Programming Assignment 3

Nishanth Ravula
Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260
nravula@buffalo.edu

1 Hidden Markov Model

A family of probabilistic graphical models called hidden Markov models allows one to predict a series of hidden or unknown variables given a collection of observable variables. It's the scenario when we see a series of emissions but are unable to determine the series of states the model utilized to produce the emissions.

Markov Model:
Let's first study the Markov model so that we may better comprehend the hidden Markov model. Markov models are a particular kind of probabilistic model that are used to forecast a system's future state from its current state. Depending on the kind of information needed to make judgments, Markov models can be categorized as hidden or observable.

As an illustration, consider the Markov model:
If there are six marbles in the bag, three of them are green and three of them are yellow, then. We have to choose a marble out of the bag at random, recall its color, and put it back. There will be a pattern after multiple repetitions of this method. Three out of six times, or 50% of the time, a yellow marble will be drawn. The quantity of marbles in the bag of a specific hue determines the likelihood of choosing that marble. The likelihood of acquiring a red marble relies on the bag's contents, which predicts how things will turn out in the future (the probability of selecting a particular color of marble).

Markov chain:
Markov chains are mathematical structures that alternate between states. Let's use a model of a baby's behavior as an example to better comprehend the Markov chain. The states of feeding, sleeping, playing, and crying may all be combined to make a state space. A Markov chain indicates the likelihood of a change in state, such as the potential of the infant sleeping through the night without crying.

Hidden Markov Models
Markov models with hidden variables allow for indirect observation of the data, which may be done by drawing conclusions from other observations. Because the variable cannot be directly viewed but may be inferred from viewing one or more states, it is known as a hidden state. The Hidden Markov models diverge from the Markov chain at this point.

Hidden Markov models consists of five components:

- **Hidden states**
  There could be one or more hidden states

- **Initial probability distribution**
  Initial probability distribution, it is the probability that a Markov chain will start in state. It is also possible for some states don’t have initial states. The initial distribution also defines every hidden variable in its initial condition at time t=0. This is the initial hidden state.

- **Transition probability distribution**
  A transition probability matrix represents the probability of each state moving from state x to state y. This matrix is used to show the hidden state to hidden state transition probabilities.

- There will be a sequence of observations

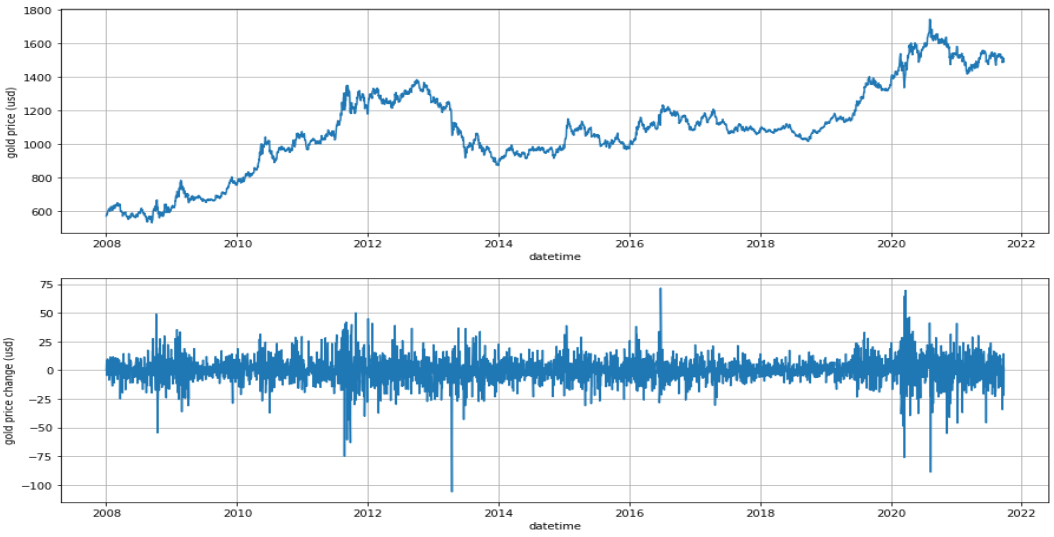
- **Emission probabilities**
  The possibility of witnessing the output is represented by the emission probability. Emission probability. The hidden variable's next hidden state is defined in terms of the emission probability. For each hidden state at time t=0, it reflects the conditional distribution across an observable output.

Dataset:

We'll use hmmlearn to evaluate historical gold prices as an application example; the data was acquired from https://www.gold.org/goldhub/data/gold-prices

We visualize the historical data in Python after importing the required modules and data. We also determine the daily change in the price of gold and only include data starting in 2008. In general, it is best to simulate the actual market circumstances by dealing with the price change rather than the actual price itself.

	datetime	gold_price_usd
0	1978-12-29	137.06
1	1979-01-01	137.06
2	1979-01-02	137.29
3	1979-01-03	134.01
4	1979-01-04	136.79
...	...	...
11146	2021-09-20	1499.15
11147	2021-09-21	1513.45
11148	2021-09-22	1511.40
11149	2021-09-23	1489.43
11150	2021-09-24	1491.27



Instead of modeling the gold price directly, we model the daily change in the gold price — this allows us to better capture the state of the market. We fit the daily change in gold prices to a Gaussian emissions model with 3 hidden states. The reason for using 3 hidden states is that we expect at the very least 3 different regimes in the daily changes — low, medium and high votality.

We find that the model does indeed return 3 unique hidden states. These numbers do not have any intrinsic meaning —which state corresponds to which volatility regime must be confirmed by looking at the model parameters.

```
Unique states:
[1 0 2]
```

We find that for this particular data set, the model will almost always start in state 0.

```
Start probabilities:
[9.00876474e-03 9.90991235e-01 1.90652535e-52]
```

The transition matrix for the 3 hidden states show that the diagonal elements are large compared to the off diagonal elements. This means that the model tends to want to remain in that particular state it is in — the probability of transitioning up or down is not high.

```
Transition matrix:
[[4.71987506e-02 9.52205396e-01 5.95853639e-04]
 [8.12868067e-01 1.35345228e-01 5.17867054e-02]
 [3.95757463e-02 4.27802116e-02 9.17644042e-01]]
```

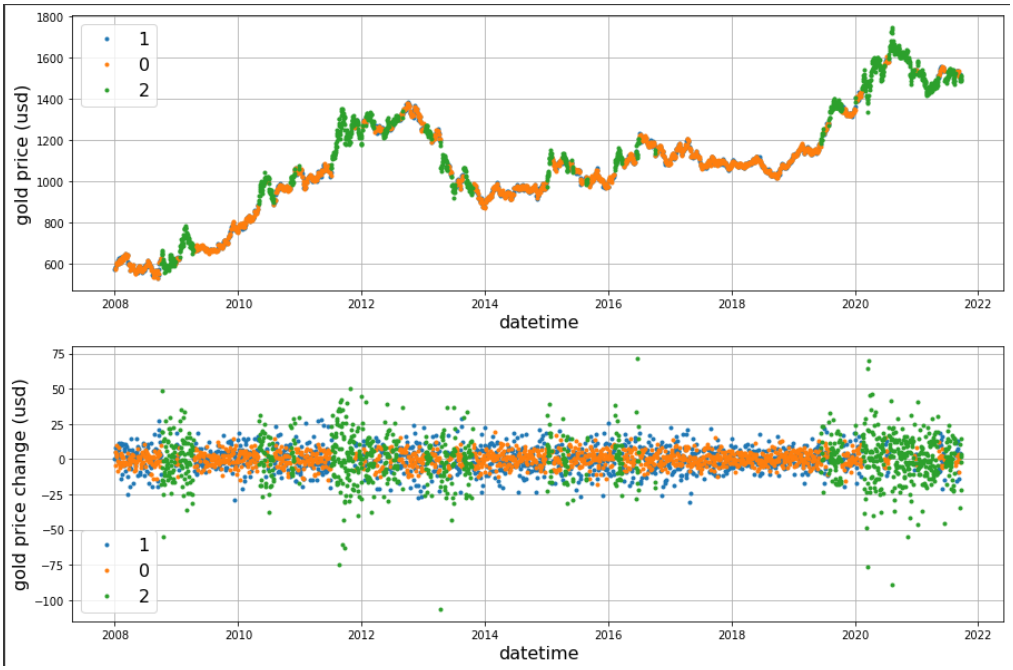
Finally, we take a look at the Gaussian emission parameters. Remember that each observable is drawn from a multivariate Gaussian distribution. For state 0, the Gaussian mean is 0.28, for state 1 it is 0.22 and for state 2 it is 0.27. The fact that states 0 and 2 have very similar means is problematic — our current model might not be too good at actually representing the data.

```
Gaussian distribution means:
[[0.27897404]
 [0.20658205]
 [0.30620104]]
```

We also have the Gaussian covariances. Note that because our data is 1 dimensional, the covariance matrices are reduced to scalar values, one for each state. For state 0, the covariance is 33.9, for state 1 it is 142.6 and for state 2 it is 518.7. This seems to agree with our initial assumption about the 3 volatility regimes — for low volatility the covariance should be small, while for high volatility the covariance should be very large.

```
Gaussian distribution covariances:
[[[ 28.11246244]]

 [[ 77.18151421]]

 [[324.43713707]]]
```

Plotting the model's state predictions with the data, we find that the states 0, 1 and 2 appear to correspond to low volatility, medium volatility and high volatility.



