2025 UT Austin SPE Paper Contest - Jupyter Notebook

February 18, 2025

The University of Texas at Austin
Hildebrand Department of Petroleum and Geosystems Engineering
Cockrell School of Engineering

1 Leveraging Public Data for Production Forecasting in the Bakken: A Machine Learning Approach

1.1 Mathéo Valena, The University of Texas at Austin

This notebook contains the data science and machine learning workflow for the 2025 UT Austin SPE Paper Contest entry titled 'Leveraging Public Data for Production Forecasting in the Bakken: A Machine Learning Approach .'

1.2 Introduction

Understanding and predicting production is critical to optimizing reserves, planning operations, allocating resources, and making informed decisions about future drilling actions. Type curves, the industry standard for predicting future production, involves curve fitting surrounding historical production data. This method does not explictly make use of subsurface information, and does not give any insight into how production is changing with respect to the local conditions.

Given the complex physical relationships governing the subsurface, machine learning aglorithms are optimal for this type of job. This approach offers an alternative approach to using data for predicting future production, utilizing features derived from the field, and allows for interpretation of production based on how the machine learns to understand the relationships between variables.

1.3 Outline

A CatBoost machine learning model was trained on public completion and production data acquired from the Bakken formation. The following steps were involved in order: 1) Preliminary Data Visualization 2) Cleaning Production Data 3) Cleaning Completion Data 4) Analysis of Production Outliers 5) Building a Baseline Model 6) Building the CatBoost Model 7) Hyperparameter Tuning (provided as excel file) 8) CatBoost Model Interpretation 9) CatBoost Model using Kriging Interpolation 10) Geologic Interpretation of SHAP Values 11) Additional Work

[158]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import matplotlib.patches as mpatches
import seaborn as sns
import plotly.express as px
import math
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import StackingRegressor, RandomForestRegressor,␣
↪GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.preprocessing import StandardScaler, MinMaxScaler,␣
↪QuantileTransformer
from sklearn.calibration import calibration_curve
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import confusion_matrix
from catboost import CatBoostRegressor, Pool
from xgboost import XGBRegressor
from pykrige.ok import OrdinaryKriging
from scipy.interpolate import Rbf
import shap
import warnings
warnings.filterwarnings('ignore')
sns.set_theme(style="darkgrid")
```

1.3.1 Import dataframes

[159]:
```
# Load in the dataframes containing completion and production information
df_prod = pd.read_csv('Downloads/Petro Data Analysis Stuff/Export - Well␣
↪Production List (bakken)(csv).csv')
df_comp = pd.read_csv('Downloads/Petro Data Analysis Stuff/Export - Well␣
↪Completion List (bakken)(csv).csv')
```

[160]:
```
# Rename UWI to API
df_comp = df_comp.rename(columns={'UWI': 'API'})
```

[161]:
```
# Determine number of common wells among both datasets
apis_prod = df_prod['API'].unique()
apis_comp = df_comp['API'].unique()
common_elements = set(apis_prod) & set(apis_comp)  # Intersection
count_common = len(common_elements)
print(f'There are {count_common} wells in the combined datasets.')
```

```
There are 7575 wells in the combined datasets.
```

It is worth noting that all wells in this dataset are horizontal and are mainly producers of oil.

1.3.2 Define functions

[162]:
```
def create_location_map(df, name_of_x, name_of_y, target_hue, zoom):
    fig = px.scatter_mapbox(df,
                            lat=name_of_x,
                            lon=name_of_y,
                            color=target_hue,
                            color_continuous_scale=px.colors.sequential.Rainbow,
                            zoom=zoom,
                            height=500)

    # Use an open street map style
    fig.update_layout(mapbox_style='open-street-map')
    fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})

    fig.show()

def create_bar_plot(categories_list, values_list, palette, xlabel, ylabel):
    # Create a DataFrame
    days_on_df = pd.DataFrame({xlabel: categories_list, ylabel: values_list})

    # Create the barplot
    sns.barplot(x=xlabel, y=ylabel, data=days_on_df, palette=palette)

    # Display the plot
    plt.show()

def create_scatter_plot(figsize, x, y, data, xlabel, ylabel, title,␣
 ↪xticks=None, yticks=None, ):
    plt.figure(figsize=figsize)
    # Create the scatterplot
    sns.scatterplot(x=x, y=y, data=data)

    # Display the plot
    plt.xticks(xticks)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.show()

def create_box_plot(data, ylabel, title, color):
    sns.boxplot(y=data, color=color)
    plt.title(title)
    plt.ylabel(ylabel)
    plt.show()
```

```
def produce_k_months_liquid_df(k):
    k_months_producing_df = months_producing_df[months_producing_df['count'] >=␣
↪k]
    k_months_producing_df_apis = k_months_producing_df['API']
    k_months_api_filtered_prod_df = prod_df[prod_df['API'].
↪isin(k_months_producing_df_apis)]

    def filter_groups(group):
        return group.head(k)

    k_months_api_filtered_prod_df = k_months_api_filtered_prod_df.
↪groupby('API').apply(filter_groups).reset_index(drop=True)
    k_months_liquid_df = k_months_api_filtered_prod_df.groupby('API').agg(
        Liquid = ('Liquid', 'sum'),
        Initial_date = ('Date', 'first'),
        Final_data = ('Date', 'last'))
    return k_months_liquid_df

def scale_train_fit_eval_model(X, y, model):
    # Train-test split (80% train, 20% test)
    X_scaled = pd.DataFrame(scaler_standard.fit_transform(X), columns=X.columns)
    X_scaled_normalized = pd.DataFrame(scaler_minmax.fit_transform(X_scaled),␣
↪columns=X.columns)

    y_scaled = pd.DataFrame(scaler_standard.fit_transform(y), columns=y.columns)
    y_scaled_normalized = pd.DataFrame(scaler_minmax.fit_transform(y_scaled),␣
↪columns=y.columns)

    X_train, X_test, y_train, y_test = train_test_split(X_scaled_normalized,␣
↪y_scaled_normalized, test_size=0.2)
    model = model
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    y_pred_scaled = scaler_minmax.inverse_transform(y_pred.reshape(-1, 1))
    y_pred_original = scaler_standard.inverse_transform(y_pred_scaled.
↪reshape(-1, 1))

    y_test_scaled = scaler_minmax.inverse_transform(np.array(y_test['Liquid']).
↪reshape(-1, 1))
    y_test_original = scaler_standard.inverse_transform(y_test_scaled.
↪reshape(-1, 1))

    mse_original = mean_squared_error(y_test_original, y_pred_original)
    rmse_original = np.sqrt(mse_original)
```

```
    print(f'RMSE: {rmse_original:.5f}')

    squared_errors = (y_test_original - y_pred_original) ** 2
    median_squared_error = np.median(squared_errors)
    mrmse = np.sqrt(median_squared_error)
    print(f'MRMSE: {mrmse}')

    absolute_percent_errors = np.abs((y_test_original - y_pred_original) /␣
↪y_test_original) * 100
    mdape = np.median(absolute_percent_errors)
    print(f"Median Absolute Percent Error (MdAPE): {mdape:.2f}%")
    return y_test_original, y_pred_original

def train_fit_eval_model(X, y, model):
    # Train-test split (80% train, 20% test)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
    model = model
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)

    mse_original = mean_squared_error(y_test, y_pred)
    rmse_original = np.sqrt(mse_original)
    print(f'RMSE: {rmse_original:.5f}')

    squared_errors = (y_test - y_pred) ** 2
    median_squared_error = np.median(squared_errors)
    mrmse = np.sqrt(median_squared_error)
    print(f'MRMSE: {mrmse}')

    absolute_percent_errors = np.abs((y_test - y_pred) / y_test) * 100
    mdape = np.median(absolute_percent_errors)
    print(f"Median Absolute Percent Error (MdAPE): {mdape:.2f}%")
    return y_test, y_pred

def haversine(lat1, lon1, lat2, lon2):
    # Convert latitude and longitude from degrees to radians
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    # Haversine formula
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon␣
↪/ 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    # Radius of Earth in kilometers
```

```
    R_km = 6371.0

    # Calculate the distance in kilometers
    distance_km = R_km * c

    # Convert distance from kilometers to feet
    km_to_feet = 3280.84
    distance_feet = distance_km * km_to_feet

    return distance_feet

def create_well_data_year_b(b):
    wells_data = pd.DataFrame(columns=['API', f'Liquid_year_{b}', f'Mean␣
↪GOR_year_{b}'])
    prod_beg_b_wells = comp_df[comp_df['Year'] <= b][['API', 'Year']]
    prod_beg_b_wells_apis = prod_beg_b_wells['API'].unique()
    for i in tqdm(range(len(prod_beg_b_wells_apis))):
        well_i_api = prod_beg_b_wells_apis[i]
        well_i_yearb_data = prod_df[(prod_df['API'] == well_i_api) &␣
↪(prod_df['Year'] <= b)]
        well_i_yearb_liquid = well_i_yearb_data['Liquid'].sum()
        well_i_yearb_mean_gor = well_i_yearb_data['Ratio Gas Oil'].mean()
        well_i_row = pd.DataFrame([{
            'API': well_i_api,
            f'Liquid_year_{b}': well_i_yearb_liquid,
            f'Mean GOR_year_{b}': well_i_yearb_mean_gor}])
        wells_data = pd.concat([wells_data, well_i_row], ignore_index=True)
    return wells_data

def month_to_number(month_abbr):
    month_dict = {
        "JAN": 1, "FEB": 2, "MAR": 3, "APR": 4, "MAY": 5, "JUN": 6,
        "JUL": 7, "AUG": 8, "SEP": 9, "OCT": 10, "NOV": 11, "DEC": 12
    }
    return month_dict.get(month_abbr, None)

def months_between(year1, month1, year2, month2):
    return (year2 - year1) * 12 + (month2 - month1)

def gather_past_data_on_wells(df):
    for w in tqdm(range(len(df))):
        # Define target well
        well_1 = pc_use_comp_df.iloc[w]
        well_1_lat = well_1['Surface Latitude']
        well_1_long = well_1['Surface Longitude']

        # Calculate distance from given well and every other well
```

```
        pc_use_comp_df['Distance_from_well_Feet'] = df_comp.apply(lambda row:␣
↪haversine(row['Surface Latitude'], row['Surface Longitude'], well_1_lat,␣
↪well_1_long), axis=1)

        # Index to only include wells where the distance is less than 2500 ft
        pc_use_comp_less_2500 =␣
↪pc_use_comp_df[pc_use_comp_df['Distance_from_well_Feet'] < 2500]
        pc_use_comp_less_2500 = pc_use_comp_less_2500.
↪sort_values(by='Distance_from_well_Feet')

        # Define list of APIs
        other_well_apis = pc_use_comp_less_2500['API'][1:]
        current_well_api = pc_use_comp_less_2500['API'].iloc[0]
        print(current_well_api)

        # Gather information for the target well
        if len(prod_df[prod_df['API'] == current_well_api]) != 0:
            current_api_start_year, current_api_start_month =␣
↪prod_df[prod_df['API'] == current_well_api].iloc[0]['Year'],␣
↪prod_df[prod_df['API'] == current_well_api].iloc[0]['Month']
            current_api_start_month = month_to_number(current_api_start_month)
            print(current_api_start_year, current_api_start_month)
        else:
            continue

        # Define data
        total_liquid_before = 0
        total_gas_before = 0
        num_of_well_prior = 0

        # Loop over every well less than 2500 ft and determine how much␣
↪production of liquid, gas, they have together (if any at all)
        if len(other_well_apis) > 0:
            for l in range(len(other_well_apis)):
                other_well_i = prod_df[prod_df['API'] == other_well_apis.
↪iloc[l]]
                print(other_well_apis.iloc[l])
                if len(other_well_i) == 0: # Ignore wells that are not in both␣
↪comp_df and prod_df
                    continue
                other_start_year, other_start_month = other_well_i.
↪iloc[0]['Year'], other_well_i.iloc[0]['Month']
                other_start_month = month_to_number(other_start_month)
                months_between_wells = months_between(current_api_start_year,␣
↪current_api_start_month, other_start_year, other_start_month)
                # print(months_between_wells)
```

```
                if months_between_wells < 0: #Calculate info for wells that␣
↪have production prior to the target well
                    num_of_well_prior += 1
                    other_well_liquid = other_well_i[:
↪abs(months_between_wells)]['Liquid'].sum()
                    other_well_gas = other_well_i[:
↪abs(months_between_wells)]['Gas'].fillna(0).sum()
                    total_liquid_before += other_well_liquid
                    total_gas_before += other_well_gas
                else:
                    continue

        # Define data dictionary
        pc_well_data = pd.DataFrame([{
            'API': current_well_api,
            'Total Liquid Before': total_liquid_before,
            'Total Gas Before': total_gas_before,
            'Number of Neighbouring Wells Prior to Prod.': num_of_well_prior
        }])

        global pc_relations_comp_df
        pc_relations_comp_df = pd.concat([pc_relations_comp_df, pc_well_data],␣
↪ignore_index=True)

def eval_model(y_test, y_pred):
    mse_original = mean_squared_error(y_test, y_pred)
    rmse_original = np.sqrt(mse_original)
    print(f'RMSE: {rmse_original:.5f}')

    squared_errors = (y_test - y_pred) ** 2
    median_squared_error = np.median(squared_errors)
    mrmse = np.sqrt(median_squared_error)
    print(f'MRMSE: {mrmse}')

    absolute_percent_errors = np.abs((y_test - y_pred) / y_test) * 100
    mape = np.mean(absolute_percent_errors)
    print(f"Mean Absolute Percent Error (MAPE): {mape:.2f}%")

    mdape = np.median(absolute_percent_errors)
    print(f"Median Absolute Percent Error (MdAPE): {mdape:.2f}%")

    maxpe = np.max(absolute_percent_errors)
    print(f"Max Absolute Percent Error (MAPE): {maxpe:.2f}%")

    minpe = np.min(absolute_percent_errors)
    print(f"Min Absolute Percent Error (MAPE): {minpe:.2f}%")
```

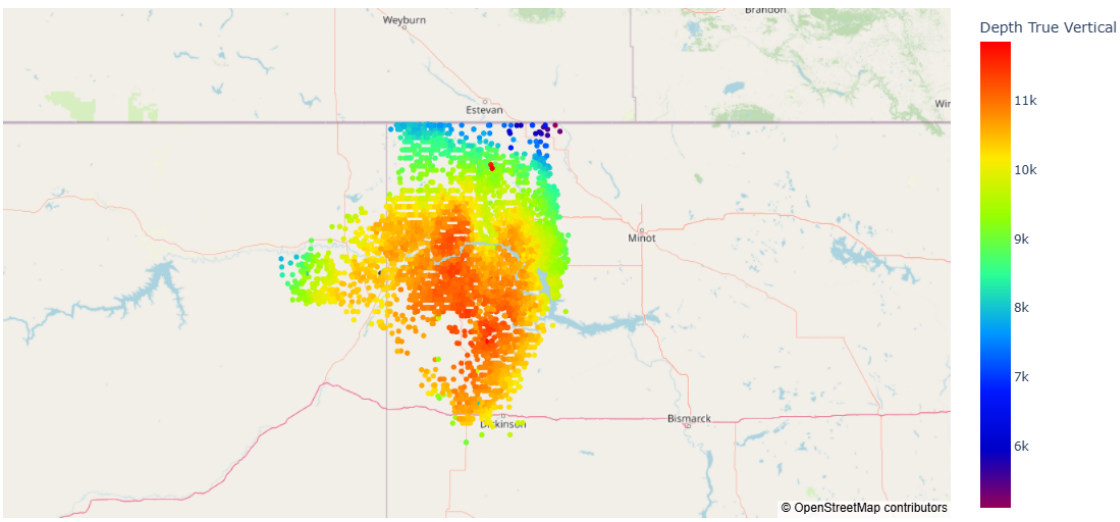
```
def eval_model_simple(y_test, y_pred):
    absolute_percent_errors = np.abs((np.array(y_test) - np.array(y_pred)) / np.
 ↪array(y_test)) * 100
    mdape = np.median(absolute_percent_errors)
    print(f"Median Absolute Percent Error (MdAPE): {mdape:.2f}%")

def highlight_high_values(val):
    color = 'red' if val > 3 else 'black'
    return f'color: {color}'

def rerun_with_SHAP():
    train_pool_2 = Pool(data=cat_train.
 ↪drop(columns=target+location_info+['prediction']+cat_features[-4:]),␣
 ↪label=cat_train[target], cat_features=['Operator Name'])
    monotonic_constraints = [0, 1, 1, 1]
    model_2 = CatBoostRegressor(iterations=400, learning_rate=0.1, depth=5,␣
 ↪verbose=0, monotone_constraints=monotonic_constraints)
    model_2.fit(train_pool_2)

    predictions = model_2.predict(cat_train.
 ↪drop(columns=target+location_info+['prediction']+cat_features[-4:]))
    y_pred = model_2.predict(cat_test.
 ↪drop(columns=target+location_info+cat_features[-4:]))

    y_train = cat_train['Liquid']
    y_train_pred = predictions
    eval_model_simple(y_train, y_train_pred)
    print("-----------------------------------")
    y_test = cat_test['Liquid']
    eval_model_simple(y_test, y_pred)
```

[163]:
```
# Remove columns with little data and relevance
prod_df_columns_to_drop = ['Well Number', 'Wells', 'Unnamed: 15', 'Unnamed:␣
 ↪16', 'JAN', '1', '31', 'Lat Len Gross Perf Intvl', 'UWI']
comp_df_columns_to_drop = ['Sub Basin Code', 'Depth Whipstock', 'PRODFit␣
 ↪Formation at TD Name', 'PRODFit Top Formation Producing Name',
                           'PRODFit Base Formation Producing Name', 'PRODFit 80␣
 ↪Degree Heel Point Formation', 'Date First Spud', 'Date Rig Release',
                           'Depth Water Value', 'Source', 'Surface LL Source']
df_comp = df_comp.rename(columns={'Unnamed: 49': 'Lateral Length'})
prod_df = df_prod.drop(columns=prod_df_columns_to_drop)
comp_df = df_comp.drop(columns=comp_df_columns_to_drop)
comp_df['Date Completion'] = pd.to_datetime(comp_df['Date Completion'])
comp_df['Year'] = comp_df['Date Completion'].dt.year
```

1.3.3 Preliminary Visualizations

[164]:
```
target_hue = 'Depth True Vertical'

df_visualize_comp = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp['API'] = comp_df['API']
df_visualize_comp['X'] = comp_df['Surface Latitude']
df_visualize_comp['Y'] = comp_df['Surface Longitude']
df_visualize_comp[target_hue] = comp_df[target_hue]
```

[165]:
```
# Visualize the distribution of values across the Bakken
create_location_map(df_visualize_comp, 'X', 'Y', target_hue, 6)
```

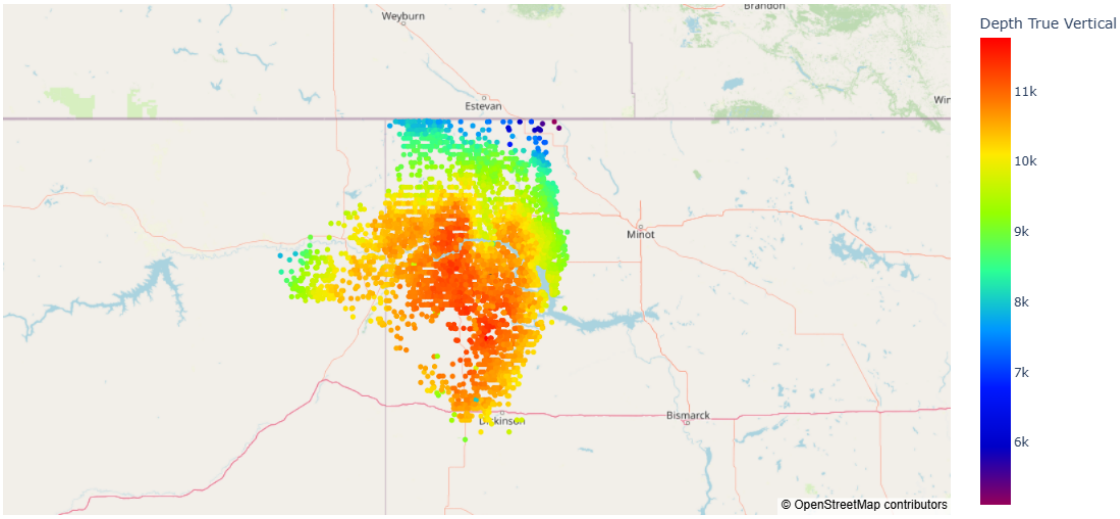


Observe that there seems to be two strong outliers with large depths (red) in the middle of the middle depths region. Depth increases as we approach the center of the basin.

Let's look at the distribution once we have merged both dataframes.

[166]:
```
df_visualize_comp_prod = df_visualize_comp[df_visualize_comp['API'].
↪isin(apis_prod)]
```

[167]:
```
# Visualize the distribution of values across the Bakken
create_location_map(df_visualize_comp_prod, 'X', 'Y', target_hue, 6)
```



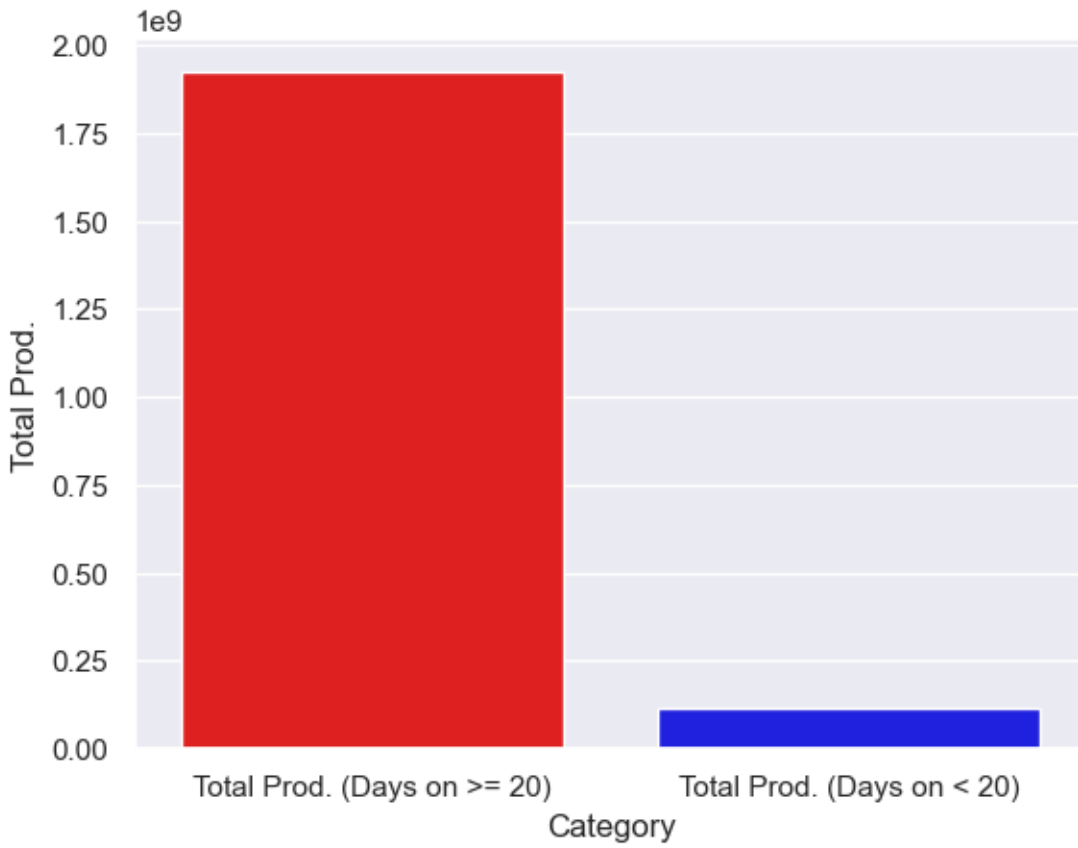
Almost no difference! That means we lose little coverage of the range of original values with our lost data.

1.4 Cleaning Production Data

Our liquid production data contains no NaN values, which is very helpful. However, other columns with missing data require cleaning.

[168]:
```
reference_days_on = 20
higher_days_on_liquid = prod_df[prod_df['Days On'] >=␣
↪reference_days_on]['Liquid'].sum()
lower_days_on_liquid = prod_df[prod_df['Days On'] <␣
↪reference_days_on]['Liquid'].sum()
categories = [f'Total Prod. (Days on >= {reference_days_on})', f'Total Prod.␣
↪(Days on < {reference_days_on})']
values = [higher_days_on_liquid, lower_days_on_liquid]
palette = ['red', 'blue']
```

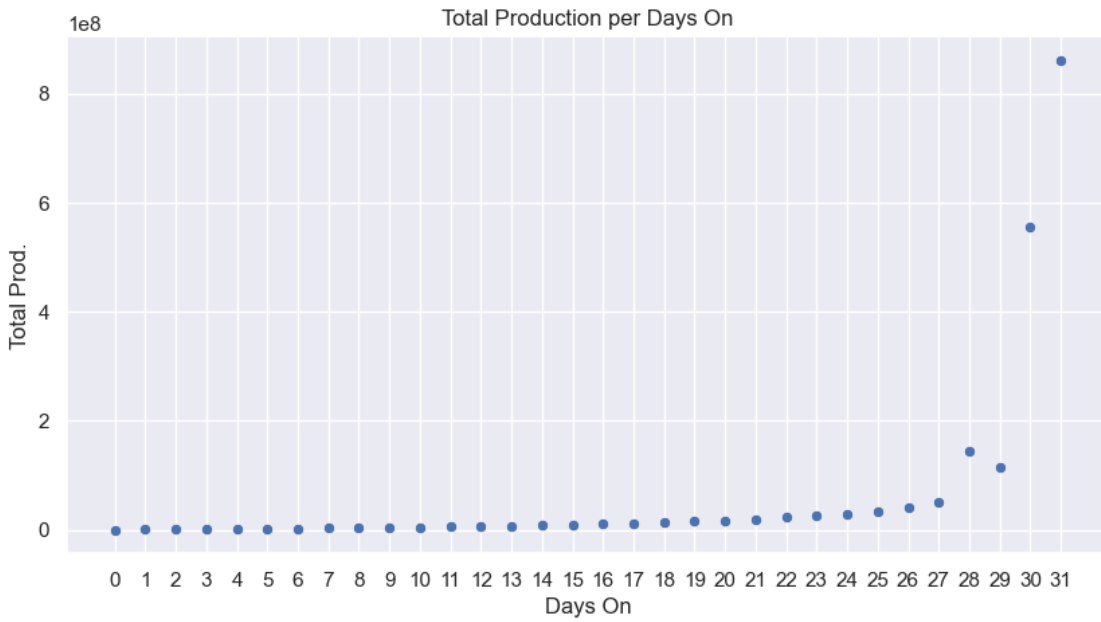
[169]:
```
create_bar_plot(categories, values, palette, 'Category', 'Total Prod.');
```



We see that production for wells where days on was greater than or equal to 20 far outmatches production where days on was less than 20. Let's visualize production over days on.

[170]:
```
visualize_days_on_prod_df = prod_df[['Liquid', 'Days On']].groupby('Days On').
 ↪sum('Liquid').reset_index()
figsize = (10, 5)
x_days_liquid = visualize_days_on_prod_df['Days On']
y_days_liquid = visualize_days_on_prod_df['Liquid']
```

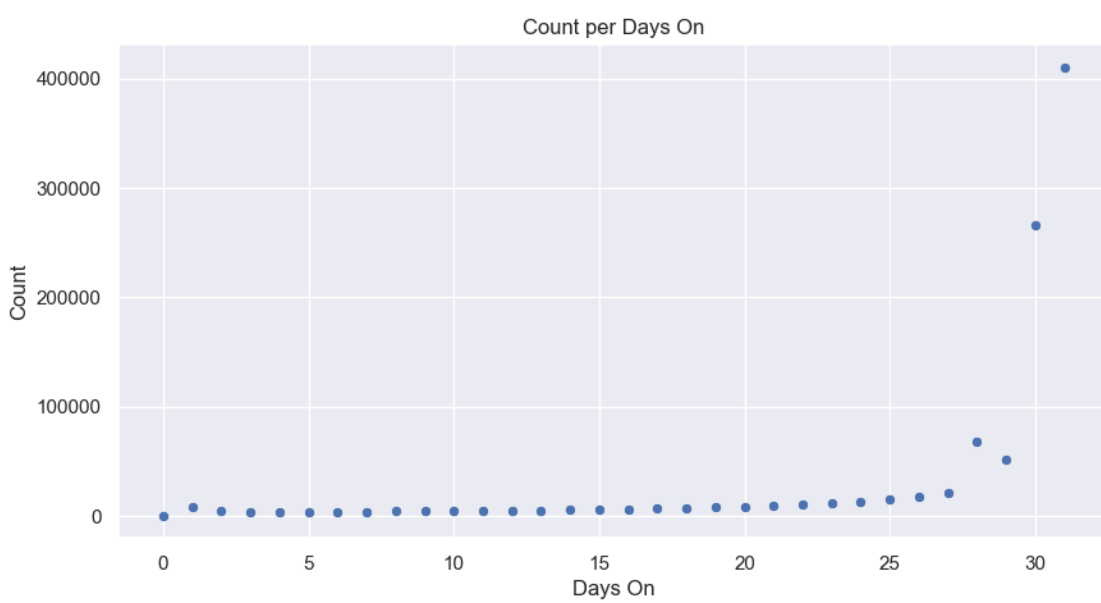
[171]:
```
create_scatter_plot(figsize, x_days_liquid, y_days_liquid,␣
 ↪visualize_days_on_prod_df, 'Days On', 'Total Prod.',
                    'Total Production per Days On',␣
 ↪xticks=visualize_days_on_prod_df['Days On'])
```



We observe that total production increases expontentially with the days on. Higher amount of values belonging to higher days will skew this graph, so let's investigate the value counts of each.

[172]:
```
volume_days_on_df = prod_df['Days On'].value_counts().reset_index()
figsize = (10, 5)
```

[173]:
```
create_scatter_plot(figsize, volume_days_on_df['Days On'],␣
↪volume_days_on_df['count'], volume_days_on_df, 'Days On', 'Count',
                    'Count per Days On')
```



As expected, the count is practically in 1:1 correspondence with the Liquid when plotted over Days On. Note that the majority of wells have 28+ days on, which is a good sign for production.

Let's determine if we can impute Days On with 0 by showing that the missing values are always correlated to low production.

[174]:
```
nan_days_on_df = prod_df[prod_df['Days On'].isna()]
```

[175]:
```
nan_days_liquid_zero = nan_days_on_df[nan_days_on_df['Liquid'] == 0][['Liquid',␣
↪'API']]
nan_days_liquid_non_zero = nan_days_on_df[nan_days_on_df['Liquid'] !=␣
↪0][['Liquid', 'API']]
```

[176]:
```
print(f'There are {len(nan_days_liquid_zero)} days where NaN "Days On" have 0␣
↪production.')
print(f'There are {len(nan_days_liquid_non_zero)} days where NaN "Days On" do␣
↪NOT have 0 production.')
```

```
There are 48313 days where NaN "Days On" have 0 production.
There are 74 days where NaN "Days On" do NOT have 0 production.
```

For the majority of cases, we have zero production where 'Days On' is missing. For these cases, we can nicely impute Days On by replacing it with 0. For the other cases, further analysis is needed. Let's investigate these cases.

[177]:
```
num_of_cases_api = len(nan_days_liquid_non_zero['API'].unique())
print(f'There are {num_of_cases_api} API values where "Days On is NaN but␣
↪production that month is nonzero."')
```

```
There are 56 API values where "Days On is NaN but production that month is
nonzero."
```

Observe that a lot of these are nonzero, but have Liquid = 1. This is essentially a placeholder for zero, and we can replace Days On with zero for these cases aswell.

[178]:
```
num_of_cases_1 =␣
↪len(nan_days_liquid_non_zero[nan_days_liquid_non_zero['Liquid'] == 1])
print(f'There are {num_of_cases_1} days where Liquid is 1, not 0, and "Days On"␣
↪is NaN')
```

```
There are 40 days where Liquid is 1, not 0, and "Days On" is NaN
```

[179]:
```
nan_days_liquid_non_zero_and_non_1_apis =␣
↪nan_days_liquid_non_zero[nan_days_liquid_non_zero['Liquid'] != 1]['API'].
↪unique()
print('APIs to investigate:', len(nan_days_liquid_non_zero_and_non_1_apis))
```

