COMP 352 Final Project Example

Author: Daniel Matlock

Date: 1/30/2024

Housing Prices

Competition: https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques

Full dataset: https://www.kaggle.com/datasets/marcopale/housing

The Ames Housing dataset was compiled by Dean De Cock for use in data science education. It's an incredible alternative for data scientists looking for a modernized and expanded version of the often cited Boston Housing dataset.

Final Project Requirements:

There are four sections of the final project. You are expected to perform the following tasks within each section to fulfill the project requirements.

- **Data Importing and Pre-processing (50 Points)**
    - Import dataset and describe characteristics such as dimensions, data types, file types, and import methods used
    - Clean, wrangle, and handle missing data
    - Transform data appropriately using techniques such as aggregation, normalization, and feature construction
    - Reduce redundant data and perform need based discretization
- **Data Analysis and Visualization (50 Points)**
    - Identify categorical, ordinal, and numerical variables within data
    - Provide measures of centrality and distribution with visualizations
    - Diagnose for correlations between variables and determine independent and dependent variables
    - Perform exploratory analysis in combination with visualization techniques to discover patterns and features of interest
- **Data Analytics (50 Points)**
    - Determine the need for a supervised or unsupervised learning method and

identify dependent and independent variables
    - Train, test, and provide accuracy and evaluation metrics for model results
- **Presentation (50 Points)**
    - In a 5 to 10 minute slde presentation, briefly explain the project workflow from the code and results in your markdown notebook State your findings from the data and provide the interpretation of results from your analysis at each stage in the project

Table of Contents:

Data Importing and Pre-processing

In [1]:

```
# import libraries needed
import pandas as pd

pd.set_option("display.max_columns", None)
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from scipy.stats import norm, skew, probplot
from scipy.special import boxcox1p
import warnings
from datetime import datetime

warnings.filterwarnings("ignore")
warnings.filterwarnings("ignore", category=FutureWarning, module="pandas.*")
%matplotlib inline
```

In [2]:

```
# read in file
# housing_df = pd.read_csv('house_prices/train.csv')
housing_df = pd.read_csv("house_prices/AmesHousing.csv")
```

In [3]:

```
# check number of rows and columns
housing_df.shape
```

Out[3]:

```
(2930, 82)
```

In [4]:

```
# count the number of categorical variables
cat_count = 0
```

```
for dtype in housing_df.dtypes:
    if dtype == "object":
        cat_count = cat_count + 1
```

In [5]:

```
print("# of categorical variables:", cat_count)

numeric_vars = housing_df.shape[1] - cat_count - 1
print(
    "# of contineous variables:", numeric_vars
)  # subtract and extra column as 1 column is an ID column
```

```
# of categorical variables: 43
# of contineous variables: 38
```

In [6]:

```
housing_df.head()
```

Out[6]:

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	L Cont
0	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	
1	2	526350040	20	RH	80.0	11622	Pave	NaN	Reg	
2	3	526351010	20	RL	81.0	14267	Pave	NaN	IR1	
3	4	526353030	20	RL	93.0	11160	Pave	NaN	Reg	
4	5	527105010	60	RL	74.0	13830	Pave	NaN	IR1	

In [7]:

```
# Drop the 'order' column and rename the 'PID' column to 'Id'
housing_df = housing_df.drop(columns=["Order"])
housing_df = housing_df.rename(columns={"PID": "Id"})
```

In [8]:

```
# check the column names
housing_df.columns
```

Out[8]:

```
Index(['Id', 'MS SubClass', 'MS Zoning', 'Lot Frontage', 'Lot Area', 'Stree
t',
       'Alley', 'Lot Shape', 'Land Contour', 'Utilities', 'Lot Config',
       'Land Slope', 'Neighborhood', 'Condition 1', 'Condition 2', 'Bldg Ty
pe',
       'House Style', 'Overall Qual', 'Overall Cond', 'Year Built',
       'Year Remod/Add', 'Roof Style', 'Roof Matl', 'Exterior 1st',
       'Exterior 2nd', 'Mas Vnr Type', 'Mas Vnr Area', 'Exter Qual',
       'Exter Cond', 'Foundation', 'Bsmt Qual', 'Bsmt Cond', 'Bsmt Exposur
e',
       'BsmtFin Type 1', 'BsmtFin SF 1', 'BsmtFin Type 2', 'BsmtFin SF 2',
       'Bsmt Unf SF', 'Total Bsmt SF', 'Heating', 'Heating QC', 'Central Ai
r',
       'Electrical', '1st Flr SF', '2nd Flr SF', 'Low Qual Fin SF',
       'Gr Liv Area', 'Bsmt Full Bath', 'Bsmt Half Bath', 'Full Bath',
       'Half Bath', 'Bedroom AbvGr', 'Kitchen AbvGr', 'Kitchen Qual',
       'TotRms AbvGrd', 'Functional', 'Fireplaces', 'Fireplace Qu',
       'Garage Type', 'Garage Yr Blt', 'Garage Finish', 'Garage Cars',
       'Garage Area', 'Garage Qual', 'Garage Cond', 'Paved Drive',
       'Wood Deck SF', 'Open Porch SF', 'Enclosed Porch', '3Ssn Porch',
       'Screen Porch', 'Pool Area', 'Pool QC', 'Fence', 'Misc Feature',
       'Misc Val', 'Mo Sold', 'Yr Sold', 'Sale Type', 'Sale Condition',
       'SalePrice'],
      dtype='object')
```

In [9]:

```
# Remove spaces from column names
housing_df.columns = [col.replace(" ", "") for col in housing_df.columns]
```

Handling missing data

In [10]:

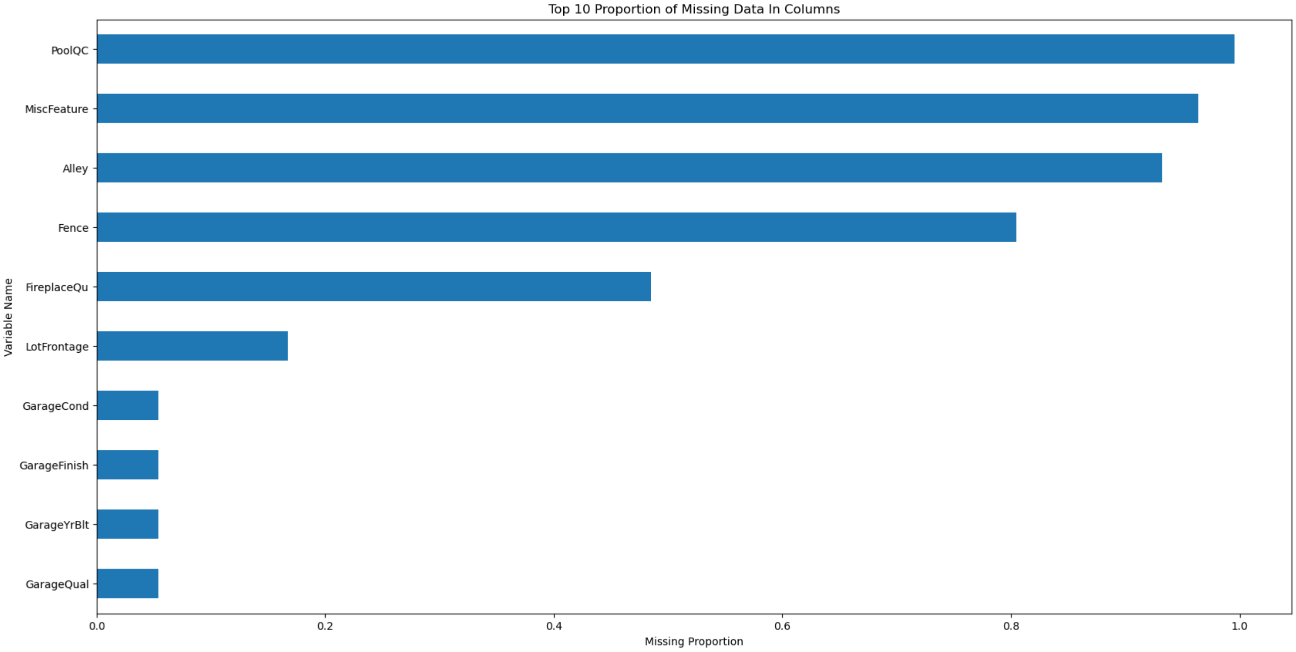
```
# missing data
total = housing_df.isnull().sum().sort_values(ascending=False)
percent = (housing_df.isnull().sum() / housing_df.isnull().count()).sort_val
    ascending=False
)
missing_data = pd.concat([total, percent], axis=1, keys=["Total", "Percent"]
missing_data.head(20)
```

Out[10]:

	Total	Percent
PoolQC	2917	0.995563
MiscFeature	2824	0.963823
Alley	2732	0.932423
Fence	2358	0.804778
FireplaceQu	1422	0.485324
LotFrontage	490	0.167235
GarageCond	159	0.054266
GarageFinish	159	0.054266
GarageYrBlt	159	0.054266
GarageQual	159	0.054266
GarageType	157	0.053584
BsmtExposure	83	0.028328
BsmtFinType2	81	0.027645
BsmtQual	80	0.027304
BsmtCond	80	0.027304
BsmtFinType1	80	0.027304
MasVnrArea	23	0.007850
MasVnrType	23	0.007850
BsmtHalfBath	2	0.000683
BsmtFullBath	2	0.000683

In [11]:

```
missing_data["Percent"].head(10).plot(
    kind="barh", figsize=(20, 10)
).invert_yaxis()  # top 10 missing columns
plt.xlabel("Missing Proportion")
plt.ylabel("Variable Name")
plt.title("Top 10 Proportion of Missing Data In Columns")
plt.show()
```



In [12]:
```
# Columns to fill with 'None'
# These fields have low cardinatlity and adding a class "None" allows us to
columns_to_fill_none = [
    "PoolQC",
    "MiscFeature",
    "Alley",
    "Fence",
    "FireplaceQu",
    "GarageType",
    "GarageFinish",
    "GarageQual",
    "GarageCond",
    "BsmtQual",
    "BsmtCond",
    "BsmtExposure",
    "BsmtFinType1",
    "BsmtFinType2",
    "MasVnrType",
    "MSSubClass",
]

for col in columns_to_fill_none:
    housing_df[col] = housing_df[col].fillna("None")
```

In [13]:
```
# Columns to fil with 0
# Where these fields are NULL, the houses do not have these items (i.e. that
# We distinguish these in the data by imputing them all as NULL so they will
numeric_cols = [
    "MasVnrArea",
    "GarageYrBlt",
    "GarageArea",
    "GarageCars",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtUnfSF",
    "TotalBsmtSF",
    "BsmtFullBath",
    "BsmtHalfBath",
]

for col in numeric_cols:
    housing_df[col] = housing_df[col].fillna(0)
```

In [14]:
```
# These fields are missing data that should be present
# We want to impute this data as best as possible
# Housing is largely geo-spatially similar (i.e. the neighbors house has a h
# So a good way to impute this missing data is ot use the median value from
housing_df["LotFrontage"] = housing_df.groupby("Neighborhood")["LotFrontage"
    lambda x: x.fillna(x.median())
)

# Some neighborhoods have NULL for all entries of LotFrontage
# For these, we use the entire datasets median value
# Another approach would be to use the median value from all neighborhoods t
# The approach would be to visualize the neighborhoods and manually check wh
housing_df["LotFrontage"] = housing_df["LotFrontage"].fillna(
    housing_df["LotFrontage"].median()
)
```

IMPORTANT QUESTION

Is imputing missing values using the median from the entire dataset data leakage?

Remember *data leakage* is the following:

- Unintentional or improper exposure of information from the training data to the model during the training process.
- It occurs when information that would not be available in a real-world scenario is used in the training set, leading the model to learn patterns that do not generalize well.

So the question for this example is:

- Can we impute the values of *LotFrontage* using the whole dataset, or can we only use those values in the training dataset?

The answer is it depends on how the data is sourced. Let's follow these 2 scenarios:

- **Scenario 1:** Suppose for our data we have both a housing property characteristics dataset and housing sales dataset.
    - In this scenario, **YES, you can impute the median of *LotFrontage* using the whole dataset**.
    - The reason you can do this is because you will have all housing property characteristics at all times for both homes that sold and did not sell. Your dataset will consist of all known properties and its characteristics and will most likely be joined to a sales dataset allowing you to impute missing values using the median from the entire dataset.
- **Scenario 2:** Suppose for our data we have only a housing sales dataset that includes the property characteristics.
    - In this scenario, **NO, you cannot impute the median of *LotFrontage* using the whole dataset**.
    - The reason for this is because you ONLY have sales data and thus you do not have all properties to use to impute missing values using the median. You can use the homes that have sold before the home in question to impute the median, but you could not use any homes after that. This is because you would not have this information at the time of building the model.

In the real world, I believe scenario 1 is more likely so I chose to presume this to be the case. However, I would not be surprised if scenario 2 is how some companies build their models.

In [15]:

```
# Missing categorical data in which we impute the mode (most common value)
# This is done because columns have low cardinality so the mode makes sense
# Imputing these values as a new value called "None" would be appropriate as
# You could train 2 models: 1 using the mode, and another using "None" and c
columns_to_fill = [
    "Functional",
    "Electrical",
    "KitchenQual",
    "Exterior1st",
    "Exterior2nd",
    "SaleType",
    "MSZoning",
]

for column in columns_to_fill:
    housing_df[column] = housing_df[column].fillna(housing_df[column].mode()
```

In [16]:

```
# We are checking to see if any columns are largely homogeneous
# Homogeneous columns provide no value to our model and can sometimes lead t
print("Categorical Columns Mode Frequency")

cat_columns = housing_df.select_dtypes(include=["object"]).columns

mode_freq_df = pd.DataFrame(columns=["column_name", "mode_frequency"])

for col in cat_columns:
    mode_value = housing_df[col].mode().values[0]
    mode_freq = (housing_df[col] == mode_value).mean()
    row_data = [{"column_name": col, "mode_frequency": mode_freq}]
    mode_freq_df = mode_freq_df.append(row_data, ignore_index=True)

mode_freq_df.sort_values("mode_frequency", ascending=False).head(10)
```

```
Categorical Columns Mode Frequency
```

Out[16]:

	column_name	mode_frequency
5	Utilities	0.998976
1	Street	0.995904
38	PoolQC	0.995563
10	Condition2	0.989761
14	RoofMatl	0.985324
26	Heating	0.984642
40	MiscFeature	0.963823
7	LandSlope	0.951877
28	CentralAir	0.933106
2	Alley	0.932423

In [17]:

```
# We see that the "Utilities" field is almost entirely homogeneous
housing_df["Utilities"].value_counts()
```

Out[17]:

```
AllPub    2927
NoSewr       2
NoSeWa       1
Name: Utilities, dtype: int64
```

In [18]:

```
# Street is largely homogeneous but still has some value because it only has
housing_df["Street"].value_counts()
```

Out[18]:

```
Pave    2918
Grvl      12
Name: Street, dtype: int64
```

In [19]:

```python
# Drop "Utilities" column as it is too homogeneous
housing_df = housing_df.drop(["Utilities"], axis=1)
```

In [20]:

```python
# Check remaining missing values if any
all_data_na = housing_df.isnull().sum() / len(housing_df)
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_val
    ascending=False
)
missing_data = pd.DataFrame({"Missing Ratio": all_data_na})
missing_data.head()
```

Out[20]:

