Startup Sucess with Optimizing Machine Learning Algorithms

Abhay Thacker, Alison Wong, Aurora Travers, Josh Hernandez

4th June 2023

Abstract

This study aims to identify the most accurate algorithm for predicting the success of startups. Four commonly used algorithms, namely logistic regression, decision tree, support vector machine (SVM), and K-nearest neighbors (KNN), were evaluated and compared in terms of their predictive performance. The evaluation results revealed that the logistic regression algorithm exhibited the highest accuracy among the tested models with the second-highest computational time. The SVM algorithm demonstrated the fastest computational time but with the lowest accuracy of 69.27%. These findings contribute to the understanding of the strengths and limitations of various algorithms in predicting startup success. Future research could focus on optimizing the computational efficiency of the decision tree algorithm, as it has the slowest computational time of 23.49 seconds, or exploring hybrid models that combine the strengths of different algorithms.

Keywords: startup, logistic regression, decision tree, SVM, KNN

1 Introduction

Startups are crucial to the expansion of our economy as they bring in innovation, new ideas, and jobs, and tackle current business or social challenges. The number of new startups has increased exponentially over the past few years. However, only “40 percent of startups are able to become profitable”, due to many various factors such as their business model, logistics, and market demand [7]. The small percentage that becomes successful does go on to achieve profit and prominence, some even go on to become unicorns - privately held and valued at over 1 billion dollars [7]. ByteDance, the parent company of TikTok, is currently the highest-valued unicorn company in the world, valued at over 350 billion dollars [7].

Investing in startups is highly lucrative and rewarding if proven successful. It is also a very risky investment due to a high percentage of failures. Therefore, we are interested in predicting a startup’s success to allow investors to identify businesses with the potential for quick growth and success using different machine-learning algorithms created from scratch: Logistic regression, Random forest algorithm, Decision tree, K-Nearest Neighbors (KNN), and Support Vector Machines (SVM). We will then analyze its optimization algorithm for each method and compare their computational speed, in search of the algorithm with the fastest computational time and accuracy of predicting a startup’s success.

This topic is meaningful to a wide variety of groups including university students seeking employment opportunities and entrepreneurs working on a new startup.

For graduating university students, understanding the factors that contribute to the success of a startup can help them make informed decisions about their future careers. With the startup economy constantly evolving, it can be challenging for students to gauge which companies are more likely to provide them with long-term career opportunities. By examining the key drivers of start success, students can gain insights into which startups have a higher probability of succeeding. As most of us are graduating soon, this topic is particularly important for us.

Entrepreneurs looking to start their own ventures can also benefit from this topic. Starting a new business is a complex and risky endeavor, and entrepreneurs need to be able to assess and mitigate risks to increase their chances of success. By analyzing data on startup success and failure, entrepreneurs can gain a deeper understanding of the factors that drive success, as well as the pitfalls to avoid. This knowledge can help them make more informed decisions about their business strategy and funding.

1.1 Dataset Background

To investigate the questions we have posed, we plan to use a dataset provided by a machine learning engineer employee at the investing company GMO [6]. This dataset is credible and has been used for data sprint competitions in the past [6]. This dataset includes 923 startup companies, and 49 different features describing different things about the company.

Some features that we predict to be the most important are:

- **Relationships**: How many relationships the company has with investors
- **Funding Rounds**: How many rounds of funding they received
- **Funding Total (USD)**: How much funding did they receive
- **Industry Type**: What industry were they in
- **VC Funding**: If they received VC funding
- **Angel Funding**: If they received angel funding
- **Average Participants**: How many startup users

Below is a map showing the distribution of the startup companies, along with their status (closed or acquired):

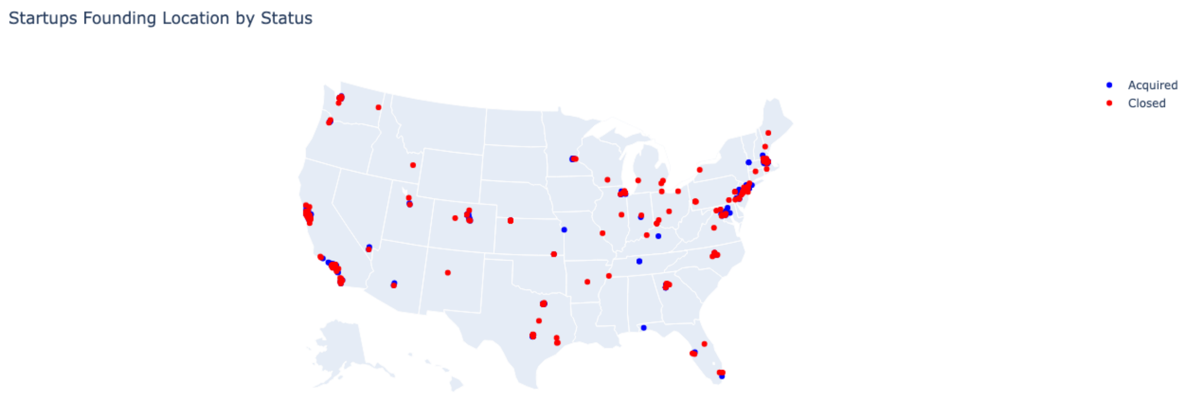


Figure 1: Distribution of Startups

The dataset is available as a downloadable file, and there was no need to use an API to gather data.

2 Data Preprocessing

The dataset was first preprocessed to fit our machine-learning algorithms. Firstly, unnecessary and repeated columns, such as `latitude`, `longitude`, and `zip_code`, were dropped to reduce noise and improve performance.

We then replaced NA values in the dataset with the averages for each feature to preserve the data distribution.

Lastly, we performed standardization on our features to have zero mean and unit variance of one. This was done to bring all features to a similar scale and have an understanding of their relative importance.

After data preprocessing, we are left with **35 features** and **923 data points**.

- **Features**: `founded_at`, `first_funding_at`, `last_funding_at`, `age_first_funding_year`, `age_last_funding_year`, `age_first_milestone_year`, `age_last_milestone_year`, `relationships`, `funding_rounds`, `funding_total_usd`, `milestones`, `is_CA`, `is_NY`, `is_MA`, `is_TX`, `is_otherstate`, `is_software`, `is_web`, `is_mobile`, `is_enterprise`, `is_advertising`, `is_gamesvideo`, `is_ecommerce`, `is_biotech`, `is_consulting`, `is_othercategory`, `has_VC`, `has_angel`, `has_roundA`, `has_roundB`, `has_roundC`, `has_roundD`, `avg_participants`, `is_top500`

- **Target Variable**: `status` - Binary classification of if a startup is closed/acquired.

2.1 Feature Importance

To gain a better understanding of which features contribute more significantly to our target variable predictions, we found the correlation between our features and the target variable using a correlation heatmap.

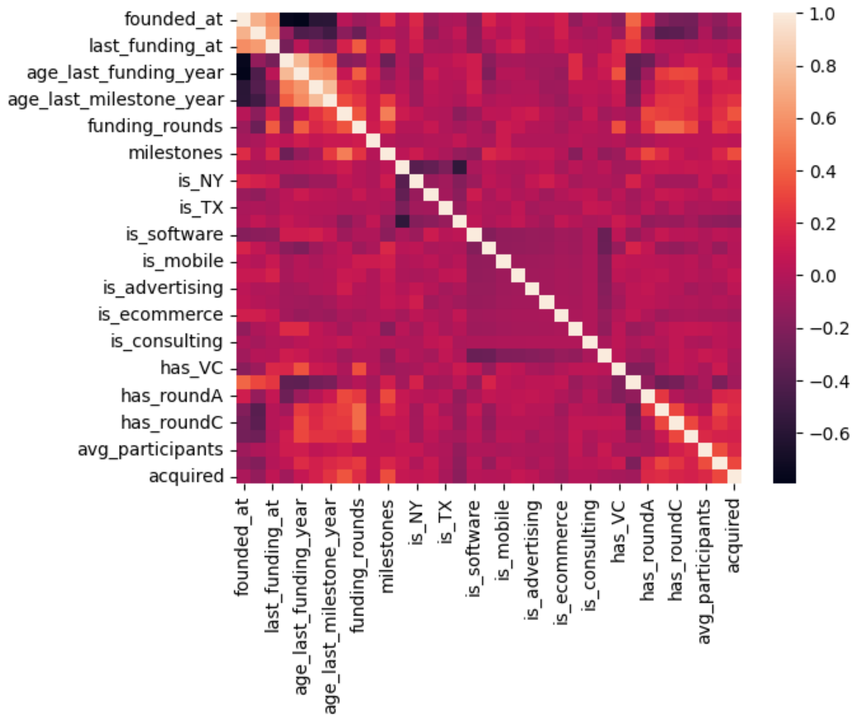


Figure 2: Distribution of Startups

We considered the following features and their importance:

- **Relationships (0.360)**: This feature indicates the number of outside relationships or help the startup has, including accountants, investors, vendors, and mentors. A higher value suggests more support from external sources.