```
APIs to investigate: 18
```

We will manually investigate these wells to see if we can impute 'Days On' based on Liquid Production

[180]:
```
# To be continued...
```

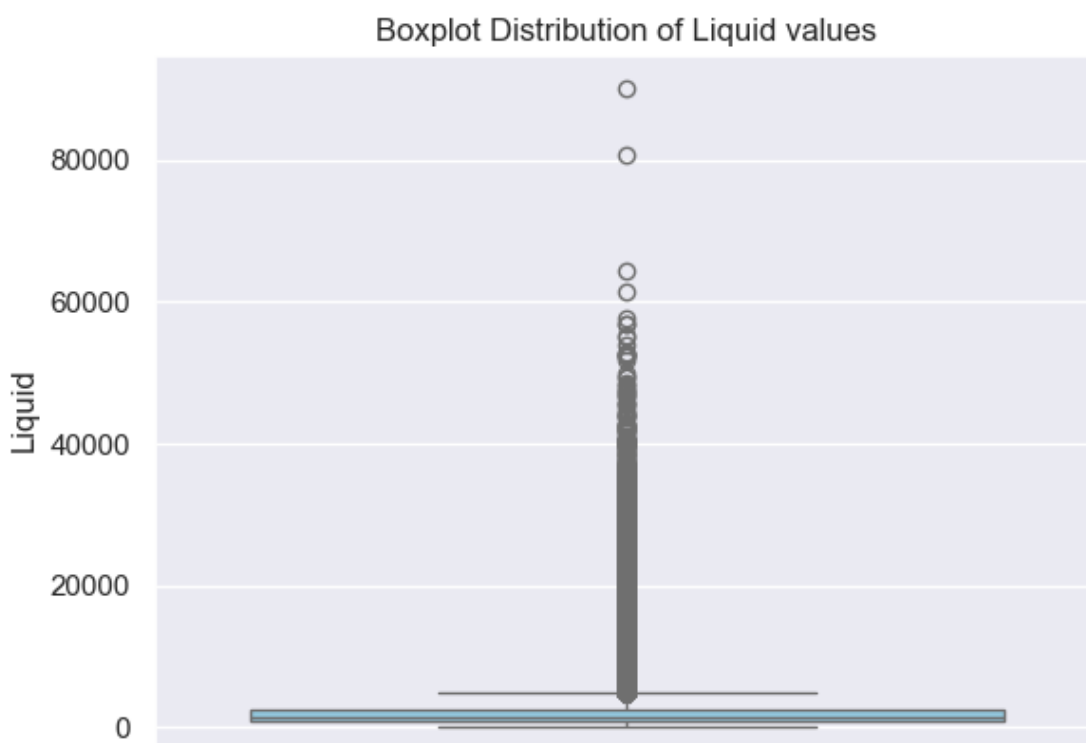
1.5 Determination of Liquid-Days On correlation

It is necessary to determine the correlation between Days On and Liquid in order to ensure our Liquid values make sense. If liquid is consistent with 'Days On,' then we can confidently say our Liquid production values are reasonable. We have shown above that most missing values align with Days On most likely being 0, so we do not need to worry about those cases, and we can turn out attention to the cases where data is not missing.

[181]:
```
liquid_days_on_corr_df = prod_df[['Liquid', 'Days On', 'API']]
```

[182]:
```
data_liquid_days_31 = liquid_days_on_corr_df[liquid_days_on_corr_df['Days On']␣
↪== 31]
```

[183]:
```
create_box_plot(data_liquid_days_31['Liquid'], 'Liquid', 'Boxplot Distribution␣
↪of Liquid values', 'skyblue')
```



There is a large range of values and many outliers for a given selection of 'Days On' (above example is 'Days On' = 31). Thus, 'Days On' is not a reliable indicator of whether production is correct or not.

Since we do not know the source, it is not clear whether we should trust 'Days On' or Liquid when there is a discrepency. We choose to trust 'Liquid' and proceed forward.

1.5.1 Notes on missing Gas, Water values

Observe the statistical distribution of the Liquid values where Gas is NaN. Most of them have Liquid being 0. This makes sense; Operators didn't input values because they didn't need to.

[184]:
```
gas_nan_df = prod_df[prod_df['Gas'].isna()]
gas_nan_df['Liquid'].describe()
```

[184]:
```
count    53694.000000
mean       101.312940
std        893.173651
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max      40491.000000
Name: Liquid, dtype: float64
```

This is further reinforced by the fact that the ratio of gas to oil is absolutely 0 for all cases where Gas was missing.

[185]:
```
gas_nan_df['Ratio Gas Oil'].describe()
```

[185]:
```
count    53694.0
mean         0.0
std          0.0
min          0.0
25%          0.0
50%          0.0
75%          0.0
max          0.0
Name: Ratio Gas Oil, dtype: float64
```

A similar analysis may be done with water.

[186]:
```
water_nan_df = prod_df[prod_df['Water'].isna()]
water_nan_df['Liquid'].describe()
```

[186]:
```
count    56777.000000
mean       163.892509
std       1423.218321
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max      57239.000000
```

```
Name: Liquid, dtype: float64
```

It is important to note that to produce oil and no water or gas together is highly improbable (if not impossible). For now, we ignore the cases where this might have happened and turn our attention to just the liquid values.

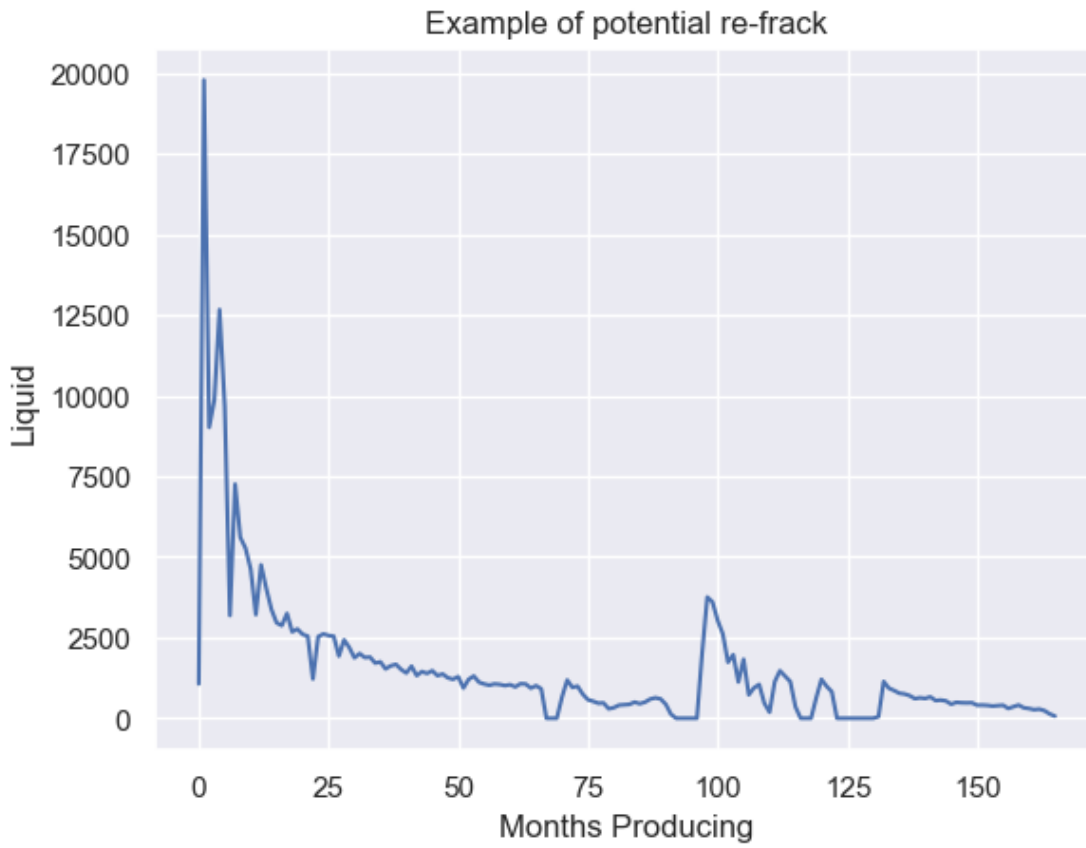
1.5.2 Analysis of the Liquid column

There are no missing Liquid values, and we can trust that instances where production is 0 are cases where 'Days On' is 0; That is to say, there is no production that month. We must still identify whether the trends of production make sense (follow a general decline). This leads us to consider several cases: - Was there a re-frack that led to a spike in production? - Was there an operator change at any point during the life of the well (operational efficiency change)? - Are there cases of duplicate wells? - Are there cases where production was systematically distributed via ratios? This means the overall lease production data was known, but not the individual well data, leading to a distribution based on the IP. This is not a proper reflection of reality. - Are there wells with less than 12 months of production? If so, how many?

Some of these questions are difficult to answer, but are worth considering moving forward. For now, we will only check for operator changess, duplicates, re-fracks, and well with less than 12 months of production.

Consider for example the well below. This well was operationally quiet for several months before it came back online and experienced a spike. This is a sound indicator of a re-frack, and may lead to later confusion by a model if ignored.

[187]:
```
example_well = prod_df[prod_df['API'] == apis_prod[10]].reset_index()['Liquid']
example_well.plot()
plt.title('Example of potential re-frack')
plt.xlabel('Months Producing')
plt.ylabel('Liquid');
```



Let's also check all instances in which there was an operator change by comparing 'Operator Name' with 'Current Operator Name.'

[188]:
```
operator_prod_comp_df = comp_df[comp_df['API'].isin(apis_prod)][['Operator␣
↪Name', 'Current Operator Name', 'API']]
```

[189]:
```
operator_change_df = operator_prod_comp_df[operator_prod_comp_df['Operator␣
↪Name'] != operator_prod_comp_df['Current Operator Name']]
operator_change_cases = len(operator_change_df)
print(f'There are {operator_change_cases} cases where the operator changed␣
↪throughout the life of the well.')
```

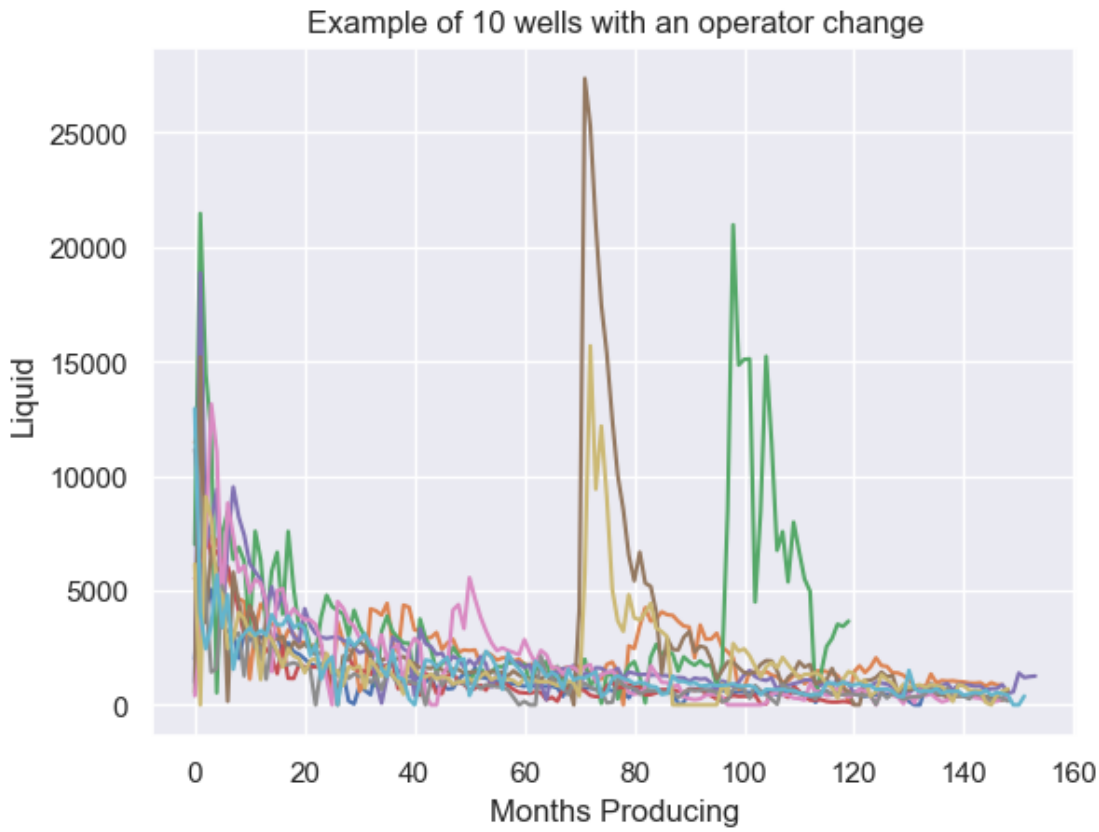
```
There are 3406 cases where the operator changed throughout the life of the well.
```

Let's create a column to track this.

[190]:
```
operator_change_api_series = operator_change_df['API']
comp_df['Operator Change'] = comp_df['API'].isin(operator_change_api_series).
↪astype(int)
```

Let's take an overlay of 10 example wells to see if we notice anything.

[191]:
```
for i in range(10):
    operator_change_well_api = operator_change_df['API'].iloc[i]
    operator_change_well_ex = prod_df[prod_df['API'] ==␣
 ↪operator_change_well_api] .reset_index()['Liquid']
    operator_change_well_ex.plot()
    plt.title('Example of 10 wells with an operator change')
    plt.xlabel('Months Producing')
    plt.ylabel('Liquid');
```



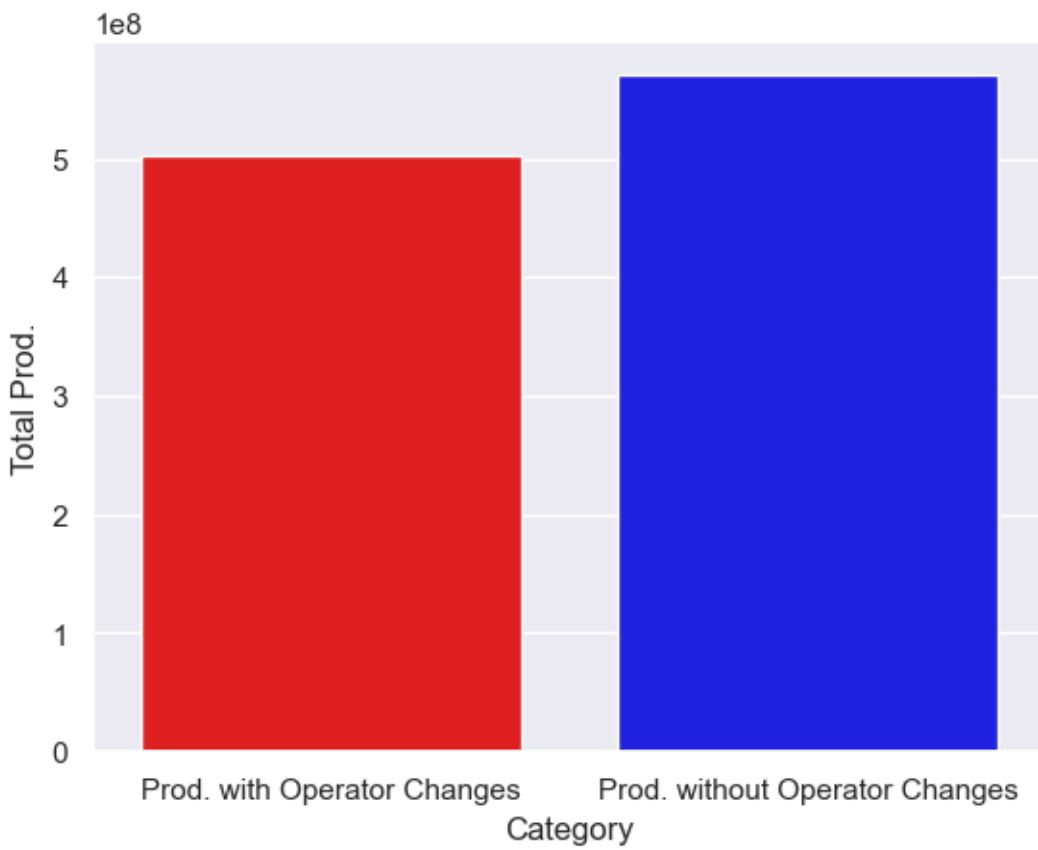
Interesting. An operator change seems to be associated with increasing in production with varying degree. Perhaps this could be important later for better understanding production values. We note that spikes in production may be a consequence of re-fracks, operational changes, or both. In building a good model, we may find it useful to remove wells that experienced an operational change, since such information won't be available prior to production.

Let's take a sample of 2000 wells from wells with operator changes and without operator changes to compare their combined productions.

[192]:
```
n = 2000

operator_change_df_apis = operator_change_df['API'].unique()
```

```
prod_with_operator_changes_df = prod_df[prod_df['API'].
 ↪isin(operator_change_df_apis)]
prod_with_operator_changes_sample = prod_with_operator_changes_df.
 ↪groupby('API')['Liquid'].sum().sample(n)

prod_without_operator_changes_df = prod_df[~prod_df['API'].
 ↪isin(operator_change_df_apis)]
prod_without_operator_changes_sample = prod_without_operator_changes_df.
 ↪groupby('API')['Liquid'].sum().sample(n)

prod_with_operator_changes_liquid = prod_with_operator_changes_sample.sum()
prod_without_operator_changes_liquid = prod_without_operator_changes_sample.
 ↪sum()

categories = ['Prod. with Operator Changes', 'Prod. without Operator Changes']
values = [prod_with_operator_changes_liquid,␣
 ↪prod_without_operator_changes_liquid]
palette = ['red', 'blue']
```

[193]:
```
create_bar_plot(categories, values, palette, 'Category', 'Total Prod.');
```



We notice that without operator changes, wells seem to produce more in total, the opposite of what we were expecting. This is likely due to the fact that an operator change is associated with delays in production and instability. Note that this difference is not super drastic, but noticeable. Note that when we perform this analysis out to 48 month production, the reverse is true. This shows that an operator change has short term negatives but long term benefits.

Let's see how many wells have less than 12 months of production data available.

[194]:
```
cases_less_12_months = sum(prod_df['API'].value_counts() < 12)
print(f'There are {cases_less_12_months} APIs with less than 12 months of␣
↪production data.')
```

```
There are 12 APIs with less than 12 months of production data.
```

To find duplicates based on identical curve structures would be computationally very expensive. This would require checking if all the Liquid values of a well are in a ratio with one or more other wells, which would necessiate a cross check of every value with every other value, for each well. This is too much. We can still find whether two liquid production values are the same. Let's check cases where more than 1 well is present per lease.

[195]:
```
leases_more_than_1_well = sum(comp_df[df_comp['API'].isin(apis_prod)]['Lease␣
↪Name'].value_counts() > 1)
total_leases = len(comp_df[df_comp['API'].isin(apis_prod)]['Lease Name'].
↪unique())
print(f'{leases_more_than_1_well*100 / total_leases:.2f}% of leases have more␣
↪than 1 well')
```

```
32.28% of leases have more than 1 well
```

[196]:
```
cases_where_prod_same = sum(prod_df.groupby('API')['Liquid'].sum().
↪value_counts() > 1)
print(f'There are {cases_where_prod_same} cases where production for the two␣
↪wells is the same.')
```

```
There are 77 cases where production for the two wells is the same.
```

Let's investigate if this is a coincidence or not.

[197]:
```
prod_df_api_liquid_grouped = prod_df.groupby('API')['Liquid'].sum().
↪reset_index()
liquid_duplicates_df = prod_df_api_liquid_grouped[prod_df_api_liquid_grouped.
↪duplicated(subset=['Liquid'], keep=False)]
duplicate_liquid_vals = liquid_duplicates_df['Liquid'].value_counts().
↪reset_index()['Liquid']
```

[198]:
```
duplicate_count = 0
for m in range(77):
    duplicate_apis = liquid_duplicates_df[liquid_duplicates_df['Liquid'] ==␣
↪duplicate_liquid_vals[m]]['API']
```

```
    well_1 = prod_df[prod_df['API'] == duplicate_apis.iloc[0]]
    well_2 = prod_df[prod_df['API'] == duplicate_apis.iloc[1]]
    well_1_count = well_1['Count'].iloc[-1]
    well_2_count = well_2['Count'].iloc[-1]
    if well_1_count == well_2_count:
        duplicate_count += 1
```

We have defined a variable called 'duplicate_count' which gets added 1 everytime two wells are identical (have both the same production and same duration of production).

[199]:
```
duplicate_count
```

[199]: 0

How interesting! Although we had 77 pairs of wells with identical production, this was a pure coincidence. We may then comfortably proceed without worrying about duplicates.

Let's ensure we have no negative production values by looking at the statistical information.

[200]:
```
prod_df['Liquid'].describe().to_frame().transpose()
```

[200]:
```
             count         mean          std  min    25%     50%     75%  \
Liquid  1045080.0  1951.373243  2736.347982  0.0  574.0  1135.0  2238.0

             max
Liquid  90101.0
```

Looks good.

1.5.3 Summary of Liquid values

- We have 12 wells with less than 12 months of production data.
- We have no negative Liquid values
- Wells without operator changes exhibit greater total production
- Operator changes can be associated with spikes in production curves
- 45% of wells have experienced an operator change
- 32.3% of leases contain more than 1 well
- There 77 cases of wells with identical production values, but there are no duplicate wells

We now have a completed list of Liquid values, and we are ready to move onto the step of feature selection, along with cleaning of the completion dataframe.

1.6 Cleaning the Completion Data

An idea of interest is to use the formation at TD as a proxy for geology, and County Name as a proxy for well similarity. This will be considered later via visualizations that attempt to distringuish production within these features.

We will create columns for production for 12 months, 24 months, 36 months, etc. First, we must remove the 12 wells that have less than 12 months of production data.

[201]:
```
months_producing_df = prod_df['API'].value_counts().reset_index()
less_than_12_df = months_producing_df[months_producing_df['count'] < 12]
less_than_12_df_apis = less_than_12_df['API']
prod_df = prod_df[~prod_df['API'].isin(less_than_12_df_apis)]
```

Let's verify the 12 have been removed.

[202]:
```
num_wells_after = len(prod_df['API'].unique())
print(f'There are {num_wells_after} wells after removal of the 12␣
↪underproducing wells.')
```

```
There are 7563 wells after removal of the 12 underproducing wells.
```

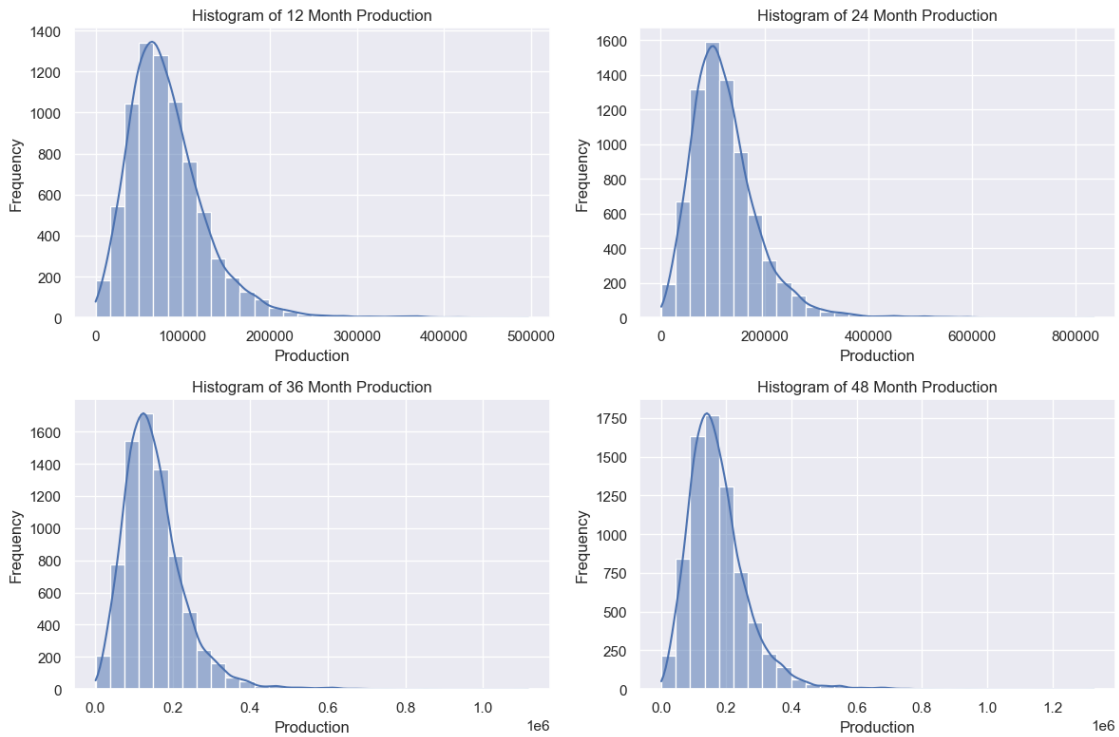
Now let's create the production values. To do this, we will define a function that takes in the production duration, and returns a dataframe with all values that have production up to that point, summed.

[203]:
```
month_12_liquid = produce_k_months_liquid_df(12)['Liquid']
month_24_liquid = produce_k_months_liquid_df(24)['Liquid']
month_36_liquid = produce_k_months_liquid_df(36)['Liquid']
month_48_liquid = produce_k_months_liquid_df(48)['Liquid']
```

Let's visualize the distribution of liquid values per each length of production duration.

[204]:
```
series_list = [month_12_liquid, month_24_liquid, month_36_liquid,␣
↪month_48_liquid]
series_labels = ['12 Month Production', '24 Month Production', '36 Month␣
↪Production', '48 Month Production']
```

[205]:
```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, series_labels):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel('Production')
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```

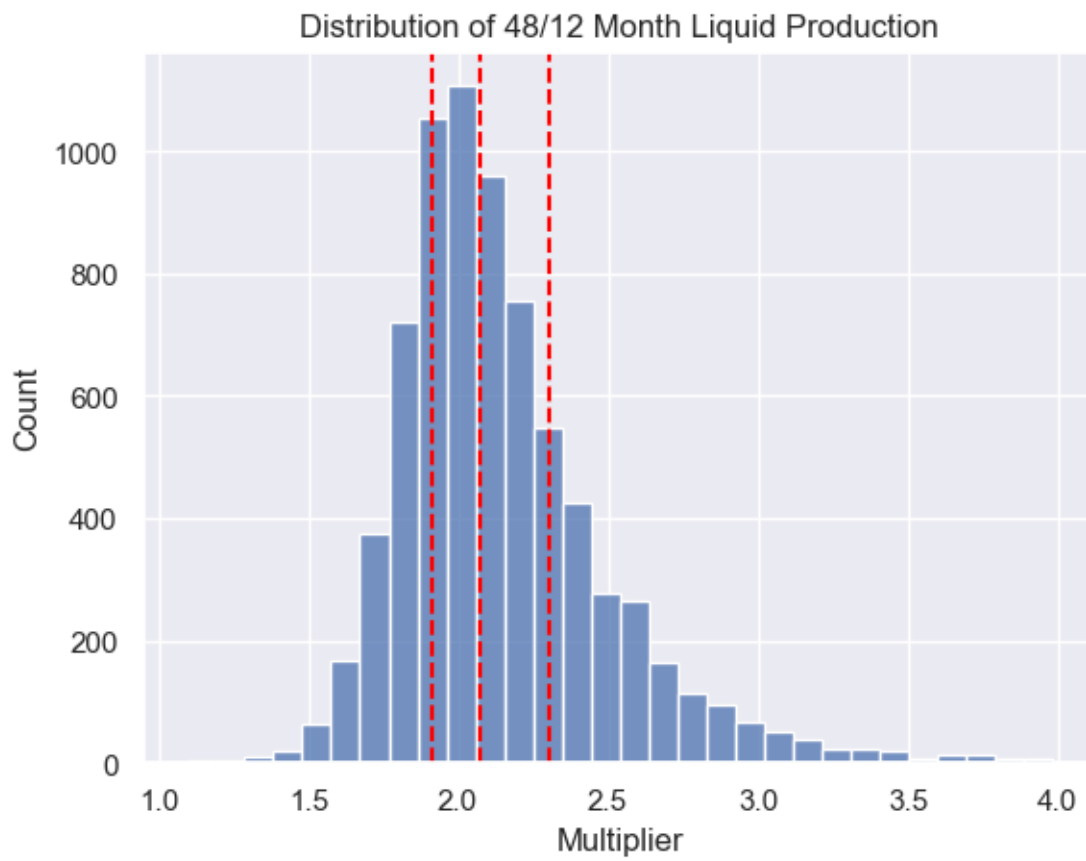


We observe a flattening of the tail as the duration of production increases, associated with less uncertainty around where values lie. However, the overall shape of the graphs remains relatively consistent. This is a good sign, and indicates 2 year production values may be indicative of 4 year production values and beyond.

[206]:
```
multiplier_12_48 = month_48_liquid / month_12_liquid[month_12_liquid.index.
↪isin(month_48_liquid.index)]
```

Let's take a look at the multiplier between the 12 month and 48 month data.

[207]:
```
percentiles = np.percentile(multiplier_12_48[multiplier_12_48 < 4], [25, 50,␣
↪75])
sns.histplot(multiplier_12_48[multiplier_12_48 < 4], bins=30)

for p in percentiles:
    plt.axvline(p, color='red', linestyle='dashed')

plt.title('Distribution of 48/12 Month Liquid Production')
plt.xlabel('Multiplier')
plt.ylabel('Count');
```

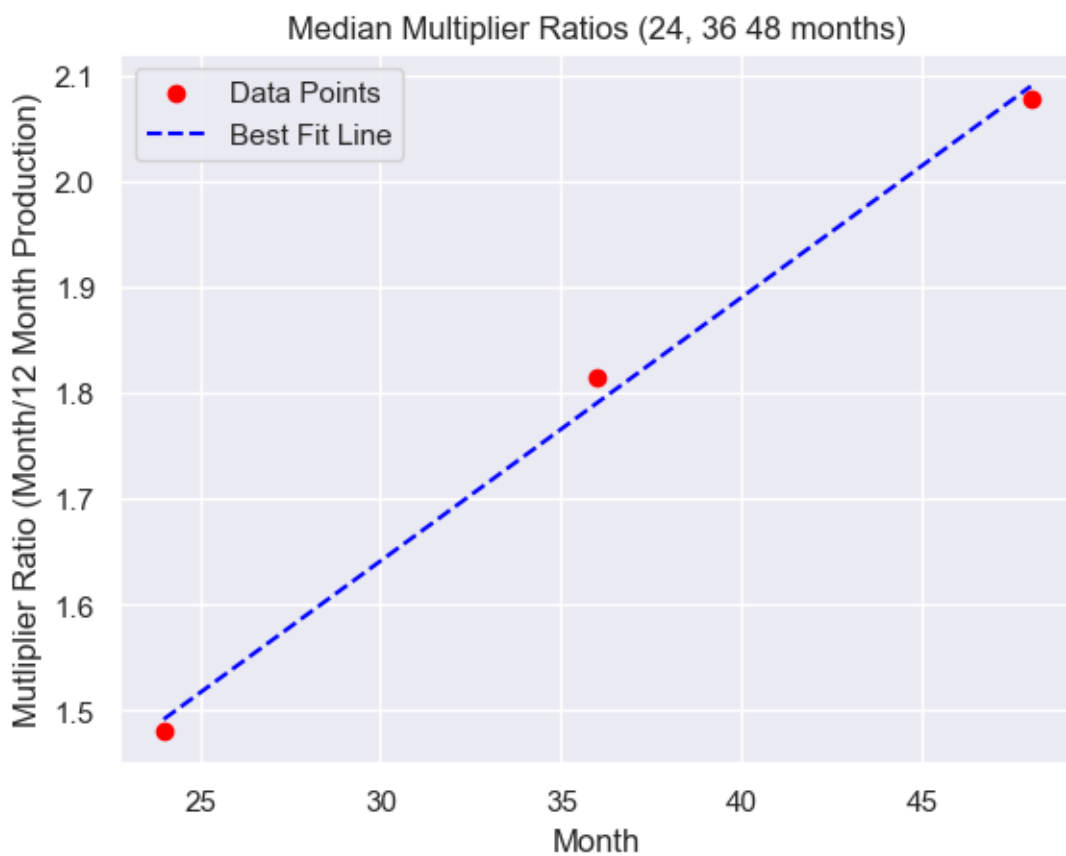


The above plots the 25, 50, 75 percentiles for the multiplier value given by 48/12 month production.