From the graphs above, we find that periods of high volatility correspond to difficult economic times such as the Lehmann shock from 2008 to 2009, the recession of 2011–2012 and the covid pandemic induced recession in 2020. Furthermore, we see that the price of gold tends to rise during times of uncertainty as investors increase their purchases of gold which is seen as a stable and safe asset.

We took a brief look at hidden Markov models, which are generative probabilistic models used to model sequential data. We reviewed a simple case study on people's moods to show explicitly how hidden Markov models work mathematically. We then introduced a very useful hidden Markov model Python library hmmlearn, and used that library to model actual historical gold prices using 3 different hidden states corresponding to 3 possible market volatility levels.

References

[1] C. M. Bishop (2006), Pattern Recognition and Machine Learning, Springer.

[2] Mark Stamp (2021), A Revealing Introduction to Hidden Markov Models, Department of Computer Science San Jose State University.

[3] https://hmmlearn.readthedocs.io/en/latest/.

[4] Changyou Chen Lecture slides.

[5] Dataset: https://www.gold.org/goldhub/data

[6] Natsume, Y. (2022, April 25). Hidden markov models with python. Medium. Retrieved December 1, 2022, from https://medium.com/@natsunoyuki/hidden-markov-models-with-python-c026f778dfa7

2 Random Forest

Random forest is a Supervised Machine Learning Algorithm that is used widely in Classification and Regression problems. It builds decision trees on different samples and takes their majority vote for classification and average in case of regression.

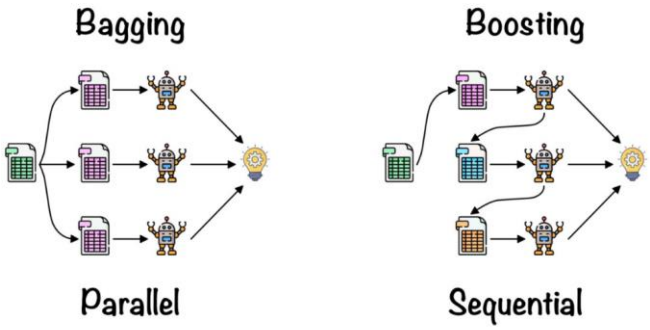
The Random Forest Algorithm's ability to handle data sets with both continuous variables, as in regression, and categorical variables, as in classification, is one of its most crucial qualities. In terms of categorization issues, it delivers superior outcomes.

Before understanding the working of the random forest algorithm in machine learning, we must look into the ensemble technique. Ensemble simply means combining multiple models. Thus a collection of models is used to make predictions rather than an individual model.

Ensemble uses two types of methods:

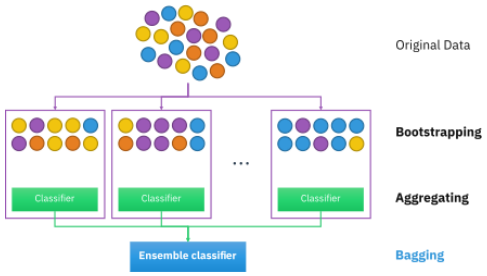
1. Bagging– It creates a different training subset from sample training data with replacement & the final output is based on majority voting. For example, Random Forest.

2. Boosting– It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy. For example, ADA BOOST, XG BOOST



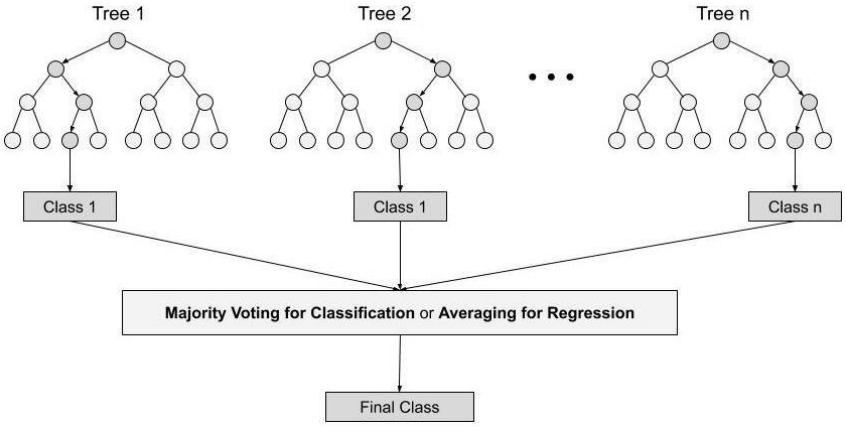
Bagging:

Bagging, also known as Bootstrap Aggregation is the ensemble technique used by random forest. Bagging chooses a random sample from the data set. Hence each model is generated from the samples (Bootstrap Samples) provided by the Original Data with replacement known as row sampling. This step of row sampling with replacement is called bootstrap. Now each model is trained independently which generates results. The final output is based on majority voting after combining the results of all models. This step which involves combining all the results and generating output based on majority voting is known as aggregation.



Steps involved in random forest algorithm:

Step 1: In Random forest n number of random records are taken from the data set having k number of records.
Step 2: Individual decision trees are constructed for each sample.
Step 3: Each decision tree will generate an output.
Step 4: Final output is considered based on Majority Voting or Averaging for Classification and regression respectively.



Important Features of Random Forest:

1. **Diversity**- Not all attributes/variables/features are considered while making an individual tree, each tree is different.
2. **Immune to the curse of dimensionality**- Since each tree does not consider all the features, the feature space is reduced.
3. **Parallelization**- Each tree is created independently out of different data and attributes. This means that we can make full use of the CPU to build random forests.
4. **Train-Test split**- In a random forest we don't have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.
5. **Stability**- Stability arises because the result is based on majority voting/ averaging.

Use Cases

This algorithm is widely used in E-commerce, banking, medicine, the stock market, etc.
For example: In the Banking industry it can be used to find which customer will default on the loan.

Advantages and Disadvantages of Random Forest Algorithm

Advantages:
1. It can be used in classification and regression problems.
2. It solves the problem of overfitting as output is based on majority voting or averaging.
3. It performs well even if the data contains null/missing values.
4. Each decision tree created is independent of the other thus it shows the property of parallelization.
5. It is highly stable as the average answers given by a large number of trees are taken.
6. It maintains diversity as all the attributes are not considered while making each decision tree though it is not true in all cases.
7. It is immune to the curse of dimensionality. Since each tree does not consider all the attributes, feature space is reduced.

8. We don’t have to segregate data into train and test as there will always be 30% of the data which is not seen by the decision tree made out of bootstrap.

Disadvantages
1. Random forest is highly complex when compared to decision trees where decisions can be made by following the path of the tree.
2. Training time is more compared to other models due to its complexity. Whenever it has to make a prediction each decision tree has to generate output for the given input data.

Dataset:
First, we need some data. To use a realistic example, I retrieved weather data for Seattle, WA from 2016 using the NOAA Climate Data Online tool. Generally, about 80% of the time spent in data analysis is cleaning and retrieving data, but this workload can be reduced by finding high-quality data sources. The NOAA tool is surprisingly easy to use and temperature data can be downloaded as clean csv files which can be parsed in languages such as Python or R.

The following is the sample data from our dataset:

	year	month	day	week	temp_2	temp_1	average	actual	forecast_noaa	forecast_acc	forecast_under	friend
0	2016	1	1	Fri	45	45	45.6	45	43	50	44	29
1	2016	1	2	Sat	44	45	45.7	44	41	50	44	61
2	2016	1	3	Sun	45	44	45.8	41	43	46	47	56
3	2016	1	4	Mon	44	41	45.9	40	44	48	46	53
4	2016	1	5	Tues	41	40	46.0	44	46	46	46	41

The information is in the tidy data format with each row forming one observation, with the variable values in the columns.

Following are explanations of the columns:
year: 2016 for all data points
month: number for month of the year
day: number for day of the year
week: day of the week as a character string
temp_2: max temperature 2 days prior
temp_1: max temperature 1 day prior
average: historical average max temperature
actual: max temperature measurement
friend: your friend’s prediction, a random number between 20 below the average and 20 above the average

Identify Anomalies/ Missing Data
If we look at the dimensions of the data, we notice only there are only 348 rows, which doesn’t quite agree with the 366 days we know there were in 2016. Looking through the data from the NOAA, I noticed several missing days, which is a great reminder that data collected in the real-world will never be perfect. Missing data can impact an analysis as can incorrect data or outliers. In this case, the missing data will not have a large effect, and the data quality is good because of the source. We also can see there are nine columns which represent eight features and the one target (‘actual’).

```
The shape of our features is: (348, 12)
```

To identify anomalies, we can quickly compute summary statistics. They following is the descriptive statistics for each column.

	year	month	day	temp_2	temp_1	average	actual	forecast_noaa	forecast_acc	forecast_under	friend
count	348.0	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000
mean	2016.0	6.477011	15.514368	62.652299	62.701149	59.760632	62.543103	57.238506	62.373563	59.772989	60.034483
std	0.0	3.498380	8.772982	12.165398	12.120542	10.527306	11.794146	10.605746	10.549381	10.705256	15.626179
min	2016.0	1.000000	1.000000	35.000000	35.000000	45.100000	35.000000	41.000000	46.000000	44.000000	28.000000
25%	2016.0	3.000000	8.000000	54.000000	54.000000	49.975000	54.000000	48.000000	53.000000	50.000000	47.750000
50%	2016.0	6.000000	15.000000	62.500000	62.500000	58.200000	62.500000	56.000000	61.000000	58.000000	60.000000
75%	2016.0	10.000000	23.000000	71.000000	71.000000	69.025000	71.000000	66.000000	72.000000	69.000000	71.000000
max	2016.0	12.000000	31.000000	117.000000	117.000000	77.400000	92.000000	77.000000	82.000000	79.000000	95.000000

There are not any data points that immediately appear as anomalous and no zeros in any of the measurement columns. Another method to verify the quality of the data is make basic plots. Often it is easier to spot anomalies in a graph than in numbers. Examining the quantitative statistics, we can feel confident in the high quality of our data. There are no clear outliers, and although there are a few missing points, they will not detract from the analysis.