- **Milestones (0.328)**: This feature represents the number of key milestones the startup has achieved, such as first funding, second funding, user launch, etc. A higher value indicates that the startup has successfully hit more significant milestones.

Relationships and Milestones exhibited a relatively higher correlation with the target variable compared to other features in the dataset. Therefore, they were considered important for our analysis and modeling.

Furthermore, the decision tree algorithm provides insights into which features are most influential in making predictions, through their feature importance value [3]. We will use our decision tree algorithm in Section 4.2 to evaluate its feature importance, in the hope that it will support our conclusion above.

3 Proposed Method

3.1 Logistic Regression

Logistic regression is a supervised learning algorithm for binary classification problems to find the relationship between the combination of input features and the probability of it belonging to a certain class in the target variable.

Our logistic regression class is separated into two methods:

Fit method: This method first gets the number of samples and features from the input data. In our case, we used our training dataset with our features. Then, the method initializes the weights and biases to zero. Gradient descent is implemented to update the weights and bias iteratively with an optimal learning rate until convergence is reached.

The linear regression predictions are computed by multiplying features by their corresponding weights, plus the bias term. The sigmoid function is then applied to obtain the predicted probability distribution from 0 to 1.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

where

$$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

- z is the linear combination of features and weights.

- $w_1, w_2, ..., w_n$ are the weights associated with the input features.

- Input features $x_1, x_2, ..., x_n$, respectively.

- b: bias term.

To improve the model's classification accuracy, the gradient descent iterative algorithm is used to calculate the minimization of the binary cross-entropy in terms of its weight and bias [2]. When the gradient is achieved after a specified number of iterations that maximizes accuracy, the new weight and bias will be obtained for prediction.

The learning rate is a hyperparameter and determines the step size at each iteration when updating the model's parameters (weight and bias) [10]. An appropriate learning rate has to be chosen that optimizes the convergence speed and the final accuracy rate of the model. The optimal learning rate is problem-dependent and varies by dataset. One way hyperparameter tuning for finding the optimal learning rate and the number of epochs is by plotting an epoch vs. accuracy graph.

Predict method: This method is used to make predictions on new data. It takes input data and computes predictions using our new weight and bias for each input feature. It then uses a decision boundary, where in our case, it predicts the class label by comparing the predicted probability to the threshold: 0.5. If the predicted probability is less than or equal to 0.5, the class label is set to 0; otherwise, it is set to 1.

3.2 Decision Tree

A decision tree is a supervised learning algorithm that can be used for both classification and regression problems. It represents a flowchart-like structure where internal nodes denote feature tests, branches represent the outcomes of those tests, and leaf nodes represent the final decision or the predicted outcome [12].

Attribute Selection Measures:

- Entropy: Entropy quantifies the level of randomness or uncertainty present in a dataset. When it comes to classification tasks, entropy specifically measures the randomness by considering the distribution of class labels within the dataset. Entropy is given by the formula below.

  $$-\sum(p_i * log2(p_i))$$

  p_i represents the probability of an instance belonging to a particular class within the subset. When deciding on the splitting attribute within a decision tree, the attribute that minimizes the entropy after the split is chosen.

- Gini Impurity: Gini impurity measures the impurity or disorder of a subset of data based on the distribution of class labels. The Gini impurity for a given subset is calculated using the formula:

  $$1-\sum(p_i)^2$$

  p_i represents the probability of an instance belonging to a particular class within the subset. Like entropy, the attribute that minimizes the Gini impurity after the split is selected as the splitting attribute within a decision tree.

- Information Gain: information gain is used to evaluate the usefulness of a feature in the process of classifying or splitting the data. It is calculated using the following formula:

  $$Entropy(parent) - [weightedaverage] * Entropy(children)$$

3.3 KNN

K Nearest Neighbors (KNN) is a supervised learning algorithm that can be used for both classification and regression problems. It works by finding the K nearest data points in the training set to a given test point and making predictions based on the labels (for classification) or values (for regression) of those nearest neighbors.

It works through the following [11]:

1. Choose the number of neighbors (K) to consider.
2. Compute the distance between the test point and each point in the training set. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance.
3. Select the K nearest neighbors based on the computed distances.
4. For classification, determine the class label by majority voting among the K neighbors. For regression, calculate the average or weighted average of the target values of the K neighbors.
5. Assign the predicted label or value to the test point.

The following distances and their respective equations can be used in the K Nearest Neighbors algorithm [5]:

- Euclidean Distance

  Euclidean distance is the most commonly used distance metric in KNN. It calculates the straight-line distance between two points in Euclidean space.

  $$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

  where x_i and y_i are the i-th features of the two points.

- Manhattan Distance

  Manhattan distance calculates the distance between two points by summing the absolute differences of their coordinates.

  $$\sum_{i=1}^{n}|x_i - y_i|$$

  where x_i and y_i are the i-th features of the two points.

- Minkowski Distance

  Minkowski distance is a generalization of Euclidean and Manhattan distances. It is defined as:

  $$\left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{\frac{1}{p}}$$

  where p is a parameter. When $p = 1$, it reduces to Manhattan distance, and when $p = 2$, it becomes Euclidean distance.

3.4 SVM

Support Vector Machines (SVMs) are a type of machine learning algorithm commonly used for binary classification problems. The goal of an SVM is to find the optimal hyperplane that separates two classes by maximizing the margin between the closest data points of each class. These closest data points are called "support vectors".

SVMs can be learned in two primary ways [9]:

1. **Closed-Form Quadratic Optimization**:
   In this approach, an SVM is learned by directly solving for the weight vector w such that
   $$\min_w \frac{1}{2}||w||^2$$
   subject to the constraint
   $$y_i(w^T x_i) \geq 1 \ \ \forall i$$

2. **Iterative Optimization using Gradient Descent**:
   In this approach, an SVM is learned by solving the following optimization problem:
   $$\min_w \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i)) + \lambda||w||^2$$
   The above equation is known as **Hinge Loss** which is a measure of classification error. The second term in the equation uses a hyperparameter λ to control the complexity of the model.

While both methods solve for the optimal hyperplane, the iterative approach involving gradient descent scales better with larger datasets.

4 Data Analysis Study

4.1 Logistic Regression [B.1]

Before beginning our training process, we will select our hyperparameters (learning rate and number of epochs) that optimizes our logistic regression model.

This is done by plotting an epoch vs. accuracy graph to provide insights into the different learning rates and how the accuracy of the model changes as the number of epochs increases.

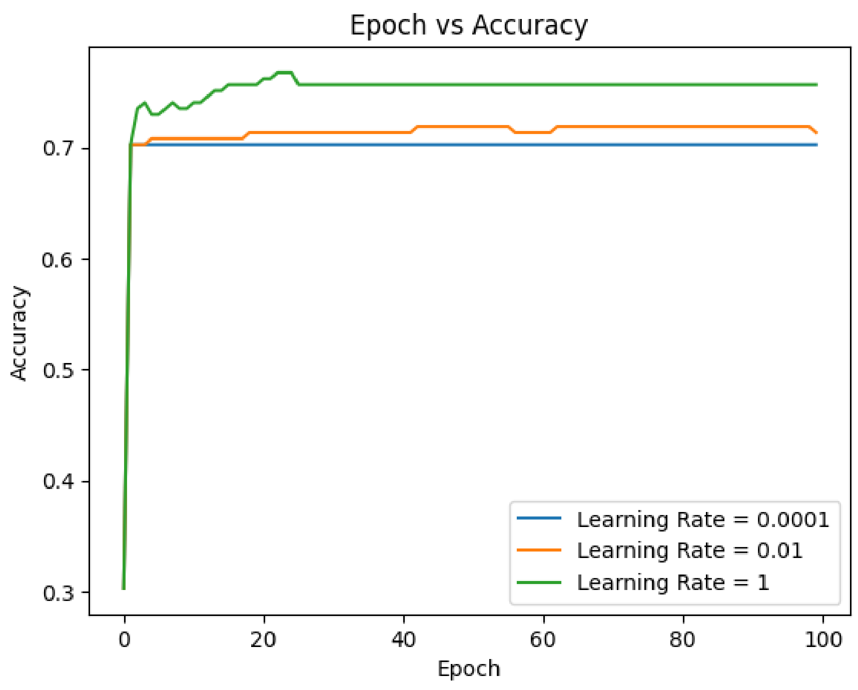


Figure 3: Epoch vs. Accuracy with different learning rates

The pattern observed for every learning rate is that as the number of epochs increases in the beginning, the accuracy rapidly increases, then starts to flatten out. When that happens, it means that the model has reached a point of convergence or near-convergence where iterations do not lead to substantial gains in performance.

