Machine Learning Stroke Prediction - Report

Group Members: Emily, Meme, Sherry and Hiam

February 19th, 2022

Github Link of Flask App | Heroku Link | Github Link to Model Code

---

Motivation

Annually, 15 million people worldwide suffer a stroke. Of these, 5 million die and another 5 million are left permanently disabled, placing a burden on family and community.(1) A stroke occurs when blood supply to parts of the brain is interrupted or reduced, which prevents brain tissue from getting it's needed nutrients. This begins to deteriorate brain cells within minutes. Strokes are the leading cause of long-term disability and carry a very high risk of death(2). It is commonly seen amongst those with high blood pressure, diabetes, obesity, and smokers of age 40 years and above mostly.(2) Are you at a high risk of developing a stroke? Let's find out!

Abstract

For this project, we used a csv file of 5110 patient data from Kaggle. The data was explored and manipulated using Pandas in order to populate a clean dataset. Once transformed and saved, the data was loaded onto MongoDB Atlas in the cloud. We further queried data in order to prepare it for model implementation and we used a variety of Sklearn libraries to test various models and optimize them to the best of our abilities.Given that the data we were working on was pre-labelled, we explored supervised classification machine learning.

Based on the models we compiled, the highest accuracy was achieved with the Random Forest Classifier. By using the optimization methods of SMOTE(oversampling), balancing and setting the following parameters: max_depth=50, max_features=5, min_sample_leaf=3, min_samples_split=8, n_estimators=200, the accuracy reached 95%. Based on the trained model, we created an app built with this model to give predictions based on the user's input information.

Project Objective

By training the dataset with the appropriate models and stating their comparisons, the most optimized model will be chosen in order to predict stroke risk with the best accuracy test score. In addition, we will connect the model to a front end flask application which will predict whether the user is at risk or not at risk to develop a stroke.

Methods

Data Sources Used: Kaggle Stroke Prediction

Link: https://www.kaggle.com/fedesoriano/stroke-prediction-dataset

Final Production Database: MongoDB Atlas

Libraries Used:

- Matplotlib.pyplot
- Pandas
- Seaborn
- Pymongo
- Sklearn
- Pickle

Data Engineering

Please refer back to the Features_Analysis_and_Data_Cleaning Jupyter Notebook in order to see in depth exploration and cleaning of the data

Understanding Features from Kaggle

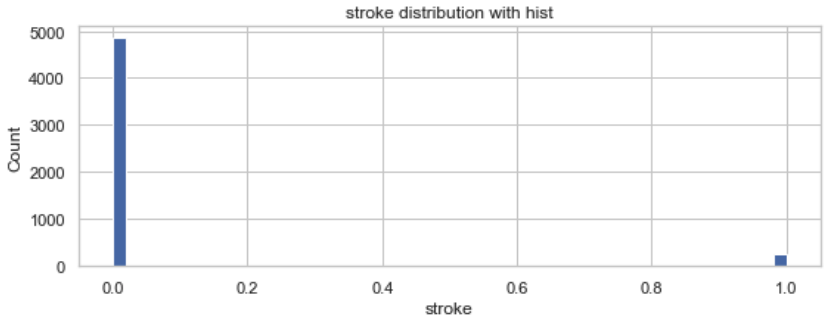
When retrieving the data from Kaggle, it was accompanied with descriptions of each feature:

- **id:** unique identifier
- **gender:** "Male", "Female" or "Other"
- **age:** age of the patient
- **hypertension:** 0 if the patient doesn't have hypertension, 1 if the patient has hypertension
- **heart_disease:** 0 if the patient doesn't have any heart diseases, 1 if the patient has a heart disease
- **ever_married:** "No" or "Yes"
- **work_type:** "children", "Govt_jov", "Never_worked", "Private" or "Self-employed"
- **Residence_type:** "Rural" or "Urban"
- **avg_glucose_level:** average glucose level in blood (mg/dL)
- **bmi:** body mass index
- **smoking_status:** "formerly smoked", "never smoked", "smokes" or "Unknown"*
- **stroke:** 1 if the patient had a stroke or 0 if not

