Using Linear Regression and Machine Learning Techniques to Predict Housing Prices Based on Economic Factors

Except where reference is made to the work of others, the work described in this project is my own or was done in collaboration with my advisory committee. Further, the content of this project is truthful in regards to my own work and the portrayal of others' work. This project does not include proprietary or classified information.

Cassandra Carpenter

Your Name Here

Certificate of Approval:

Michael J. Reale, Ph.D., Chair
Associate Professor
Department of Computer & Information Science

Scott Spetka, Ph.D.
Professor
Department of Computer & Information Science

Jorge Novillo, Ph.D.
Professor
Department of Computer & Information Science

Cassandra Companion

Using Linear Regression and Machine Learning Techniques to Predict Housing Prices Based on Economic Factors

Submitted to the Graduate Faculty of the State University of New York Polytechnic Institute

in Partial Fulfillment of the Requirements for the Degree of Master of Science

Utica, New York

January 13, 2023

Cassandra Companion

Signature of Author

1/13/23

Date of Graduation

Cassandra Companion

Master of Science, January 13, 2023

(B.S., SUNY Polytechnic Institute, 2021)

Directed by Michael J. Reale

Abstract

When trying to predict housing prices, most studies rely on data from a specific area, and the features of the homes there. In this study, the goal is to use linear regression and machine learning to predict housing prices based on overarching economic factors. A mix of machine learning and linear regression was used, including TensorFlow Keras, OLS, Ridge, Lasso, Elastic Net, XGBoost, Random Forest and SVM. Datasets featured include Average Sales Price of House Sold for the United States, closing stock prices (NASDAQ, S&P), 30-year mortgage interest rates, average monthly rent, number of houses sold, number of houses constructed, mean family income, median family income, GDP, and unemployment rate.

Table of Contents

Chapter 1: Introduction

Housing price prediction is a fairly common machine learning problem. Most of these focus on the features of homes in a specific area, such as square footage and number of baths. However, there have not been many studies on how overarching economic factors affect prices, so that will be the focus of this project. Also, the various methods and models typically used in price prediction will be used to determine which one performs the best for problems such as these. The approaches being tested are: a Keras dense Layer model, Ordinary Least Squares linear regression, several linear regression variants (Ridge, Lasso, Elastic Net), XGBoost, Random Forest and SVM.

This project's biggest challenge was to actually find data that could be used. Most datasets available online either 1) were not up to date (ending in 2016, for example), 2) did not have enough data (only go back five years or so), or 3) were not able to be easily converted into a file format usable for machine learning. That is why the majority of the datasets used were found in the same place, and why they have such varied time frequencies. Additionally, I am not an economist, so I had to research what kinds of factors can affect the prices of houses. Articles from Economics Help [12] and PVS Builders [13] were used as a starting point when gathering ideas about what should be included. The data used can fall into one of two categories: factors that affect the overall economy (GDP, unemployment, etc.) and factors that I thought would affect housing prices in particular (interest rates on mortgages, total houses sold, total houses constructed, etc.). Specifics on the data will be discussed in Chapter 4.

Chapter 2: Related Works

An interesting problem faced with this project was in most cases, when a group wants to predict housing prices, they look at more of a specific area and at features of the property instead

of trying to find the effect of overall economic trends. Still, these other projects can provide some insight on potential machine learning strategies.

Gupta et al. [3] used various linear regression models such as OLS, Ridge, Lasso, and XGBoost to predict house prices based on features of the homes in the area of Bengaluru such as price, size, total square footage and number of baths. They used R^2, RMSE, and RMSLE as metrics. Between the OLS, Ridge, Lasso, and XGBoost models, they had an unnormalized RMSE of 0.2077, 0.5224, 0.5224, 0.3462 and R-Squared of -2.12, 0.4358, 0.4430, 0.7584 respectively.

Ho et al [4] used a support vector machine (SVM), random forest model, and gradient boosting to predict property prices in the housing district of Hong Kong with factors such as floor area, age of property, floor level and accessibility of the building. They used R^2, cross-fold validation, MSE, RMSE, and MAPE for validation. Using random forest, their model had an R-Squared as high as 0.89690 test set, and the MSE, RMSE and MAPE are 0.00848, 0.09210 and 0.33348, respectively.

Truong et al. [5] used the housing price index (HPI), area, population and location to determine house prices using Random Forest, Stacked Generalization and a hybrid of linear regression models Lasso and XGBoost. They only used RMSLE to validate the results.

Similar machine learning methods can be used for similar problems. Therefore, I decided to also look into strategies used in other types of price prediction studies. Mehtab and Sen [1] wanted to predict using daily stock price data of a very well-known company that is listed in the National Stock Exchange (NSE) of India. Their variables included date, time, open price, high value, low value, closing price and the volume of the stock traded in a given interval. Similar to this project, they used a neural network, decision trees, random forest, boosting and SVMs.

Additionally, they implemented bagging, K-nearest neighbor, logistic regression and multivariate regression.

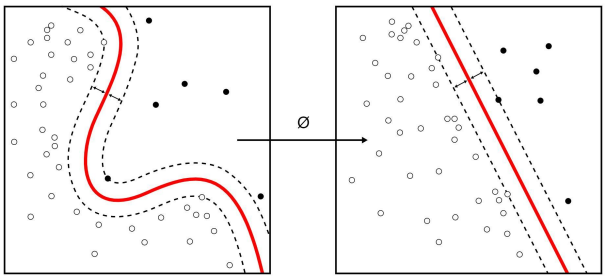
Chapter 3: Method

3.1: Background

3.1.1: Linear Regression

In this particular case, we will be looking at the built in Sci-Kit Learn (also known as Sklearn) linear regression package. Sklearn uses Ordinary Least Squares Regression (OLS) in linear regression, but also includes several variations with different regularization, including:

(Figure 3.1) Example of regularizers [14]



- Ridge: Linear least squares with L2 regularization [6]
- Lasso: Linear Model trained with L1 prior as regularizer (aka the "lasso") [7]
- Elastic Net: Linear regression with combined L1 and L2 priors as regularizer [8]

Regularizers are used in models that have datasets that do not exhibit linear relationships between the independent and the dependent variables. These problems often need a polynomial model, but such models are prone to overfitting [9]. Regularizers make the data more "simple" so it can be more easily worked with by adding bias to data points to shift them. The difference between L1 and L2 regularizers is how they add bias, or "penalty", to the data. The L1 penalty is the absolute value of the magnitude of coefficient, and L2 is the square of the magnitude of coefficients.

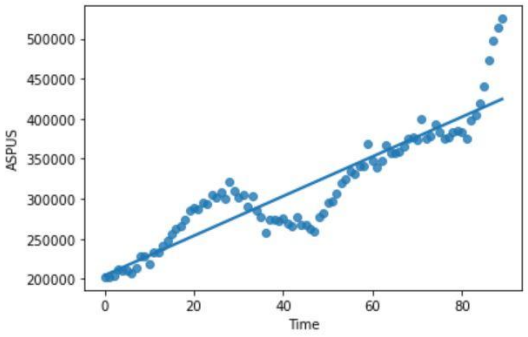
According to the Sklearn documentation, the linear regression library fits a linear model with coefficients $w = (w_1, \ldots, w_p)$ to minimize the residual sum of squares between the observed

targets in the dataset, and the targets predicted by the linear approximation. Assuming the input values are represented by $\{x_1, x_2, x_3 \ldots x_p\}$ and the output is y_i, the regression formula can be written as

$$y_i = w_1 x_1 + w_2 x_2 + \ldots + w_p x_{ip} + \varepsilon_i$$

with ε representing the unobserved random variables (errors) of the i-th observation. Basically, if all the data points were to be graphed, the model is trying to find the linear line that best fits the data. For example, the linear regression graph of just the housing price data would look like Figure 3.1.