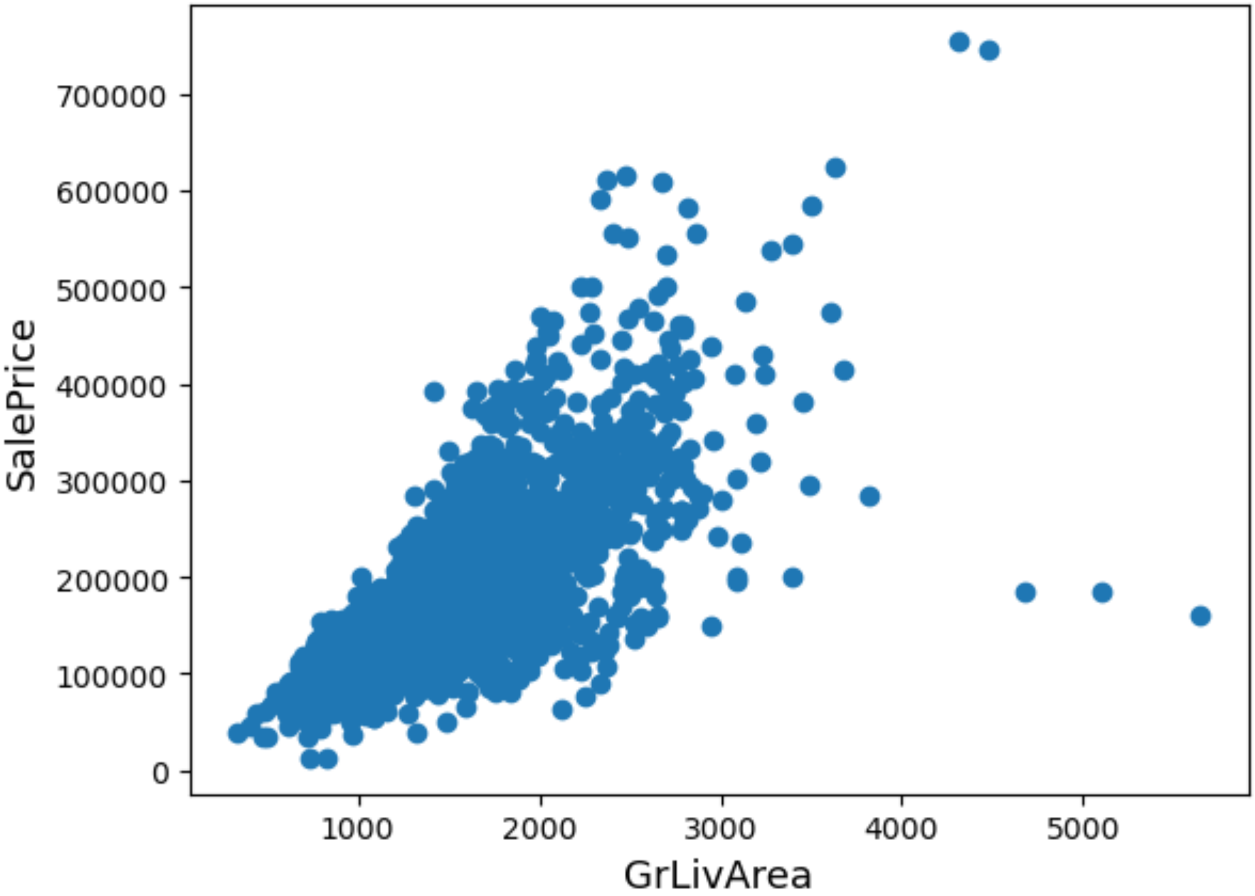
	Missing Ratio

Handling Outliers

Target Variable

In [21]:

```python
fig, ax = plt.subplots()
ax.scatter(x=housing_df["GrLivArea"], y=housing_df["SalePrice"])
plt.ylabel("SalePrice", fontsize=13)
plt.xlabel("GrLivArea", fontsize=13)
plt.show()
```

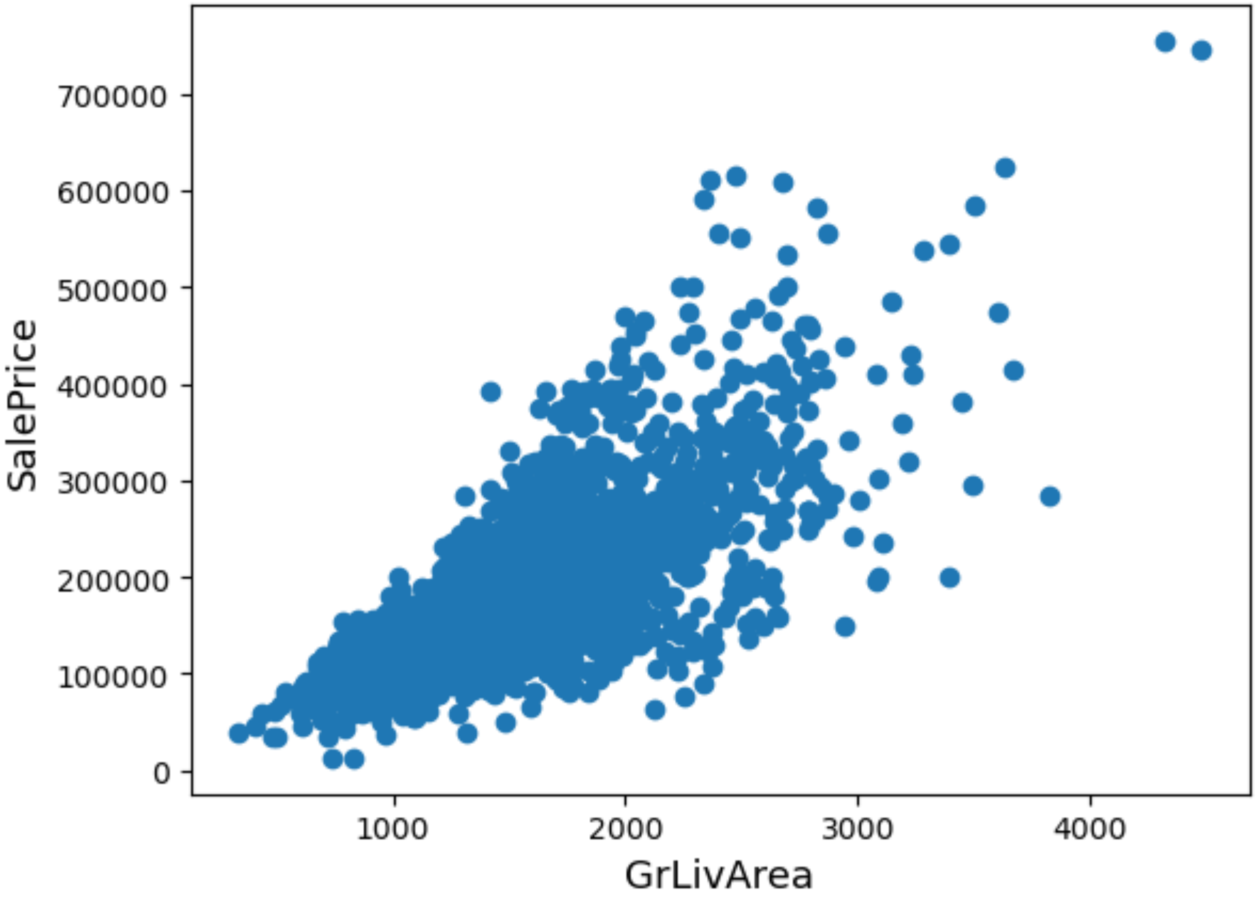


Looks like there is a few large outliers (living area is very large is sale price is not.)

For training we remove these outliers as it may skew our performance in the validation set. This will have the most impact on the linear model performance

In [22]:

```
# Deleting outliers
housing_df = housing_df.drop(
    housing_df[
        (housing_df["GrLivArea"] > 4000) & (housing_df["SalePrice"] < 300000
    ].index
)

# Check the graphic again
fig, ax = plt.subplots()
ax.scatter(housing_df["GrLivArea"], housing_df["SalePrice"])
plt.ylabel("SalePrice", fontsize=13)
plt.xlabel("GrLivArea", fontsize=13)
plt.show()
```

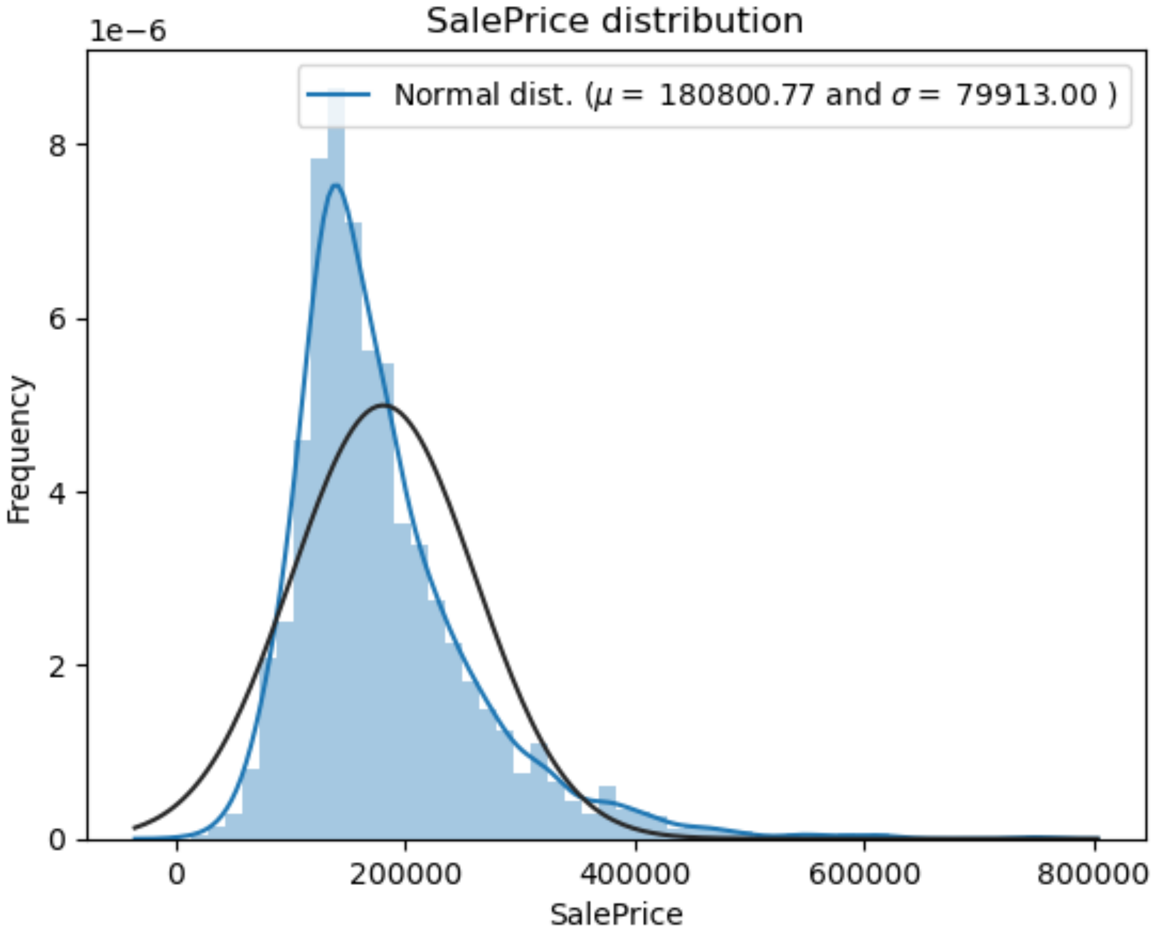


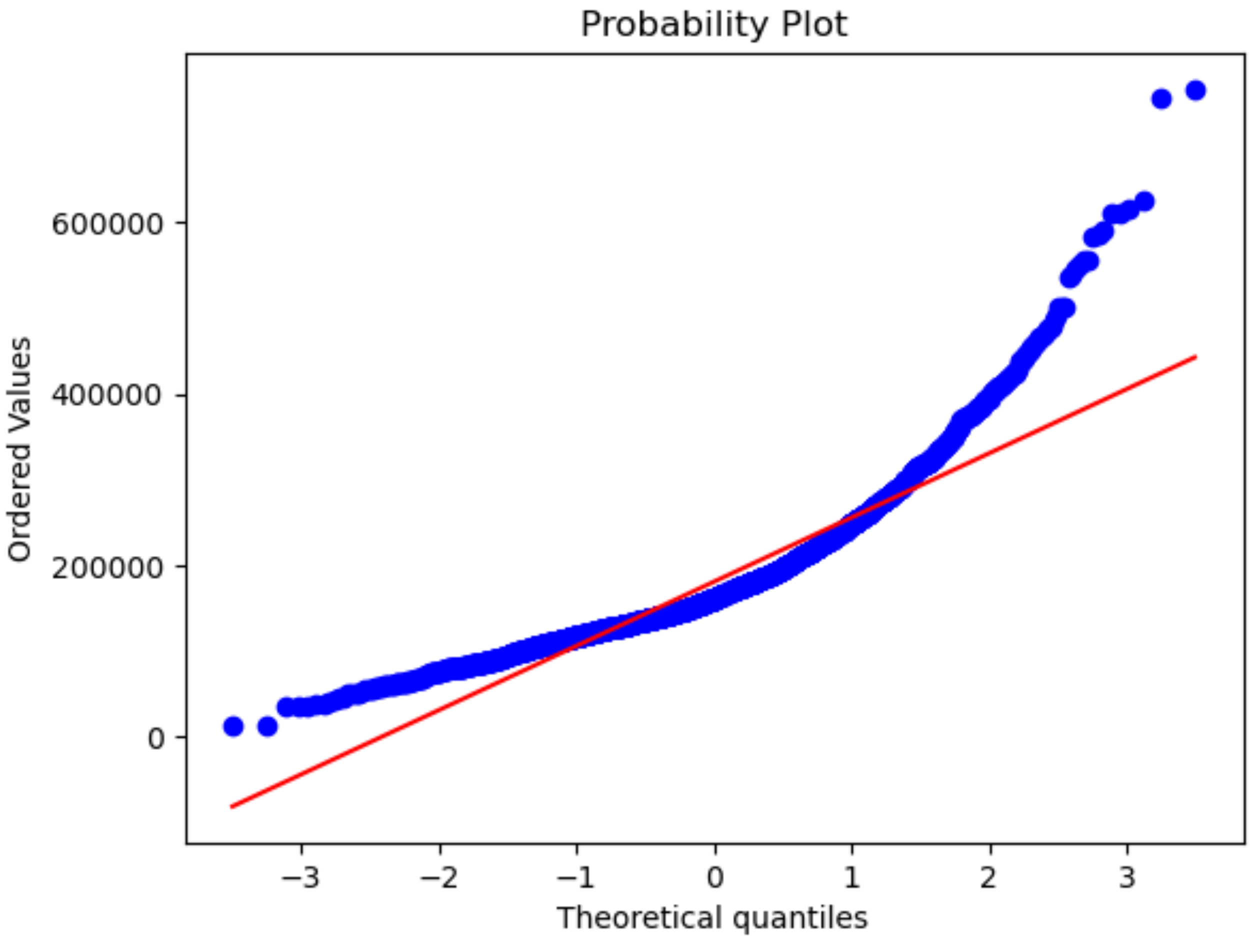
Normalize Target Variable

Normalizing the target variable is important for linear model performance. It does not have an impact for tree models, thus it is best practice to do so for preprocessing.

