Hidden Markov Models Report

Darryl Vas Prabhu
Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260
dvasprab@buffalo.edu

1 Hidden Markov Models

[1] There are 2 different interpretations of probability. One is called the frequentest interpretation. In this view, probabilities represent long run frequencies of events that can happen multiple times. For example , if we flip the coin multiple times, we expect it to land heads about half the time. The other is called the Bayesian interpretation of probability and is the topic of interest here. We quantify our uncertainty or ignorance, and this model is fundamentally related to information rather than repeated trials .In the Bayesian view, we believe a coin for example, is equally likely to land heads or tails on the next toss.

The Hidden Markov model is one of the most widely used probabilistic graphical models in applications like biomedical informatics, natural language processing, speech recognition etc. These ideas seem complex, however under the hood, every random variable is related and linked by what we call as interactions. For example in language models, Figure 1 we make an certain interactions between the words in a sentence and their relation. Based on these interactions between the states of the words we can predict the future word based on the previous outcome or word. In a Markov Chain, future states of variables depend only on the previous state of the variable. The probabilities of the states observed after n iterations is observed. After certain iterations the probability of the states converges and does not change. This is the probability we need to find i.e. the probability of the states which is same as the initial probability states presumed.

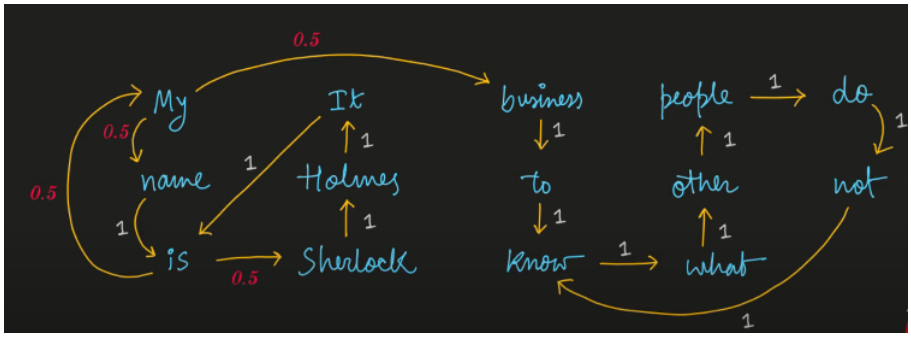


Figure 1: Markov Chain Example

HMM's is a model on the basic of Markov chains. Bayes theorem helps us in calculating the most probable hidden state, given observed variables. The hidden state variables only depend on the previous states outcome. The observed variables depend on the hidden state variable only that instance of the iteration. For example Figure 2: If we consider weather as the hidden states and the mood of a person which is observed by us as the observed variable, we have the following representation as shown in figure. Ex : The probability 0.2 above the green arrow from rainy to sunny

[1] Hidden Markov Models

indicates the probability of today being rainy given that yesterday it was sunny. The transition matrix and the observed variables is indicated in the green and red Figure 2

The weather is hidden from us since we live in a different town , however understanding the person's mood on that day , we can predict the weather for that day. The probability of his mood determines what the weather on that day is and nothing else. Hidden Markov Model is dependent on Hidden Markov chain (transition matrix) + Observed variables (emission matrix) . An combination of initial hidden states and it's observed variables is assumed at 0th iteration . The joint probabilities of the hidden state is calculated given the observed variables. Next the most likely weather sequence of the observed variable is calculated , by computing the probability corresponding to each state sequence , and find the maximum joint probability. The sequence which gives the maximum probability is the hidden which is most likely to determine the person's mood.

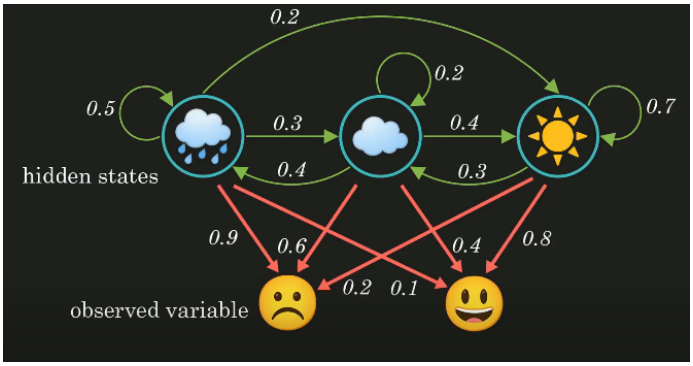


Figure 2: HMM exmample of weather and persons mood

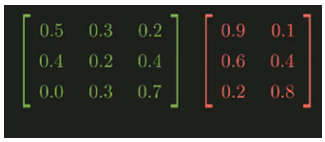


Figure 3: Transition and emission matrix

The formal mathematics behind the hidden Markov model is summarized in the figure Figure 2. Various methodologies can be employed to compute the maximum hidden states joint probability given the observed variables Ex: Viterbi algorithm Figure 5 employed in our experiment.

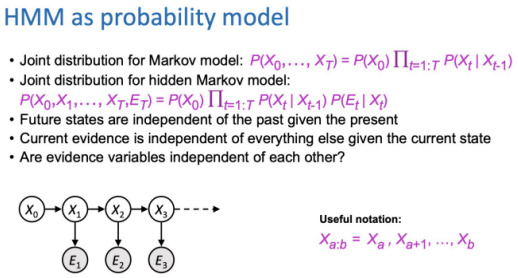


Figure 4: HMM model

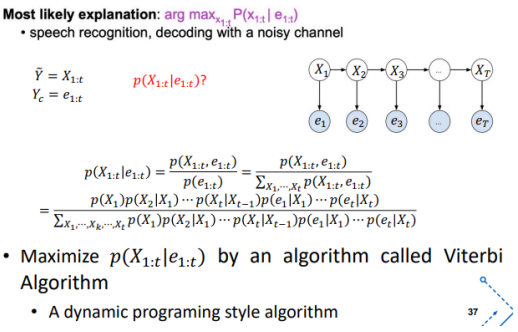


Figure 5: Viterbi Algorithm

2 Experiments

For demonstrating the experiment, since HMM is most likely to be used while there is a varying time series distribution, we make use of the dollar rupee transition over the years to see how it was changed.

The goal is to see if we can separate out different COVID Positive cases that caused a fluctuation based on their frequency of occurrence. The idea is that each boundary may cause a increase or decrease with a particular distribution of waiting times depending on how active it is. This might help us predict Positive cases increase.

```
dataframe = pd.read_csv('New_York_State.csv')
```

```
dataframe
```

	Test Date	County	New Positives	Cumulative Number of Positives	Total Number of Tests Performed	Cumulative Number of Tests Performed	Test % Positive	Geography
0	12/1/2022	Erie	88	259844	1194	4240709	8.24%	COUNTY
1	11/30/2022	Erie	138	259756	1735	4239515	7.30%	COUNTY
2	11/29/2022	Erie	149	259618	1404	4237780	9.27%	COUNTY
3	11/28/2022	Erie	82	259469	1057	4236376	6.61%	COUNTY
4	11/27/2022	Erie	68	259387	1045	4235319	6.35%	COUNTY
...	...	...	...	...	...	...	...	...
1001	3/5/2020	Erie	0	0	6	12	0.00%	COUNTY
1002	3/4/2020	Erie	0	0	6	6	0.00%	COUNTY
1003	3/3/2020	Erie	0	0	0	0	0.00%	COUNTY
1004	3/2/2020	Erie	0	0	0	0	0.00%	COUNTY
1005	3/1/2020	Erie	0	0	0	0	0.00%	COUNTY

1006 rows × 8 columns

Figure 6: Dataset for COVID

The Poisson distribution is useful in this case since it determines a distribution of a random event occurring over a time-interval. There is no normal distribution of events here, however it is known to have a constant mean rate.

