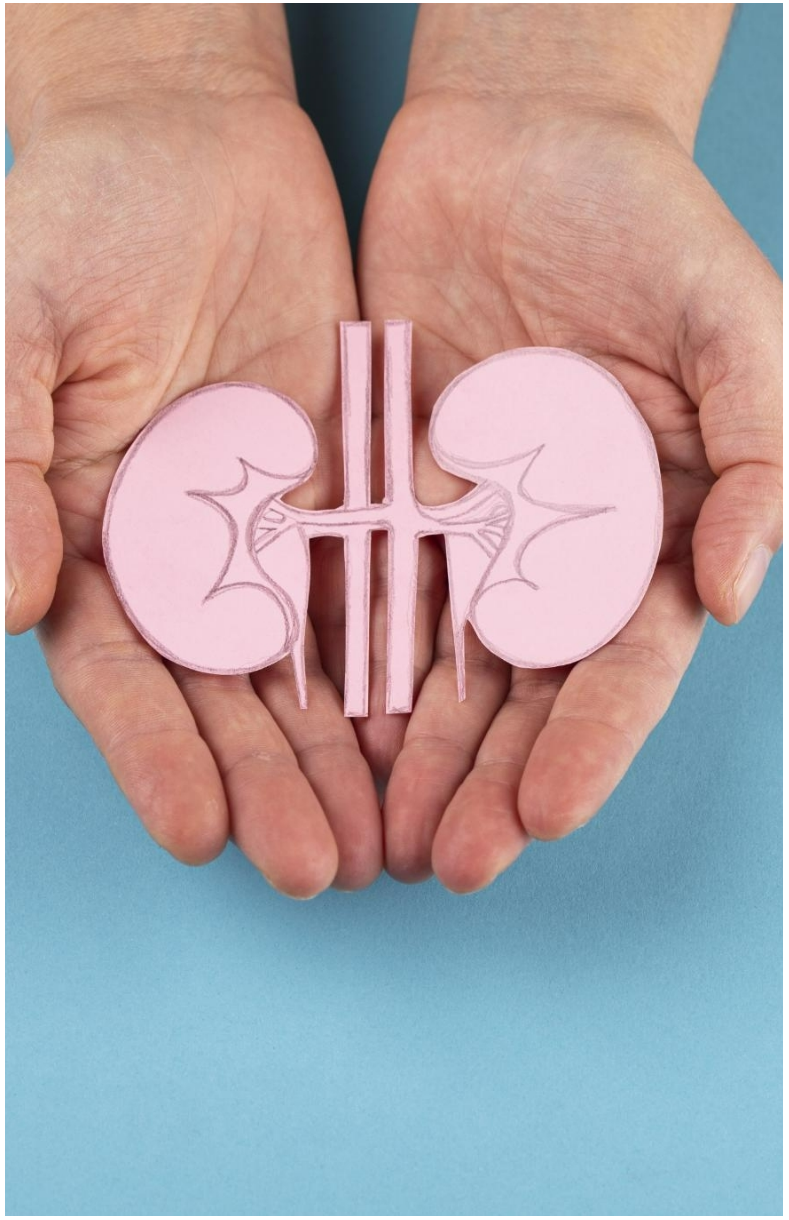
BUILDING A MODEL FOR PREDICTING ACUTE KIDNEY FAILURE BASED ON THE MIMIC-IV CLINICAL DATABASE

Joseph Barnes

AI in Healthcare

University of Texas at Austin

THE CLINICAL CHALLENGE OF ACUTE KIDNEY FAILURE (AKI)



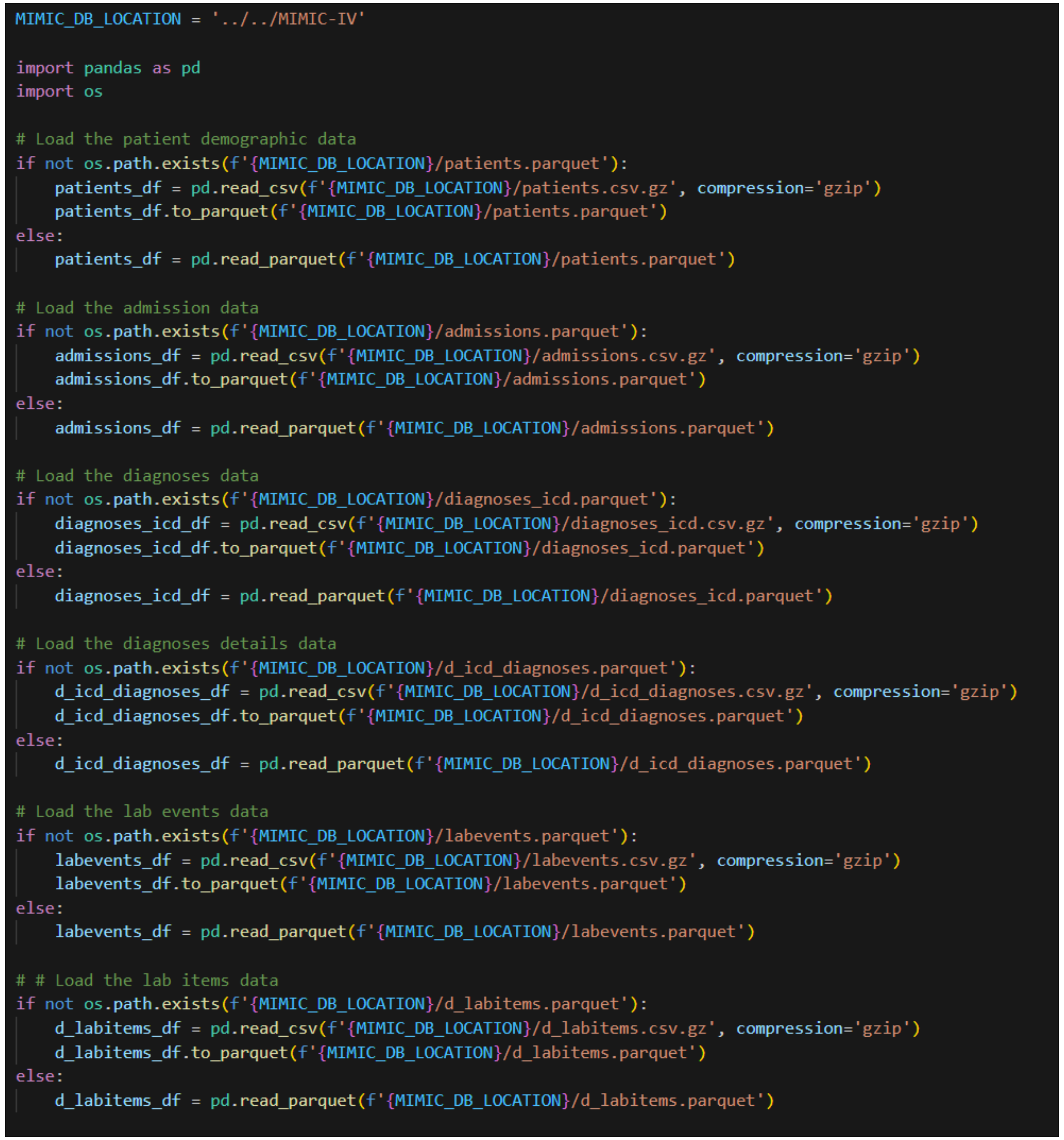
Acute Kidney Failure is a common problem with a high impact on patient outcomes