Data Preparation:

Unfortunately, we aren't quite at the point where you can just feed raw data into a model and have it return an answer (although people are working on this)! We will need to do some minor modification to put our data into machine-understandable terms. We will use the Python library Pandas for our data manipulation relying, on the structure known as a dataframe, which is basically an excel spreadsheet with rows and columns.

The exact steps for preparation of the data will depend on the model used and the data gathered, but some amount of data manipulation will be required for any machine learning application.

One-Hot Encoding

The first step for us is known as one-hot encoding of the data. This process takes categorical variables, such as days of the week and converts it to a numerical representation without an arbitrary ordering. Days of the week are intuitive to us because we use them all the time. You will (hopefully) never find anyone who doesn't know that 'Mon' refers to the first day of the workweek, but machines do not have any intuitive knowledge. What computers know is numbers and for machine learning we must accommodate them. We could simply map days of the week to numbers 1–7, but this might lead to the algorithm placing more importance on Sunday because it has a higher numerical value. Instead, we change the single column of weekdays into seven columns of binary data. This is best illustrated pictorially. One hot encoding takes this:

week
Mon
Tue
Wed
Thu
Fri

and it turns into

Mon	Tue	Wed	Thu	Fri
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

So, if a data point is a Wednesday, it will have a 1 in the Wednesday column and a 0 in all other columns. This process can be done in pandas in a single line! The following is the snapshot of data one-hot encoding:

	average	actual	forecast_noaa	forecast_acc	forecast_under	friend	week_Fri	week_Mon	week_Sat	week_Sun	week_Thurs	week_Tues	week_Wed
0	45.6	45	43	50	44	29	1	0	0	0	0	0	0
1	45.7	44	41	50	44	61	0	0	1	0	0	0	0
2	45.8	41	43	46	47	56	0	0	0	1	0	0	0
3	45.9	40	44	48	46	53	0	1	0	0	0	0	0
4	46.0	44	46	46	46	41	0	0	0	0	0	1	0

The shape of our data is now 349 x 15 and all of the column are numbers, just how the algorithm likes it!

Features and Targets and Convert Data to Arrays:
Now, we need to separate the data into the features and targets. The target, also known as the label, is the value we want to predict, in this case the actual max temperature and the features are all the columns the model uses to make a prediction. We will also convert the Pandas dataframes to Numpy arrays because that is the way the algorithm works.

Training and Testing Sets:
There is one final step of data preparation: splitting data into training and testing sets. During training, we let the model 'see' the answers, in this case the actual temperature, so it can learn how to predict the temperature from the features. We expect there to be some relationship between all the features and the target value, and the model's job is to learn this relationship during training. Then, when it comes time to evaluate the model, we ask it to make predictions on a testing set where it only has access to the features. Because we do have the actual answers for the test set, we can compare these predictions to the true value to judge how accurate the model is. Generally, when training a model, we randomly split the data into training and testing sets to get a representation of all data points. Setting the random state to 42 which means the results will be the same each time we run the split for reproducible results.

We can look at the shape of all the data to make sure we did everything correctly. We expect the training features number of columns to match the testing feature number of columns and the number of rows to match for the respective training and testing features and the labels :

```
Training Features Shape: (261, 17)
Training Labels Shape: (261,)
Testing Features Shape: (87, 17)
Testing Labels Shape: (87,)
```

It looks as if everything is in order! Just to recap, to get the data into a form acceptable for machine learning we:

1. One-hot encoded categorical variables
2. Split data into features and labels
3. Converted to arrays
4. Split data into training and testing sets

Depending on the initial data set, there may be extra work involved such as removing outliers, imputing missing values, or converting temporal variables into cyclical representations. These steps may seem arbitrary at first, but once you get the basic workflow, it will be generally the same for any machine learning problem. It's all about taking human-readable data and putting it into a form that can be understood by a machine learning model.

Establish Baseline:
Before we can make and evaluate predictions, we need to establish a baseline, a sensible measure that we hope to beat with our model. If our model cannot improve upon the baseline, then it will be a failure and we should try a different model or admit that machine learning is not right for our problem. The baseline prediction for our case can be the historical max temperature averages. In

other words, our baseline is the error we would get if we simply predicted the average max temperature for all days.

```
Average baseline error:  5.06
```

Train Model:
After all the work of data preparation, creating and training the model is pretty simple using Scikit-learn. We import the random forest regression model from skicit-learn, instantiate the model, and fit (scikit-learn's name for training) the model on the training data. This entire process is only 3 lines in scikit-learn!

Make Predictions on the Test Set:
Our model has now been trained to learn the relationships between the features and the targets. The next step is figuring out how good the model is! To do this we make predictions on the test features (the model is never allowed to see the test answers). We then compare the predictions to the known answers. When performing regression, we need to make sure to use the absolute error because we expect some of our answers to be low and some to be high. We are interested in how far away our average prediction is from the actual value so we take the absolute value (as we also did when establishing the baseline).
Making predictions with out model is another 1-line command in Skicit-learn.

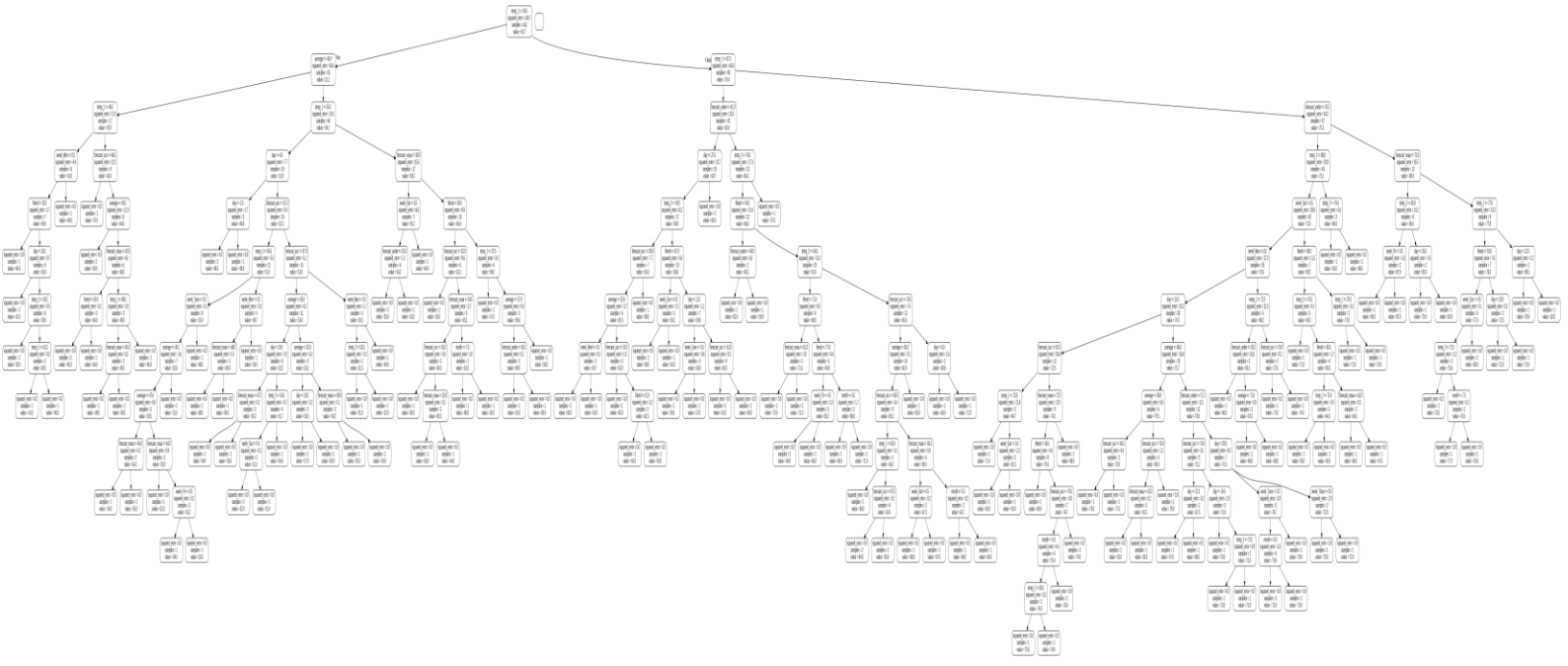
```
Mean Absolute Error: 3.87 degrees.
```

Our average estimate is off by 3.83 degrees. That is more than a 1 degree average improvement over the baseline. Although this might not seem significant, it is nearly 25% better than the baseline, which, depending on the field and the problem, could represent millions of dollars to a company.