Exploring the Data

We were able to understand that the data was composed of various different data types, which were further emphasized by applying the .info() function. Overall, our features were grouped as such:

- Categorical Features: ‘gender’, ‘ever_married’, ‘work_type’, and ‘Residence_type’
- Numerical Features: ‘Hypertension’, ‘heart_disease’, ‘age’, ‘bmi’ and ‘avg_glucose_level’
- Target Feature: ‘stroke’



As seen in the above figure, we noticed a large imbalance in our target feature, with 4861 labeled as 0 (no stroke) and 249 labeled as 1 (stroke). Given the evident sparseness in the numbers, we wanted to explore stroke cases on particular categorical features.

Based on the results obtained, out of those who did get a stroke, 56.6% were Females, 43.4% were Males, 88.4% are or have been married, 11.6% have not. As for work type, those who work in the private sector are the most prone to getting a stroke with a count of 149 out of the 249 cases. Rural residents represent 45.8% of those who get strokes, and 54.2% are in an urban residence. As for the smoking status, those who have formerly smoked represented 70 out of 249 cases, unlike those who do smoke with only 42 out of the 249. Surprisingly, those who have never smoked represent the highest count of stroke victims.

Features Correlation

As seen in the visual above, it is understood that all relationships between any of the numerical features does not have a strong correlation. All the values seem to be closer to 0, rather than 1. If we look closely, age may be a feature that has the highest correlation with stroke with 0.25, but that is still considered low. Overall, the above visual indicates that each feature is contributing independently and thus, all features will be considered for training the model.

Data Cleaning - Anomaly Detection

Anomaly detection is generally understood to be the identification of rare items, events or observation which deviate significantly from the majority of the data. An anomaly detection pattern produces two different results: categorical tag and score or trust value. The categorical tag just tells you if an observation is abnormal or not, and the score will tell you 'how' abnormal that observation is. [3]

```
def detect_outliers(df,features):
    outlier_indices = []

    for c in features:
        # 1st quartile
        Q1 = np.percentile(df[c],25)
        # 3rd quartile
        Q3 = np.percentile(df[c],75)
        # IQR
        IQR = Q3 - Q1
        # Outlier step
        outlier_step = IQR * 1.5
        # detect outlier and their indeces
        outlier_list_col = df[(df[c] < Q1 - outlier_step) | (df[c] > Q3 + outlier_step)].index
        # store indeces
        outlier_indices.extend(outlier_list_col)

    outlier_indices = Counter(outlier_indices)
    multiple_outliers = list(i for i, v in outlier_indices.items() if v > 2)

    return multiple_outliers
```

By running the above function and reading the dataset, it resulted with 75 outliers that were then removed.

Data Cleaning - Null Values

Based on our previous analysis of the data, we had noticed that the BMI column was missing values. When we ran an .isnull() function on the BMI column, 192 rows showed up. Now rather than completely dismissing them, we looked at the BMI distribution amongst the gender types provided, as gender is one of the factors that help define the range of BMI.

Although not a large difference amongst men and women, the above boxplot shows that there is a slight higher BMI for women than men. For the ‘Other’ gender type, there isn’t enough data to help identify the BMI range, which led us to also removing that one row of data before using the data for model training.