Early detection can significantly improve patient outcomes

Early detection has the potential to reduce morbidity and healthcare costs

LOADING THE MIMIC-IV DATA TABLES

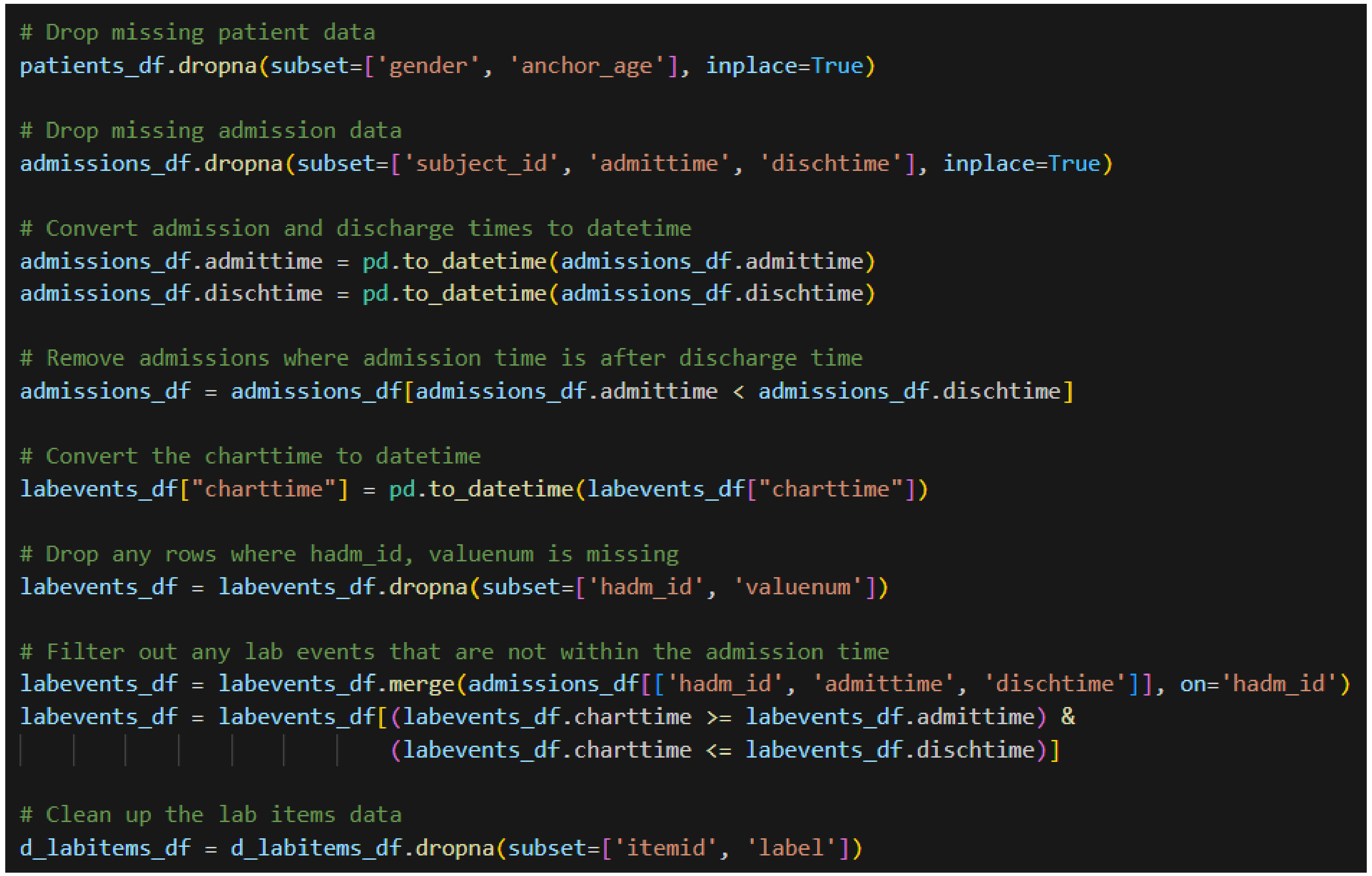
- Patients and Admissions data
- Diagnoses data and details
- Lab events data and details



```python
MIMIC_DB_LOCATION = '../../MIMIC-IV'

import pandas as pd
import os

# Load the patient demographic data
if not os.path.exists(f'{MIMIC_DB_LOCATION}/patients.parquet'):
    patients_df = pd.read_csv(f'{MIMIC_DB_LOCATION}/patients.csv.gz', compression='gzip')
    patients_df.to_parquet(f'{MIMIC_DB_LOCATION}/patients.parquet')
else:
    patients_df = pd.read_parquet(f'{MIMIC_DB_LOCATION}/patients.parquet')

# Load the admission data
if not os.path.exists(f'{MIMIC_DB_LOCATION}/admissions.parquet'):
    admissions_df = pd.read_csv(f'{MIMIC_DB_LOCATION}/admissions.csv.gz', compression='gzip')
    admissions_df.to_parquet(f'{MIMIC_DB_LOCATION}/admissions.parquet')
else:
    admissions_df = pd.read_parquet(f'{MIMIC_DB_LOCATION}/admissions.parquet')

# Load the diagnoses data
if not os.path.exists(f'{MIMIC_DB_LOCATION}/diagnoses_icd.parquet'):
    diagnoses_icd_df = pd.read_csv(f'{MIMIC_DB_LOCATION}/diagnoses_icd.csv.gz', compression='gzip')
    diagnoses_icd_df.to_parquet(f'{MIMIC_DB_LOCATION}/diagnoses_icd.parquet')
else:
    diagnoses_icd_df = pd.read_parquet(f'{MIMIC_DB_LOCATION}/diagnoses_icd.parquet')

# Load the diagnoses details data
if not os.path.exists(f'{MIMIC_DB_LOCATION}/d_icd_diagnoses.parquet'):
    d_icd_diagnoses_df = pd.read_csv(f'{MIMIC_DB_LOCATION}/d_icd_diagnoses.csv.gz', compression='gzip')
    d_icd_diagnoses_df.to_parquet(f'{MIMIC_DB_LOCATION}/d_icd_diagnoses.parquet')
else:
    d_icd_diagnoses_df = pd.read_parquet(f'{MIMIC_DB_LOCATION}/d_icd_diagnoses.parquet')

# Load the lab events data
if not os.path.exists(f'{MIMIC_DB_LOCATION}/labevents.parquet'):
    labevents_df = pd.read_csv(f'{MIMIC_DB_LOCATION}/labevents.csv.gz', compression='gzip')
    labevents_df.to_parquet(f'{MIMIC_DB_LOCATION}/labevents.parquet')
else:
    labevents_df = pd.read_parquet(f'{MIMIC_DB_LOCATION}/labevents.parquet')

# # Load the lab items data
if not os.path.exists(f'{MIMIC_DB_LOCATION}/d_labitems.parquet'):
    d_labitems_df = pd.read_csv(f'{MIMIC_DB_LOCATION}/d_labitems.csv.gz', compression='gzip')
    d_labitems_df.to_parquet(f'{MIMIC_DB_LOCATION}/d_labitems.parquet')
else:
    d_labitems_df = pd.read_parquet(f'{MIMIC_DB_LOCATION}/d_labitems.parquet')
```

CLEANING AND PREPROCESSING THE DATA

- Converting dates to datetime objects
- Removing implausible or erroneous records
- Merging data tables on relevant keys



```python
# Drop missing patient data
patients_df.dropna(subset=['gender', 'anchor_age'], inplace=True)

# Drop missing admission data
admissions_df.dropna(subset=['subject_id', 'admittime', 'dischtime'], inplace=True)

# Convert admission and discharge times to datetime
admissions_df.admittime = pd.to_datetime(admissions_df.admittime)
admissions_df.dischtime = pd.to_datetime(admissions_df.dischtime)

# Remove admissions where admission time is after discharge time
admissions_df = admissions_df[admissions_df.admittime < admissions_df.dischtime]

# Convert the charttime to datetime
labevents_df["charttime"] = pd.to_datetime(labevents_df["charttime"])

# Drop any rows where hadm_id, valuenum is missing
labevents_df = labevents_df.dropna(subset=['hadm_id', 'valuenum'])

# Filter out any lab events that are not within the admission time
labevents_df = labevents_df.merge(admissions_df[['hadm_id', 'admittime', 'dischtime']], on='hadm_id')
labevents_df = labevents_df[(labevents_df.charttime >= labevents_df.admittime) &
                            (labevents_df.charttime <= labevents_df.dischtime)]

# Clean up the lab items data
d_labitems_df = d_labitems_df.dropna(subset=['itemid', 'label'])
```