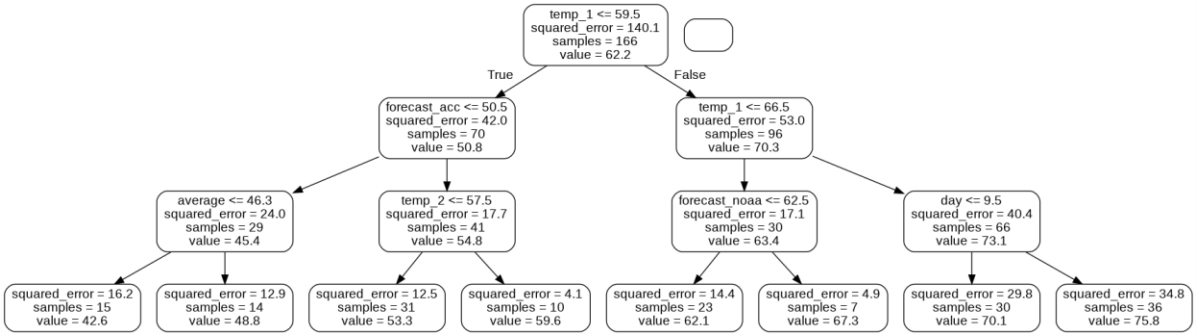
Determine Performance Metrics:
To put our predictions in perspective, we can calculate an accuracy using the mean average percentage error subtracted from 100 %.

```
Accuracy: 93.93 %.
```

Visualizing a Single Decision Tree
One of the coolest parts of the Random Forest implementation in Skicit-learn is we can actually examine any of the trees in the forest. We will select one tree, and save the whole tree as an image.



Here is the reduced size tree annotated with labels



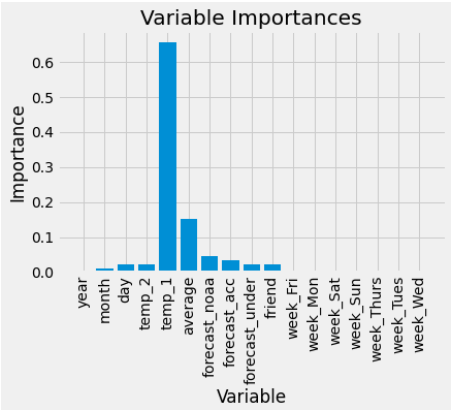
Variable Importance:

In order to quantify the usefulness of all the variables in the entire random forest, we can look at the relative importance's of the variables. The importance's returned in Skicit-learn represent how much including a particular variable improves the prediction. The actual calculation of the importance is beyond the scope of this post, but we can use the numbers to make relative comparisons between variables.

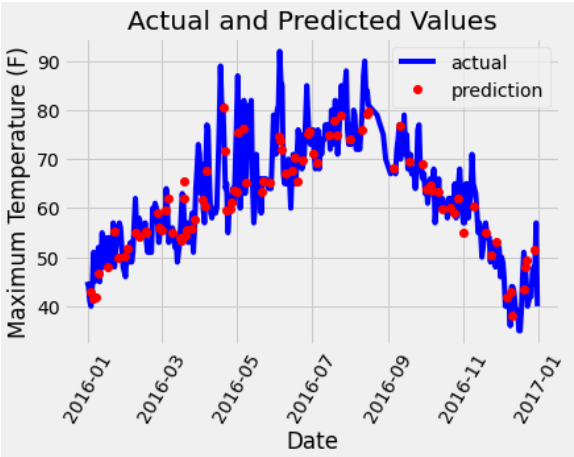
```
Variable: temp_1               Importance: 0.66
Variable: average              Importance: 0.15
Variable: forecast_noaa        Importance: 0.05
Variable: forecast_acc         Importance: 0.03
Variable: day                  Importance: 0.02
Variable: temp_2               Importance: 0.02
Variable: forecast_under       Importance: 0.02
Variable: friend               Importance: 0.02
Variable: month                Importance: 0.01
Variable: year                 Importance: 0.0
Variable: week_Fri             Importance: 0.0
Variable: week_Mon             Importance: 0.0
Variable: week_Sat             Importance: 0.0
Variable: week_Sun             Importance: 0.0
Variable: week_Thurs           Importance: 0.0
Variable: week_Tues            Importance: 0.0
Variable: week_Wed             Importance: 0.0
```

Visualizations:

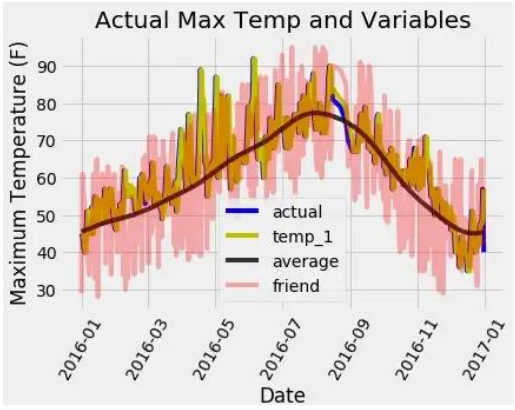
A simple bar plot of the feature importance to illustrate the disparities in the relative significance of the variables. Plotting in Python is kind of non-intuitive.



Next, we can plot the entire dataset with predictions highlighted. This requires a little data manipulation, but its not too difficult. We can use this plot to determine if there are any outliers in either the data or our predictions.

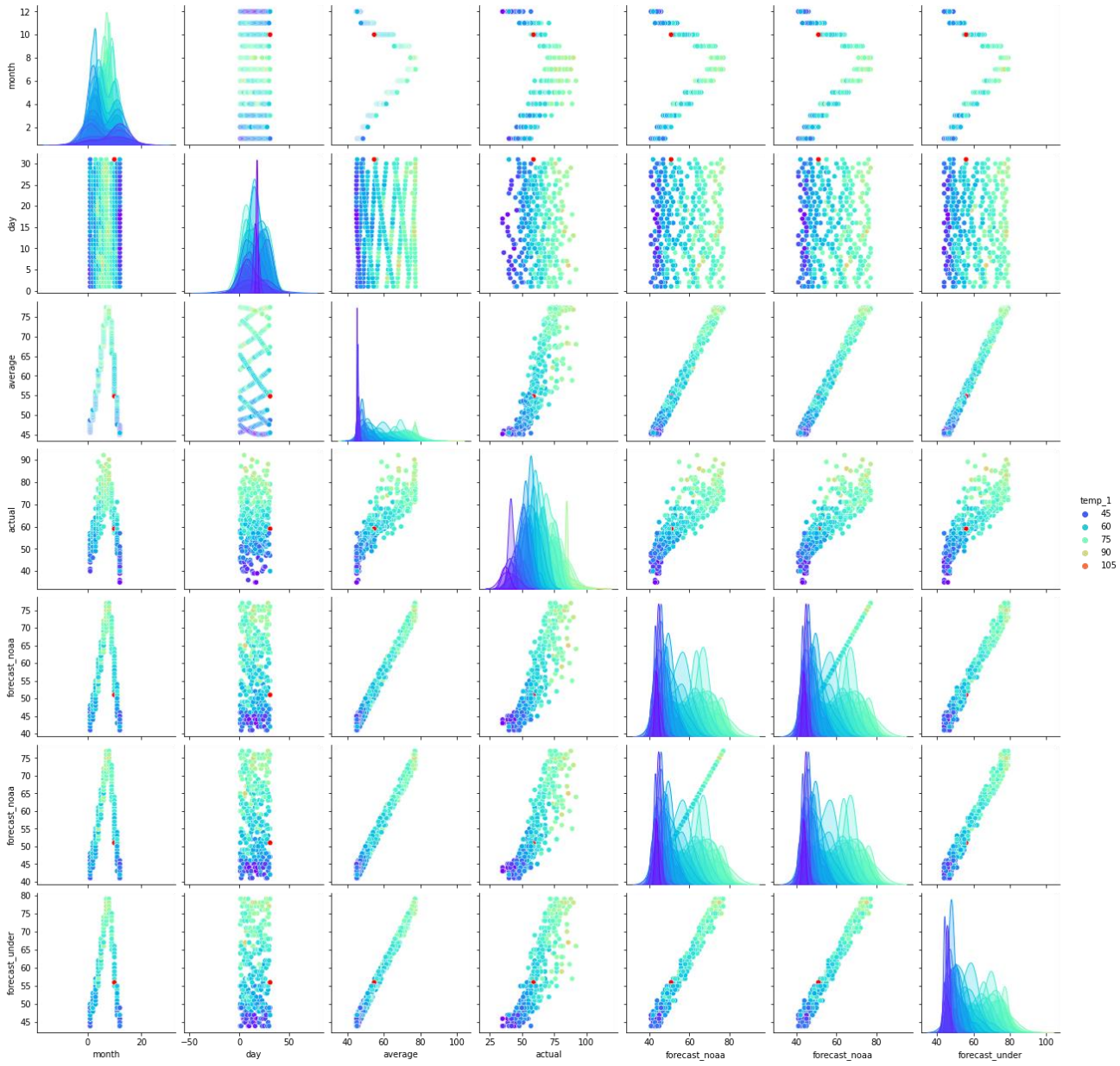


A little bit of work for a nice looking graph! It doesn't look as if we have any noticeable outliers that need to be corrected. To further diagnose the model, we can plot residuals (the errors) to see if our model has a tendency to over-predict or under-predict, and we can also see if the residuals are normally distributed. However, I will just make one final chart showing the actual values, the temperature one day previous, the historical average, and our friend's prediction. This will allow us to see the difference between useful variables and those that aren't so helpful.



With those graphs, we have completed an entire end-to-end machine learning example! At this point, if we want to improve our model, we could try different hyperparameters (settings) try a different algorithm, or the best approach of all, gather more data! The performance of any model is directly proportional to the amount of valid data it can learn from, and we were using a very limited amount of information for training.