In [23]:
```
sns.distplot(housing_df["SalePrice"], fit=norm)

# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(housing_df["SalePrice"])
print("\n mu = {:.2f} and sigma = {:.2f}\n".format(mu, sigma))

# Now plot the distribution
plt.legend(
    ["Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )".format(mu, sigma)]
)
plt.ylabel("Frequency")
plt.title("SalePrice distribution")

# Get also the QQ-plot
fig = plt.figure()
res = probplot(housing_df["SalePrice"], plot=plt)
plt.show()
```

```
mu = 180800.77 and sigma = 79913.00
```

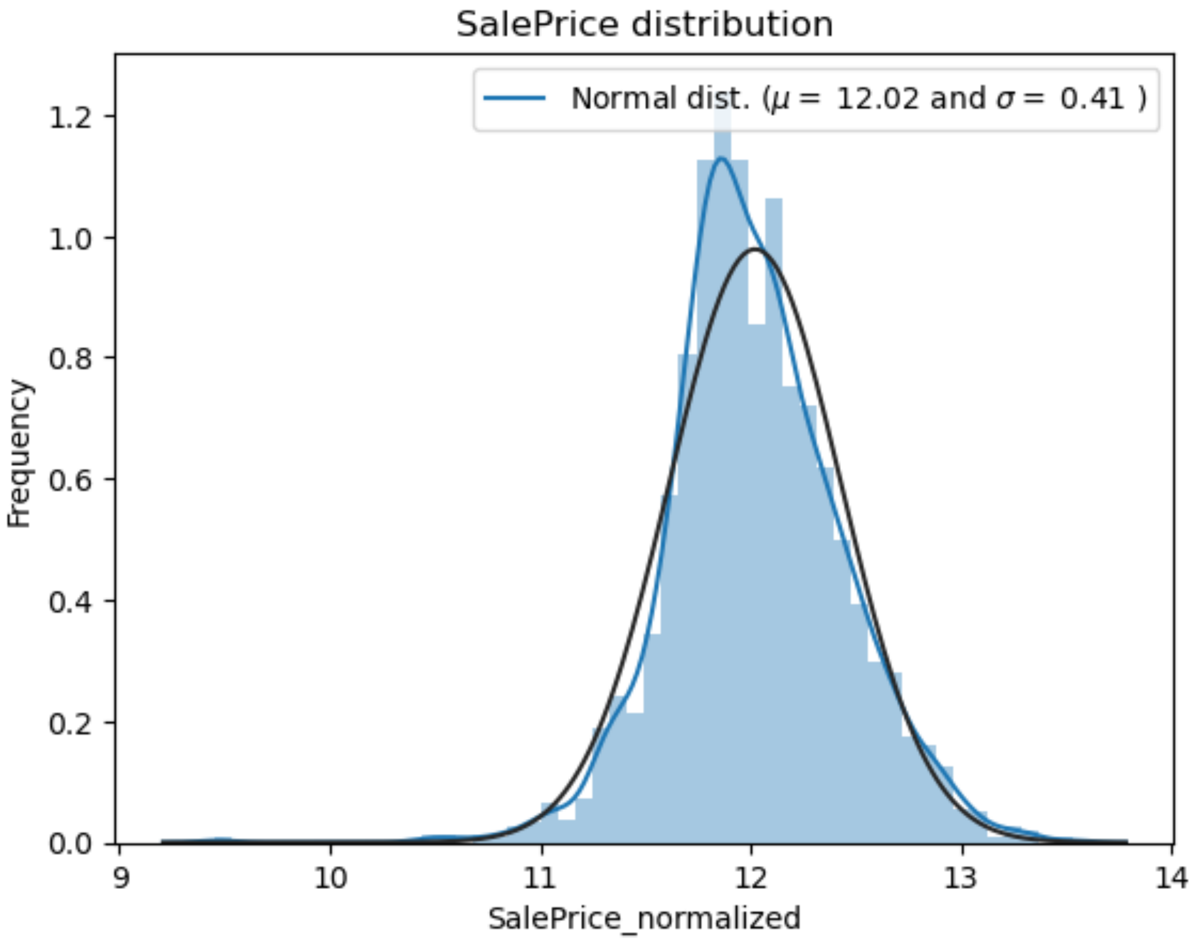


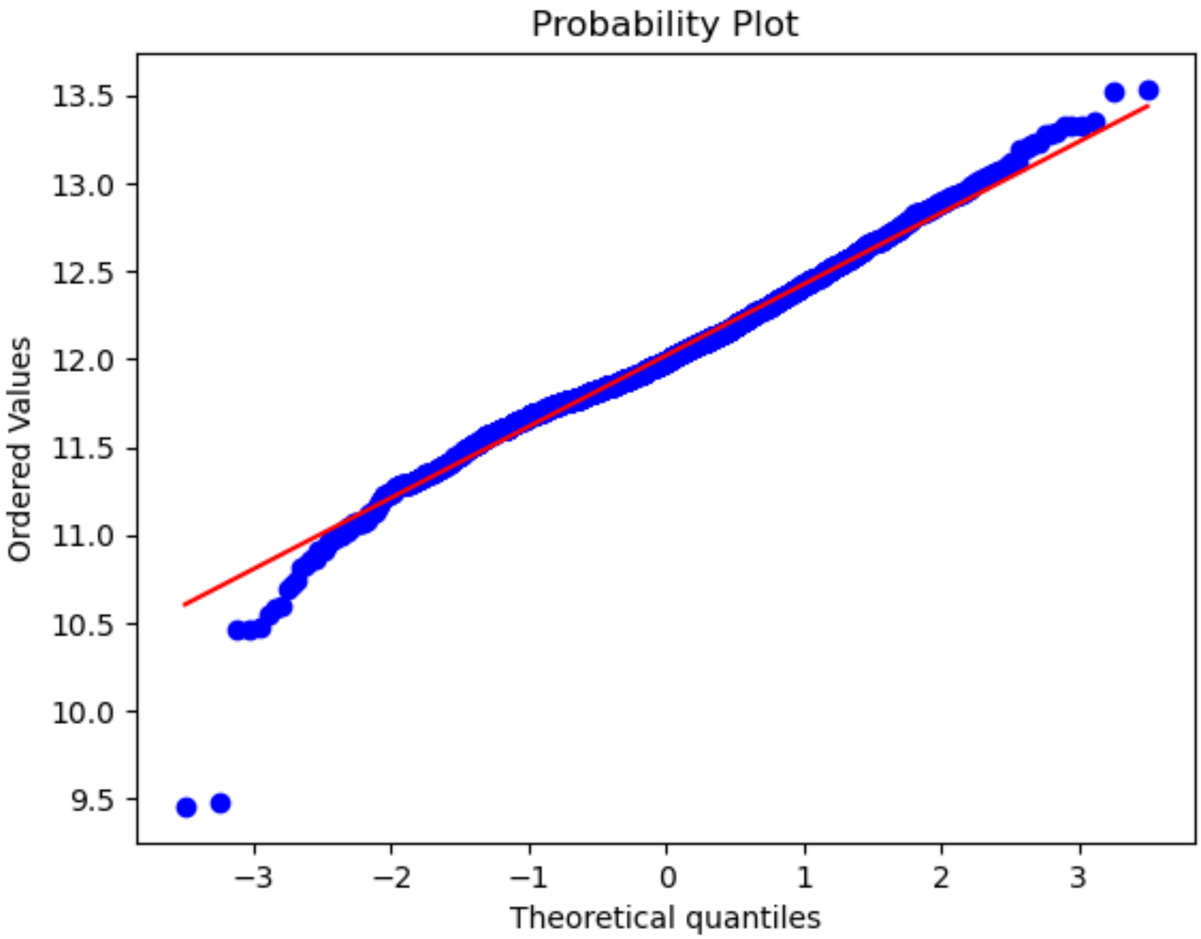


In [24]:

```
# We use the numpy fuction log1p which  applies log(1+x) to all elements of
housing_df["SalePrice_normalized"] = np.log1p(housing_df["SalePrice"])

# Check the new distribution
sns.distplot(housing_df["SalePrice_normalized"], fit=norm)

# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(housing_df["SalePrice_normalized"])
print("\n mu = {:.2f} and sigma = {:.2f}\n".format(mu, sigma))

# Now plot the distribution
plt.legend(
    ["Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )".format(mu, sigma)]
)
plt.ylabel("Frequency")
plt.title("SalePrice distribution")

# Get also the QQ-plot
fig = plt.figure()
res = probplot(housing_df["SalePrice_normalized"], plot=plt)
plt.show()
```

```
mu = 12.02 and sigma = 0.41
```





Data Analysis and Visualization

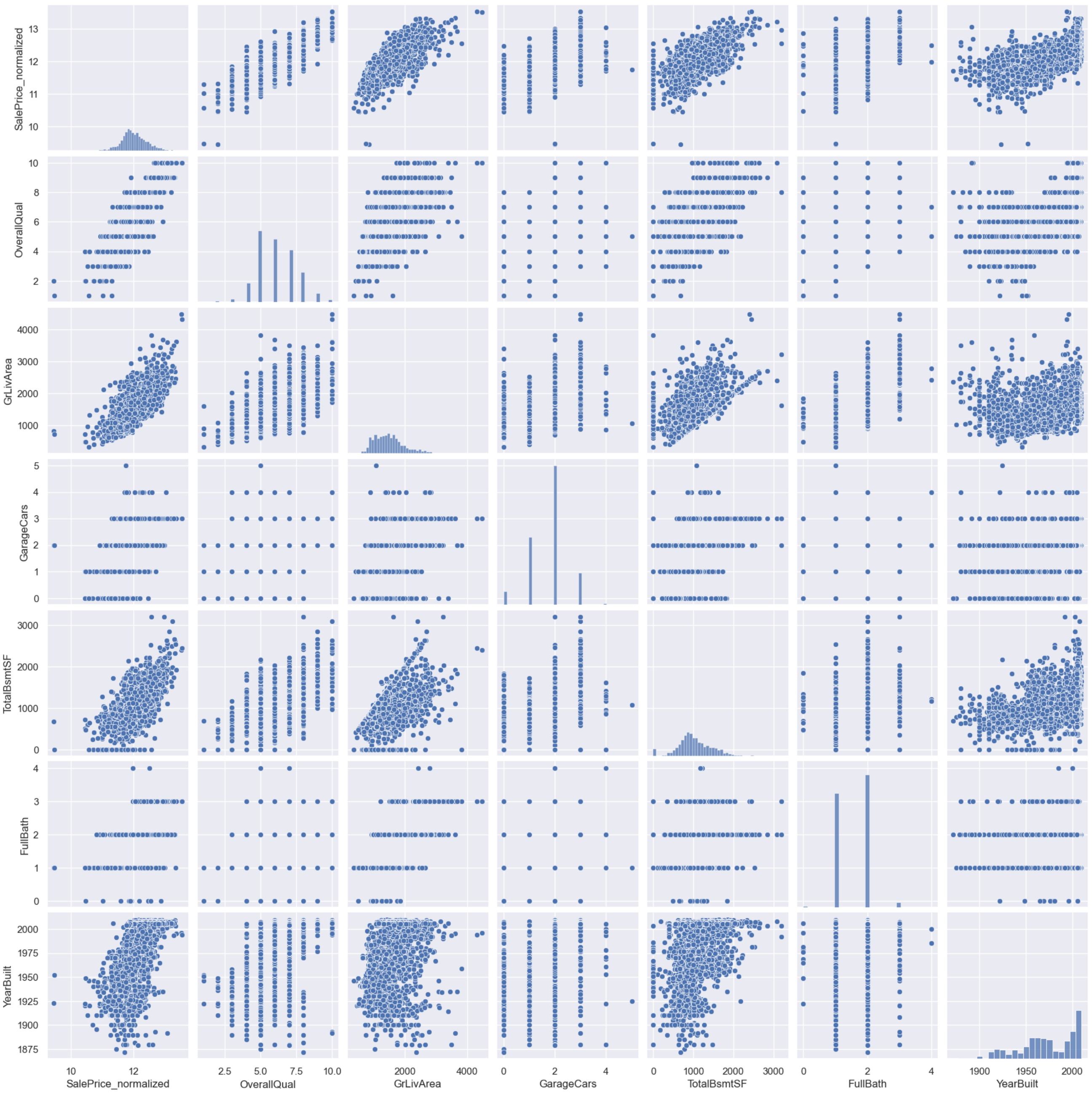
In [25]:

```
from sklearn.preprocessing import LabelEncoder
```

Target Variable Scatterplots

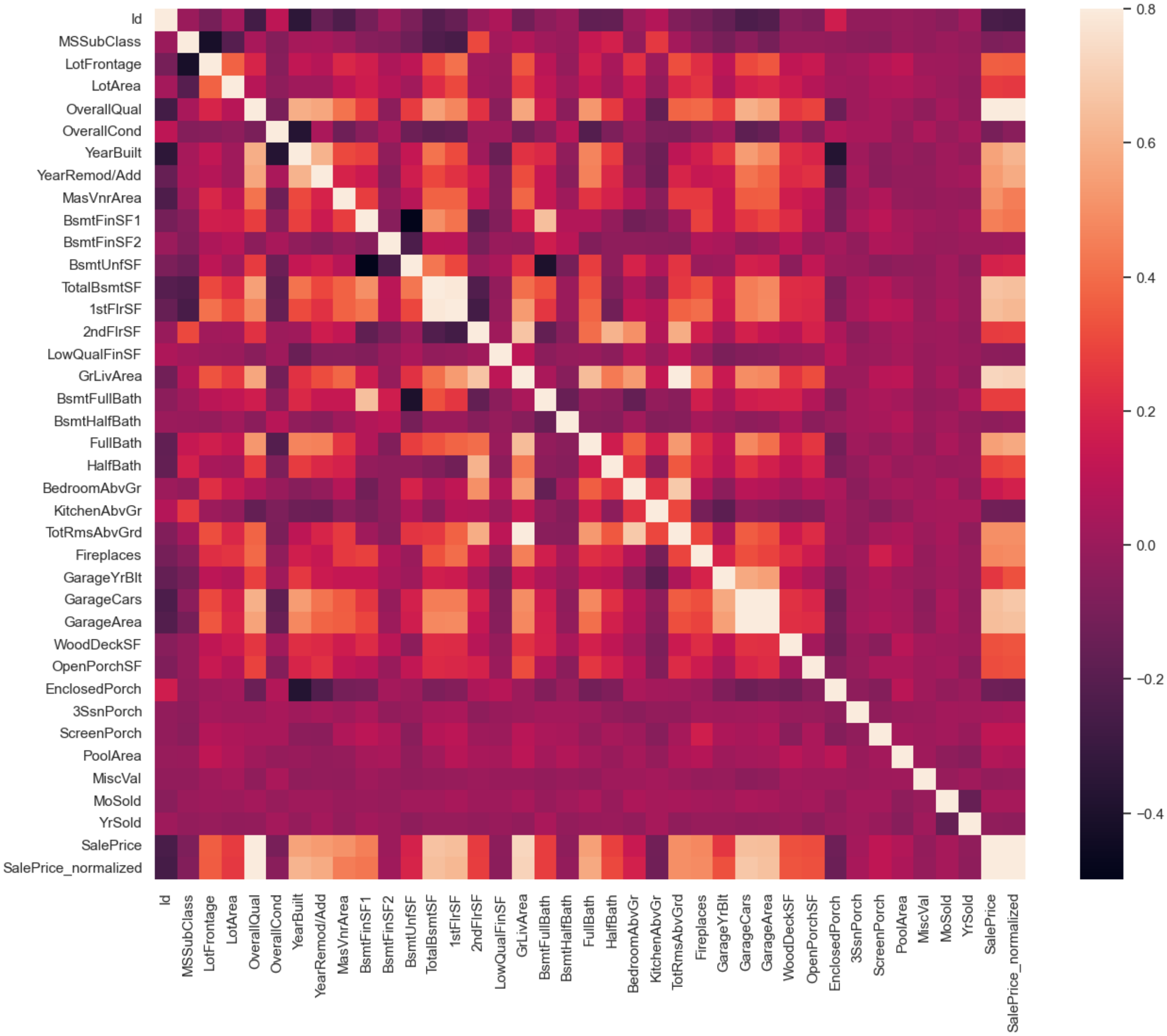
In [26]:

```
# scatterplot
sns.set()
cols = [
    "SalePrice_normalized",
    "OverallQual",
    "GrLivArea",
    "GarageCars",
    "TotalBsmtSF",
    "FullBath",
    "YearBuilt",
]
sns.pairplot(housing_df[cols], size=2.5)
plt.show();
```



Correlation Matrix

In [27]:
```
# Correlation map to see how features are correlated with SalePrice
corrmat = housing_df.corr()
f, ax = plt.subplots(figsize=(15, 12))
sns.heatmap(corrmat, vmax=0.8, square=True);
```



Let's visualize the data geo-spatially

Let's visualize the median sale price by zip code on a map

In [28]:
```
# Zip code is not in the data, so we must first map the neighborhoods to zip
# This was done manually by looking up each of the neighborhoods in the data
# Ames, Iowa neighborhood to zip code mapping

neighborhood_to_zip = {
    "NAmes": "50010",
    "CollgCr": "50014",
    "OldTown": "50010",
    "Edwards": "50014",
    "Somerst": "50010",
    "NridgHt": "50014",
    "Gilbert": "50105",
    "Sawyer": "50014",
    "NWAmes": "50010",
    "SawyerW": "50014",
    "Mitchel": "50010",
    "BrkSide": "50010",
    "Crawfor": "50014",
    "IDOTRR": None,
    "Timber": "50014",
    "NoRidge": "50014",
    "StoneBr": "50010",
    "SWISU": "50011",
    "ClearCr": "50010",
    "MeadowV": "50010",
    "BrDale": "50010",
    "Blmngtn": "50010",
    "Veenker": "50011",
    "NPkVill": "50010",
    "Blueste": "50014",
    "Greens": "50010",
    "GrnHill": "50010",
    "Landmrk": None,
}

# zip to lat lon dataset https://www.kaggle.com/datasets/joeleichter/us-zip-
housing_df["zip_code"] = housing_df["Neighborhood"].map(neighborhood_to_zip)
zip_to_lat_lon = pd.read_csv("house_prices/zip_lat_long.csv", dtype=str)

housing_df = pd.merge(
    housing_df, zip_to_lat_lon, left_on="zip_code", right_on="ZIP", how="lef
)
housing_df = housing_df.drop(["ZIP"], axis=1)

# Let's rename LAT and LNG to be more accurate field names
# Also let's encode LAT and LNG as numerics
housing_df.rename(
    columns={"LAT": "ZIP_centroid_LAT", "LNG": "ZIP_centroid_LON"}, inplace=
)

housing_df["ZIP_centroid_LAT"] = housing_df["ZIP_centroid_LAT"].astype(float
housing_df["ZIP_centroid_LON"] = housing_df["ZIP_centroid_LON"].astype(float

# compute the median sale price by zip code
avg_price_by_zip = (
    housing_df.groupby(["zip_code", "ZIP_centroid_LAT", "ZIP_centroid_LON"])
        "SalePrice"
    ]
    .median()
    .reset_index()
)
```

In [29]:

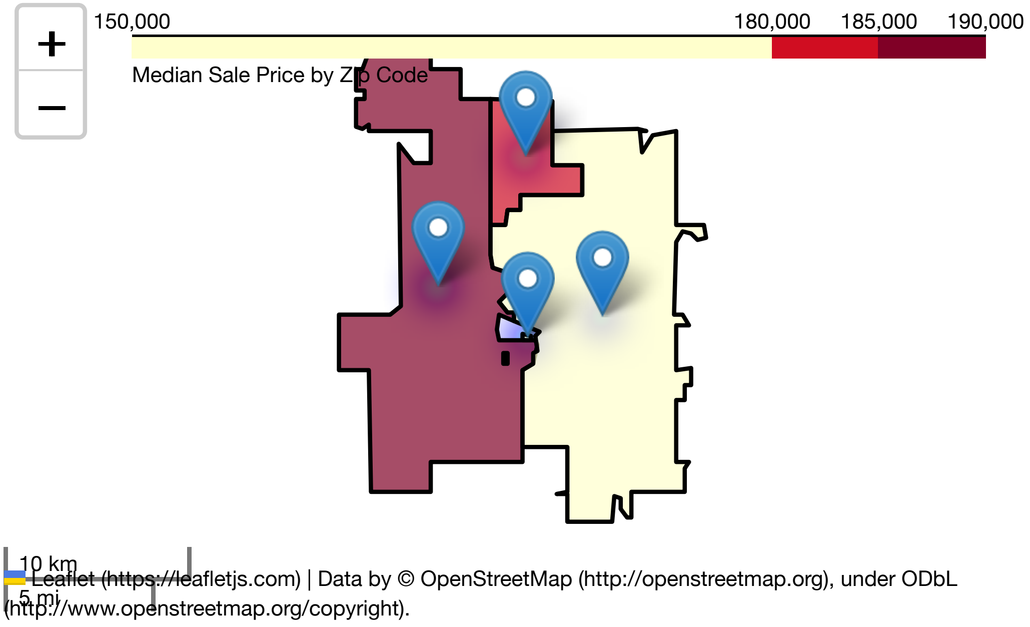
```python
import folium
import geopandas as gpd
from folium.plugins import HeatMap
from folium import Marker
from branca.element import Figure
import branca

# Download shape file from census
# https://www.census.gov/geographies/mapping-files/2013/geo/carto-boundary-f
zip_shapefile_path = "cb_2013_us_zcta510_500k/cb_2013_us_zcta510_500k.shp"

# Load the shapefile
zip_geo_df = gpd.read_file(zip_shapefile_path)

ames_iowa_zips = list(avg_price_by_zip["zip_code"].unique())
ames_iowa_zips_df = zip_geo_df[zip_geo_df["ZCTA5CE10"].isin(ames_iowa_zips)]

zip_geo_df = ames_iowa_zips_df.merge(
    avg_price_by_zip, how="left", left_on="ZCTA5CE10", right_on="zip_code"
)
```

In [30]:

```python
# Create a GeoDataFrame with centroids
centroids_df = gpd.GeoDataFrame(zip_geo_df[["geometry", "SalePrice"]])
centroids_df["geometry"] = centroids_df["geometry"].centroid
centroids_df["latitude"] = centroids_df["geometry"].y
centroids_df["longitude"] = centroids_df["geometry"].x

# Center the map around the median latitude and longitude of your data
center_lat, center_lon = (
    zip_geo_df.geometry.centroid.y.median(),
    zip_geo_df.geometry.centroid.x.median(),
)

fig = Figure(width=500, height=300)

# Create a folium map
ames_iowa_map = folium.Map(
    location=[center_lat, center_lon], zoom_start=10, control_scale=True
)
fig.add_child(ames_iowa_map)

# specify the min and max values of your data
colormap = branca.colormap.linear.YlOrRd_09.scale(150000, 190000)
colormap = colormap.to_step(index=[150000, 180000, 185000, 190000])
colormap.caption = "Median Sale Price by Zip Code"
colormap.add_to(ames_iowa_map)

# Create a GeoJson layer with your GeoDataFrame and add it to the map
geojson_data = zip_geo_df.to_json()
```

```
folium.GeoJson(
    geojson_data,
    name="choropleth",
    style_function=lambda feature: {
        "fillColor": colormap(feature["properties"]["SalePrice"]),
        "color": "black",
        "weight": 2,
        "fillOpacity": 0.7,
    },
).add_to(ames_iowa_map)

# Ensure that the size of latitude, longitude, and SalePrice arrays are the
heat_data = [
    [row["latitude"], row["longitude"], row["SalePrice"]]
    for idx, row in centroids_df.iterrows()
]

# Create a HeatMap using the original coordinates from the GeoDataFrame
HeatMap(heat_data, radius=10, blur=15).add_to(ames_iowa_map)

# Add markers with SalePrice as pop-up information
for idx, row in centroids_df.iterrows():
    Marker(
        [row["latitude"], row["longitude"]], popup=f'SalePrice: {row["SalePr
    ).add_to(ames_iowa_map)

# Display the map
ames_iowa_map
```

Out[30]:


+
−
150,000
180,000
185,000
190,000
Median Sale Price by Zip Code
10 km
5 mi
Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

Let's do a little bit of feature engineering to help transform our data.

In [31]:

```
# Some numeric fields are best converted to categorical variables to encode
# This may provide the most value as it increases the complexity; be wary of

# MSSubClass=The building class
housing_df["MSSubClass"] = housing_df["MSSubClass"].apply(str)


# Changing OverallCond into a categorical variable
housing_df["OverallCond"] = housing_df["OverallCond"].astype(str)

# Year and month sold are transformed into integer features
housing_df["YrSold"] = housing_df["YrSold"].astype(int)
housing_df["MoSold"] = housing_df["MoSold"].astype(int)

# Adding a date_sold feature to allow us to train-test split our data via a
# Also allows us to do some temporal analysis
housing_df["date_sold"] = pd.to_datetime(
    housing_df["YrSold"].astype(str) + housing_df["MoSold"].astype(str), for
)

# Adding total sqfootage feature
housing_df["TotalSF"] = (
    housing_df["TotalBsmtSF"] + housing_df["1stFlrSF"] + housing_df["2ndFlrS
)
```

Time series visualizations

Let's do some time series visualizations

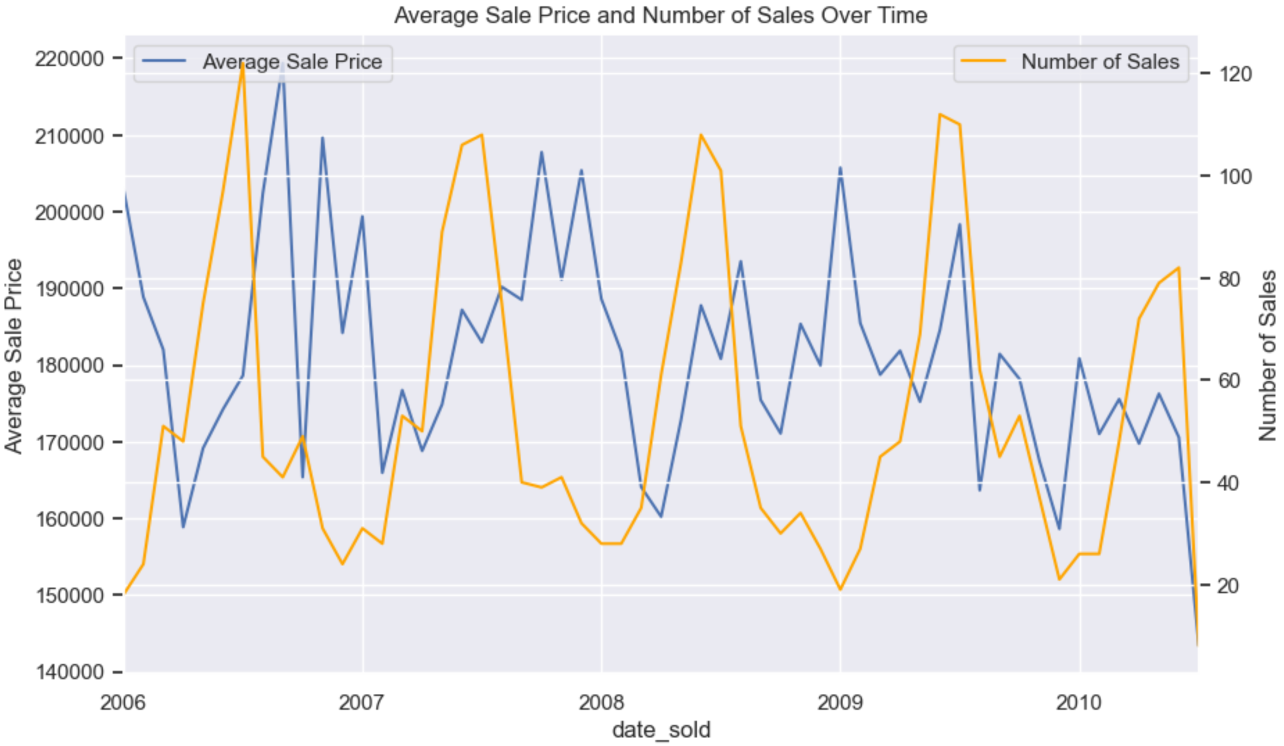
In [32]:

```
avg_sale_price_by_month = housing_df.groupby("date_sold")["SalePrice"].agg(
    ["mean", "count"]
)
avg_sale_price_by_month = avg_sale_price_by_month.rename(
    columns={"mean": "AverageSalePrice", "count": "NumberOfSales"}
)
```

In [33]:

```
# Set a larger figure size
fig, ax1 = plt.subplots(figsize=(10, 6))

ax1 = avg_sale_price_by_month["AverageSalePrice"].plot(label="Average Sale P
ax1.set_ylabel("Average Sale Price")

# Create a twin Axes for the second y-axis
ax2 = ax1.twinx()
avg_sale_price_by_month["NumberOfSales"].plot(
    ax=ax2, color="orange", label="Number of Sales"
)
ax2.set_ylabel("Number of Sales")
```

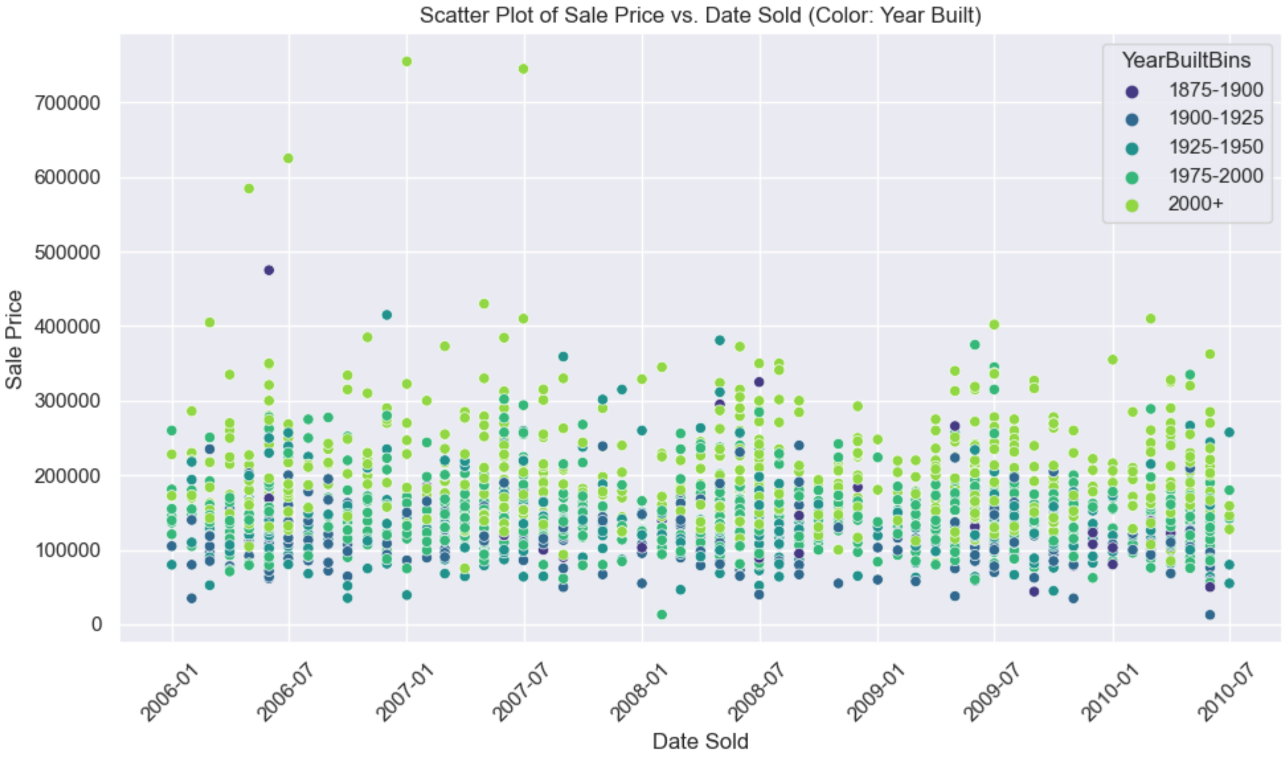
```
# Show legend
ax1.legend(loc="upper left")
ax2.legend(loc="upper right")

# Set title and x-axis label
plt.title("Average Sale Price and Number of Sales Over Time")
plt.xlabel("Sale Date")
plt.show()
```



In [34]:

```
# Scatter plot
plt.figure(figsize=(10, 6))

# Define custom bins for YrBuilt
bins = [1875, 1900, 1925, 1950, 1975, 2000]  # Adjust the bin edges as neede
housing_df["YearBuiltBins"] = pd.cut(
    housing_df["YearBuilt"],
    bins=bins,
    labels=["1875-1900", "1900-1925", "1925-1950", "1975-2000", "2000+"],
)

# Scatter plot with custom bins for legend
plt.figure(figsize=(10, 6))
scatter_plot = sns.scatterplot(
    x="date_sold",
    y="SalePrice",
    hue="YearBuiltBins",
    data=housing_df,
    palette="viridis",
    legend="full",
    cmap="viridis",
)

# Beautify the plot
plt.title("Scatter Plot of Sale Price vs. Date Sold (Color: Year Built)")
plt.xlabel("Date Sold")
plt.ylabel("Sale Price")
plt.xticks(rotation=45)
plt.tight_layout()

# Show the plot
plt.show()
```

```
<Figure size 1000x600 with 0 Axes>
```



```
In [35]: housing_df = housing_df.drop(["YearBuiltBins"], axis=1)
```

Label encode categorical variables

```
In [36]: cols = (
             "FireplaceQu",
             "BsmtQual",
             "BsmtCond",
             "GarageQual",
             "GarageCond",
             "ExterQual",
             "ExterCond",
             "HeatingQC",
             "PoolQC",
             "KitchenQual",
             "BsmtFinType1",
             "BsmtFinType2",
             "Functional",
             "Fence",
             "BsmtExposure",
             "GarageFinish",
             "LandSlope",
             "LotShape",
             "PavedDrive",
             "Street",
             "Alley",
             "CentralAir",
             "MSSubClass",
             "OverallCond",
             "YrSold",
             "MoSold",
             "Exterior1st",
             "Exterior2nd",
         )

         # process columns, apply LabelEncoder to categorical features
         for c in cols:
             lbl = LabelEncoder()
             lbl.fit(list(housing_df[c].values))
             housing_df[c] = lbl.transform(list(housing_df[c].values))

         # shape
         print("Shape housing_df: {}".format(housing_df.shape))
```

```
Shape housing_df: (2927, 86)
```

In [37]:

```
numeric_feats = housing_df.dtypes[
    (housing_df.dtypes != "object") & (housing_df.dtypes != "datetime64[ns]"
].index

# Check the skew of all numerical features
skewed_feats = (
    housing_df[numeric_feats]
    .apply(lambda x: skew(x.dropna()))
    .sort_values(ascending=False)
)
print("\nSkew in numerical features: \n")
skewness = pd.DataFrame({"Skew": skewed_feats})
skewness.head(20)
```