KEY LAB EVENTS FOR AKI PREDICTION

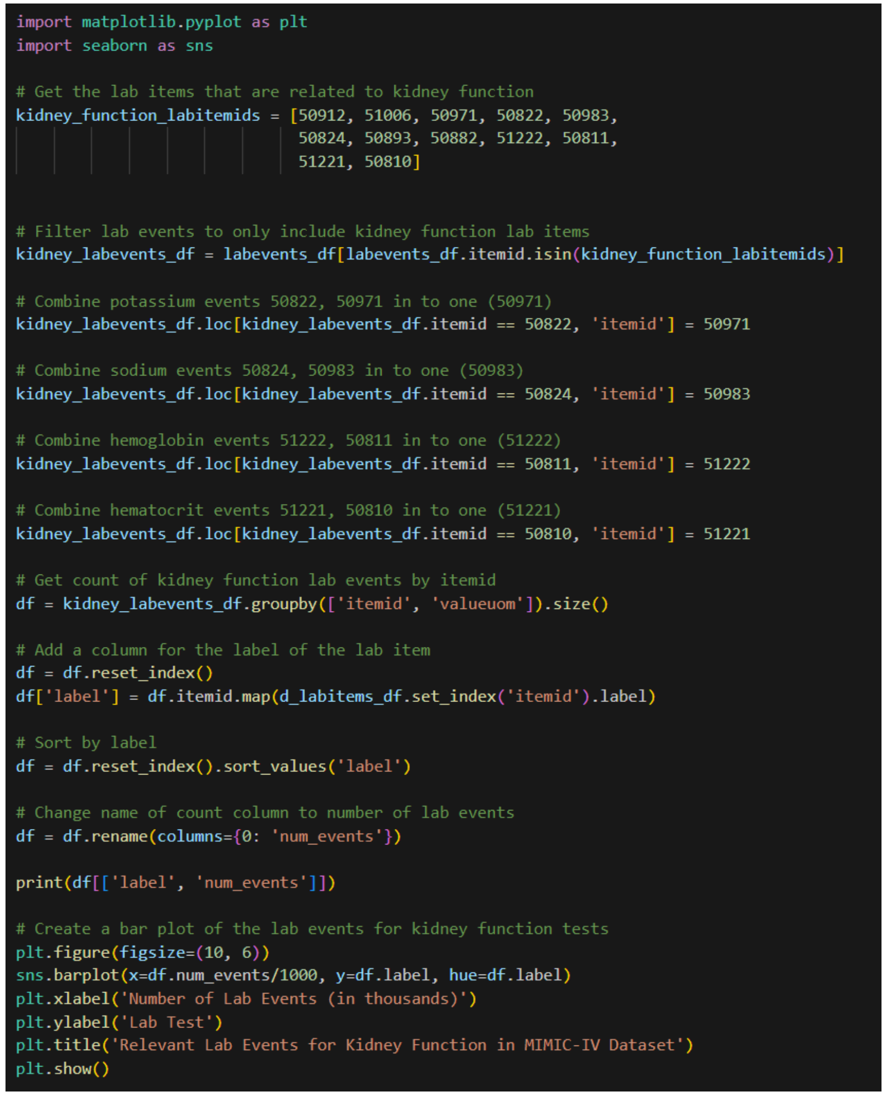
1. **Serum Creatinine**: This is the most critical marker for kidney function, and changes in serum creatinine levels are a primary indicator used to diagnose acute kidney failure. Increases from baseline levels are indicative of decreased kidney function.

2. **Blood Urea Nitrogen (BUN)**: Elevated BUN levels in conjunction with high creatinine levels can indicate impaired kidney function. BUN alone isn't as reliable as when interpreted with creatinine levels.

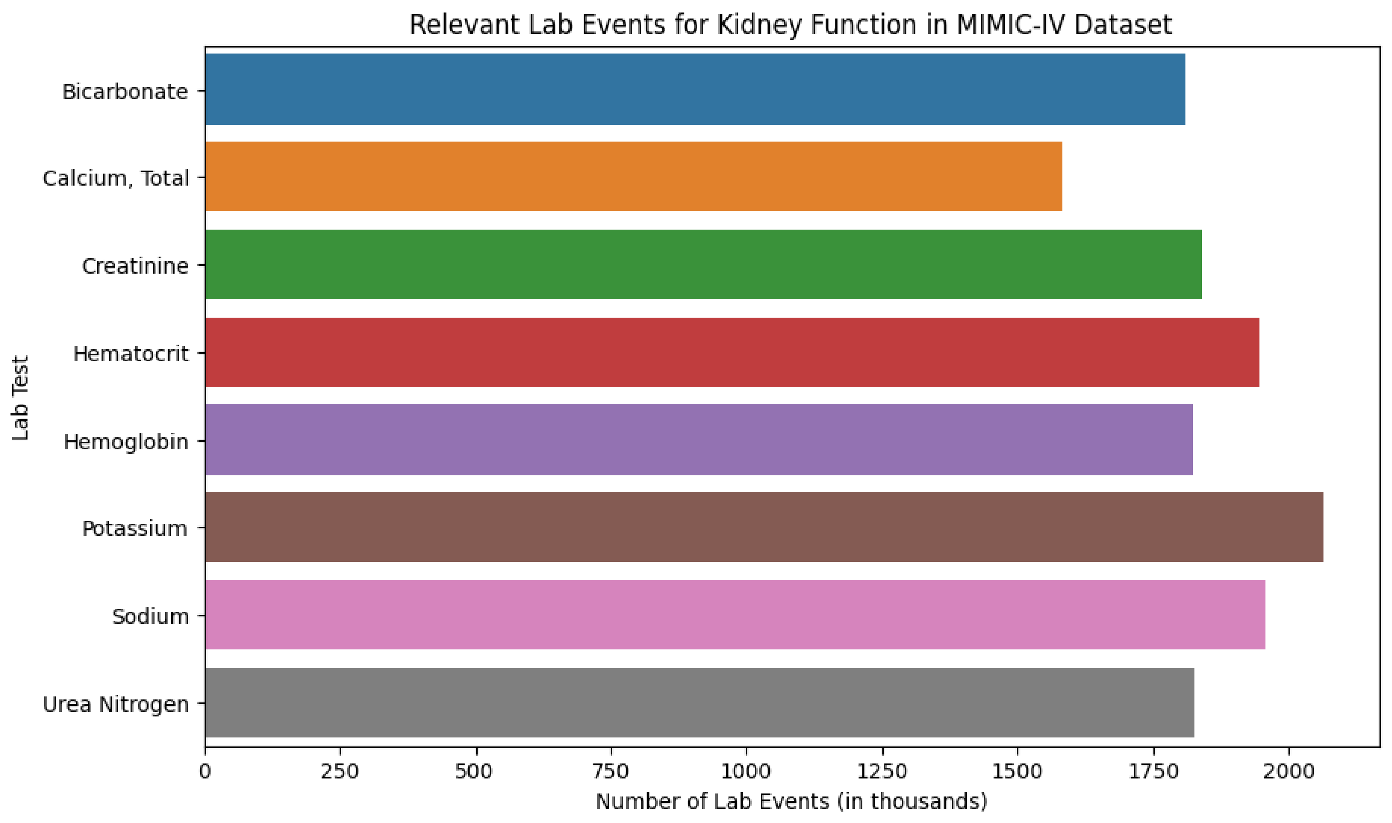
3. **Electrolytes**: Imbalances in electrolytes, such as potassium (K+), sodium (Na+), calcium (Ca2+), and bicarbonate (HCO3-), can be indicative of kidney dysfunction. Potassium levels are particularly worth monitoring for hyperkalemia, which is associated with AKI.

