```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import KNNImputer
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from scipy import stats
try:
    from statsmodels.stats.outliers_influence import variance_inflation_factor
    statsmodels_available = True
except ImportError:
    statsmodels_available = False
import warnings
warnings.filterwarnings('ignore')

# Set random seed for reproducibility
np.random.seed(42)

# 1. Load the dataset
def load_data(url):
    try:
        df = pd.read_csv (url)
        print("Data loaded successfully.")
        print("\nDataset Column Types:")
        print(df.dtypes)
        return df
    except Exception as e:
        print(f"Error loading data: {e}")
        return None

url =
"https://raw.githubusercontent.com/ageron/handson-ml2/master/datasets/housing/housing.csv"
df = load_data(url)
if df is None:
    raise SystemExit("Exiting due to data loading failure.")
```

```
# Subsample dataset for faster training (comment out to use full dataset)
df = df.sample(frac=0.3, random_state=42)
print(f"\nSubsampled dataset to {len(df)} rows for faster training.")

# 2. Data Preprocessing
def preprocess_data(df):
    print("\n=== Data Preprocessing ===")

    # Display missing values before imputation
    print("Missing Values Before Imputation:")
    print(df.isnull().sum())

    # Separate categorical and numerical columns
    categorical_cols = ['ocean_proximity'] if 'ocean_proximity' in df.columns else []
    numerical_cols = [col for col in df.columns if col not in categorical_cols]

    # Debug: Confirm columns
    print("\nNumerical Columns for Imputation:", numerical_cols)
    print("Categorical Columns:", categorical_cols)

    # Handle missing values with KNN imputation for numerical columns
    try:
        if numerical_cols:
            imputer = KNNImputer(n_neighbors=5)
            df[numerical_cols] = pd.DataFrame(
                imputer.fit_transform(df[numerical_cols]),
                columns=numerical_cols,
                index=df.index
            )
    except Exception as e:
        print(f"Error during KNN imputation: {e}")
        print("Falling back to median imputation.")
        for col in numerical_cols:
            df[col].fillna(df[col].median(), inplace=True)

    # Display missing values after imputation
    print("\nMissing Values After Imputation:")
    print(df.isnull().sum())

    # Remove duplicates
    initial_rows = len(df)
    df.drop_duplicates(inplace=True)
```

```
    print(f"\nDuplicates Removed: {initial_rows - len(df)}")

    # Cap outliers for numerical columns and track counts
    outlier_counts = {}
    def cap_outliers(series, col_name):
        Q1 = series.quantile(0.25)
        Q3 = series.quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        outliers = ((series < lower_bound) | (series > upper_bound)).sum()
        outlier_counts[col_name] = outliers
        return series.clip(lower_bound, upper_bound)

    for col in numerical_cols:
        df[col] = cap_outliers(df[col], col)

    print("\nOutliers Detected and Capped:")
    for col, count in outlier_counts.items():
        print(f"{col}: {count} outliers")

    # Display skewness before log-transformation
    print("\nSkewness Before Log-Transformation:")
    print(df[numerical_cols].skew())

    # Log-transform skewed features
    skewed_cols = ['total_rooms', 'population', 'median_house_value']
    for col in skewed_cols:
        if col in df.columns:
            df[col] = np.log1p(df[col])

    # Display skewness after log-transformation
    print("\nSkewness After Log-Transformation:")
    print(df[numerical_cols].skew())

    # Verify median_house_value presence
    print("\nColumns After Preprocessing:", df.columns.tolist())

    return df

df = preprocess_data(df)

# 3. Feature Engineering
```

```
def engineer_features(df):
    print("\n=== Feature Engineering ===")
    # Add core features (reduced set)
    df['rooms_per_household'] = df['total_rooms'] / df['households']
    df['bedrooms_per_room'] = df['total_bedrooms'] / df['total_rooms']
    df['population_per_household'] = df['population'] / df['households']
    # Removed 'distance_to_coast' and 'median_income_poly1' to reduce feature count

    # Summarize new features
    print("New Features Created:")
    print(df[['rooms_per_household', 'bedrooms_per_room', 'population_per_household']].head())

    return df

df = engineer_features(df)

# 4. Statistical Analysis
def statistical_analysis(df):
    print("\n=== Statistical Analysis ===")
    stat, p_value = stats.shapiro(df['median_house_value'])
    normality_result = f"Shapiro-Wilk Test for median_house_value: p-value = {p_value:.4f}"

    # Filter numerical columns for descriptive stats, skewness, and kurtosis
    numerical_cols = df.select_dtypes(include=['float64', 'int64']).columns
    print("\nNumerical Columns for Statistical Analysis:", numerical_cols.tolist())

    desc_stats = df[numerical_cols].describe().T
    desc_stats['skewness'] = df[numerical_cols].skew()
    desc_stats['kurtosis'] = df[numerical_cols].kurtosis()

    # Multcollinearity analysis (VIF)
    if statsmodels_available:
        numerical_df = df[numerical_cols]
        vif_data = pd.DataFrame()
        vif_data['Feature'] = numerical_df.columns
        vif_data['VIF'] = [variance_inflation_factor(numerical_df.values, i) for i in
range(numerical_df.shape[1])]
        print("\nVariance Inflation Factor (VIF) Analysis:")
        print(vif_data)

    return normality_result, desc_stats

normality_result, desc_stats = statistical_analysis(df)
```

```
print("\nNormality Test:")
print(normality_result)
print("\nDescriptive Statistics with Skewness and Kurtosis:")
print(desc_stats)

# 5. Exploratory Data Analysis (EDA)
def perform_eda(df):
    print("\n=== Exploratory Data Analysis ===")
    # Correlation matrix
    corr_matrix = df.select_dtypes(include=['float64', 'int64']).corr()
    fig_corr = go.Figure(data=go.Heatmap(
        z=corr_matrix.values,
        x=corr_matrix.columns,
        y=corr_matrix.columns,
        colorscale='RdBu',
        zmin=-1, zmax=1,
        text=corr_matrix.values.round(2),
        texttemplate="%{text}",
        textfont={"size": 10}
    ))
    fig_corr.update_layout(title='Interactive Correlation Matrix', width=800, height=800)
    fig_corr.show()

    # Scatter plot (fixed syntax error)
    fig_scatter = px.scatter(df, x='median_income', y='median_house_value', title='Median
Income vs Median House Value (USD)',
                            hover_data=['longitude', 'latitude'],
                            trendline='ols')
    fig_scatter.update_yaxes(title_text='Median House Value (USD)')
    fig_scatter.update_traces(hovertemplate='Income: %{x}House Value: $%{y:,.2f} USD')
    fig_scatter.show()

    # Geographical plot (requires Mapbox token)
    px.set_mapbox_access_token('your_mapbox_token')
    fig_geo = px.scatter_mapbox(df, lat='latitude', lon='longitude', color='median_house_value',
                                size='population', zoom=5, mapbox_style='open-street-map',
                                title='House Prices by Location (USD)',
                                color_continuous_scale=px.colors.sequential.Plasma)
    fig_geo.update_coloraxes(colorbar_title='Median House Value (USD)')
    fig_geo.show()

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

