df = pd.read_csv('C:/scripts/capstone2/h_m_df.csv', index_col='DATE', parse_dates=True)
```

The Rectified Linear Activation Function

An "activation function" is a function applied at each node. It converts the node's input into some output. The rectified linear activation function (called ReLU) has been shown to lead to very high-performance networks. This function takes a single number as an input, returning 0 if the input is negative, and the input if the input is positive.

Here are some examples:
relu(3) = 3
relu(-3) = 0
Fill in the definition of the relu() function:
Use the max() function to calculate the value for the output of relu().
Apply the relu() function to node_0_input to calculate node_0_output.
Apply the relu() function to node_1_input to calculate node_1_output.

```
def relu(input):
   '''Define your relu activation function here'''
   # Calculate the value for the output of the relu function: output
   output = max(0, input)

   # Return the value just calculated
   return(output)
```

Specifying a model

Now you'll get to work with your first model in Keras, and will immediately be able to run more complex neural network models on larger datasets compared to the first two chapters.

To start, you'll take the skeleton of a neural network and add a hidden layer and an output layer. You'll then fit that model and see Keras do the optimization so your model continually gets better.

As a start, you'll predict workers wages based on characteristics like their industry, education and level of experience. You can find the dataset in a pandas dataframe called df. For convenience, everything in df except for the target has been converted to a NumPy matrix called predictors. The target, wage_per_hour, is available as a NumPy matrix called target.

we've imported the Sequential model constructor, the Dense layer constructor, and pandas.

Store the number of columns in the predictors data to n_cols.
Start by creating a Sequential model called model.
Use the .add() method on model to add a Dense layer.
Add 50 units, specify activation='relu', and the input_shape parameter to be the tuple (n_cols,) which means it has n_cols items in each row of data, and any number of rows of data are acceptable as inputs.
Add another Dense layer. This should have 32 units and a 'relu' activation.
Finally, add an output layer, which is a Dense layer with a single node. Don't use any activation function here.

```
# Import necessary modules
import keras
from keras.layers import Dense
from keras.models import Sequential
import pandas as pd

housing_df= pd.read_csv('C:/scripts/capstone2/housing_df.csv', index_col=0)

predictors = np.array(housing_df.drop(['ASPUS_3A_PCT_CHG'],1))
target = np.array(housing_df['ASPUS_3A_PCT_CHG'])

print(predictors.shape)
print(target.shape)

# Save the number of columns in predictors: n_cols
n_cols = predictors.shape[1]

# Set up the model: model
model = Sequential()

# Add the first layer
model.add(Dense(50, activation='relu', input_shape=(n_cols,)))

# Add the second layer
model.add(Dense(32, activation='relu'))

# Add the output layer
model.add(Dense(1))
```

Compiling the model

We're now going to compile the model you specified earlier. To compile the model, you need to specify the optimizer and loss function to use. The Adam optimizer is an excellent choice.
Here we'll use the Adam optimizer and the mean squared error loss function.

Compile the model using model.compile(). Your optimizer should be 'adam' and the loss should be 'mean_squared_error'.

Fitting the model

We'll now fit the model. Recall that the data to be used as predictive features is loaded in a NumPy matrix called predictors and the data to be predicted is stored in a NumPy matrix called target. Your model is pre-written and it has been compiled with the code from the previously compiled.

Fit the model. Remember that the first argument is the predictive features (predictors), and the data to be predicted (target) is the second argument.

```
import keras
from keras.layers import Dense
from keras.models import Sequential

# Specify the model
n_cols = predictors.shape[1]
model = Sequential()
model.add(Dense(50, activation='relu', input_shape = (n_cols,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Fit the model
model.fit(predictors, target)
Epoch 1/1
684/684 [==============================] - 20s 30ms/step - loss: 34.9388
```

Classification models

We'll now create a classification model using the **housing_df** dataset. The predictive variables are stored in a NumPy array predictors. The target to predict is in df.survived, though you'll have to manipulate it for keras. The number of predictive features is stored in n_cols.

Here, we'll use the 'sgd' optimizer, which stands for Stochastic Gradient Descent.

Convert df.survived to a categorical variable using the to_categorical() function.
Specify a Sequential model called model.
Add a Dense layer with 32 nodes. Use 'relu' as the activation and (n_cols,) as the input_shape.
Add the Dense output layer. Because there are two outcomes, it should have 2 units, and because it is a classification model, the activation should be 'softmax'.
Compile the model, using 'sgd' as the optimizer, 'categorical_crossentropy' as the loss function, and metrics=['accuracy'] to see the accuracy (what fraction of predictions were correct) at the end of each epoch.
Fit the model using the predictors and the target.