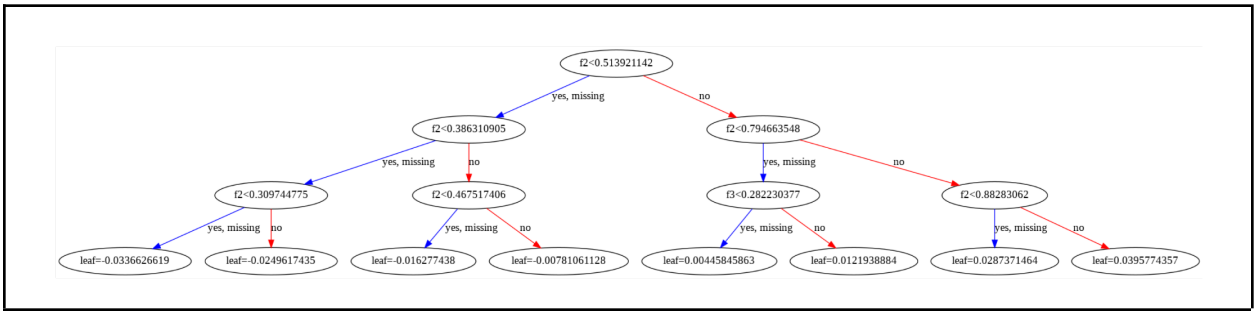
(Figure 3.2) Linear regression of the ASPUS housing price data



3.1.2: XGBoost

Extreme Gradient Boosting (abbreviated to XGBoost) is a distributed and scalable decision tree machine learning library that provides parallel tree boosting. Decision trees are models that evaluate a tree of true/false questions based on the features. For a visual representation of this, see Figure 3.3, depicting the first fold of the unadjusted XGBoost experiment in Section 4.2.6. A tree with gradient boosting has the added benefit of an ensemble learning algorithm, which combines multiple machine learning algorithms to get a better result. In particular, XGBoost uses a Newton-Raphson root-finding function.

Figure 3.3: Tree depicting the first fold of the unadjusted XGBoost experiment



The following is a generic example as to how it works with a training set $(\{xi,\ yi\})_{i=1}^{N}$ and loss function $L(y,F(x))$, number of weak learners M and learning rate α. First the model is initialized with a constant value:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Then, for $m = 1$ to M, compute the gradients and hessians:

$$\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}_{(m-1)}(x)}.$$
$$\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2}\right]_{f(x)=\hat{f}_{(m-1)}(x)}.$$

Next, fit the base learner/tree for the training set using an optimization problem:

$$\hat{\phi}_m = \arg\min_{\phi\in\Phi} \sum_{i=1}^{N} \frac{1}{2}\hat{h}_m(x_i)\left[-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)} - \phi(x_i)\right]^2.$$
$$\hat{f}_m(x) = \alpha\hat{\phi}_m(x).$$

Update the model:

$$f_{(m)}(x) = f_{(m-1)}(x) + f_m(x).$$

Output:

$$f(x) = f_{(M)}(x) = \sum_{m=0}^{M} f_m(x)$$

It also provides both L1 and L2 boosting on each leaf. XGBoost's structure allows it to create models with more stability and less variance, at a higher execution speed [3].

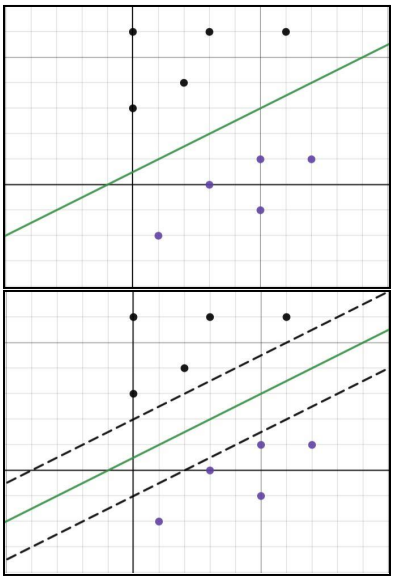
3.1.3: SVM

A Support Vector Machine (SVM) is a linear classifier similar to a perceptron, which uses binary sorting. They are more commonly used in classification problems, but can also be utilized for regression. It uses weights to determine the importance of each input value *x*:

$y = x_1w_1+x_2w_2+...+x_iw_i$

and outputs a value that is either above or below a desired threshold, such as y < 1.5 = "True" and y ≥ 1.5 = "False". This can be used to find a hyperplane and suitable margins in the data (figure 3.4), if it exists. SVM aims to find a hyperplane with the largest possible margin, so that the distance of the plane to the closest data point in both classes is maximized. In regression, SVMs can be used by mapping the input onto a m-dimensional feature space employing nonlinear mapping, to construct a linear model [4].

(Figure 3.4) Two examples of hyperplanes, first without margins and second with margins



3.1.4: Random Forest

Random forest uses ensemble learning for regression and classification. It creates many decision trees at training time, to make a *forest*. It then combines them using the mean of the individual trees to create one model. The trees are built by taking a random number of attributes for each node [10]. The number of trees is determined by the value *n_estimators*; in this project it was set to 100. The randomness is supposed to decrease the likelihood of overfitting.

3.1.5: Keras Neural Networks

Keras is a library in TensorFlow, a free and open-source software library for machine learning and artificial intelligence. It is an API that makes creating a machine learning model

easier by increasing readability and simplifying steps the user needs to take. In this case, the Sequential model was used as it was the simplest to implement. All the user needs to do is create a list of dense layers, but this is at the expense of only being able to read one frame at a time. This wasn't a major problem for this project as there are not that many frames (less than 500), but for a larger model TensorFlow recommends the Model API instead.

The Sequential class stacks all the layers into a *tf.keras.Model*, and the *Model* groups layers into an object with training and inference features. *Model* takes the input(s) and the output of the model, and *Sequential* then provides training and inference features for the model. In machine learning, a *layer* is a group of processing nodes. In a dense layer model, the layers' nodes are deeply connected to the previous layer as all the neurons of the layer are connected to every neuron of its preceding layer.

After the inputs (x) and output (y) are declared split into training and testing sets for later, the dense layers are added. Dense layers have several parameters that can be used in certain situations to better fit the problem but in our case, we are only worried about three features. First is the units. This is the dimensionality of the output space, or how many nodes should be returned in the output. Second is the activation function, which determines the output of a node. There are a wide range of options, and for this project Scaled Exponential Linear Unit (SELU) was utilized:

$$\lambda \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

With $\lambda = 1.0507$ and $\alpha = 1.67326$. This is a modification to the Rectified linear unit (RELU) function:

$$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$
$$= \max(0, x) = x\mathbf{1}_{x>0}$$

Not only does SELU converge faster due to its ability to handle output below zero, but is best used in models composed of many stacked dense layers [2]. Finally, we have the *input shape*, used in the first dense layer. Inputs are formatted as arrays, and the input shape is the dimension of that array. For example, (20, 30) would be a 2-dimensional array, and (20, 10, 30) would be a 3-dimensional array. This model features a shape that is a one-dimensional array that is the size of the length of the features. When compiling the model, an optimizer needs to be chosen. Optimizers find the loss function, which is used to find the ideal weights for the model. This model uses *adam*, or Adaptive Moment estimation, a gradient descent method designed for optimization. Adam is the most commonly used optimizer in machine learning; unless the task is very specific, it will be fine as an optimizer for most projects [11].

Next the model must be fitted. “Fitting” is when the model tries to find the relationship between the input and output variables in the training data and creates an equation that can guess the output of newly given data. It runs through the various layers for multiple cycles known as *epochs*, trying to learn more each time. The model tests its equation on the test data, which can be used for comparison.

Chapter 4: Data, Experiments and Results

4.1: Data

The data used in this project was retrieved from various sources. See table below for information on where it initially came from and where it was retrieved from. It was decided that the data would be cut off at 2020 due to that being the start of the Coronavirus pandemic. It had a severe effect on the economy, creating outliers in several datasets and making analysis difficult.