We can observe from the plot that `learning rate = 0.0001` reaches a plateau in accuracy the quickest at `epochs = 1`. However, it has the lowest accuracy rate of 70.270%. This low accuracy can be attributed to the fact that a smaller learning rate takes smaller steps during each iteration, slowing down the convergence process. As a result, the maximum number of epochs provided may not be sufficient for the model to fully converge, leading to suboptimal performance. To improve the accuracy, we have to increase the number of epochs or use a higher learning rate to allow the model to converge more effectively.

`Learning rate = 0.01` is slightly unstable and has not reached its flattening-out phase with an epoch of 100. The model reaches its highest accuracy of 71.892% at `epochs = 40`.

`Learning rate = 1` reaches a plateau, signifying convergence or near-convergence to the optimal solution at `epochs = 25`, with the highest accuracy of 75.676%.

To find the optimal balance between convergence speed and accuracy, the **learning rate** $=$ **1** at **epochs** $=$ **25** for our model seems to be the ideal it has the highest accuracy and the flattening out of the accuracy means that the model has reached a stable convergence into an optimal solution. We will use those hyperparameters to predict our testing dataset to find the computational time and accuracy.

To have a comparison with our model, we trained our training data using sklearn's Logistic Regression function and predicted the testing data to compute its computational time and accuracy.

To ensure a fair comparison between our scratch-made logistic regression gradient descent algorithm and Scikit-learn's `LogisticRegression`, we set the solver parameter to 'sag' in Sklearn [8]. This choice allows us to use the Stochastic Average Gradient descent method for optimization, which aligns closely with our iterative gradient descent approach.

The following results were observed for Sklearns's `LogisticRegression`, compared with our model's result:

	Accuracy	Computation Time
Our Implementation	75.68%	0.0339 sec
Sklearn Implementation	75.68%	0.0592 sec

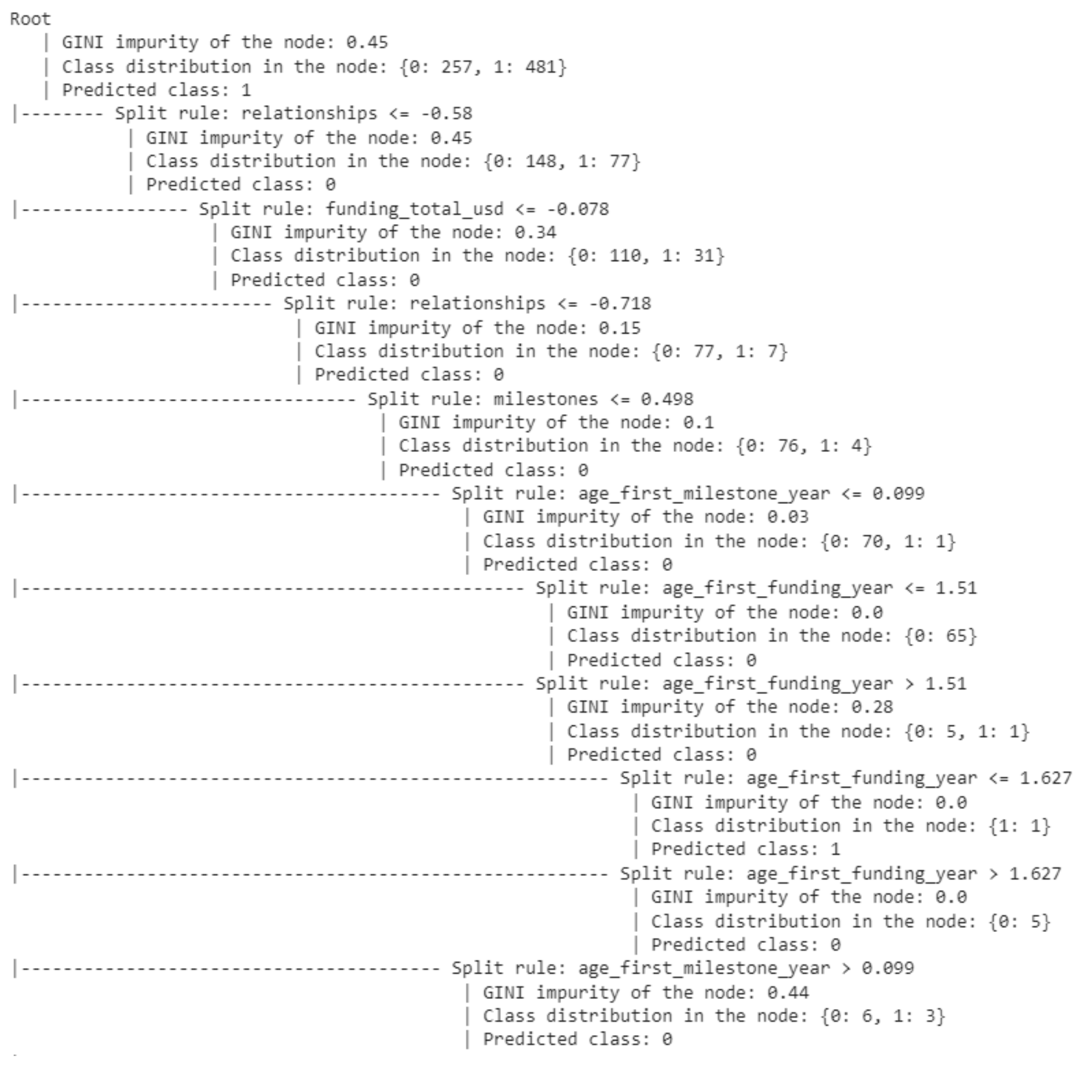
It is interesting to note that our accuracy at learning rate = 1 and epochs = 25 matches the accuracy of Sklearn's algorithm. Our implementation is faster which can be explained by having more specialized hyperparameters for this specific data set problem. Sklearn's functions need to be tuned for a larger general purpose thus it has more complex functions.

Overall, we conclude that our Logistic Regression implementation is a success as it matches Sklearn's accuracy and has a faster computational time.

4.2 Decision Tree [B.2]

To assess the accuracy and computational time of our scratch-made decision tree implementation, we conducted a comparison with Sklearn's `Decision Tree`. It is important to note that Sklearn's implementation is highly optimized. Thus, it is expected that our implementation would have a longer computational time in comparison.

We used Eligijus Bujokas's implementation of the decision tree algorithm as a starting point [4]. The code snippet below illustrates the word visualization of the first few splitting of the tree.