To handle the Null values of BMI, we assigned the total BMI mean that was calculated on the gender of the observation.

```
print("Mean of BMI value for Females: ", np.mean(dataset[dataset['gender'] == 'Female']['bmi']))
print("Mean of BMI value for Males: ", np.mean(dataset[dataset['gender'] == 'Male']['bmi']))
print("Mean of BMI value: ", np.mean(dataset['bmi']))
```

```
Mean of BMI value for Females:  29.035926055109872
Mean of BMI value for Males:  28.5946835443038
Mean of BMI value:  28.854614908114808
```

```
dataset['bmi'] = dataset['bmi'].fillna(0)
```

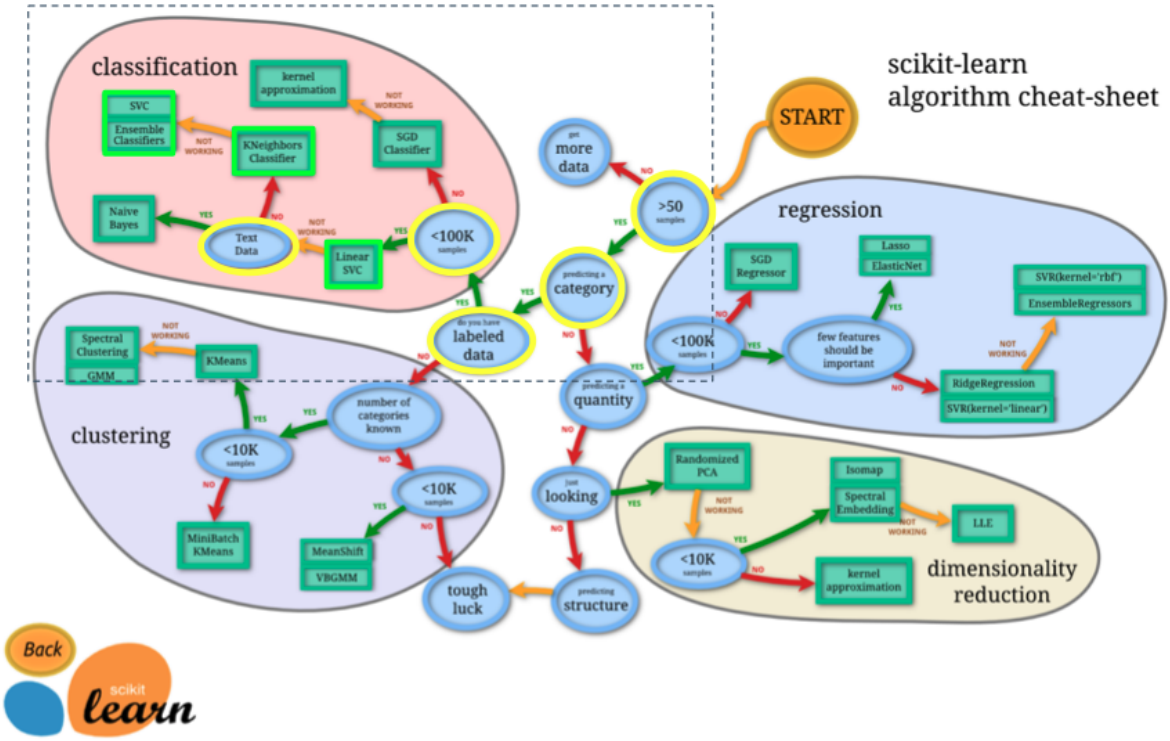
```
for i in range(0, 5034):
    if(dataset['bmi'][i] == 0):
        if(dataset['gender'][i] == 'Male'):
            dataset['bmi'][i] = 28.5946835443038
        elif(dataset['gender'][i] == 'Female'):
            dataset['bmi'][i] = 29.035275645498956
        else:
            dataset['bmi'][i] = 28.85419248244527
```

Once that was complete, the dataset was saved as a clean CSV locally and then loaded onto MongoDB Atlas for Model Data Implementation.

Model Data Implementation

Please refer back to the [Model Data Implementation] Folder in order to see in depth Model Data Implementation and Optimization processes.

One of the most challenging parts of solving a machine learning problem can be choosing the right estimator for the job. Different estimators are suited for different types of data and different problems. The flowchart below is designed to give a rough guide on how to approach problems with regard to which estimators to try on the data. Since we have less than 100K rows of data but more than 50, we started by testing out the SVM Algorithm, followed by the KNN model, then Decision Tree Classification, and finally the Random Forest Classification.