Table 4.1: Datasets Utilized

Dataset	Retrieved From	Initial Source of Data	Frequency
Average Sales Price of House Sold for the United States (ASPUS) [15]	Federal Reserve Economic Data (FRED)	U.S. Census Bureau, U.S. Department of Housing and Urban Development	Quarterly
ASPUS (adjusted for inflation)	Data was adjusted using a Python script, based on the data in the linearly interpolated ASPUS data	U.S. Census Bureau, U.S. Department of Housing and Urban Development	Monthly
Historical NASDAQ data [16]	Yahoo! Finance	ICE Data Services	Monthly
Historical S&P 500 data [17]	Yahoo! Finance	ICE Data Services	Monthly
30-Year mortgage interest rates [18]	Federal Reserve Economic Data (FRED)	Freddie Mac Primary Mortgage Market Survey	Monthly
Median monthly rent prices [19]	iProperty Management	U.S. Bureau of Labor, U.S. Census Bureau	Mixed
Houses sold [20]	U.S. Census Bureau	U.S. Census Bureau	Monthly
New privately owned houses completed [21]	U.S. Census Bureau	U.S. Census Bureau	Monthly
Mean household income [22]	Federal Reserve Economic Data (FRED)	U.S. Census Bureau	Annual
Median household income [23]	Federal Reserve Economic Data (FRED)	U.S. Census Bureau	Annual
Unemployment rate [24]	Federal Reserve Economic Data (FRED)	U.S. Bureau of Labor Statistics	Monthly
Gross Domestic Product [25]	Federal Reserve Economic Data (FRED)	U.S. Bureau of Economic Analysis	Quarterly

The Average Sales Price of House Sold for the United States (ASPUS) is a quarterly dataset derived from data given by the U.S. Census Bureau and U.S. Department of Housing and Urban Development. It was retrieved from Federal Reserve Economic Data (FRED), an online

database consisting of hundreds of thousands of economic data time series from scores of national, international, public, and private sources, and is maintained by the Research Department at the Federal Reserve Bank of St. Louis. In order for the data to be monthly, it was linearly interpolated using Pandas. A version adjusted for inflation was created by using a script utilizing the CPI Python library.

The historical NASDAQ and S&P stock price data was retrieved from Yahoo! Finance. Yahoo! claims that the data was provided by ICE Data Services. It was already monthly, so no changes had to be made. Stock prices were included as part of the data because they are often said to be economic indicators.

The 30-year mortgage interest rates are from the Freddie Mac Primary Mortgage Market Survey, and was found on FRED. Freddie Mac (as known as Federal Home Loan Mortgage Corporation [FHLMC]) is a government-sponsored enterprise that purchases mortgages from banks and other loan makers to “promote stability” [10]. 30-year was chosen instead of 5- or 15-year mortgages because they are the most common and most accessible to the average American.

Median monthly rent prices were found on iProperty, a website that provides research and tools for property owners and tenants. The data was sourced from the U.S. Bureau of Labor and U.S. Census Bureau. An oddity with this data was that it had a variable time frequency. 2023 through 2010 is annual, and 2010 through 1970 is every five years. Linear interpolation was used to make it monthly.

Total houses sold and new privately owned houses completed both come from the U.S. Census Bureau. These were chosen because they help show supply (new constructions) and demand (total houses sold) in the housing industry.

Mean and median income are both good economic indicators. The more money someone has, the more they purchase, and more capital movement benefits the economy. These were retrieved from FRED and originally sourced from the U.S. Census Bureau.

Unemployment rate is also a good economic indicator for the same reason mean and median income is. People without jobs often cannot afford to make purchases or pay their bills, making home ownership and paying rent difficult. This data was found on FRED, and sourced from the U.S. Bureau of Labor Statistics.

To validate the data, n-fold validation was used. First, the data is split into equally-sized folds. Since the years range from 1985 to 2020, the data was split into 12 folds of three years each. For each fold, a group is taken out as the test data, and the rest is used for training data. Fit the model as usual, and repeat for each fold. This technique can provide a more accurate estimate of out-of-sample accuracy and can be a more efficient use of data as every observation is used for both training and testing. Root mean squared error (RMSE) and R-Squared values are used to judge the effectiveness of each model.

Due to the wide variation in values, the data must be normalized to avoid values being out of range. To do this the following formula is used for both ‘x’ (the features) and ‘y’ (the data being predicted):

```
x = (x- x.min())/ (x.max() - x.min())
y = (y- y.min())/ (y.max() - y.min())
```

See tables below for the mean, median, minimum and maximum of the housing prices of each fold, both original (ASPUS) and adjusted for inflation.

Table 4.2: ASPUS Statistics

	Mean	Median	Max	Min
Total	240111.086	233100	418600	98500

Fold 1 (1985/1986/1987)	114586.1111	114000	136433.3333	98500
Fold 2 (1988/1989/1990)	145650	147216.6667	151200	134800
Fold 3 (1991/1992/1993)	146544.4444	146216.6667	151833.3333	141700
Fold 4 (1994/1995/1996)	159666.6667	158633.3334	171800	152800
Fold 5 (1997/1998/1999)	184419.4444	181966.6667	204800	172200
Fold 6 (2000/2001/2002)	215213.8889	211550	232900	202400
Fold 7 (2003/2004/2005)	271322.2222	272550	301600	233100
Fold 8 (2006/2007/2008)	299583.3333	303116.6667	322100	263533.3333
Fold 9 (2009/2010/2011)	269241.6667	269400	278000	257000
Fold 10 (2012/2013/2014)	320386.1111	323733.3334	369400	278000
Fold 11 (2015/2016/2017)	364488.8889	361850	399700	339700
Fold 12 (2018/2019/2020)	385272.2222	383116.6667	413700	374500

Table 4.3: Adjusted Prices Statistics

	Mean	Median	Max	Min
Total	342230.52	331708.0261	441851.7828	248053.3922
Fold 1 (1985/1986/1987)	281108.4911	281848.3577	325434.3339	248053.3922
Fold 2 (1988/1989/1990)	317593.9073	316507.9421	330408.5806	301653.6725
Fold 3 (1991/1992/1993)	283100.3255	280691.1856	300613.5609	271345.045
Fold 4 (1994/1995/1996)	283650.3501	282408.5849	296702.6514	272925.8202
Fold 5 (1997/1998/1999)	305747.7278	302500.0471	333101.1765	290722.9533
Fold 6 (2000/2001/2002)	330472.9004	328646.9961	350799.9611	317942.2134
Fold 7 (2003/2004/2005)	387761.2179	390962.8031	418456.4875	343277.7554
Fold 8 (2006/2007/2008)	390694.4124	398640.3253	422852.9861	331670.3777
Fold 9 (2009/2010/2011)	333052.691	334773.2601	346705.9994	312844.4112
Fold 10 (2012/2013/2014)	372169.6292	376558.8556	422818.3208	328099.4277
Fold 11 (2015/2016/2017)	410186.5525	411856.2909	441851.7828	388362.4761

Fold 12 (2018/2019/2020)	409106.6609	407018.106	433135.7207	392094.0957

4.2: Experiments

This section contains the results of the twenty-two experiments ran during this study. There are two metrics used in this project: RMSE and R^2. Root mean square error (RMSE) is a metric that allows us to see how well a regression model fits the dataset by calculating and comparing the predicted values and the actual values of the data. It uses the following formula:

$$\sqrt{\Sigma(P_i - O_i)^2/n}$$

with P_i being the predicted value for the i^{th} observation, O_i being the observed value for the i^{th} observation, and n being the sample size. The RMSE should fall between 0 and 1, with lower scores showing better predictions. Unlike RMSE, R-squared (R^2) measures how much variation in the dependent variable can be explained by the independent variables of the model. It uses the following formula:

$$R^2 = 1 - \frac{ss_{res}}{ss_{tot}}$$

with SS_{res} being the residual sum of squares and SS_{tot} being the total sum of squares. It typically falls between 0 and 1, with a higher score being preferred because it shows that more variance can be explained by the model. Interpretation of the results will be discussed in a later section. The average RMSE of all folds in each approach is below. More detailed results and interpretation of the results are in later sections. Please note that some tables spill over onto the next page.