```
# Import necessary modules
import keras
import numpy as np
from keras.layers import Dense, Activation
from keras.models import Sequential
from keras.utils import to_categorical
print( keras.__version__ )

#from keras.activations import softmax
predictors = np.array(housing_df.drop(['ASPUS_3A_PCT_CHG'],1))
target = np.array(housing_df['ASPUS_3A_PCT_CHG'])

print(predictors.shape)
print(target.shape)

# Set up the model
model = Sequential()
#n_cols=10
# Add the first layer
#model.add(Dense(32, activation='relu', input_shape=((10),)))
model.add(Dense(32, activation='relu', input_shape=(n_cols,)))

# Add the output layer
#model.add( Dense(6, input_shape=(6,), activation = 'softmax' ) )
model.add(Dense(1, activation='linear'))

# Compile the model
model.compile(loss='mean_squared_logarithmic_error', optimizer='adam', metrics=['mse'])

# Fit the model
model.fit(predictors, target, epochs=100)
```

```
Epoch 100/100 684/684 [==============================] - 0s 53us/step -
loss: 0.0080 - mean_squared_error: 53.7972
```

Making predictions¶

The trained network from your previous coding exercise is now stored as model. New data to make predictions is stored in a NumPy array as pred_data. Use model to make predictions on your new data.

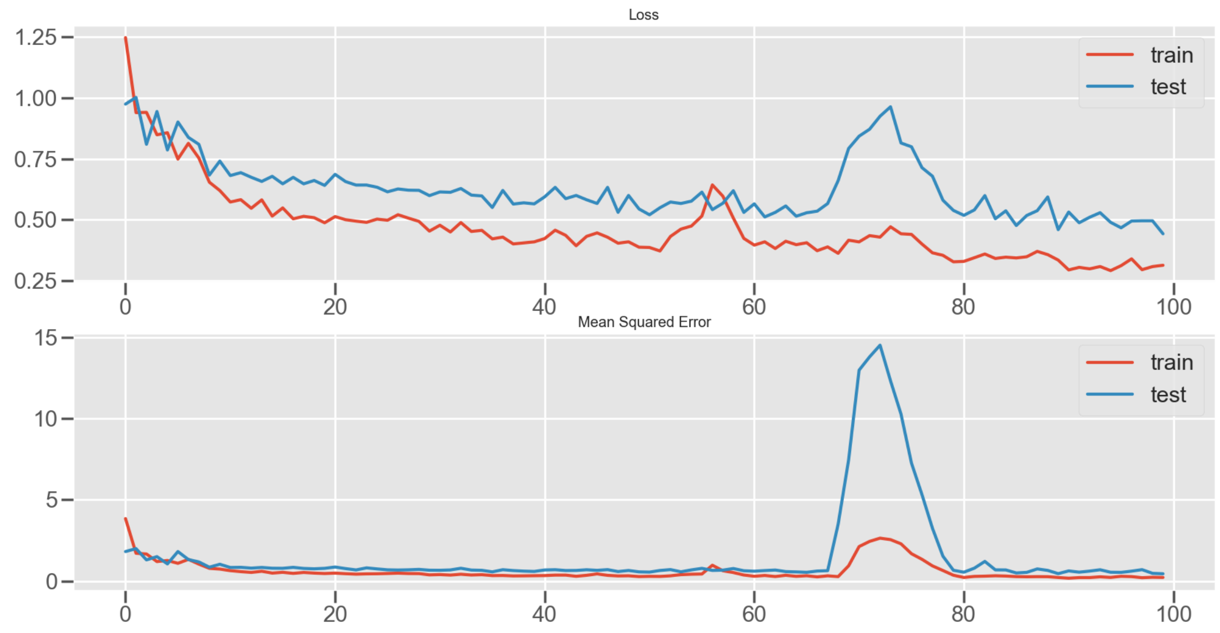
In this exercise, your predictions will be probabilities, which is the most common way for data scientists to communicate their predictions to colleagues.

Create your predictions using the model's .predict() method on pred_data. Use NumPy indexing to find the column corresponding to predicted probabilities of survival being True. This is the second column (index 1) of predictions. Store the result in predicted_prob_true and print it.

mlp for regression with mae loss function

```
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
from sklearn.model_selection import train_test_split # for train and test set split
from sklearn.model_selection import cross_val_score #Sklearn.model_seletion is used

# Construct data for the model
type(housing_df)
#type(model)
housing_df.info()
X = np.array(housing_df.drop(['ASPUS_3A_PCT_CHG'],1))
y = np.array(housing_df['ASPUS_3A_PCT_CHG'])

X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y),1))[:,0]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("size of the training feature set is",X_train.shape)
print("size of the test feature set is",X_test.shape)
print("size of the training Target set is",y_train.shape)
print("size of the test Target set is",y_test.shape)

# standardize dataset
# split into train and test
n_cols=9
# define model
model = Sequential()
model.add(Dense(100, activation='relu', kernel_initializer='he_uniform', input_shape = ((n_cols),)))
model.add(Dense(1, activation='linear'))
```

```
opt = SGD(lr=0.001, momentum=0.99)
model.compile(loss='mean_absolute_error', optimizer=opt, metrics=['mse'])