```
    sns.histplot(df['median_house_value'], kde=True)
    plt.title('Distribution of Median House Value (USD)')
    plt.xlabel('Median House Value (USD)')
    plt.show()

perform_eda(df)

# 6. Prepare Data for Modeling
def prepare_data(df):
    X = df.drop('median_house_value', axis=1)
    y = df['median_house_value']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = prepare_data(df)

# 7. Modeling and Impact Analysis
def build_pipeline(model):
    numerical_cols = X_train.select_dtypes(include=['float64', 'int64']).columns
    categorical_cols = ['ocean_proximity'] if 'ocean_proximity' in df.columns else []

    preprocessor = ColumnTransformer([
        ('num', StandardScaler(), numerical_cols),
        ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), categorical_cols)
    ])

    pipeline = Pipeline([
        ('preprocessor', preprocessor),
        ('regressor', model)
    ])

    return pipeline

def plot_feature_importance(model, feature_names):
    importances = model.feature_importances_
    indices = np.argsort(importances)[::-1]
    plt.figure(figsize=(10, 6))
    plt.bar(range(len(importances)), importances[indices], align='center')
    plt.xticks(range(len(importances)), [feature_names[i] for i in indices], rotation=90)
    plt.title('Feature Importance')
    plt.tight_layout()
    plt.show()
```

```
def plot_learning_curves(model, X_train, y_train):
    train_sizes, train_scores, val_scores = learning_curve(
        model, X_train, y_train, cv=3, scoring='r2', n_jobs=-1, train_sizes=np.linspace(0.1, 1.0, 5))
    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_mean, label='Training R²')
    plt.plot(train_sizes, val_mean, label='Validation R²')
    plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1)
    plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1)
    plt.xlabel('Training Examples')
    plt.ylabel('R² Score')
    plt.title('Learning Curves')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

def train_and_evaluate(X_train, X_test, y_train, y_test):
    print("\n=== Model Training and Evaluation ===")