Support Vector Machines (SVM)

A support vector machine (SVM) is a supervised machine learning model that uses classification algorithms for two group classification problems.[4] It is fast and dependable and performs very well with a limited amount of data to analyze. SVM has several kernel types to be used in the algorithm: linear and rbf are the most commonly used. Generally, the ultimate goal is to find the optimal hyperplanes that best separates the dataset and typically these hyperplanes are chosen such that the 'margin' between the place and data points of different classes on either side is maximized. Now specifically for Linear SVM, it is assumed that the training data is 'linearly' separable.

Our first attempt of the linear SVM model resulted in the following confusion matrix:

	precision	recall	f1-score	support
0	0.96	1.00	0.98	962
1	0.00	0.00	0.00	45
accuracy			0.96	1007
macro avg	0.48	0.50	0.49	1007
weighted avg	0.91	0.96	0.93	1007

It is important to note that the parameters of this model were kept at default to see what we would get as an initial result. The confusion matrix shows a high accuracy of 96%, however the recall, precision and f1-score are at a 0 for the Stroke (1) group. Because those that do get strokes are a minority in our dataset, that means that the model is unable to correctly identify any of the True Positives. We then tested as another first attempt the non-linear SVM model to see if the result would be any different with our imbalance dataset, and it was in fact the exact same.

K-Nearest Neighbors (KNN)

The K-nearest neighbor (KNN) algorithm is a simple, easy to implement supervised machine learning algorithm that can be used for classification problems.[4] The KNN algorithm assumes that similar data points exist in close proximity. Specifically, a class label assigned to the majority of K-nearest neighbors from the training data set is considered as a predicted class for any new data point. As a first attempt, this was the resulting confusion matrix:

	precision	recall	f1-score	support
0	0.96	1.00	0.98	962
1	0.20	0.02	0.04	45
accuracy			0.95	1007
macro avg	0.58	0.51	0.51	1007
weighted avg	0.92	0.95	0.93	1007

Just like for the SVM model, all parameters were at default for this first attempt to establish the baseline for all models being tested. The confusion matrix shows a high accuracy of 95%, however the recall was of 2%, precision of 20% and F1-score of 4% for the Stroke (1) group. This model, just like the previous one, has a hard time identifying True Positives when testing. Our precision score is higher than our previous model, meaning that it may have caught some True Positive and False Positives. But given the recall is so low, it most likely means it caught some False Positives mostly, which is further supported by the F1 score.

Decision Tree Classification (DTC)

Decision Trees are a non-parametric supervised learning method used for classification and regression.[4] The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from data features.

Our first attempt of the Decision Tree model resulted in the following confusion matrix:

	precision	recall	f1-score	support
0	0.94	0.90	0.92	978
1	0.90	0.94	0.92	957
accuracy			0.92	1935
macro avg	0.92	0.92	0.92	1935
weighted avg	0.92	0.92	0.92	1935

The Decision Tree model, all parameters were at default for this first attempt to establish the baseline for all models being tested. The confusion matrix shows a high accuracy of 92%, the recall was of 94%, precision of 90%.

Random Forest Classification (RFC)

Random Forest Classification (RFC) is a supervised learning model that uses bagging and features randomness to build each tree independently and give more precise predictions for the dataset.[4] The model operates by building multiple trees at a time and gives the mode/mean training score for all the individual trees.