# fit model
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, verbose=0)
# evaluate the model
_, train_mse = model.evaluate(X_train, y_train, verbose=0)
_, test_mse = model.evaluate(X_test, y_test, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Loss')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot mse during training
pyplot.subplot(212)
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['mean_squared_error'], label='train')
pyplot.plot(history.history['val_mean_squared_error'], label='test')
pyplot.legend()
pyplot.show()
```



Housing price Prediction_Using_CNN

```
h_m_df = pd.read_csv('C:/scripts/capstone2/h_m_df.csv', index_col='DATE', parse_dates=True)

# mlp for regression with mae loss function
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
from sklearn import metrics
from keras.layers.core import Dense, Activation
from keras.callbacks import EarlyStopping

# generate regression dataset
from sklearn.model_selection import train_test_split # for train and test set split
from sklearn.model_selection import cross_val_score #Sklearn.model_seletion is used

# Construct data for the model
X = np.array(h_m_df.drop(['ASPUS_M'],1))
y = np.array(h_m_df['ASPUS_M'])

X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y),1))[:,0]

from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0, 1))
X = sc.fit_transform(X)
y = sc.fit_transform(y.reshape(len(y),1))[:,0]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

print("size of the training feature set is",X_train.shape)
print("size of the test feature set is",X_test.shape)
print("size of the training Target set is",y_train.shape)
print("size of the test Target set is",y_test.shape)
```

Modeling

Now predict the close price for day
m
m
based on data observed in the past 30 days
$\{m-30,-29,...m-1\}$
{m−30,m−29,...m−1}
.

Define Network

Define a Sequential Model and add:
input layer with dimension (30, 6);
two LSTM layers with 256 neurons;
one hidden layers with 32 neurons with 'Relu';

one linear output layer.

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

# Split into train/test
model = Sequential()
model.add(Dense(100, input_dim=X.shape[1], activation='relu')) # Hidden 1
model.add(Dense(50, activation='relu')) # Hidden 2
model.add(Dense(1,activation='softmax')) # Output

model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mse'])
model.fit(X,y,verbose=1,epochs=100)

model.summary()
print(model.summary())
```

```
Epoch 100/100 684/684 [==============================] - 0s 129us/step -
loss: 0.4963 - mean_squared_error: 0.4963

____________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_20 (Dense)             (None, 100)               1000
_________________________________________________________________
dense_21 (Dense)             (None, 50)                5050
_________________________________________________________________
dense_22 (Dense)             (None, 1)                 51
=================================================================
Total params: 6,101
Trainable params: 6,101
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_20 (Dense)             (None, 100)               1000
_________________________________________________________________
dense_21 (Dense)             (None, 50)                5050
_________________________________________________________________
dense_22 (Dense)             (None, 1)                 51
=================================================================
Total params: 6,101
Trainable params: 6,101
Non-trainable params: 0
_________________________________________________________________
None
```

Compile Network¶

```
Before training, configure the learning process by specifying:
Optimizer to be 'adam';
Loss function to be 'mse';
Evaluation metric to be 'accuracy'.

decay = .001
import tensorflow as tf
import keras
adam = keras.optimizers.Adam(decay=decay)
model.compile(loss='mse',optimizer='adam', metrics=['accuracy'])
```

Train Network¶

```
Fit the model to training data to learn the parameters.

model.fit(X_train,y_train,
    batch_size=512,
    epochs=100,
    validation_split=0.2,
    verbose=2)
```

Evaluate Network

```
Evaluate model on the test set
'''
mse, acc = model.evaluate(X_test, y_test)
print("mean square error = ", mse, acc)

#X_test = X_test.reshape(-1,1)
model.compile(optimizer="Nadam", loss="mean_squared_error", metrics=["mean
_squared_error", 'accuracy'])
# enable early stopping based on mean_squared_error
earlystopping=EarlyStopping(monitor="mean_squared_error", patience=40, ver
bose=1, mode='auto')
result1= model.fit(X,y,verbose=2,epochs=120)
#, batch_size=5, validation_data=(X_test, y_test)
# fit model
result = model.fit(X_train,y_train,verbose=2,epochs=120, batch_size=5, val
idation_data=(X_test, y_test), callbacks=[earlystopping])
# get predictions
y_pred = model.predict(X_test)


# root mean squared error (rmse) for regression
def rmse(y_true, y_pred):
    from keras import backend
    return backend.sqrt(backend.mean(backend.square(y_pred - y_true), axis
=-1))

# mean squared error (mse) for regression
def mse(y_true, y_pred):
    from keras import backend
    return backend.mean(backend.square(y_pred - y_true), axis=-1)

# coefficient of determination (R^2) for regression
def r_square(y_true, y_pred):
    from keras import backend as K
    SS_res =  K.sum(K.square(y_true - y_pred))
    SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return (1 - SS_res/(SS_tot + K.epsilon()))

def r_square_loss(y_true, y_pred):
    from keras import backend as K
    SS_res =  K.sum(K.square(y_true - y_pred))
    SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return 1 - ( 1 - SS_res/(SS_tot + K.epsilon()))
```

LSTM-RNN to forecast time-series

RNN's (LSTM's) are pretty good at extracting patterns in input feature space, where the input data spans over long sequences. Given the gated architecture of LSTM's that has this ability to manipulate its memory state, they are ideal for such problems.