The following is the pairplot of the dataset:


month
day
average
actual
forecast_noaa
forecast_noaa
forecast_under
temp_1
45
60
75
90
105

References:

[1] Wikimedia Foundation. (2022, November 29). Random Forest. Wikipedia. Retrieved December 1, 2022, from https://en.wikipedia.org/wiki/Random_forest

[2] Navlani, A. (2018, May 16). Sklearn Random Forest classifiers in python tutorial. DataCamp. Retrieved December 1, 2022, from https://www.datacamp.com/tutorial/random-forests-classifier-python

[3] National Centers for Environmental Information (NCEI). (n.d.). Climate Data Online Data Tools. Data Tools | Climate Data Online (CDO) | National Climatic Data Center (NCDC). Retrieved December 1, 2022, from https://www.ncdc.noaa.gov/cdo-web/datatools

[4] Random Forest regression in python. GeeksforGeeks. (2022, August 23). Retrieved December 1, 2022, from https://www.geeksforgeeks.org/random-forest-regression-in-python/

[5] Changyou Chen Lecture Slides

3 AdaBoost

AdaBoost, short for Adaptive Boosting, has become one of the most popular "off-the-shelf" algorithms in Data Science. The AdaBoost algorithm is based on the concept of "boosting". The idea behind boosting is that a set of "weak" classifiers can make up to a robust classifier using a voting mechanism. A weak classifier is one that will only yield slightly better results than tossing a (fair) coin. In other words, if by randomly guessing a binary label we would be right about 50% of times, a weak classifier would be right, say, 55% (or any other number close to 50%).

With boosting, we train a sequence of weak classifiers on successive modified versions of our sample. Let's put an example to see how this works. Say that we have a sample with five observations. Each observation is labelled either -1 or 1, such that our labels are y = [1, 1, -1, -1, -1]. We also have some explanatory variables x. In the first boosting round, we train a weak classifier $G_1(x)$, that yields the following prediction: $G_1(x) = [1, -1, -1, 1, -1]$. We can see that the second and fourth observations have been misclassified. In the second round of boosting, we will weight each observation in our sample using some weights w_i. We will set w_i so that the two misclassified observations have a larger impact in the learning process of a new classifier $G_2(x)$ than the other three. The new classifier may miss some of the previously well-classified observations now, but it will be better at predicting the difficult ones.

After repeating the process described above M times, we will get a set of M weak classifiers that we can combine in a final, robust meta-classifier G(x). This meta-classifier will allocate a prediction for each observation based on a weighted majority vote, as shown in the formula below.

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

See that the weight of each weak classifier in the majority vote is denoted by α. These are not the weights that we apply to each observation in each boosting round, which we have called w_i.

Weak classifiers

Remember that we loosely defined a weak classifier as one that would only yield slightly better results than random guessing. A more formal version of this definition would be that the error rate of the classifier is less than but close to 50%. In this case, the in-sample error rate would simply be the number of misclassified observations (i.e. $y_i \neq G(x_i)$) over the total sample size, as shown below.

$$\overline{\text{err}} = \frac{1}{N}\sum_{i=1}^{N} I(y_i \neq G(x_i))$$

Many algorithms could qualify as weak classifiers but, in the case of AdaBoost, we typically use "stumps"; that is, decision trees consisting of just two terminal nodes. Intuitively, in a binary classification problem a stump will try to divide the sample with just one cut across the one of the multiple explanatory variables of the dataset. As you can imagine, very often this is not enough to separte classes in a neat weay, especially in complex datasets.

Now, if we trained a decision tree classifier for 1,000 times on the same data, we would always get the same cut (for a given random seed, if you are using an approximation heuristic). When drawing the line, the decision tree is minimizing a metric that's based on the distribution your target values, usually the Gini coefficient or information gain from the resulting categories. And here is where our observation weights, w_i, kick in. In every boosting interation, we tell our stump

to weigh each observation differently when estimating the metric that it is minimizing. That way, in each iteration we get a different decision tree.

Our Algorithm:
To solve a binary classification problem, our AdaBoost algorithm will fit a sequence of weak classifiers (stumps) through a series of boosting iterations. These classifiers will form a meta-classifier that will yield a prediction based on a weighted majority vote mechanism. In each boosting iteration, we will give more weight to those observations that were misclassified in the previous iteration. We can formalise this process as shown in the algorithm below. One can show that the formulas for α and w come from minimizing the exponential loss function.

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

We are going to start our implementation by identifying independent chunks of the process that we can code outside of the loop in step 2. Steps 2(b), 2(c) and 2(d) contain operations that are repeated with every iteration, so we are going to define them as Python functions.

Note that these functions are independent of the algorithm. One could calculate the error rate of any classifier with the compute_error function. Similarly, we could estimate α and w for other contexts.
It is time to define our AdaBoost classifier. To do so, we are going to construct a Python class called AdaBoost that will be composed of several methods to fit the model and make predictions. It will also contain other relevant information about our fitted model, such as the sequence of weak classifiers and their weights in the final vote, as well as training and prediction errors.

The .fit() method has a very similar structure to the typical Scikit-learn class. As necessary arguments, it takes a matrix of independent variables X in the form of an array-like object, and a vector with the target variable, y. The target variable has to be binary, and it has to be encoded so that its values are either -1 or 1 (this makes voting easier). The boosting round paramter is optional, with 100 being the default.

Steps 1 and 2 of the AdaBoost.M1 algorithm are characterised within the for loop.
The if statement at the begining initialises the weight for each observation to 1/N in the first round (step 1), and then it updates them after the stump has been fitted and the α for that iteration has been calculated (step 2(d)). The output of this method is a set of stumps, alphas and training errors that we store for further use in the predict method. Step 3 of the algorithm is defined in a separate method, which we call .predict().

Similar to the .fit() method, .predict() takes as argument an array-like matrix of explanatory variables. It then calculates a prediction for each observation and stump. These are stored in a dataframe which will be used in the majority-vote part of the process to calculate the final prediction for each observation.

Dataset:

We will be using iris dataset. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other. The AdaBoost algorithm can be used for both regression and classification problems, In this section, we will use the iris dataset to train the model and make predictions about the output values of categorical data. As soon as we'll use the Iris dataset, the output values would be three species of flowers.

The sklearn's submodule datasets contains iris data, so we just need to load the dataset from there. Now, let's print a few rows of the dataset to get familiar with the type of data.

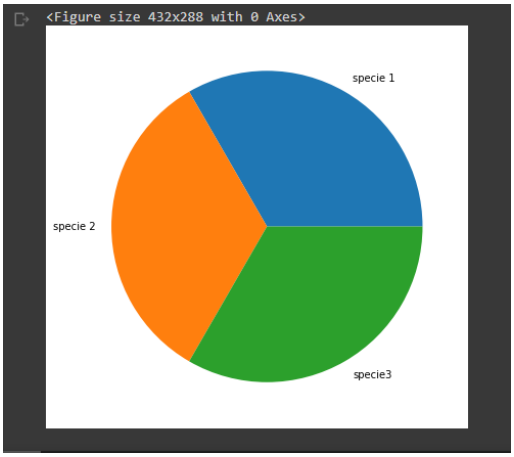
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

We can also print out the output class to see the kind of data we have in the output class.

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Notice that the output class contains only three categorical values.

The next step is to check if the dataset is balanced or unbalanced. We will use a pie chart to plot the output categories to see how well our dataset is balanced.



This shows that our data is perfectly balanced.

Splitting the dataset

Before training the model, we will split the dataset into training and testing part so that later we can evaluate the performance of our model.

We will store the input and output data in separate variables.
The next step is to split the dataset into the testing and training parts. We will assign 75% of the data to the training and the remaining 25% to the testing.
Once splitting is done, we can start training the model. We will train the model on a different number of stump trees and will compare the results.

Training the model on 1 stump tree
For demonstration purposes, let's initialize the AdaBoost with only 1 stump tree. Now we will train the classifier on the training dataset. Once the training is complete, we will use the testing data to predict the output values and store them in a variable. Now we have predictions of the model trained on 1 stump tree. Let's now calculate the accuracy of the model.

```
The accuracy of the model is:   0.631578947368421
```

This shows that 60% of the testing dataset was classified correctly by the AdaBoost algorithm when trained using only one stump tree.

Training the model on 20 stump trees
Now let's increase the number of stump trees to 20 and see how the predictions of the model change. Let us now find the accuracy of the model.

```
The accuracy of the model is:   0.9736842105263158
```

This time we've got much better (accurate) predictions. That became possible because we've increased the number of stump trees to help the model to decrease the total prediction errors. But it does not means increasing the stump trees will always increase the accuracy.

Also, it is not always possible to guess with the optimum number of stump trees. However, there are various ways to find the optimum number of trees. In this article, we will be using the GridSearchCV helper class.

Using GridSearchCV to find optimum stump trees number:
GridSearchCV class allows us to find the best set of parameters from the provided ranges for such parameters. Basically, it calculates the model's performance for every single combination of provided parameters and outputs the best parameters combination.

Let's apply the GridSearchCV to find an optimum number of stump trees for the AdaBoost algorithm.

```
The best estimator returned by GridSearch CV is: AdaBoostClassifier(n_estimators=4)
```

This shows that the optimum number of stump trees for the Adaboost on our dataset is 5.
Let's train the model on 5 stump trees and see the accuracy:

```
The accuracy of the model is:   0.9736842105263158
```

We've got 97% accuracy for our model for this dataset using only 5 stump trees.