[208]:
```
median_12_48 = multiplier_12_48.median()

multiplier_12_36 = month_36_liquid / month_12_liquid[month_12_liquid.index.
↪isin(month_36_liquid.index)]
median_12_36 = multiplier_12_36.median()

multiplier_12_24 = month_24_liquid / month_12_liquid[month_12_liquid.index.
↪isin(month_24_liquid.index)]
median_12_24 = multiplier_12_24.median()
```

[209]:
```
x = np.array([24, 36, 48])
y = np.array([median_12_24, median_12_36, median_12_48])

# Fit a line (linear regression)
coeffs = np.polyfit(x, y, 1)  # 1st-degree polynomial (linear)
line = np.poly1d(coeffs)  # Create a function from the coefficients

x_fit = np.linspace(min(x), max(x), 100)
```

```
y_fit = line(x_fit)

plt.scatter(x, y, color='red', label='Data Points')
plt.plot(x_fit, y_fit, color='blue', linestyle='dashed', label='Best Fit Line')
plt.xlabel('Month')
plt.ylabel('Mutliplier Ratio (Month/12 Month Production)')
plt.title('Median Multiplier Ratios (24, 36 48 months)')
plt.legend()
plt.show()
```

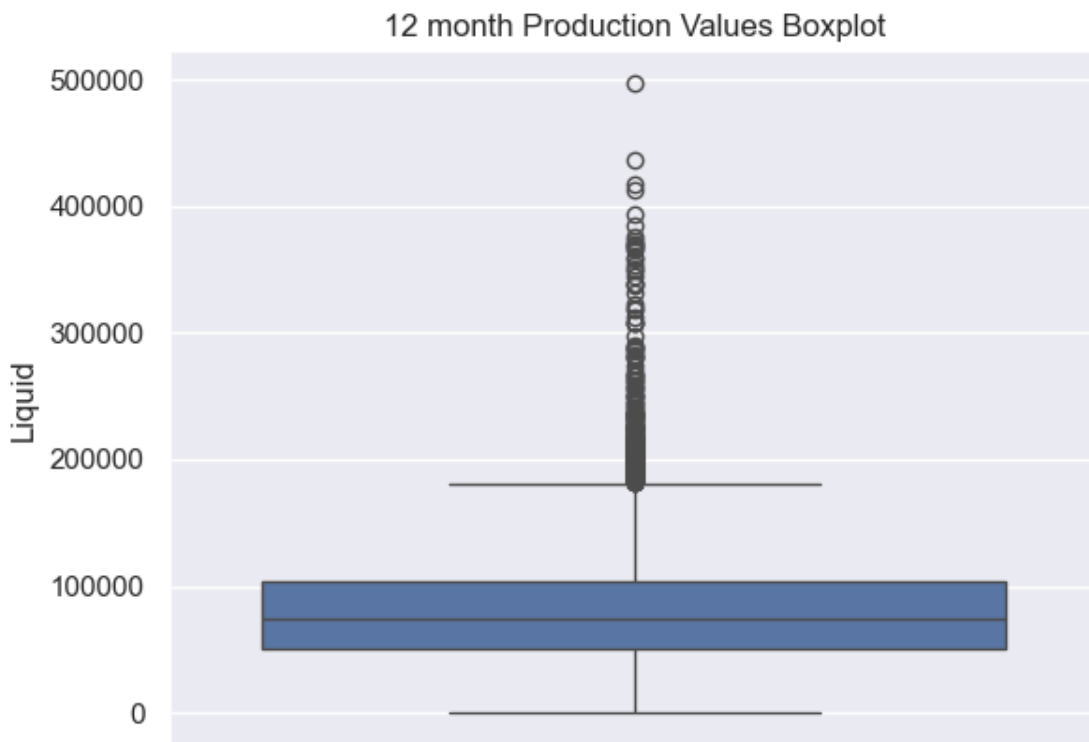


Plotting the median multiplier ratio for 24, 36, 48 months gives a linear relationship. This means that the ratio given to get production out to month n from 12 month production is linear. Once we know 12 month production, we can reasonably calculate our production out to 48 months. The given values in order are: 12–>24 Months, multiply by 1.5, 12–>36 months, multiply by 1.8, 12–>48 months, multiply by 2.1

1.7 Honing In On the 12 Month Production Values

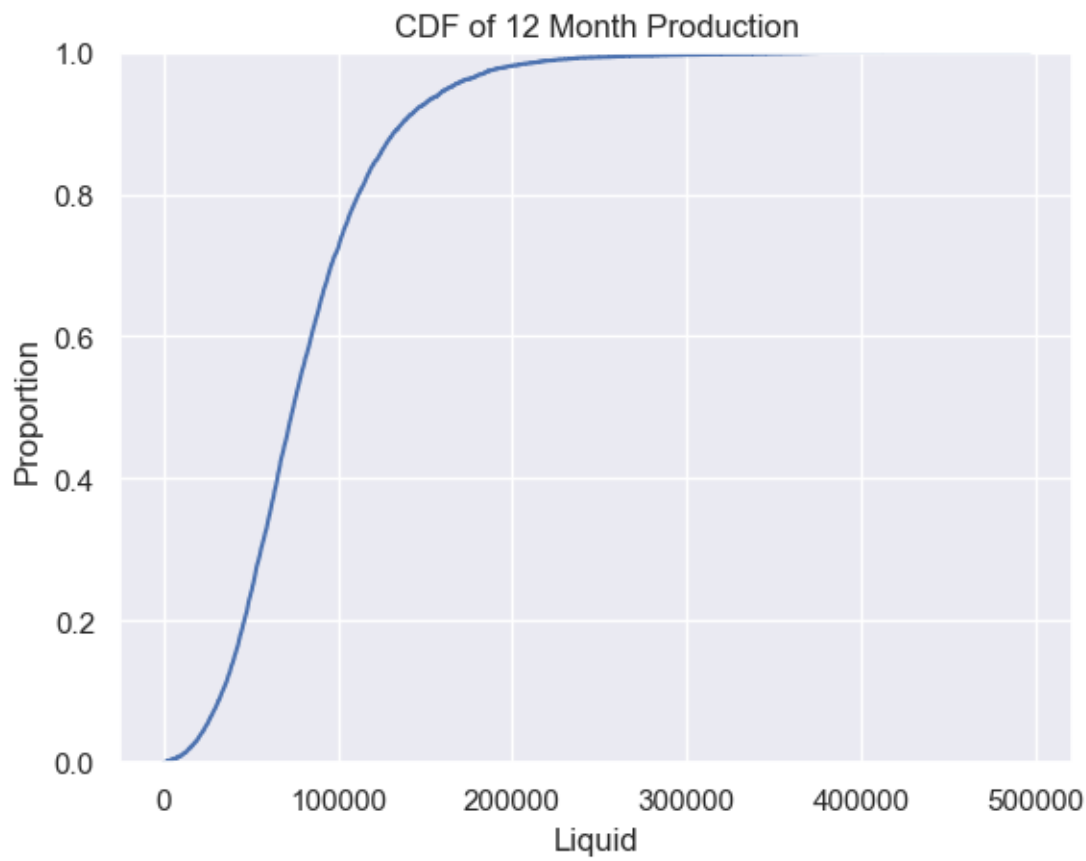
Let's observe the boxplot for the 12 month distribution.

[210]:
```
sns.boxplot(month_12_liquid)
plt.title('12 month Production Values Boxplot');
```



We observe a significant amount of outliers, associated with the right-skew behaviour observed above. Let's also take a look at the CDF.

[211]:
```
sns.ecdfplot(month_12_liquid)
plt.title('CDF of 12 Month Production');
```



The right-skew is visible as a very long tail at the end of the CDF. Notice that nearly 90% of observations follow a reasonable Gaussian distribution.

Let's take a look at the statistical information.

[212]:
```
month_12_liquid.describe().to_frame().transpose()
```

[212]:
```
          count          mean           std  min      25%      50%       75%  \
Liquid  7563.0  82061.424699  45919.794362  0.0  51255.5  74472.0  103440.5

             max
Liquid  496835.0
```

We notice that the max value significantly deviates from the 75% percentile value. Forcing our distribution into a normal via Gaussian Anamorphosis may distort physical meaning, since it is often typical to observe a right-skew in production. Dropping these values may introduce bias. We also observe that Liquid has a min of 0, meaning there are wells which produced nothing over the 12 months. These need to be dropped. First, let's drop these wells.

[213]:
```
cases_prod_is_0 = sum(month_12_liquid == 0)
```

```
print(f'{cases_prod_is_0} well(s) have 0 production over the span of 12 months.
↪')
```

```
1 well(s) have 0 production over the span of 12 months.
```

Luckily for us, only one well has 0 production. Let's drop it now.

[214]:
```
month_12_liquid = month_12_liquid.loc[month_12_liquid != 0]
```

Next, let's identify the high outliers and try to better understand them.

[215]:
```
Q1 = month_12_liquid.quantile(0.25)
Q3 = month_12_liquid.quantile(0.75)
IQR = Q3 - Q1

# Define outlier thresholds
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

outliers = month_12_liquid[month_12_liquid > upper_bound]
```

[216]:
```
prop_upper_outliers = len(outliers)*100 / len(month_12_liquid)
print(f'{prop_upper_outliers:.2f}% of the total data are upper outliers.')
```

```
3.16% of the total data are upper outliers.
```

We need to merge the outliers with the completion dataframe in order to acquire their latitude and longitude positions so we may plot them. This requires first cleaning the completion dataframe to ensure we only keep data we will actually have.

1.8 Cleaning the Completion Data

The completion dataframe contains missing values and non-sensical values that need to be addressed. First define a list of features of interest to reduce the amount of columns and work.

[217]:
```
comp_features_interest = ['API', 'Operator Name', 'Current Operator Name',␣
↪'Field Name', 'County Name', 'Final Status',
                          'Depth True Vertical', 'Formation at TD Name', 'Date␣
↪Completion', 'Surface Latitude', 'Surface Longitude',
                          'BH Latitude', 'BH Longitude', 'Lateral Length', 'Num␣
↪Frac Stages', 'Num Treatment Records', 'Total Proppant (lbs)',
                          'Total Fluid (gals)', 'Proppant - Sand (lbs)',␣
↪'Proppant - Sand, Ceramic (lbs)', 'Acid (Gals)', 'Year', 'Operator Change',
                          'Permit Date', 'Lease Name']
```

[218]:
```
comp_df = comp_df[comp_features_interest]
```

Let's visualize the completeness of the data.

[219]:
```
comp_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9066 entries, 0 to 9065
Data columns (total 25 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   API                             9066 non-null   int64
 1   Operator Name                   9066 non-null   object
 2   Current Operator Name           9066 non-null   object
 3   Field Name                      9066 non-null   object
 4   County Name                     9066 non-null   object
 5   Final Status                    9066 non-null   object
 6   Depth True Vertical             9063 non-null   float64
 7   Formation at TD Name            9066 non-null   object
 8   Date Completion                 9066 non-null   datetime64[ns]
 9   Surface Latitude                9066 non-null   float64
 10  Surface Longitude               9066 non-null   float64
 11  BH Latitude                     9062 non-null   float64
 12  BH Longitude                    9062 non-null   float64
 13  Lateral Length                  9062 non-null   float64
 14  Num Frac Stages                 9066 non-null   int64
 15  Num Treatment Records           9066 non-null   int64
 16  Total Proppant (lbs)            9066 non-null   int64
 17  Total Fluid (gals)              9066 non-null   float64
 18  Proppant - Sand (lbs)           9066 non-null   int64
 19  Proppant - Sand, Ceramic (lbs)  9066 non-null   int64
 20  Acid (Gals)                     9066 non-null   float64
 21  Year                            9066 non-null   int32
 22  Operator Change                 9066 non-null   int32
 23  Permit Date                     9066 non-null   object
 24  Lease Name                      9066 non-null   object
dtypes: datetime64[ns](1), float64(8), int32(2), int64(6), object(8)
memory usage: 1.7+ MB
```

Only a few values are missing, so we just drop them.

[220]:
```
comp_df = comp_df.dropna()
```

Let's visualize the statistical distribution of features.

[221]:
```
comp_df.describe().transpose()
```

[221]:
```
                                 count                           mean  \
API                             9059.0          32611372830562.207031
Depth True Vertical             9059.0                   10312.414726
Date Completion                   9059  2012-10-21 09:03:00.019869952
Surface Latitude                9059.0                      47.997869
Surface Longitude               9059.0                    -103.068465
BH Latitude                     9059.0                      47.996898
```

BH Longitude	9059.0	-103.067852
Lateral Length	9059.0	8644.592891
Num Frac Stages	9059.0	21.899768
Num Treatment Records	9059.0	1.823049
Total Proppant (lbs)	9059.0	2972523.768186
Total Fluid (gals)	9059.0	2564575.230666
Proppant - Sand (lbs)	9059.0	1941429.778121
Proppant - Sand, Ceramic (lbs)	9059.0	720613.613644
Acid (Gals)	9059.0	2893.032542
Year	9059.0	2012.279832
Operator Change	9059.0	0.375759

	min	25% \
API	25083225440101.0	33025017330000.0
Depth True Vertical	5115.0	9988.0
Date Completion	2001-12-02 00:00:00	2011-10-21 00:00:00
Surface Latitude	46.688496	47.76021
Surface Longitude	-105.178358	-103.43036
BH Latitude	46.674791	47.760791
BH Longitude	-105.178232	-103.433023
Lateral Length	8.0	8800.0
Num Frac Stages	0.0	8.0
Num Treatment Records	0.0	1.0
Total Proppant (lbs)	0.0	1804100.0
Total Fluid (gals)	0.0	1241352.0
Proppant - Sand (lbs)	0.0	0.0
Proppant - Sand, Ceramic (lbs)	0.0	0.0
Acid (Gals)	0.0	0.0
Year	2001.0	2011.0
Operator Change	0.0	0.0

	50%	75% \
API	33053049780100.0	33061024455000.0
Depth True Vertical	10513.0	10871.0
Date Completion	2012-12-20 00:00:00	2014-01-25 00:00:00
Surface Latitude	47.994007	48.242097
Surface Longitude	-102.965701	-102.622939
BH Latitude	47.993786	48.241452
BH Longitude	-102.966573	-102.619348
Lateral Length	9448.0	9761.0
Num Frac Stages	27.0	31.0
Num Treatment Records	1.0	2.0
Total Proppant (lbs)	2772000.0	3699000.0
Total Fluid (gals)	2139853.0	3098571.0
Proppant - Sand (lbs)	1664000.0	2913047.5
Proppant - Sand, Ceramic (lbs)	0.0	0.0
Acid (Gals)	0.0	0.0

```
Year                                        2012.0                2014.0
Operator Change                                0.0                   1.0

                                                max                   std
API                                 33105042410100.0  1831239392574.302246
Depth True Vertical                          11854.0            802.957091
Date Completion                  2022-10-04 00:00:00                   NaN
Surface Latitude                           48.988025              0.429279
Surface Longitude                        -102.082853              0.568671
BH Latitude                                 48.99897                  0.43
BH Longitude                             -102.083315              0.569066
Lateral Length                               15601.0           2098.582331
Num Frac Stages                                 97.0             14.281918
Num Treatment Records                          487.0              6.905038
Total Proppant (lbs)                        35028200.0        2272577.528712
Total Fluid (gals)                         103383976.0        2537533.405369
Proppant - Sand (lbs)                       35028200.0        2475828.649635
Proppant - Sand, Ceramic (lbs)              19258000.0        1500223.304435
Acid (Gals)                                  2054346.0          32109.391159
Year                                            2022.0              1.506284
Operator Change                                    1.0              0.484345
```

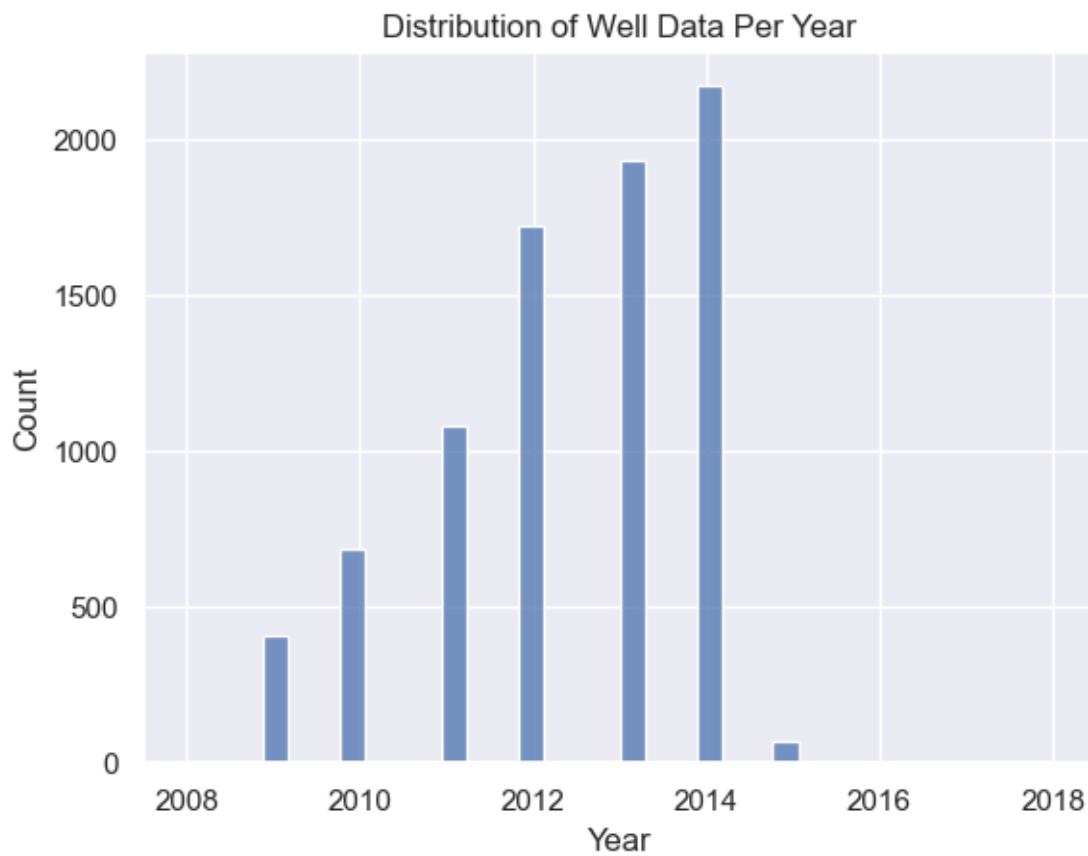
There are a few things that don't make sense: - Minimum lateral length of 8 (not realistic) - Num Treatment Records is 0 for some wells, which is impossible for horizontal wells - Some wells have no proppant or fluid which is also impossible for horizontal wells

Note that the column Num Frac Stages contains 0 only because the majority of values are missing. This is not an error, just a limit of the dataset. We could impute the number of frac stages, but for now we proceed by removing the non-sensical values.

[222]:
```
comp_df = comp_df.loc[comp_df['Total Proppant (lbs)'] != 0.0]
comp_df = comp_df.loc[comp_df['Total Fluid (gals)'] != 0.0]
comp_df = comp_df.loc[comp_df['Lateral Length'] > 3000]
```

We keep only wells that have a lateral length greater than 2000 ft. Let's visualize the amount of well data available per year.

[223]:
```
sns.histplot(comp_df['Year'])
plt.title('Distribution of Well Data Per Year');
```



We see that there is an unequal distribution of data between the years, and most of the data lies between the years 2010-2014. Let's remove all data prior to 2009 and above 2014.

[224]:
```
lower_years_comp_df = comp_df[comp_df['Year'] <= 2008]
```

[225]:
```
comp_df = comp_df[(comp_df['Year'] > 2008) & (comp_df['Year'] < 2015)]
```

We now have a cleaned completion dataframe.

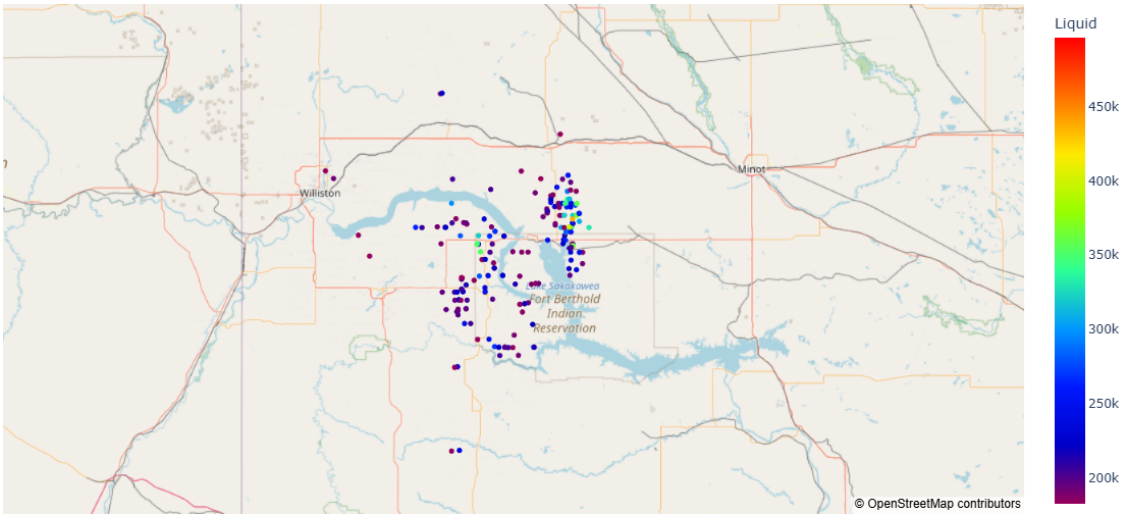
1.9 Back to Liquid Outlier Analysis

With our completion dataframe cleaned, we are now ready to tackle the upper outliers in the liquid data.

[226]:
```
outliers_prod_comp_df = pd.merge(comp_df, outliers.reset_index(), on='API')
```

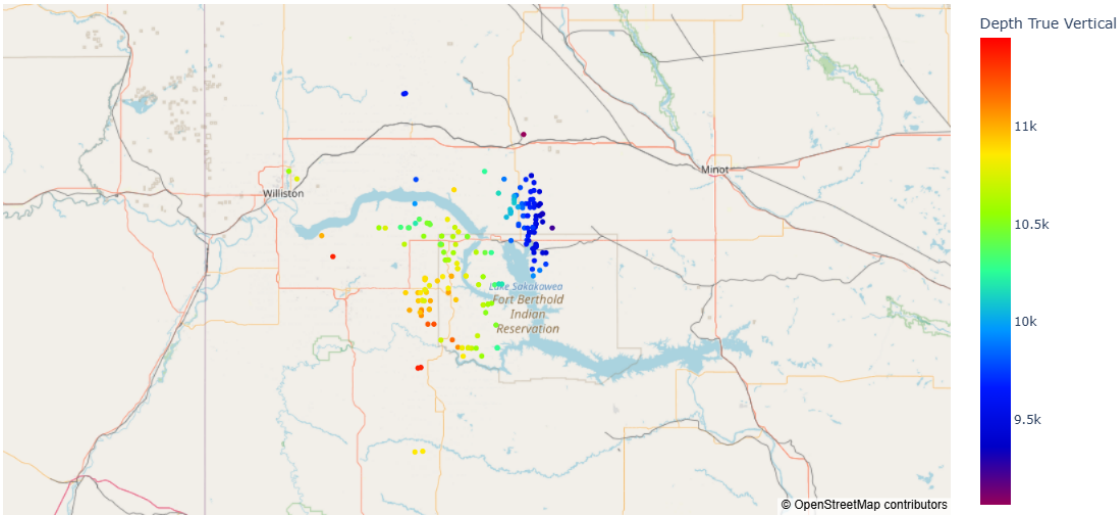
[227]:
```
target_hue = 'Liquid'

df_visualize_comp_2 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_2['API'] = outliers_prod_comp_df['API']
df_visualize_comp_2['X'] = outliers_prod_comp_df['Surface Latitude']
```

```
df_visualize_comp_2['Y'] = outliers_prod_comp_df['Surface Longitude']
df_visualize_comp_2[target_hue] = outliers_prod_comp_df[target_hue]
create_location_map(df_visualize_comp_2, 'X', 'Y', target_hue, 7)
```



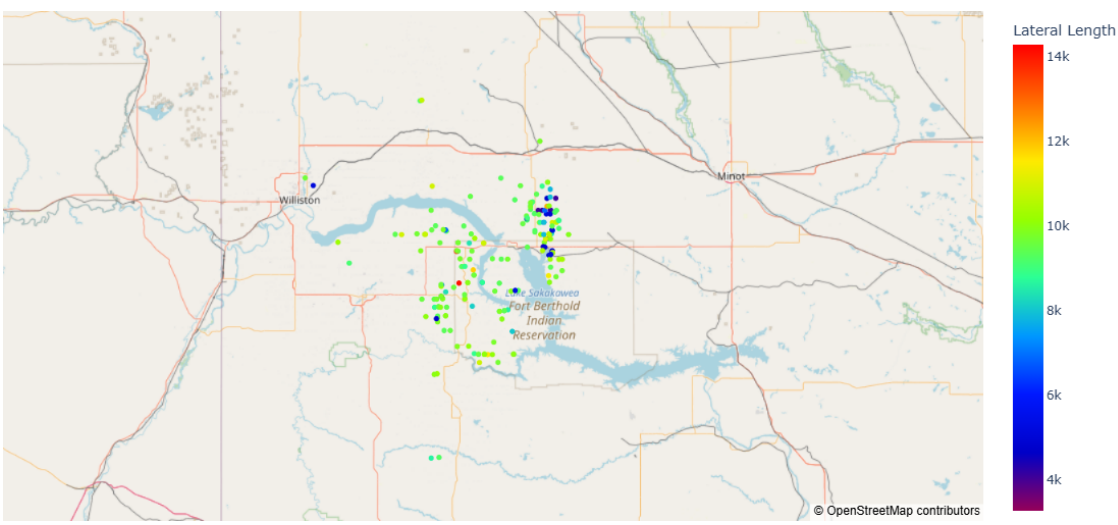
It seems the majority of the wells are in the 200-250 range, while there are smaller clusters of very high producing wells. Let's investigate further. We will plot the outliers over several features including: - Depth True Vertical - Lateral Length - Operator - Year - Formation at TD - Total Proppant (gals) - Lease Name

[228]:
```
target_hue = 'Depth True Vertical'

df_visualize_comp_2 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_2['API'] = outliers_prod_comp_df['API']
df_visualize_comp_2['X'] = outliers_prod_comp_df['Surface Latitude']
df_visualize_comp_2['Y'] = outliers_prod_comp_df['Surface Longitude']
df_visualize_comp_2[target_hue] = outliers_prod_comp_df[target_hue]
create_location_map(df_visualize_comp_2, 'X', 'Y', target_hue, 7)
```



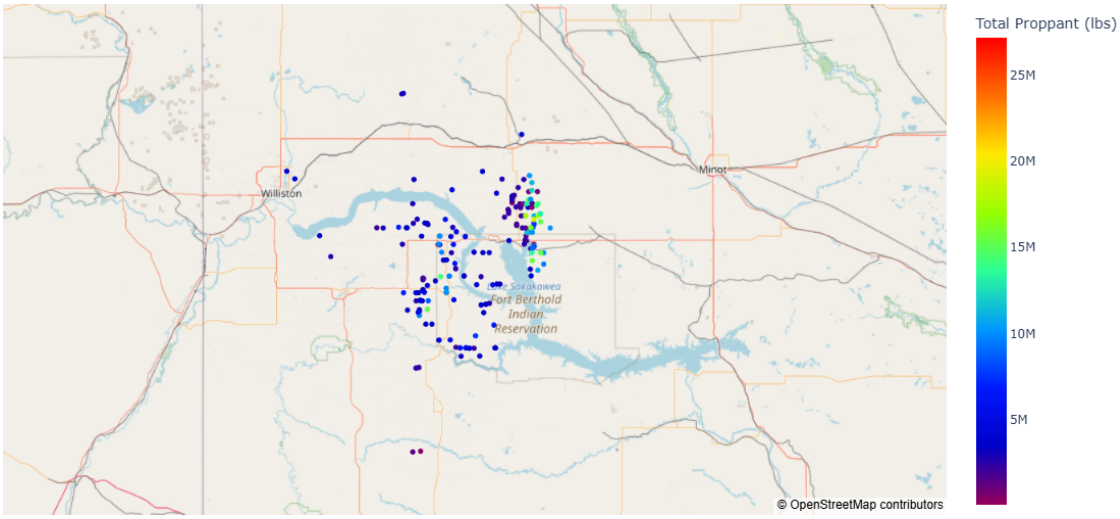
The right cluster of larger production values have lower depths.

[229]:
```
target_hue = 'Lateral Length'

df_visualize_comp_2 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_2['API'] = outliers_prod_comp_df['API']
df_visualize_comp_2['X'] = outliers_prod_comp_df['Surface Latitude']
df_visualize_comp_2['Y'] = outliers_prod_comp_df['Surface Longitude']
df_visualize_comp_2[target_hue] = outliers_prod_comp_df[target_hue]
create_location_map(df_visualize_comp_2, 'X', 'Y', target_hue, 7)
```



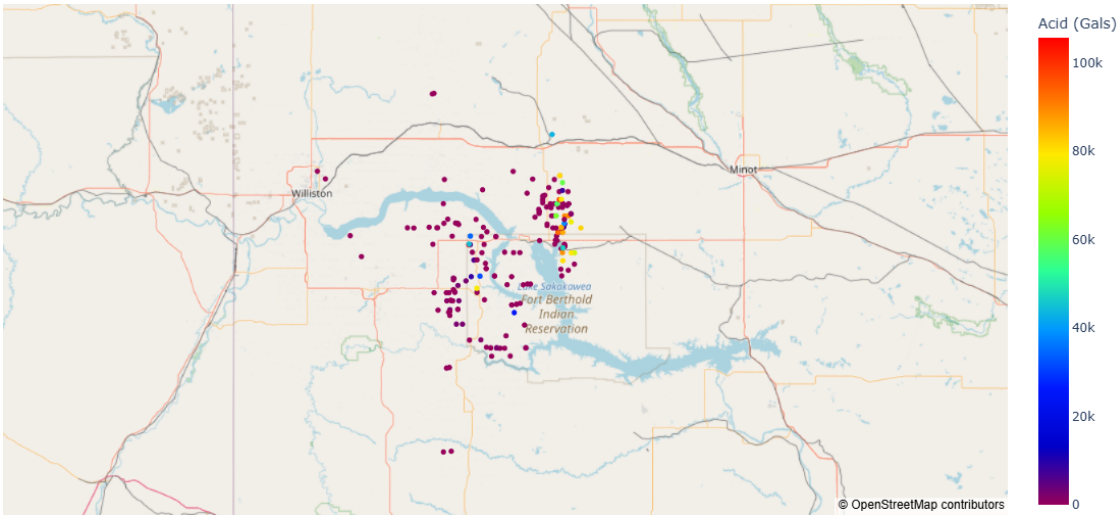
Shorter lateral lengths in the right cluster.

[230]:
```
target_hue = 'Total Proppant (lbs)'

df_visualize_comp_2 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_2['API'] = outliers_prod_comp_df['API']
df_visualize_comp_2['X'] = outliers_prod_comp_df['Surface Latitude']
df_visualize_comp_2['Y'] = outliers_prod_comp_df['Surface Longitude']
df_visualize_comp_2[target_hue] = outliers_prod_comp_df[target_hue]
create_location_map(df_visualize_comp_2, 'X', 'Y', target_hue, 7)
```



Total proppant looks convincing as a differentiator between the higher and lower producing wells. We can see if this proppant has anything to do with acid injections.

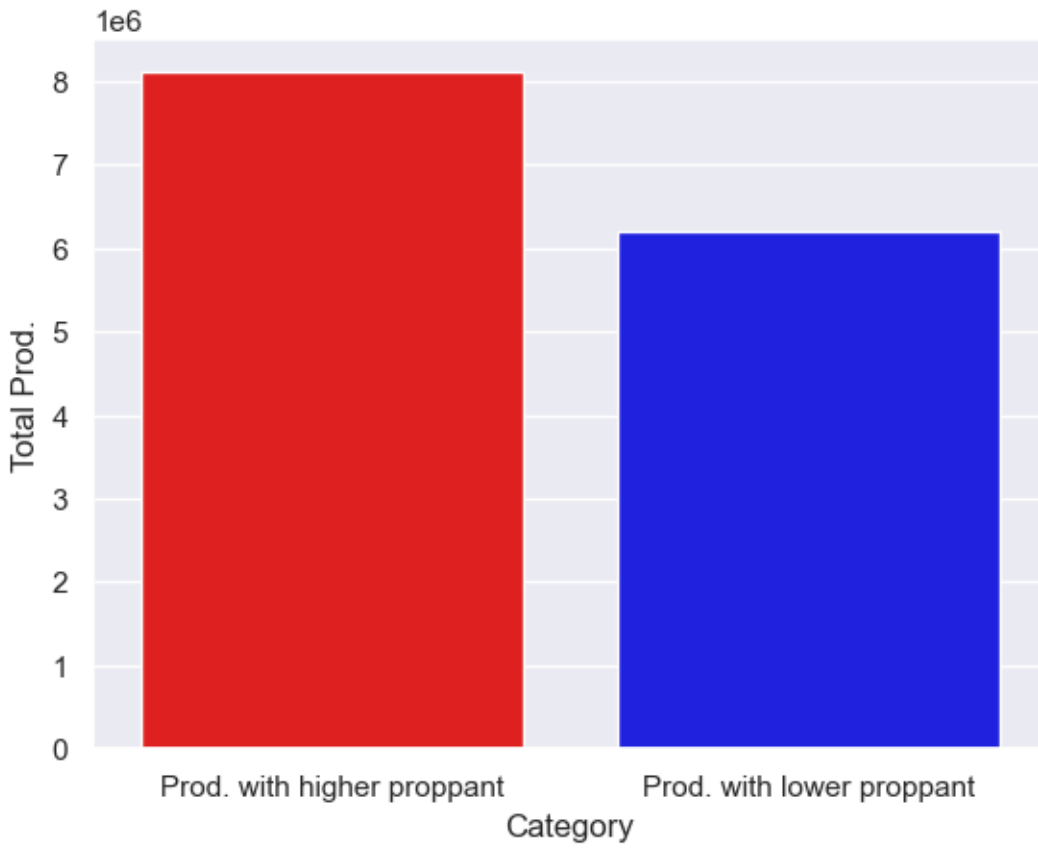
[231]:
```
target_hue = 'Acid (Gals)'

df_visualize_comp_2 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_2['API'] = outliers_prod_comp_df['API']
df_visualize_comp_2['X'] = outliers_prod_comp_df['Surface Latitude']
df_visualize_comp_2['Y'] = outliers_prod_comp_df['Surface Longitude']
df_visualize_comp_2[target_hue] = outliers_prod_comp_df[target_hue]
create_location_map(df_visualize_comp_2, 'X', 'Y', target_hue, 7)
```



Amount of acid injected is similarly present in the regions of large production, and seems to follow total proppant similarly. Let's determine the difference in production between wells with higher proppants and wells with lower proppants.

[232]:
```
v = 30
outliers_upper_above_b_prop =␣
↪outliers_prod_comp_df[outliers_prod_comp_df['Total Proppant (lbs)'] >=␣
↪5000000]
outliers_upper_below_b_prop =␣
↪outliers_prod_comp_df[outliers_prod_comp_df['Total Proppant (lbs)'] <␣
↪5000000]
sample_outliers_upper_above_b_prop = outliers_upper_above_b_prop.sample(v)
sample_outliers_upper_below_b_prop = outliers_upper_below_b_prop.sample(v)
sample_outliers_upper_above_b_prop_sum =␣
↪sample_outliers_upper_above_b_prop['Liquid'].sum()
sample_outliers_upper_below_b_prop_sum =␣
↪sample_outliers_upper_below_b_prop['Liquid'].sum()
```

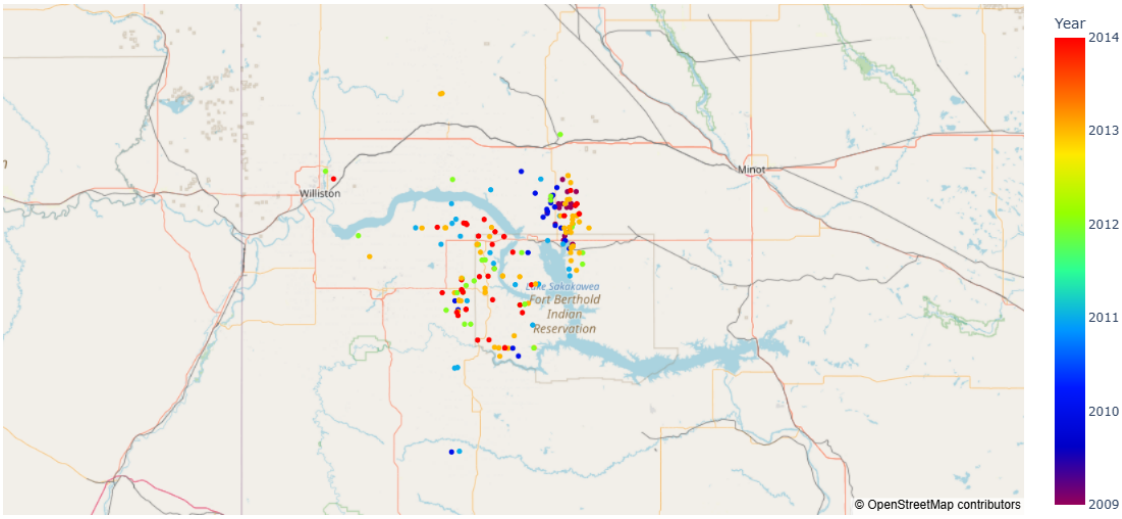
[233]:
```
categories = ['Prod. with higher proppant', 'Prod. with lower proppant']
values = [sample_outliers_upper_above_b_prop_sum,␣
↪sample_outliers_upper_below_b_prop_sum]
palette = ['red', 'blue']
create_bar_plot(categories, values, palette, 'Category', 'Total Prod.');
```



It is pretty clear with this that the higher production outliers are a consequence of greater stimulation of the region via proppant injections. Let's see the year in which most of these were done.