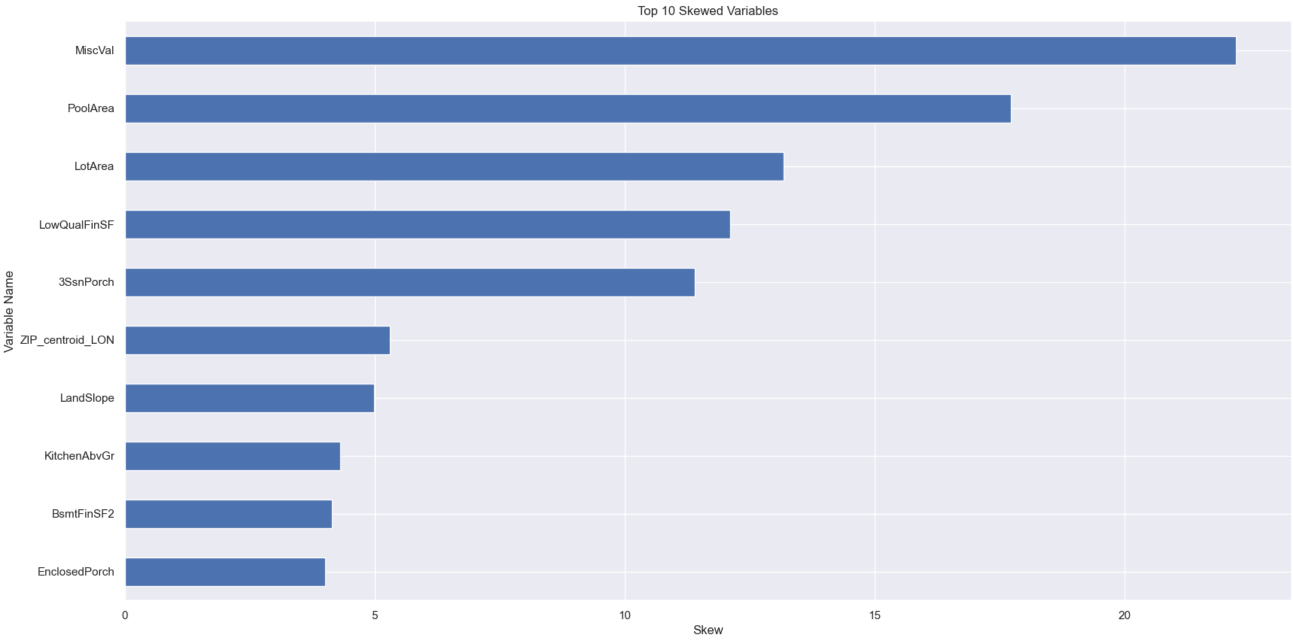
```
Skew in numerical features:
```

Out[37]:

	Skew
MiscVal	22.221198
PoolArea	17.719247
LotArea	13.178732
LowQualFinSF	12.105635
3SsnPorch	11.391990
ZIP_centroid_LON	5.307604
LandSlope	4.982762
KitchenAbvGr	4.309065
BsmtFinSF2	4.136226
EnclosedPorch	4.010074
ScreenPorch	3.953057
BsmtHalfBath	3.952498
MasVnrArea	2.598862
OpenPorchSF	2.493974
WoodDeckSF	1.843854
SalePrice	1.741608
LotFrontage	1.098549
GrLivArea	0.976093
1stFlrSF	0.950714
BsmtUnfSF	0.924088

In [38]:

```
skewness["Skew"].head(10).plot(
    kind="barh", figsize=(20, 10)
).invert_yaxis()  # top 10 missing columns
plt.xlabel("Skew")
plt.ylabel("Variable Name")
plt.title("Top 10 Skewed Variables")
plt.show()
```



In [39]:
```
# Highly skewed features are difficult for linear models to use; thus we tra
# This doesn't have a deleterious impact on other model's performance, thus 
skewness = skewness[abs(skewness) > 0.75]
print(
    "There are {} skewed numerical features to Box Cox transform (normalize)
        skewness.shape[0]
    )
)

skewed_features = skewness.index
lam = 0.15
for feat in skewed_features:
    # all_data[feat] += 1
    # any fields with negative values will return NULL values when boxcox tr
    # to avoid this let's skip the column ZIP_centroid_LON
    if feat not in [
        "Id",
        "SalePrice",
        "SalePrice_normalized",
        "date_sold",
        "ZIP_centroid_LON",
    ]:
        housing_df[feat] = boxcox1p(housing_df[feat], lam)

# all_data[skewed_features] = np.log1p(all_data[skewed_features])
```

```
There are 66 skewed numerical features to Box Cox transform (normalize)
```

In [40]:
```
# check that the box cot did not add any NULL values
null_columns = housing_df.columns[housing_df.isnull().any()]
null_count = housing_df[null_columns].isnull().sum()
```

```
print("Column Name: NULL Count")
for i in range(0, len(null_columns)):
    print(f"{null_columns[i]}: {null_count[i]}")
```

```
Column Name: NULL Count
zip_code: 94
```

In [41]:
```
# although Neighborhood has 28 unique values, we want to 1-hot encode this v
# one-hot encoding Neighborhood will add more complexity to our model, but b
cat_columns = housing_df.dtypes[
    (housing_df.dtypes != "float64")
    & (housing_df.dtypes != "int64")
    & (housing_df.dtypes != "datetime64[ns]")
].index
for col in cat_columns:
    num_vals = len(housing_df[col].unique())
    col_type = housing_df[col].dtype
    print(f"{col} ({col_type}): # of unique vals: {num_vals}")
```

```
MSZoning (object): # of unique vals: 7
LandContour (object): # of unique vals: 4
LotConfig (object): # of unique vals: 5
Neighborhood (object): # of unique vals: 28
Condition1 (object): # of unique vals: 9
Condition2 (object): # of unique vals: 8
BldgType (object): # of unique vals: 5
HouseStyle (object): # of unique vals: 8
RoofStyle (object): # of unique vals: 6
RoofMatl (object): # of unique vals: 7
MasVnrType (object): # of unique vals: 5
Foundation (object): # of unique vals: 6
Heating (object): # of unique vals: 6
Electrical (object): # of unique vals: 5
GarageType (object): # of unique vals: 7
MiscFeature (object): # of unique vals: 5
SaleType (object): # of unique vals: 10
SaleCondition (object): # of unique vals: 6
zip_code (object): # of unique vals: 5
```

Notice the large increase in columns when we one-hot encode all these variables using the pd.get_dummies() method. We need to be careful not too increase complexity too much that we are not overfitting.

In [42]:
```
housing_df = pd.get_dummies(housing_df)
print(housing_df.shape)
housing_df.head()
```

```
(2927, 208)
```

Out[42]:

	Id	MSSubClass	LotFrontage	LotArea	Street	Alley	LotShape
0	526301100	2.055642	7.353462	24.898884	0.730463	0.730463	0.000000
1	526350040	2.055642	6.221214	20.479373	0.730463	0.730463	1.540963
2	526351010	2.055642	6.244956	21.327220	0.730463	0.730463	0.000000
3	526353030	2.055642	6.512196	20.314716	0.730463	0.730463	1.540963
4	527105010	2.885846	6.073289	21.196905	0.730463	0.730463	0.000000

Data Analytics

In [43]:

```
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
import xgboost as xgb
import lightgbm as lgb
from skopt import BayesSearchCV
```

Below are 3 critical functions for evaluating our model properly:

- hyperparameter_tune_bayesian
- time_series_split_regression
- print_rmse_and_dates

hyperparameter_tune_bayesian:

- Takes the train dataset and the algorithm type as an input and tunes the hyperparameters
- Uses a 5 fold cross validated Bayesian Search on the train set to find the best hyperparameters
- It only takes in gradient boosted machines as a algorithm type
- It returns a dict of the optimal hyperparameters found

time_series_split_regression:

- This is the main function where the training happens
- By default, it performs a 5 fold time series cross-validated train of a specified regressor algorithm
- The *tune_hyperparameters* parameter allows the hyperparameters for each fold to be tuned by calling the *hyperparameter_tune_bayesian* function
- Some of the key things this function returns are:
  - DataFrame containing the Id, actual value, predicted value, fold, and whether it was in the test or train set
  - List of RMSE scores for each split

compute_rmse_std:

- This prints the RMSE and dates of each fold for a specified model

In [44]:

```python
def hyperparameter_tune_bayesian(X_train, y_train, regressor):
    """
    Perform hyperparameter tuning for XGBoost or LightGBM using Bayesian sea

    Parameters:
    - X_train: pandas DataFrame
        Training features.
    - y_train: pandas Series
        Training target variable.
    - regressor_type: str
        Type of regressor to tune ('xgboost' or 'lightgbm').

    Returns:
    - best_params: dict
        Best hyperparameters found during tuning.
    """
    # Define the common parameter space for both XGBoost and LightGBM
    param_space_common = {
        "n_estimators": (100, 1200),
        "learning_rate": (0.01, 0.2, "log-uniform"),
        "max_depth": (3, 10),
    }

    regressor_type = regressor.lower()
    if regressor_type == "xgboost":
        regressor = xgb.XGBRegressor()
    elif regressor_type == "lightgbm":
        regressor = lgb.LGBMRegressor()
    else:
        raise ValueError("Unsupported regressor type. Choose 'xgboost' or 'l
    # Update the search space with common parameters
    param_space = param_space_common.copy()

    # Perform Bayesian search
    bayes_search = BayesSearchCV(
        estimator=regressor,
        search_spaces=param_space,
        scoring="neg_mean_squared_error",
        cv=5,
        n_jobs=-1,  # Set the number of parallel jobs
    )
    bayes_search.fit(X_train, np.log1p(y_train))

    # Get the best hyperparameters
    best_params = bayes_search.best_params_

    return best_params
```

In [45]:

```
def time_series_split_regression(
    data,
    regressor,
    date_column="date_sold",
    target_column="SalePrice",
    cols_to_ignore=["Id", "SalePrice_normalized"],
    n_splits=5,
    tune_hyperparameters=False,
):
    """
    Perform time series split on a pandas DataFrame based on a date column a
    train a regression model, calculating RMSE for each split.

    Parameters:
    - data: pandas DataFrame
    - regressor: scikit-learn regressor object
        The regression algorithm to use.
    - date_column: str, default="date_sold"
        The name of the date column in the DataFrame.
    - target_column: str, default="SalePrice"
        The name of the target column in the DataFrame.
    - n_splits: int, default=5
        Number of splits for TimeSeriesSplit.
    - tune_hyperparameters: bool, default=False

    Returns:
    - result_df: pandas DataFrame
        DataFrame containing the Id, actual value, predicted value, fold, ar
    - rmse_scores: list of floats
        List of RMSE scores for each split.
    - split_dates: list of tuples
        List of (min_date, max_date) tuples for each split.
    - num_records: list of tuples
        List of (train_size, test_size) tuples for each split.
    """

    # Sort the DataFrame based on the date column
    data = data.sort_values(by=date_column)

    # Initialize TimeSeriesSplit
    tscv = TimeSeriesSplit(n_splits=n_splits)

    rmse_scores = []
    split_dates = []
    num_records = []
    all_predictions = []

    # Perform the time series split and train regression model for each spli
    for fold, (train_index, test_index) in enumerate(tscv.split(data)):
        train_data, test_data = data.iloc[train_index], data.iloc[test_index

        cols_to_ignore = cols_to_ignore + [target_column, date_column]

        # Bayesian hyperparameter tuning for XGBoost
        if (
            isinstance(regressor, (xgb.XGBRegressor, lgb.LGBMRegressor))
            and tune_hyperparameters
        ):  # Add LGBMRegressor to the isinstance check
            # Determine the regressor_type based on the type of the regresso
            if isinstance(regressor, xgb.XGBRegressor):
                regressor_type = "XGBoost"
            elif isinstance(regressor, lgb.LGBMRegressor):
                regressor_type = "LightGBM"
            else:
                raise ValueError(
                    "Unsupported regressor type. Supported types: XGBRegress
                )

            X_train_hyper, y_train_hyper = (
                data.drop(cols_to_ignore, axis=1),
                data[target_column],
            )
            best_params = hyperparameter_tune_bayesian(
                X_train_hyper, y_train_hyper, regressor_type
            )  # Specify 'xgboost' or 'lightgm' as the regressor type
            print(
                f"Best hyperparameters for {regressor_type} Fold {fold}: {be
            )
            regressor.set_params(**best_params)  # Set the best hyperparamet

        X_train = train_data.drop(cols_to_ignore, axis=1)
        X_test = test_data.drop(cols_to_ignore, axis=1)
        y_train, y_test = train_data[target_column], test_data[target_colum

        # Record the minimum and maximum dates for each split
        min_date, max_date = test_data[date_column].min(), test_data[date_c
        split_dates.append((min_date, max_date))

        # Train regression model
        regressor.fit(
            X_train, np.log1p(y_train)
        )  # Apply log1p transformation to the target variable during traini

        # Make predictions
        y_pred_log = regressor.predict(X_test)
        y_pred_train_log = regressor.predict(X_train)

        # Inverse transform predictions to get back the original scale
        y_pred = np.expm1(y_pred_log)
        y_pred_train = np.expm1(y_pred_train_log)

        # Check for NaN or infinity values in y_pred or y_test
        if (
            np.isnan(y_pred).any()
            or np.isinf(y_pred).any()
            or np.isnan(y_test).any()
            or np.isinf(y_test).any()
        ):
            print(
                f"Warning: NaN or infinity values found in predictions or tr
            )
            y_pred[np.isnan(y_pred) | np.isinf(y_pred)] = 0
            # Optionally, you can also handle y_test in a similar way if nee
            # y_test[np.isnan(y_test) | np.isinf(y_test)] = 0

        # Calculate RMSE on the original scale
        rmse = np.sqrt(mean_squared_error(y_test, y_pred))
        rmse_scores.append((rmse, fold))

        # Record results for 'Id', 'Actual', 'Predicted', 'Fold', and 'Set'
        fold_predictions = list(
            zip(
                test_data["Id"],
                y_test,
                y_pred,
                [fold] * len(test_data),
                ["test"] * len(test_data),
            )
        )
        fold_predictions += list(
            zip(
                train_data["Id"],
                y_train,
                y_pred_train,
                [fold] * len(train_data),
                ["train"] * len(train_data),
            )
        )
        all_predictions.extend(fold_predictions)

        # Calculate the size of each train-test split
        num_records.append((len(train_data), len(test_data)))

    # Create a DataFrame from the results
    result_df = pd.DataFrame(
        all_predictions, columns=["Id", "Actual", "Predicted", "Fold", "Set"
    )

    return result_df, rmse_scores, split_dates, num_records


# Example usage:
# result_df, rmse_scores, split_dates, num_records = time_series_split_regre
```

In [46]:

```
def compute_rmse_std(tuple_list):
    first_elements = [t[0] for t in tuple_list]
    mean = np.mean(first_elements)
    std = np.std(first_elements)
    return mean, std

    return average


def print_rmse_and_dates(model_rmse, model_split_dates, num_records, model_r
    # Print RMSE scores and split dates for each split
    for i, (rmse, dates, records) in enumerate(
        zip(model_rmse, model_split_dates, num_records)
    ):
        min_date, max_date = dates
        num_train_records, num_test_records = records

        min_date = min_date.date()
        max_date = max_date.date()

        print(
            f"Split {i + 1}: Min Date: {min_date}, Max Date: {max_date}, RMS
        )

    rmse_std = compute_rmse_std(model_rmse)
    print(model_name, "RMSE score: {:.4f} ({:.4f})\n".format(rmse_std[0], rm
```

In [47]:

```
# Linear regression models
# 3 types here: with intercept, without intercept, and Elastic Net (both L1
lr_w_int = LinearRegression()
lr_no_int = LinearRegression(fit_intercept=False)
elastic_net = ElasticNet(
    alpha=0.01, l1_ratio=0.1
)  # Adjust alpha and l1_ratio as needed
```

In [48]:

```
neigh = KNeighborsRegressor(n_neighbors=10)
```

In [49]:

```
rf = RandomForestRegressor(n_estimators=500)
```

In [50]:

```
dt = DecisionTreeRegressor(max_depth=10)
```

In [51]:

```
model_xgb = xgb.XGBRegressor(max_depth=5, n_estimators=1000, learning_rate=0
```

In [52]:

```
model_lgb = lgb.LGBMRegressor(max_depth=5, n_estimators=1000, learning_rate=
```

Algotithm Results on a 5 Fold Cross Validation

The method has many default parameters, so we do not need to feed in parameters for values such as the number of folds.

In [53]:

```
(
    lr_w_int_preds_df,
    lr_w_int_rmse,
    lr_w_int_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=lr_w_int,
)

# Print RMSE scores and split dates for each split
print_rmse_and_dates(
    lr_w_int_rmse, lr_w_int_split_dates, num_records, "Linear Regression (w/
)
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 86074.4537479546,
Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 24514.89355830879
5, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 21754.6200949873
1, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 20174.60343677393
4, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 19977.93420873059
2, Train Records: 2440, Test Records: 487
Linear Regression (w/ Intercept) RMSE score: 34499.3010 (25838.6483)
```

Linear regression sometimes does not generalize well. Removing the intercept sometimes helps generalizes better.