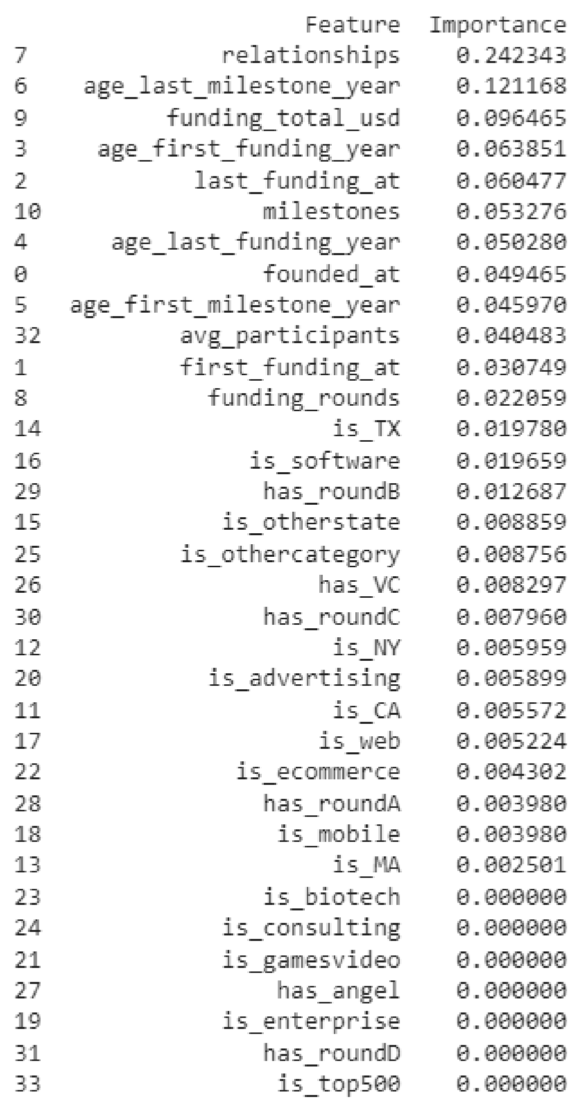


```
Root
   | GINI impurity of the node: 0.45
   | Class distribution in the node: {0: 257, 1: 481}
   | Predicted class: 1
|-------- Split rule: relationships <= -0.58
           | GINI impurity of the node: 0.45
           | Class distribution in the node: {0: 148, 1: 77}
           | Predicted class: 0
|---------------- Split rule: funding_total_usd <= -0.078
                   | GINI impurity of the node: 0.34
                   | Class distribution in the node: {0: 110, 1: 31}
                   | Predicted class: 0
|------------------------ Split rule: relationships <= -0.718
                           | GINI impurity of the node: 0.15
                           | Class distribution in the node: {0: 77, 1: 7}
                           | Predicted class: 0
|-------------------------------- Split rule: milestones <= 0.498
                                   | GINI impurity of the node: 0.1
                                   | Class distribution in the node: {0: 76, 1: 4}
                                   | Predicted class: 0
|---------------------------------------- Split rule: age_first_milestone_year <= 0.099
                                           | GINI impurity of the node: 0.03
                                           | Class distribution in the node: {0: 70, 1: 1}
                                           | Predicted class: 0
|------------------------------------------------ Split rule: age_first_funding_year <= 1.51
                                                   | GINI impurity of the node: 0.0
                                                   | Class distribution in the node: {0: 65}
                                                   | Predicted class: 0
|------------------------------------------------ Split rule: age_first_funding_year > 1.51
                                                   | GINI impurity of the node: 0.28
                                                   | Class distribution in the node: {0: 5, 1: 1}
                                                   | Predicted class: 0
|-------------------------------------------------------- Split rule: age_first_funding_year <= 1.627
                                                           | GINI impurity of the node: 0.0
                                                           | Class distribution in the node: {1: 1}
                                                           | Predicted class: 1
|-------------------------------------------------------- Split rule: age_first_funding_year > 1.627
                                                           | GINI impurity of the node: 0.0
                                                           | Class distribution in the node: {0: 5}
                                                           | Predicted class: 0
|---------------------------------------- Split rule: age_first_milestone_year > 0.099
                                           | GINI impurity of the node: 0.44
                                           | Class distribution in the node: {0: 6, 1: 3}
                                           | Predicted class: 0
```

Figure 4: Decision Tree Visualization

Our accuracy came out as 72.43%, which is identical to Sklearn's package. However, our computational time with a much slower computational time of 23.494 seconds, while Sklearn's computational time was 0.0092 seconds. To further optimize our computational time, we tried utilizing parallel computation as an optimization method, but it made our algorithm even slower. This outcome may be attributed to the inherent imbalanced nature of the dataset, where a majority of start-ups' statuses are closed. Furthermore, considering the relatively modest size of our dataset, the introduction of parallel processes appeared to introduce unnecessary harm.

We also calculated the feature importance listed below:



```
                        Feature  Importance
7                 relationships    0.242343
6      age_last_milestone_year    0.121168
9             funding_total_usd    0.096465
3       age_first_funding_year    0.063851
2              last_funding_at    0.060477
10                   milestones    0.053276
4        age_last_funding_year    0.050280
0                   founded_at    0.049465
5     age_first_milestone_year    0.045970
32             avg_participants    0.040483
1             first_funding_at    0.030749
8               funding_rounds    0.022059
14                        is_TX    0.019780
16                  is_software    0.019659
29                   has_roundB    0.012687
15                is_otherstate    0.008859
25             is_othercategory    0.008756
26                       has_VC    0.008297
30                   has_roundC    0.007960
12                        is_NY    0.005959
20               is_advertising    0.005899
11                        is_CA    0.005572
17                       is_web    0.005224
22                 is_ecommerce    0.004302
28                   has_roundA    0.003980
18                    is_mobile    0.003980
13                        is_MA    0.002501
23                   is_biotech    0.000000
24                is_consulting    0.000000
21                is_gamesvideo    0.000000
27                    has_angel    0.000000
19                is_enterprise    0.000000
31                   has_roundD    0.000000
33                    is_top500    0.000000
```

Figure 5: Feature Importance

The results support the correlation heat map we created in Section 2.1 where **relationship** was identified as the best predictive feature for our decision tree model. We can also see some feature has an importance of zero, to save computational time of our model, we can exclude those features.

	Accuracy	Computation Time
Our Implementation	72.43%	23.4940 sec
Sklearn Implementation	72.43%	0.0092 sec

4.3 KNN [B.3]

An important part of using a KNN is deciding the value of K. A good estimate of K is the square root of the number of data points, divided by two. Since we have 900 data points, this would equal K = 15.

We test the model for accuracy at different values. The respective results are here.

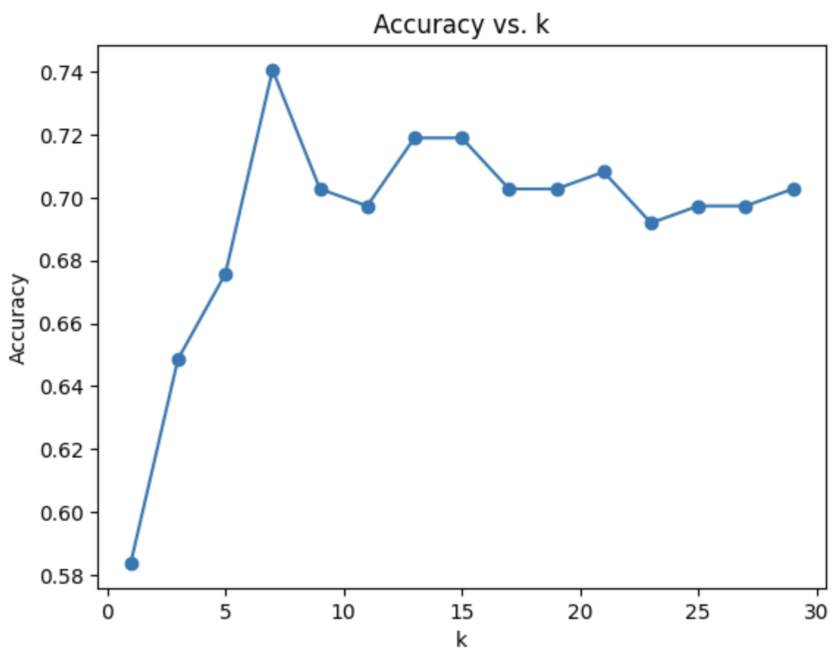


Figure 6: KNN Accuracy vs K

Comparing our implementation with Sklearn's `KNeighborsClassifier`, we can see that the accuracy is highest for K = 7. We now run the prediction with this K value.

	Accuracy	Computation Time
Our Implementation	74.05%	0.07693433761596668 sec
Sklearn Implementation	74.05%	0.012268781661987305 sec

As we can see, we have correctly implemented our KNN algorithm, as the computed accuracies are the same.

It is also important to note that using KNN becomes very slow as the number of data points increases. If we were to have a bigger dataset, computational time would be a lot larger than most other methods. KNN is a great classifier for a modest-sized dataset like ours, but for big data, memory efficiency and therefore computation may become an issue.

4.4 SVM [B.4]

To begin fitting an SVM classification model to the problem, we first performed hyperparameter tuning to identify the optimal values for the regularization parameter λ and the learning rate.

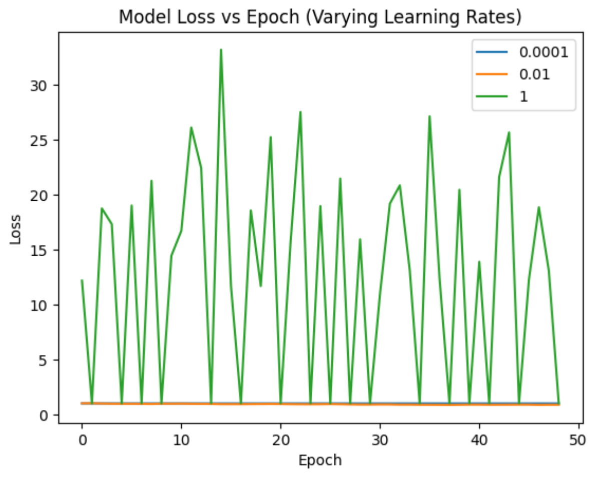


Figure 7: SVM Hyperparameter Tuning: Learning Rate

From the above plot, we can see that of the three learning rates tested (0.0001, 0.01, and 1), using the learning rate of 1 produces very unstable results. The model fails to converge to any viable solution as the loss increases and decreases sporadically within increased training time. This is expected as higher learning rates often overshoot local minima.

The other two learning rates seem to converge at comparable rates, so we opted to use the larger of the two learning rates, 0.01, to train our SVM model.