Implementation of AdaBoost algorithm on the regression problem
A dataset having continuous output values is known as a regression dataset. In this section, we will be using a dataset that contains information about the prices of houses in Dushanbe Tajikistan. The dataset contains the number of floors, number of rooms, area, and location of

houses as independent variables and the price of the house as an output. Let's first, explore the dataset.

Importing and exploring the dataset

We will use the pandas module to import the dataset. Once we import the dataset, we can use head() method to print a few rows to get familiar with the dataset.

	Unnamed: 0	number_of_rooms	floor	area	latitude	longitude	price
0	0	1	1	58.0	38.585834	68.793715	330000
1	1	1	14	68.0	38.522254	68.749918	340000
2	2	3	8	50.0	NaN	NaN	700000
3	3	3	14	84.0	38.520835	68.747908	700000
4	4	3	3	83.0	38.564374	68.739419	415000

Our dataset contains null values, so we need to take care of them before using this data for training the model. Let's use isnull() and sum() method to get the total number of null values.

```
Unnamed: 0            0
number_of_rooms       0
floor                 0
area                  0
latitude           1849
longitude          1849
price                 0
dtype: int64
```

There're various ways for handling dataset null values problem, but for simplicity, we'll just drop them. Let's use the dropna() method to do this.

```
Unnamed: 0         0
number_of_rooms    0
floor              0
area               0
latitude           0
longitude          0
price              0
dtype: int64
```

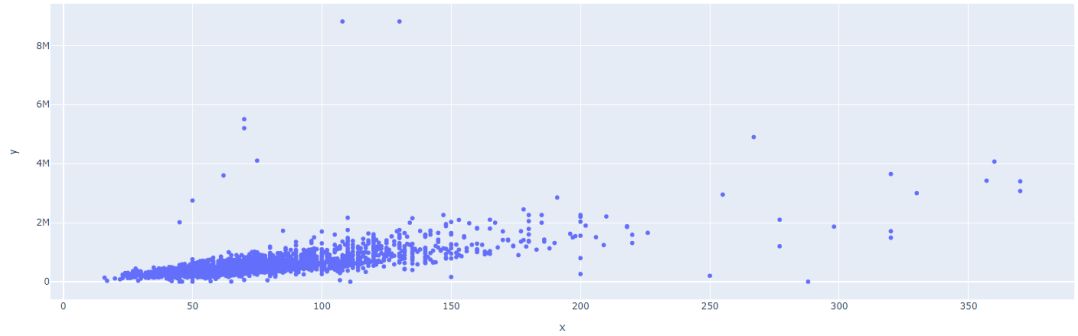
Let's check the shape of the dataset:

```
(3730, 7)
```

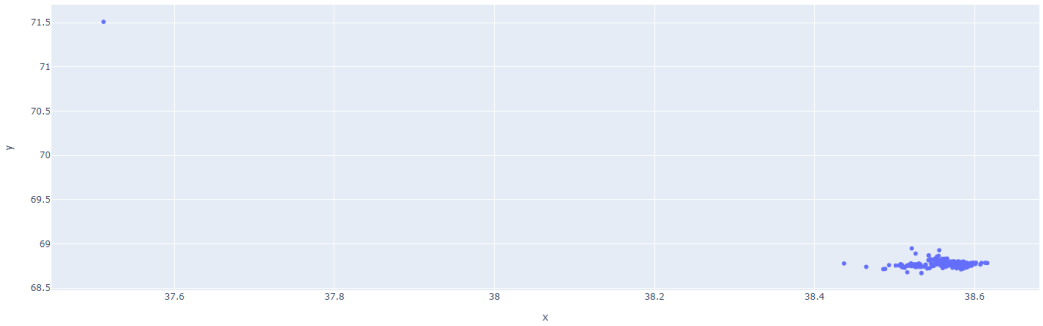
Visualizing dataset

We will now use the plotly module to visualize the dataset to analyze how the input data is related to the price.

Let's first see how the area of the houses affects the price of the house. We can plot the price on the y-axis and the area of the house on the x-axis



This shows that as the area of the houses increase, the price also increases with few exceptions. Let's now visualize the locations of these houses by plotting them using latitude and longitude.



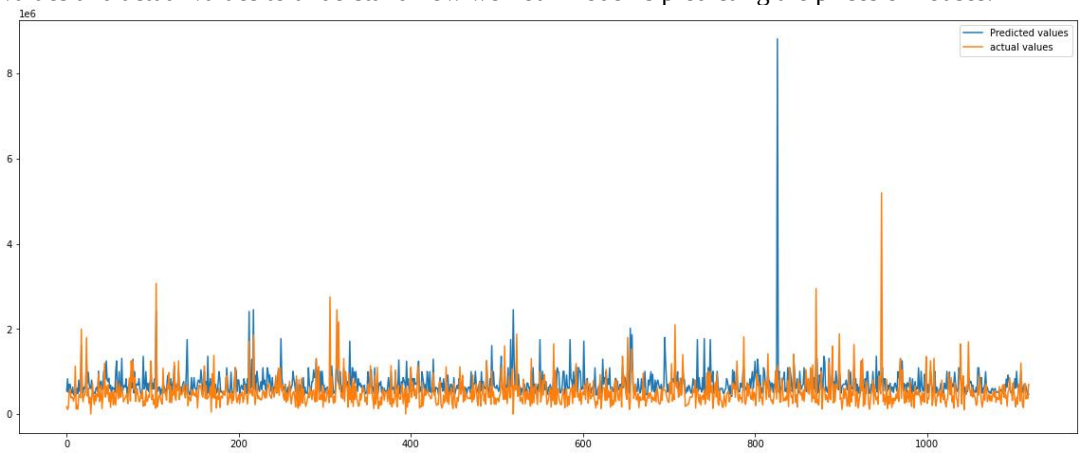
Notice that except for one house, all other houses are located near to each other.

Splitting the dataset

As we understand the dataset clearly, we can now split it into testing and training parts to that we can train the model. Now we can split the dataset into the testing and training parts. We will specify 70% of the data for training and 30% for the testing part. Once the splitting is complete, we can then start training the model.

AdaBoost regressor

We can train the model using AdaBoost regressor using its default values. Once the training is complete, we can then predict values by passing the testing dataset. Let's visualize predicted values and actual values to understand how well our model is predicting the prices of houses.



The graph shows that our predictions are a little bit lower than the actual values in most of the cases. We can evaluate the model's performance using the R2 score. The R2 score is usually between one and zero. The higher the value will be, the better the model is in making predictions.

```
R-square score is : -0.5859793262439643
```

R2-score is negative only when the chosen model does not follow the trend of the data, which means our model was not trained properly.

GridSearchCV to find optimum stump trees

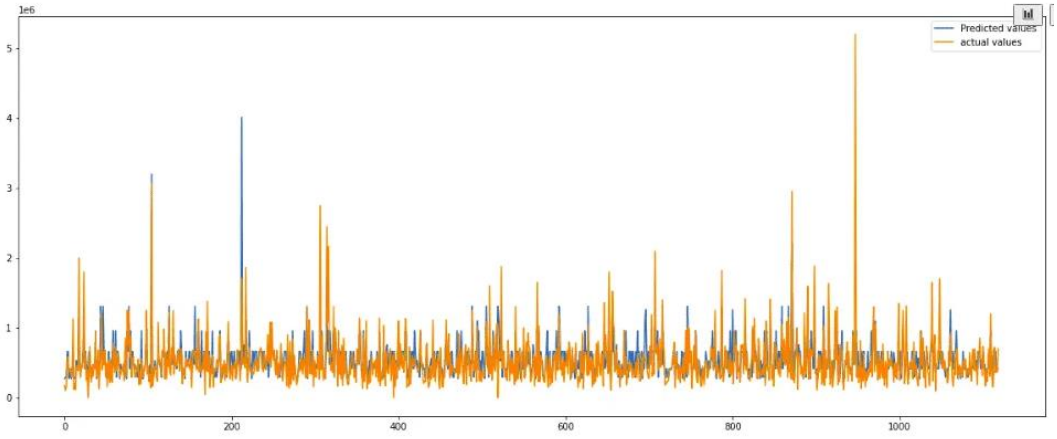
Now let's use the GridSearchCV helper to find the optimum stump trees number.

```
The best estimator returned by GridSearch CV is: AdaBoostRegressor(n_estimators=11)
```

This returns that the optimum number of stump trees for the given data is 4.

AdaBoost regressor using optimum stump trees

Let's again train our model using the optimal number of stump trees which we found in the previous section of the article. We will visualize the actual and predicted values of the model again.