What's also very encouraging in using this model specifically is the result we received in the correlation heatmap* Based on the plot, all the features had low correlations, which informs us that the features are all working independently from each other. So this would satisfy the RM prerequisite of building independent trees.

	precision	recall	f1-score	support
0	0.94	0.98	0.96	507
1	0.98	0.93	0.96	500
accuracy			0.96	1007
macro avg	0.96	0.96	0.96	1007
weighted avg	0.96	0.96	0.96	1007

* Please refer back to the ***Data Engineering - Features Correlation*** Section

For our first attempt, with the data being unbalanced and not scaled, we got 95% in accuracy with a recall score of 93%. Based on this model, we are trying to improve the results without compromising the recall performance, as it was the best one out of all four first attempts seen.

Model Data Optimization

As optimization tools, we collectively used the following methods

1. *GridSearchCV Optimization*

GridSearchCV Optimization is an effective method for adjusting the parameters in supervised learning and improving the generalization performance of the model. With GridSearchCV, all possible combinations of the parameters of interest are tested and the best ones are displayed. (5)

Here is a sample code from the GridSearchCV Optimization for the KNN Model*:

```
# defining parameter range
param_grid = {'n_neighbors': [1,3,5,7,9,11,15,19],
              'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
             'weights': ['uniform', 'distance'],
             'metric': ['manhattan', 'euclidean', 'minkowski', 'cosine', 'jaccard', 'hamming']}

grid = GridSearchCV(knn_model, param_grid, refit = True, verbose = 3)

# fitting the model for grid search
grid.fit(X_train_scaled, y_train)
```

* *The parameters vary from model to model.*

2. *SMOTE*

SMOTE is an oversampling technique where the 'synthetic' samples are generated for the **minority** class, in our case being the 'stroke' class 1. This algorithm focuses on the feature space to generate new instances with the help of interpolation between the positive instances that lie together. In other words, it will randomly increase the number of minority class examples by replicating them. Oversampling our data will allow for less bias generation from the machine learning algorithm towards the majority class. (6)

Here is a sample code from the SMOTE function:

```
from imblearn.over_sampling import SMOTE
#Define indpendent and dependent variables - and remove the variable to be predicted
X = en_dataset.drop('stroke', axis=1)
y = en_dataset['stroke']
smote = SMOTE()
X,y = smote.fit_resample(X,y)
```

```
smote_dataset = pd.concat([X,y], axis=1)
```

3. *Scaling Data*

When we want all features to be shifted to similar numeric scales so that the magnitude of one feature doesn't bias the modely during training. Using StandardScaler from Scikit-Learn scales data to have a mean of 0 and variance of 1. It is a recommended tool when you do not have complete knowledge of your data. However, as we will see later on, scaling data does not always mean better performance of the model.

```
sc = StandardScaler()
X_train_scaled = sc.fit_transform(X_train)
X_test_scaled = sc.transform(X_test)
```

1. *Stratifying*

Stratify is a function in classification setting which is designed to equally distribute the features of the dataset.In the preprocessing part, we noticed the dataset is unbalanced and biased, the test data and training data could not be split as the exact portion we designed. We use this function to ensure that the train and test sets have approximately the same percentage of samples of each target class as the complete set.

```
# split into train test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42, stratify=y)
print(Counter(y_train))
print(Counter(y_test))
# Balanced when test_size is 50%
# Counter({0: 4794, 1: 306})
# Counter({0: 2397, 1: 153})
# Counter({0: 2397, 1: 153})
```

Support Vector Machines (SVM)

Attempt #2: **kernel = 'linear' + SMOTE + scaled**

	precision	recall	f1-score	support
0	0.82	0.81	0.82	978
1	0.81	0.82	0.81	957
accuracy			0.81	1935
macro avg	0.82	0.82	0.81	1935
weighted avg	0.82	0.81	0.81	1935

Attempt #3: **kernel = 'rbf' +** ***SMOTE*** **+ scaled**

	precision	recall	f1-score	support
0	0.89	0.85	0.87	978
1	0.85	0.89	0.87	957
accuracy			0.87	1935
macro avg	0.87	0.87	0.87	1935
weighted avg	0.87	0.87	0.87	1935