4. **Hemoglobin and Hematocrit**: These can indirectly indicate kidney issues; for example, a decrease in these values could suggest anemia related to chronic kidney disease (CKD) or acute blood loss contributing to AKI.

IDENTIFYING KEY LAB EVENTS FOR KIDNEY FUNCTION

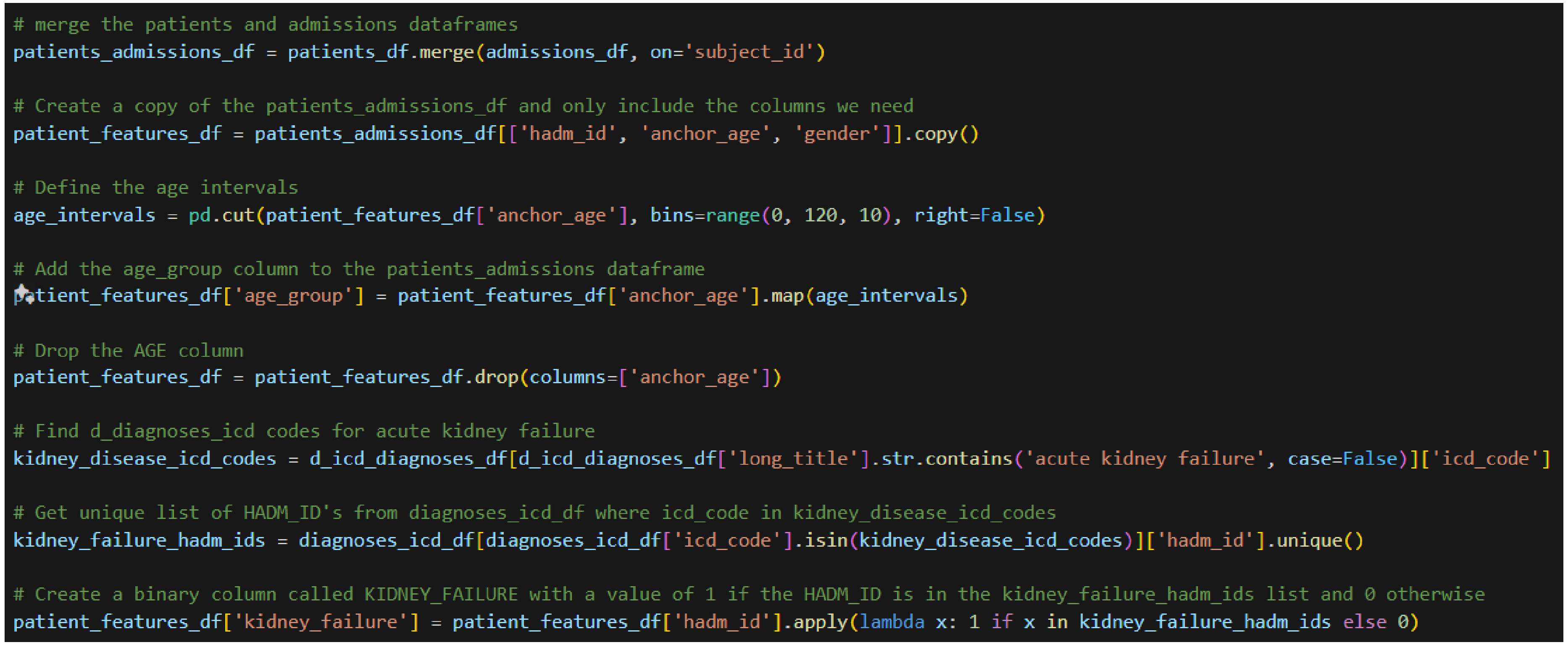


```python
import matplotlib.pyplot as plt
import seaborn as sns

# Get the lab items that are related to kidney function
kidney_function_labitemids = [50912, 51006, 50971, 50822, 50983,
                              50824, 50893, 50882, 51222, 50811,
                              51221, 50810]


# Filter lab events to only include kidney function lab items
kidney_labevents_df = labevents_df[labevents_df.itemid.isin(kidney_function_labitemids)]

# Combine potassium events 50822, 50971 in to one (50971)
kidney_labevents_df.loc[kidney_labevents_df.itemid == 50822, 'itemid'] = 50971

# Combine sodium events 50824, 50983 in to one (50983)
kidney_labevents_df.loc[kidney_labevents_df.itemid == 50824, 'itemid'] = 50983

# Combine hemoglobin events 51222, 50811 in to one (51222)
kidney_labevents_df.loc[kidney_labevents_df.itemid == 50811, 'itemid'] = 51222

# Combine hematocrit events 51221, 50810 in to one (51221)
kidney_labevents_df.loc[kidney_labevents_df.itemid == 50810, 'itemid'] = 51221

# Get count of kidney function lab events by itemid
df = kidney_labevents_df.groupby(['itemid', 'valueuom']).size()

# Add a column for the label of the lab item
df = df.reset_index()
df['label'] = df.itemid.map(d_labitems_df.set_index('itemid').label)

# Sort by label
df = df.reset_index().sort_values('label')

# Change name of count column to number of lab events
df = df.rename(columns={0: 'num_events'})

print(df[['label', 'num_events']])

# Create a bar plot of the lab events for kidney function tests
plt.figure(figsize=(10, 6))
sns.barplot(x=df.num_events/1000, y=df.label, hue=df.label)
plt.xlabel('Number of Lab Events (in thousands)')
plt.ylabel('Lab Test')
plt.title('Relevant Lab Events for Kidney Function in MIMIC-IV Dataset')
plt.show()
```



DEFINING FEATURES FROM PATIENT DATA

- Selecting demographic features: Age, Gender, etc.
- Categorizing ages into meaningful groups
- Creating binary outcomes for acute kidney failure