Model	ASPUS	Adjusted for Inflation
Keras (16 nodes)	0.0388218	0.0895687
Keras (32 nodes)	0.0388138	0.0999783

Keras (64 nodes)	0.0639872	0.0894668
Keras (128 nodes)	0.0366228	0.0881861
OLS	0.0597302	0.1415260
Ridge	0.0597709	0.1335983
Lasso	0.0627283	0.1441637
Elastic Net	0.0588917	0.1374890
XGBoost	0.0386741	0.0977472
SVM	0.0556597	0.0584879
Random Forest	0.0032475	0.0095926

4.2.1: Keras - no inflation, 16 nodes

Fold	RMSE	R-Squared
1	0.06226946633395503	-1.7474057645470809
2	0.02393737240910172	-1.2015021833752955
3	0.012549161535338992	-1.8460590037832958
4	0.007105777161823788	0.8413396237323129
5	0.050430709158273246	0.7945399042898229
6	0.032870811623221655	-0.3158253553558037
7	0.1064624466182276	0.8311558417823707
8	0.013268920885436235	-0.5976105007359791
9	0.03537558428755321	-3.305230566974087
10	0.02642185965975101	-0.8058149194069253
11	0.026601005618690176	0.6660751006143815
12	0.06856798626450346	-3.817373242188287
Average	0.03882175846298967	-0.8753092554956555

Architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                208
dense_2 (Dense)              (None, 16)                272
dense_3 (Dense)              (None, 16)                272
dense_4 (Dense)              (None, 16)                272
dense_5 (Dense)              (None, 16)                272
dense_6 (Dense)              (None, 16)                272
dense_7 (Dense)              (None, 1)                 17
=================================================================
Total params: 1,585
Trainable params: 1,585
Non-trainable params: 0
_________________________________________________________________
```

4.2.2.: Keras - no inflation, 32 nodes

Fold	RSME	R-Squared
1	0.04490077292224996	-0.4284996941634067
2	0.011133968320972555	0.5237158487852904
3	0.024757243289627084	-10.07692076653441
4	0.012879256851764713	0.47877297998353996
5	0.018753853900590763	0.5895716671670266
6	0.028259311260953098	0.16031933428963296
7	0.029114844344132376	0.7949837802111948
8	0.056054163185063795	-0.9737698144840703
9	0.08028822868894024	-24.684984819663057
10	0.0711062615604702	0.19444343030565592
11	0.03762762607578985	0.33186175326369316
12	0.050889686323441145	-1.6535486462771969
Average	0.03881376806033299	-2.8953379122596754

Architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                416
dense_2 (Dense)              (None, 32)                1056
dense_3 (Dense)              (None, 32)                1056
dense_4 (Dense)              (None, 32)                1056
dense_5 (Dense)              (None, 32)                1056
dense_6 (Dense)              (None, 32)                1056
dense_7 (Dense)              (None, 1)                 33
=================================================================
Total params: 5,729
Trainable params: 5,729
Non-trainable params: 0
_________________________________________________________________
```

4.2.3: Keras - no inflation, 64 nodes

Fold	RMSE	R-Squared
1	0.061785866705969444	-1.7048974471378662
2	0.01183734406431781	0.4616376135089939
3	0.018147560395636687	-4.951836344285029
4	0.02865183194390661	-1.5795865422503454
5	0.018082618097693917	0.6184258925003125
6	0.3009759410453999	-94.24771581422269
7	0.04531877046090644	0.5032755776511915
8	0.025677402588662083	0.5858259006456095
9	0.06881667324238162	-17.869609298430245
10	0.06759820427938529	0.2719676622228099
11	0.029669826353367457	0.5845844941119064
12	0.09128404976050361	-7.538022676638173
Average	0.06398717407817757	-10.405495915193628

Architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                832
dense_2 (Dense)              (None, 64)                4160
dense_3 (Dense)              (None, 64)                4160
dense_4 (Dense)              (None, 64)                4160
dense_5 (Dense)              (None, 64)                4160
dense_6 (Dense)              (None, 64)                4160
dense_7 (Dense)              (None, 1)                 65
=================================================================
Total params: 21,697
Trainable params: 21,697
Non-trainable params: 0
_________________________________________________________________
```

4.2.4: Keras - no inflation, 128 nodes

Fold	RSME	R-Squared
1	0.04644183742229042	-0.5282390944038118
2	0.014067981492316707	0.23962164761225302
3	0.010565450082231176	-1.017392886648497
4	0.033245278112112295	-2.473003755933887
5	0.03741317428806391	-0.6334482913947694
6	0.03072348013946407	0.00749701103429945
7	0.029686543058213107	0.7868533399174517
8	0.03813392369143856	0.08651011623093141
9	0.04660232562494287	-7.653478923935994
10	0.043498524874446796	0.6985401973167151
11	0.036416079855872446	0.374194926927673
12	0.0726795661432482	-4.412428324512962
Average	0.03662284706538671	-1.2103978364825496

Architecture:

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 128)	1664
dense_2 (Dense)	(None, 128)	16512
dense_3 (Dense)	(None, 128)	16512
dense_4 (Dense)	(None, 128)	16512
dense_5 (Dense)	(None, 128)	16512
dense_6 (Dense)	(None, 128)	16512
dense_7 (Dense)	(None, 1)	129

Total params: 84,353
Trainable params: 84,353
Non-trainable params: 0

4.2.5: Linear Regression - no inflation

Fold	RMSE	R-Squared
1	0.11609548254929605	-8.549995341419159
2	0.06385966840783848	-14.668213685302312
3	0.0442019277541924	-34.30993361119704
4	0.017713693007062402	0.014030366037894648
5	0.06220610280967167	-3.5156707038718586
6	0.034806849803137384	-0.2738563191726855
7	0.0458269802369	0.49207246462612597
8	0.029644343526158377	0.44796777495489926
9	0.0810114975736314	-25.149830685014255
10	0.05811724632192447	0.4618660286909917
11	0.055457168058223406	-0.4513363434922417
12	0.10782166231461687	-10.911859223393943
Average	0.059730218530221084	-8.034563273212799

4.2.6: XGBoost - no inflation

Fold	RMSE	R-Squared
1	0.08395010096357239	-3.9936139352686704
2	0.015114631968002385	0.12226923420428804
3	0.019489129853486938	-5.864349359584148
4	0.01957519950672846	-0.2040865527924789
5	0.03273922166867099	-0.2508147029964054
6	0.030611891785062247	0.014693503641000527
7	0.04818141974738293	0.4385404639788486
8	0.043188246826526416	-0.1716877931186791
9	0.032271543757218295	-3.149684151236202
10	0.07210858529183713	0.17157289089656325
11	0.03239231014279256	0.5048502798950187
12	0.03446731105569234	-0.21725709032252483
Average	0.03867413271391443	-1.0499639343919493

4.2.7: Ridge - no inflation

Fold	RMSE	R-Squared
1	0.028428967563461936	0.42734284191437644
2	0.06859093898870727	-17.07588893243904
3	0.0374906533931454	-24.40156643845852
4	0.04323647457524949	-4.874161228936532
5	0.07347914687799344	-5.300637409080607
6	0.055388706369980736	-2.225770393706615
7	0.08506381270376005	-0.7500439902794702
8	0.05722369724660907	-1.0569919140459207
9	0.07161205732243897	-19.433739916673094
10	0.060264023211438464	0.42137579752360976

11	0.05512590488710944	-0.43404954764555814
12	0.08134644538289136	-5.78023374599552
Average	0.0597709023768988	-6.707030406485241

4.2.8: Elastic Net - no inflation

Fold	RMSE	R-Squared
1	0.03425131271543891	0.16875899352774848
2	0.022309581589419845	-0.9122691474320674
3	0.041805324523300796	-30.58476706696502
4	0.0672590305468708	-13.214988696313304
5	0.06932052697131748	-4.607638229562664
6	0.03992452785029243	-0.6759867663410388
7	0.08866215782542444	-0.9012353029150078
8	0.09728467641830096	-4.945243315563036
9	0.07426649528872817	-20.976646336656888
10	0.06124592674814844	0.40236672103814897
11	0.08161155723523855	-2.1430842575262847
12	0.028759190951173692	0.15253682730291507
Average	0.058891692388637874	-6.5198497147838745

4.2.9: Lasso - no inflation

Fold	RMSE	R-Squared
1	0.07682402905993961	-3.181832928179447
2	0.008760752436425786	0.7051175108539827
3	0.05407476456557236	-51.84498204883514
4	0.06528985035123273	-12.39481311152054
5	0.05158471970499916	-2.1052632737746224

6	0.02665683918765115	0.2528491214608257
7	0.09492317871205917	-1.1792337813046947
8	0.09644274245301615	-4.842784370917575
9	0.07164653180749045	-19.453418508495872
10	0.05837490897787972	0.45708382014245386
11	0.09327006997913709	-3.105228158720317
12	0.054891532059690176	-2.0872955764614813
Average	0.06272832660792448	-8.231650108812703

4.2.10: Random Forest - no inflation

Fold	RMSE	R-Squared
1	0.001014993136091269	0.9992700405616117
2	0.001555771706695605	0.990700536249079
3	0.001385405251421472	0.9653128882698832
4	0.0009675035030290671	0.9970586249480917
5	0.0013687646987633638	0.9978136792898158
6	0.002314479924241426	0.994367543535861
7	0.00358906649626964	0.9968845392380762
8	0.0060362683299599275	0.9771114737437352
9	0.0042308922792336605	0.9286753751137471
10	0.006556398536763174	0.993151254293396
11	0.006015115158109567	0.9829257912658259
12	0.003935773303899445	0.9841281461034435
Average	0.0032475360270398015	0.9839499910510473

4.2.11: SVM - no inflation

Fold	RMSE	R-Squared
1	0.0632404264789762	-1.8337536844876743
2	0.027282062358278562	-1.859700571256361
3	0.07094548967604981	-89.96285051463347
4	0.04493276277807169	-5.344122465864431
5	0.07916751512878908	-6.313921659371865
6	0.06324576368695946	-3.2058510627392396
7	0.05317448482045398	0.31614211594393926
8	0.062132476685986295	-1.425035423637309
9	0.057506624705613324	-12.176841312162843
10	0.041687209258156104	0.7231235636961738
11	0.04559595380023208	0.018918008831574795
12	0.059006174146512894	-2.567487164343227
Average	0.055659745293673284	-10.30261501416873

4.2.12: Keras - adjusted for inflation, 16 nodes

Fold	RMSE	R-Squared
1	0.21265444574524808	-2.197439480014395
2	0.03702999920464994	0.1898294176016886
3	0.08477485114591118	-4.180815577190048
4	0.05590011192438896	-3.7896288218716556
5	0.032257165431858165	0.6824069721298065
6	0.05076757743184746	0.09410833535087859
7	0.07752164352046143	0.48448644897925064
8	0.10179474046972325	0.3377786939803754
9	0.1394693540787248	-5.09775477415843

10	0.059188476235258095	0.7871939687533502
11	0.048138840900040586	0.3738813870672607
12	0.17532699559351653	-11.425208542359345
Average	0.08956868347346904	-1.9784301643109388

Architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                208
dense_2 (Dense)              (None, 16)                272
dense_3 (Dense)              (None, 16)                272
dense_4 (Dense)              (None, 16)                272
dense_5 (Dense)              (None, 16)                272
dense_6 (Dense)              (None, 16)                272
dense_7 (Dense)              (None, 1)                 17
=================================================================
Total params: 1,585
Trainable params: 1,585
Non-trainable params: 0
_________________________________________________________________
```