The New Positive Covid data we have used is taken from COVID-19 Data in New York | Department of Health for the past 2 years. Data processing was performed to get the data for past 700 days.

The plot of the data shows a spike in the positive cases and probably stabilizing after vaccinations were given.

We fit a Poisson Hidden Markov Model to the data with 10 different states and 4 components. For each component and hidden state we find the sequence of states to fit the mode and see the convergence. The convergence score is calculated such that the model fits according to the initial state chosen. After multiple iterations, the score converges at a point. The best model with the maximum score is selected and the sequence of states which gives rise to this model is predicted using the Viterbi dynamic algorithm,

```
dataframe = dataframe[['New Positives','Test Date']]
```

```
dataframe
```

	New Positives	Test Date
0	88	12/1/2022
1	138	11/30/2022
2	149	11/29/2022
3	82	11/28/2022
4	68	11/27/2022
...	...	...
1001	0	3/5/2020
1002	0	3/4/2020
1003	0	3/3/2020
1004	0	3/2/2020
1005	0	3/1/2020

1006 rows × 2 columns

Figure 7: Filtered data of Positive cases for 700 days

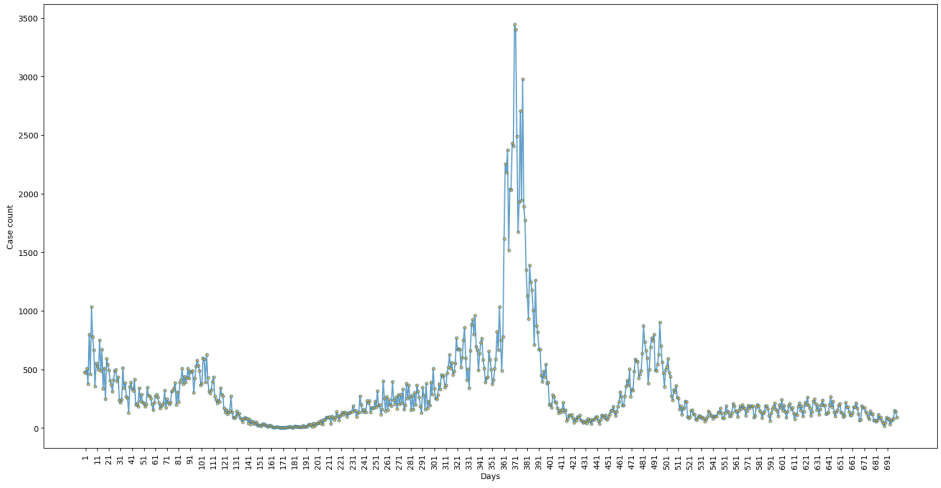


Figure 8: Plot of the data

In our case the best model converged to initial state at score -14051.05636957947

```
fig, ax = plt.subplots()
ax.plot(model.lambdas_[states], ".-", ms=6, mfc="red")
ax.plot(arr)
ax.set_title('States compared to generated')
ax.set_xlabel('State')
```

```
Text(0.5, 0, 'State')
```

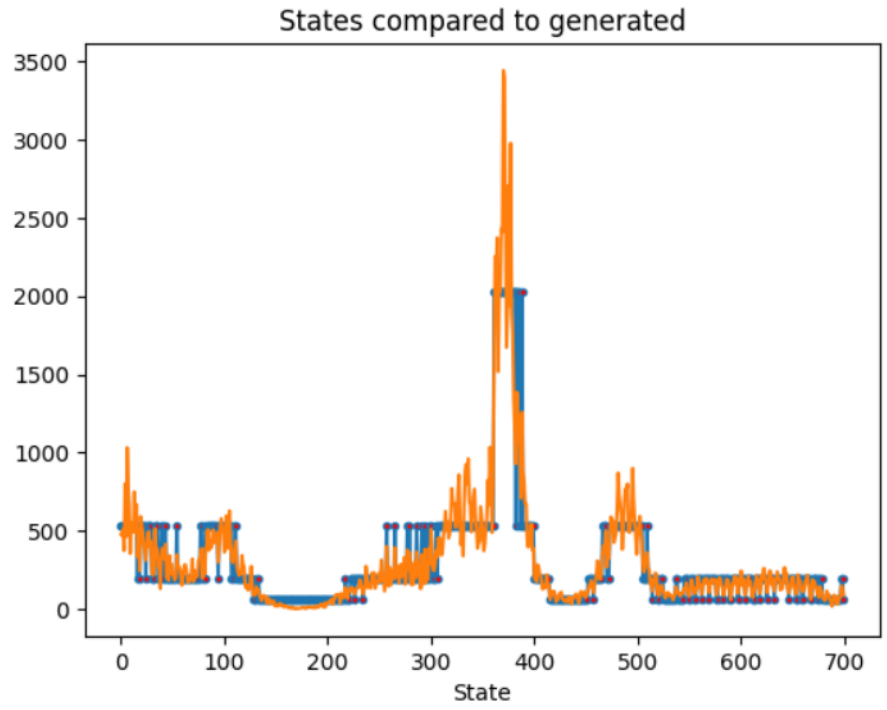


Figure 10: States generated

Plotting the series of states of new positive cases with the waiting time , shows the fluctuation on positive COVID cases. We can see, that the model with the maximum likelihood had different states which reflects the sharp increase in number of cases.

After approximately 500 days since January 1st 2021 , the situation of COVID stabilized with the positive cases not varying much. If we look at our transition matrix, we can see that the off-diagonal terms are light colored, indicating that the state transitions are rare and it's unlikely that there will be further increase in the COVID spike rate.

Looking the distribution of Positive cases compared to our number of Positive case count values, we can see that our model fits the distribution quite well, replicating results from the reference.Figure 12

```
Converged: True       Score: -46288.105981377004
Converged: True       Score: -45936.13354403511
Converged: True       Score: -46019.20451906189
Converged: True       Score: -45942.37439665158
Converged: True       Score: -46021.056799993115
Converged: True       Score: -45951.002559702545
Converged: True       Score: -45869.643024079196
Converged: True       Score: -31704.735095188407
Converged: True       Score: -28024.479404250465
Converged: True       Score: -21432.666262829756
Converged: True       Score: -21586.01223083583
Converged: True       Score: -29779.066029898928
Converged: True       Score: -21326.860835911735
Converged: True       Score: -30080.498921598708
Converged: True       Score: -21316.606713466823
Converged: True       Score: -31054.318075010004
Converged: True       Score: -21347.336250033943
Converged: True       Score: -22798.87990507437
Converged: True       Score: -17000.225101322972
Converged: True       Score: -14256.20960722582
Converged: True       Score: -15755.6011500442
Converged: True       Score: -14769.125848450047
Converged: True       Score: -14051.05636957947
Converged: True       Score: -15398.847094835019
Converged: True       Score: -14141.202297711445
Converged: True       Score: -23996.262266706854
Converged: True       Score: -14063.335063765084
```

Figure 9: Covergence Score

```
fig, ax = plt.subplots()
ax.imshow(model.transmat_, aspect='auto', cmap='OrRd')
ax.set_title('Transition Matrix')
ax.set_xlabel('State To')
ax.set_ylabel('State From')
```

```
Text(0, 0.5, 'State From')
```