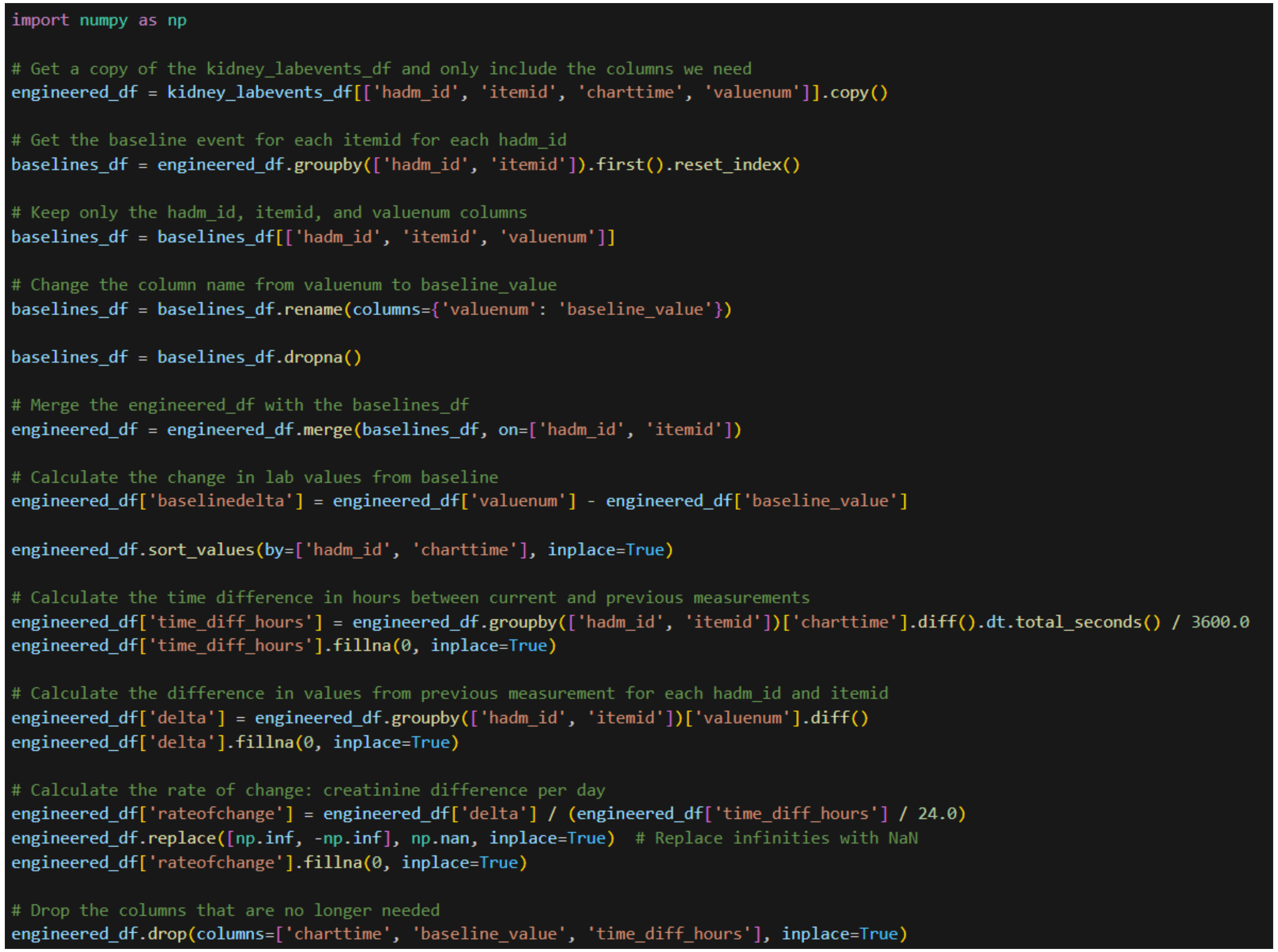
```python
# merge the patients and admissions dataframes
patients_admissions_df = patients_df.merge(admissions_df, on='subject_id')

# Create a copy of the patients_admissions_df and only include the columns we need
patient_features_df = patients_admissions_df[['hadm_id', 'anchor_age', 'gender']].copy()

# Define the age intervals
age_intervals = pd.cut(patient_features_df['anchor_age'], bins=range(0, 120, 10), right=False)

# Add the age_group column to the patients_admissions dataframe
patient_features_df['age_group'] = patient_features_df['anchor_age'].map(age_intervals)

# Drop the AGE column
patient_features_df = patient_features_df.drop(columns=['anchor_age'])

# Find d_diagnoses_icd codes for acute kidney failure
kidney_disease_icd_codes = d_icd_diagnoses_df[d_icd_diagnoses_df['long_title'].str.contains('acute kidney failure', case=False)]['icd_code']

# Get unique list of HADM_ID's from diagnoses_icd_df where icd_code in kidney_disease_icd_codes
kidney_failure_hadm_ids = diagnoses_icd_df[diagnoses_icd_df['icd_code'].isin(kidney_disease_icd_codes)]['hadm_id'].unique()

# Create a binary column called KIDNEY_FAILURE with a value of 1 if the HADM_ID is in the kidney_failure_hadm_ids list and 0 otherwise
patient_features_df['kidney_failure'] = patient_features_df['hadm_id'].apply(lambda x: 1 if x in kidney_failure_hadm_ids else 0)
```

FEATURE ENGINEERING FOR AKI PREDICTION

- **Baseline Measurement**: Establish a baseline for key lab events. This could be the value recorded at the time of admission or the lowest value within a window before AKI diagnosis.
- **Delta Features**: Calculate the change (∆) from baseline for key lab values. This captures the trajectory of kidney function.
- **Rate of Change**: For key lab values, calculating the rate of change could help identify rapid deteriorations in kidney function.
- **Aggregated Features**: Use statistical measures (mean, median, max, min, standard deviation) of lab events over specific time windows to capture trends and variability in kidney function.

ENGINEERING FEATURES BASED CHANGES IN LAB EVENT VALUES

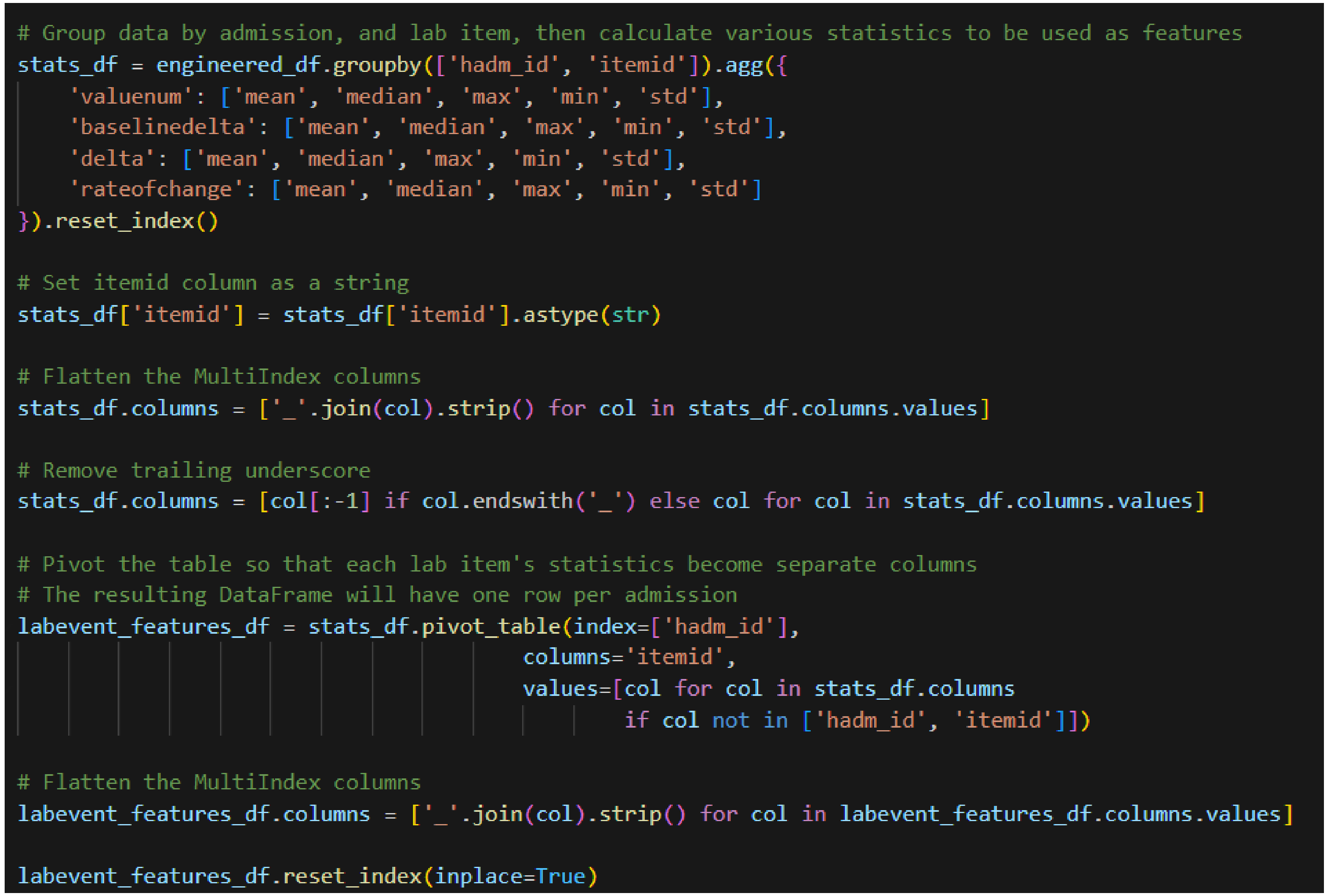
- Change in value relative to baseline
- Change in value since last measurement
- Rate of Change / Day



```python
import numpy as np

# Get a copy of the kidney_labevents_df and only include the columns we need
engineered_df = kidney_labevents_df[['hadm_id', 'itemid', 'charttime', 'valuenum']].copy()

# Get the baseline event for each itemid for each hadm_id
baselines_df = engineered_df.groupby(['hadm_id', 'itemid']).first().reset_index()

# Keep only the hadm_id, itemid, and valuenum columns
baselines_df = baselines_df[['hadm_id', 'itemid', 'valuenum']]

# Change the column name from valuenum to baseline_value
baselines_df = baselines_df.rename(columns={'valuenum': 'baseline_value'})

baselines_df = baselines_df.dropna()

# Merge the engineered_df with the baselines_df
engineered_df = engineered_df.merge(baselines_df, on=['hadm_id', 'itemid'])

# Calculate the change in lab values from baseline
engineered_df['baselinedelta'] = engineered_df['valuenum'] - engineered_df['baseline_value']

engineered_df.sort_values(by=['hadm_id', 'charttime'], inplace=True)

# Calculate the time difference in hours between current and previous measurements
engineered_df['time_diff_hours'] = engineered_df.groupby(['hadm_id', 'itemid'])['charttime'].diff().dt.total_seconds() / 3600.0
engineered_df['time_diff_hours'].fillna(0, inplace=True)

# Calculate the difference in values from previous measurement for each hadm_id and itemid
engineered_df['delta'] = engineered_df.groupby(['hadm_id', 'itemid'])['valuenum'].diff()
engineered_df['delta'].fillna(0, inplace=True)

# Calculate the rate of change: creatinine difference per day
engineered_df['rateofchange'] = engineered_df['delta'] / (engineered_df['time_diff_hours'] / 24.0)
engineered_df.replace([np.inf, -np.inf], np.nan, inplace=True)  # Replace infinities with NaN
engineered_df['rateofchange'].fillna(0, inplace=True)

# Drop the columns that are no longer needed
engineered_df.drop(columns=['charttime', 'baseline_value', 'time_diff_hours'], inplace=True)
```