4.2.13: Keras - adjusted for inflation, 32 nodes

Fold	RMSE	R-Squared
1	0.19454270209568963	-1.6759826141116188
2	0.03362092476843443	0.3321354774910096
3	0.10473736949093831	-6.908013201532698
4	0.062054389987268886	-4.902303332782438
5	0.047532288164496096	0.31040199009122005
6	0.048001851602348274	0.1901224436079848
7	0.07474351282719315	0.5207731384508154
8	0.09475044031711095	0.4262602441537333
9	0.1521552064930196	-6.257482734095203

10	0.1482918678978973	-0.3358087520661881
11	0.05410510552842031	0.2090630709719299
12	0.18520353825580863	-12.864514425838518
Average	0.09997826645238546	-2.5796123913049978

Architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                416
dense_2 (Dense)              (None, 32)                1056
dense_3 (Dense)              (None, 32)                1056
dense_4 (Dense)              (None, 32)                1056
dense_5 (Dense)              (None, 32)                1056
dense_6 (Dense)              (None, 32)                1056
dense_7 (Dense)              (None, 1)                 33
=================================================================
Total params: 5,729
Trainable params: 5,729
Non-trainable params: 0
_________________________________________________________________
```

4.2.14: Keras - adjusted for inflation, 64 nodes

Fold	RMSE	R-Squared
1	0.19243274689306117	-1.6182514855278245
2	0.0657628302035779	-1.5552303382941854
3	0.029113413682801112	0.38898717785382764
4	0.08622350350109481	-10.3953543512535
5	0.035437989529790624	0.6166842116351916
6	0.05598245124620695	-0.10155758425219497
7	0.08179723252544208	0.42605356410704076
8	0.0594429028488318	0.7741851544607322
9	0.12386480872032495	-3.809592131573636

10	0.11700164169800643	0.16844012663239294
11	0.04781554130925096	0.38226314906924663
12	0.17872615666055938	-11.911667622487997
Average	0.08946676823491234	-2.219586677469242

Architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                832
dense_2 (Dense)              (None, 64)                4160
dense_3 (Dense)              (None, 64)                4160

dense_4 (Dense)              (None, 64)                4160
dense_5 (Dense)              (None, 64)                4160
dense_6 (Dense)              (None, 64)                4160
dense_7 (Dense)              (None, 1)                 65
=================================================================
Total params: 21,697
Trainable params: 21,697
Non-trainable params: 0
_________________________________________________________________
```

4.2.15: Keras - adjusted for inflation, 128 nodes

Fold	RMSE	R-Squared
1	0.1690868646740031	-1.0214971925660445
2	0.05437095590747059	-0.7466383893435358
3	0.060722325936059564	-1.6580359981026627
4	0.05695645025205767	-3.972357350909226
5	0.03163198757313778	0.6945982512907387
6	0.05472361915183965	-0.052574894886612045
7	0.10816471396079556	-0.003610866718354089
8	0.0662273429225366	0.719697395547779

9	0.12574274490509138	-3.956535817601389
10	0.13186435196762364	-0.05624440865619351
11	0.058794806593344906	0.06600771553461349
12	0.13994675374888466	-6.916473507903083
Average	0.08818607646607042	-1.4086387553594975

Architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               1664
dense_2 (Dense)              (None, 128)               16512
dense_3 (Dense)              (None, 128)               16512
dense_4 (Dense)              (None, 128)               16512
dense_5 (Dense)              (None, 128)               16512
dense_6 (Dense)              (None, 128)               16512
dense_7 (Dense)              (None, 1)                 129
=================================================================
Total params: 84,353
Trainable params: 84,353
Non-trainable params: 0
_________________________________________________________________
```

4.2.16: Linear Regression - adjusted for inflation

Fold	RMSE	R-Squared
1	0.3159778118516575	-6.059380234558706
2	0.22152853676910444	-27.995354453310597
3	0.06331628015012843	-1.8899800524438102
4	0.09605566535049649	-13.142379037544279
5	0.13954238100471278	-4.943343274248768
6	0.051926899194557334	0.052262286449265116
7	0.13888269993653976	-0.6545901871128983

8	0.07896027490608726	0.6015537223610303
9	0.1274989177927246	-4.095952518046362
10	0.11507923409168654	0.19554169624398376
11	0.11916497695133475	-2.836742237284652
12	0.23037839767789134	-20.453077996235884
Average	0.1415260063064101	-6.768453523810972

4.2.17: XGBoost - adjusted for inflation

Fold	RMSE	R-Squared
1	0.20731869918420234	-2.038997553768289
2	0.09903890301682712	-4.795363843566382
3	0.043212733263117	-0.34613164829951404
4	0.0901373175934426	-11.453337641322552
5	0.051115998695139934	0.2024971247130709
6	0.06385438423447276	-0.4331274888946386
7	0.15138637442383043	-0.9659285015091177
8	0.1260708918312586	-0.015738983281177177
9	0.05182462120030782	0.1580532040281719
10	0.13464311497814663	-0.10122969652606795
11	0.054680833994192025	0.19214090983750554
12	0.09968274407113606	-3.0164843075990833
Average	0.09774721804050612	-1.88447070218234

4.2.18: Ridge - adjusted for inflation

Fold	RMSE	R-Squared
1	0.15031628187991006	-0.5975904277416393
2	0.23266958065424453	-30.985141658843308
3	0.07052821923498535	-2.5858316659504395
4	0.13686586630993178	-27.71222211599308
5	0.1400375825622869	-4.985601046945924
6	0.1014672451538446	-2.618721182770797
7	0.19792408850189336	-2.3604036665684434
8	0.100126415459134	0.35930725264596486
9	0.11080708551328129	-2.8489958238151556
10	0.10227414596340188	0.36460859384104827
11	0.10592847116607601	-2.0317317016701453
12	0.15423456471818703	-8.615448126317965
Average	0.13359829559309805	-7.0514809641774905

4.2.19: Elastic Net - adjusted for inflation

Fold	RMSE	R-Squared
1	0.12622889426166853	-0.12660313624011477
2	0.18175076880814547	-18.51738268126299
3	0.08948845399501393	-4.7729536631483525
4	0.1596819857924397	-38.08304145619242
5	0.1418346843208396	-5.140213180824356
6	0.08239741590200583	-1.3863303651191181
7	0.2224042049868221	-3.2430693603371292
8	0.18568968167022415	-1.2035759000052573
9	0.14211682544264534	-5.331452767077338

10	0.1009884544186392	0.380483233591037
11	0.13654759677351	-4.0377136549135635
12	0.08073916371560165	-1.634963468534103
Average	0.1374890108406296	-6.92473470000531

4.2.20: Lasso - adjusted for inflation

Fold	RMSE	R-Squared
1	0.1413374376205568	-0.4124329064817316
2	0.15607790073570463	-13.393018861894479
3	0.1261042299599052	-10.463661933947815
4	0.16870086448337518	-42.62255710145403
5	0.1266427723095934	-3.895300708196859
6	0.05989631058327947	-0.26096643954074494
7	0.25624624547637936	-4.632603288789254
8	0.21217434231279772	-1.8769889896174887
9	0.18452008447585508	-9.673323316562536
10	0.10377949742553323	0.3457665582382966
11	0.15027980215387599	-5.101921071311618
12	0.04420448084971965	0.21016078220712242
Average	0.14416366403221467	-7.648070606445927

4.2.21: Random Forest - adjusted for inflation

Fold	RMSE	R-Squared
1	0.006818621558813755	0.9967126452364581
2	0.00934204570157713	0.9484352354126857
3	0.006288714097869961	0.9714906206354879
4	0.007146795673869124	0.9217112722801045

5	0.006733078619745397	0.9861628729695532
6	0.007851348326099232	0.9783333446801258
7	0.007738737815326137	0.9948627061333146
8	0.01582626656876559	0.9839930192202729
9	0.012637850232973173	0.9499321682071178
10	0.011541563688644788	0.9919083159248504
11	0.013686948833251666	0.9493850696143663
12	0.009499361682196393	0.9635250292788884
Average	0.009592611066594363	0.9697043582994356

4.2.22: SVM - adjusted for inflation

Fold	RMSE	R-Squared
1	0.06871177258660921	0.6661774234732996
2	0.06938728020524411	-1.8446497005922695
3	0.06739447555676656	-2.2742561908291044
4	0.07024814790481727	-6.563908398132312
5	0.05962906706385195	-0.08526197319923368
6	0.057092849328642906	-0.1456892009839732
7	0.04525102257489355	0.8243488739398654
8	0.04517756032206451	0.8695638356113995
9	0.07021099944103398	-0.5453351606931884
10	0.03931056572311336	0.9061297010375061
11	0.05088596231698419	0.30038141073568225
12	0.05855568368005665	-0.38593932296803746
Average	0.058487948892006515	-0.6898698918833639

4.3: Interpretation of the results

4.3.1: RMSE

Overall, all the models have a good RMSE, values falling between the values of 0.0032475 and 0.1441637, with an average of 0.0730342. The models that used the unadjusted housing prices performed better than the models with inflation-adjusted prices, with the unadjusted prices having a range of 0.0032475 to 0.0639872 and average of 0.0469952, and the adjusted prices having a range of 0.0095926 to 0.1441637 with and average of 0.0990731. Random Forest performed the best in both cases with RMSE scores of 0.0032475 and 0.0095926, with and average of 0.00642005. For comparison, Ho et al. had a Random Forest RMSE of 0.09210. Lasso was the weakest overall, with an average RMSE of 0.1034460. Gupta et al. [3] had a Lasso RMSE of .5224, so while it may be the worst here, it is still a fairly good model when compared to other studies. Of the several Keras models, the unadjusted 128 node model did the best, and the adjusted 32 node model did the worst. The overall ranking of all 22 models according to RSME is:

Unadjusted	Adjusted	Combined
1. Random Forest: 0.0032475	1. Random Forest: 0.0095926	1. Random Forest: 0.0032475
2. Keras (128 nodes): 0.0366228	2. SVM: 0.0584879	2. (Adj) Random Forest: 0.0095926
3. XGBoost: 0.0386741	3. Keras (128 nodes): 0.0881861	3. Keras (128 nodes): 0.0366228
4. Keras (32 nodes): 0.0388138	4. Keras (64 nodes): 0.0894668	4. XGBoost: 0.0386741
5. Keras (16 nodes): 0.0388218	5. Keras (16 nodes): 0.0895687	5. Keras (32 nodes): 0.0388138
6. SVM: 0.0556597	6. XGBoost: 0.0977472	6. Keras (16 nodes): 0.0388218
7. Elastic Net: 0.0588917	7. Keras (32): 0.0999783	7. SVM: 0.0556597
8. Linear Regression: 0.0597302	8. Ridge: 0.1335983	8. (Adj) SVM: 0.0584879
9. Ridge: 0.0597709	9. Elastic Net: 0.1374890	9. Elastic Net: 0.0588917
10. Lasso: 0.0627283	10. Linear Regression: 0.1415260	10. Linear Regression: 0.0597302

11. Keras (64 nodes): 0.0639872	11. Lasso: 0.1441637	11. Ridge: 0.0597709
		12. Lasso: 0.0627283
		13. Keras (64 nodes): 0.0639872
		14. (Adj) Keras (128 nodes): 0.0881861
		15. (Adj) Keras (64 nodes): 0.0894668
		16. (Adj) Keras (16 nodes): 0.0895687
		17. (Adj) XGBoost: 0.0977472
		18. (Adj) Keras (32 nodes): 0.0999783
		19. (Adj) Ridge: 0.1335983
		20. (Adj) Elastic Net: 0.1374890
		21. (Adj) Linear Regression: 0.1415260
		22. (Adj) Lasso: 0.1441637

4.3.2: R-Squared

As seen above, there are some anomalies with the R^2 values in these models; some are negative, which is highly unusual for this metric. The folds that have this issue are 1, 2, 3, 6, 8, 9, 12. However, there is an explanation for this. According to the graphs on FRED, these folds fall in or around periods of economic recession, which adds more variance than the metric handle. FRED highlights recessions in gray. In this example, the unemployment chart was used (Figure 4.3). There are four recessions visible on the graph: July 1990 to March 1991 (folds 2 and 3), April 2001 to October 2001 (fold 6), January 2008 to June 2009 (folds 8 and 9), February 2020 to April 2020 (fold 12). There is another highlighted recession beyond the left edge of the chart, which may account for abnormalities in fold 1.

Figure 4.3: Graph of unemployment rate on FRED [24]

To remedy this, more economic datasets would have to be added, although this may overfit the model for just housing price prediction. While R^2 may not be a good metric for this specific problem, being able to identify periods of recession in this manner may be beneficial for future experiments and machine learning solutions. Gupta et al. [3] also had an issue with out-of-range R^2

values. Random Forest once again performed best according to this metric, having the only R^2 value that falls in the desired range. The unadjusted Keras (64 node) model performed the worst of all the individual models, with an R^2 of 10.4054959. Lasso performed worst overall, with both models R^2 averaging to -7.93986035.

Unadjusted	Adjusted	Combined
1. Random Forest: 0.9839500	1. Random Forest: 0.9697044	1. (Adj) Random Forest: 0.9697044
2. Keras (16 nodes): -0.8753093	2. SVM: -0.6898699	2. Random Forest: 0.9839500
3. XGBoost: -1.0499639	3. Keras (128 nodes): -1.4086388	3. (Adj) SVM: -0.6898699
4. Keras (128 nodes): -1.2103978	4. XGBoost: -1.8844707	4. Keras (16 nodes): -0.8753093
5. Keras (32 nodes): -2.8953379	5. Keras (16 nodes): -1.9784302	5. XGBoost: -1.0499639
6. Elastic Net: -6.5198497	6. Keras (64 nodes): -2.2195867	6. Keras (128 nodes): -1.2103978
7. Ridge: -6.7070304	7. Keras (32 nodes): -2.5796124	7. (Adj) Keras (128 nodes): -1.4086388
8. Linear Regression: -8.0345633	8. Linear Regression: -6.7684535	8. (Adj) XGBoost: -1.8844707

9. Lasso: -8.2316501	9. Elastic Net: -6.9247347	9. (Adj) Keras (16 nodes): -1.9784302