- LSTMs can almost seamlessly model problems with multiple input variables. All we need is a 3D input vector that needs to be fed into the input shape of the LSTM. So long as we figure out a way to convert all our input variables to be represented in a 3D vector form, we are good use LSTM. This adds a great benefit in time series forecasting, where classical linear methods can be difficult to adapt to multivariate or multiple input forecasting problems (A side note here for multivariate forecasting — keep in mind that when we use multivariate data for forecasting, then we also need "future multi-variate" data to predict the future outcome!)
- In general, while using LSTM's, I found that they offer lot of flexibility in modelling the problem — meaning we have a good control over several parameters of the time series.

```
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
from keras.callbacks import EarlyStopping
# generate regression dataset
from sklearn.model_selection import train_test_split # for train and test set split
from sklearn.model_selection import cross_val_score

import pandas as pd
import numpy as np
h_m_df = pd.read_csv('C:/scripts/capstone2/h_m_df.csv')
X = np.array(h_m_df.drop(['ASPUS_M'],1))
y = np.array(h_m_df['ASPUS_M'])

n_sample = h_m_df.shape[0]

n_train=int(0.8*n_sample)+1
n_forecast=n_sample-n_train
#ts_df
dataset_train = h_m_df.iloc[:n_train]['ASPUS_M']
dataset_test = h_m_df.iloc[n_train:]['ASPUS_M']
print(dataset_train.shape)
print(dataset_test.shape)
print("Training Series Tail:", "\n", dataset_train.tail(), "\n")
print("Testing Series Head:", "\n", dataset_test.head())

# Importing the training set & Test Set
# Data Scaling

training_set = dataset_train.values.reshape(-1, 1)
test_set = dataset_test.values.reshape(-1, 1)
# Feature Scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0, 1))
training_set_scaled = sc.fit_transform(training_set)

from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0, 1))
test_set_scaled = sc.fit_transform(test_set)

# Creating a 3D data structure with input and output
X_train = []
y_train = []
for i in range(6, 548):
   X_train.append(training_set_scaled[i-6:i, 0])
   y_train.append(training_set_scaled[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)

# Reshaping
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))

X_test = []
y_test = []
for i in range(6, 136):
   X_test.append(test_set_scaled[i-6:i, 0])
   y_test.append(test_set_scaled[i, 0])
X_test, y_test = np.array(X_test), np.array(y_test)
```

```
# Reshaping
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))


print("size of the training feature set is",X_train.shape)
print("size of the test feature set is",X_test.shape)
print("size of the training Target set is",y_train.shape)
print("size of the test Target set is",y_test.shape)

# Building the RNN

# Importing the Keras libraries and packages
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers.normalization import BatchNormalization
from keras.layers import LeakyReLU

from keras import backend as K

# root mean squared error (rmse) for regression (only for Keras tensors)
def rmse(y_true, y_pred):
    from keras import backend
    return backend.sqrt(backend.mean(backend.square(y_pred - y_true), axis=-1))

# mean squared error (mse) for regression  (only for Keras tensors)
def mse(y_true, y_pred):
    from keras import backend
    return backend.mean(backend.square(y_pred - y_true), axis=-1)


def r_sq(y_true, y_pred):
    SS_res =  K.sum(K.square( y_true-y_pred ))
    SS_tot = K.sum(K.square( y_true - K.mean(y_true) ) )
    return ( 1 - SS_res/(SS_tot + K.epsilon()) )

decay = .001
import tensorflow as tf
import keras
adam = keras.optimizers.Adam(decay=decay)

# Initialising the RNN
regressor = Sequential()
#regressor.add(BatchNormalization())
# Adding the first LSTM layer and some Dropout regularisation
```

```
regressor.add(LSTM(units = 200, activation='relu', return_sequences = True, input_shape =
(X_train.shape[1], 1)))
regressor.add(Dropout(0.3))


# now add a ReLU layer explicitly:
regressor.add(LeakyReLU(alpha=0.05))

# Adding a second LSTM layer and some Dropout regularisation
regressor.add(LSTM(units = 150,  return_sequences = True)) #activation='sigmoid'
regressor.add(Dropout(0.3))

# Adding a third LSTM layer and some Dropout regularisation
regressor.add(LSTM(units = 100, return_sequences = True))
regressor.add(Dropout(0.3))

# Adding a fourth LSTM layer and some Dropout regularisation
regressor.add(LSTM(units = 50))
regressor.add(Dropout(0.3))

# Adding the output layer
regressor.add(Dense(units = 1, activation='sigmoid')) #, activation='sigmoid'

# Compiling the RNN

regressor.compile(optimizer='adam', loss=rmse, metrics=[rmse, r_sq])
# enable early stopping based on mean_squared_error
earlystopping=EarlyStopping(monitor="r_sq", patience=100, verbose=1, mode='auto')
# fit model validation_data=(X_test, y_test)
result = regressor.fit(X_train, y_train, epochs=1000, batch_size=32, validation_data=(X_test, y_test),
callbacks=[earlystopping])

# Fitting the RNN to the Training set
#regressor.fit(X_train, y_train, epochs = 20, batch_size = 32)

print(regressor.summary())
```