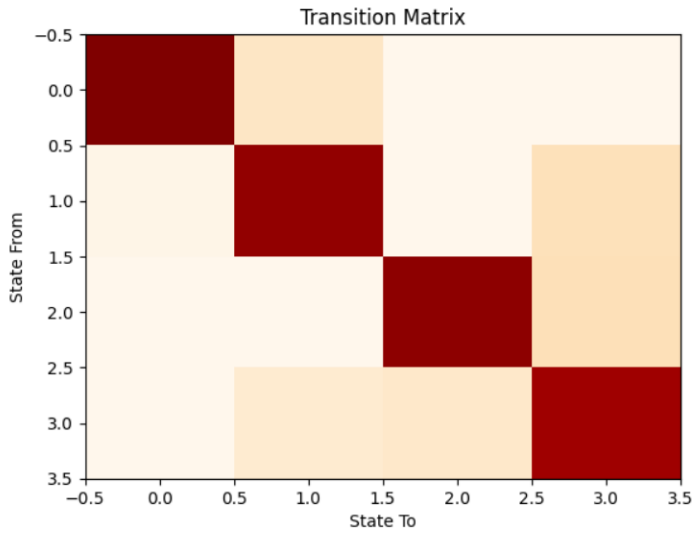


Figure 11: Transition-matrix

```
plt.show()
```

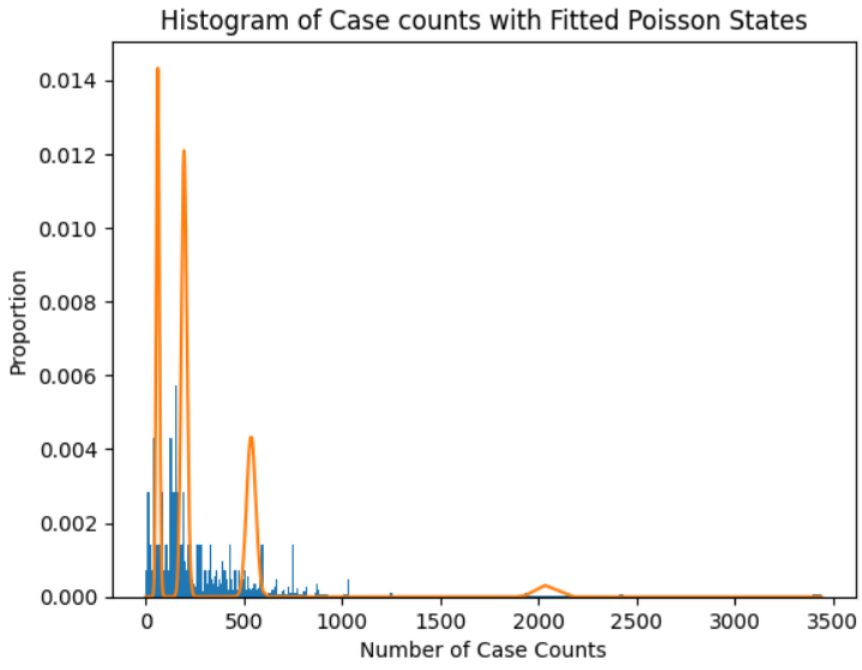


Figure 12: Histogram of Case count with Poisson State

References

[1] *Probabilistic Machine Learning An Introduction*. The MIT Press, 2022.

[2] Changyou Chen. Hmm learn. https://github.com/cchangyou/hmmlearn/tree/main/examples.

[3] MatplotLib Docs. Color map. https://matplotlib.org/stable/tutorials/colors/colormaps.html.

[4] NY Gov. Dataset covid-19. https://coronavirus.health.ny.gov/covid-19-data-new-york.

[5] Will Koehrsen. Poisson distribution. https://builtin.com/data-science/poisson-process.

[6] Normalized Nerd. Hidden markov model clearly explained! part - 5. https://youtu.be/RWkHJnFj5rY.

Random Forest Report

Darryl Vas Prabhu
Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260
dvasprab@buffalo.edu

1 Random Forest

[1] A big part of the machine learning umbrella falls under the classification model. There are various models such as the classic binary logistic regression, support vector machines, decision trees. All these models are used to classify data belonging to a target class. However there is a methodology of training known as ensemble learning which is used to use multiple weak models to obtain a strong machine learning model. Ensemble methods use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone. If a model classifies data into categories it is called a Classification model and if the model is used to predict numerical values, then it is a regression model. Random Forest makes use of ensemble learning models and is mainly used in classification and regression problems.

We cannot touch upon random forests without exploring the basis of Random Forest i.e. Decision trees. In decision trees , each feature is used as the root node, 2nd level root node and third level root nodes and so on until there are only leaves left. The selection of the root nodes, sub-root nodes or features and their priorities is performed based on quantifying and employing various methodologies such as Gini Impurity, Entropy and Information gain. In Gini Impurity , we explore a feature and find the count of yes or no for the feature with respect to the target label. A left indicates True and right indicates False in the decision tree. Say if a person loves Popcorn or not and we want to predict if he watched the movie 'Cool as Ice', we check if he loves pop corn, which directs the node to left and then check it against target variable if he watched Cool as Ice or not Figure 1. In this way the decision tree is checked for all samples for this feature. The Gini Impurity is calculated for this variable and the same is performed for all the other features in the decision tree. The feature with the lowest Gini Impurity is selected as the root feature node. Recursively the same is calculated for the children and sub children building the decision tree until we reach leaves which determine the category. The output of the leaves which has the most values determines the category or target that the new data sample belongs to.

In practise, it is found that decision trees work well with data used to model them, but do show signs of inaccuracy with new data samples. This is why the Random Forest model was born. Random Forest uses the underlying simplistic decision tree model and adds to it a flexibility to reach great accuracy. Random Forest consists of 2 steps.

- Creating the randomly selected data samples same as the size of the original data, however duplicates are allowed.
- Creating a decision tree , however using only a subset of the input's samples

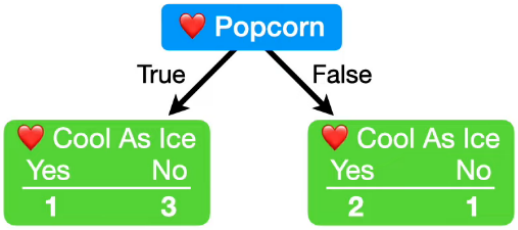


Figure 1: Example of building decision tree

[1] Random Forest

selected arbitrarily, to perform the nodes splitting and recursively using decision tree algorithm.

Repeat 1 and 2. This is called as bagging.

In the subsequent iterations , the randomly sampled selection of features are done from features that are not previously used. This is recursively performed to build a variety of decision trees.