In [54]:

```
(
    lr_no_int_preds_df,
    lr_no_int_rmse,
    lr_no_int_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=lr_no_int,
)

print_rmse_and_dates(
    lr_no_int_rmse,
    lr_no_int_split_dates,
    num_records,
    "Linear Regression (No Intercept)",
)
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 87972.9141614172
7, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 33146.11685301016
4, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 30432.07555500267
3, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 20174.6034367738
4, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 19977.93420873014
5, Train Records: 2440, Test Records: 487
Linear Regression (No Intercept) RMSE score: 38340.7288 (25377.4898)
```

In [55]:

```
(
    elastic_net_preds_df,
    elastic_net_rmse,
    elastic_net_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=elastic_net,
)

print_rmse_and_dates(
    elastic_net_rmse,
    elastic_net_split_dates,
    num_records,
    "Elastic Net",
)
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 21698.74398562490
4, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 25738.29660309296
7, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 20775.9878832011
6, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 21492.84725612027
3, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 20425.14844766645
2, Train Records: 2440, Test Records: 487
Elastic Net RMSE score: 22026.2048 (1912.9950)
```

In [56]:

```
rf_preds_df, rf_rmse, rf_split_dates, num_records = time_series_split_regres
    housing_df,
    regressor=rf,
)
print_rmse_and_dates(rf_rmse, rf_split_dates, num_records, "Random Forest")
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 27837.5378923951
9, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 29334.68091752136
4, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 24261.33999779366
4, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 24005.0421410035
9, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 22054.15619761820
2, Train Records: 2440, Test Records: 487
Random Forest RMSE score: 25498.5514 (2676.2611)
```

In [57]:

```
nn_preds_df, nn_rmse, nn_split_dates, num_records = time_series_split_regres
    housing_df,
    regressor=neigh,
)
print_rmse_and_dates(nn_rmse, nn_split_dates, num_records, "Nearest Neighbor
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 57442.9665355711
7, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 55450.08524744958
4, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 45562.6545264609
9, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 43584.8615920475
3, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 39858.9960707192
4, Train Records: 2440, Test Records: 487
Nearest Neighbors RMSE score: 48379.9128 (6865.3006)
```

In [58]:

```
dt_preds_df, dt_rmse, dt_split_dates, num_records = time_series_split_regres
    housing_df,
    regressor=dt,
)
print_rmse_and_dates(dt_rmse, dt_split_dates, num_records, "Decision Tree")
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 43056.51419585573
6, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 40894.4938710668
8, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 39597.92435854953
5, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 37905.6750377820
8, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 33625.0115168552
5, Train Records: 2440, Test Records: 487
Decision Tree RMSE score: 39015.9238 (3177.6944)
```

In [59]:

```
xg_preds_df, xg_rmse, xg_split_dates, num_records = time_series_split_regres
    housing_df,
    regressor=model_xgb,
)
print_rmse_and_dates(xg_rmse, xg_split_dates, num_records, "XGBoost")
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 25484.15400425771
4, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 26820.4723244690
3, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 20313.23491011552
7, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 21357.82494042346
3, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 19621.7801777474
1, Train Records: 2440, Test Records: 487
XGBoost RMSE score: 22719.4933 (2887.9551)
```

In [60]:

```
lgbm_preds_df, lgbm_rmse, lgbm_split_dates, num_records = time_series_split_
    housing_df,
    regressor=model_lgb,
)
print_rmse_and_dates(lgbm_rmse, lgbm_split_dates, num_records, "LightGBM")
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 26884.60549685378
3, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 27307.8634544887
9, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 22139.8735512076
1, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 22265.4674809563
1, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 19204.36105108585
3, Train Records: 2440, Test Records: 487
LightGBM RMSE score: 23560.4342 (3090.7490)
```

Let's hyperparameter tune the XGBoost model using Bayesian Optimization.

This should take some time.

In [58]:

```
(
    xg_hyper_preds_df,
    xg_hyper_rmse,
    xg_hyper_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=model_xgb,
    tune_hyperparameters=True,
)
print_rmse_and_dates(xg_hyper_rmse, xg_hyper_split_dates, num_records, "XGBo
Best hyperparameters for XGBoost Fold 0: OrderedDict([('learning_rate', 0.05
8793964801230385), ('max_depth', 3), ('n_estimators', 368)])
Best hyperparameters for XGBoost Fold 1: OrderedDict([('learning_rate', 0.02
6772180941643877), ('max_depth', 6), ('n_estimators', 1200)])
Best hyperparameters for XGBoost Fold 2: OrderedDict([('learning_rate', 0.04
1966968181542184), ('max_depth', 3), ('n_estimators', 829)])
Best hyperparameters for XGBoost Fold 3: OrderedDict([('learning_rate', 0.09
888939749760209), ('max_depth', 3), ('n_estimators', 236)])
Best hyperparameters for XGBoost Fold 4: OrderedDict([('learning_rate', 0.01
557837106072331), ('max_depth', 3), ('n_estimators', 1200)])
Split 1: Min Date: 2006-09-01 00:00:00, Max Date: 2007-06-01 00:00:00, RMSE:
25277.228676957293, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01 00:00:00, Max Date: 2008-04-01 00:00:00, RMSE:
26085.315278833186, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01 00:00:00, Max Date: 2009-01-01 00:00:00, RMSE:
20066.531584068714, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01 00:00:00, Max Date: 2009-09-01 00:00:00, RMSE:
21545.589079345158, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01 00:00:00, Max Date: 2010-07-01 00:00:00, RMSE:
19131.47983052531, Train Records: 2440, Test Records: 487
XGBoost hyper RMSE score: 22421.2289 (2782.6627)
```

In [60]:

```
(
    lgb_hyper_preds_df,
    lgb_hyper_rmse,
    lgb_hyper_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=model_lgb,
    tune_hyperparameters=True,
)
print_rmse_and_dates(
    lgb_hyper_rmse, lgb_hyper_split_dates, num_records, "LightGBM hyper"
)
```

```
Best hyperparameters for LightGBM Fold 0: OrderedDict([('learning_rate', 0.0
37975934093380126), ('max_depth', 3), ('n_estimators', 868)])
Best hyperparameters for LightGBM Fold 1: OrderedDict([('learning_rate', 0.0
7018811248733267), ('max_depth', 3), ('n_estimators', 743)])
Best hyperparameters for LightGBM Fold 2: OrderedDict([('learning_rate', 0.0
3681371942374106), ('max_depth', 3), ('n_estimators', 1117)])
Best hyperparameters for LightGBM Fold 3: OrderedDict([('learning_rate', 0.0
704355350251849), ('max_depth', 3), ('n_estimators', 907)])
Best hyperparameters for LightGBM Fold 4: OrderedDict([('learning_rate', 0.0
7815414495100309), ('max_depth', 3), ('n_estimators', 589)])
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 25322.02945523245
2, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 27032.85285814005
7, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 21580.05180523900
3, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 21283.66370993852
6, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 18882.6055457090
8, Train Records: 2440, Test Records: 487
LightGBM hyper RMSE score: 22820.2407 (2946.4947)
```

In our final model we can only select 1 set of hyperparameters. Let's select the hyperparameters from the last fold as this has the most data to train with, and the lowest overall RMSE.

- Best hyperparameters for XGBoost Fold 4: OrderedDict([('learning_rate', 0.01557837106072331), ('max_depth', 3), ('n_estimators', 1200)])
- Best hyperparameters for LightGBM Fold 4: OrderedDict([('learning_rate', 0.06724242523308076), ('max_depth', 3), ('n_estimators', 781)])

In [61]:

```python
model_xgb_hyper = xgb.XGBRegressor(
    max_depth=3, n_estimators=1200, learning_rate=0.01557837106072331
)

model_lgb_hyper = lgb.LGBMRegressor(
    max_depth=3, n_estimators=781, learning_rate=0.06724242523308076
)
```

In [62]:

```python
(
    xg_hyper_select_preds_df,
    xg_hyper_select_rmse,
    xg_hyper_select_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=model_xgb_hyper,
)
print_rmse_and_dates(
    xg_hyper_select_rmse, xg_hyper_select_split_dates, num_records, "XGBoost
)
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 24261.58246255899
6, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 26325.71517891939
6, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 20260.6422988026
5, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 21114.14159597009
3, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 19131.4798305253
1, Train Records: 2440, Test Records: 487
XGBoost hyper RMSE score: 22218.7123 (2669.3037)
```

In [63]:

```
(
    lgb_hyper_select_preds_df,
    lgb_hyper_select_rmse,
    lgb_hyper_select_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=model_lgb_hyper,
)
print_rmse_and_dates(
    lgb_hyper_select_rmse, lgb_hyper_select_split_dates, num_records, "XGBoo
)
```

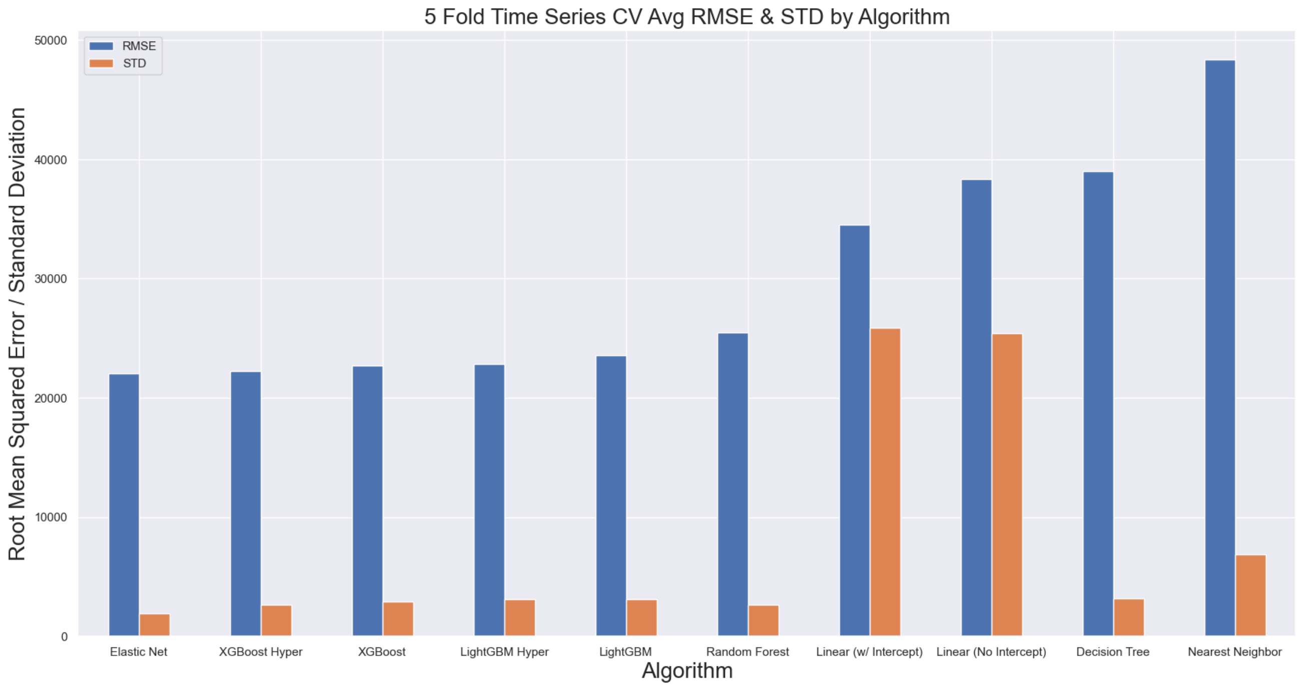
```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 25225.7690606001
3, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 27188.5712889930
3, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 22143.75504486790
7, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 21518.34865083397
4, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 18118.13654724195
7, Train Records: 2440, Test Records: 487
XGBoost hyper RMSE score: 22838.9161 (3134.6283)
```

In [64]:

```
# plot RMSE and STD for each Algorithm
data = {
    "Linear (No Intercept)": [
        compute_rmse_std(lr_no_int_rmse)[0],
        compute_rmse_std(lr_no_int_rmse)[1],
    ],
    "Linear (w/ Intercept)": [
        compute_rmse_std(lr_w_int_rmse)[0],
        compute_rmse_std(lr_w_int_rmse)[1],
    ],
    "Elastic Net": [
        compute_rmse_std(elastic_net_rmse)[0],
        compute_rmse_std(elastic_net_rmse)[1],
    ],
    "Nearest Neighbor": [compute_rmse_std(nn_rmse)[0], compute_rmse_std(nn_r
    "Decision Tree": [compute_rmse_std(dt_rmse)[0], compute_rmse_std(dt_rmse
    "Random Forest": [compute_rmse_std(rf_rmse)[0], compute_rmse_std(rf_rmse
    "XGBoost": [compute_rmse_std(xg_rmse)[0], compute_rmse_std(xg_rmse)[1]],
    "LightGBM": [compute_rmse_std(lgbm_rmse)[0], compute_rmse_std(lgbm_rmse)
    "XGBoost Hyper": [
        compute_rmse_std(xg_hyper_select_rmse)[0],
        compute_rmse_std(xg_hyper_select_rmse)[1],
    ],
    "LightGBM Hyper": [
        compute_rmse_std(lgb_hyper_select_rmse)[0],
        compute_rmse_std(lgb_hyper_select_rmse)[1],
    ],
}
data_df = pd.DataFrame(data=data).T.reset_index().sort_values(by=[0], ascend
data_df.columns = ["Algorithm", "RMSE", "STD"]
```

In [66]:

```
# creating the bar plot
data_df.plot(kind="bar", x="Algorithm", y=["RMSE", "STD"], figsize=(20, 10),
plt.xlabel("Algorithm", fontsize=20)
plt.ylabel("Root Mean Squared Error / Standard Deviation", fontsize=20)
plt.title("5 Fold Time Series CV Avg RMSE & STD by Algorithm", fontsize=20)
plt.savefig("5_fold_results.png", bbox_inches="tight")
plt.show()
```



Linear models had extremely high standard deviation from the rest; this is due to 2 reasons:

- fold 0 had a much higher RMSE compared to the other folds, leading to a much higher standard deviation. This is because, for a linear model, the train set size was too small to create a performant model.
- imputation of 0 for model w/ intercept for nan or inf values (depending on the data cleaning process this may not have occurred)

The decision tree and the nearest neighbor model appeared to perform significantly worse then the rest; we will be removing these from our analysis moving forward.

The train-test split size for the early splits are significantly small; to make sure we are evaluating the linear models correctly, we will be removing the first 2 folds (fold 0 and fold 1) from our analysis.

If the linear models did have 0 imputed for predictions, we would want to evaluate how these models performance imputing our best performing model (elastic net and/or xgboost) instead of 0. You could use the dataframes named *{model_name}_preds_df* to do this.

Side Note about Elastic Net:

We see that elastic net is our best performing model, however I want to emphasize how sensitive linear models (remember elastic net is a linear model with both L1 and L2 regularization) Let's evaluate how the model performs changing the following:

- *alpha*: 0.01 -> 0.1
- *l1_ratio*: 0.1 -> 0.5.