We then tuned the regularization parameter λ.

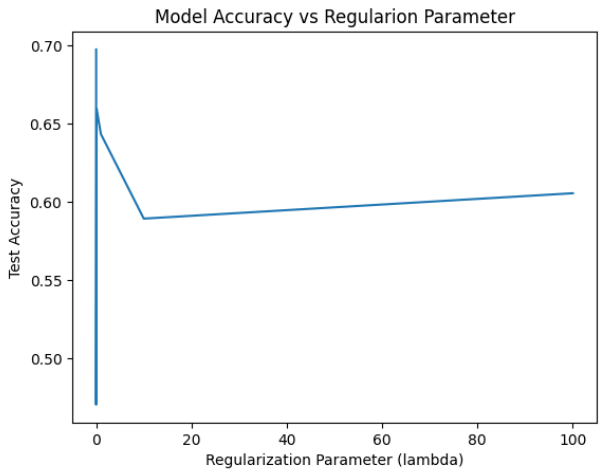


Figure 8: SVM Hyperparameter Tuning: Regularization Parameter

From the plot above, we can see that the choice of regularization parameter greatly affects the model's performance as the model's accuracy varies from 60-70% depending on the value of λ. From our analysis we found that a value of $\lambda = 0.01$ produced the best accuracy, so that is what we decided to use when training our model.

After tuning our parameters, we fit our model to our testing set and evaluated its performance against a separate testing set. We also compared the results to Sklearn's `SGDClassifier` which is also a gradient descent-based SVM implementation. The following results were observed:

	Accuracy	Computation Time
Our Implementation	69.27%	0.022 sec
Sklearn Implementation	70.27%	0.015 sec

4.5 All Model Comparison

The following is a summary of all our experiment results:

	Accuracy	Computation Time
Logistic Regression	75.68%	0.034 sec
Decision Tree	72.43%	23.49 sec
KNN	74.05%	0.077 sec
SVM	69.27%	0.022 sec

5 Conclusion

The objective of this research study was to identify the most accurate algorithm for predicting the success of startups. To achieve this objective, four commonly used algorithms, namely logistic regression, decision tree, support vector machine (SVM), and K-nearest neighbors (KNN), were evaluated and compared in terms of their predictive performance and computational efficiency. The study results demonstrated that of the models tested, the logistic regression classifier exhibited the highest accuracy on the chosen test dataset. It also had a fairly low computation speed of 0.034 seconds, making it the best model for this problem across both metrics of interest. However, the KNN algorithm was shown to be another viable model as it achieved the second-highest accuracy while maintaining a reasonable computation time of 0.077 seconds. Of all the models tested, SVM had the lowest performance with an accuracy of 69.27%. Similarly, the decision tree model has the slowest computational time of 23.49 seconds. Consequently, the current implementations of these two methods are not viable options for this problem.

Based on our research we also believe that if we were to use a larger dataset, logistic regression would still be the best model when considering computational time due to its simplicity. We also expect KNN to not be viable in larger datasets due to its memory requirements as the entire dataset is used during the prediction process.

Future research in this area could focus on optimizing the computational efficiency of the decision tree algorithm by exploring techniques such as pruning or ensemble methods. Similarly, to improve the accuracy of SVM, kernel-based SVM methods could be explored to make it perform better on this dataset. Additionally, exploring hybrid models that leverage the strengths of the various algorithms tested could also provide more promising results.

Overall the results of this study provide valuable insights for investors, university students, and entrepreneurs seeking to make informed decisions about startup investments, employment opportunities, and business strategies, respectively.

References

[1] Qandeel Abbassi. Svm from scratch – python. `https://towardsdatascience.com/svm-implementation-from-scratch-python-2db2fc52e5c2`.

[2] AssemblyAI. Assemblyai - how to implement logistic regression from scratch with python. `https://www.youtube.com/watch?v=YYEJ_GUguHw&ab_channel=AssemblyAI`, 2022.

[3] Jason Brownlee. How to calculate feature importance with python. https://machinelearningmastery.com/calculate-feature-importance-with-python/#:~:text=based%20importance%20scores.-,Decision%20Tree%20Feature%20Importance,points%2C%20like%20Gini%20or%20entropy., 2020.

[4] Eligijus Bujokas. Decision tree algorithm in python from scratch. *Towards Data Science*, 2021.

[5] Sarang Anil Gokte. Most popular distance metrics used in k-nearest neighbors (knn). https://www.kdnuggets.com/2020/11/most-popular-distance-metrics-knn.html, 2023.

[6] Manish Kumar. Startup success prediction. https://www.kaggle.com/datasets/manishkc06/startup-success-prediction, 2021.

[7] Artem Minaev. Startup statistics. https://firstsiteguide.com/startup-stats/, 2023.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Berwick R. An idiot's guide to support vector machines (svms). https://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf.

[10] Aditya Rakhecha. Understanding learning rate. *Towards Data Science*, 2019.

[11] Doug Steen. How to build k-nearest neighbors (knn) from scratch in python. *Towards Data Science*, 2020.

[12] Philip Wilkinson. Introduction to decision tree classifiers from scikit-learn. *Towards Data Science*, 2020.

Appendix

A Data Preprocessing code

```
# Read Data
import pandas as pd
import io
df = pd.read_csv(io.BytesIO(uploaded['startup data.csv']))

# Data Visualization
import plotly.express as px
import plotly.graph_objects as go

# Dataframe for plotting
location = df[['name', 'latitude', 'longitude', 'status']]

# Visualize startups founding locations by status
# Match the status values to colors
status_colors = {'closed': 'red', 'acquired': 'blue'}

# Create figure
fig = go.Figure()
```

```
# Iterate over unique statuses
for status in location['status'].unique():
    # Filter locations by status
    filtered_location = location[location['status'] == status]

    # Scatter points for startups of the current status
    loc = go.Scattergeo(
        lat=filtered_location['latitude'],
        lon=filtered_location['longitude'],
        mode='markers',
        marker=dict(
            color=status_colors[status], # color of points by status
            size=6,
            symbol='circle'
        ),
        text=filtered_location['name'], # show name of startup in label
        name=status.capitalize()
    )

    fig.add_trace(loc)

# Add title
fig.update_layout(
    title_text="Startups Founding Location by Status",
    geo_scope="usa"
)
fig.show()

# Drop unnecessary columns
df.drop(['Unnamed: 0', 'latitude', 'longitude', 'zip_code', 'id', 'Unnamed: 6', 'labels', 'state_code.1
df.drop(['city', 'closed_at'],axis=1,inplace=True)

# Data Maniputation for fitting machine learning algorithms
import numpy as np

# Feature variables
df_X = df[df.columns[:-1]]

# Target variable
df_y = df[df.columns[-1]]

# Change binary target to 1 and 0
df_y = np.where((df_y == 'acquired'), 1, 0)

# Replace missing values with the average for each column
df_X = df_X.fillna(df_X.mean())

# Replace all date columns with their year
year_col = ["founded_at", "first_funding_at", "last_funding_at"]
for column in year_col:
  df_X[column] = pd.DatetimeIndex(df_X[column]).year

# Standardization
```

```
from sklearn.preprocessing import StandardScaler
column_names = df_X.columns
X_scaled = StandardScaler().fit_transform(df_X.loc[:])
X_scaled = pd.DataFrame(X_scaled, columns = column_names)

# Split dataset
from sklearn.model_selection import train_test_split
random.seed(2023)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, df_y, test_size = 0.2)

# Check if scaled data has zero mean and unit variance
print(X_scaled.mean(axis=0))
print(X_scaled.std(axis=0))

# Correlation heat map
import seaborn as sns
full_data = X_scaled
full_data["acquired"] = df_y
sns.heatmap(full_data.corr())
```