We run all the inputs through each of the decision trees to predict the outcome. For each input , an observation is made on the label which received in the most votes in the target variable. The accuracy of the data is tested on generally 1/3 of the original data which is not selected in the training of the model. This is called the "Out of Bag Dataset" . The Random forest is tested on this data and the count of how well each decision tree performs is noted. The higher the value of vote , the better is the classification and the better is the decision tree and considered the final prediction of the random forest algorithm. A visual representation of the Random Forest is given in Figure 2

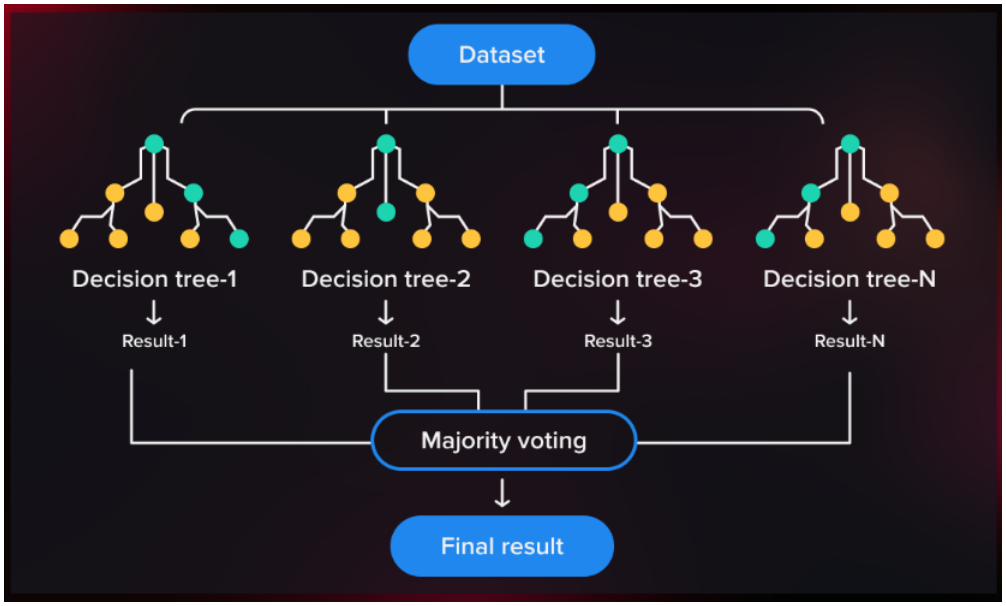


Figure 2: Visual Representation of Random Forest

The algorithm for the bagging procedure is given in Figure 3

Algorithm 1 Bagging

1: Let n be the number of bootstrap samples
2:
3: **for** i=1 to n **do**
4: Draw bootstrap sample of size m, $\mathcal{D}_i$
5: Train base classifier h_i on $\mathcal{D}_i$
6: $\hat{y} = mode\{h_1(\mathbf{x}), ..., h_n(\mathbf{x})\}$

Figure 3: Bagging Algorithm

2 Experiments

For demonstrating the Random Forest Experiment, data-set from the kaggle of patients to predict diabetes was used Figure 4

```
dataset_2
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
...	...	...	...	...	...	...	...	...	...
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

768 rows × 9 columns

Figure 4: Original Dataset

Data processing is used to filter the data required as inputs which in our case are : Glucose and BMI. The target variable 'Outcome' outputs 1 for diabetes ,and 0 for no diabetes. Figure 5 A snippet of the data-set and the correlation matrix given for reference Figure 6

```
X_1 = dataset_2[['Glucose','BMI']]
X_1.head()
```

	Glucose	BMI
0	148	33.6
1	85	26.6
2	183	23.3
3	89	28.1
4	137	43.1

Figure 5: Dataset filtered

```
sns.heatmap(dataset_2.corr())
```

```
<AxesSubplot: >
```

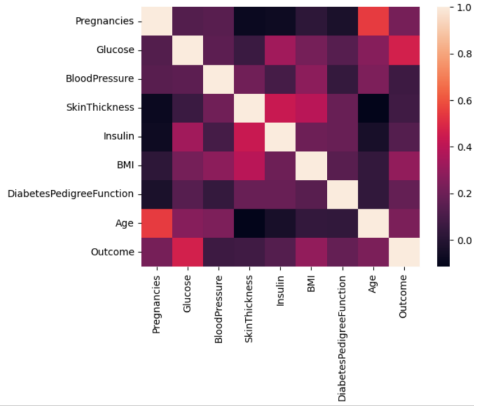


Figure 6: Correlation Matrix

Scikit-learn's model selection is used for splitting the training and testing dataset. Then we make used of the RandomForestClassifer Figure 7 that fits a number of decision tree classifiers on various sub-samples of the data-set and uses averaging to improve the predictive accuracy and control over-fitting. For our case we have take 10 decision trees as the number of estimators and selection of features as nodes in decision trees uses the entropy method.

After training and testing the data set, we find that the number of true positives and negatives are higher , indicating that the model is performing well. We have 114 true positives and 35 true negatives out of the 192 testing data points as denoted in the confusion matrix Figure 8

Finally we use the matplotlib's contour plot to layout the random forest classification. Most of the dots are classified correctly according to the diabetes outcome.We do have certain misclassified datapoints, which may be special cases of having diabetes and does not fall inline with the general pattern of the random forest.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
classifier.fit(X_train, y_train)
```

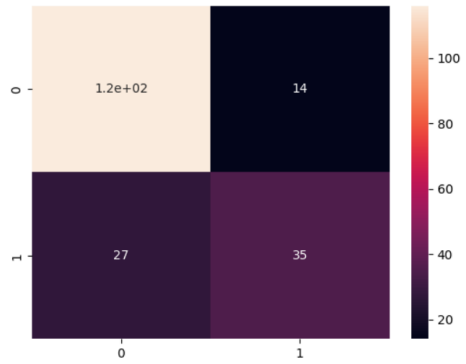
```
RandomForestClassifier
RandomForestClassifier(criterion='entropy', n_estimators=10, random_state=0)
```

```
y_pred = classifier.predict(X_test)
```

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

Figure 7: RandomForestClassifier from scikit-learn

```
sns.heatmap(cm,annot=True)
<AxesSubplot: >
```




```
cm
array([[116,  14],
       [ 27,  35]], dtype=int64)
```

Figure 8: Confusion Matrix y_pred vs y_test

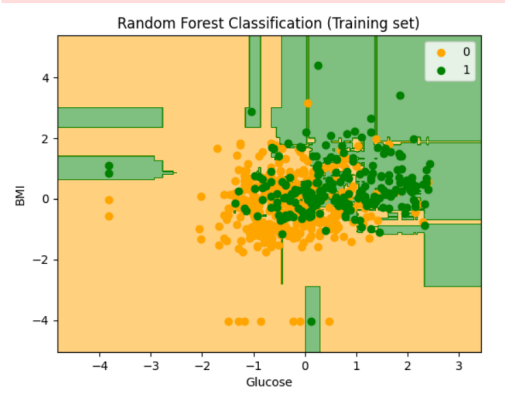


Figure 9: Random Forest Classification of Training-Set

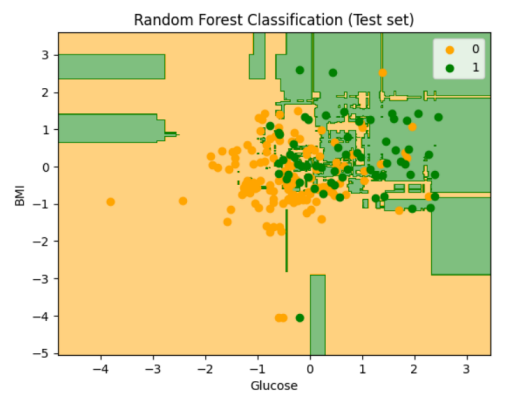


Figure 10: Random Forest Classification of Testng-Set

References

[1] *Probabilistic Machine Learning An Introduction*. The MIT Press, 2022.

[2] Changyou Chen. Random forest. https://github.com/cchangyou/100-Days-Of-ML-Code/blob/master/Code/Day%2034%20Random_Forest.md.

[3] Kaggle. Dataset. https://www.kaggle.com/datasets/whenamancodes/predict-diabities.

[4] Scikit Learn. Scikit learn documentation. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[5] Inna Logunova. Random-forest-classification. https://serokell.io/blog/random-forest-classification,.

[6] StatsQuest. Decision and classification trees, clearly explained!!! https://www.youtube.com/watch?v=_L39rN6gz7Y.