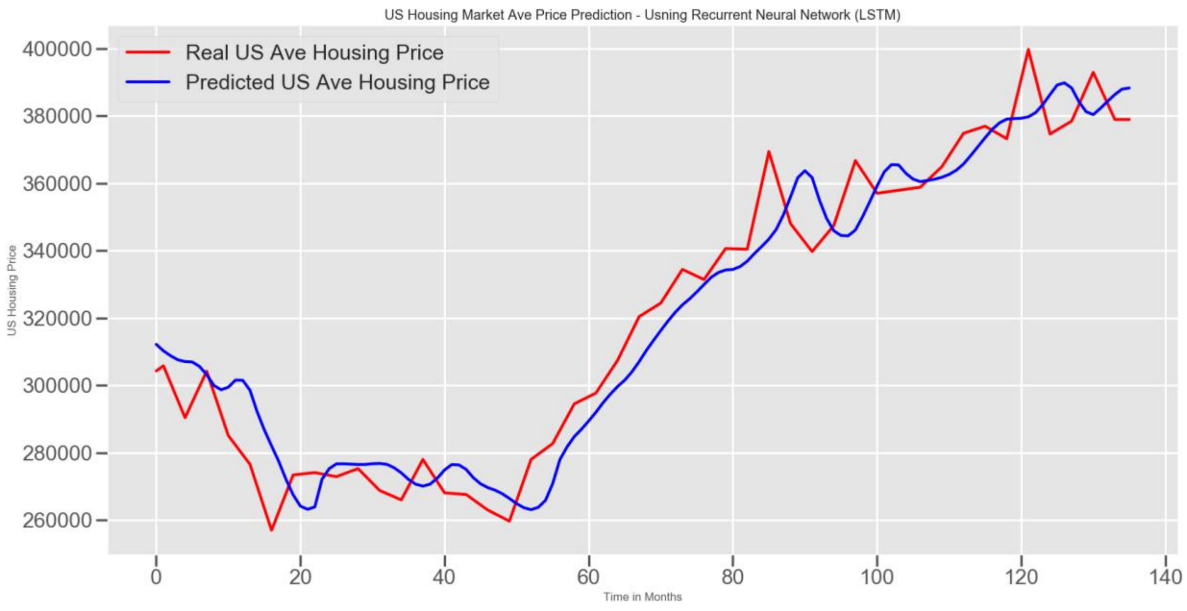
```
Epoch 101/1000 542/542 [==============================] - 1s 2ms/step -
loss: 0.0157 - rmse: 0.0157 - r_sq: 0.9935 - val_loss: 0.0595 - val_rmse:
0.0595 - val_r_sq: -1553.0401 Epoch 00101: early stopping
_________________________________________________________________ Layer
(type) Output Shape Param #
================================================================= lstm_1
(LSTM) (None, 6, 200) 161600
_________________________________________________________________
dropout_1 (Dropout) (None, 6, 200) 0
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU) (None, 6, 200) 0
_________________________________________________________________ lstm_2
```

```
(LSTM) (None, 6, 150) 210600
_________________________________________________________________
dropout_2 (Dropout) (None, 6, 150) 0
_________________________________________________________________ lstm_3
(LSTM) (None, 6, 100) 100400
_________________________________________________________________
dropout_3 (Dropout) (None, 6, 100) 0
_________________________________________________________________ lstm_4
(LSTM) (None, 50) 30200
_________________________________________________________________
dropout_4 (Dropout) (None, 50) 0
_________________________________________________________________ dense_26
(Dense) (None, 1) 51
================================================================= Total
params: 502,851 Trainable params: 502,851 Non-trainable params: 0
_________________________________________________________________ None
```

```
# root mean squared error (rmse) for regression (only for Keras tensors)
# Get accuracy of model on validation data. It's not AUC but it's something at least!

score = regressor.evaluate(X_test,y_test, batch_size=32)
print('Test accuracy:', score)
# Part 3 - Making the predictions and visualising the results

# Getting the real Housing price of 2017

real_housing_price = test_set
#dataset_train['ASPUS_M']
# Getting the predicted Housing price of 2017
dataset_total = pd.concat((dataset_train, dataset_test), axis = 0)
inputs = dataset_total[len(dataset_total) - len(dataset_test) - 6:].values
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)
X_test = []
for i in range(6, 142):
   X_test.append(inputs[i-6:i, 0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
predicted_housing_price = regressor.predict(X_test)
predicted_housing_price = sc.inverse_transform(predicted_housing_price)

print(predicted_housing_price.shape)
```