Attempt #4: **kernel = 'rbf' + SMOTE + scaled + GridSearchCV**

```
{'C': 100, 'gamma': 1, 'kernel': 'rbf'}
SVC(C=100, gamma=1)
```

	precision	recall	f1-score	support
0	0.95	0.91	0.93	978
1	0.91	0.95	0.93	957
accuracy			0.93	1935
macro avg	0.93	0.93	0.93	1935
weighted avg	0.93	0.93	0.93	1935

K-Nearest Neighbors (KNN)

Attempt #2: **SMOTE + scaled**

	precision	recall	f1-score	support
0	0.95	0.85	0.90	978
1	0.86	0.96	0.91	957
accuracy			0.90	1935
macro avg	0.91	0.90	0.90	1935
weighted avg	0.91	0.90	0.90	1935

Attempt #3: **SMOTE + scaled + GridSearchCV**

```
{'algorithm': 'auto', 'metric': 'manhattan', 'n_neighbors': 2, 'weights': 'uniform'}
KNeighborsClassifier(metric='manhattan', n_neighbors=2)
```

	precision	recall	f1-score	support
0	0.94	0.92	0.93	978
1	0.92	0.94	0.93	957
accuracy			0.93	1935
macro avg	0.93	0.93	0.93	1935
weighted avg	0.93	0.93	0.93	1935

Decision Trees

Attempt #2: **SMOTE + splitter='random'**

	precision	recall	f1-score	support
0	0.50	0.49	0.50	968
1	0.50	0.51	0.51	967
accuracy			0.50	1935
macro avg	0.50	0.50	0.50	1935
weighted avg	0.50	0.50	0.50	1935

Attempt #3: **SMOTE + criterion='gini', splitter='random'**

	precision	recall	f1-score	support
0	0.94	0.90	0.92	968
1	0.91	0.94	0.92	967
accuracy			0.92	1935
macro avg	0.92	0.92	0.92	1935
weighted avg	0.92	0.92	0.92	1935

Random Forest Classification

Attempt #2: **Stratify**

	precision	recall	f1-score	support
0	0.93	0.97	0.95	504
1	0.97	0.93	0.95	503
accuracy			0.95	1007
macro avg	0.95	0.95	0.95	1007
weighted avg	0.95	0.95	0.95	1007

Attempt #3: **SMOTE** + **StandardScaler**

	precision	recall	f1-score	support
0	0.95	0.93	0.94	979
1	0.93	0.96	0.94	956
accuracy			0.94	1935
macro avg	0.94	0.94	0.94	1935
weighted avg	0.94	0.94	0.94	1935

Attempt #4 **SMOTE** + **StandardScaler** + **Stratify**

	precision	recall	f1-score	support
0	0.97	0.95	0.96	970
1	0.95	0.97	0.96	965
accuracy			0.96	1935
macro avg	0.96	0.96	0.96	1935
weighted avg	0.96	0.96	0.96	1935

Attempt #5: ***SMOTE + GridSearchCV + Stratify***

```
{'bootstrap': True, 'max_depth': 50, 'max_features': 5, 'min_samples_leaf': 3, 'min_samples_split': 8, 'n_estimators': 200}
RandomForestClassifier(max_depth=50, max_features=5, min_samples_leaf=3,
                       min_samples_split=8, n_estimators=200)
```

	precision	recall	f1-score	support
0	0.96	0.93	0.95	968
1	0.93	0.96	0.95	967
accuracy			0.95	1935
macro avg	0.95	0.95	0.95	1935
weighted avg	0.95	0.95	0.95	1935