Adaboost Classifier Report

Darryl Vas Prabhu
Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260
dvasprab@buffalo.edu

1 Adaboost Classifier

[1] Adaboost classifier is an adaptive classifier method in the sense that we use multiple weak learners in order to create a strong learner. However , there is a different methodology employed. Rather than using the same dataset each time, the dataset is weighted at each iteration based on each outcome of the model. Some learner models have more priority or have more votes compared to others. In summary , Adaboost combines multiple weak learners to make a classification. Some learners have more priority over other learners and each learner makes an improvement on itself , from learning the previous learner's error-ed data. In essence, the ordering of features selected for each model matters when using Adaboost. Instead of training the models in parallel, they are trained sequentially. This is the essence of Boosting and the underlying foundation of Adaboost.

Boosting trains a series a low performing algorithms, called weak learners, by adjusting the error metric over time. Weak learners are algorithms whose error rate is slightly under 50%. In Adaboost, we make use of a shortened version of a decision tree called tree stumps. Tree stumps defines a 1-level decision tree. The main idea is that at each step, we want to find the best stump, i.e the best data split that minimizes the overall error. The key points are :

- We combine a lot of weak learners to make the classification.
- Some stumps have more priority than others
- Each stump works on the mistakes made by the previous stump.

An example of a stump is provided in the Figure 1

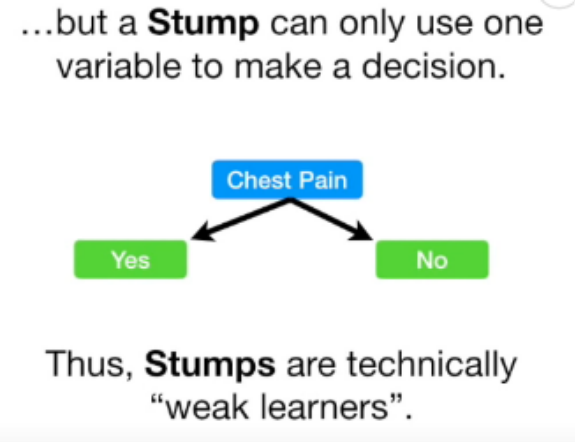


Figure 1: Example of a stump

[1] Adaboost Classifier

Initially we introduce a weight for each input sample and call it the sample weight. Since all weights are equally likely at the initialization stage , we quantify it as 1 divided by the total number of samples. We make stumps for each feature of the sample and see how each feature classifies the sample with respect to the output label. The same is repeated for each of the features of the data, and the best weight that labels the target correctly is found. Now the information gain, entropy or Gini Index is calculated for the three stumps based on which one had the optimal labeling strength and is to be selected as the first stump. To find out how much say or voting power a stump has in the overall classification model, we find out how well the stump classified the samples based on the total number of incorrectly classified samples. Total Error parameter is then used to determine the amount of say or alpha that the stump has in the final classification model.

The amount of say or alpha is given by the formula :

$$\alpha = \frac{1}{2} ln(\frac{1 - TotalError}{TotalError})$$

In other words the amount of say depends on the total error and the total error will always be between 0 and 1. Next the new sample weight is scaled by a factor which is exponentially dependant on the previous weight and the exponential of the amount of say. If the amount of say is positive , then we can say that the new sample weight has a positive slope i.e the last stump did a great job at classifying the data Figure 3 and if the amount of weight is negative the new sample weight will have a negative slope Figure 4. In this way new new weights are assigned to each sample and the new sample weights are normalized. These modified sample weights are used to make the second stump and so on. Since the previous stump might have incorrectly classified certain samples, those incorrectly classified sample's weights are then scaled by the factor of amount of ,say in order for the new stump to improve this sample's classification. In this way the sample's weights are scaled and the same process is repeated till all the samples are correctly classified. This is how errors that are transferred from one stumps to another stump influence the learners to classify the data appropriately.

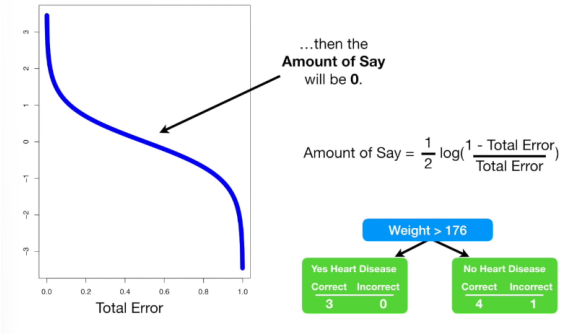


Figure 2: Plot of Amount of say vs Total Error

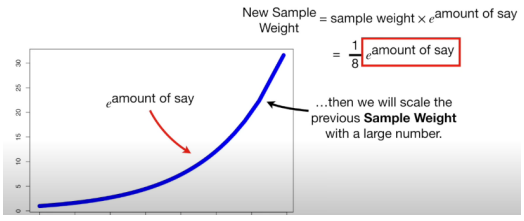


Figure 3: New Sample weight if alpha positive

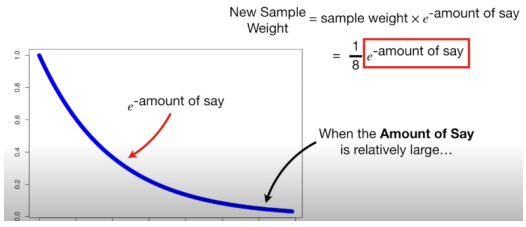


Figure 4: New Sample weight if alpha negative

A summary of the Adaboost algortithms is given below for reference Figure 5

2 Experiments

For demonstrating the Adaboost Experiment, data-set make_hastie_1_2 from the internal library of scikit learn was used Hastie et al. 2009 since it generates data for binary classification.

The last two lines figure 6 indicate that the error is returned. Using the error of this stage, the further stages are modelled as shown in figure. Adaboost classifier is created using a decision tree as base estimator and tested with different number of iterations 7

- Learn a linear combination of base learners $H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$ to minimize the exponential loss function:

Input: Training set $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
Base learning algorithm $\mathfrak{L}$;
Training iterations T.

Procedure :

1: $\mathcal{D}_1(\boldsymbol{x}) = 1/m.$
2: **for** $t = 1, 2, \ldots, T$ **do**
3: $h_t = \mathfrak{L}(D, \mathcal{D}_t);$ — Prediction
4: $\epsilon_t = P_{\boldsymbol{x} \sim \mathcal{D}_t}(h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}));$ — Error
5: **if** $\epsilon_t > 0.5$ **then break**
6: $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right);$ — Weight for the prediction in this round
7: $\mathcal{D}_{t+1}(\boldsymbol{x}) = \frac{\mathcal{D}_t(\boldsymbol{x})}{Z_t} \times \begin{cases} \exp(-\alpha_t), & \text{if } h_t(\boldsymbol{x}) = f(\boldsymbol{x}) \\ \exp(\alpha_t), & \text{if } h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}) \end{cases}$ — Adjust data distribution for the next round
$= \frac{\mathcal{D}_t(\boldsymbol{x}) \exp(-\alpha_t f(\boldsymbol{x}) h_t(\boldsymbol{x}))}{Z_t}$
8: **end for**
Output : $H(\boldsymbol{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x}) \right)$ — Final prediction