In [67]:

```
(
    elastic_net_l1_preds_df,
    elastic_net_l1_rmse,
    elastic_net_l1_split_dates,
    num_records,
) = time_series_split_regression(
    housing_df,
    regressor=ElasticNet(alpha=0.1, l1_ratio=0.5),
)

print_rmse_and_dates(
    elastic_net_l1_rmse,
    elastic_net_l1_split_dates,
    num_records,
    "Elastic Net (alpha=0.1, l1_ratio=0.5)",
)
```

```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 45187.6803405552
2, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 48860.8934554959
3, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 38305.76882757767
5, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 40295.0116332937
7, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 37716.3775804961
3, Train Records: 2440, Test Records: 487
Elastic Net (alpha=0.1, l1_ratio=0.5) RMSE score: 42073.1464 (4292.0895)
```

We see that now, it is one of our worst performing models. The main point I want to drive home here:

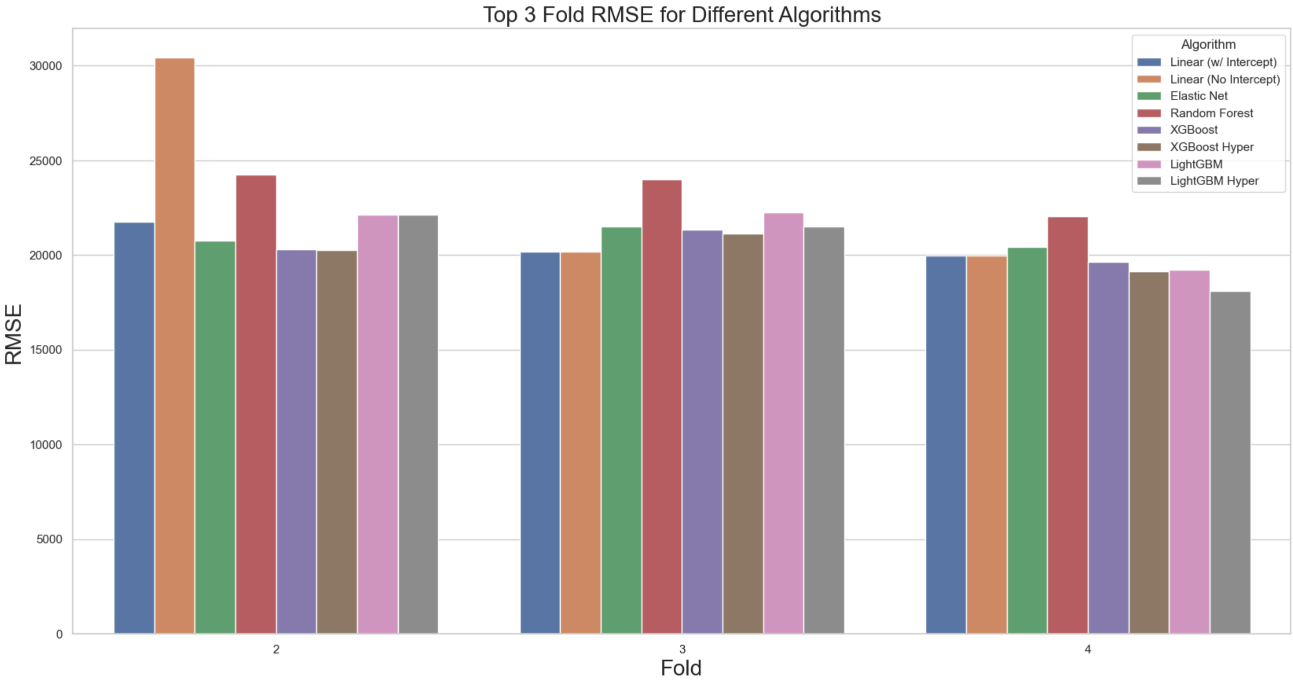
- Linear models often have a solution that is about as good as more powerful algorithms
- But, gradient boosted machines are usually best out of the box (no to little parameter tuning) for structured data problems

In [68]:

```
# Define the data dictionary
# remove a few models to make the next chart easier to digest
rmse_data = {
    "Linear (w/ Intercept)": lr_w_int_rmse,
    "Linear (No Intercept)": lr_no_int_rmse,
    "Elastic Net": elastic_net_rmse,
    "Random Forest": rf_rmse,
    "XGBoost": xg_rmse,
    "XGBoost Hyper": xg_hyper_select_rmse,
    "LightGBM": lgbm_rmse,
    "LightGBM Hyper": lgb_hyper_select_rmse,
    # "Linear (w/ Int & XG Impute)": linear_xg_impute_rmse,
    # "Decision Tree": dt_rmse,
    # "Nearest Neighbor": nn_rmse,
}

# Create DataFrames for each algorithm and add a 'Algorithm' column
dfs = {
    key: pd.DataFrame(
        values, columns=["RMSE", "Fold"], index=range(1, len(values) + 1)
    ).assign(Algorithm=key)
    for key, values in rmse_data.items()
}

# Concatenate all DataFrames into a single DataFrame
rmse_df = pd.concat(dfs.values(), ignore_index=True)

# Filter out folds 0 and 1 because train-test ratio is less than 70/30
# Also, the number of records is below 500 in these fold's train sets
rmse_df = rmse_df[rmse_df["Fold"].isin([2, 3, 4])]
```

In [70]:

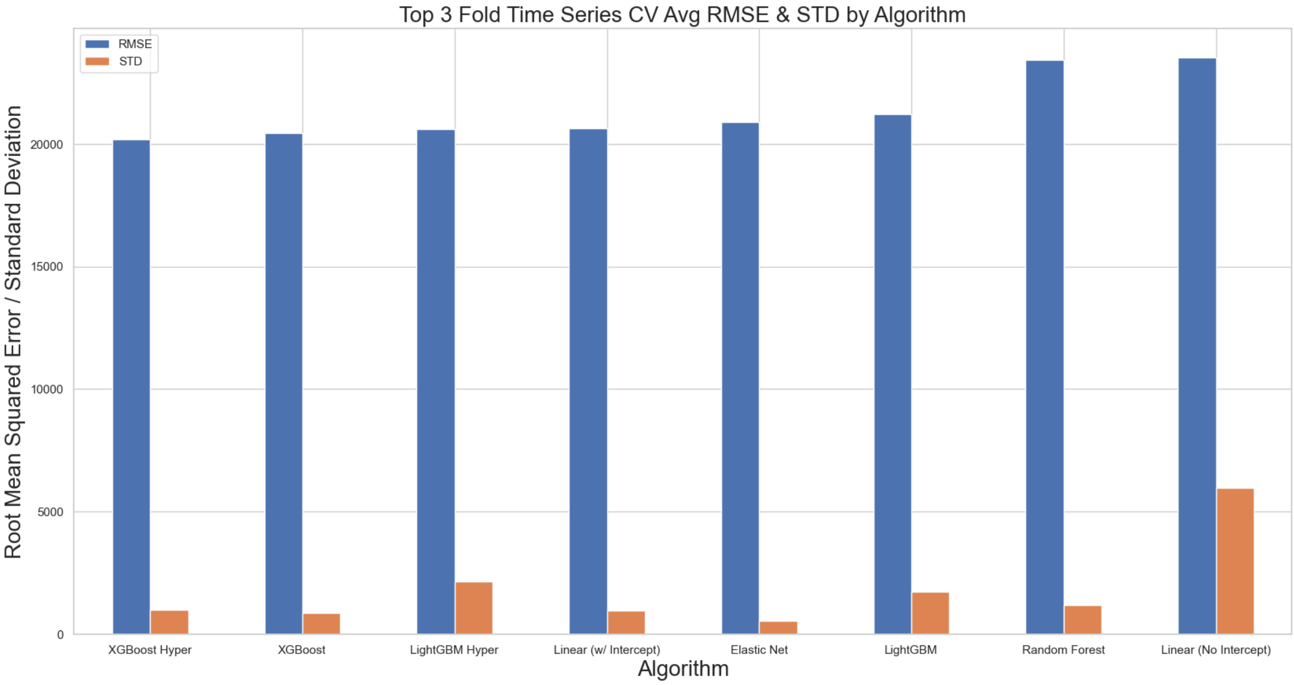
```
# Set the style of seaborn for better aesthetics (optional)
sns.set(style="whitegrid")

# Create a bar plot
plt.figure(figsize=(20, 10))
sns.barplot(x="Fold", y="RMSE", hue="Algorithm", data=rmse_df)
plt.title("Top 3 Fold RMSE for Different Algorithms", fontsize=20)
plt.xlabel("Fold", fontsize=20)
plt.ylabel("RMSE", fontsize=20)
plt.show()
```



We see a clear trend that as we train with more data, the RMSE in the holdout set decreases (i.e. the RMSE for out of fold set is the lowest in the final fold, 4, as it has the most data to train with)

In [71]:

```
upper_folds_rmse = (
    rmse_df.groupby("Algorithm")["RMSE"].agg(["mean", "std"]).reset_index()
)
upper_folds_rmse = upper_folds_rmse.rename(columns={"mean": "RMSE", "std": "
upper_folds_rmse = upper_folds_rmse.sort_values(by="RMSE", ascending=True)

# creating the bar plot
upper_folds_rmse.plot(
    kind="bar", x="Algorithm", y=["RMSE", "STD"], figsize=(20, 10), rot=0
)
plt.xlabel("Algorithm", fontsize=20)
plt.ylabel("Root Mean Squared Error / Standard Deviation", fontsize=20)
plt.title("Top 3 Fold Time Series CV Avg RMSE & STD by Algorithm", fontsize=
plt.show()
```



Moving forward we will be focusing on tuning the XGBoost, LightGBM (both hyperparameter tuned), linear with intercept, and elastic net models

Let's create a meta-learner model to try and maximize our model performance

Meta learner taken from: https://www.kaggle.com/code/serigne/stacked-regressions-top-4-on-leaderboard

Meta Model

In this approach, we add a meta-model on base models and use the out-of-folds predictions of these base models to train our meta-model.

The procedure, for the training part, may be described as follows:

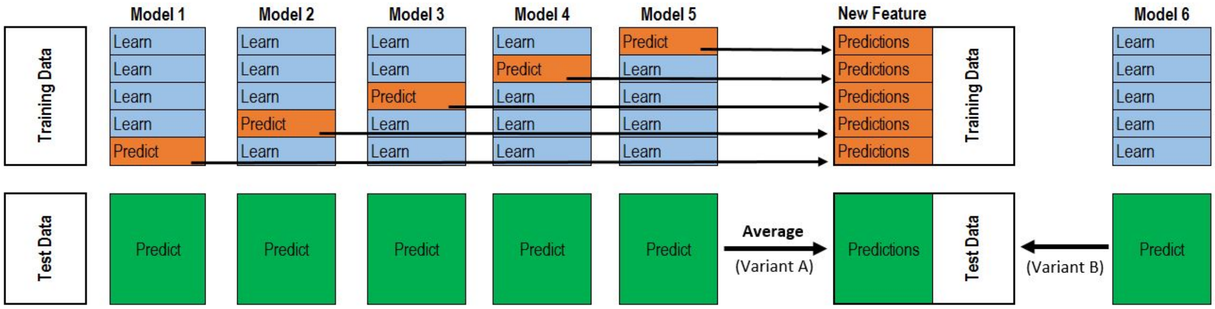
1. Split the total training set into two disjoint sets (train and test/holdout)
2. Train several base models on the first part (train)
3. Test these base models on the second part (holdout)
4. Use the predictions from 3 (called out-of-folds predictions) as the inputs, and the correct responses (target variable) as the outputs to train a higher level learner called meta-model.

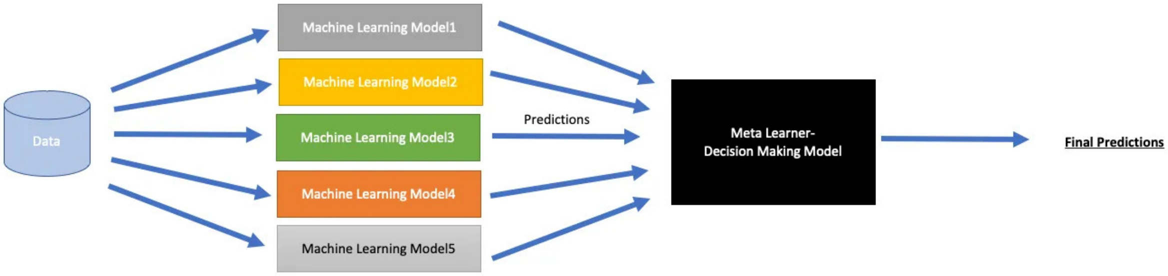
The first three steps are done iteratively. If we take for example a 5-fold stacking, we first split the training data into 5 folds. Then we will do 5 iterations. In each iteration, we train every base model on 4 folds and predict on the remaining fold (holdout fold).

So, we will be sure, after 5 iterations , that the entire data is used to get out-of-folds predictions that we will then use as new feature to train our meta-model in the step 4.

For the prediction part, We average the predictions of all base models on the test data and used them as meta-features on which, the final prediction is done with the meta-model.

For our metal models, it is best to use more complex models for the base learner, and a more simplistic model for the meta-model to avoid overfitting.





In [72]:
```
from sklearn.base import clone
from sklearn.model_selection import KFold


class StackedEnsembleCVRegressor:
    def __init__(self, base_models, meta_model, n_folds=5):
        self.base_models = base_models
        self.meta_model = meta_model
        self.n_folds = n_folds

    # We again fit the data on clones of the original models
    def fit(self, X, y):
        X, y = np.array(X), np.array(y)

        self.base_models_ = [list() for x in self.base_models]
        self.meta_model_ = clone(self.meta_model)
        kfold = KFold(n_splits=self.n_folds, shuffle=True, random_state=156)
        # Train cloned base models then create out-of-fold predictions
        # that are needed to train the cloned meta-model
        out_of_fold_predictions = np.zeros((X.shape[0], len(self.base_models
        for i, model in enumerate(self.base_models):
            for train_index, holdout_index in kfold.split(X, y):
                instance = clone(model)
                self.base_models_[i].append(instance)
                instance.fit(X[train_index], y[train_index])
                y_pred = instance.predict(X[holdout_index])
                out_of_fold_predictions[holdout_index, i] = y_pred

        # Now train the cloned  meta-model using the out-of-fold predictions
        self.meta_model_.fit(out_of_fold_predictions, y)
        return self

    # Do the predictions of all base models on the test data and use the ave
    # meta-features for the final prediction which is done by the meta-model
    def predict(self, X):
        meta_features = np.column_stack(
            [
                np.column_stack([model.predict(X) for model in base_models])
                    axis=1
                )
                for base_models in self.base_models_
            ]
        )
        return self.meta_model_.predict(meta_features)


# Example usage:
# base_models = [RandomForestRegressor(random_state=42), LinearRegression()]
# meta_model = RandomForestRegressor(random_state=42)
# stacked_averaged_models = StackedEnsembleCVRegressor(base_models, meta_mod
# score = rmsle_cv(stacked_averaged_models)
```

In [73]:

```
base_models = [model_lgb_hyper, model_xgb_hyper, lr_w_int]
meta_model = elastic_net
elastic_ensemble = StackedEnsembleCVRegressor(base_models, meta_model)
```

In [74]:

```
(
    elastic_ensemble_preds_df,
    elastic_ensemble_rmse,
    elastic_ensemble_split_dates,
    num_records,
) = time_series_split_regression(housing_df, regressor=elastic_ensemble)

print_rmse_and_dates(
    elastic_ensemble_rmse,
    elastic_ensemble_split_dates,
    num_records,
    "Elastic Ensemble Meta Model",
)
```

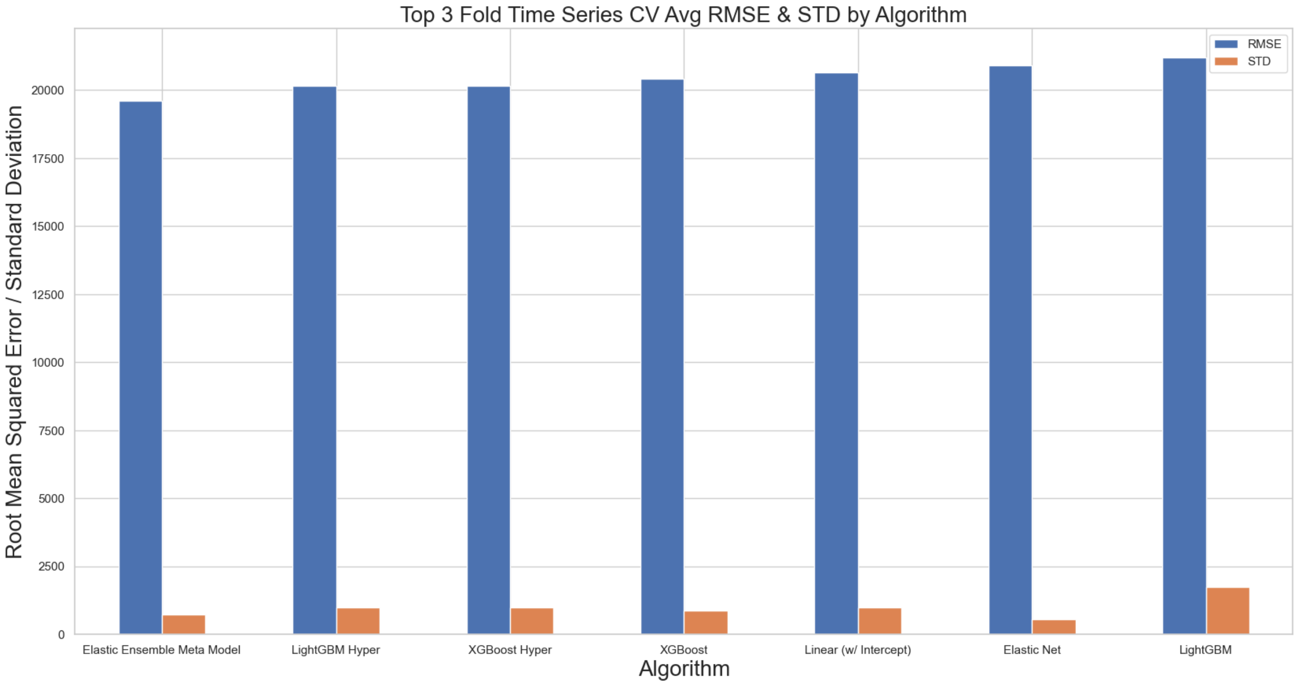
```
Split 1: Min Date: 2006-09-01, Max Date: 2007-06-01, RMSE: 25078.2950377734
9, Train Records: 492, Test Records: 487
Split 2: Min Date: 2007-06-01, Max Date: 2008-04-01, RMSE: 26921.82343303355
4, Train Records: 979, Test Records: 487
Split 3: Min Date: 2008-04-01, Max Date: 2009-01-01, RMSE: 19765.1795364598
1, Train Records: 1466, Test Records: 487
Split 4: Min Date: 2009-01-01, Max Date: 2009-09-01, RMSE: 20257.22787662862
5, Train Records: 1953, Test Records: 487
Split 5: Min Date: 2009-09-01, Max Date: 2010-07-01, RMSE: 18831.7445361225
3, Train Records: 2440, Test Records: 487
Elastic Ensemble Meta Model RMSE score: 22170.8541 (3213.2168)
```