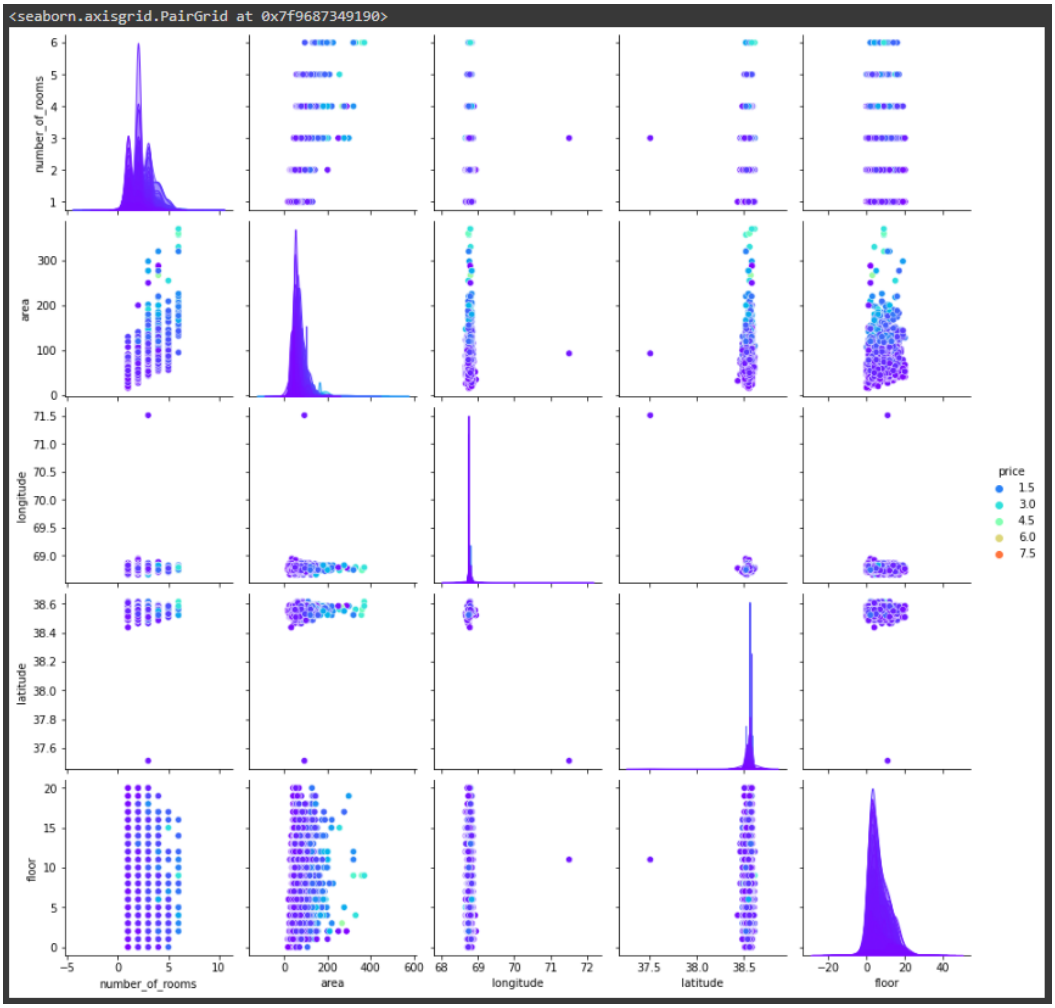


This time we can see that our model has predicted well as compared to the previous one. Let's calculate the R2-score as well.

```
R score is : -0.1267792522442921
```

This shows that our model has performed pretty well this time.

The following is the pairplot of the dataset.



Summary

AdaBoost was the first really successful boosting algorithm developed for the purpose of binary classification. In this article, we've covered how the AdaBoost algorithm implementation and demonstrated how to use it for solving regression and classification problems.

Reference:

[1] Cano, A. C. (2021, August 9). AdaBoost from scratch. Medium. Retrieved December 1, 2022, from https://towardsdatascience.com/adaboost-from-scratch-37a936da3d50

[2] Changyou Chen Lecture Slides

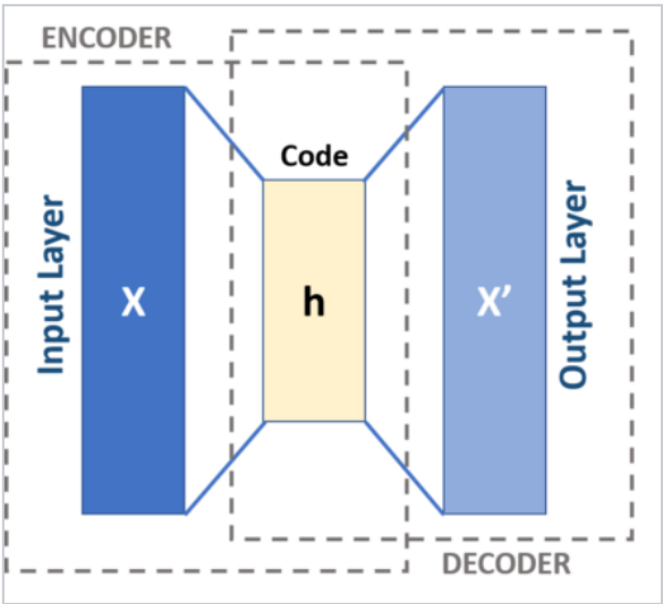
[3] Learning, U. C. I. M. (2016, September 27). Iris species. Kaggle. Retrieved December 1, 2022, from https://www.kaggle.com/datasets/uciml/iris?resource=download

[4] Wikimedia Foundation. (2022, October 27). Adaboost. Wikipedia. Retrieved December 1, 2022, from https://en.wikipedia.org/wiki/AdaBoost

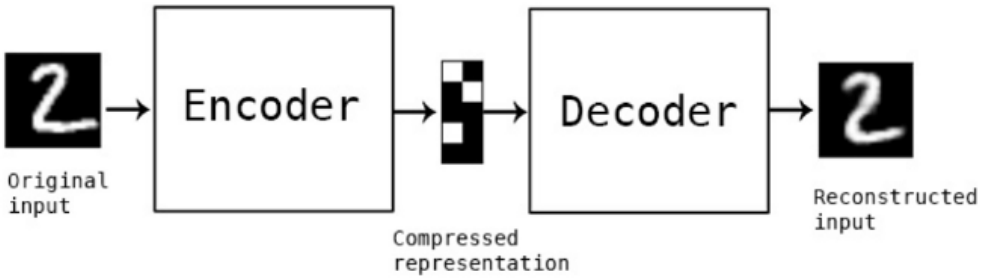
[5] Brownlee, J. (2021, April 26). How to develop an ADABOOST ensemble in python. MachineLearningMastery.com. Retrieved December 1, 2022, from https://machinelearningmastery.com/adaboost-ensemble-in-python/

4 Autoencoder

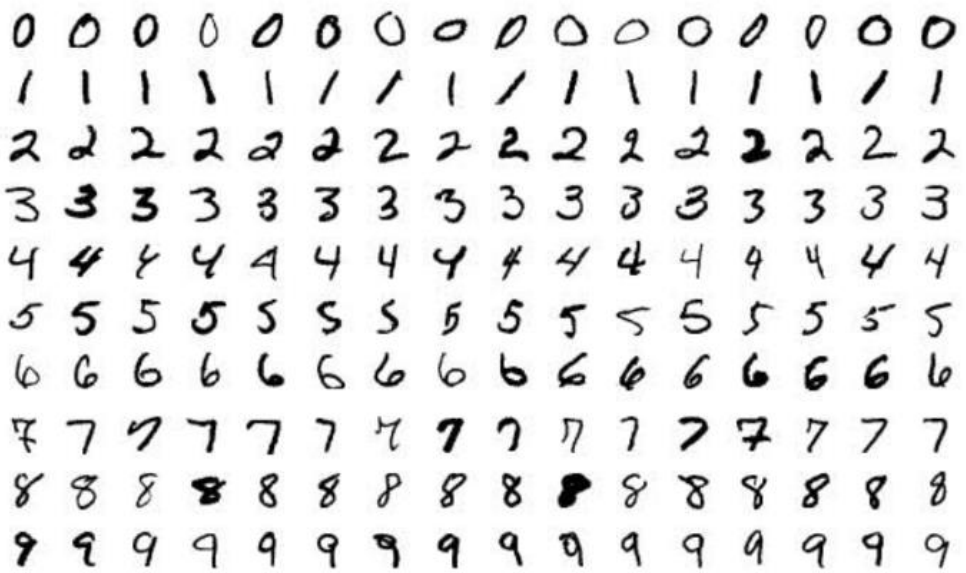
With autoencoders, we employ neural networks for representation learning, which is a form of unsupervised learning approach. They are frequently referred to as self-supervised algorithms as the label used to train example x will be x itself. However, because there are no categorization labels, they fall under the unsupervised category.



In the most basic terms, an autoencoders is an artificial neural network that takes in input, and then outputs a reconstruction of this input. Based on everything we've learned so far on neural networks, this seems pretty strange, but let's explain this idea further using an example.



Suppose we have a set of images of handwritten digits, and we want to pass them through an autoencoder. Remember, an autoencoder is just a neural network. This neural network will take in this image of a digit, and it will then encode the image. Then, at the end of the network, it will decode the image and output the decoded reconstructed version of the original image. The goal here is for the reconstructed image to be as close as possible to the original image.



we can think of the loss function for this autoencoder as measuring how similar the reconstructed version of the image is to the original version. The more similar the reconstructed image is to the original image, the lower the loss. Since this is an artificial neural network after all, we'll still be using some variation of SGD during training, and so we'll still have the same objective of minimizing our loss function. During training, our model is incentivized to make the reconstructed images closer and closer to the originals.

Any machine learning algorithm's effectiveness is dependent on the data representation since different data representations might conceal various elements, characteristics, and information that data may include. Because of this, a substantial percentage of the actual work required in implementing machine learning algorithms goes into building preprocessing pipelines and data transformations that give the data a representation that can allow efficient machine learning. Such feature engineering is important but time-consuming, emphasizing the weakness in the learning algorithms used today that prohibits them from organizing and retrieving discriminative information from the data.
Representation learning involves embedding the components and characteristics of the original data in order to better interpret, display, and extract meaningful information. Additionally makes it easier to retrieve pertinent data for building classifiers or other predictions. In probabilistic models, a good representation of the posterior distribution of the underlying explanatory components for the observed input is typically used.
Knowledge autoencoders will be simpler now that we have a better understanding of representation learning. A particular class of neural network called an autoencoder makes an effort to closely resemble its input to its output. It starts with an input, reduces it into a code-like reduced representation, and then reconstructs the output using this representation. The code that was generated is known as a latent-space representation, and it will be a condensed description of the input.
An autoencoder consists of 3 components:

1. Encoder
2. Code
3. Decoder

The encoder compresses the input and after that it produces the code, and then the decoder reconstructs the input using this code.