[234]:
```
outliers_prod_comp_df['Date Completion'] = pd.
 ↪to_datetime(outliers_prod_comp_df['Date Completion'])
outliers_prod_comp_df['Year'] = outliers_prod_comp_df['Date Completion'].dt.year
```

[235]:
```
target_hue = 'Year'

df_visualize_comp_2 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_2['API'] = outliers_prod_comp_df['API']
df_visualize_comp_2['X'] = outliers_prod_comp_df['Surface Latitude']
df_visualize_comp_2['Y'] = outliers_prod_comp_df['Surface Longitude']
df_visualize_comp_2[target_hue] = outliers_prod_comp_df[target_hue]
create_location_map(df_visualize_comp_2, 'X', 'Y', target_hue, 7)
```



Let's take a look at the operators behind these higher production wells.

[236]:
```
outliers_prod_comp_df[outliers_prod_comp_df['Liquid'] > 250000]['Operator␣
↪Name'].value_counts()
```

[236]:
```
Operator Name
EOG RESOURCES INC                 38
WHITING OIL & GAS CORP             3
HRC OPERATING LLC                  3
ENERPLUS RESOURCES (USA) CORP      3
MARATHON OIL CO                    1
HESS CORP                          1
HELIS OIL & GAS CO LLC             1
CONTINENTAL RESOURCES INC          1
PETRO-HUNT LLC                     1
Name: count, dtype: int64
```

Let's also check the entire set of outliers.

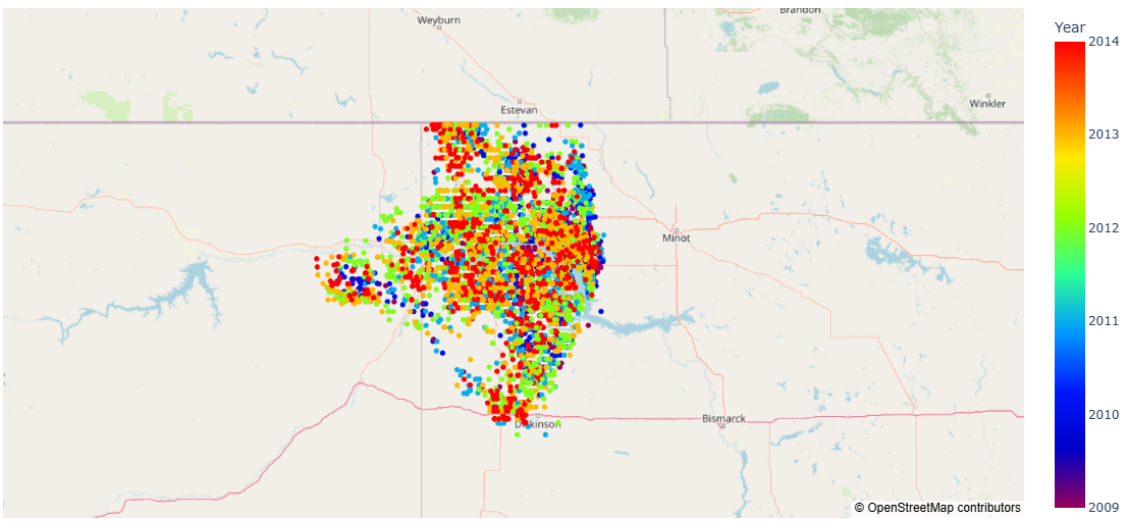
[237]:
```
outliers_prod_comp_df['Operator Name'].value_counts()[:10]
```

[237]:
```
Operator Name
EOG RESOURCES INC                 70
WHITING OIL & GAS CORP            28
HRC OPERATING LLC                 15
WPX ENERGY WILLISTON LLC          11
QEP ENERGY CO                     10
BURLINGTON RESOURCES O&G CO LP    10
HELIS OIL & GAS CO LLC             8
ENERPLUS RESOURCES (USA) CORP      8
CONTINENTAL RESOURCES INC          7
```

```
MARATHON OIL CO                    7
Name: count, dtype: int64
```

Interesting! The majority of these higher production wells are likely exploration wells drilled by EOG that were very successful. As a side check, let's visualize the years in which all wells were produced.

[238]:
```
target_hue = 'Year'

df_visualize_comp_2 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_2['API'] = comp_df['API']
df_visualize_comp_2['X'] = comp_df['Surface Latitude']
df_visualize_comp_2['Y'] = comp_df['Surface Longitude']
df_visualize_comp_2[target_hue] = comp_df[target_hue]
create_location_map(df_visualize_comp_2, 'X', 'Y', target_hue, 6)
```



There is a seemingly random scatter of production years. We are in a range between 2009-2014.

Let's briefly check the formation being targetted and the county name.

[239]:
```
outliers_prod_comp_df['Formation at TD Name'].value_counts()
```

[239]:
```
Formation at TD Name
BAKKEN MIDDLE        158
THREE FORKS           44
BAKKEN                 5
BAKKEN MIDDLE C        3
BAKKEN LOWER           2
BAKKEN UPPER           1
THREE FORKS UPPER      1
Name: count, dtype: int64
```

[240]:
```
outliers_prod_comp_df['County Name'].value_counts()
```

[240]:
```
County Name
MOUNTRAIL    101
MCKENZIE      72
DUNN          33
WILLIAMS       5
STARK          3
Name: count, dtype: int64
```

1.9.1 Summary of Liquid Outlier Analysis

- We have found that extreme outliers have higher values that are associated with higher proppant amounts
- These extreme outliers are mainly EOG wells that were most likely exploration/discovery wells that hit big success
- Majority of the extreme outliers target the common Bakken Middle formation
- The wells are disperesed among the Mountrail, McKenzie, and Dunn counties
- The single extreme outlier is a Continental Well situated near the bottom left of the area

[241]:
```
# Come later to create a separate model for high producing wells? To be␣
↪continued...
```

1.10 Time Variance of the dataset

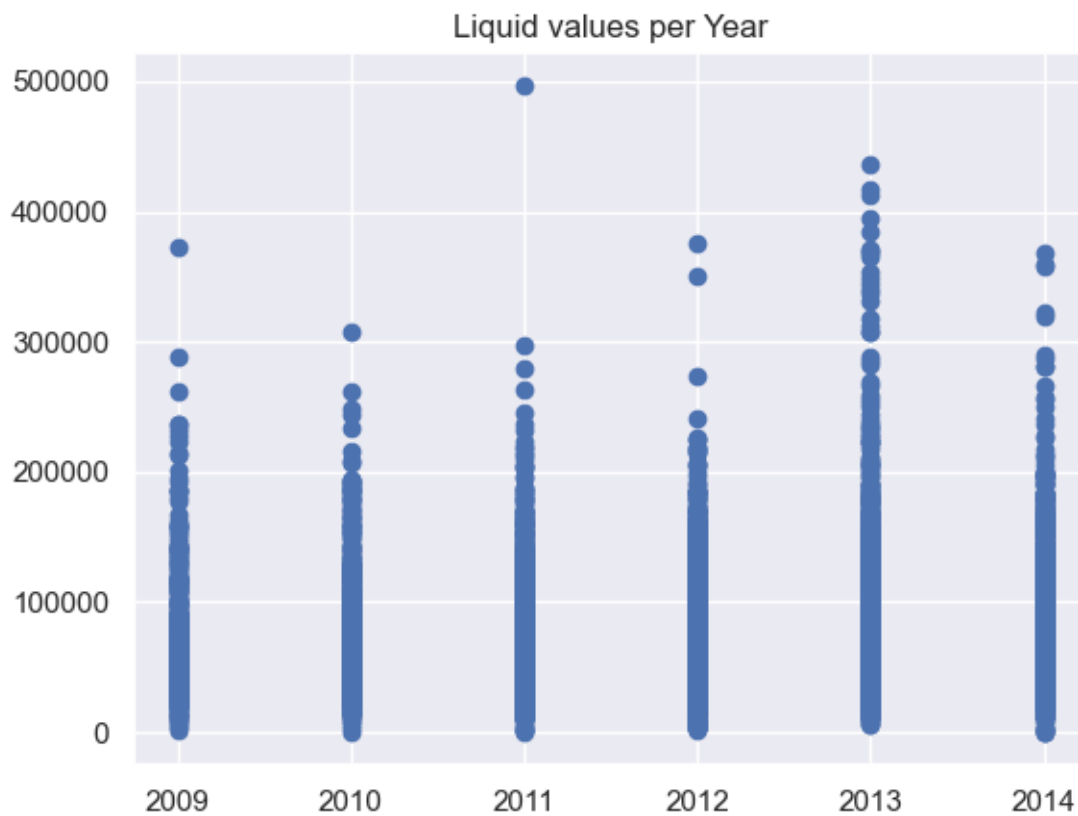
An important detail has not been brought up to this point. How does the timing of wells impact their production? This is of significance as wells produced at earlier years may be technologically less advanced, but have the benefit of drilling the formation earlier and accessing more resources. These factors are likely not negligible. If they were, we could consider the dataset time invariant, and future work would be simpler.

Three potential considerations are wrapped in time: - Earlier access to resources, and future resources depletion - Technological advancement - Seasonal trends

Let's check production over year. This is important because if production increases with year, we will need to account for that in our model.

[242]:
```
prod_comp_df = pd.merge(comp_df, month_12_liquid.reset_index(), on='API')
```

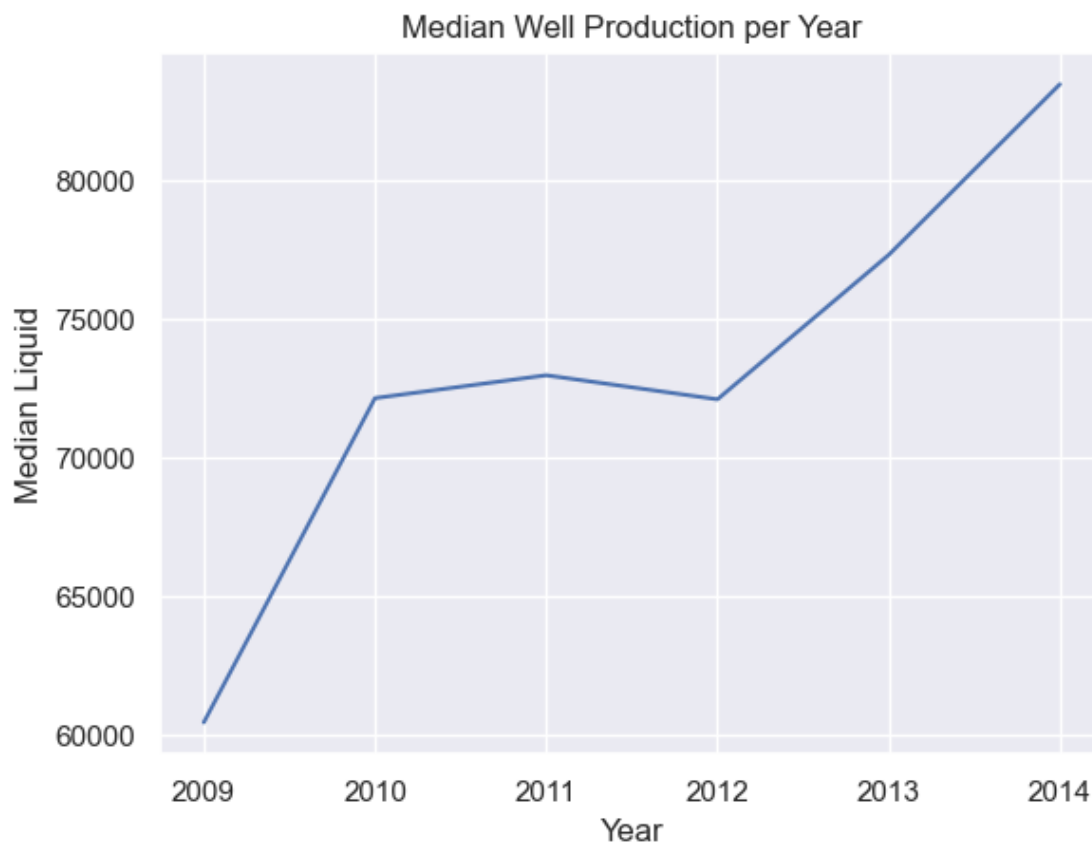
[243]:
```
plt.scatter(prod_comp_df['Year'], prod_comp_df['Liquid'])
plt.title('Liquid values per Year');
```



This gives us insight about the distribution of values over time. We see a massive value somewhere in 2011 that should be removed. Let's compare the median Liquid values for each year.

[244]:
```
median_prod_over_year = prod_comp_df.groupby('Year')['Liquid'].median().
 ↪reset_index()
mean_prod_over_year = prod_comp_df.groupby('Year')['Liquid'].mean().
 ↪reset_index()
```

[245]:
```
plt.plot(median_prod_over_year['Year'], median_prod_over_year['Liquid'])
plt.title('Median Well Production per Year')
plt.xlabel('Year')
plt.ylabel('Median Liquid');
```



This plot shows us that the median well is producing more over time. The exact nature of this increase is not obvious. Is it due to better technology, or greater completion parameters? If other variables scale with production, then year will not need to be a model input.

Let's briefly remove the extreme outlier.

[246]:
```
prod_comp_df = prod_comp_df[prod_comp_df['Liquid'] < 490000]
```

1.10.1 Operator efficiency

By comparing the median production of various operators, we can see who was the most successful at producing oil. Note that this is susceptible to the amount of data present on each operator.

[247]:
```
operators = prod_comp_df['Operator Name'].unique()
```

[248]:
```
num_operators_to_test = 10
operators_most_wells = prod_comp_df['Operator Name'].value_counts()[:
 ↪num_operators_to_test].reset_index()['Operator Name']
prod_comp_df['Operator Name'].value_counts()[:num_operators_to_test]
```

[248]:
```
Operator Name
CONTINENTAL RESOURCES INC            797
WHITING OIL & GAS CORP               631
HESS CORP                            453
EOG RESOURCES INC                    419
XTO ENERGY INC                       378
OASIS PETROLEUM NORTH AMERICA LLC    345
MARATHON OIL CO                      303
BURLINGTON RESOURCES O&G CO LP       300
SLAWSON EXPLORATION CO INC           218
KODIAK OIL & GAS USA INC             209
Name: count, dtype: int64
```

Above are the 10 producers with the most wells in the Bakken between 2010-2014. Let's actually determine each operators median production.

[249]:
```
operator_median_prod = prod_comp_df[prod_comp_df['Operator Name'].
 ↪isin(operators_most_wells)].groupby('Operator Name')['Liquid'].median().
 ↪reset_index()
operator_median_prod.sort_values(by='Liquid', ascending=False)[:10]
```

[249]:

	Operator Name	Liquid
0	BURLINGTON RESOURCES O&G CO LP	90234.0
4	KODIAK OIL & GAS USA INC	89534.0
2	EOG RESOURCES INC	86426.0
7	SLAWSON EXPLORATION CO INC	81269.0
3	HESS CORP	76270.0
8	WHITING OIL & GAS CORP	71862.0
9	XTO ENERGY INC	71171.0
5	MARATHON OIL CO	71107.0
1	CONTINENTAL RESOURCES INC	66412.0
6	OASIS PETROLEUM NORTH AMERICA LLC	66323.0

The producers above have been placed in order of the greatest to lowest median well production, out of the list of producers with the most wells in the Bakken. It might be of interest now to add a separate column that considers these operators as having an impact on production.

[250]:
```
# To be continued...
```

2 Baseline Model

We now have a dataframe with our liquid values and our features. Let's build a baseline model.

[251]:
```
prod_comp_df = prod_comp_df[prod_comp_df['Num Frac Stages'] > 2]
```

[252]:
```
prod_comp_df = prod_comp_df[prod_comp_df['Liquid'] > 1000]
```

[253]:
```
prod_comp_df['Stages / ft'] = prod_comp_df['Num Frac Stages'] /␣
↪prod_comp_df['Lateral Length']
prod_comp_df['Proppant / ft'] = prod_comp_df['Total Proppant (lbs)'] /␣
↪prod_comp_df['Lateral Length']
```

[254]:
```
pc_relations_df = pd.read_csv('AMPP1_PCREL')
prod_comp_df = pd.merge(prod_comp_df, pc_relations_df, on='API')
```

[255]:
```
target = ['Liquid']
features = ['Depth True Vertical', 'Lateral Length', 'Proppant / ft', 'Stages /␣
↪ft', 'Total Gas Before', 'Number of Neighbouring Wells Prior to Prod.']
features_df = prod_comp_df[target + features]
```

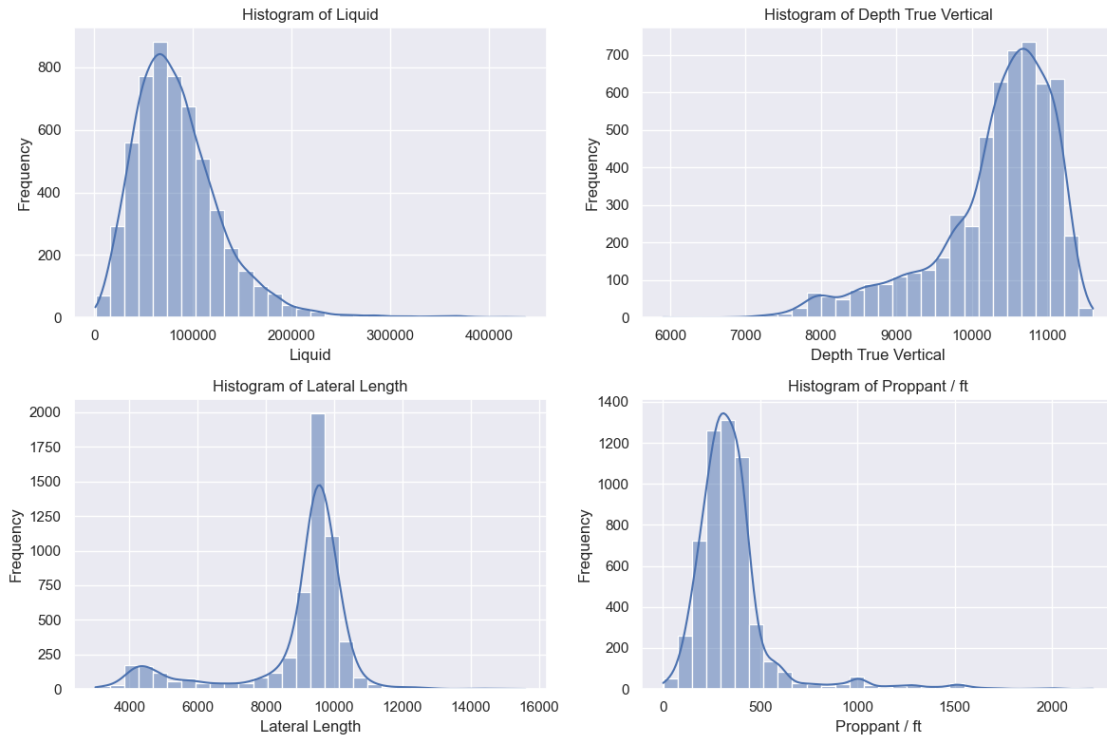
Let's standardize and normalize our dataframe.

[256]:
```
scaler_standard = StandardScaler()
scaler_minmax = MinMaxScaler()
```

2.0.1 Univariate Distributions

[257]:
```
series_list = [features_df[feat] for feat in (target + features)]
```

[258]:
```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



[259]:
```
X = features_df[features]
y = features_df[target]
```

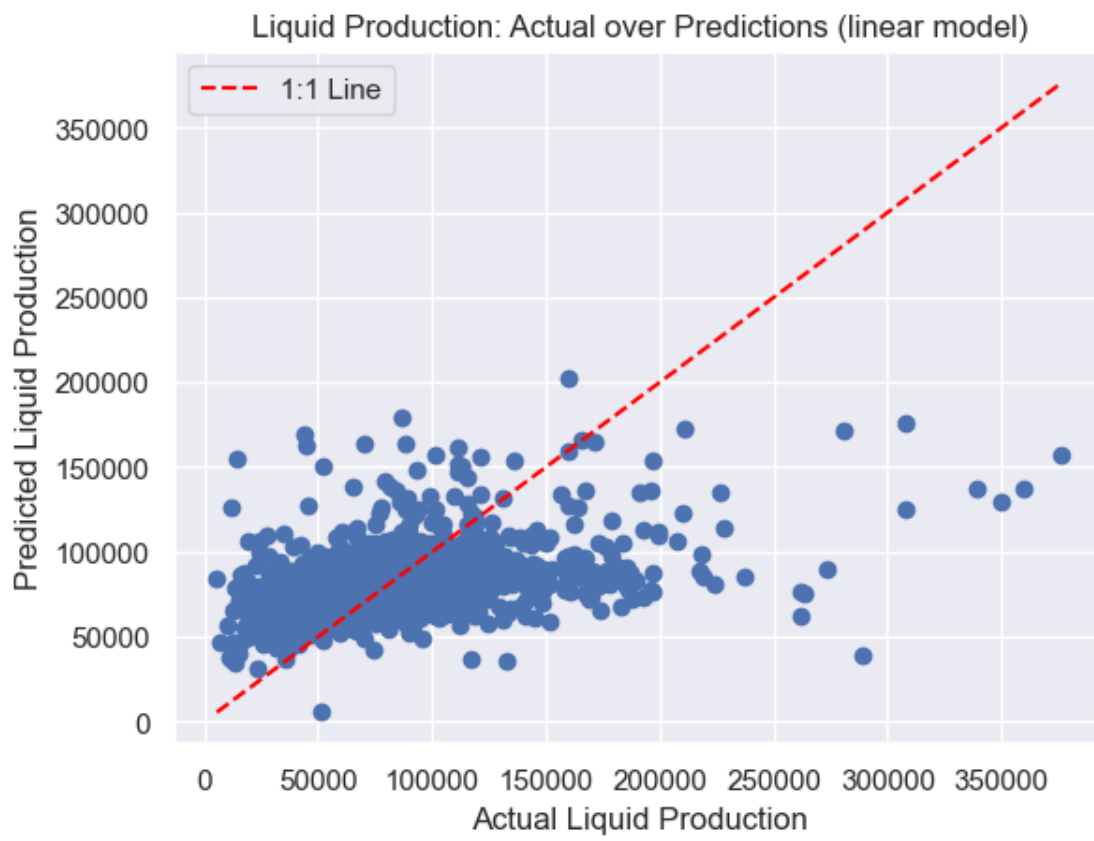
[260]:
```
y_test, y_pred = scale_train_fit_eval_model(X, y, LinearRegression())
```

```
RMSE: 40699.73634
MRMSE: 26204.982576590348
Median Absolute Percent Error (MdAPE): 31.27%
```

[261]:
```
y_test, y_pred = train_fit_eval_model(X, y, LinearRegression())
```

```
RMSE: 42827.38549
MRMSE: 23698.530191239977
Median Absolute Percent Error (MdAPE): 29.37%
```

[262]:
```
plt.scatter(y_test, y_pred)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],␣
 ↪color='red', linestyle='--', label="1:1 Line")  # Diagonal line
plt.title('Liquid Production: Actual over Predictions (linear model)')
plt.xlabel('Actual Liquid Production')
plt.ylabel('Predicted Liquid Production')
plt.legend()
plt.show();
```

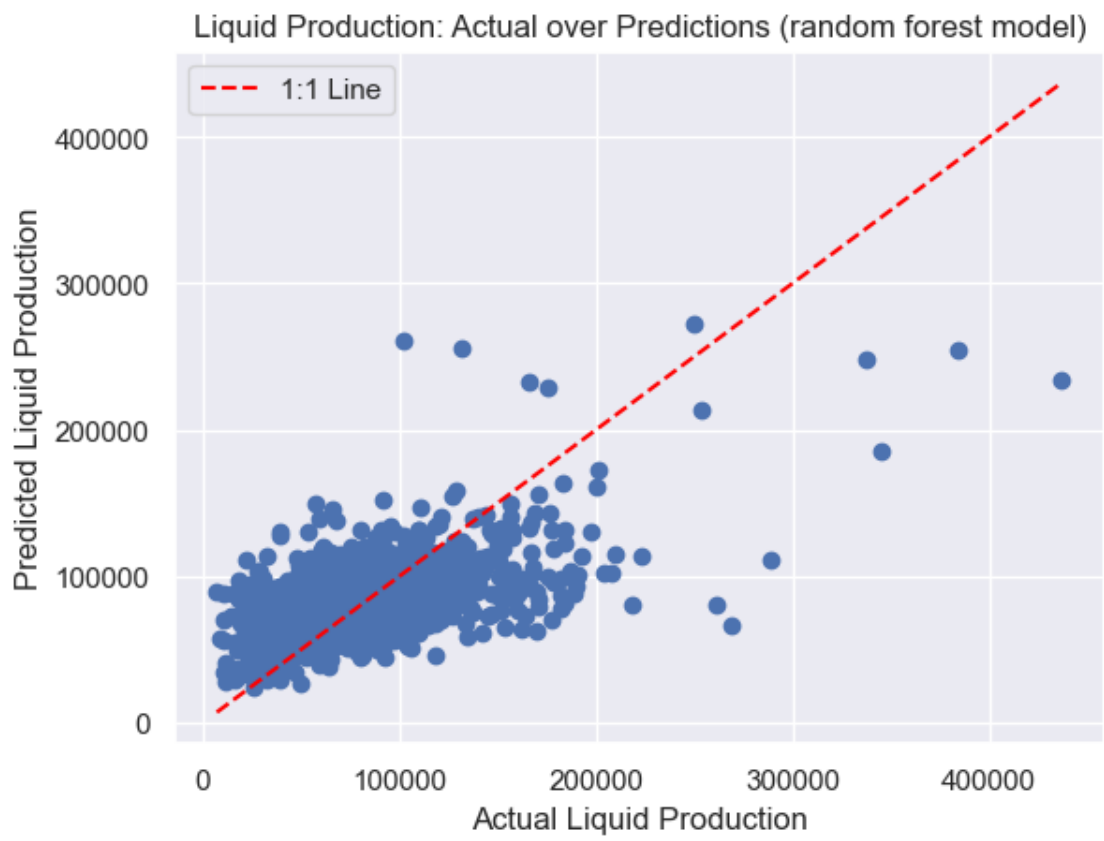


Let's test a decision tree model.

[263]:
```
y_test, y_pred = scale_train_fit_eval_model(X, y,␣
↪RandomForestRegressor(n_estimators=100, random_state=42))
```

```
RMSE: 35588.69456
MRMSE: 20681.9062011375
Median Absolute Percent Error (MdAPE): 26.11%
```

[264]:
```
plt.scatter(y_test, y_pred)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red',␣
↪linestyle='--', label="1:1 Line")  # Diagonal line
plt.title('Liquid Production: Actual over Predictions (random forest model)')
plt.xlabel('Actual Liquid Production')
plt.ylabel('Predicted Liquid Production')
plt.legend()
plt.show();
```



2.1 Buidling the CatBoost model

Define data. Run the cells below for various segmentation tasks (optional).

[265]:
```
# prod_comp_df = prod_comp_df[prod_comp_df['Year'] == 2011]
```

[266]:
```
# sns.histplot(prod_comp_df[prod_comp_df['Lateral Length'] > 7000]['Lateral␣
↪Length']);
```

[267]:
```
# prod_comp_df = prod_comp_df[prod_comp_df['Lateral Length'] >= 7000]
```

[268]:
```
# len(prod_comp_df[prod_comp_df['Operator Name'].map(prod_comp_df['Operator␣
↪Name'].value_counts()) > 520]) / len(prod_comp_df)
```

[269]:
```
# prod_comp_df = prod_comp_df[prod_comp_df['Operator Name'].
↪map(prod_comp_df['Operator Name'].value_counts()) > 520]
```

[270]:
```
# prod_comp_df['Operator Name'].value_counts()
```

2.1.1 1

Index on original data to include specific columns of interest

[271]:
```
target = ['Liquid']
num_features = ['Depth True Vertical', 'Lateral Length', 'Proppant / ft',␣
↪'Stages / ft', 'Total Liquid Before',
                'Total Gas Before', 'Number of Neighbouring Wells Prior to Prod.
↪']
location_info = ['Surface Latitude', 'Surface Longitude']
cat_features = ['Operator Name', 'Formation at TD Name', 'County Name', 'Field␣
↪Name', 'Lease Name']
features_df = prod_comp_df[target + num_features + location_info + cat_features]
```

2.1.2 2

Split into training and testing data

[290]:
```
cat_test = features_df.sample(int(0.1*len(features_df)))
cat_train = features_df.drop(cat_test.index)
```

2.1.3 3

Train CatBoost model on training and total data

[327]:
```
train_pool = Pool(data=cat_train.drop(columns=target+location_info),␣
↪label=cat_train[target], cat_features=cat_features)
monotonic_constraints = [0, 1, 1, 1]
model = CatBoostRegressor(iterations=400, learning_rate=0.1, depth=5,␣
↪verbose=0, monotone_constraints=monotonic_constraints)
model.fit(train_pool)
```

[327]:
```
<catboost.core.CatBoostRegressor at 0x22fc2089790>
```

[328]:
```
predictions = model.predict(cat_train.drop(columns=target+location_info))
y_pred = model.predict(cat_test.drop(columns=target+location_info))

y_train = cat_train['Liquid']
y_train_pred = predictions
eval_model_simple(y_train, y_train_pred)
print("-----------------------------------")
y_test = cat_test['Liquid']
eval_model_simple(y_test, y_pred)
```