B Machine Learning Algorithm code

B.1 Logistic Regression

```
# Application of Class Material:
# The logistic regression iteratively uses the gradient descent algorithm that was
# discussed in class. The learning rate for the model was also tuned to overcome the problems with conv

# Sigmoid function
def sigmoid(x):
  return 1 / (1 + np.exp(-x))

# Create Logistic Regression class
class LogisticRegression():

    def __init__(self, learning_rate=0.0001, epochs=10):
      self.learning_rate = learning_rate
      self.epochs = epochs
      self.weights = None
      self.bias = None

    def fit(self, X, y):

      # Get sample size and features from input data
      no_samples, no_features = X.shape

      # Initialize weights and bias to zero
      self.weights = np.zeros(no_features)
      self.bias = 0

      # Use gradient descent for the specified number of iterations
      # Weights & bias are adjusted iteratively to minimize the difference
      # between the predicted probabilities and the true labels
      for _ in range(self.epochs):

        # Compute the linear predictions
        # linear combination of features multiplied by their corresponding weights,
        # plus the bias term
        linear_pred = np.dot(X, self.weights) + self.bias

        # Apply the sigmoid function to get the predicted probabilities
        predictions = sigmoid(linear_pred)

        # Compute gradients of the weights and bias
        dw = (1 / no_samples) * np.dot(X.T, (predictions - y))
        db = (1 / no_samples) * np.sum(predictions - y)

        # Update the weights and bias using the learning rate to get the gradient
        self.weights -= self.learning_rate * dw
        self.bias -= self.learning_rate * db

    def predict(self, X):

      # Compute predictions
      linear_pred = np.dot(X, self.weights) + self.bias
      y_pred = sigmoid(linear_pred)

      # Convert the predicted probabilities to class labels
      class_pred = np.where(y_pred <= 0.5, 0, 1)

      return class_pred

# Find accuracy function
def accuracy(y_pred, y_test):
  return np.sum(y_pred == y_test)/len(y_test)

# Training model
import time

# Start the timer
start_time = time.time()

# Begin training and fitting model
model = LogisticRegression(learning_rate = 1, epochs = 25)
model.fit(X_train.to_numpy(), y_train)
test_pred = model.predict(X_test.to_numpy())

end_time = time.time()

# Calculate the elapsed time
execution_time = end_time - start_time
print(f"Computational time: {execution_time} seconds")

# Compute accuracy of my implementation
print("accuracy on test dataset: {}".format(accuracy(y_test, test_pred)))
```

```
# Plot accuracy vs. epoch
import matplotlib.pyplot as plt

# Define the accuracy function
def accuracy(y_pred, y_test):
  return np.sum(y_pred == y_test)/len(y_test)

# Lists to store the epoch and accuracy values
epochs = []
accuracies_lr_0001 = []
accuracies_lr_1 = []
accuracies_lr_01 = []

learning_rates = [0.0001, 0.01, 1]

# Train the model and calculate accuracy on the validation set after each epoch
for epoch in range(100):
    model_lr_0001 = LogisticRegression(learning_rate=0.0001, epochs=epoch)
    model_lr_1 = LogisticRegression(learning_rate=1, epochs=epoch)
    model_lr_01 = LogisticRegression(learning_rate=0.01, epochs=epoch)

    model_lr_0001.fit(X_train.to_numpy(), y_train)
    model_lr_1.fit(X_train.to_numpy(), y_train)
    model_lr_01.fit(X_train.to_numpy(), y_train)

    test_pred_lr_0001 = model_lr_0001.predict(X_test.to_numpy())
    test_pred_lr_1 = model_lr_1.predict(X_test.to_numpy())
    test_pred_lr_01 = model_lr_01.predict(X_test.to_numpy())

    accuracy_val_lr_0001 = accuracy(y_test, test_pred_lr_0001)
    accuracy_val_lr_1 = accuracy(y_test, test_pred_lr_1)
    accuracy_val_lr_01 = accuracy(y_test, test_pred_lr_01)

    epochs.append(epoch)
    accuracies_lr_0001.append(accuracy_val_lr_0001)
    accuracies_lr_1.append(accuracy_val_lr_1)
    accuracies_lr_01.append(accuracy_val_lr_01)

# Plot the epoch vs accuracy graph
plt.plot(epochs, accuracies_lr_0001, label='Learning Rate = 0.0001')
plt.plot(epochs, accuracies_lr_01, label='Learning Rate = 0.01')
plt.plot(epochs, accuracies_lr_1, label='Learning Rate = 1')

plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Epoch vs Accuracy')
plt.legend()
plt.show()

# Compare with sklearn's model
from sklearn.linear_model import LogisticRegression

# Start the timer
```

```
start_time = time.time()

model = LogisticRegression(solver = "sag")
model = model.fit(X_train.to_numpy(), y_train)
y_pred = model.predict(X_test)

end_time = time.time()

# Calculate the elapsed time
execution_time = end_time - start_time
print(f"Computational time: {execution_time} seconds")
print("Accuracy on test dataset: {}".format(accuracy(y_test, y_pred)))
```