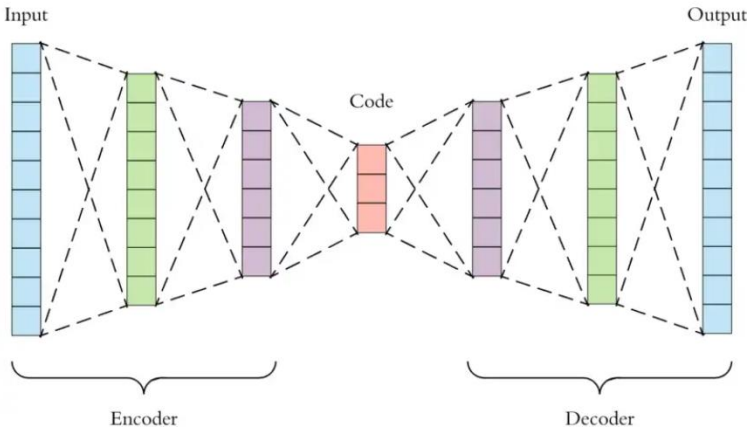
To construct an autoencoder there are 3 things required: An encoding method, decoding method and a loss function to compare the output with the expected target.

Below are the properties of an autoencoder:

- **Data Specific:** Autoencoders can only effectively compress data that is identical to that on which they have been trained. They differ from a typical data compression technique like gzip because they learn features relevant to the provided training data. Therefore, it is unrealistic to think that a handwritten digit autoencoder to compress landscape photographs.
- **Lossy:** The output of the autoencoder will not be exactly the same as the input, it will be a close but degraded representation.
- **Unsupervised:** We don't need to do anything elaborate to train an autoencoder; we can simply feed it the raw input data. Since autoencoders don't require explicit labels to train on, they are regarded as an unsupervised learning technique. But since they create their own labels from the training data, they can actually be considered self-supervised.

Architecture

Let's understand the architecture of an autoencoder. Both the encoder and decoder are fully connected feedforward neural network. A single layer of an ANN with the dimension of our choice is called code. A hyperparameter that we specify prior to training the autoencoder is the number of nodes in the code layer (also known as the code size).



Above is the detailed visualization of an autoencoder. To create the code, the input first goes via the encoder, which is a fully-connected ANN. The output is subsequently produced solely using the code by the decoder, which has a similar ANN structure. Getting a result that matches the input is the aim. Keep in mind that the architecture of the encoder and decoder are identical. Although not required, this is frequently the case. The only prerequisite is that the input and output dimensions must match. Any object in the middle is playable.

Before training an autoencoder there are 4 hyperparameters that need to be set.

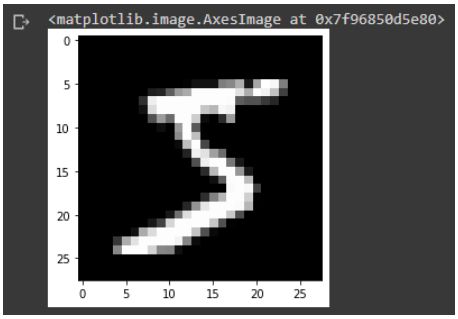
- **Code size**
  This is the number of nodes in the middle layer. Smaller size usually results in more compression.
- **Number of layers**
  The autoencoder can be as deep as we would want.
- **Number of nodes per layer**
  Additionally, you can decide how many units or nodes you want in your levels. Usually, when we add more layers to the encoder, there are fewer nodes, and the opposite is true for the decoder.

- **Loss function**
  We either employ binary cross entropy or mean squared error (mse). Cross entropy is typically used if the input values fall within the [0, 1] range; otherwise, mean square error is employed.

Keras with TensorFlow — Data Processing for Autoencoder Training

Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images). We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784. Below are the shapes of x_train and x_test. The actual data looks as follows:

```
(60000, 784)
(10000, 784)
```



Create An Autoencoder with TensorFlow's Keras API:
We'll start simple, with a single fully-connected neural layers

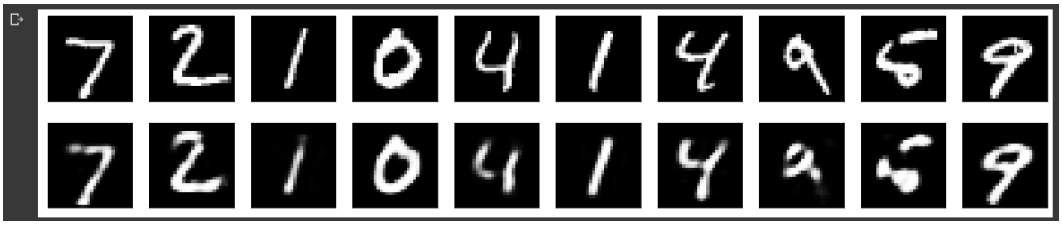
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0

 dense (Dense)               (None, 32)                25120

 dense_1 (Dense)             (None, 784)               25872

=================================================================
Total params: 50,992
Trainable params: 50,992
Non-trainable params: 0
_________________________________________________________________
```

Train An Autoencoder with TensorFlow's Keras API
First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer. Now let's train our autoencoder for 20 epochs.

```
Epoch 1/20
235/235 [==============================] - 6s 22ms/step - loss: 0.2772 - val_loss: 0.1911
Epoch 2/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1723 - val_loss: 0.1542
Epoch 3/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1447 - val_loss: 0.1339
Epoch 4/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1291 - val_loss: 0.1219
Epoch 5/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1192 - val_loss: 0.1139
Epoch 6/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1124 - val_loss: 0.1084
Epoch 7/20
235/235 [==============================] - 3s 13ms/step - loss: 0.1073 - val_loss: 0.1038
Epoch 8/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1034 - val_loss: 0.1003
Epoch 9/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1003 - val_loss: 0.0978
Epoch 10/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0982 - val_loss: 0.0960
Epoch 11/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0967 - val_loss: 0.0948
Epoch 12/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0957 - val_loss: 0.0939
Epoch 13/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0950 - val_loss: 0.0934
Epoch 14/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0946 - val_loss: 0.0930
Epoch 15/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0943 - val_loss: 0.0928
Epoch 16/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0941 - val_loss: 0.0926
Epoch 17/20
235/235 [==============================] - 4s 19ms/step - loss: 0.0939 - val_loss: 0.0925
Epoch 18/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0937 - val_loss: 0.0923
Epoch 19/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0936 - val_loss: 0.0923
Epoch 20/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0935 - val_loss: 0.0921
<keras.callbacks.History at 0x7f969325a7f0>
```

After 20 epochs, the autoencoder seems to reach a stable train/validation loss value of about 0.10.

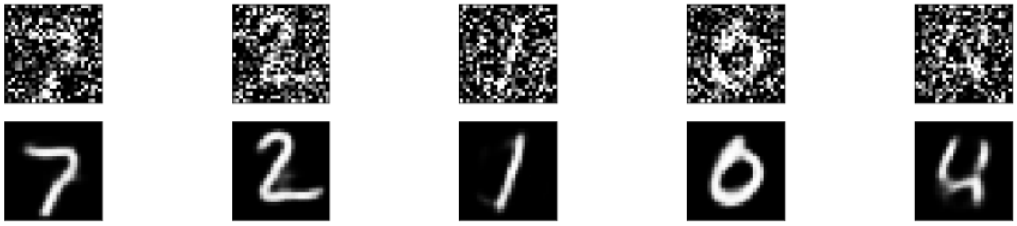
Create A Visualization for Autoencoder Predictions
We can try to visualize the reconstructed inputs and the encoded representations. We will use Matplotlib.



Here's what we get. The top row is the original digits, and the bottom row is the reconstructed digits. We are losing quite a bit of detail with this basic approach.

Denoising Autoencoder:
Now we will see how the model performs with noise in the image. What we mean by noise is blurry images, changing the color of the images, or even white markers on the image.

Now the images are barely identifiable and to increase the extent of the autoencoder, we will modify the layers of the defined model to increase the filter so that the model performs better and then fit the model.



We have gone through the structure of how autoencoders work and worked with 2 types of autoencoders. There are multiple uses for autoencoders like dimensionality reduction image compression, recommendations system for movies and songs and more. The performance of the model can be increased by training it for more epochs or also by increasing the dimension of our network.

Reference:

[1] Wikimedia Foundation. (2022, November 5). Autoencoder. Wikipedia. Retrieved December 1, 2022, from https://en.wikipedia.org/wiki/Autoencoder

[2] Changyou Chen Lecture Slides

[3] Badr, W. (2019, July 1). Auto-encoder: What is it? and what is it used for? (part 1). Medium. Retrieved December 1, 2022, from https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726

[4] Intro to autoencoders ： Tensorflow Core. TensorFlow. (n.d.). Retrieved December 1, 2022, from https://www.tensorflow.org/tutorials/generative/autoencoder

[5] Chollet, F. (n.d.). The keras blog. The Keras Blog ATOM. Retrieved December 1, 2022, from https://blog.keras.io/building-autoencoders-in-keras.html