```
Median Absolute Percent Error (MdAPE): 17.05%
-----------------------------------
Median Absolute Percent Error (MdAPE): 22.93%
```

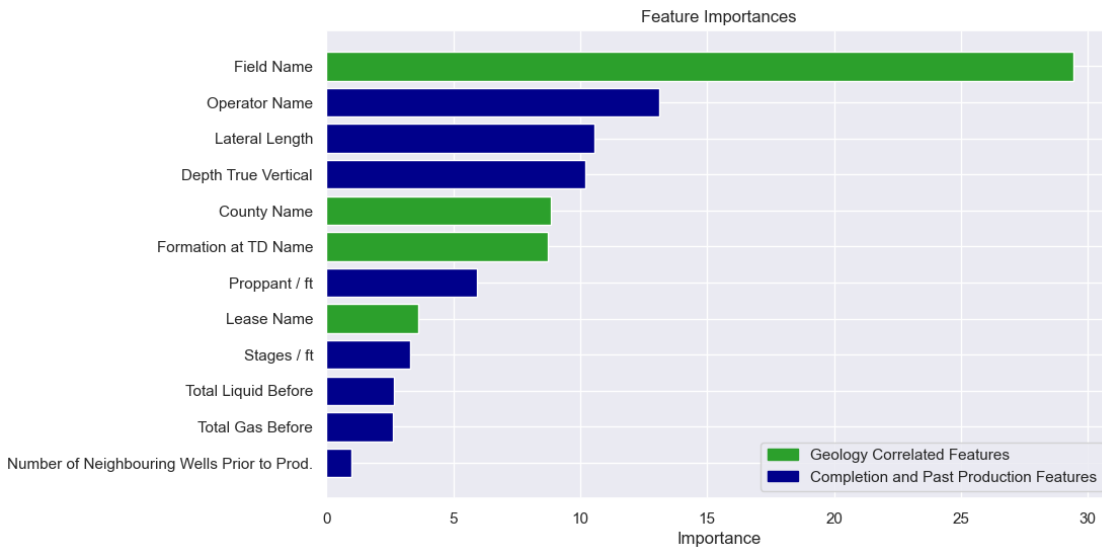
3 Model Interpretation

3.0.1 4

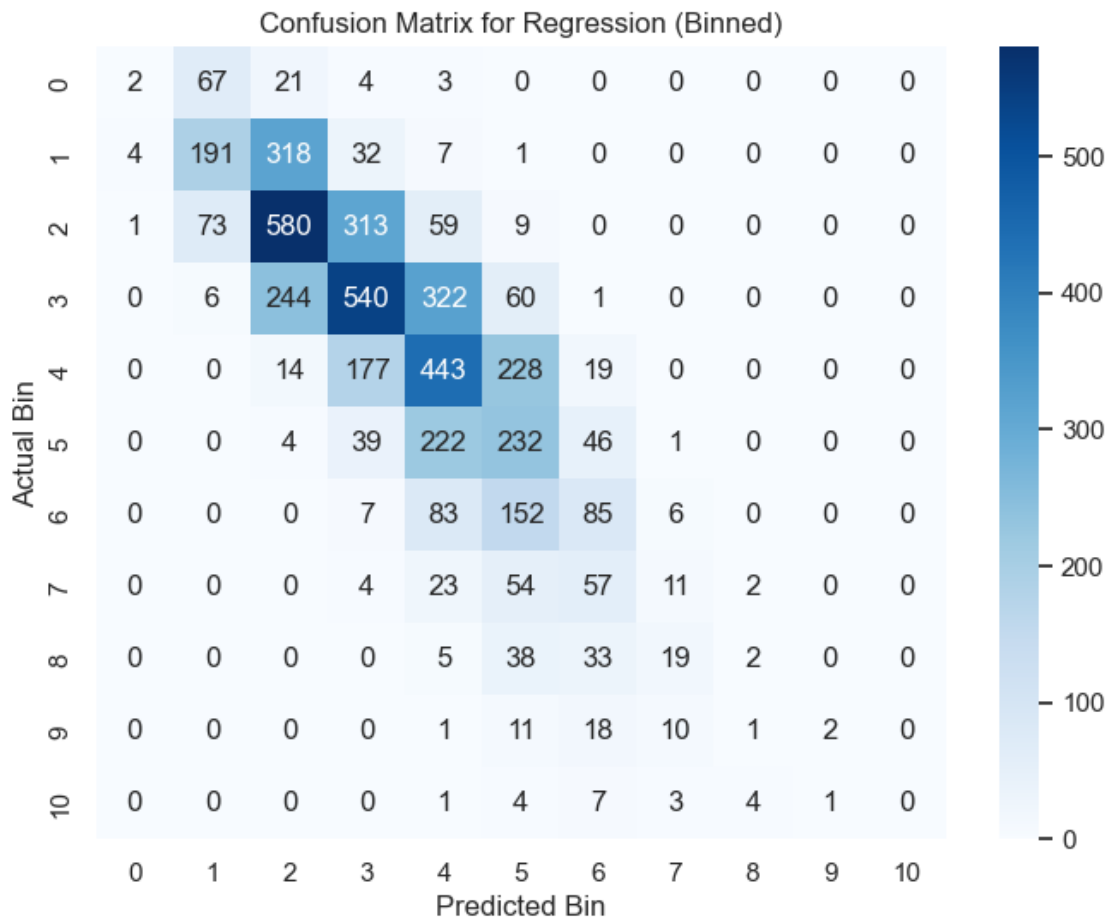
Generate SHAP values from training data, then append them to the training data

[118]:
```
shap_values = model.get_feature_importance(train_pool, type="ShapValues")
```

[119]:
```
feature_importances = model.get_feature_importance()
geologic_features = ['Formation at TD Name', 'County Name', 'Field Name',␣
↪'Lease Name']

feature_importance_df = pd.DataFrame({
    'Feature': cat_train.drop(columns=target+location_info).columns,
    'Importance': feature_importances
})

feature_importance_df = feature_importance_df.sort_values(by='Importance',␣
↪ascending=True)
colors = ['tab:green' if feature in geologic_features else 'darkblue' for␣
↪feature in feature_importance_df['Feature']]

plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'],␣
↪color=colors)
plt.xlabel('Importance')
plt.title('Feature Importances')
important_patch = mpatches.Patch(color='tab:green', label='Geology Correlated␣
↪Features')
other_patch = mpatches.Patch(color='darkblue', label='Completion and Past␣
↪Production Features')
plt.legend(handles=[important_patch, other_patch], loc='lower right')
plt.show()
```



[120]:
```
bins = np.linspace(y_train.min(), y_train.max(), 12)  # 5 bins
bin_labels = range(len(bins)-1)

y_true_binned = np.digitize(y_train, bins) - 1
y_pred_binned = np.digitize(y_train_pred, bins) - 1

# Ensure bin indices stay within bounds
y_true_binned = np.clip(y_true_binned, 0, len(bin_labels) - 1)
y_pred_binned = np.clip(y_pred_binned, 0, len(bin_labels) - 1)

conf_matrix = confusion_matrix(y_true_binned, y_pred_binned)
conf_df = pd.DataFrame(conf_matrix, index=bin_labels, columns=bin_labels)

plt.figure(figsize=(8, 6))
sns.heatmap(conf_df, annot=True, fmt="d", cmap="Blues", cbar=True)
plt.xlabel("Predicted Bin")
plt.ylabel("Actual Bin")
plt.title("Confusion Matrix for Regression (Binned)")
plt.show()
bin_labels = [f"{bins[i]:.0f} - {bins[i+1]:.0f}" for i in range(len(bins)-1)]
for l in range(len(bin_labels)):
    print(f'Bin {l}: {bin_labels[l]}')
```




```
Bin 0: 1424 - 45493
Bin 1: 45493 - 89562
Bin 2: 89562 - 133631
Bin 3: 133631 - 177700
Bin 4: 177700 - 221769
Bin 5: 221769 - 265837
Bin 6: 265837 - 309906
Bin 7: 309906 - 353975
Bin 8: 353975 - 398044
Bin 9: 398044 - 442113
Bin 10: 442113 - 486182
```

Based on this, we can tell two things: the model tends to overpredict lower values, and underpredict higher values.

[121]:
```
# Get interactions from CatBoost
interactions = model.get_feature_importance(type="Interaction")
```

[122]:
```
interactions = model.get_feature_importance(type="Interaction")
interaction_df = pd.DataFrame(interactions, columns=["Feature 1", "Feature 2",␣
↪"Score"])

# Pivot to create a matrix format
feature_names = cat_train.drop(columns=target+location_info).columns
interaction_matrix = interaction_df.pivot(index="Feature 1", columns="Feature␣
↪2", values="Score").fillna(0)
interaction_matrix.columns = [feature for feature in cat_train.
↪drop(columns=target+location_info).columns[1:]]
interaction_matrix.index = [feature for feature in cat_train.
↪drop(columns=target+location_info).columns[:-1]]
```

[123]:
```
high_interaction = interaction_df['Score'].describe()['75%']
interaction_matrix.style.applymap(highlight_high_values)
```

[123]:
```
<pandas.io.formats.style.Styler at 0x2301e469090>
```

From this we observe that completion parameters predictive ability is increased when paired with the operator, and geology predictive ability is increased when including completion and operator information.

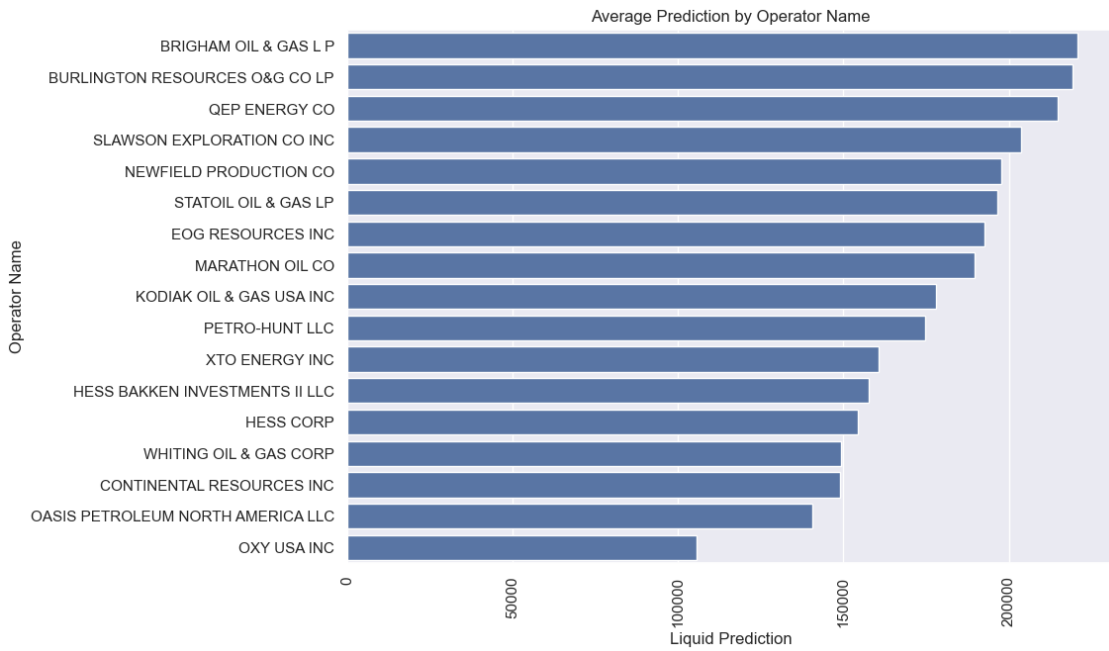
Let's visualize predictions per various categorical features.

[124]:
```
cat_train['prediction'] = y_train_pred
feature_name = 'Operator Name'
```

[125]:
```
cat_operator_train = cat_train.copy(deep=True)
cat_operator_train = cat_operator_train[cat_operator_train['Operator Name'].
↪map(cat_operator_train['Operator Name'].value_counts()) > 100]
```

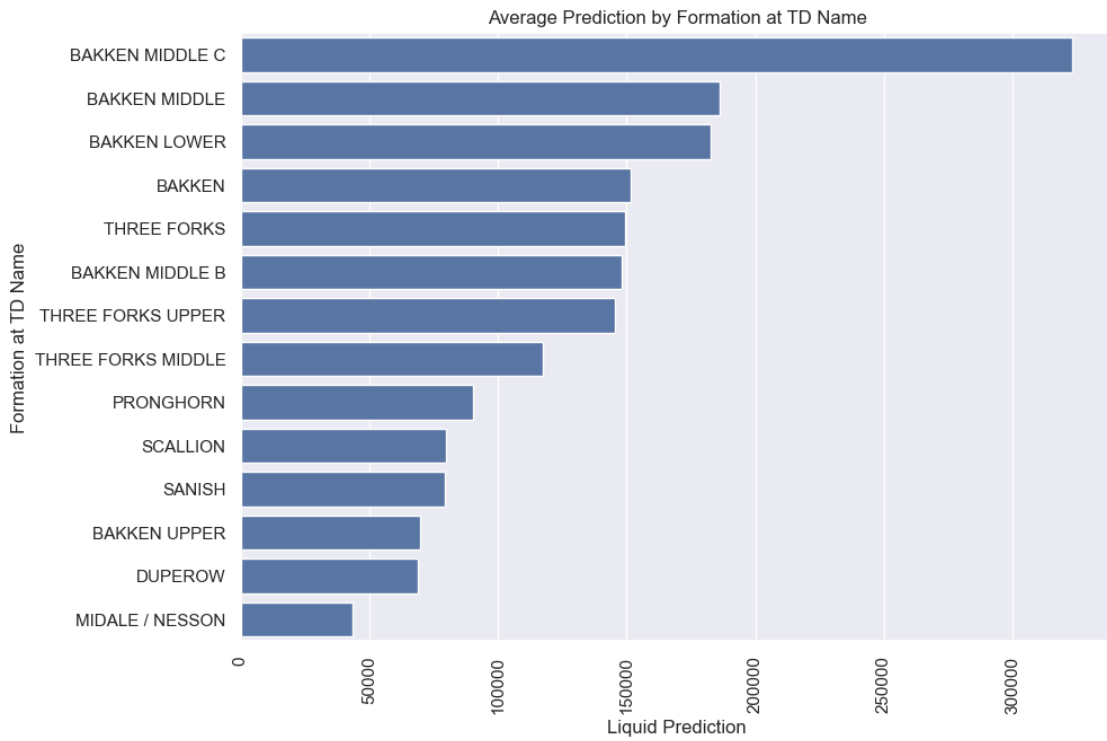
[126]:
```
cat_effects = cat_operator_train.groupby(feature_name)['prediction'].median().
↪reset_index()
```

[127]:
```
plt.figure(figsize=(10, 7))
sns.barplot(data=cat_effects, x="prediction", y=feature_name, order=cat_effects.
↪sort_values("prediction", ascending=False)[feature_name])
plt.xticks(rotation=90)
plt.title(f"Average Prediction by {feature_name}")
plt.xlabel('Liquid Prediction')
plt.show()
```



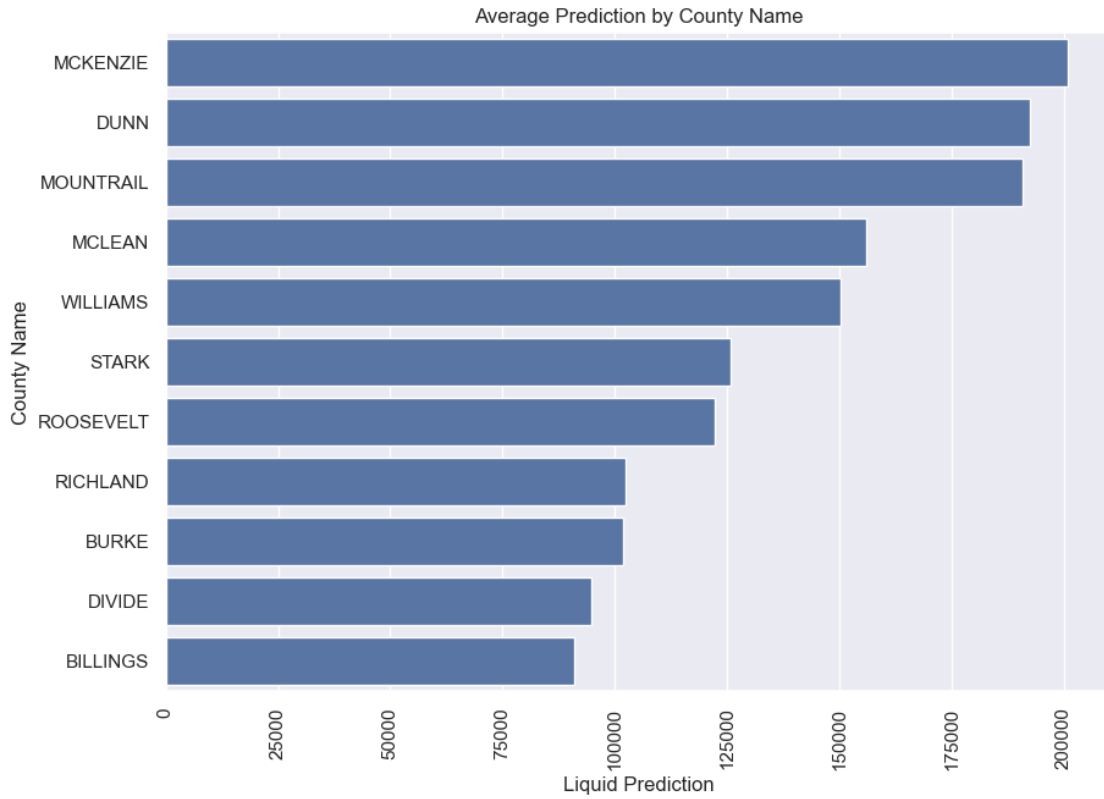
[128]:
```
cat_formation_train = cat_train.copy(deep=True)
feature_name = 'Formation at TD Name'
cat_effects = cat_formation_train.groupby(feature_name)['prediction'].median().
 ↪reset_index()
```

[129]:
```
plt.figure(figsize=(10, 7))
sns.barplot(data=cat_effects, x="prediction", y=feature_name, order=cat_effects.
 ↪sort_values("prediction", ascending=False)[feature_name])
plt.xticks(rotation=90)
plt.title(f"Average Prediction by {feature_name}")
plt.xlabel('Liquid Prediction')
plt.show()
```



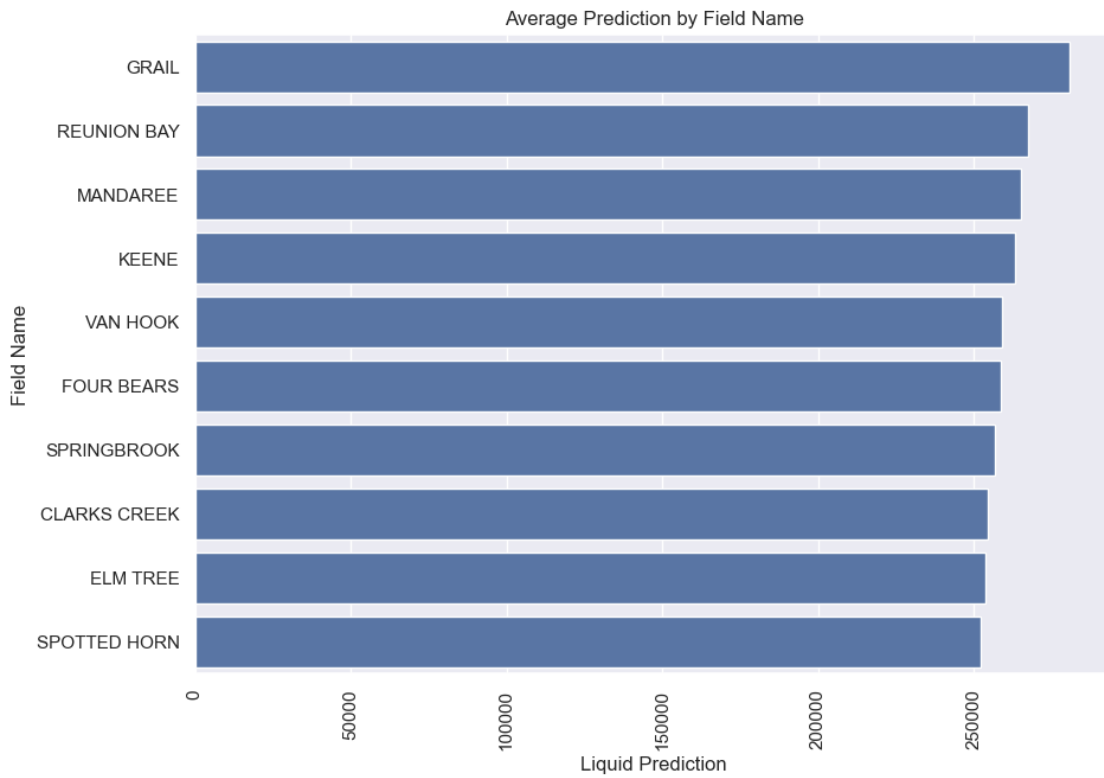
[130]:
```
cat_county_train = cat_train.copy(deep=True)
feature_name = 'County Name'
cat_effects = cat_county_train.groupby(feature_name)['prediction'].median().
 ↪reset_index()
```

[131]:
```
plt.figure(figsize=(10, 7))
sns.barplot(data=cat_effects, x="prediction", y=feature_name, order=cat_effects.
 ↪sort_values("prediction", ascending=False)[feature_name])
plt.xticks(rotation=90)
plt.title(f"Average Prediction by {feature_name}")
plt.xlabel('Liquid Prediction')
plt.show()
```



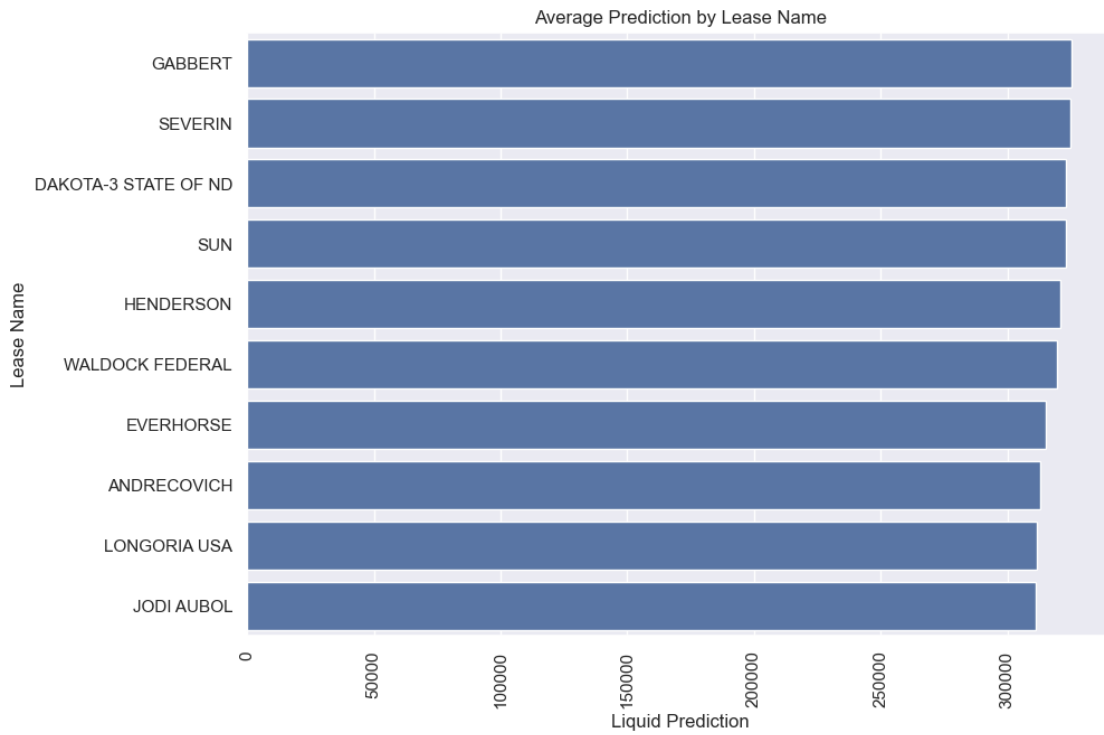
[132]:
```
cat_county_train = cat_train.copy(deep=True)
feature_name = 'Field Name'
cat_effects = cat_county_train.groupby(feature_name)['prediction'].median().
 ↪reset_index()
cat_effects = cat_effects.sort_values("prediction",␣
 ↪ascending=False)[[feature_name, 'prediction']]
```

[133]:
```
plt.figure(figsize=(10, 7))
sns.barplot(data=cat_effects[:10], x="prediction", y=feature_name)
plt.xticks(rotation=90)
plt.title(f"Average Prediction by {feature_name}")
plt.xlabel('Liquid Prediction')
plt.show()
```



[134]:
```
cat_county_train = cat_train.copy(deep=True)
feature_name = 'Lease Name'
cat_effects = cat_county_train.groupby(feature_name)['prediction'].median().
 ↪reset_index()
cat_effects = cat_effects.sort_values("prediction",␣
 ↪ascending=False)[[feature_name, 'prediction']]
```

[135]:
```
plt.figure(figsize=(10, 7))
sns.barplot(data=cat_effects[:10], x="prediction", y=feature_name)
plt.xticks(rotation=90)
plt.title(f"Average Prediction by {feature_name}")
plt.xlabel('Liquid Prediction')
plt.show()
```



[136]:
```
operators_behind_top_leases = cat_train[cat_train['Lease Name'].
 ↪isin(cat_effects[:10]['Lease Name'])]['Operator Name'].unique()
print('Operators behind top leases:')
for i in operators_behind_top_leases:
    print(i)
```

```
Operators behind top leases:
HELIS OIL & GAS CO LLC
MARATHON OIL CO
ZENERGY OPERATING CO LLC
WHITING OIL & GAS CORP
ENERPLUS RESOURCES (USA) CORP
BURLINGTON RESOURCES O&G CO LP
QEP ENERGY CO
```

[137]:
```
tree = model.plot_tree(tree_idx=50)
```

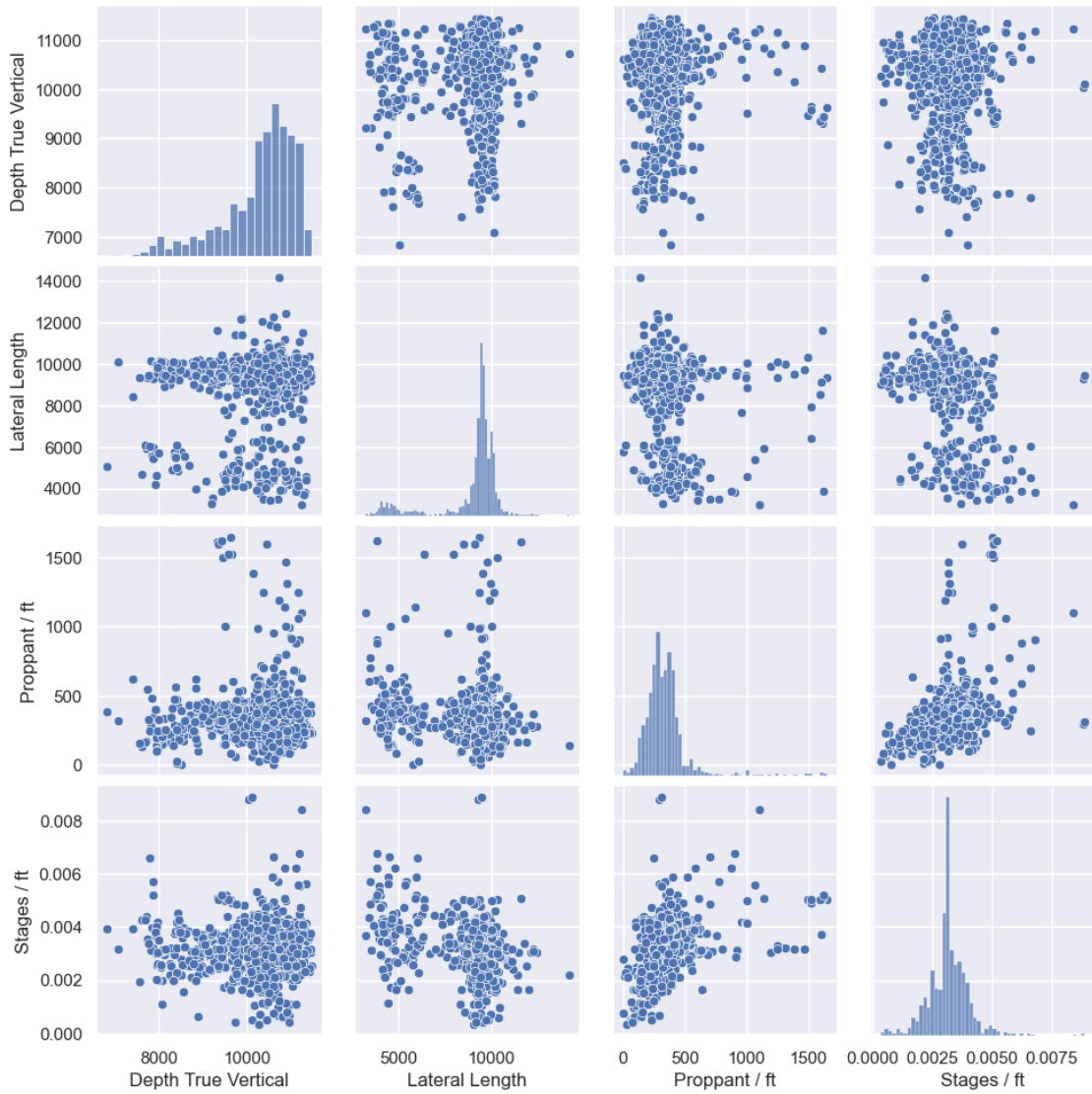
[138]:
```
frequently_seen_variables = [1, 8, 10, 9, 5, 7, 0]
frequently_seen_features = [cat_train.drop(columns=target+location_info).
 ↪columns[i] for i in frequently_seen_variables]
```

[139]:
```
print('| Frequently used features in trees |')
for i in frequently_seen_features:
    print(f'- {i}')
```

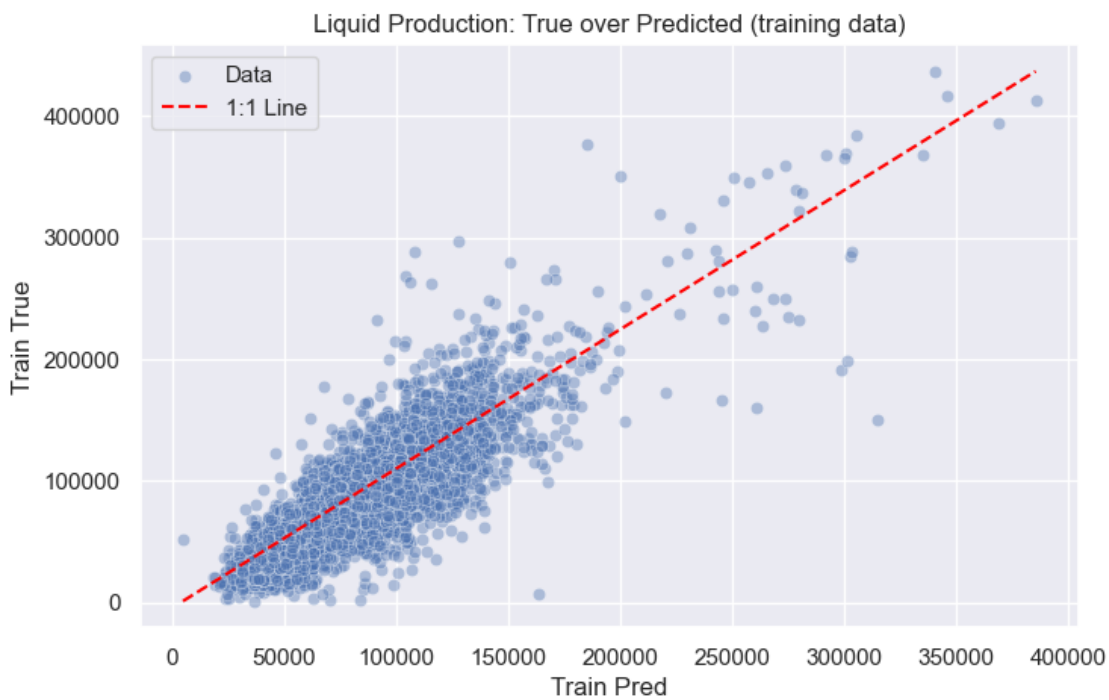
```
| Frequently used features in trees |
- Lateral Length
- Formation at TD Name
- Field Name
- County Name
- Total Gas Before
- Operator Name
- Depth True Vertical
```

Of particular importance is Field Name, which the model is using as a final check in almost every tree to split the predictions. This is a clear reflection of localized production trends.

[140]:
```
sns.pairplot(features_df[num_features[:4]].sample(1000, random_state=42));
```

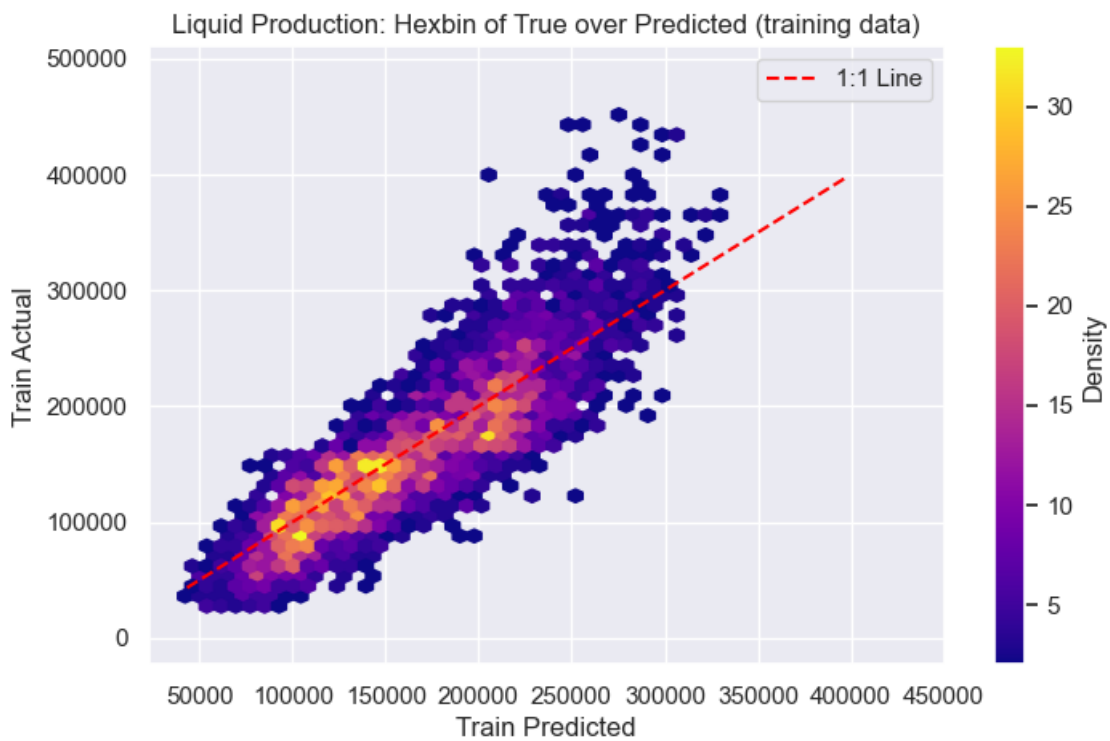


[331]:
```
train_pred_true_df = pd.DataFrame({'Train True': y_train, 'Train Pred':␣
↪y_train_pred})

plt.figure(figsize=(8, 5))
sns.scatterplot(data=train_pred_true_df, x='Train Pred', y='Train True',␣
↪label='Data', alpha=0.4)
plt.plot([min(train_pred_true_df['Train Pred']), max(train_pred_true_df['Train␣
↪Pred'])], [min(train_pred_true_df['Train True']),␣
↪max(train_pred_true_df['Train True'])], linestyle='--', color='red',␣
↪label='1:1 Line')
plt.title('Liquid Production: True over Predicted (training data)')
plt.legend();
```

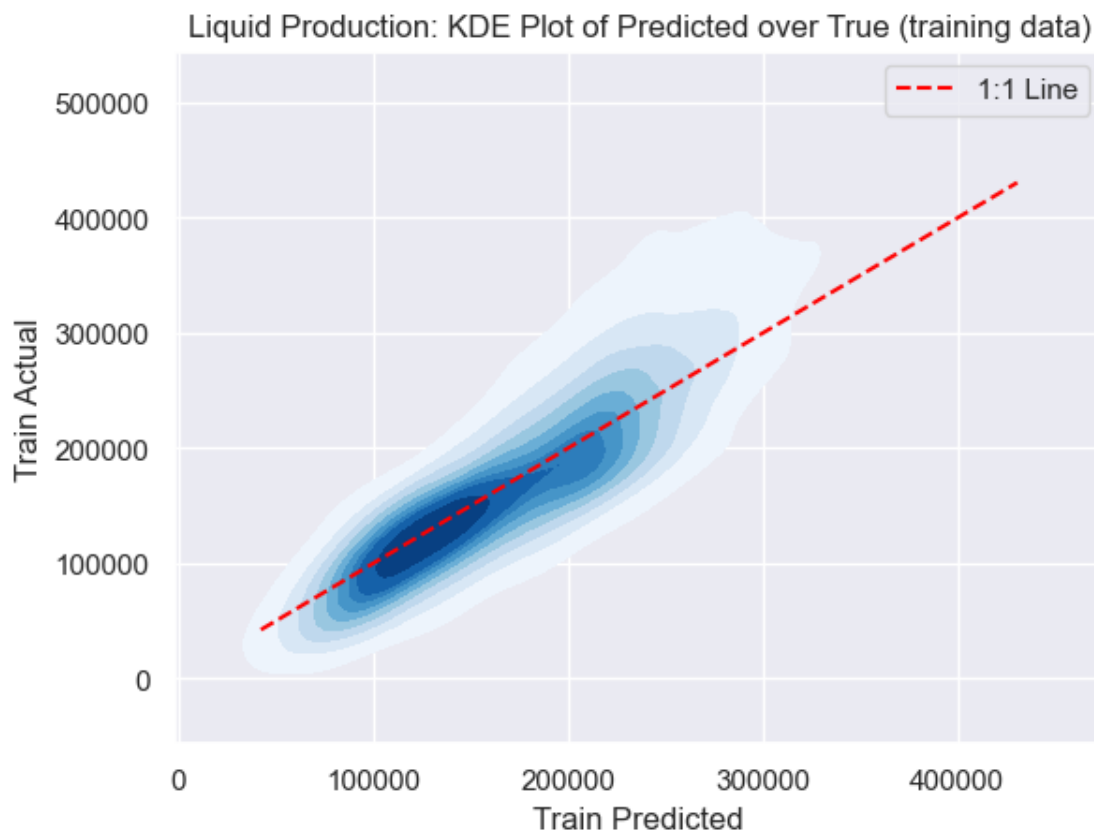


[142]:
```
plt.figure(figsize=(8, 5))
plt.hexbin(y_train_pred, y_train, gridsize=50, cmap='plasma', mincnt=2)  #␣
↪Adjust gridsize as needed