Figure 5: Adaboost algorithm summary

```
""" ADABOOST IMPLEMENTATION ================================================"""
def adaboost_clf(Y_train, X_train, Y_test, X_test, M, clf):
    n_train, n_test = len(X_train), len(X_test)
    # Initialize weights
    w = np.ones(n_train) / n_train
    pred_train, pred_test = [np.zeros(n_train), np.zeros(n_test)]

    for i in range(M):
        # Fit a classifier with the specific weights
        clf.fit(X_train, Y_train, sample_weight = w)
        pred_train_i = clf.predict(X_train)
        pred_test_i = clf.predict(X_test)
        # Indicator function
        miss = [int(x) for x in (pred_train_i != Y_train)]
        # Equivalent with 1/-1 to update weights
        miss2 = [x if x==1 else -1 for x in miss]
        # Error
        err_m = np.dot(w,miss) / sum(w)
        # Alpha
        alpha_m = 0.5 * np.log( (1 - err_m) / float(err_m))
        # New weights
        w = np.multiply(w, np.exp([float(x) * alpha_m for x in miss2]))
        # Add to prediction
        pred_train = [sum(x) for x in zip(pred_train,
                                          [x * alpha_m for x in pred_train_i])]
        pred_test = [sum(x) for x in zip(pred_test,
                                         [x * alpha_m for x in pred_test_i])]

    pred_train, pred_test = np.sign(pred_train), np.sign(pred_test)
    # Return error rate in train and test set
    return get_error_rate(pred_train, Y_train), \
           get_error_rate(pred_test, Y_test)
```

Figure 6: Adaboost implementation

You can observe that in the plot of error vs number of iterations , that the error rate decreases with the training of samples which are initially not classified correctly.

```
# Fit Adaboost classifier using a decision tree as base estimator
# Test with different number of iterations
er_train, er_test = [er_tree[0]], [er_tree[1]]
x_range = range(10, 410, 10)
for i in x_range:
    er_i = adaboost_clf(Y_train, X_train, Y_test, X_test, i, clf_tree)
    er_train.append(er_i[0])
    er_test.append(er_i[1])
```

Figure 7: Implementation of Adaboost classifer

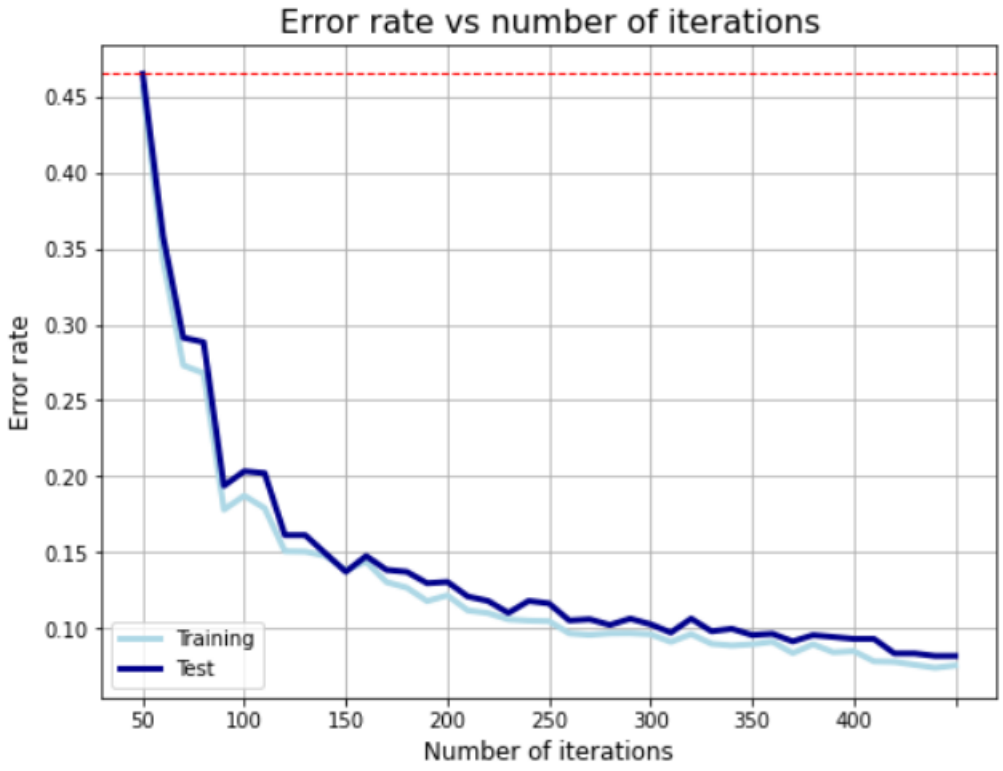


Figure 8: Error Vs Iterations

References

[1] *Probabilistic Machine Learning An Introduction*. The MIT Press, 2022.

[2] Changyou Chen. Adaboost. https://piazza.com/class_profile/get_resource/l6rvn2gsj8r6uw/labhxq6t5w638z.

[3] Changyou Chen. Adaboost classified. https://github.com/cchangyou/adaboost-implementation/blob/master/adaboost.py.

[4] Maël Fabien. Boosting and adaboost clearly explained. https://towardsdatascience.com/boosting-and-adaboost-clearly-explained-856e21152d3e,.

[5] StatsQuest. Adaboost, clearly explained. https://youtu.be/LsK-xG1cLYA.

Autoencoder Report

Darryl Vas Prabhu
Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260
dvasprab@buffalo.edu

1 Autoencoder

[1] Autoencoders are unsupervised neural network models that take in some input and efficiently encodes the data into code. The code here is what we call the hidden layer in nueral network lingo. After encoding of data is performed, the hidden layer can be used as input to decode the data and reconstruct the same input data. So if we're reconstructing the same data , one might imagine what is the use of autoencoder ? In real world, it appears that a collection of data might seem random. Say we have a dataset with a number of transactions and we wish to filter all those transactions that are fraud. Also to make things more complex , there are no labels here since it would take time and effort to manually go through every transaction and find out whether it was fraud or not. This is why we make use of Autoencoders.

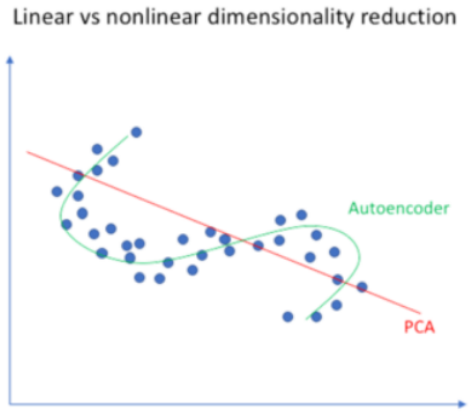


Figure 1: Autoencoder vs PCA

Figure 2: Image Pixelation removed

In real world, data may seem random ,however underlying data does have structures in it. These structures are hidden and can be explored using autoencoders. In other words, autoencoders use only those structures that are necessary to predict the same input. We try to find all those minimal features which can be extracted to reconstuct the same data input and train the model by providing new data or re-iterating the non-linear function on the input to minimize the error. This is the underlying concept of autoencoders. Note that autoencoders use non-linear mapping functions unlike PCA (Principal Component Analysis) in order to reduce the dimension of the data. Hence we can make use of convolution

The encoded data is a type of artificial neural network used to learn efficient coding of unlabeled data (unsupervised learning). It learns to recognize which aspects or features of the data are relevant and reduces the dimensionality of the data in the hidden layer. For Example : The autoencoder learns a representation (encoding) for a set of data, by training the network to ignore insignificant data what

[1] Autoencoder

we call “noise”. Various applications for autoencoders include image compression, speech signal noise reduction, noise reduction in images, anomaly detection, etc, watermark removal etc. Example of pixelation removal is shown in Figure 2

An autoencoder is any neural network which consists of three main components:

- An Encoder or Encoding function (that translates the data into a simpler space, e.g. a hidden layer with fewer nodes than on the input layer),
- a Decoder or decoding function (which reverses this process, e.g. an output layer with equivalently many nodes as the input layer) and
- a distance metric (which measures loss as some distance between the original input and learned representation) Ex : Cross entropy, Mean square error

Given Figure 2 is the representation of an autoencoder as a neural network.