10. SVM: -10.3026150	10. Ridge: -7.0514810	10. (Adj) Keras (64 nodes): -2.2195867
11. Keras (64 nodes): -10.4054959	11. Lasso: -7.6480706	11. (Adj) Keras (32 nodes): -2.5796124
		12. Keras (32 nodes): -2.8953379
		13. Elastic Net: -6.5198497
		14. Ridge: -6.7070304
		15. (Adj) Linear Regression: -6.7684535
		16. (Adj) Elastic Net: -6.9247347
		17. (Adj) Ridge: -7.0514810
		18. (Adj) Lasso: -7.6480706
		19. Linear Regression: -8.0345633
		20. Lasso: -8.2316501
		21. SVM: -10.3026150
		22. Keras (64 nodes): -10.4054959

Chapter 5: Conclusion

In this paper, multiple models were investigated in their performance in predicting housing prices based on economic indicators. Three types of linear regression models were used, including OLS, Ridge, Lasso and Elastic Net, and four machine learning algorithms including Random Forest, SVM, TensorFlow Keras, and XGBoost. There were two forms of each model: one trained on raw prices, and one trained on inflation-adjusted prices. These models also utilized n-fold validation to allow us to see trends with a smaller span. There were twelve folds of three years each.

Overall, Random Forest performed the best in both cases by a significant margin. Lasso did the worst overall between both versions of the model. In regards to all models, those trained

with unadjusted prices did the best overall, with only the adjusted Random Forest making it into the top five best results. R^2 was not a helpful metric for determining model accuracy in this particular case due to economic recessions causing unmanageable variance, although it did yield some interesting results in regards to detecting those recessions in the data. Due to the models producing generally good RMSE scores, it can be concluded that housing prices can be accurately predicted using indicators of total economic health.

Chapter 6: Future Work

While the experiments yielded good results, there is always room for improvement. One way to possibly improve the models would be to look at feature importance. This is a ranking based on how useful they are at predicting a target variable. This is determined by finding and comparing the coefficients assigned to the feature during the fitting process. Using this, it may be possible to decrease the complexity of the model. Also, more features can be added and checked for overfitting, which should improve the R^2 values.

Additionally, the Truong et al. paper [5] included a hybrid model. Perhaps a hybrid of the two best models in this project could generate better results than either model by itself. Experiments could be done on implementing the models in different ratios.

Chapter 7: Sources

7.1: Works Cited

[1] S. Mehtab and J. Sen, "A Time Series Analysis-Based Stock Price Prediction Using Machine Learning and Deep Learning Models," *International Journal of Business Forecasting and Marketing Intelligence (IJBFMI)*, vol. 6, no. 4, pp. 272–335, 2020, Accessed: Jan. 05, 2023. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/2004/2004.11697.pdf

[2] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-Normalizing Neural Networks," in *The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, Long Beach, California, USA, Sep. 2017. Accessed: Jan. 05, 2023. [Online]. Available: https://arxiv.org/pdf/1706.02515.pdf

[3] Gupta et al., "Machine Learning based Predicting House Prices using Regression Techniques," in *Proceedings of the Second International Conference On Innovative Mechanisms For Industry Applications (icimia 2020)*, Dayananda Sagar College of Engineering Bangalore, India, 2020, pp. 624–630. Accessed: Jan. 05, 2023. [Online]. Available: https://www.researchgate.net/profile/Radha-Gupta-3/publication/340896273_Machine_Learning_based_Predicting_House_Prices_using_Regression_Techniques/links/60262ef992851c4ed5669e1e/Machine-Learning-based-Predicting-House-Prices-using-Regression-Techniques.pdf

[4] W. K. O. Ho, B.-S. Tang, and S. W. Wong, "Predicting property prices with machine learning algorithms," *Journal of Property Research*, vol. 38, no. 1, pp. 1–23, Oct. 2020, doi: 10.1080/09599916.2020.1832558.

[5] Q. Truong, M. Nguyen, H. Dang, and B. Mei, "Housing Price Prediction via Improved Machine Learning Techniques," in *2019 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI2019)*, Beijing, China, vol. 174, pp. 433–442. Accessed: Jan. 05, 2023. [Online]. Available: https://reader.elsevier.com/reader/sd/pii/S1877050920316318

[6] "sklearn.linear_model.Ridge — scikit-learn 0.23.2 documentation," *scikit-learn.org*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html

[7] "sklearn.linear_model.Lasso," *scikit-learn*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

[8] “sklearn.linear_model.ElasticNet,” *scikit-learn*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html

[9] R. T. John, “Regularization of Linear Models with SKLearn,” *Coinmonks*, Jun. 26, 2022. https://medium.com/coinmonks/regularization-of-linear-models-with-sklearn-f88633a93a2 (accessed Jan. 05, 2023).

[10] “Federal Home Loan Mortgage Corporation (Freddie Mac) | USAGov,” *www.usa.gov*. https://www.usa.gov/federal-agencies/federal-home-loan-mortgage-corporation-freddie-mac

[11] “TensorFlow Adam optimizer,” *EDUCBA*, Jul. 13, 2022. https://www.educba.com/tensorflow-adam-optimizer/ (accessed Jan. 05, 2023).

[12] T. Pettinger, “Factors that affect the housing market - Economics Help,” *Economics Help*, Oct. 15, 2017. https://www.economicshelp.org/blog/377/housing/factors-that-affect-the-housing-market/

[13] “Top 11 Important Economic Factors Affecting Housing Market,” *Builders in Calicut | Apartments for Sale in Kozhikode | Flats in Calicut*, Oct. 22, 2019. https://pvsbuilders.com/economic-factors-affecting-housing-market/

[14] By Original: Alisneaky Vector: Zirguezi - Own work based on: Kernel Machine.png, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47868867

7.2: Dataset Sources

[15] U.S. Census Bureau and U.S. Department of Housing and Urban Development, Average Sales Price of Houses Sold for the United States [ASPUS], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/ASPUS, January 4, 2023.

[16] “NASDAQ Composite (^IXIC) Historical Data - Yahoo Finance,” *finance.yahoo.com*. https://finance.yahoo.com/quote/%5EIXIC/history?p=%5EIXIC

[17] “S&P 500 (^GSPC) Historical Data,” *@YahooFinance*, 2019. https://finance.yahoo.com/quote/%5EGSPC/history

[18] Freddie Mac, 30-Year Fixed Rate Mortgage Average in the United States [MORTGAGE30US], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/MORTGAGE30US, January 5, 2023.

[19] iPropertyManagement, “Average Rent by Year [1940-2021]: Historical Rental Rates,” *iPropertyManagement.com*, Oct. 26, 2021. https://ipropertymanagement.com/research/average-rent-by-year

[20] U. C. Bureau, “NRS - Historical Time Series,” *www.census.gov*. https://www.census.gov/construction/nrs/data/series.html

[21] U. C. Bureau, “NRC - Historical Time Series,” *www.census.gov*. https://www.census.gov/construction/nrc/data/series.html

[22] U.S. Census Bureau, Mean Family Income in the United States [MAFAINUSA646N], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/MAFAINUSA646N, January 5, 2023.

[23]U.S. Census Bureau, Median Household Income in the United States [MEHOINUSA646N], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/MEHOINUSA646N, January 5, 2023.

[24] U.S. Bureau of Labor Statistics, Unemployment Rate [UNRATE], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/UNRATE, January 5, 2023.

[25] U.S. Bureau of Economic Analysis, Gross Domestic Product [GDP], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/GDP, January 5, 2023.

Chapter 8: Appendix

8.1: Required packages

These are the packages used in the program. No specific versions required, but they should be somewhat recent.

- datetime as dt
- pandas as pd
- numpy as np
- pandas as pd
- Keras (from tensorflow)*
- Sequential (from keras.models)
- Dense, Activation (from keras.layers)
- regularizers (from keras)
- SGD f(rom keras.optimizers)
- sklearn
- train_test_split (from sklearn.model_selection)
- xgboost as xg
- KFold (from sklearn.model_selection)
- mean_squared_error (from sklearn.metrics) as MSE
- LinearRegression (from sklearn.linear_model)
- SVR (from sklearn.svm)
- Lasso (from sklearn.linear_model)
- Ridge (from sklearn.linear_model)
- Elastic Net (from sklearn.linear_model)

- RandomForestRegressor (from sklearn.ensemble)
- r2_score (from sklearn.metrics) as r2
- load_model (from keras.models)

*While an 'import keras' does exist, it is outdated. It was folded into the TensorFlow library some time ago, so that's the most recent version

8.2: Code

Please note that this program was written in notebook format. It could technically work in an IDE like Visual Code, but it is not recommended. If your computer has a decent GPU, Jupyter Notebook is a good choice. If it doesn't, Google Colab offers one free GPU through cloud processing.

[1]	`#this is only needed if using Google Colab` `#otherwise, comment it out` `from google.colab import files` `uploaded = files.upload()`
[2]	`import datetime as dt` `import pandas as pd` `import numpy as np` `np.random.seed(21)` `import pandas as pd` `from tensorflow import keras` `#import keras` `from keras.models import Sequential` `from keras.layers import Dense, Activation, Dropout, BatchNormalization` `from keras import regularizers` `from keras.optimizers import SGD` `import sklearn` `from sklearn.model_selection import train_test_split` `import xgboost as xg`
[3]	`merged = pd.read_csv("mergedint.csv")` `merged['Date'] = pd.to_datetime(merged['Date'])` `merged=merged.set_index('Date')` `merged.head()`

[4]