Saving the Model

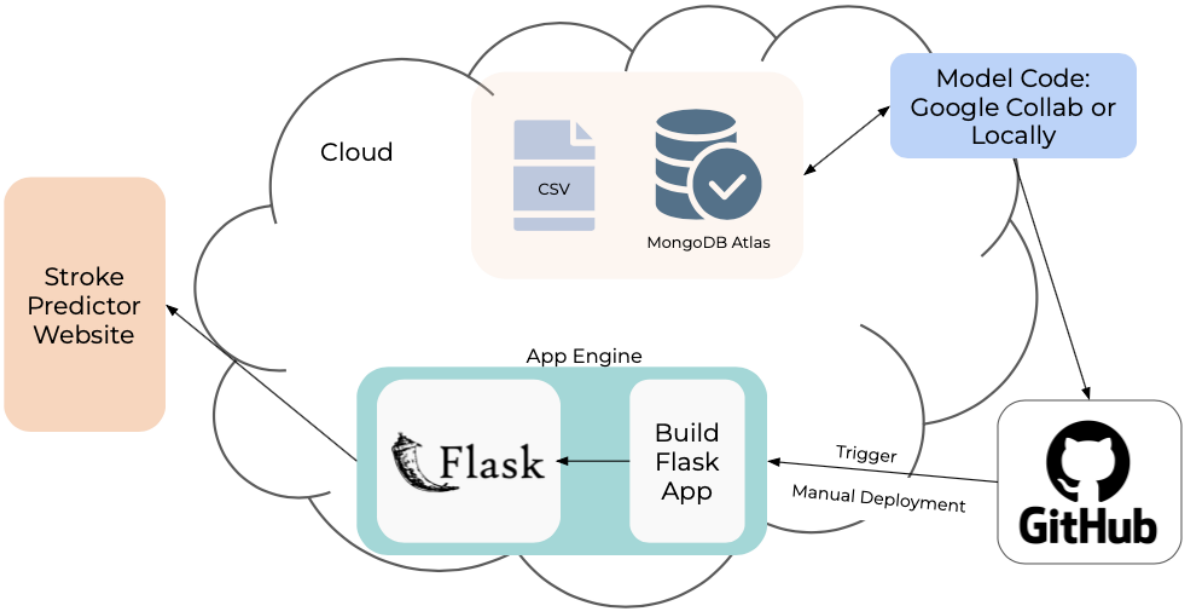
Considered for the model we tested. The attempt 5 would be our final choice for stroke prediction that can provide the most well-rounded performance so far. To save the model for the front end app, we import the pickle function to export the model as a pickle file.

```
# #Save best model using Pickle
with open ('../Models/rf_model_final_4.pkl','wb') as f:
    pickle.dump(grid_search.best_params_,f)
```

Front-End Application & Demo

Please refer back to the [Stroke Predicting Model]Repository in order to see details on deploying the application.