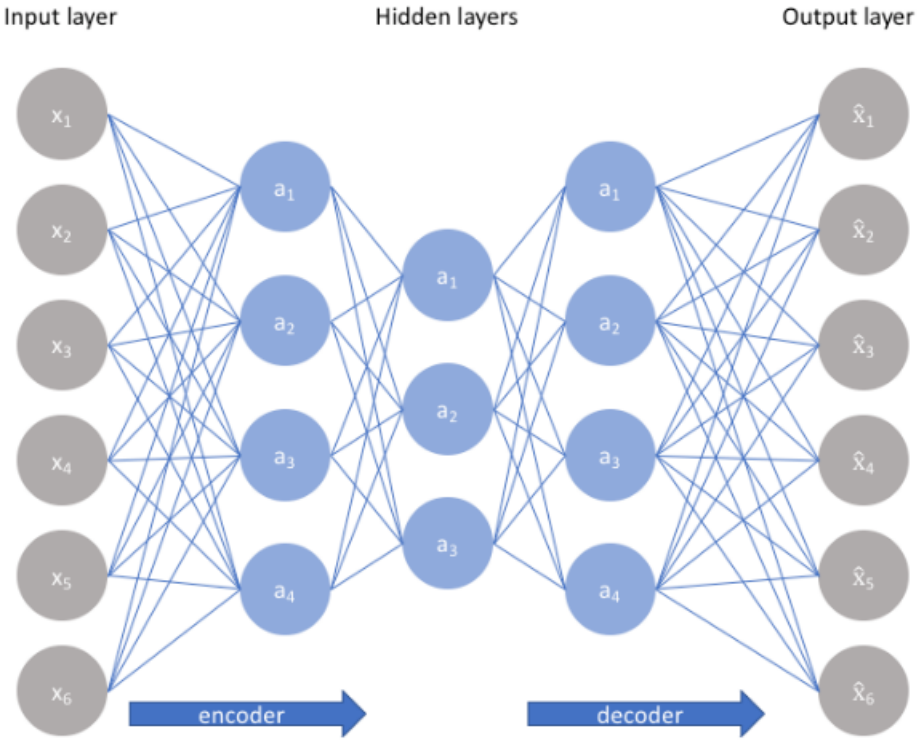


Figure 3: Representation of Autoencoder as Neural network

The underlying mathematics for autoencoders are summarized in the images Figure 6 , Figure 5 below where the input X is mapped to a lower dimensional vector Z also called as latent variable via the encoding layer and a composite function is applied to the latent variable z to decode and reconstuct the output. The model is trained such that the error i.e the error between original data and reconstructed data is minimized. The minimization of error may be performed using Mean Square error or Cross Entropy.

Here are a few examples of autoencoders used in industries.

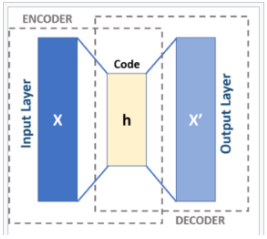


Figure 4: Block diagram of autoencoder

An autoencoder is defined by the following components:

> Two sets: the space of decoded messages $\mathcal{X}$; the space of encoded messages $\mathcal{Z}$. Almost always, both $\mathcal{X}$ and $\mathcal{Z}$ are Euclidean spaces, that is, $\mathcal{X} = \mathbb{R}^m, \mathcal{Z} = \mathbb{R}^n$ for some m, n.
>
> Two parametrized families of functions: the encoder family $E_\phi : \mathcal{X} \to \mathcal{Z}$, parametrized by ϕ; the decoder family $D_\theta : \mathcal{Z} \to \mathcal{X}$, parametrized by θ.

For any $x \in \mathcal{X}$, we usually write $z = E_\phi(x)$, and refer to it as the code, the latent variable, latent representation, latent vector, etc. Conversely, for any $z \in \mathcal{Z}$, we usually write $x' = D_\theta(z)$, and refer to it as the (decoded) message.

Usually, both the encoder and the decoder are defined as multilayer perceptrons. For example, a one-layer-MLP encoder E_ϕ is:

$$E_\phi(\mathbf{x}) = \sigma(Wx + b)$$

where σ is an element-wise activation function such as a sigmoid function or a rectified linear unit, W is a matrix called "weight", and b is a vector called "bias".

Figure 5: Definition of Autoencoder

An autoencoder, by itself, is simply a tuple of two functions. To judge its *quality*, we need a *task*. A task is defined by a reference probability distribution μ_{ref} over $\mathcal{X}$, and a "reconstruction quality" function $d : \mathcal{X} \times \mathcal{X} \to [0, \infty]$, such that $d(x, x')$ measures how much x' differs from x.

With those, we can define the loss function for the autoencoder as

$$L(\theta, \phi) := \mathbb{E}_{x \sim \mu_{ref}}[d(x, D_\theta(E_\phi(x)))]$$

The *optimal* autoencoder for the given task (μ_{ref}, d) is then $\arg\min_{\theta,\phi} L(\theta, \phi)$.

The search for the optimal autoencoder can be accomplished by any mathematical optimization technique, but usually by gradient descent. This search process is referred to as "training the autoencoder".

In most situations, the reference distribution is just the empirical distribution given by a dataset $\{x_1, \ldots, x_N\} \subset \mathcal{X}$, so that

$$\mu_{ref} = \frac{1}{N} \sum_{i=1}^{N} \delta_{x_i}$$

and the quality function is just L2 loss: $d(x, x') = \|x - x'\|_2^2$. Then the problem of searching for the optimal autoencoder is just a least-squares optimization:

$$\min_{\theta,\phi} L(\theta, \phi), \text{where } L(\theta, \phi) = \frac{1}{N} \sum_{i=1}^{N} \|x_i - D_\theta(E_\phi(x_i))\|_2^2$$

Figure 6: Training the Autoencoder

Figure 7: Removal of watermark

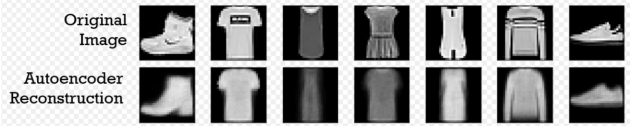


Figure 8: Image classification

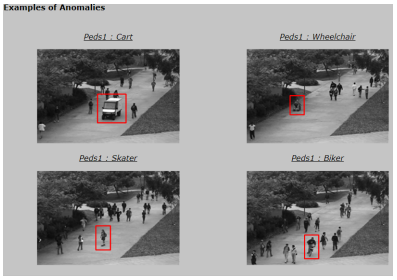


Figure 9: Anomaly detection

2 Experiments

For demonstrating the auto-encoders , the convolutions neural network auto-encoder was used. The data-set used for this experiment is the famous MNIST which is stands for **Modified National Institute of Standards and Technology database** which consists of a large database of handwritten digits that is commonly used for training various image processing systems Figure 10

Figure 10: MNIST Dataset

Torch tensor is used to represent the images in the dataset. A PyTorch Tensor is basically the same as a numpy array. It is used for as a generic n-dimensional array representation to be used for arbitrary numeric computation. The goal is to use convolution networks to compress the image so it can be stored in much less space rather than raw data. The encoder portion will be made of convolutional and pooling layers and the decoder will be made of transpose convolutional layers that learn to "upsample" a compressed representation.