In [75]:

```
# Define the data dictionary
# remove a few models to make the next chart easier to digest
rmse_data = {
    "Linear (w/ Intercept)": lr_w_int_rmse,
    # "Linear (No Intercept)": lr_no_int_rmse,
    "Elastic Net": elastic_net_rmse,
    # "Random Forest": rf_rmse,
    "XGBoost": xg_rmse,
    "XGBoost Hyper": xg_hyper_select_rmse,
    "LightGBM": lgbm_rmse,
    "LightGBM Hyper": xg_hyper_select_rmse,
    "Elastic Ensemble Meta Model": elastic_ensemble_rmse,
    # "Linear (w/ Int & XG Impute)": linear_xg_impute_rmse,
    # "Decision Tree": dt_rmse,
    # "Nearest Neighbor": nn_rmse,
}

# Create DataFrames for each algorithm and add a 'Algorithm' column
dfs = {
    key: pd.DataFrame(
        values, columns=["RMSE", "Fold"], index=range(1, len(values) + 1)
    ).assign(Algorithm=key)
    for key, values in rmse_data.items()
}

# Concatenate all DataFrames into a single DataFrame
rmse_df = pd.concat(dfs.values(), ignore_index=True)

# Filter out folds 0 and 1 because train-test ratio is less than 70/30
# Also, the number of records is below 500 in these fold's train sets
rmse_df = rmse_df[rmse_df["Fold"].isin([2, 3, 4])]
```

In [76]:

```
upper_folds_rmse = (
    rmse_df.groupby("Algorithm")["RMSE"].agg(["mean", "std"]).reset_index()
)
upper_folds_rmse = upper_folds_rmse.rename(columns={"mean": "RMSE", "std": "
upper_folds_rmse = upper_folds_rmse.sort_values(by="RMSE", ascending=True)

# creating the bar plot
upper_folds_rmse.plot(
    kind="bar", x="Algorithm", y=["RMSE", "STD"], figsize=(20, 10), rot=0
)
plt.xlabel("Algorithm", fontsize=20)
plt.ylabel("Root Mean Squared Error / Standard Deviation", fontsize=20)
plt.title("Top 3 Fold Time Series CV Avg RMSE & STD by Algorithm", fontsize=
plt.show()
```

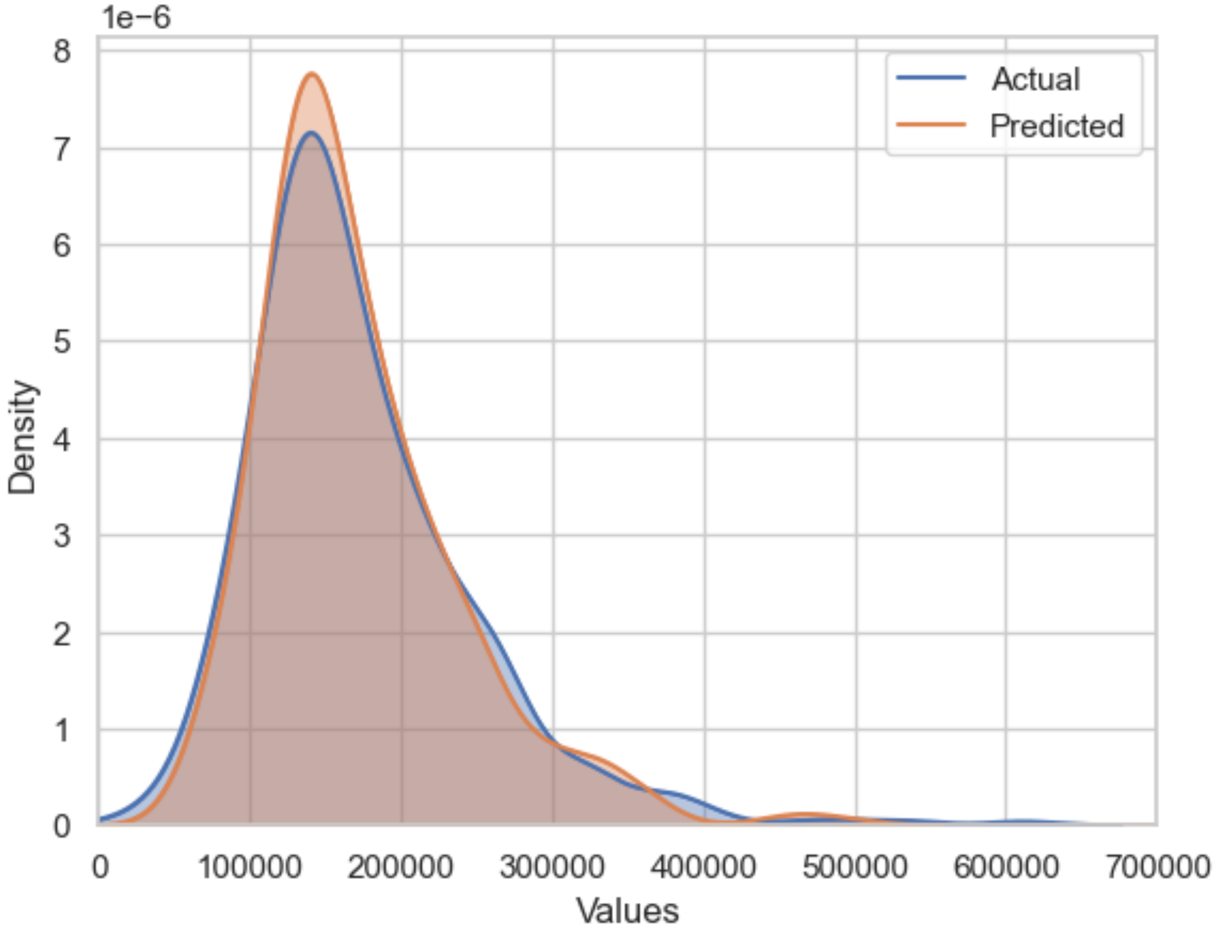


Looks like our meta learner is our best performer. Lets start digging into where our model is performs well, and where it does not.

To do this, let's look at the final fold as this has the most data to train with and is the most recent sale dates.

```
In [73]: ee_final_fold_preds = elastic_ensemble_preds_df[
             (elastic_ensemble_preds_df["Fold"] == 4)
             & (elastic_ensemble_preds_df["Set"] == "test")
         ]
         ee_final_fold_preds["difference"] = (
             ee_final_fold_preds["Actual"] - ee_final_fold_preds["Predicted"]
         )
```

```
In [76]: ee_final_fold_preds[["Actual", "Predicted"]].plot(kind="density", layout=(1,
# Assuming ee_final_fold_preds is a DataFrame with columns "Actual" and "Pre
sns.kdeplot(
    data=ee_final_fold_preds["Actual"],
    fill=True,
    common_norm=False,
    alpha=0.4,
    label="Actual",
)
sns.kdeplot(
    data=ee_final_fold_preds["Predicted"],
    fill=True,
    common_norm=False,
    alpha=0.4,
    label="Predicted",
)

# Add labels and legend
plt.xlabel("Values")
plt.ylabel("Density")
plt.xlim((0, 700000))
plt.show()
```



In [75]:

```
# sanity check on predictions
print("min prediction: ", ee_final_fold_preds["Predicted"].min())
print("max prediction: ", ee_final_fold_preds["Predicted"].max())
print("max error: ", ee_final_fold_preds["difference"].max())
print("mean error: ", abs(ee_final_fold_preds["difference"]).mean())
print("median error: ", abs(ee_final_fold_preds["difference"]).median())
```

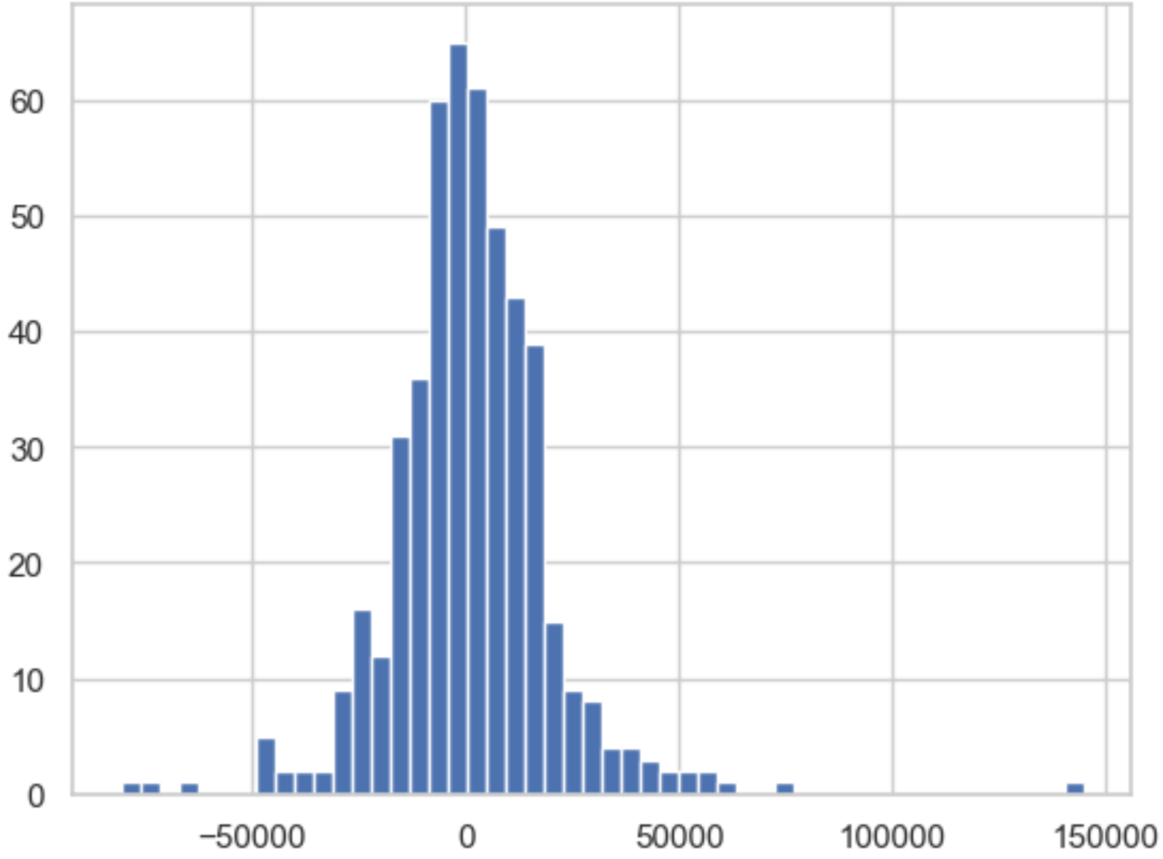
```
min prediction:  48500.071427404335
max prediction:  499652.1420222249
max error:  144726.12773093273
mean error:  12968.952084609497
median error:  9405.13514078573
```

In [77]:

```
# plot a histogram of the difference of our actuals and predictions
ee_final_fold_preds["difference"].hist(bins=50)
```

Out[77]:

```
<Axes: >
```



Looks like there is a large outlier we are off by (~150,000). Let's examine this record.

In [168…

```
ee_final_fold_preds[ee_final_fold_preds["difference"] >= 120000]
```

Out[168…

	Id	Actual	Predicted	Fold	Set	difference
7076	528150070	611657	466930.872269	4	test	144726.127731

In [170…

```
housing_df[housing_df["Id"] == 528150070]
```

Out[170…

	Id	MSSubClass	LotFrontage	LotArea	Street	Alley	LotShape
44	528150070	2.055642	6.65495	20.913578	0.730463	0.730463	0.0

We can further examine this record to see if there is something erroneous about it. For our purposes, we see our model performs well and our predicted and actual distributions are similar. Some other analysis that could occur:

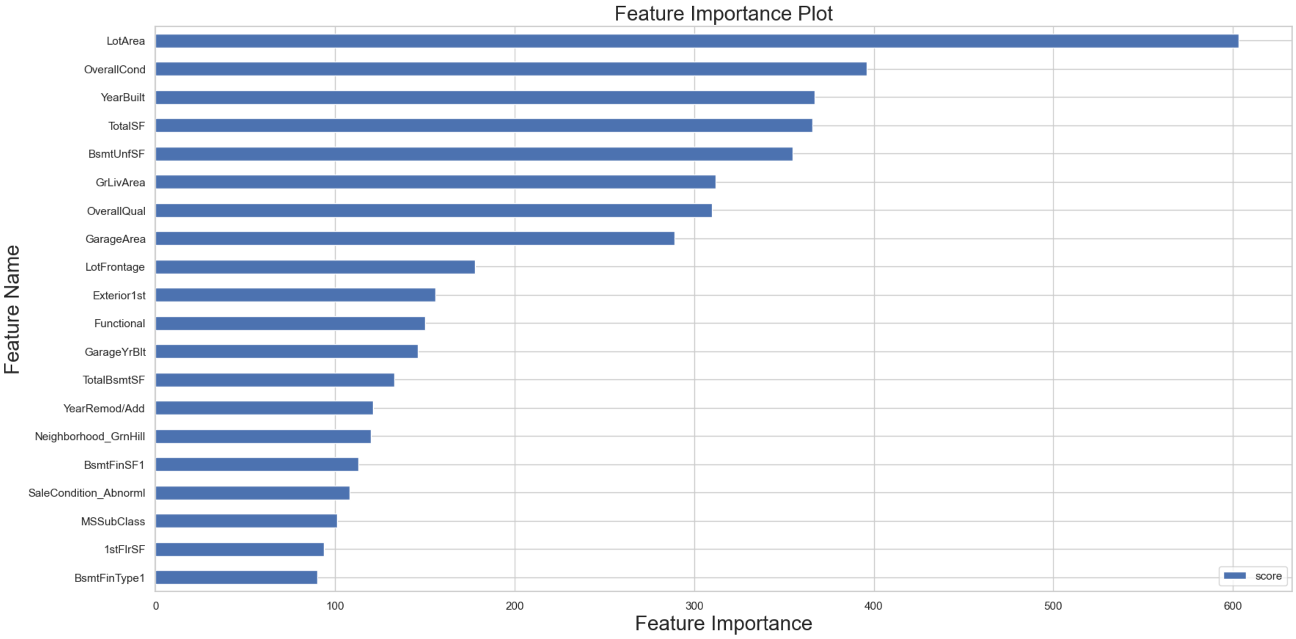
- EDA on where our model is strong/weak
- Further analysis around records that have the highest RMSE/Error

Variable Importance Plot

Tree based models (Decision Trees, Random Forest, GBMs) have feature importance plots that allow you to see which features have the most impact on our model. Let's take a look at our XGBoost model that we used in our meta-model to get a sense of which features are the most important.

In [181…

```
feature_important = model_xgb_hyper.get_booster().get_score(importance_type=

keys = list(feature_important.keys())
values = list(feature_important.values())

data = pd.DataFrame(data=values, index=keys, columns=["score"]).sort_values(
    by="score", ascending=False
)
data[:20].plot(kind="barh", figsize=(20, 10)).invert_yaxis()
## plot top 20 features
plt.xlabel("Feature Importance", fontsize=20)
plt.ylabel("Feature Name", fontsize=20)
plt.title("Feature Importance Plot", fontsize=20)
plt.show()
```


Feature Importance Plot
Feature Name
LotArea
OverallCond
YearBuilt
TotalSF
BsmtUnfSF
GrLivArea
OverallQual
GarageArea
LotFrontage
Exterior1st
Functional
GarageYrBlt
TotalBsmtSF
YearRemod/Add
Neighborhood_GrnHill
BsmtFinSF1
SaleCondition_Abnorml
MSSubClass
1stFlrSF
BsmtFinType1
score
0
100
200
300
400
500
600
Feature Importance