# Add 1:1 reference line
plt.plot([min(y_pred), max(y_pred)], [min(y_pred), max(y_pred)],␣
↪linestyle='--', color='red', label='1:1 Line')

# Labels and legend
plt.xlabel("Train Predicted")
```

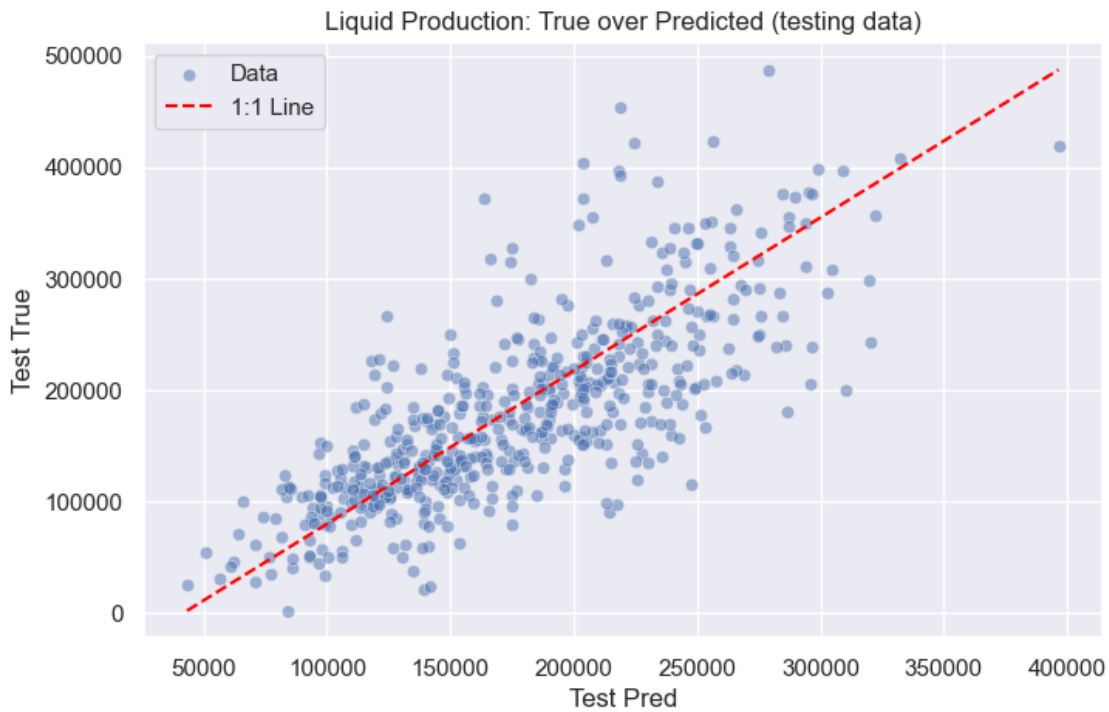
```
plt.ylabel("Train Actual")
plt.colorbar(label="Density")
plt.title('Liquid Production: Hexbin of True over Predicted (training data)')
plt.legend()

plt.show()
```



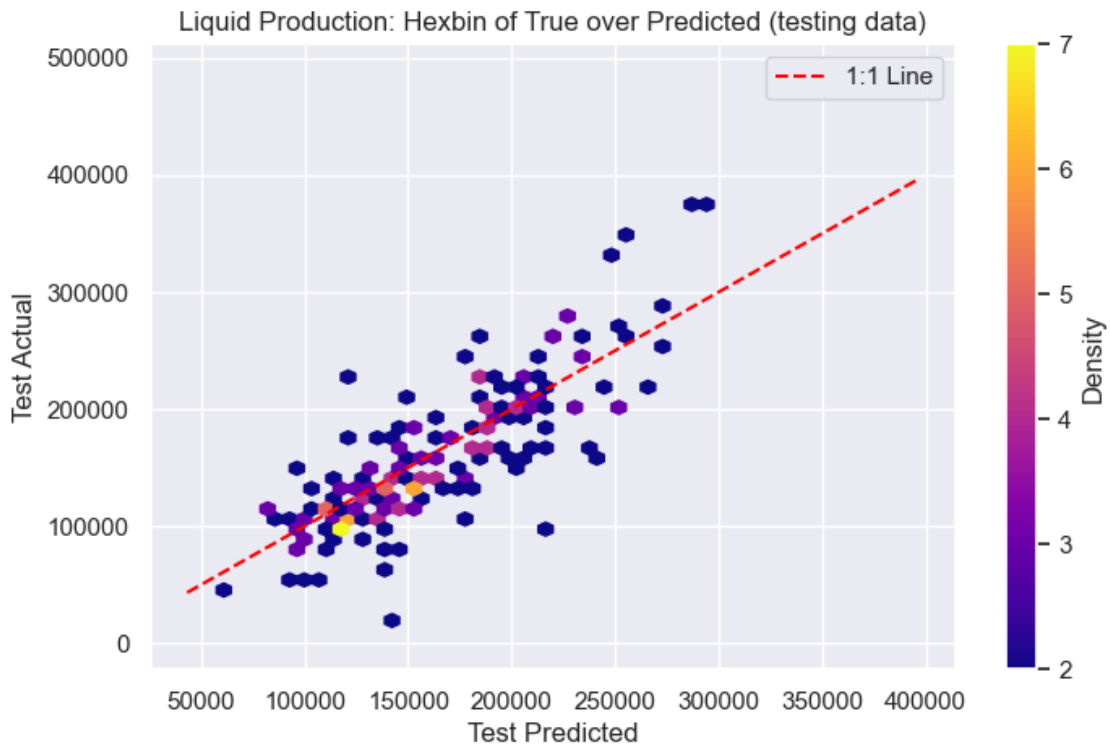
[143]:
```
sns.kdeplot(x=train_pred_true_df['Train Pred'], y=train_pred_true_df['Train␣
↪True'], cmap="Blues", fill=True)

# Add 1:1 line
plt.plot([min(y_train_pred), max(y_train_pred)], [min(y_train_pred),␣
↪max(y_train_pred)], linestyle='--', color='red', label='1:1 Line')

plt.xlabel("Train Predicted")
plt.ylabel("Train Actual")
plt.title('Liquid Production: KDE Plot of Predicted over True (training data)')
plt.legend()
plt.show()
```



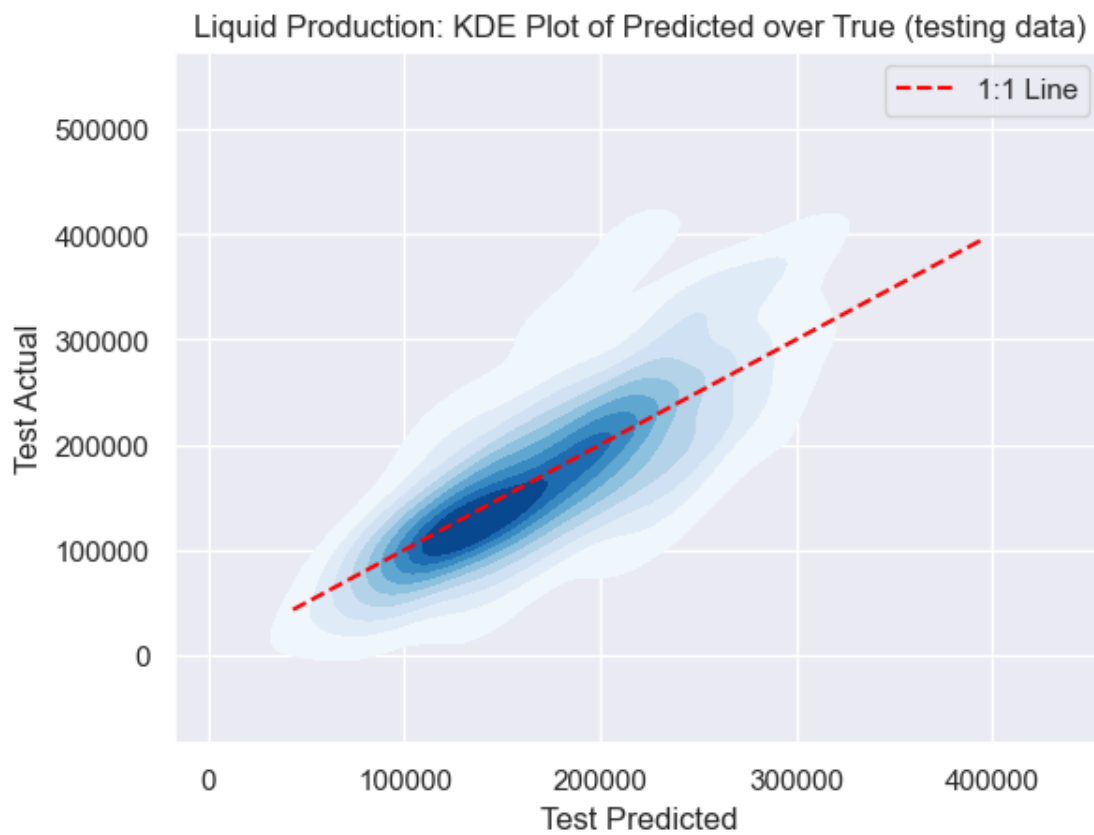
[144]:
```
test_pred_true_df = pd.DataFrame({'Test True': y_test, 'Test Pred': y_pred})

plt.figure(figsize=(8, 5))
sns.scatterplot(data=test_pred_true_df, x='Test Pred', y='Test True',␣
↪label='Data', alpha=0.5)
plt.plot([min(test_pred_true_df['Test Pred']), max(test_pred_true_df['Test␣
↪Pred'])], [min(test_pred_true_df['Test True']), max(test_pred_true_df['Test␣
↪True'])], linestyle='--', color='red', label='1:1 Line')
plt.title('Liquid Production: True over Predicted (testing data)')
plt.legend();
```



[145]:
```
plt.figure(figsize=(8, 5))
plt.hexbin(y_pred, y_test, gridsize=50, cmap='plasma', mincnt=2)  # Adjust␣
↪gridsize as needed

# Add 1:1 reference line
plt.plot([min(y_pred), max(y_pred)], [min(y_pred), max(y_pred)],␣
↪linestyle='--', color='red', label='1:1 Line')

# Labels and legend
plt.xlabel("Test Predicted")
plt.ylabel("Test Actual")
plt.colorbar(label="Density")
plt.title('Liquid Production: Hexbin of True over Predicted (testing data)')
plt.legend()

plt.show()
```



[146]:
```
sns.kdeplot(x=test_pred_true_df['Test Pred'], y=test_pred_true_df['Test True'],␣
↪cmap="Blues", fill=True)

# Add 1:1 line
plt.plot([min(y_pred), max(y_pred)], [min(y_pred), max(y_pred)],␣
↪linestyle='--', color='red', label='1:1 Line')

plt.xlabel("Test Predicted")
plt.ylabel("Test Actual")
plt.title('Liquid Production: KDE Plot of Predicted over True (testing data)')
plt.legend()
plt.show()
```

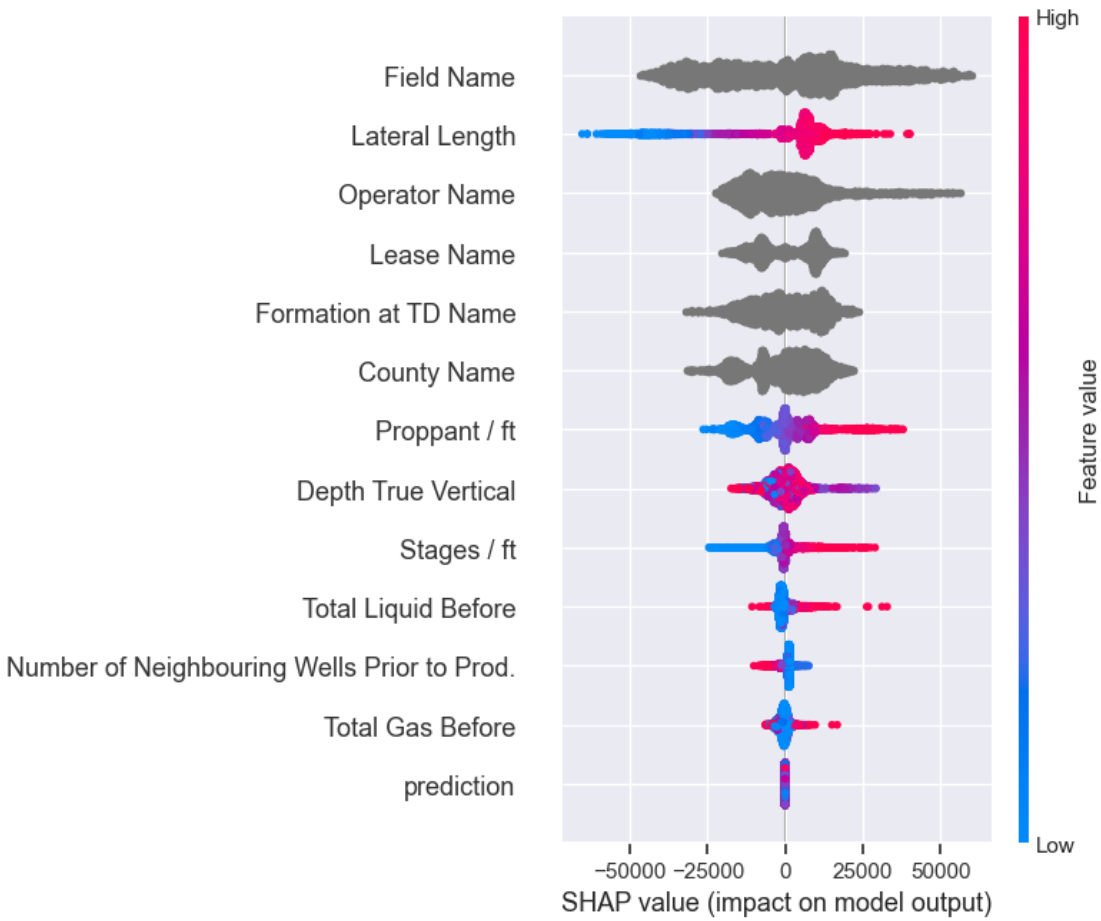


An observation can be made at this point. After looking at the KDE and Hexbin plots for both training and testing data, it is evident that accuracy of the model is correlated to the density of points in a region. The denser the region, or the more available data is within that range of production values, the greater model accuracy is. Uneven sampling is one of the greatest challenges to ML models within the oil and gas industry.

Calculation of SHAP Values for further explainability:

[147]:
```
explainer = shap.TreeExplainer(model,␣
↪feature_perturbation="tree_path_dependent")
shap_values = explainer.shap_values(cat_train.
↪drop(columns=target+location_info))
```

[148]:
```
# Summary plot
shap.summary_plot(shap_values, cat_train.drop(columns=target+location_info))
```



Field name dominates as the most important feature. Lateral length stands out as the most important completion parameter.

[149]:
```
geology_SHAP = np.array([])
for q in tqdm(range(len(cat_train))):
    geology_contribution = shap_values[q][-4:].sum()
    geology_SHAP = np.append(geology_SHAP, geology_contribution)
cat_train['Geology_SHAP'] = geology_SHAP
```

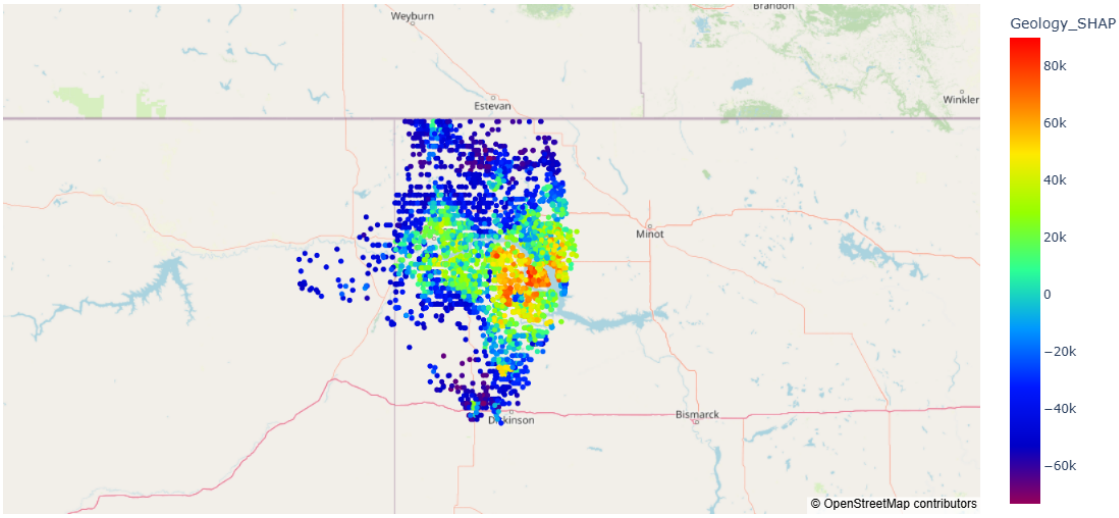
```
100%|

                    | 4927/4927 [00:00<00:00, 22628.12it/s]
```

3.0.2 5

Visualize DataFrame containing Lat, Long, Geology_SHAP

[150]:
```
target_hue = 'Geology_SHAP'
```

```
df_visualize_comp_7 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_7['X'] = cat_train['Surface Latitude']
df_visualize_comp_7['Y'] = cat_train['Surface Longitude']
df_visualize_comp_7[target_hue] = cat_train[target_hue]
create_location_map(df_visualize_comp_7, 'X', 'Y', target_hue, 6)
```



3.0.3 6

Interpolate geology SHAP values for testing data

Method 1: Kriging

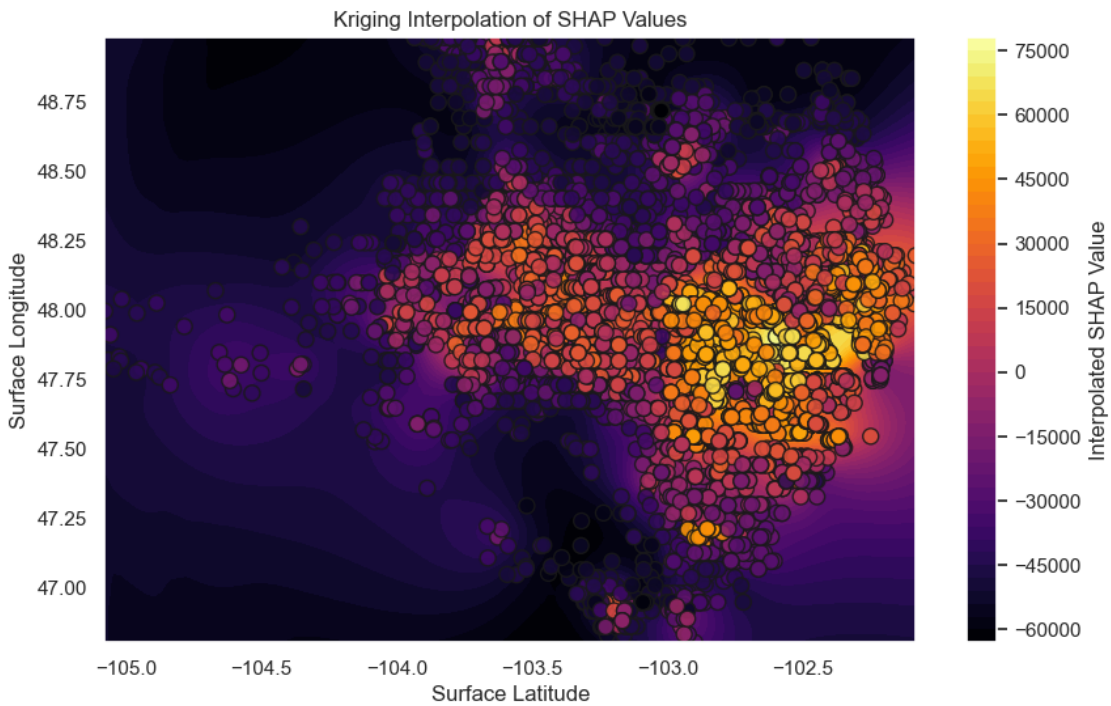
[151]:
```
OK = OrdinaryKriging(cat_train["Surface Latitude"], cat_train["Surface␣
↪Longitude"], cat_train["Geology_SHAP"], variogram_model="spherical")
```

[152]:
```
%%time
X_pred = np.linspace(cat_train["Surface Latitude"].min(), cat_train["Surface␣
↪Latitude"].max(), 100)
Y_pred = np.linspace(cat_train["Surface Longitude"].min(), cat_train["Surface␣
↪Longitude"].max(), 100)
X_grid, Y_grid = np.meshgrid(X_pred, Y_pred)

# Perform Kriging Interpolation
Shap_pred, _ = OK.execute("grid", X_pred, Y_pred)
```

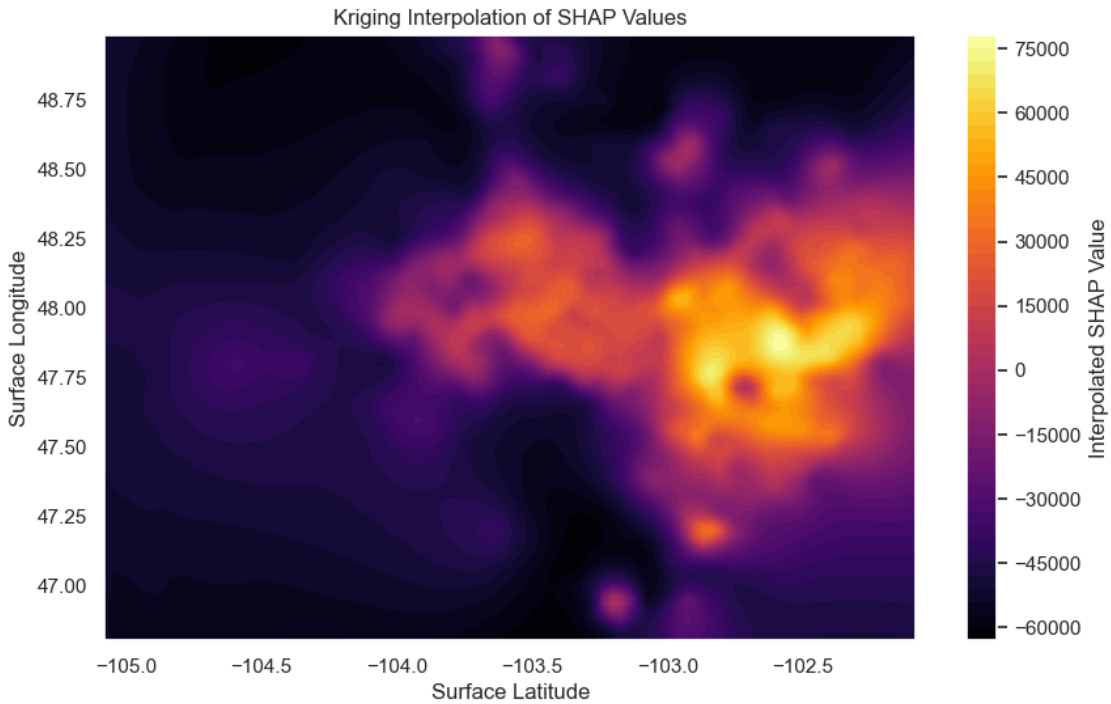
```
CPU times: total: 16min 2s
Wall time: 2min 31s
```

[153]:
```
plt.figure(figsize=(10, 6))
plt.contourf(Y_grid, X_grid, Shap_pred, cmap="inferno", levels=50)  # Kriging␣
↪SHAP heatmap
```

```
plt.colorbar(label="Interpolated SHAP Value")

X_known = cat_train["Surface Latitude"].values  # Surface Latitude
Y_known = cat_train["Surface Longitude"].values  # Surface Longitude
Shap_known = cat_train["Geology_SHAP"].values  # SHAP values

# Overlay known SHAP points
plt.scatter(Y_known, X_known, c=Shap_known, edgecolors="k", cmap="inferno",␣
↪s=75, alpha=0.88)
plt.xlabel("Surface Latitude")
plt.ylabel("Surface Longitude")
plt.title("Kriging Interpolation of SHAP Values")
plt.show()
```



[154]:
```
plt.figure(figsize=(10, 6))
plt.contourf(Y_grid, X_grid, Shap_pred, cmap="inferno", levels=50)  # Kriging␣
↪SHAP heatmap
plt.colorbar(label="Interpolated SHAP Value")
plt.xlabel("Surface Latitude")
plt.ylabel("Surface Longitude")
plt.title("Kriging Interpolation of SHAP Values")
plt.show();
```



[155]:
```
%%time
cat_test["Geology_SHAP"], _variance = OK.execute("points", cat_test["Surface␣
↪Latitude"], cat_test["Surface Longitude"])
```

```
CPU times: total: 12min 5s
Wall time: 1min 43s
```

3.0.4 7

Re-train model using Geology_SHAP

[156]:
```
# For retraining CatBoost model using interpolated shap values:
rerun_with_SHAP()
```

```
Median Absolute Percent Error (MdAPE): 14.78%
-----------------------------------
Median Absolute Percent Error (MdAPE): 17.61%
```

3.0.5 Geologic interpretation of Model Predictions

The North Dakota Geologic Survey published a report in 2008 describing the geology of the Bakken and containing useful related information such as porosity and thermal maturity.

The figure above shows thermal maturity of the different counties. The counties with top maturities, such as McKenzie and Montrail, are in the top 5 counties with the highest median predicted production. The model is in agreement with this chart.

The above figure shows a map of the Bakken and relevant geologic properties. There are 3 vertical anticlines in North Dakota, and 1 anticline facing perpendicular to the Brockton-Froid fault zone. Notable formations are the Parshall Sanish, Bailey, and Elm Coulee. The Parshall Sanish is situated approximately where the predictive model anticipates good geology. The anticline hinge lines are overlaying wells that show high Gas-Oil ratios, as a result of gas that has risen to the top of the trap. Major oil production is along the edges of the anticline. The Brockton-Froid Fault Zone creates a migration path for hydrocarbons. The Poplar Dome serves as a trap.

The following table gives mean porosity, water saturation, and original oil in place by county. The porosity and water saturations of the different counties are relatively similar. McKenzie, which has the greatest thermal maturity and production, has similar porosity to Golden Valley, one of the worst producing counties. The use of geology over county would likely confuse the model. McKenzie has the larget original oil in place, which is the reason it has the largest median oil production. This shows that OOIP dominates as a predictor for well production over any geology such as porosity or water saturation, especially when the geology is consistent across counties. The model agrees with this, since it has great accuracy in the absence of any explicit geologic features.

4 Conclusions

The CatBoost model achieved good accuracy with limited public data, and in the absence of any explicit geologic features. This implies that the model learned the impact of geology simply through the categorical features describing location. This is significant, as it implies that measuring geologic properties isn't necessary for arriving at accurate productions of wells.

5 Additional Work

The work below is additional material that was done in the experimental stages of data exploration and model creation. Everything above reflects whatever implementations of interest were found below, making it optional for understanding model results.

5.0.1 Diagnosing Outliers

Lets try to better understand the cases with high absolute errors.

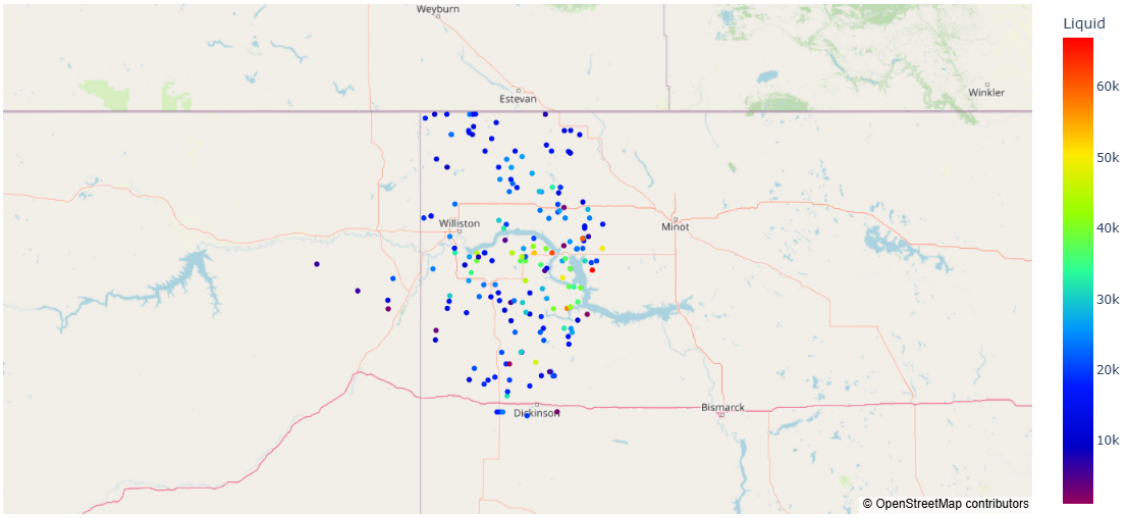
[1092]:
```
absolute_percent_errors = np.abs((y_train - y_train_pred) / y_train) * 100
high_outlier_indices = absolute_percent_errors[absolute_percent_errors > 100].
 ↪reset_index()['index']
```

[1096]:
```
pred_df = prod_comp_df[prod_comp_df.index.isin(high_outlier_indices)]
pred_df = pred_df[target + location_info + num_features + cat_features]
```

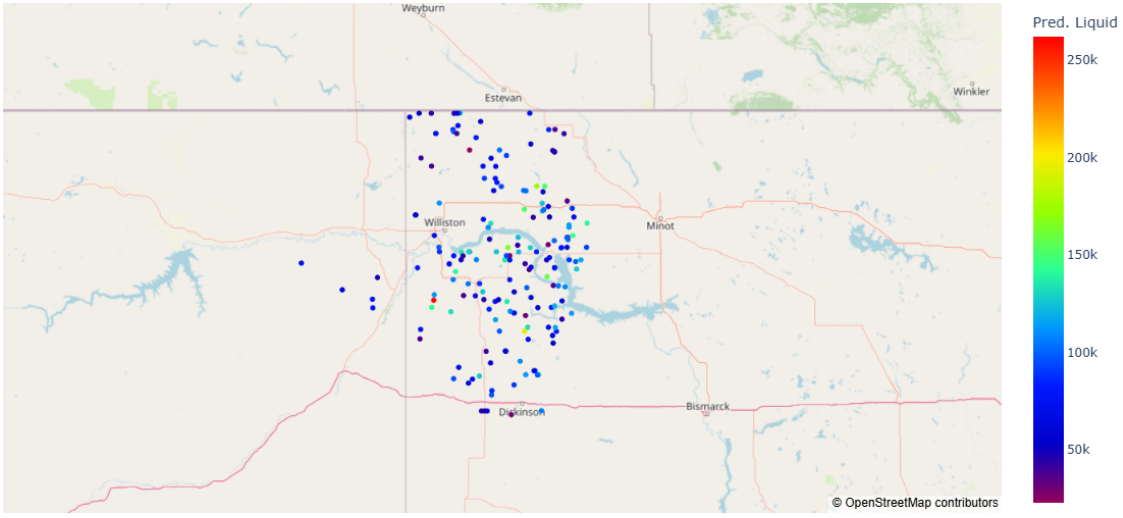
Show high outlier true production values on map.

[1099]:
```
target_hue = 'Liquid'

df_visualize_comp_3 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_3['X'] = pred_df['Surface Latitude']
df_visualize_comp_3['Y'] = pred_df['Surface Longitude']
```