ENGINEERING FEATURES BASED STATISTICAL ANALYSIS

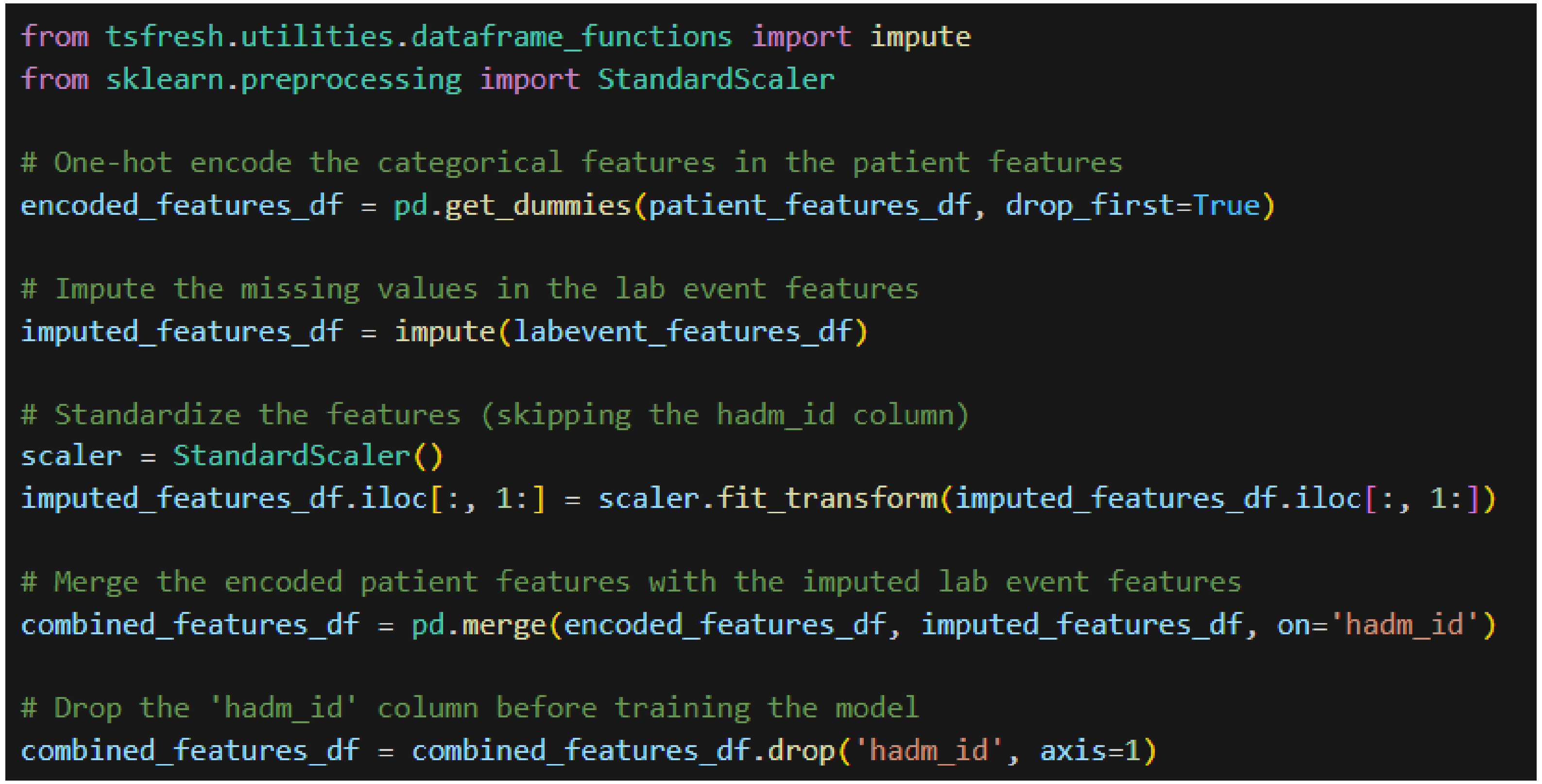
- Mean
- Median
- Max
- Min
- Standard Deviation



```python
# Group data by admission, and lab item, then calculate various statistics to be used as features
stats_df = engineered_df.groupby(['hadm_id', 'itemid']).agg({
    'valuenum': ['mean', 'median', 'max', 'min', 'std'],
    'baselinedelta': ['mean', 'median', 'max', 'min', 'std'],
    'delta': ['mean', 'median', 'max', 'min', 'std'],
    'rateofchange': ['mean', 'median', 'max', 'min', 'std']
}).reset_index()

# Set itemid column as a string
stats_df['itemid'] = stats_df['itemid'].astype(str)

# Flatten the MultiIndex columns
stats_df.columns = ['_'.join(col).strip() for col in stats_df.columns.values]

# Remove trailing underscore
stats_df.columns = [col[:-1] if col.endswith('_') else col for col in stats_df.columns.values]

# Pivot the table so that each lab item's statistics become separate columns
# The resulting DataFrame will have one row per admission
labevent_features_df = stats_df.pivot_table(index=['hadm_id'],
                                        columns='itemid',
                                        values=[col for col in stats_df.columns
                                                if col not in ['hadm_id', 'itemid']])

# Flatten the MultiIndex columns
labevent_features_df.columns = ['_'.join(col).strip() for col in labevent_features_df.columns.values]

labevent_features_df.reset_index(inplace=True)
```

NORMALIZING AND ENCODING FEATURES

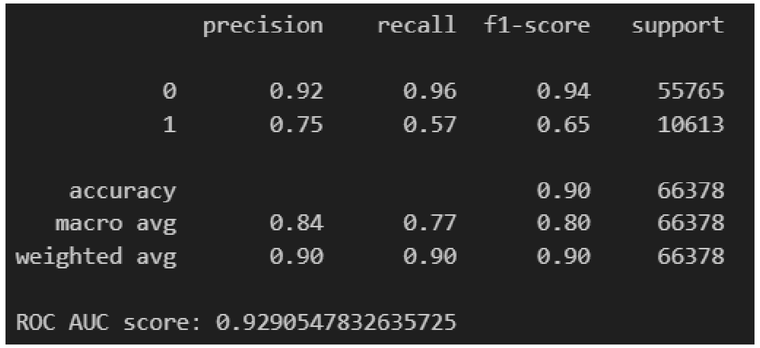
- Normalizing lab event statistics for model input
- One-hot encoding categorical variables
- Imputing missing values and handling outliers