Visualising the results

```
plt.plot(real_housing_price, color = 'red', label = 'Real US Ave Housing Price')
plt.plot(predicted_housing_price, color = 'blue', label = 'Predicted US Ave Housing Price')
plt.title('US Housing Market Ave Price Prediction - Usning Recurrent Neural Network (LSTM)')
plt.xlabel('Time in Months')
plt.ylabel('US Housing Price')
plt.legend()
plt.show()
```

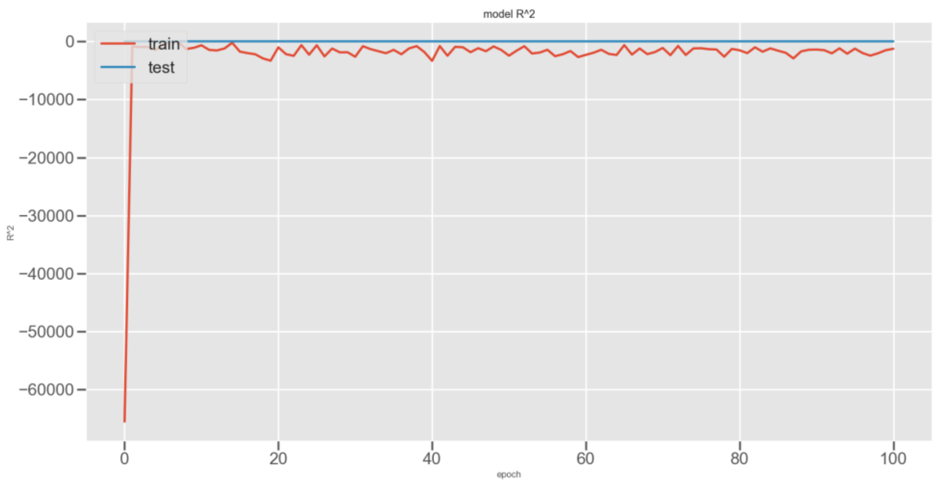


R^2 is a statistic that will give some information about the goodness of fit of a model. In regression, the R^2 coefficient of determination is a statistical measure of how well the regression predictions approximate the real data points. An R^2 of 1 indicates that the regression predictions perfectly fit the data.

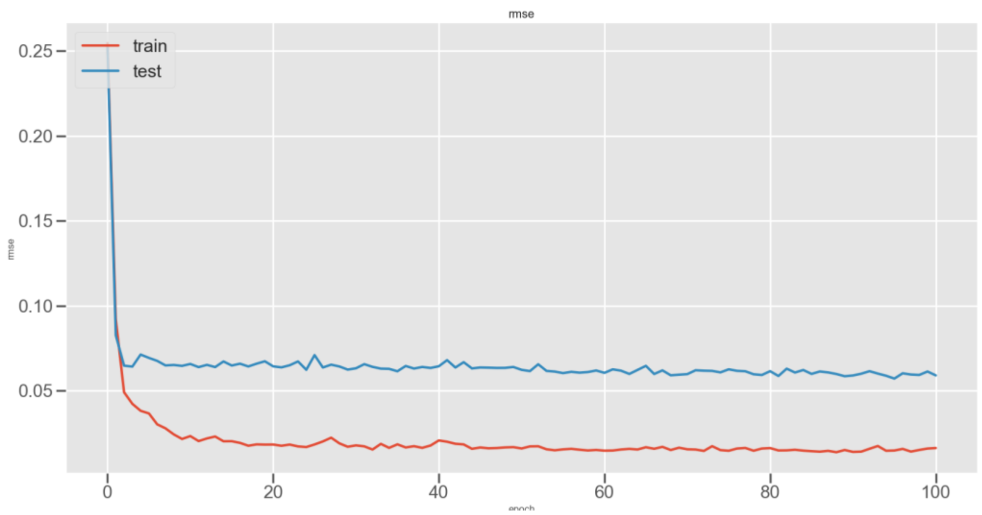
Values of R^2 outside the range 0 to 1 can occur when the model fits the data worse than a horizontal hyperplane. This would occur when the wrong model was chosen, or nonsensical constraints were applied by mistake. If equation 1 of Kvålseth[11] is used (this is the equation used most often), R^2 can be less than zero.

Both Actual price and Predicted price is indicating that overheated housing market is slowing down. Pick Ave Housing Price was $400000 and now down to $378000. We can see 5.5% down from High Price.

```
--------------------------
# Plot learning curves including R^2 and RMSE
#---------------------------------------------------------------------------

# plot training curve for R^2 (beware of scale, starts very low negative)
plt.plot(result.history['val_r_sq'])
plt.plot(result.history['r_sq'])
plt.title('model R^2')
plt.ylabel('R^2')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



```
# plot training curve for rmse
plt.plot(result.history['rmse'])
plt.plot(result.history['val_rmse'])
plt.title('rmse')
plt.ylabel('rmse')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