```
df_visualize_comp_3[target_hue] = pred_df[target_hue]
create_location_map(df_visualize_comp_3, 'X', 'Y', target_hue, 6)
```



[1048]:
```
target_hue = 'Pred. Liquid'

df_visualize_comp_3 = pd.DataFrame(columns=['X', 'Y'])
# df_visualize_comp_3['API'] = prod_comp_df['API']
df_visualize_comp_3['X'] = pred_outliers_df['Surface Latitude']
df_visualize_comp_3['Y'] = pred_outliers_df['Surface Longitude']
df_visualize_comp_3[target_hue] = pred_outliers_df[target_hue]
create_location_map(df_visualize_comp_3, 'X', 'Y', target_hue, 6)
```



[1100]:
```
pred_df['Operator Name'].value_counts()
```

[1100]:
```
Operator Name
CONTINENTAL RESOURCES INC                19
WHITING OIL & GAS CORP                   17
XTO ENERGY INC                           17
OXY USA INC                              14
BURLINGTON RESOURCES O&G CO LP           13
EOG RESOURCES INC                        12
MARATHON OIL CO                          11
STATOIL OIL & GAS LP                     10
OASIS PETROLEUM NORTH AMERICA LLC         9
HESS CORP                                 7
BAYTEX ENERGY USA LTD                     7
SLAWSON EXPLORATION CO INC                7
SAMSON RESOURCES CO                       5
PETRO-HUNT LLC                            4
KODIAK OIL & GAS USA INC                  4
CORNERSTONE NATURAL RESOURCES LLC         3
FIDELITY EXPLORATION & PRODUCTION CO      3
NEWFIELD PRODUCTION CO                    3
HUNT OIL CO                               3
QEP ENERGY CO                             3
ENERPLUS RESOURCES (USA) CORP             3
AMERICAN EAGLE ENERGY CORP                3
SM ENERGY CO                              2
WPX ENERGY WILLISTON LLC                  2
BAKKEN HUNTER LLC                         1
KRAKEN OPERATING LLC                      1
SINCLAIR OIL & GAS CO                     1
HESS BAKKEN INVESTMENTS II LLC            1
NADEL & GUSSMAN ROCKIES LLC               1
DAKOTA-3 E&P CO LLC                       1
HRC OPERATING LLC                         1
TRACKER RESOURCES DEVELOPMENT II LLC      1
MUREX PETROLEUM CORP                      1
EMERALD OIL INC                           1
Name: count, dtype: int64
```

[1101]:
```
pred_df['County Name'].value_counts()
```

[1101]:
```
County Name
MCKENZIE     48
MOUNTRAIL    39
DUNN         36
DIVIDE       19
WILLIAMS     17
```

```
BURKE        10
BILLINGS     10
STARK         7
RICHLAND      5
Name: count, dtype: int64
```

[1102]:
```
pred_df['Field Name'].value_counts()
```

[1102]:
```
Field Name
PARSHALL            11
AMBROSE              7
BANKS                6
ALGER                6
ELM COULEE           5
                    ..
FAYETTE              1
LAKE ILO             1
MCGREGORY BUTTES     1
ENGET LAKE           1
BARTA                1
Name: count, Length: 114, dtype: int64
```

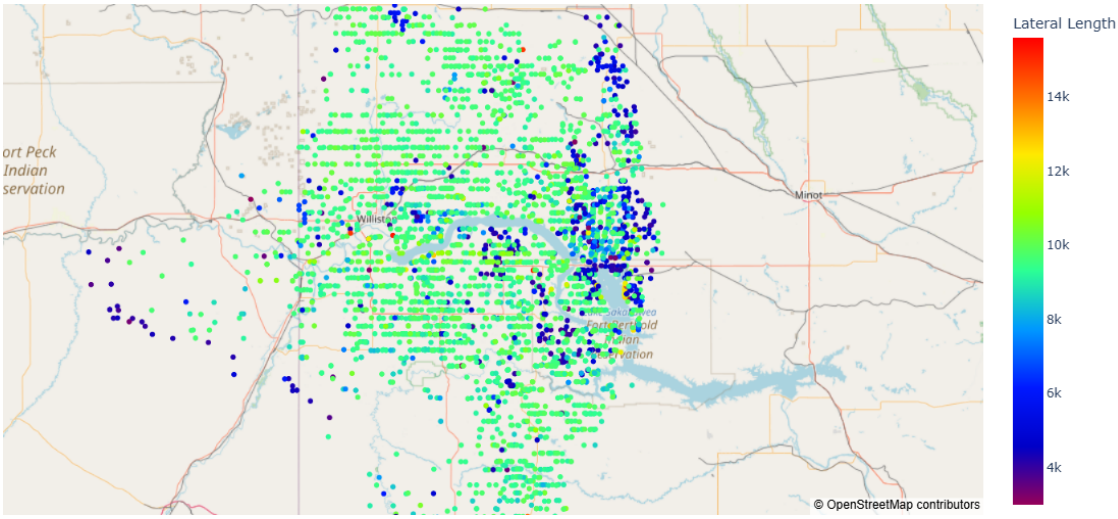
The outliers are being caused by wells that should have had high production given their location, but didn't. This is causing the model difficulty in predictions. This emphasizes the importance of considering the median well over the mean, since there will always be outliers that aren't in accordance with reality.

[390]:
```
# To be continued...
```

5.1 Revisiting some of the data

We must ask ourselves why lateral length and depth true vertical are bimodal, regardless of whether we keep year, field, and operator constant.

[1346]:
```
target_hue = 'Lateral Length'

df_visualize_comp_3 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_3['API'] = prod_comp_df['API']
df_visualize_comp_3['X'] = prod_comp_df['Surface Latitude']
df_visualize_comp_3['Y'] = prod_comp_df['Surface Longitude']
df_visualize_comp_3[target_hue] = prod_comp_df[target_hue]
create_location_map(df_visualize_comp_3, 'X', 'Y', target_hue, 7)
```

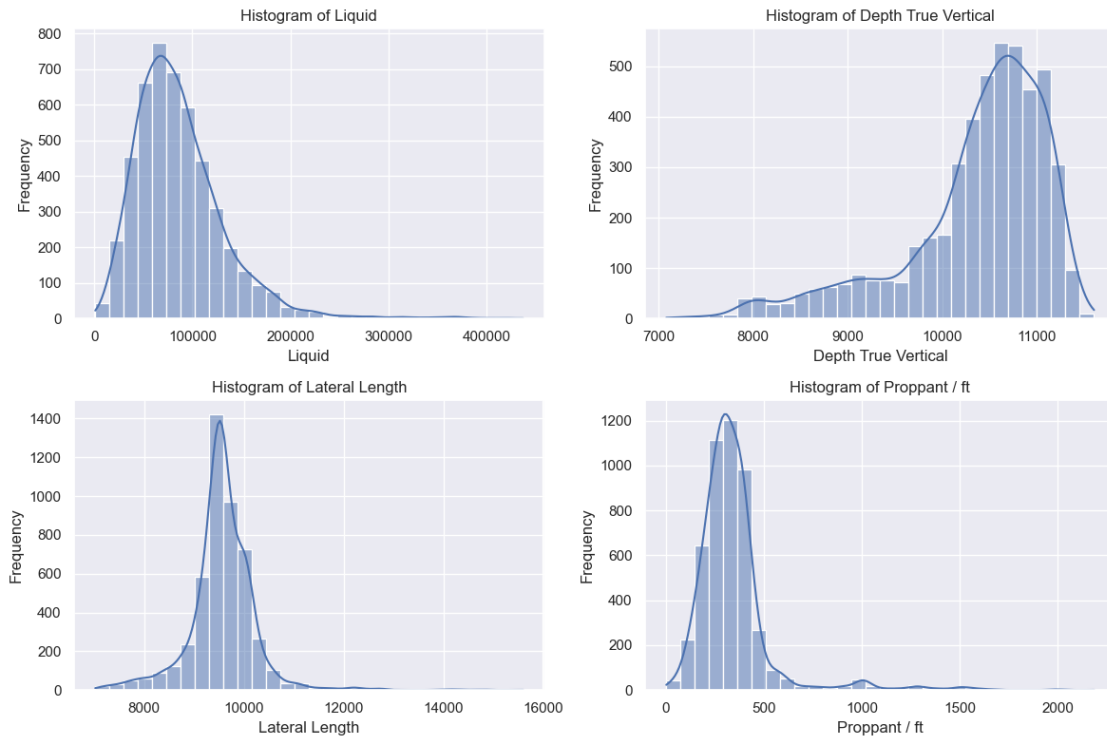


We observe that lateral length splits the population in two. There are either high lateral length or low lateral lengths, which is likely causing model confusion.

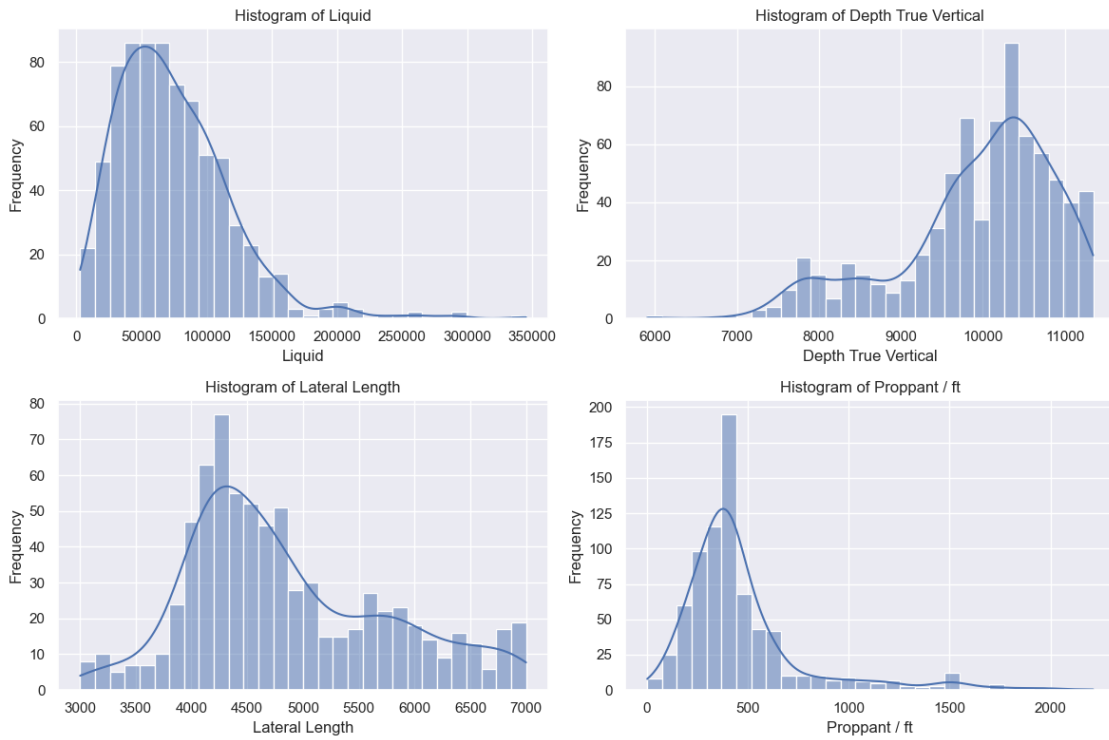
5.1.1 Segmenting based on Lateral Length

Based on the above observation, it is of interest to treat lower lateral lengths and higher lateral lengths as separate populations. Let's investigate the effects of doing that.

[1347]:
```
upper_laterals_df = prod_comp_df[prod_comp_df['Lateral Length'] > 7000]
features_df = upper_laterals_df[target + features]
series_list = [features_df[feat] for feat in (target + features)]
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



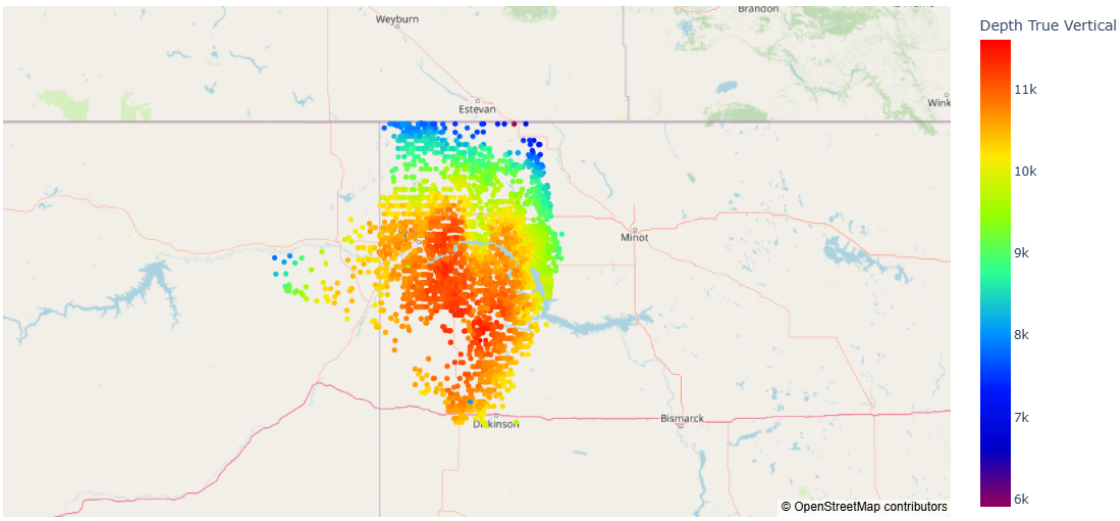
[1348]:
```
lower_laterals_df = prod_comp_df[prod_comp_df['Lateral Length'] <= 7000]
features_df = lower_laterals_df[target + features]
series_list = [features_df[feat] for feat in (target + features)]
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



Interesting. Our upper lateral length distribution becomes much nicer to deal with. Our lower lateral length distribution is still non-normal, but has improved overall since before. Perhaps this hints at the existence of a third distribution of lateral lengths between 5000-6000. Notice that our TVD distribution does not get fixed.

[1349]:
```
# Add in a 3rd lateral length divisor? To be continued...
```

[1350]:
```
target_hue = 'Depth True Vertical'

df_visualize_comp_3 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_3['API'] = prod_comp_df['API']
df_visualize_comp_3['X'] = prod_comp_df['Surface Latitude']
df_visualize_comp_3['Y'] = prod_comp_df['Surface Longitude']
df_visualize_comp_3[target_hue] = prod_comp_df[target_hue]
create_location_map(df_visualize_comp_3, 'X', 'Y', target_hue, 6)
```



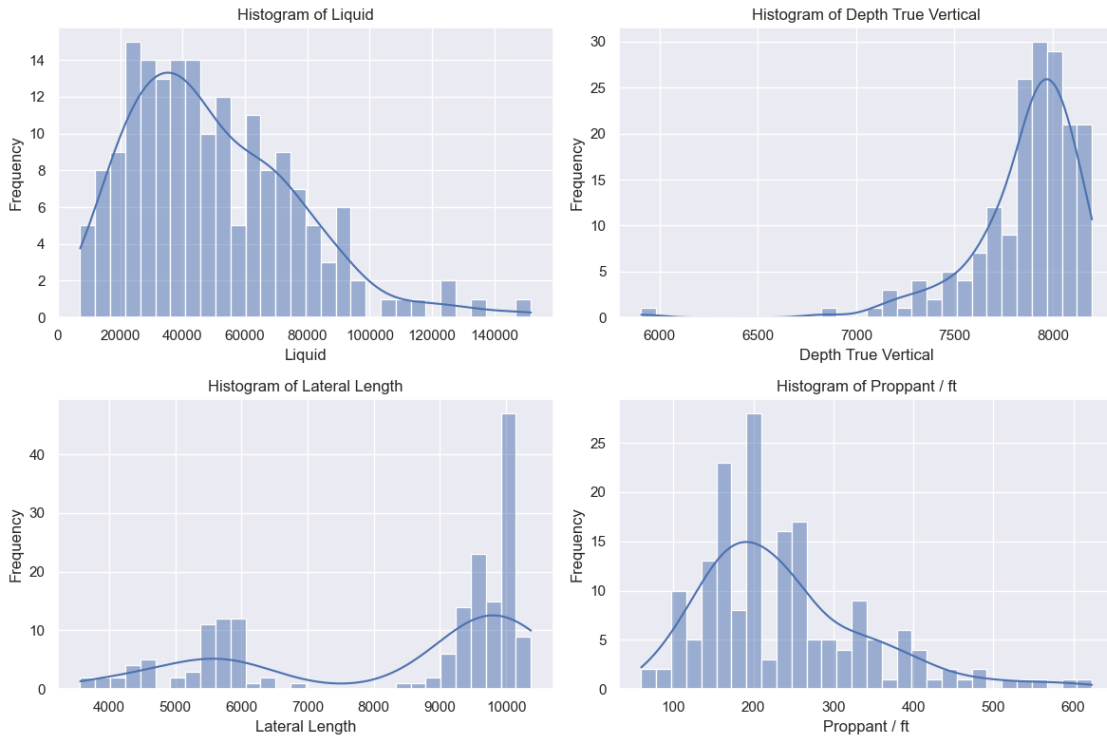
The skew in our data is due to the lack of proper representation of our data. We have biased sampling, where there are more wells that scale with certain depths.

5.1.2 Splitting on TVD and evening out the sample

We do not have even number of observations for depth true vertical. It is worth seeing what would happen if we kept an even number of wells from varying true vertical depths. We will apply stratified sampling, which requires us to select a feature and equally distrbute from clusters (ranges) within that cluster. Let's define the ranges: - Cluster 1: Well TVD is less than 9000 - Cluster 2: Well TVD is between 9000-10000 - Cluster 3: Well TVD is greater than 100000

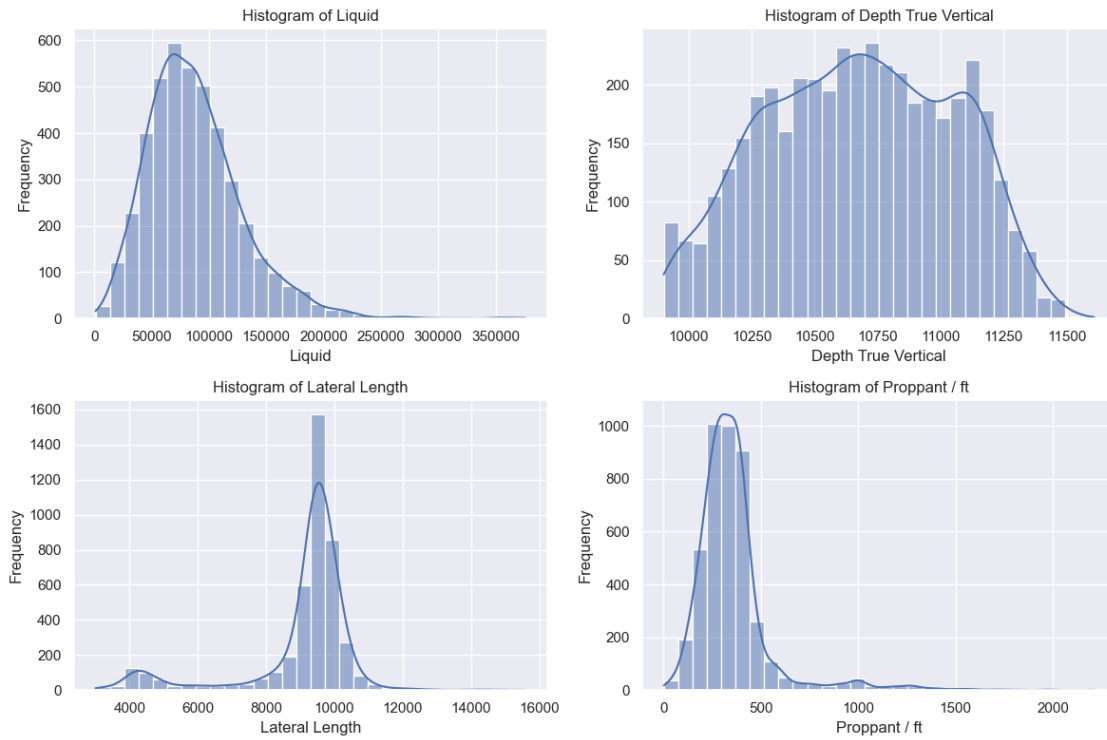
Lets define our clusters.

[1351]:
```
lower_tvd_wells = prod_comp_df[prod_comp_df['Depth True Vertical'] < 8200]
features_df = lower_tvd_wells[target + features]
series_list = [features_df[feat] for feat in (target + features)]
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



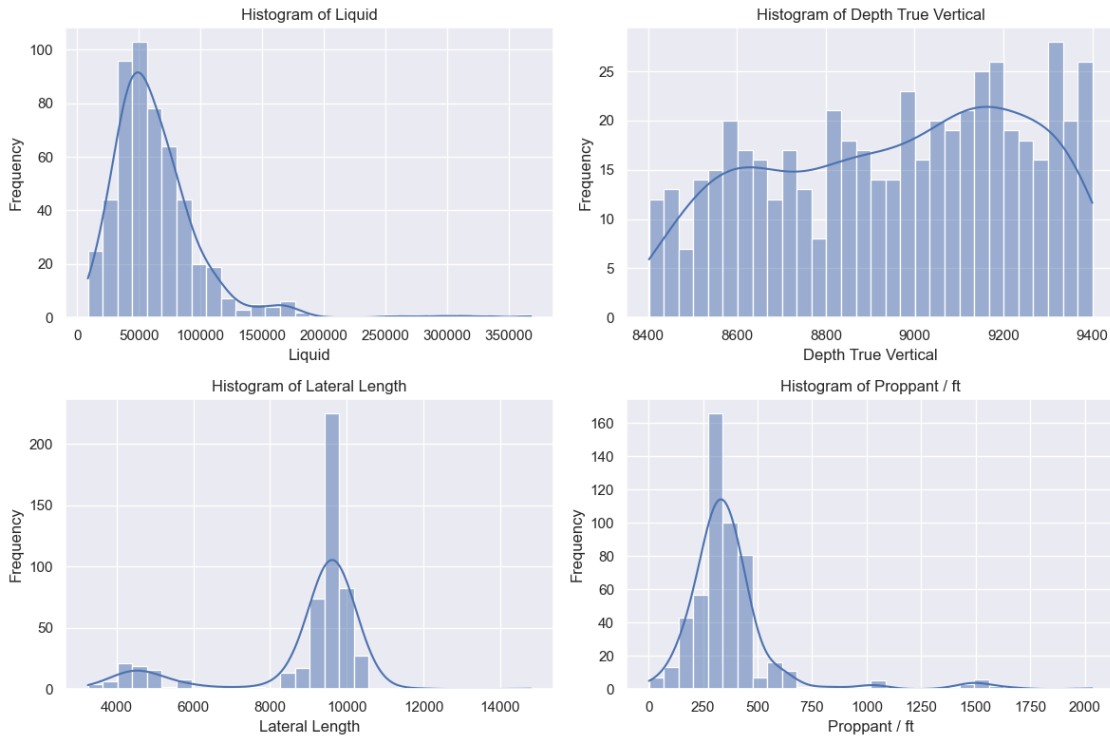
Interesting note. Fixing TVD helps to correct the TVD distribution, but not does correct lateral length. It would seem that these transformations should almost be used together.

[1352]:
```
upper_tvd_wells = prod_comp_df[prod_comp_df['Depth True Vertical'] > 9900]
features_df = upper_tvd_wells[target + features]
series_list = [features_df[feat] for feat in (target + features)]
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



Interesting. When we consider wells with lateral lengths, the TVD distribution becomes much cleaner. Notice as observed above that the distribution of lateral lengths remains bimodal. It is becoming very clear that segmentation based on TVD and lateral length may prove helpful.

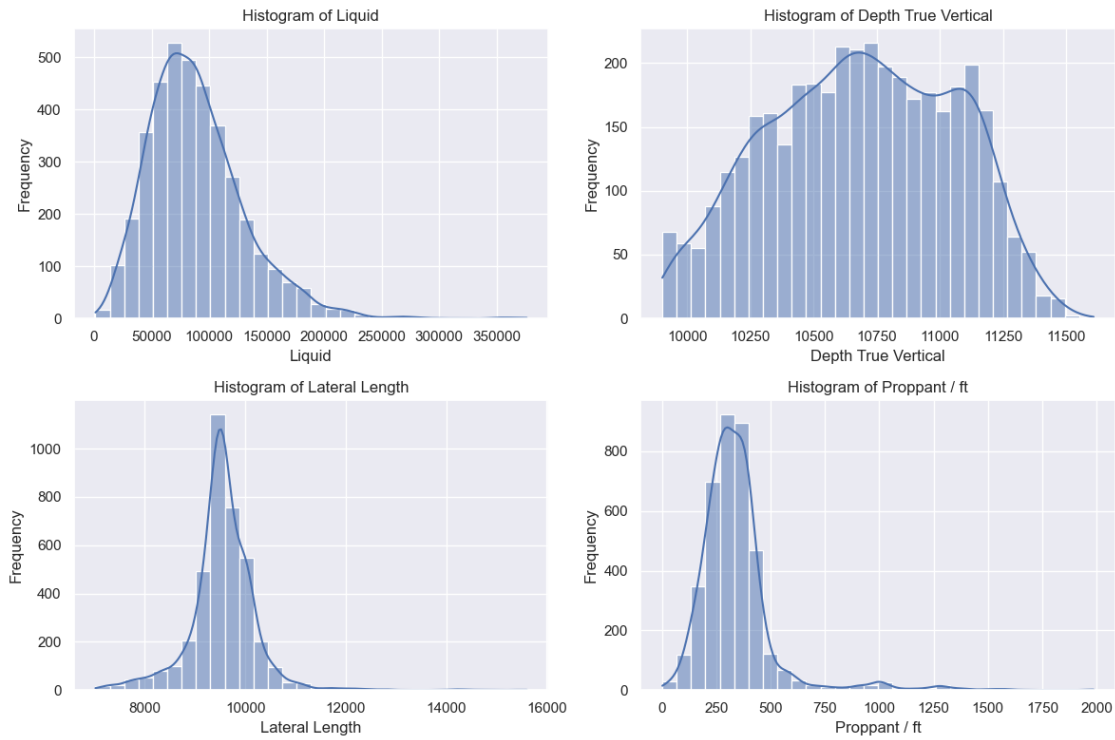
[1353]:
```
middle_tvd_wells = prod_comp_df[(prod_comp_df['Depth True Vertical'] <= 9400) &␣
↪(prod_comp_df['Depth True Vertical'] >= 8400)]
features_df = middle_tvd_wells[target + features]
series_list = [features_df[feat] for feat in (target + features)]
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



As for the middle values, we see an increasing distribution of values towards larger lateral lengths. However, it is reasonably safe to assume that this is due to spatial bias. In reality, our distribution is simply short of values to fill the space on the shorter side.

We conclude that TVD can be decomposed into 3 separate populations, but that this does not correct the bimodal behaviour of the lateral length. This suggests a combination of the features would be best to truly isolate populations. Lets look at an example of this for the upper TVD wells.

[1354]:
```
upper_tvd_and_lateral_wells = upper_tvd_wells[upper_tvd_wells['Lateral Length']␣
↪> 7000]
features_df = upper_tvd_and_lateral_wells[target + features]
series_list = [features_df[feat] for feat in (target + features)]
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=30, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



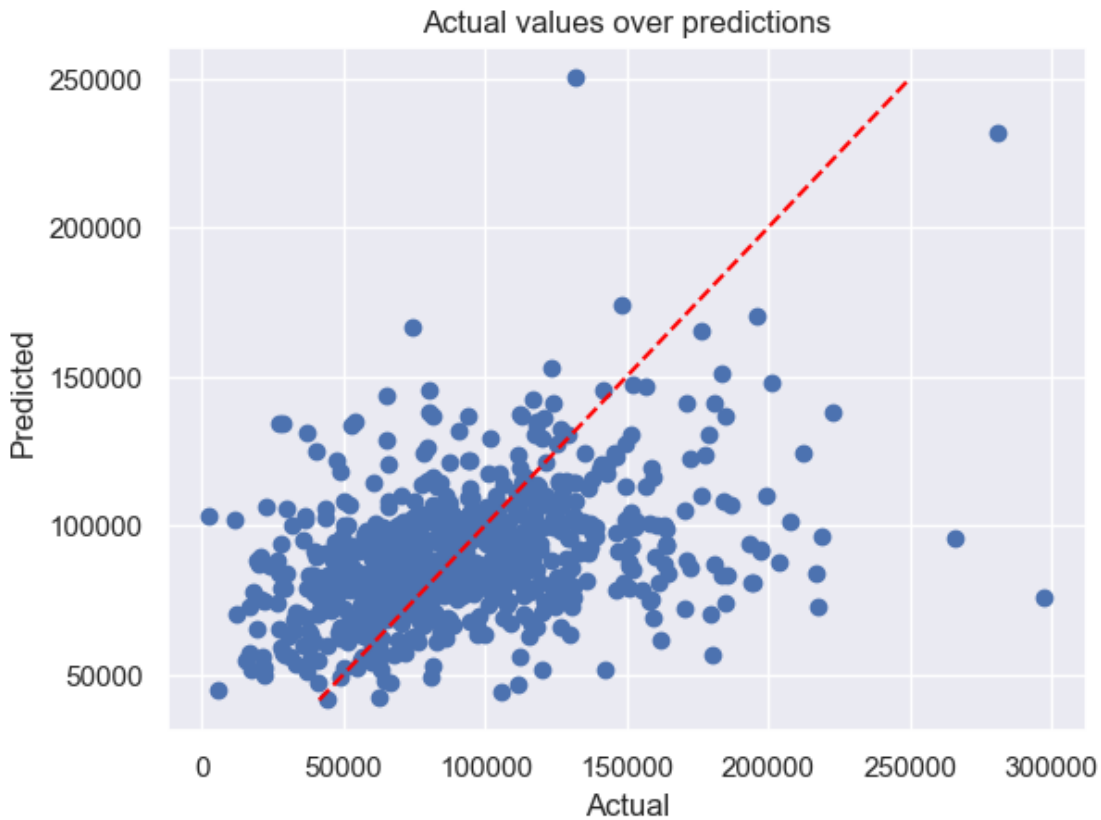
Incredible! By segmenting based on lateral length and TVD, we have reduced our distributions to ones we can work with in the case of an upper producing well. Let's quickly run this through a model and check the accuracy.

[1355]:
```
features_df = upper_tvd_and_lateral_wells[target + features]
X = features_df[features]
y = features_df[target]
```

[1356]:
```
y_test, y_pred = scale_train_fit_eval_model(X, y, LinearRegression())
```

```
RMSE: 41447.74124
MRMSE: 25875.194622676398
Median Absolute Percent Error (MdAPE): 29.61%
```

[1382]:
```
plt.scatter(y_test, y_pred)
plt.title('Actual values over predictions')
plt.plot([min(y_pred), max(y_pred)], [min(y_pred), max(y_pred)],␣
 ↪linestyle='--', color='red', label='1:1 Line')
plt.xlabel('Actual')
plt.ylabel('Predicted');
```



[1358]:
```
y_test, y_pred = scale_train_fit_eval_model(X, y,␣
↪RandomForestRegressor(n_estimators=100, random_state=42))
```

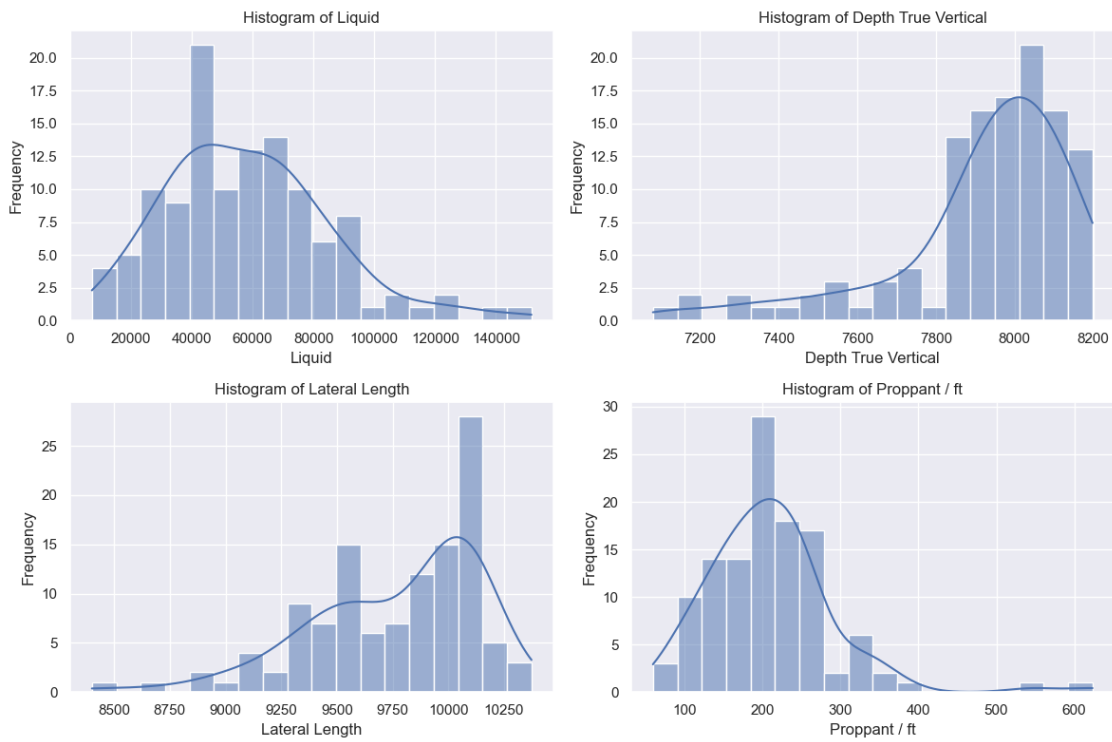
```
RMSE: 38630.05860
MRMSE: 21933.98000000004
Median Absolute Percent Error (MdAPE): 24.83%
```

Interestingly enough, the accuracy of our model does not increase when we fix the distribution of our wells. That implies that this issue is related to the inputs of the model. We do not have good enough input features (as expected of a baseline model).

We can apply a similar segmentation to the lower producing wells.

[1359]:
```
lower_tvd_and_higher_lateral_wells = lower_tvd_wells[lower_tvd_wells['Lateral␣
↪Length'] > 7000]
features_df = lower_tvd_and_higher_lateral_wells[target + features]
series_list = [features_df[feat] for feat in (target + features)]
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

for ax, series, label in zip(axes.flatten(), series_list, target+features):
    sns.histplot(series, bins=18, kde=True, ax=ax)
    ax.set_title(f'Histogram of {label}')
```