Architecture

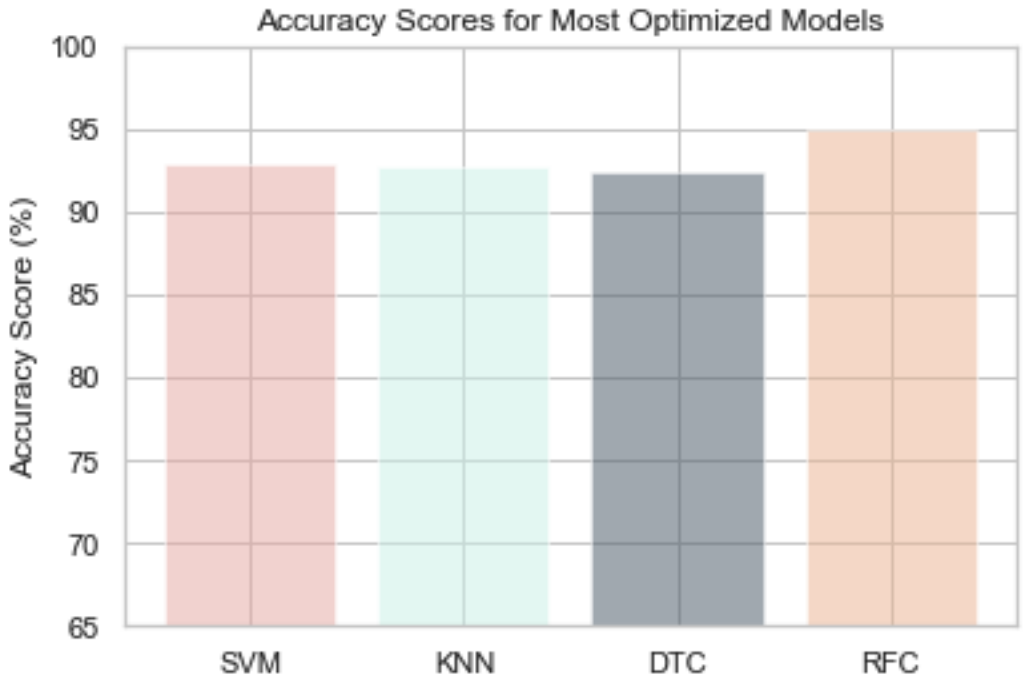


The above diagram is the cloud architecture of our stroke prediction system. Inside the cloud diagram we have our cloud services. The mongoDb Atlas database stores the cleaned kaggle stroke dataset that is loaded there. MongoDB Atlas retrieves the clean data to build the final model. The final model file is pushed to GitHub. We update our GitHub repo by merging the feature branch into the master branch and push up the changes. The changed part of the code of the website or layout will now be updated on GitHub. The 'Cloud Build' is set to manually deploy the code updates into the production flask container. Once this is done the changes will be seen on the website.

Wep Application

'app.py' has the main function and contains all the required functions for the flask app. In the code, we have created the instance of the Flask() and loaded the model.The model.predict() method takes input from the request (once the 'compute' button from index.html is pressed) and converts it into an array. The results are checked for missing values and temporarily saved in a dataframe 'stroke_df'. This dataframe is passed to the model.predict() method where the result is rendered back to the user as 'feedback' for the prediction of risk or not at risk.

Conclusions



As seen in the visual above, after optimization procedures, all models accuracy reached above 90%. The Random Forest Classification Model was the most accurate algorithm after optimization with a 95% accuracy score. We considered it as the “best choice” so far based on several considerations: Higher accuracy score, better performance on recall score and individual features of input information.

Limitations

- Limited Model Types:
  - For further improvement on the models, we would look into other models such as logistical regression and try other optimization methods like improving the bagging error.
- Using SMOTE in intervals:
  - Our optimized accuracy could have been achieved prior to having fully replicated our minority classification.
- Stroke Data:
  - Effect of external factors on the stroke output (ex. Location, race)
  - Limited number of features overall
  - Timeline of data collection is not indicated
- Low correlations among features
  - Limitation of choice of models
  - Need grid search to help with best features
  - But good for Random forest prediction as they are individual trees predictions

---

Future Considerations

- Testing the data with new dataset observations to check the accuracy of the model.
- Considering incorporating more data with similar datasets.
- Consider incorporating this data with other datasets, for example pollution, or pandemic data. It would be great to see if the possibility of stroke would be increased/decreased with any of the pollution data. As well, it would be nice to see if stroke increased/decreased during the pandemic or after it. Can we tell if getting the coronavirus increased our risk of stroke for a short or long period of time?

References

Data set Link: https://www.kaggle.com/fedesoriano/stroke-prediction-dataset

(1) http://www.emro.who.int/health-topics/stroke-cerebrovascular-accident/index.html
(2) https://www.cdc.gov/stroke/about.htm
(3) https://avinetworks.com/glossary/anomaly-detection/
(4) https://scikit-learn.org/stable/supervised_learning.html#supervised-learning
(5) https://towardsdatascience.com/gridsearchcv-for-beginners-db48a90114ee
(6) https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/
(7) https://scikit-learn.org/stable/supervised_learning.html
(8) https://realpython.com/flask-by-example-part-1-project-setup/