Simple TensorFlow Classification: Housing 3 Years growth data set
This is a very simple example of how to perform the Housing 3 Years growth classification using TensorFlow

```
from tensorflow import set_random_seed
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.callbacks import EarlyStopping
from keras.layers import Dense
import matplotlib.pyplot as plt
import numpy as np

import random
# set the seeds for reproducible results with TF (wont work with GPU, only CPU)
np.random.seed(12345)
# set the TF seed
set_random_seed(12345)
# Import data, assign seed for same results, do train/test split 80/20

from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
from keras import backend as K

def r_sq(y_true, y_pred):
   SS_res =  K.sum(K.square( y_true-y_pred ))
   SS_tot = K.sum(K.square( y_true - K.mean(y_true) ) )
   return ( 1 - SS_res/(SS_tot + K.epsilon()) )


# generate regression dataset
from sklearn.model_selection import train_test_split # for train and test set split
from sklearn.model_selection import cross_val_score #Sklearn.model_seletion is used

# Construct data for the model
type(housing_df)
#type(model)
housing_df.info()
X = np.array(housing_df.drop(['ASPUS_3A_PCT_CHG'],1))
y = np.array(housing_df['ASPUS_3A_PCT_CHG'])

X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y),1))[:,0]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

print("size of the training feature set is",X_train.shape)
print("size of the test feature set is",X_test.shape)
print("size of the training Target set is",y_train.shape)
print("size of the test Target set is",y_test.shape)

# built Keras sequential model
model = Sequential()
```

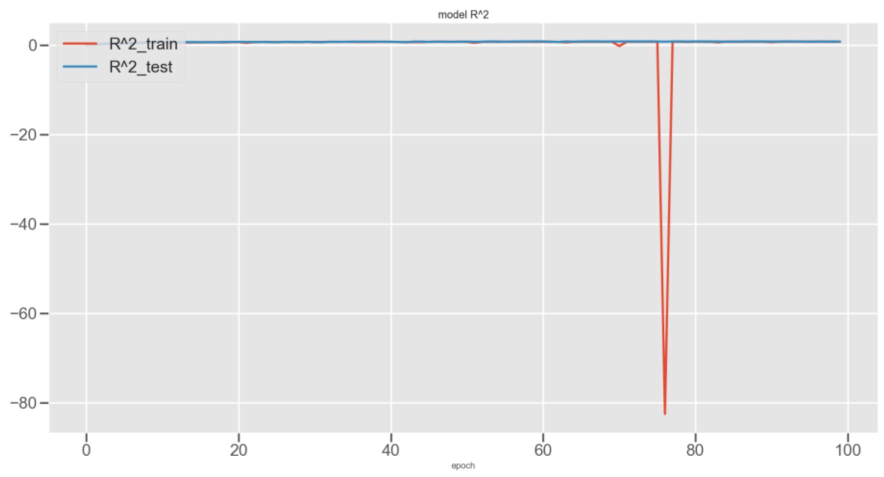
```
# add batch normalization
model.add(BatchNormalization())  # add layer to the MLP for data (404,13)

model.add(Dense(100, input_dim=X.shape[1], activation='relu')) # Hidden 1
model.add(Dense(50, activation='relu')) # Hidden 2
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
# compile regression model loss should be mean_squared_error //
model.compile(optimizer="adam", loss=rmse, metrics=[rmse, r_square])
# enable early stopping based on mean_squared_error
earlystopping=EarlyStopping(monitor="mean_squared_error", patience=40, verbose=2, mode='auto')
#result1= model.fit(X_train, y_train,verbose=2,epochs=500)
result1 = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test),
callbacks=[earlystopping])
```

`Epoch 200/200 547/547 [==============================] - 0s 181us/step - loss: 0.2716 - rmse: 0.2716 - r_square: 0.8139 - val_loss: 0.2153 - val_rmse: 0.2153 - val_r_square: 0.8609`

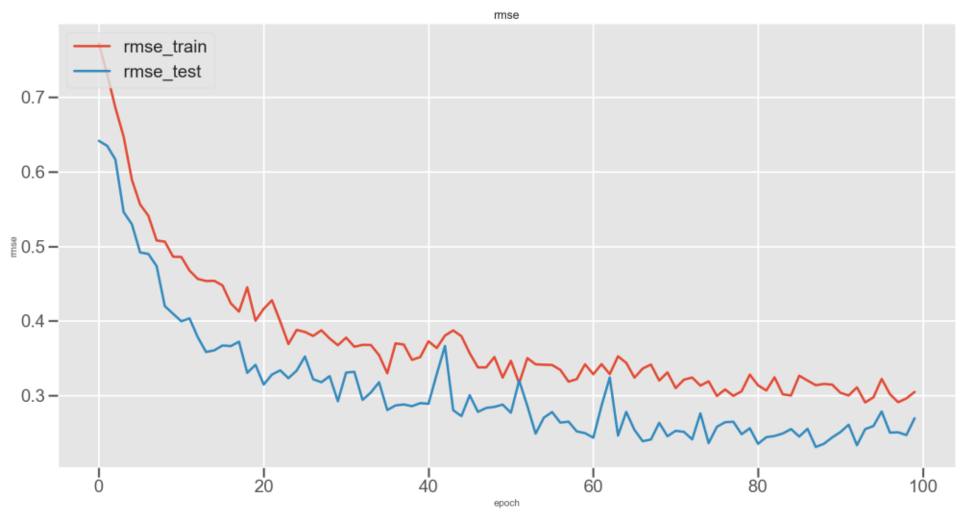
```
y_pred = model.predict(X_test)
score = model.evaluate(X_test,y_test, batch_size=32)
print('Test accuracy:', score)
```