```
    ax.set_xlabel(label)
    ax.set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()
plt.show()
```



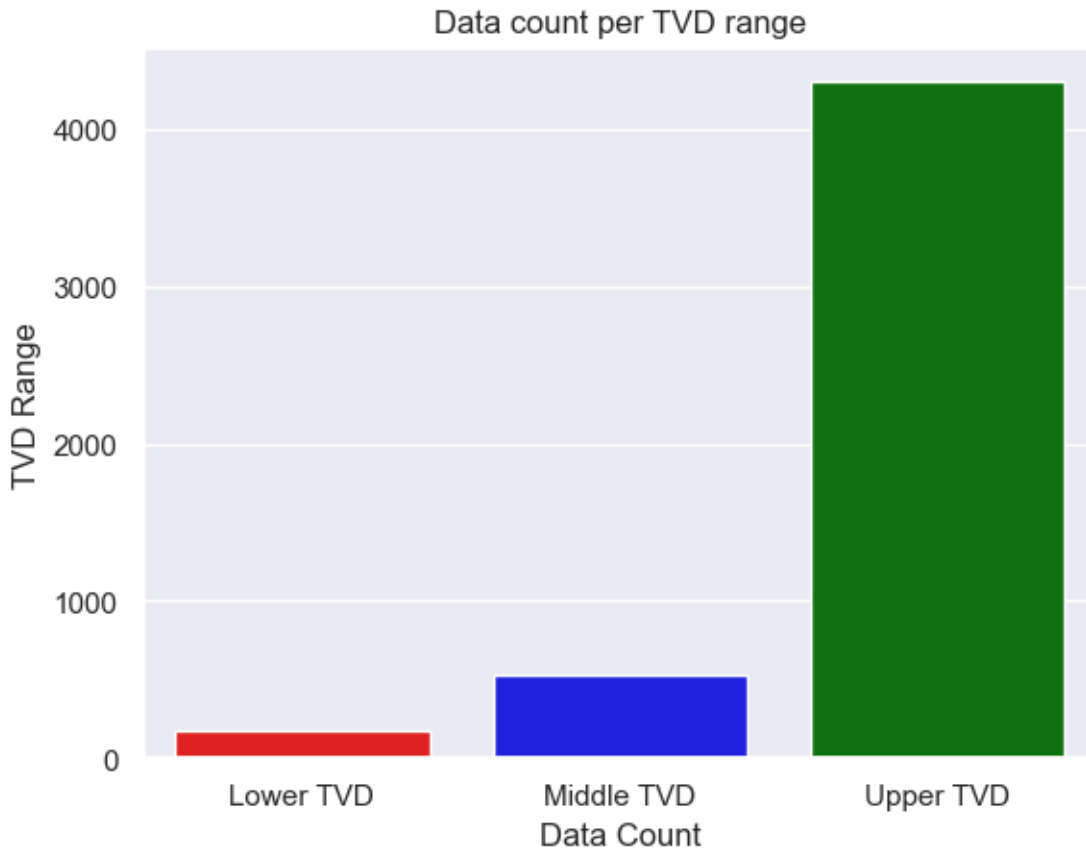
Notice that the distsibution here is much worse. However, this subset of data only has around 190 wells. The data is minimal. This is one of the drawbacks of segmentation, but if we have reason to believe that the feature follows a normal distribution, we can fix this.

5.1.3 TVD Based debiasing

Let's prove that the distribution of data is not equal across the different vertical depth ranges.

[1360]:
```
categories = ["Lower TVD", "Middle TVD", "Upper TVD"]
tvd_series_values = [len(lower_tvd_wells), len(middle_tvd_wells),␣
 ↪len(upper_tvd_wells)]
tvd_diff_viz = pd.DataFrame({"Category": categories, "Value":␣
 ↪tvd_series_values})

sns.barplot(x="Category", y="Value", data=tvd_diff_viz, palette=['red', 'blue',␣
 ↪'green'])
plt.xlabel("Data Count")
```

```
plt.ylabel("TVD Range")
plt.title("Data count per TVD range")
plt.show()
```



As expected, we have a lot of data for wells that target the Upper TVD, but little to no data for the lower and middle TVDs. We have so few data that spatial debiasing may be improbable. It may make sense to concentrate on just Upper TVD wells and proceed.

[1361]:
```
lower_tvd_wells['Formation at TD Name'].value_counts()
```

[1361]:
```
Formation at TD Name
THREE FORKS        100
BAKKEN MIDDLE       74
BAKKEN               1
SANISH               1
MIDALE / NESSON      1
Name: count, dtype: int64
```

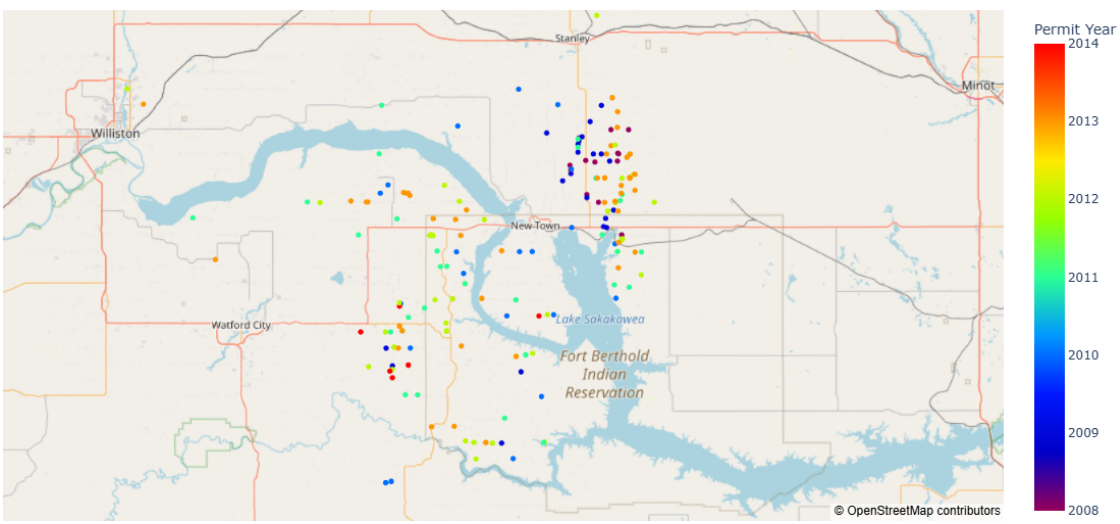
We observe that the majority of these wells were targeting the Three Forks formation. Thus, it may make sense to include formation type as a feature in our model, or remove that data altogether.

5.1.4 Removing high outlier wells

We can consider the removal of the upper outlier wells since they are mainly EOG Resources Inc wells with exceptional operational and technical foundation. Furthermore, if we wanted a general idea of production, including outlier cases due to operational differences would be skewing the model to higher values. It is also possible that higher production is associated with lease dates that were earlier. Let's check this.

[1362]:
```
outliers_prod_comp_df['Permit Date'] = pd.
 ↪to_datetime(outliers_prod_comp_df['Permit Date'])
outliers_prod_comp_df['Permit Year'] = outliers_prod_comp_df['Permit Date'].dt.
 ↪year
```

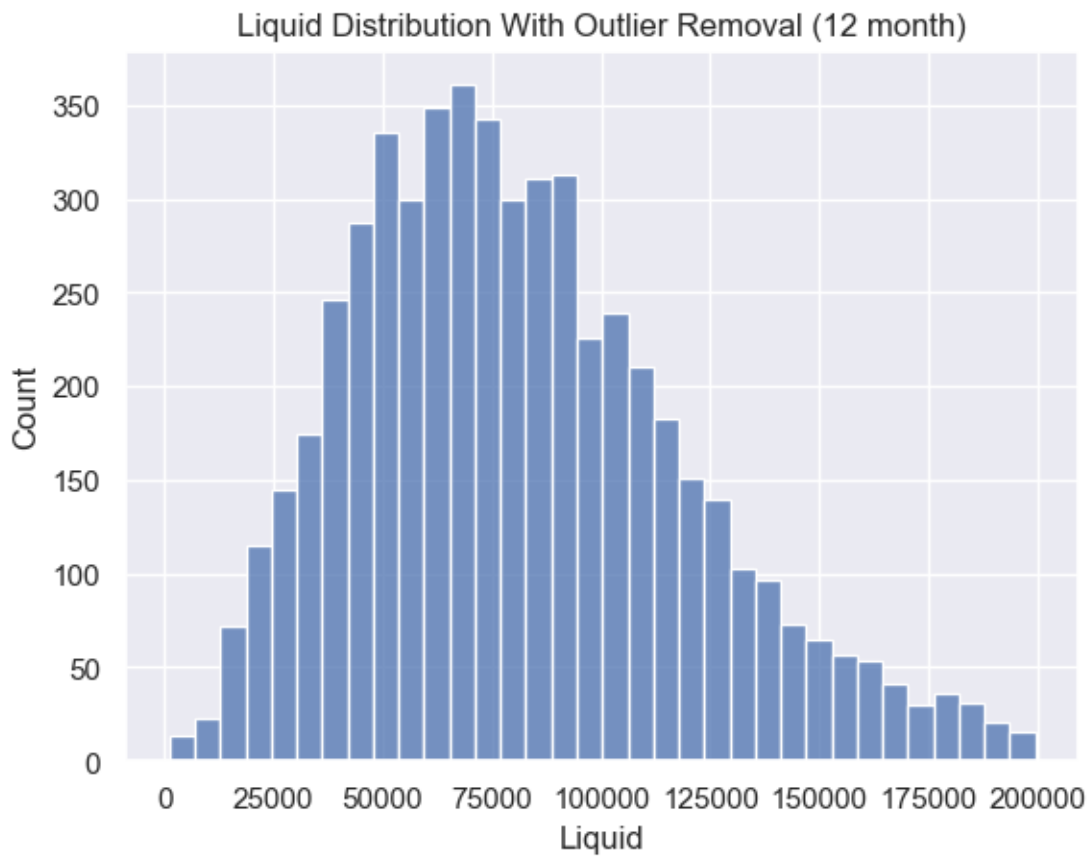
[1363]:
```
target_hue = 'Permit Year'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['API'] = outliers_prod_comp_df['API']
df_visualize_comp_4['X'] = outliers_prod_comp_df['Surface Latitude']
df_visualize_comp_4['Y'] = outliers_prod_comp_df['Surface Longitude']
df_visualize_comp_4[target_hue] = outliers_prod_comp_df[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 8)
```



It seems that most of the permits are evenly distributed. Most permits for the higher producing area were obtained in 2013.

[1364]:
```
# Code for removing only upper EOG outliers
# eog_outliers_to_remove =␣
 ↪outliers_prod_comp_df[(outliers_prod_comp_df['Operator Name'] == 'EOG␣
 ↪RESOURCES INC') & (outliers_prod_comp_df['Liquid'] > 210000)]['API']
# prod_comp_df = prod_comp_df[~prod_comp_df['API'].isin(eog_outliers_to_remove)]
```

[1385]:
```
prod_comp_df_liquid_outliers_removed = prod_comp_df[prod_comp_df['Liquid'] <␣
↪200000]
sns.histplot(prod_comp_df_liquid_outliers_removed['Liquid'])
plt.title('Liquid Distribution With Outlier Removal (12 month)');
```



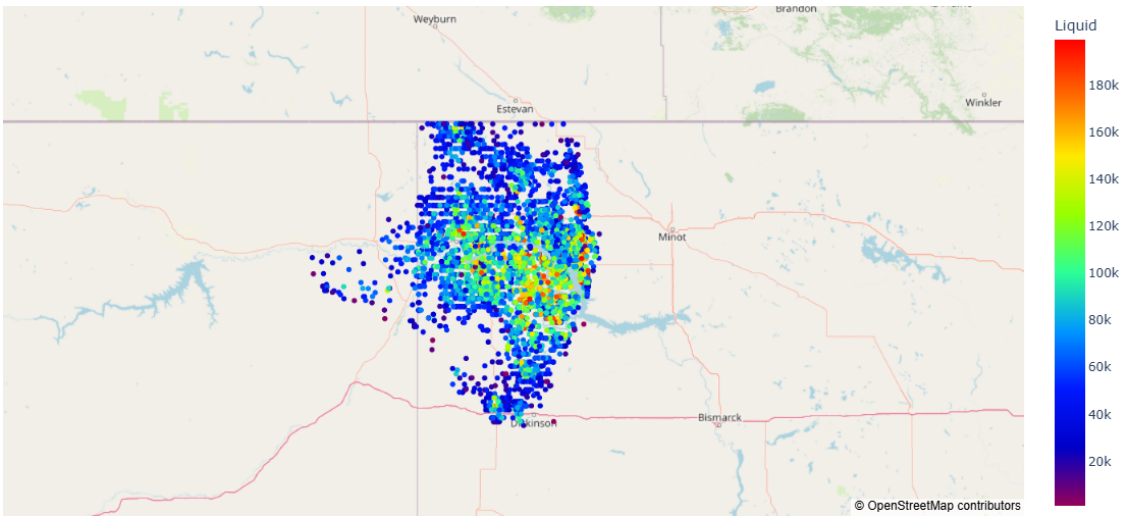
6 Important Observations

The results of the following tests have shown promising impact on model performance. - Selecting to predict on only the Bakken and not other formations mitigated the impact of unkown geology - Selecting to remove wells with low TVD, since they were unrepresented, increased accuracy - Selecting to remove high outlier wells increased accuracy - Selecting to segment based on lateral length increased accuracy

Lets visualize what liquid looks like after removing upper outliers.

[1366]:
```
target_hue = 'Liquid'

df_visualize_comp_3 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_3['API'] = prod_comp_df_liquid_outliers_removed['API']
```

```
df_visualize_comp_3['X'] = prod_comp_df_liquid_outliers_removed['Surface␣
↪Latitude']
df_visualize_comp_3['Y'] = prod_comp_df_liquid_outliers_removed['Surface␣
↪Longitude']
df_visualize_comp_3[target_hue] =␣
↪prod_comp_df_liquid_outliers_removed[target_hue]
create_location_map(df_visualize_comp_3, 'X', 'Y', target_hue, 6)
```



Observe the trends of production. We identify a distinct outer shelf region, where a lot of low production wells are. A middle inner region, and then a large production region centered around the river.

6.1 Adding in 'Past Depletion'

'Past Depletion,' which will be a measure of how much has been produced up until that point. It is to account for the geomechanical effects that past production may have.

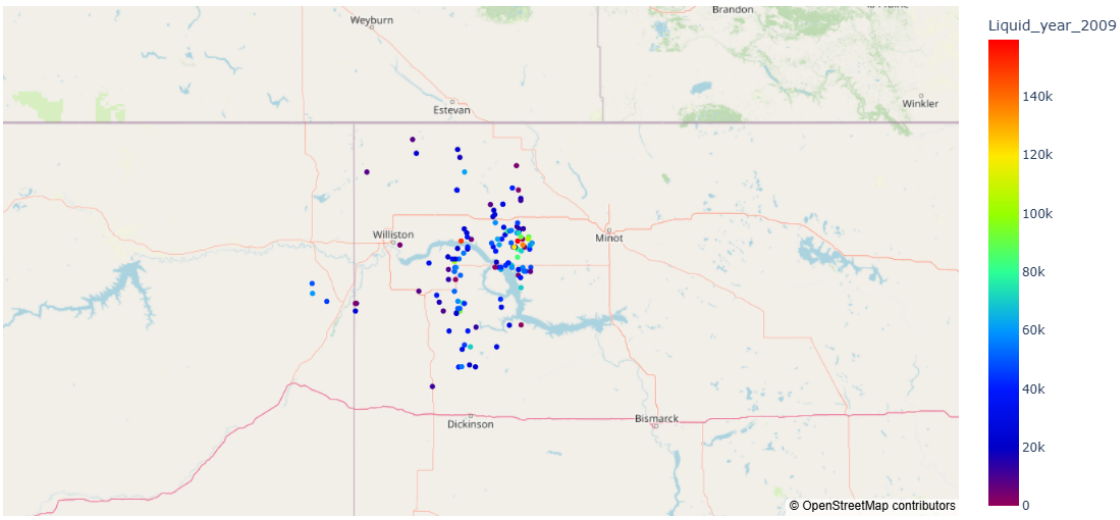
Here's the process for finding depletion numbers: 1) Calculate Haversine distance and index on each API, selecting to consider only wells that are within a certain distance for each API (may be computationaly intense) 2) Calculate the amount of months before production of this well started for all other wells. Also, calculate the mean GOR of the wells. 3) Add this information as a new row to pc_relations_comp_df

First, let's visualize production in the region by year. This will show us how much has been produced prior to new wells starting.

[1367]:
```
prod_in_2009 = create_well_data_year_b(2009)
prod_wells_in_2009 = pd.merge(prod_comp_df, prod_in_2009, on='API')
prod_wells_in_2009['Liquid_year_2009'] = prod_wells_in_2009['Liquid_year_2009'].
↪astype(float)
```

```
100%|
                    | 411/411 [00:04<00:00, 93.45it/s]
```

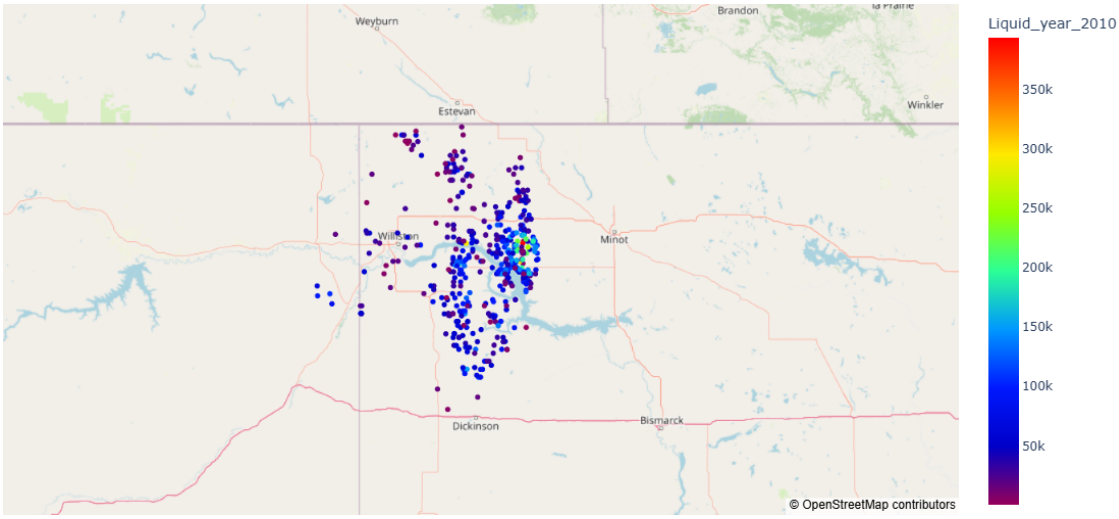
[1368]:
```
target_hue = 'Liquid_year_2009'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2009['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2009['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2009[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



[1369]:
```
prod_in_2010 = create_well_data_year_b(2010)
prod_wells_in_2010 = pd.merge(prod_comp_df, prod_in_2010, on='API')
prod_wells_in_2010['Liquid_year_2010'] = prod_wells_in_2010['Liquid_year_2010'].
 ↪astype(float)
```

```
100%|
                    | 1099/1099 [00:12<00:00, 84.75it/s]
```

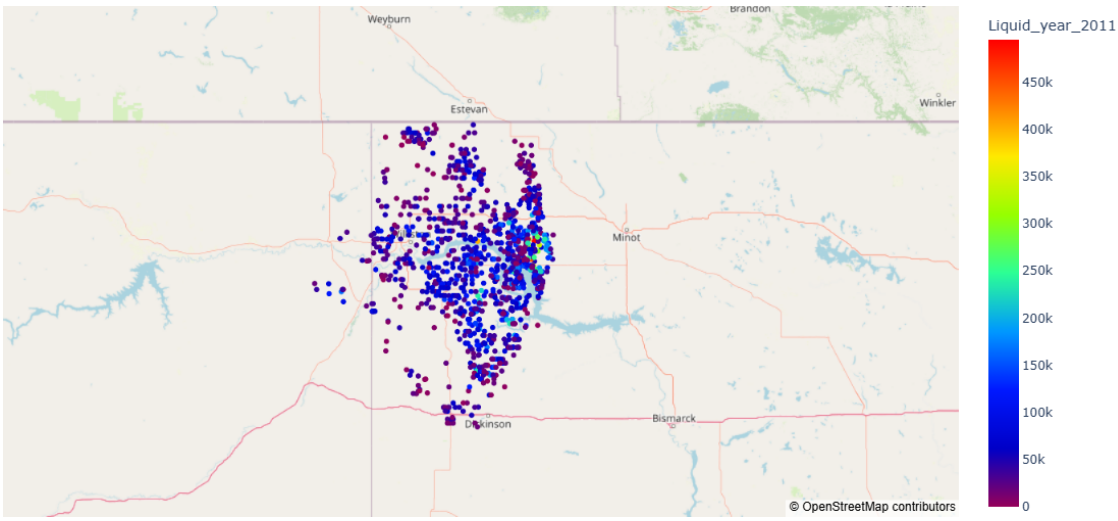
[1370]:
```
target_hue = 'Liquid_year_2010'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2010['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2010['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2010[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



[1371]:
```
prod_in_2011 = create_well_data_year_b(2011)
prod_wells_in_2011 = pd.merge(prod_comp_df, prod_in_2011, on='API')
prod_wells_in_2011['Liquid_year_2011'] = prod_wells_in_2011['Liquid_year_2011'].
↪astype(float)
```

```
100%|
                    | 2178/2178 [00:22<00:00, 96.34it/s]
```

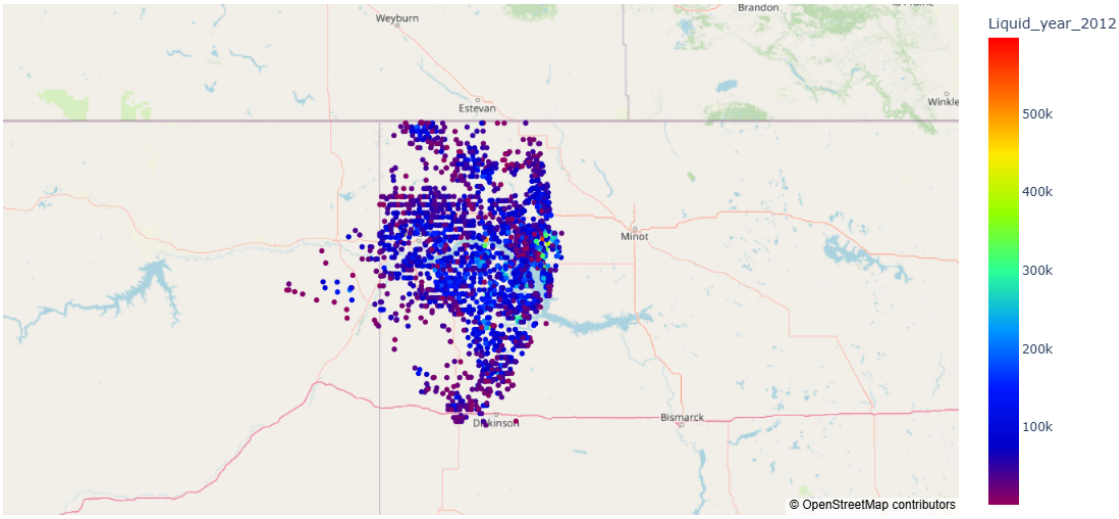
[1372]:
```
target_hue = 'Liquid_year_2011'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2011['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2011['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2011[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



[1373]:
```
prod_in_2012 = create_well_data_year_b(2012)
prod_wells_in_2012 = pd.merge(prod_comp_df, prod_in_2012, on='API')
prod_wells_in_2012['Liquid_year_2012'] = prod_wells_in_2012['Liquid_year_2012'].
↪astype(float)
```

```
100%|
                    | 3904/3904 [00:40<00:00, 97.55it/s]
```

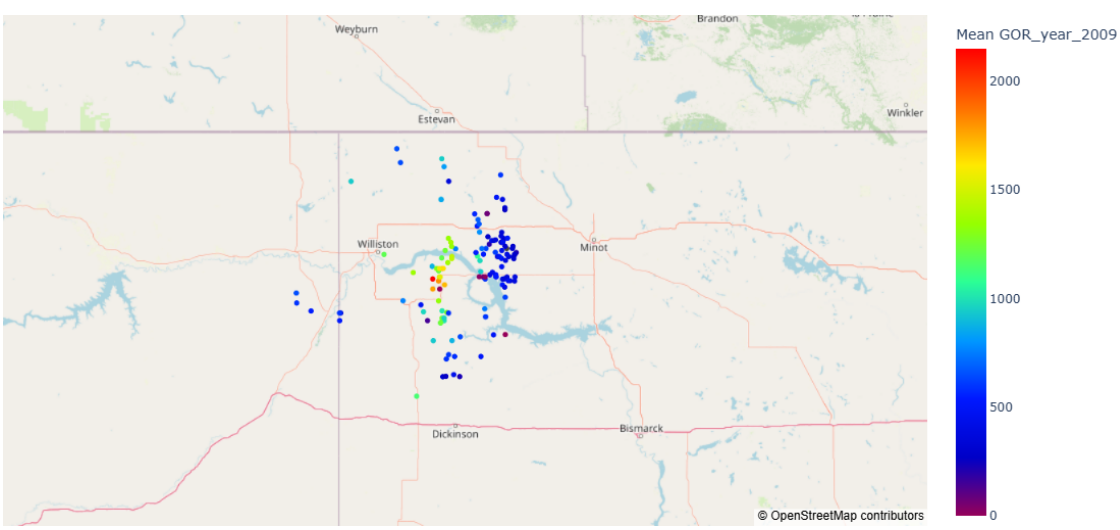
[1374]:
```
target_hue = 'Liquid_year_2012'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2012['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2012['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2012[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



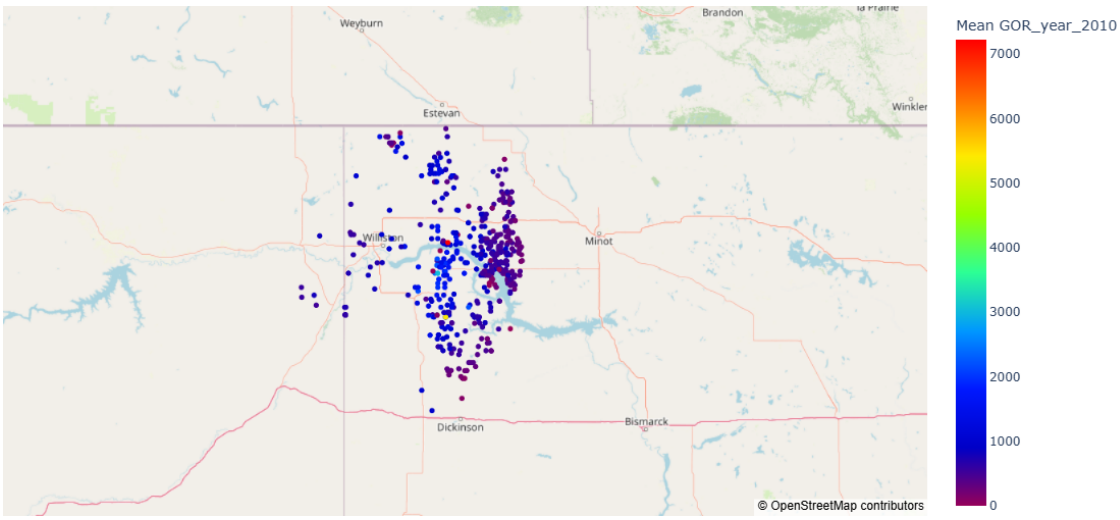
We observe the gradual onset of production over time. Imagine a well was to be drilled in 2013. It would be drilling against all production prior to 2013, which involves the production shown above. This has the chance of impacting production locally.

Let's also take a look at GOR to see how that changes over the years.

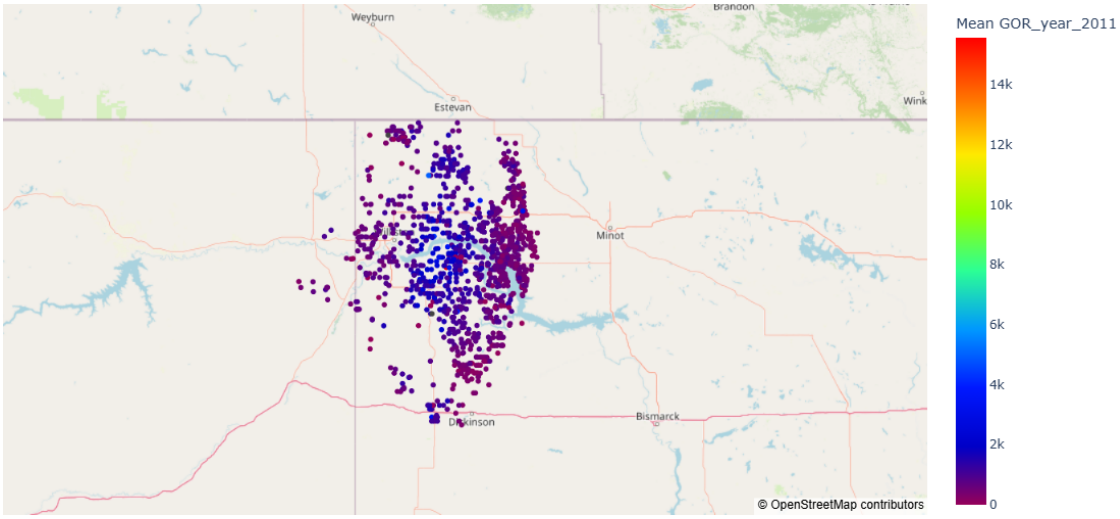
[1375]:
```
target_hue = 'Mean GOR_year_2009'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2009['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2009['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2009[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



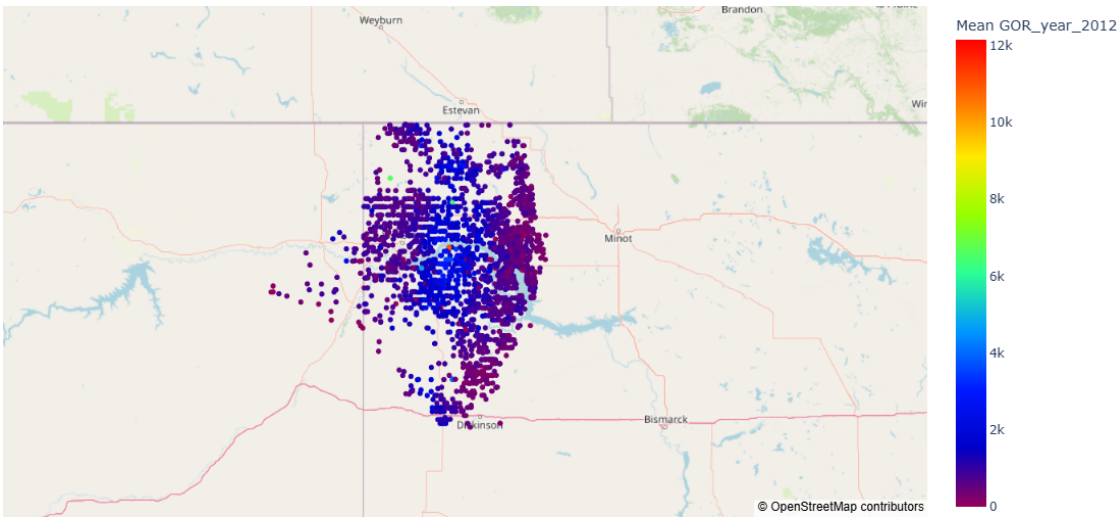
[1376]:
```
target_hue = 'Mean GOR_year_2010'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2010['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2010['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2010[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



[1377]:
```
target_hue = 'Mean GOR_year_2011'

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2011['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2011['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2011[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



[1378]:
```
target_hue = 'Mean GOR_year_2012'
prod_wells_in_2012 = prod_wells_in_2012[prod_wells_in_2012['Mean␣
↪GOR_year_2012'] < 15000]

df_visualize_comp_4 = pd.DataFrame(columns=['X', 'Y'])
df_visualize_comp_4['X'] = prod_wells_in_2012['Surface Latitude']
df_visualize_comp_4['Y'] = prod_wells_in_2012['Surface Longitude']
df_visualize_comp_4[target_hue] = prod_wells_in_2012[target_hue]
create_location_map(df_visualize_comp_4, 'X', 'Y', target_hue, 6)
```



We see a band of higher GOR in the middle of the region that develops over time. This data is in agreement with the fact that several vertical anticlines are present in the middle of the Bakken. Let's run the algorithm to find the previous production information for each well.

[1379]:
```
pc_use_comp_df = comp_df[['API', 'Surface Latitude', 'Surface Longitude']]
pc_relations_comp_df = pd.DataFrame(columns=['API', 'Total Liquid Before',␣
↪'Total Gas Before', 'Number of Neighbouring Wells Prior to Prod.'])
```

For the sake osf space, the calculation was calculated separately and saved as a dataframe, which will be imported here.

[1380]:
```
pc_relations_df = pd.read_csv('AMPP1_PCREL')
```

[1381]:
```
prod_comp_df = pd.merge(prod_comp_df, pc_relations_df, on='API')
```