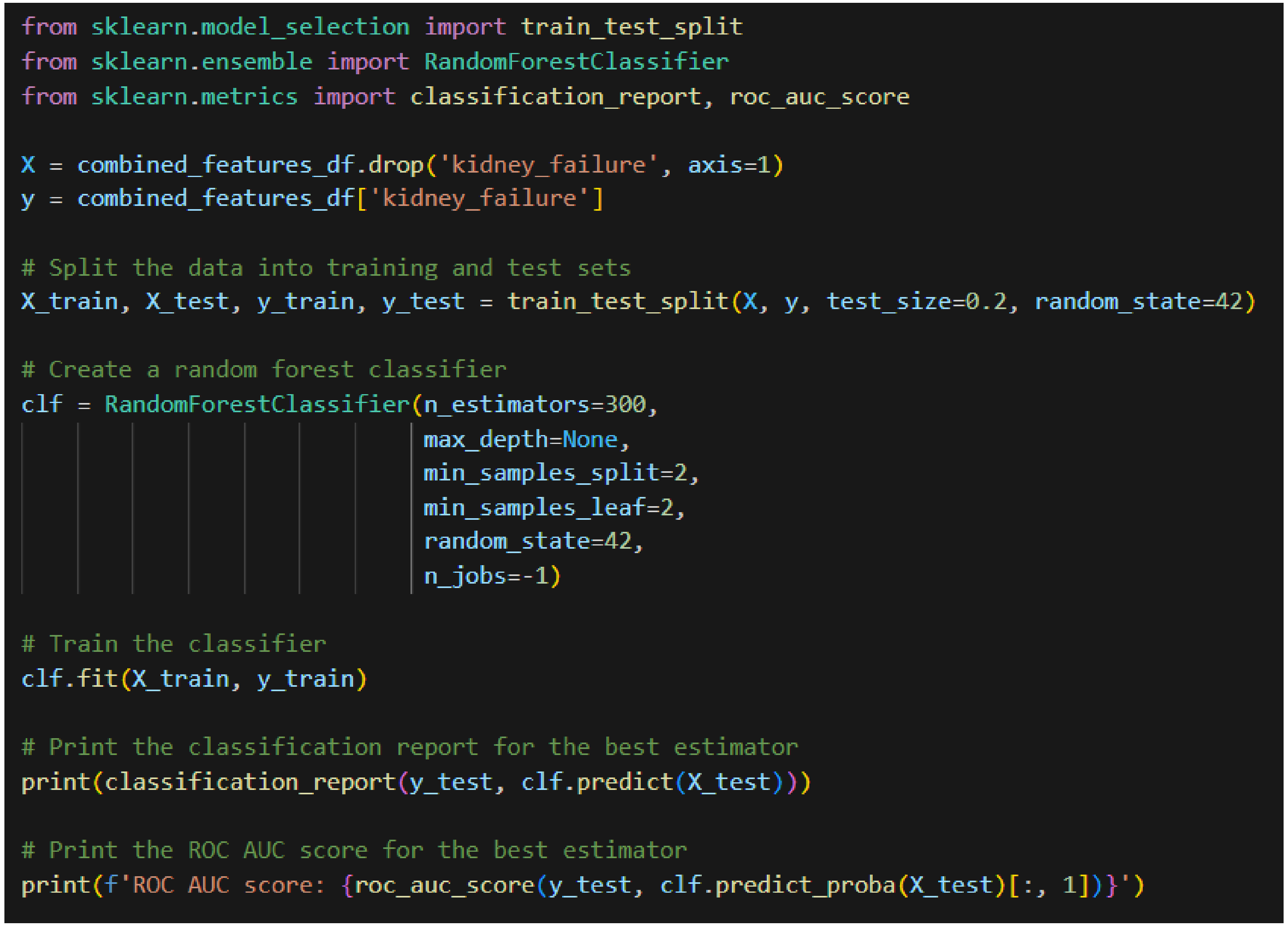
```python
from tsfresh.utilities.dataframe_functions import impute
from sklearn.preprocessing import StandardScaler

# One-hot encode the categorical features in the patient features
encoded_features_df = pd.get_dummies(patient_features_df, drop_first=True)

# Impute the missing values in the lab event features
imputed_features_df = impute(labevent_features_df)

# Standardize the features (skipping the hadm_id column)
scaler = StandardScaler()
imputed_features_df.iloc[:, 1:] = scaler.fit_transform(imputed_features_df.iloc[:, 1:])

# Merge the encoded patient features with the imputed lab event features
combined_features_df = pd.merge(encoded_features_df, imputed_features_df, on='hadm_id')

# Drop the 'hadm_id' column before training the model
combined_features_df = combined_features_df.drop('hadm_id', axis=1)
```

MODEL TRAINING AND EVALUATION

- Data Preparation
- Random Forest Classifier Initialization
- Model Evaluation
  - Classification Report
  - ROC-AUC

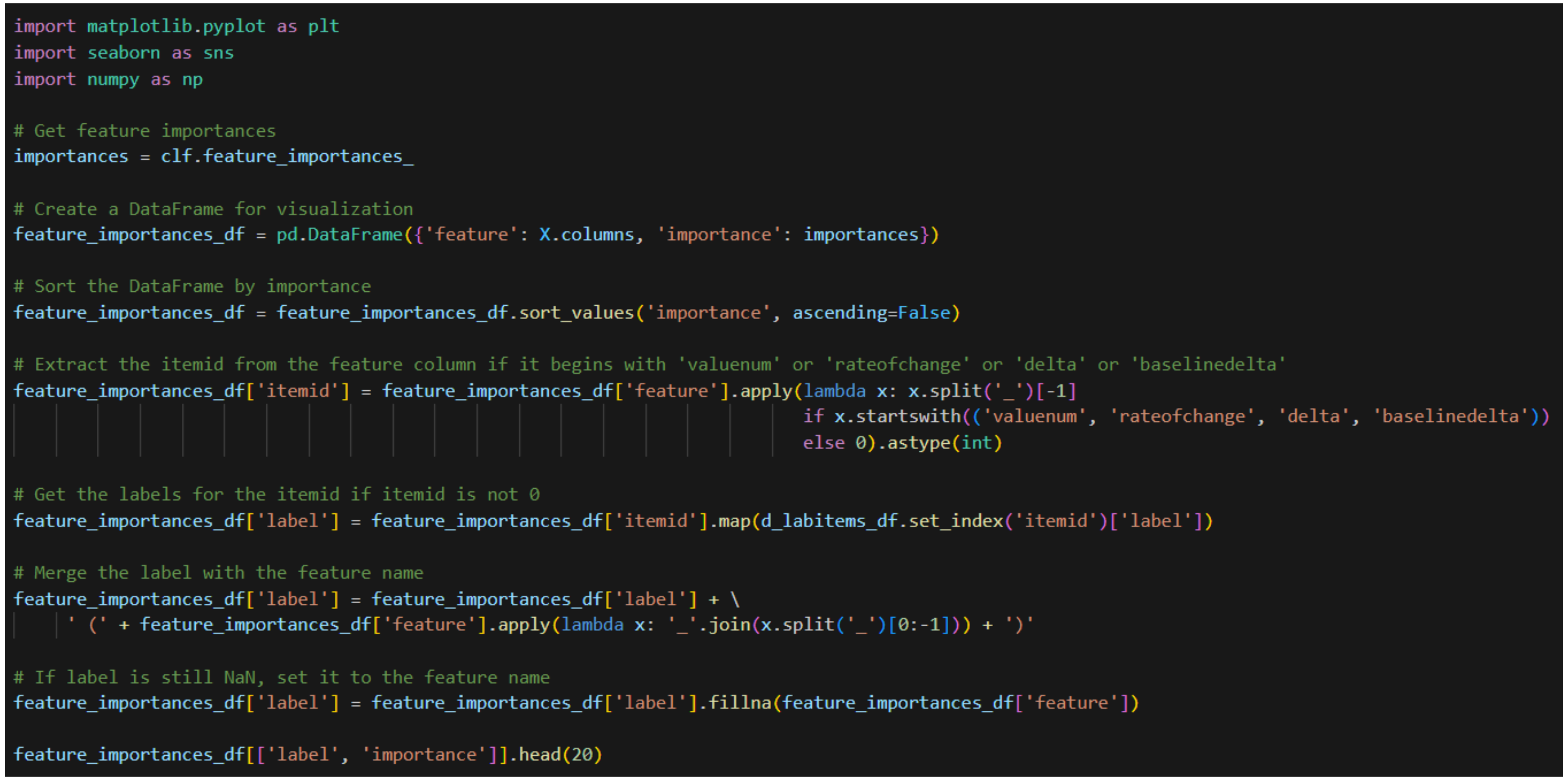


```
              precision    recall  f1-score   support

           0       0.92      0.96      0.94     55765
           1       0.75      0.57      0.65     10613

    accuracy                           0.90     66378
   macro avg       0.84      0.77      0.80     66378
weighted avg       0.90      0.90      0.90     66378

ROC AUC score: 0.9290547832635725
```