Test accuracy: [0.26940978497919377, 0.26940978497919377, 0.7912574997783577]
Accuracy is 80%

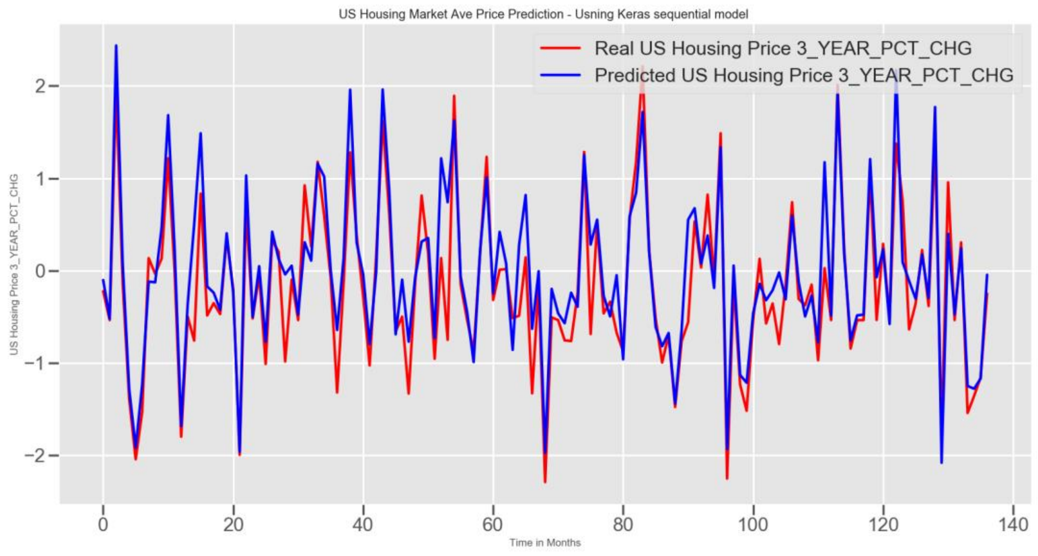
```
#----------------------------------------------------------------------------
# Plot learning curves including R^2 and RMSE
#----------------------------------------------------------------------------
plt.plot(result1.history['r_square'])
plt.plot(result.history['val_r_square'])
plt.title('model R^2')
plt.ylabel('R^2')
plt.xlabel('epoch')
plt.legend(['R^2_train', 'R^2_test'], loc='upper left')
plt.show()
```



```
# plot training curve for rmse
plt.plot(result1.history['rmse'])
plt.plot(result1.history['val_rmse'])
plt.title('rmse')
plt.ylabel('rmse')
plt.xlabel('epoch')
plt.legend(['rmse_train', 'rmse_test'], loc='upper left')
plt.show()
```



```
plt.plot(y_test, color = 'red', label = 'Real US Housing Price 3_YEAR_PCT_CHG')
plt.plot(y_pred, color = 'blue', label = 'Predicted US Housing Price 3_YEAR_PCT_CHG')
plt.title('US Housing Market Ave Price Prediction - Usning Keras sequential model ')
plt.xlabel('Time in Months')
plt.ylabel('US Housing Price 3_YEAR_PCT_CHG')
plt.legend()
plt.show()
```



Summary Report:

Compare US average housing price with stock market, bonds and interest rates: Even though, housing market is over heated, Strong Stock Market, Bond and low interest are driving housing market higher.

Compare US average housing price with Consumer expenditures, Unemployment market and US GDP Growth: US GDP has improved, Personal Consumer expenditures is also high at this moment. Even though, US Long run unemployment rate is very low at this moment (Helping Housing market), it may change quickly, once housing market starting to go down rapidly. As we have seen unemployment rate jump quickly (2007, 1982) as soon as stock market or housing market goes down.

Compare US average housing price with Housing Supply Ratio, New 1F house sold in US and US Construction Permit: These are the key parameter directly related to US Housing Market which can move housing sharply. Historically, we have seen that, housing supply ratio is inversely proportional to Housing prices. Both numbers of new 1F houses sold & number of construction permits directly proportional to housing price. Recently supply ratio is increasing, both number of new 1F houses sold & number of construction permits are dropping slowly. All three indicators strongly suggesting that housing market is slowing down.

Forecasting using ARIMA model, FBProphet and Deep Learning, are clearly showing that downtrend in housing market. Ave Housing went down only 5.5%, which may be simple correction. It is not a Housing market crash yet but it may moves towards the market crash. FED is holding interest rate low, if interest rates go up slightly and unemployment rate goes over 6% which could be bad news for housing market.