B.2 Decision Tree

```
# Application of Class Material:
# We tried parallel computation as mentioned in class for our optimization method,
# but this slowed down our decision tree even more. This might be due to our data being
# imbalanced. Furthermore, considering the relatively modest size of our dataset
# using parallel processes is unnecessary.

# Import necessary libraries
import pandas as pd
import numpy as np
from collections import Counter
class Node:
    """
    Class for creating the nodes for a decision tree
    """
    # Initiate variables
    def __init__(
        self,
        Y: list,
        X: pd.DataFrame,
        min_samples_split=None,
        max_depth=None,
        depth=None,
        node_type=None,
        rule=None
    ):
        # Store the target values and feature values of the node
        self.Y = Y
        self.X = X

        # Store the provided or default values
        self.min_samples_split = min_samples_split if min_samples_split else 2
        self.max_depth = max_depth if max_depth else 100

        # Stores the current depth of the node
        self.depth = depth if depth else 0

        # Extracting all the features
        self.features = list(self.X.columns)

        # Stores the type of node
        self.node_type = node_type if node_type else 'root'

        # Rule for spliting
        self.rule = rule if rule else ""

        # Counts Y in the node
        self.counts = Counter(Y)

        # Calculate the GINI impurity based on the Y distribution
        self.gini_impurity = self.get_GINI()

        # Sorting the counts and saving the final prediction of the node
        counts_sorted = list(sorted(self.counts.items(), key=lambda item: item[1]))

        # Getting the last item
        yhat = None
        if len(counts_sorted) > 0:
            yhat = counts_sorted[-1][0]

        # Saving to object attribute. This node will predict the class with the most frequent class
        self.yhat = yhat

        # Saving the number of observations in the node
        self.n = len(Y)

        # Initiating the left and right nodes as empty nodes
        self.left = None
        self.right = None

        # Default values for splits
        self.best_feature = None
        self.best_value = None

    # Define a static method within a class
    @staticmethod
    def GINI_impurity(y1_count: int, y2_count: int) -> float:
        """
        Given the observations of a binary class calculate the GINI impurity
        """
        # Ensuring the correct types
        if y1_count is None:
            y1_count = 0

        if y2_count is None:
            y2_count = 0

        # Getting the total observations
        n = y1_count + y2_count

        # If n is 0 then we return the lowest possible gini impurity
        if n == 0:
            return 0.0

        # Getting the probability to see each of the classes
        p1 = y1_count / n
        p2 = y2_count / n

        # Calculating GINI
        gini = 1 - (p1 ** 2 + p2 ** 2)

        # Returning the gini impurity
        return gini

    # Define a static method within a class
    @staticmethod
    def ma(x: np.array, window: int) -> np.array:
        """
        Calculates the moving average of the given list.
        """
        # x: The input array on which the convolution is performed
        # np.ones(window): An array of ones with a length equal to the window parameter
        # valid: The mode of convolution to use
        return np.convolve(x, np.ones(window), 'valid') / window

    def get_GINI(self):
        """
        Function to calculate the GINI impurity of a node
        """
        # Getting the 0 and 1 counts
        y1_count, y2_count = self.counts.get(0, 0), self.counts.get(1, 0)

        # Getting the GINI impurity
        return self.GINI_impurity(y1_count, y2_count)

    def best_split(self) -> tuple:
        """
        Given the X features and Y targets calculates the best split
        for a decision tree
        """
        # Creating a dataset for spliting
        df = self.X.copy()
        df['Y'] = self.Y

        # Getting the GINI impurity for the base input
        GINI_base = self.get_GINI()

        # Finding which split yields the best GINI gain
        max_gain = 0

        # Default best feature and split
        best_feature = None
        best_value = None

        for feature in self.features:
            # Droping missing values
            Xdf = df.dropna().sort_values(feature)

            # Sorting the values and getting the rolling average
            xmeans = self.ma(Xdf[feature].unique(), 2)

            for value in xmeans:
                # Spliting the dataset
                left_counts = Counter(Xdf[Xdf[feature]<value]['Y'])
                right_counts = Counter(Xdf[Xdf[feature]>=value]['Y'])

                # Getting the Y distribution from the dicts
                y0_left, y1_left, y0_right, y1_right = left_counts.get(0, 0), left_counts.get(1, 0), ri

                # Getting the left and right gini impurities
                gini_left = self.GINI_impurity(y0_left, y1_left)
                gini_right = self.GINI_impurity(y0_right, y1_right)

                # Getting the obs count from the left and the right data splits
                n_left = y0_left + y1_left
                n_right = y0_right + y1_right

                # Calculating the weights for each of the nodes
                w_left = n_left / (n_left + n_right)
                w_right = n_right / (n_left + n_right)

                # Calculating the weighted GINI impurity
                wGINI = w_left * gini_left + w_right * gini_right

                # Calculating the GINI gain
                GINIgain = GINI_base - wGINI

                # Checking if this is the best split so far
                if GINIgain > max_gain:
                    best_feature = feature
                    best_value = value

                    # Setting the best gain to the current one
                    max_gain = GINIgain

        # Check if the best feature and best value are not None
        if best_feature is not None and best_value is not None:
            return (best_feature, best_value)
        else:
            return (None, None)

    def grow_tree(self):
        """
        Recursive method to create the decision tree
        """
        # Making a dataframe from the data
        df = self.X.copy()
        df['Y'] = self.Y

        # If there is GINI to be gained, we split further
```

```
if (self.depth < self.max_depth) and (self.n >= self.min_samples_split) and len(set(self.Y)) > 

    # Getting the best split
    best_feature, best_value = self.best_split()

    if best_feature is not None:
        # Saving the best split to the current node
        self.best_feature = best_feature
        self.best_value = best_value

        # Getting the left and right nodes
        left_df, right_df = df[df[best_feature]<=best_value].copy(), df[df[best_feature]>best_va

        # Creating the left and right nodes
        left = Node(
            # The target variable values of the left subset converted to a list
            left_df['Y'].values.tolist(),
            # The features of the left subset
            left_df[self.features],
            # The depth of the left child node
            depth=self.depth + 1,
            # The maximum depth is passed to the left child node
            max_depth=self.max_depth,
            # The minimum number of samples required to perform a split is passed to the left c
            min_samples_split=self.min_samples_split,
            # The node type of the left child node
            node_type='left_node',
             # The split rule
            rule=f"{best_feature} <= {round(best_value, 3)}"
            )

        # Assigning the left child node
        self.left = left
        # Growing the tree from the left child node
        self.left.grow_tree()

        right = Node(
            # The target variable values of the right subset converted to a list
            right_df['Y'].values.tolist(),
            # The features of the right subset
            right_df[self.features],
            # The depth of the right child node
            depth=self.depth + 1,
            # The maximum depth is passed to the right child node
            max_depth=self.max_depth,
            # The minimum number of samples required to perform a split is passed to the right 
            min_samples_split=self.min_samples_split,
            # The node type of the right child node
            node_type='right_node',
            # The split rule
            rule=f"{best_feature} > {round(best_value, 3)}"
            )

        # Assigning the right child node
```

```
                self.right = right
                # Growing the tree from the right child node
                self.right.grow_tree()

    def print_info(self, width=4):
        """
        Method to print the infromation about the tree
        """
        # Defining the number of spaces
        const = int(self.depth * width ** 1.5)
        spaces = "-" * const

        # print information
        if self.node_type == 'root':
            print("Root")
        else:
            print(f"|{spaces} Split rule: {self.rule}")
        print(f"{' ' * const}   | GINI impurity of the node: {round(self.gini_impurity, 2)}")
        print(f"{' ' * const}   | Class distribution in the node: {dict(self.counts)}")
        print(f"{' ' * const}   | Predicted class: {self.yhat}")

    def print_tree(self):
        """
        Prints the whole tree from the current node to the bottom
        """
        # Traverses the entire tree structure and prints the information
        # for each node in a depth-first manner
        self.print_info()

        if self.left is not None:
            self.left.print_tree()

        if self.right is not None:
            self.right.print_tree()

    def predict(self, X:pd.DataFrame):
        """
        Batch prediction method
        """
        # empty predictions list
        predictions = []

        for _, x in X.iterrows():
            # initializes an empty dictionary
            values = {}
            for feature in self.features:
                # Retrieves the corresponding value
                values.update({feature: x[feature]})

            # The predicted class label is appended
            predictions.append(self.predict_obs(values))

        return predictions
    def predict_obs(self, values: dict) -> int:
        """
        Method to predict the class given a set of features
        """
        # Assigning the current node to the root of the decision tree
        cur_node = self

        # Loop until the current node depth reaches the max depth
        while cur_node.depth < cur_node.max_depth:
            # Retrieves the best feature and best value for splitting at the current node
            best_feature = cur_node.best_feature
            best_value = cur_node.best_value

            # Flag variable to indicate if the loop should break
            should_break = False

            # Check if best_feature and best_value are not None
            if best_feature is not None and best_value is not None:
                # Check if there is a left child node
                if values.get(best_feature) < best_value:
                    if cur_node.left is not None:
                        cur_node = cur_node.left
                # Check if there is a right child node
                else:
                    if cur_node.right is not None:
                        cur_node = cur_node.right
            else:
                should_break = True

            # Break the loop if the flag is set to True
            if should_break:
                break

        # Return the predicted class
        return cur_node.yhat

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import time

# Start the timer
start_time = time.time()

# Setting the available hyper parameters of a tree
hp = {
    'max_depth': 100,
    'min_samples_split': 2
}

# Create an instance of the Node class
tree = Node(y_train, X_train, **hp)

# Grow the decision tree
```

```
tree.grow_tree()

# Print the tree
tree.print_tree()

# Make predictions on the test set
y_pred = tree.predict(X_test)

# End the timer
end_time = time.time()

# Calculate the elapsed time
execution_time = end_time - start_time

# Convert the predicted labels to a NumPy array
predicted_labels = np.array(y_pred)

# Calculate the accuracy by comparing predicted labels with true labels
accuracy = np.sum(predicted_labels == y_test) / len(y_test) * 100

# Print the accuracy
print("Accuracy: {:.2f}%".format(accuracy))
# Print the computational time
print(f"Computational time: {execution_time} seconds")

#@title sklearn decision tree

# Compare with built-in decision tree
# Import libraries

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Start timer
start_time = time.time()

# Create an instance of DecisionTreeClassifier
tree = DecisionTreeClassifier(min_samples_split=2, max_depth=100)

# Fit the decision tree on the training data
tree.fit(X_train, y_train)

# Make predictions on the test data
predictions = tree.predict(X_test)

# Print the accuracy
print("Accuracy: {:.2f}%".format(accuracy))

# End timer
end_time = time.time()

# Calculate the elapsed time
execution_time = end_time - start_time
```

```
# Print the computational time
print(f"Computational time: {execution_time} seconds")

#@title feature weights importance
# Get the feature importances
feature_importance = tree.feature_importances_

# Create a DataFrame to display the feature importances
importance_df = pd.DataFrame({'Feature': X_test.columns, 'Importance': feature_importance})
importance_df = importance_df.sort_values(by='Importance', ascending=False)

# Print the feature importances
print(importance_df)
```