```
#index
merged=merged.copy()
merged['Time'] = np.arange(len(merged.index))
print(merged[0:3])
```

[5]

```
#NOTE: for whatever reason this cell needs to be run again before running the the
actual learning part.
#the compiler says "values out of range try normalizing it :)" IT ALREADY WAS,
COLAB
#maybe it's just a colab thing, idk how it would be in other notebooks programs

inflation = 1

feats = ['NASDAQ', 'Rate', 'Time', 'SP', 'Sold', 'Sold_Adj', 'Constr',
'Constr_Adj', 'Mean_inc', 'Med_inc', 'Unemp', 'GDP']

x=merged.loc[:,feats]

#if inflation = 0, use unadjusted prices
#if inflation = 1, use prices adjusted for inflation
if inflation == 0:
  y=merged.loc[:, ['ASPUS']]
if inflation == 1:
  y=merged.loc[:,['Inf']]

x = (x- x.min())/ (x.max() - x.min())
y = (y- y.min())/ (y.max() - y.min())


x = x.to_numpy()
y = y.to_numpy()
```

[6]

```
from sklearn.model_selection import KFold
#from sklearn.model_selection import StratifiedKFold
```

[7]

```
from sklearn.metrics import mean_squared_error as MSE
#import statsmodels.api as sm
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score as r2 # evaluation metric
#from sklearn.metrics import mean_absolute_percentage_error
```

[8]

```
#check if any nan values bc csv hates me
np.any(np.isnan(merged))
```

[9]

```
nsplit = 12

kfold = KFold(n_splits=nsplit, shuffle=False, random_state=np.random.seed(21))
```

[10]
```
rmse = []
rsq = []
```

[11]
```
#0=keras
#1=xgboost
#2=OLS
#3=ridge
#4=e net
#5=lasso
#6=random forest
#7=svm
rows = 432

mode = 0

nodes = 128
```

[12]
```
#SELU activation is good for models with just a BUNCH of dense layers supposedly

foldnum = 1
for train, test in kfold.split(x, y):
  if mode == 0:
      print("START KERAS")
      model = Sequential()

      model.add(Dense(nodes, activation='selu', kernel_initializer='lecun_normal',
input_shape=(len(feats),)))


      model.add(Dense(nodes, activation='selu',
kernel_initializer='lecun_normal'))


      model.add(Dense(nodes, activation='selu',
kernel_initializer='lecun_normal'))


      model.add(Dense(nodes, activation='selu',
kernel_initializer='lecun_normal'))


      model.add(Dense(nodes, activation='selu',
kernel_initializer='lecun_normal'))


      model.add(Dense(nodes, activation='selu',
kernel_initializer='lecun_normal'))


      model.add(Dense(1))


      model.compile(loss='mse', optimizer='adam', metrics=['mse', 'mae', 'mape'])
```

```
print('------------------------------------------------------------------------')
      print('Training for fold '+ str(foldnum))



      foldxtrain = x[train]
      foldytrain = y[train]
      foldxtest = x[test]
      foldytest = y[test]

      model.fit(foldxtrain, foldytrain, batch_size=60, epochs=2000, verbose=0,\
            validation_data=(foldxtest, foldytest))


      print("Evaluate model on test data")
      trainresults = model.evaluate(foldxtrain, foldytrain, batch_size=128)
      results = model.evaluate(foldxtest, foldytest, batch_size=128)

      print("train loss, train acc:", trainresults)
      print("test loss, test acc:", results)
      print("Generate a prediction")
      prediction = model.predict(foldxtest)
      print(foldytest.shape)
      print(prediction.shape)
      rmse.append(np.sqrt(MSE(foldytest, prediction)))
      print("KERAS RMSE: ")
      print(rmse)
      rsq.append(r2(foldytest, prediction))
      print("KERAS R-SQUARED")
      print(rsq)
      model.summary()



  if mode == 1:
      print("START REGRESSION")
      xgb_r = xg.XGBRegressor(objective ='reg:squarederror',n_estimators = 100,
seed = 123)

      x_train = x[train]
      y_train = y[train]
      x_test = x[test]
      y_test = y[test]

      xgb_r.fit(x_train, y_train)


      pred = xgb_r.predict(x_test)
      rmseval = np.sqrt(MSE(y_test, pred))
      rmse.append(rmseval)
      print("XGBOOST RMSE: ")
      print(rmse)
```

```
        rsq.append(r2(y_test, pred))
        print("XGBOOST R-SQUARED")
        print(rsq)
        model = xgb_r
        import matplotlib.pyplot as plt

        xg.plot_tree(model,num_trees=0)
        plt.rcParams['figure.figsize'] = [50, 10]
        plt.show()

    if mode == 2:
        print("MODE: OLS")
        model = LinearRegression()
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]
        #regular linear regression
        model.fit(x_train, y_train)
        #predict
        pred = model.predict(x_test)
        #RMSE
        rmseval = np.sqrt(MSE(y_test, pred))
        rmse.append(rmseval)
        print("OLS RMSE: ")
        print(rmse)
        #r-squared
        rsq.append(r2(y_test, pred))
        print("OLS R-SQUARED")
        print(rsq)


    if mode == 3:
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]
        print("MODE: RIDGE")
        ridge = Ridge(alpha = 0.5)
        ridge.fit(x_train, y_train)
        pred = ridge.predict(x_test)
        rmseval = np.sqrt(MSE(y_test, pred))
        rmse.append(rmseval)
        print("RIDGE RMSE: ")
        print(rmse)
        rsq.append(r2(y_test, pred))
        print("RIDGE R-SQUARED")
        print(rsq)


    if mode == 4:
        print("MODE: ELASTIC NET")
        x_train = x[train]
```

```
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]
        en = ElasticNet(alpha = 0.01)
        en.fit(x_train, y_train)
        pred = en.predict(x_test)
        rmseval = np.sqrt(MSE(y_test, pred))
        rmse.append(rmseval)
        print("ELASTIC NET RMSE: ")
        print(rmse)
        rsq.append(r2(y_test, pred))
        print("ELASTIC NET R-SQUARED")
        print(rsq)

    if mode == 5:
        print("MODE: LASSO")
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]
        lasso = Lasso(alpha = 0.01)
        lasso.fit(x_train, y_train)
        pred = lasso.predict(x_test)
        rmseval = np.sqrt(MSE(y_test, pred))
        rmse.append(rmseval)
        print("LASSO RMSE: ")
        print(rmse)
        rsq.append(r2(y_test, pred))
        print("LASSO R-SQUARED")
        print(rsq)

    if mode == 6:
        print("MODE: RANDOM FOREST")
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]
        rf = RandomForestRegressor(n_estimators = 100, random_state = 123)
        rf.fit(x,y)
        pred = rf.predict(x_test)
        rmseval = np.sqrt(MSE(y_test, pred))
        rmse.append(rmseval)
        print("RANDOM FOREST RMSE: ")
        print(rmse)
        rsq.append(r2(y_test, pred))
        print("RANDOM FOREST R-SQUARED")
        print(rsq)

    if mode == 7:
        print("MODE: SVM")
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
```

	`` y_test = y[test] svm = SVR(kernel = 'rbf') svm.fit(x, y) pred = svm.predict(x_test) rmseval = np.sqrt(MSE(y_test, pred)) rmse.append(rmseval) print("SVM RMSE: ") print(rmse) rsq.append(r2(y_test, pred)) print("SVM R-SQUARED") print(rsq) foldnum = foldnum + 1 ```
[13]	``` if mode == 0: model.save("modelfold.h5") ```
[14]	``` from keras.models import load_model if mode == 0: model = load_model("modelfold.h5") ```
[15]	``` if mode == 0: model.summary() ```