    # Define models
    models = {
        'Linear Regression': LinearRegression(),
        'Random Forest': RandomForestRegressor(n_estimators=50, max_depth=10,
random_state=42, n_jobs=-1),
        'Gradient Boosting': GradientBoostingRegressor(n_estimators=50, random_state=42)
    }

    results = []
    best_model = None
    best_r2 = -np.inf

    numerical_cols = X_train.select_dtypes(include=['float64', 'int64']).columns
    categorical_cols = ['ocean_proximity'] if 'ocean_proximity' in df.columns else []
    feature_names = numerical_cols.tolist() + [f"ocean_proximity_{cat}" for cat in
OneHotEncoder(drop='first').fit(X_train[categorical_cols]).get_feature_names_out()] if
categorical_cols else numerical_cols.tolist()

    for name, model in models.items():
        print(f"\nTraining {name}...")
```

```
pipeline = build_pipeline(model)
pipeline.fit(X_train, y_train)

# Predictions
y_pred = pipeline.predict(X_test)

# Evaluation metrics (convert back to USD)
y_test_usd = np.expm1(y_test)
y_pred_usd = np.expm1(y_pred)
mae = mean_absolute_error(y_test_usd, y_pred_usd)
rmse = np.sqrt(mean_squared_error(y_test_usd, y_pred_usd))
r2 = r2_score(y_test_usd, y_pred_usd)

results.append({
    'Model': name,
    'MAE (USD)': mae,
    'RMSE (USD)': rmse,
    'R²': r2
})

print(f"\n{name} Performance (in USD):")
print(f"Mean Absolute Error: ${mae:,.2f}")
print(f"Root Mean Squared Error: ${rmse:,.2f}")
print(f"R²: {r2:.4f}")

# Feature importance (for Random Forest and Gradient Boosting)
if name in ['Random Forest', 'Gradient Boosting']:
    plot_feature_importance(pipeline.named_steps['regressor'], feature_names)

# Learning curves
plot_learning_curves(pipeline, X_train, y_train)

# Residual analysis
residuals = y_test_usd - y_pred_usd
plt.figure(figsize=(10, 6))
stats.probplot(residuals, dist="norm", plot=plt)
plt.title(f'Q-Q Plot of Residuals ({name})')
plt.show()

# Update best model
if r2 > best_r2:
    best_r2 = r2
    best_model = pipeline
```

```
    # Display model comparison
    print("\n=== Model Comparison ===")
    results_df = pd.DataFrame(results)
    print(results_df)

    return best_model

best_model = train_and_evaluate(X_train, X_test, y_train, y_test)

# 8. Gradio Interface for Hugging Face Spaces
try:
    import gradio as gr

    def predict_house_value(longitude, latitude, housing_median_age, total_rooms,
total_bedrooms,
                            population, households, median_income, ocean_proximity):
        input_data = pd.DataFrame({
            'longitude': [longitude],
            'latitude': [latitude],
            'housing_median_age': [housing_median_age],
            'total_rooms': [np.log1p(total_rooms)],
            'total_bedrooms': [total_bedrooms],
            'population': [np.log1p(population)],
            'households': [households],
            'median_income': [median_income],
            'ocean_proximity': [ocean_proximity],
            'rooms_per_household': [np.log1p(total_rooms) / households],
            'bedrooms_per_room': [total_bedrooms / np.log1p(total_rooms)],
            'population_per_household': [np.log1p(population) / households]
        })
        prediction = best_model.predict(input_data)
        usd_value = np.expm1(prediction[0])  # Reverse log transformation
        return f"Predicted House Value: ${usd_value:,.2f} USD"

    iface = gr.Interface(
        fn=predict_house_value,
        inputs=[
            gr.Slider(-124, -114, step=0.1, label="Longitude"),
            gr.Slider(32, 42, step=0.1, label="Latitude"),
            gr.Slider(0, 52, step=1, label="Housing Median Age"),
            gr.Slider(0, 40000, step=100, label="Total Rooms"),
            gr.Slider(0, 7000, step=10, label="Total Bedrooms"),
```

```
            gr.Slider(0, 50000, step=100, label="Population"),
            gr.Slider(0, 7000, step=10, label="Households"),
            gr.Slider(0, 15, step=0.1, label="Median Income"),
            gr.Dropdown(choices=['<1H OCEAN', 'INLAND', 'NEAR OCEAN', 'NEAR BAY',
'ISLAND'], label="Ocean Proximity")
        ],
        outputs="text",
        title="California House Price Predictor",
        description="Enter features to predict the median house value (in USD)."
    )

    print("\nGradio interface is ready. Run iface.launch() in a Hugging Face Space to use it.")
except ImportError:
    print("\nGradio not installed. Skipping Gradio interface. Install gradio to enable.")
```