B.3 KNN

```
# Application of Class Material:
# Initially, the distance matrix was computed every time, making our implementation run in O(n^2)
# and having an abnormally high flop count because of it. In this new implementation,
# we only compute the euclidean distance matrix once, caching the results and referencing
# the same instance over and over, now just running in O(1).

import math
import numpy as np
from scipy.stats import mode

class KNN:
    def __init__(self, k):
        self.k = k
        self.X_train = None
        self.y_train = None
        self.distances = None

    def fit(self, X_train, y_train):
        self.X_train = np.array(X_train)
        self.y_train = np.array(y_train)
        self.distances = self.compute_distance_matrix()

    def compute_distance_matrix(self):
        num_test = len(self.X_train)
        num_train = len(self.X_train)
        dist_matrix = np.zeros((num_test, num_train))

        for i in range(num_test):
            dist_matrix[i] = np.sqrt(np.sum(np.square(self.X_train - self.X_train[i]), axis=1))

        return dist_matrix

    def predict(self, X_test):
        y_pred = []
        X_test = np.array(X_test)
        for i in range(len(X_test)):
            dist = np.sqrt(np.sum(np.square(self.X_train - X_test[i]), axis=1))
            indices = np.argsort(dist)[:self.k]
            k_nearest_labels = self.y_train[indices]
            y_pred.append(mode(k_nearest_labels).mode[0])

        return y_pred

import time
knn = KNN(k=15)

# Train the model
knn.fit(X_train, y_train)

# Make predictions
start_time = time.time()
predictions = knn.predict(X_test)
total_time = time.time() - start_time
print("Training Time:", total_time)
# Print the predictions
print(predictions)

y_testval = y_test
correct = 0
true_positives = 0
false_positives = 0
#Y_test_numeric = []
#predictions_numeric = []
#for x in range(len(predictions)):
#  if predictions[x].equals
#  Y_test_numeric.append()
for x in range(len(y_testval)):
  if y_testval[x] == predictions[x]:
    correct = correct+1
accuracy = correct / len(y_testval)
print("Accuracy:", accuracy)

# Compute precision
for x in range(len(y_testval)):
  if y_testval[x] == 1 and predictions[x] == 1:
    true_positives = true_positives+1

for x in range(len(y_testval)):
  if y_testval[x] == 0 and predictions[x] == 1:
    false_positives = false_positives+1

#false_positives = sum((predictions == 'acquired') & (y_testval == 'closed'))
precision = true_positives / (true_positives + false_positives)
print("Precision:", precision)

from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=15)
neigh.fit(X_train, y_train)
```

```
start_time = time.time()
predictions = neigh.predict(X_test)
total_time = time.time() - start_time
print("Training Time:", total_time)

y_testval = y_test
correct = 0
true_positives = 0
false_positives = 0
#Y_test_numeric = []
#predictions_numeric = []
#for x in range(len(predictions)):
#  if predictions[x].equals
#  Y_test_numeric.append()
for x in range(len(y_testval)):
  if y_testval[x] == predictions[x]:
    correct = correct+1
accuracy = correct / len(y_testval)
print("Accuracy:", accuracy)

# Compute precision
for x in range(len(y_testval)):
  if y_testval[x] == 1 and predictions[x] == 1:
    true_positives = true_positives+1

for x in range(len(y_testval)):
  if y_testval[x] == 0 and predictions[x] == 1:
    false_positives = false_positives+1

#false_positives = sum((predictions == 'acquired') & (y_testval == 'closed'))
precision = true_positives / (true_positives + false_positives)
print("Precision:", precision)

# Find best K value with highest accuracy
import matplotlib.pyplot as plt

# Generate odd values of k from 1 through 30
k_values = list(range(1, 31, 2))

# Store accuracy values and corresponding k values
accuracies = []
best_accuracy = 0.0
best_k = 0

# Iterate over each odd value of k
for k in k_values:
    # Create an instance of the KNN class with the current value of k
    knn = KNN(k)

    # Fit the model
    knn.fit(X_train, y_train)

    # Predict labels for the test set
    y_pred = knn.predict(X_test)
    # Calculate accuracy
    accuracy = np.mean(y_pred == y_test)

    # Append accuracy and k value to the lists
    accuracies.append(accuracy)

    # Check if the current accuracy is the best so far
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_k = k

# Plot the accuracy vs. k values
plt.plot(k_values, accuracies, marker='o')
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. k')
plt.show()

print("Best k value:", best_k)
print("Best accuracy:", best_accuracy)
```

B.4 SVM

[1]

```
# Load necessary libraries
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import random

# Application of Class Material:
# Rather than solving for SVM using the closed form quadratic solve, this SVM
# class is trained iteratively using the gradient descent algorithm that was
# discussed in class. Further, SGD is used with a batch size of 1 to speed up
# training time. The learning rate for the model was also tuned to overcome
# the problems with convergence discussed during lecture.

class SVM():
    def __init__(self, X, y, C, learning_rate, eps, random_state=40):
        # Initialize all class members
        self.X = X
        self.y = y
        self.weights = np.zeros(self.X.shape[1])
        self.C = C
        self.learning_rate = learning_rate
        self.eps = eps
        self.losses = []
        self.random_state = random_state
        random.seed(random_state)

    def _compute_cost(self):
        N = self.X.shape[0]
        # Compute model loss
        distances = 1 - np.multiply(self.y[:,0], (np.dot(self.X, self.weights)))

        # Only track positive loss (based on hingle loss formulation)
        distances[distances < 0] = 0
        hinge_loss = self.C * (np.sum(distances) / N)

        # Compute and return model loss at current state
        cost = 1/2 * np.dot(self.weights, self.weights) + hinge_loss
        return cost

    def _calculate_cost_gradient(self, W, X_batch, Y_batch):
        # Compute error/loss
        distance = 1 - np.multiply(Y_batch[:,0], (np.dot(X_batch, W)))

        # Initialize gradient to zeros
        dw = np.zeros(len(W))

        for idx, d in enumerate(distance):
            if max(0, d) == 0: # Ignore negative distances
                di = W
            else: # Update gradient for given weight
                di = W - (self.C * Y_batch[idx] * X_batch[idx, :])
            dw += di

        # Normalize gradient
        dw = dw/len(Y_batch)
        return dw

    def fit_model(self, max_epochs=5000):
        # Initialize all weights to zero
        self.weights = np.zeros(self.X.shape[1])

        old_cost = 0
        # Run gradient descent for specified number of epochs
        for epoch in range(1, max_epochs):
            # Shuffle data to ensure independence of observations
            X_shuffled, y_shuffled = shuffle(self.X, self.y, random_state=self.random_state + epoch)

            # Set the number of random samples to retrieve
            n = 1

            # Get n random samples from the shuffled sequences
            random_indices = random.sample(range(len(X_shuffled)), n)
            random_X_samples = np.array([X_shuffled[i] for i in random_indices])
            random_y_samples = np.array([y_shuffled[i] for i in random_indices])

            # Compute gradient
            ascent = self._calculate_cost_gradient(self.weights, random_X_samples, random_y_samples)
            # Update weights based on gradient
            self.weights = self.weights - (self.learning_rate * ascent)

            # Compute new cost
            cost = self._compute_cost()

            # Store cost
            self.losses.append(cost)

            # Terminate if cost does not decrease substantially
            if abs(cost - old_cost) < self.eps * old_cost:
                print("terminating on epoch {0}".format(epoch))
                return

            # Update old cost
            old_cost = cost

    def predict(self, X):
        predictions = np.array([])
        for i in range(X.shape[0]):
            prediction = np.sign(np.dot(X[i], self.weights))
            predictions = np.append(predictions, prediction)
        return predictions

# Prepare data
y_train_svm = pd.DataFrame(y_train.copy())
y_test_svm = pd.DataFrame(y_test.copy())

# Make output class -1 and 1
y_train_svm[0] = y_train_svm[0].replace({0: -1})
y_train_svm = y_train_svm.astype(np.float64)

y_test_svm[0] = y_test_svm[0].replace({0: -1})
y_test_svm = y_test_svm.astype(np.float64)

X_train_svm = pd.DataFrame(X_train.copy())
X_test_svm = pd.DataFrame(X_test.copy())

# Insert column of 1s into input data (for bias term)
X_train_svm.insert(0, "intercept", [1 for _ in range(X_train_svm.shape[0])])
X_test_svm.insert(0, "intercept", [1 for _ in range(X_test_svm.shape[0])])

# Perform hyperparameter tuning for learning rate
learning_rates = [0.0001, 0.01, 1]

for learning_rate in learning_rates:
    model = SVM(X_train_svm.to_numpy(), y_train_svm.to_numpy(), \
                C=1, learning_rate = learning_rate, eps = 1e-10)
    model.fit_model(max_epochs=50)
    plt.plot(model.losses, label=learning_rate)
plt.legend()
plt.title("Model Loss vs Epoch (Varying Learning Rates)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()

# Perform hyperparameter tuning for lambda
reg_params = [0.01, 0.001, 0.1, 1, 10, 100]

test_accs = []
for reg_param in reg_params:
    model = SVM(X_train_svm.to_numpy(), y_train_svm.to_numpy(), \
                C=reg_param, learning_rate = 0.01, eps = 1e-5)
    model.fit_model(max_epochs=50)

    test_predictions = model.predict(X_test_svm.to_numpy())
    test_accs.append(accuracy_score(y_test_svm, test_predictions))

plt.plot(reg_params, test_accs)
plt.title("Model Accuracy vs Regularion Parameter")
plt.xlabel("Regularization Parameter (lambda)")
plt.ylabel("Test Accuracy")
plt.show()

# Fit model on training set using optimal hyperparameters
import time
start_time = time.time()
model = SVM(X_train_svm.to_numpy(), y_train_svm.to_numpy(), \
            C=0.01, learning_rate = 0.01, eps = 1e-5)
model.fit_model(max_epochs=50)
total_time = time.time() - start_time
print("Training Time:", total_time)
train_predictions = model.predict(X_train_svm.to_numpy())
test_predictions = model.predict(X_test_svm.to_numpy())
print("accuracy on test dataset: {}".format(accuracy_score(y_test_svm, test_predictions)))

# Compare performance to sklearn implementation
from sklearn.linear_model import SGDClassifier
svm = SGDClassifier(loss='hinge', alpha=0.01, learning_rate='constant', eta0=0.01, \
                    max_iter=50, tol=1e-5, early_stopping=True)

start_time = time.time()
svm.fit(X_train_svm.to_numpy(), y_train_svm.to_numpy())
total_time = time.time() - start_time
print("Training Time:", total_time)

# Evaluate the accuracy of the model
accuracy = svm.score(X_test_svm.to_numpy(), y_test_svm.to_numpy().flatten())
print("Accuracy:", accuracy)
```