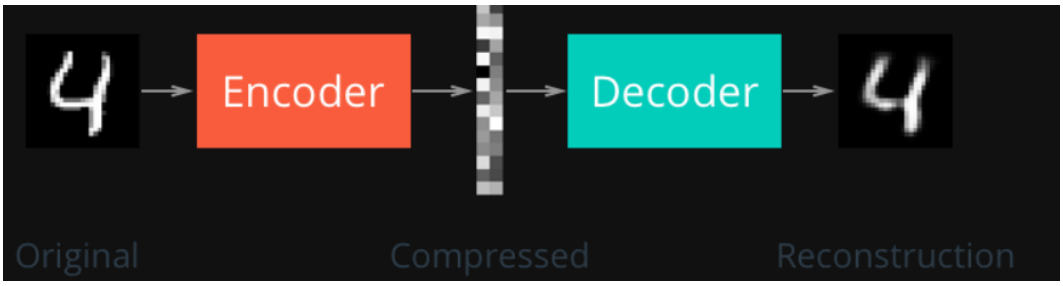


Figure 11: Convolutional Autoencoder

This kind of image compression often holds vital information about an input image and can be used for denoising images or reconstruction and transformation. Data-loaders Figure 13 using torch utils are used inorder to input the training data to be used for the model to learn and the testing data to load for the model to test the data.

```
# convert data to torch.FloatTensor
transform = transforms.ToTensor()

# load the training and test datasets
train_data = datasets.MNIST(root='~/.pytorch/MNIST_data/', train=True,
                                   download=True, transform=transform)
test_data = datasets.MNIST(root='~/.pytorch/MNIST_data/', train=False,
                                  download=True, transform=transform)
```

Figure 13: DataLoaders

```
ax.imshow(img, cmap='BuGn')
```

```
<matplotlib.image.AxesImage at 0x7fa010b1b370>
```

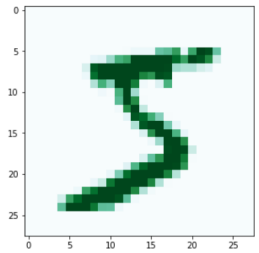


Figure 12: Sample Image in MNIST(color changed)

A sample of the Figure 12 is shown, where we have used cmap to change color. Not that since the MNIST is 1 channel only, it gives hue of one color.

A neural network architecture is defined using the inbuilt convolution functions from torch library.

nn.Conv1d : Applies a 1D convolution over an input signal composed of several input planes.

nn.Conv2d : Applies a 2D convolution over an input signal composed of several input planes. The feedforward neural network is futher subject

```
# conv layer (depth from 1 --> 16), 3x3 kernels
self.conv1 = nn.Conv2d(1, 16, 3, padding=1)
# conv layer (depth from 16 --> 4), 3x3 kernels
self.conv2 = nn.Conv2d(16, 4, 3, padding=1)
# pooling layer to reduce x-y dims by two; kernel and stride of 2
self.pool = nn.MaxPool2d(2, 2)

## decoder layers ##
## a kernel of 2 and a stride of 2 will increase the spatial dims by 2
self.t_conv1 = nn.ConvTranspose2d(4, 16, 2, stride=2)
self.t_conv2 = nn.ConvTranspose2d(16, 1, 2, stride=2)
```

Figure 14: Torch nn Convolution functions

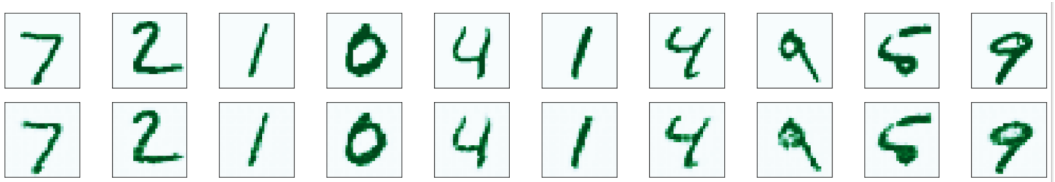


Figure 16: Input images row vs Output images row

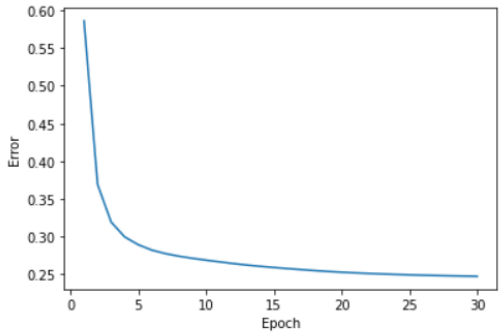


Figure 17: Error Vs Epoch

to ReLU and sigmoid activation functions as well.

A number of epochs are run in order to train the model. Initially the error is high, and after training the model for many number of epochs with different samples, the error rate decreases.Figure 15

```
Epoch: 1        Training Loss: 0.586210
Epoch: 2        Training Loss: 0.369061
Epoch: 3        Training Loss: 0.319110
Epoch: 4        Training Loss: 0.299360
Epoch: 5        Training Loss: 0.289013
Epoch: 6        Training Loss: 0.281864
Epoch: 7        Training Loss: 0.277198
Epoch: 8        Training Loss: 0.273749
Epoch: 9        Training Loss: 0.270959
Epoch: 10       Training Loss: 0.268498
Epoch: 11       Training Loss: 0.266242
Epoch: 12       Training Loss: 0.264002
Epoch: 13       Training Loss: 0.262040
Epoch: 14       Training Loss: 0.260332
Epoch: 15       Training Loss: 0.258755
Epoch: 16       Training Loss: 0.257336
Epoch: 17       Training Loss: 0.255983
Epoch: 18       Training Loss: 0.254691
Epoch: 19       Training Loss: 0.253517
Epoch: 20       Training Loss: 0.252485
Epoch: 21       Training Loss: 0.251596
Epoch: 22       Training Loss: 0.250821
Epoch: 23       Training Loss: 0.250129
Epoch: 24       Training Loss: 0.249533
Epoch: 25       Training Loss: 0.249006
Epoch: 26       Training Loss: 0.248537
Epoch: 27       Training Loss: 0.248122
Epoch: 28       Training Loss: 0.247746
Epoch: 29       Training Loss: 0.247420
Epoch: 30       Training Loss: 0.247091
```

Figure 15: Error reduced with increasing epochs

Finally we check the output of the image with respect to the input Figure 16. You can observe that the output row on top is a cleaner version of the input. For instane the number 9 does not have a nub in it's upper part as seen in the input 9.

References

[1] *Probabilistic Machine Learning An Introduction*. The MIT Press, 2022.

[2] Changyou Chen. Autoencoders. https://github.com/udacity/deep-learning-v2-pytorch/blob/master/autoencoder/convolutional-autoencoder/Convolutional_Autoencoder_Solution.ipynb.

[3] Edureka. Autoencoders tutorial : A beginner's guide to autoencoders. https://www.edureka.co/blog/autoencoders-tutorial/#need.

[4] Jeremy Jordan. Introduction to autoencoders. https://www.jeremyjordan.me/autoencoders/,.

[5] Normalized Nerd. Autoencoders made easy! (with convolutional autoencoder). https://youtu.be/m2AyljDHYes.

[6] Tensorflow. Autoencoder tutorial. https://www.tensorflow.org/tutorials/generative/autoencoder.

[7] Wikipedia. Autoencoder. https://en.wikipedia.org/wiki/Autoencoder.

[8] Wikipedia. Autoencoder. https://en.wikipedia.org/wiki/MNIST_database.