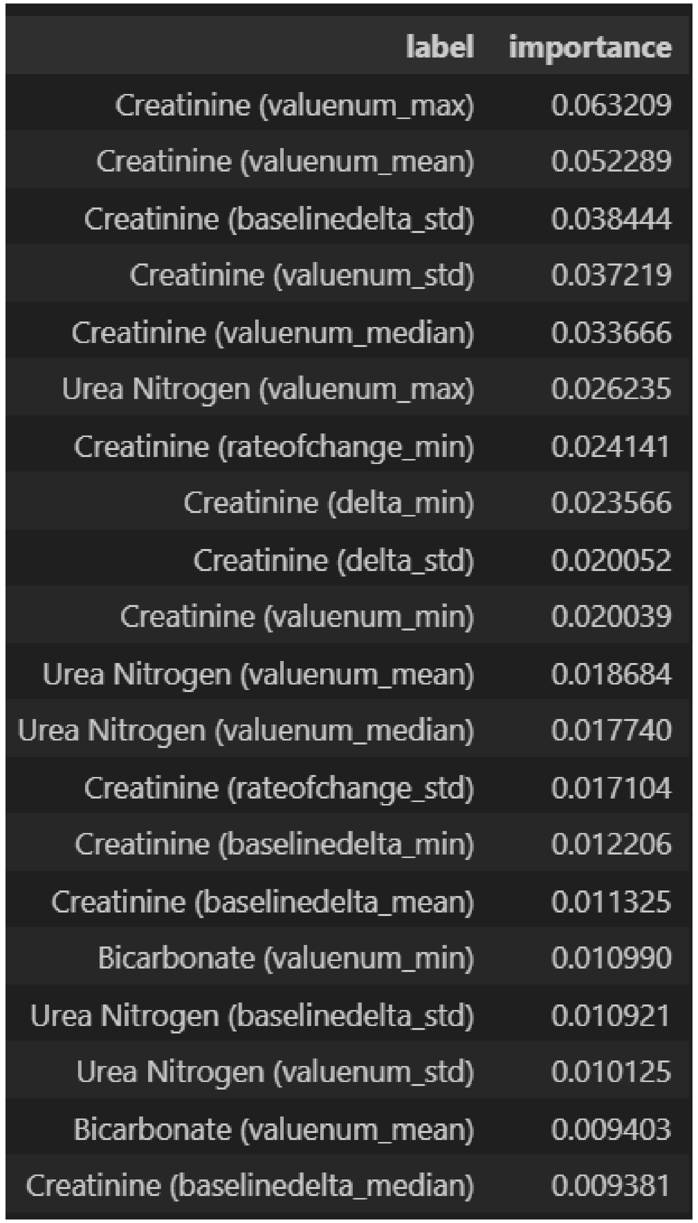


```python
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, roc_auc_score

X = combined_features_df.drop('kidney_failure', axis=1)
y = combined_features_df['kidney_failure']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a random forest classifier
clf = RandomForestClassifier(n_estimators=300,
                             max_depth=None,
                             min_samples_split=2,
                             min_samples_leaf=2,
                             random_state=42,
                             n_jobs=-1)

# Train the classifier
clf.fit(X_train, y_train)

# Print the classification report for the best estimator
print(classification_report(y_test, clf.predict(X_test)))

# Print the ROC AUC score for the best estimator
print(f'ROC AUC score: {roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])}')
```

ANALYZING FEATURE IMPORTANCE

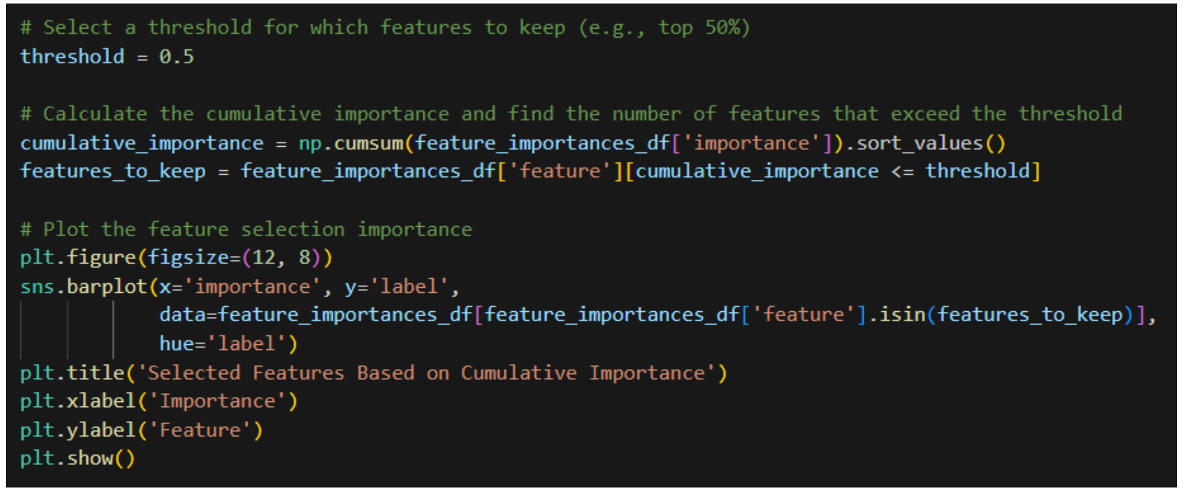


```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Get feature importances
importances = clf.feature_importances_

# Create a DataFrame for visualization
feature_importances_df = pd.DataFrame({'feature': X.columns, 'importance': importances})

# Sort the DataFrame by importance
feature_importances_df = feature_importances_df.sort_values('importance', ascending=False)

# Extract the itemid from the feature column if it begins with 'valuenum' or 'rateofchange' or 'delta' or 'baselinedelta'
feature_importances_df['itemid'] = feature_importances_df['feature'].apply(lambda x: x.split('_')[-1]
                                                                            if x.startswith(('valuenum', 'rateofchange', 'delta', 'baselinedelta'))
                                                                            else 0).astype(int)

# Get the labels for the itemid if itemid is not 0
feature_importances_df['label'] = feature_importances_df['itemid'].map(d_labitems_df.set_index('itemid')['label'])

# Merge the label with the feature name
feature_importances_df['label'] = feature_importances_df['label'] + \
        ' (' + feature_importances_df['feature'].apply(lambda x: '_'.join(x.split('_')[0:-1])) + ')'

# If label is still NaN, set it to the feature name
feature_importances_df['label'] = feature_importances_df['label'].fillna(feature_importances_df['feature'])

feature_importances_df[['label', 'importance']].head(20)
```

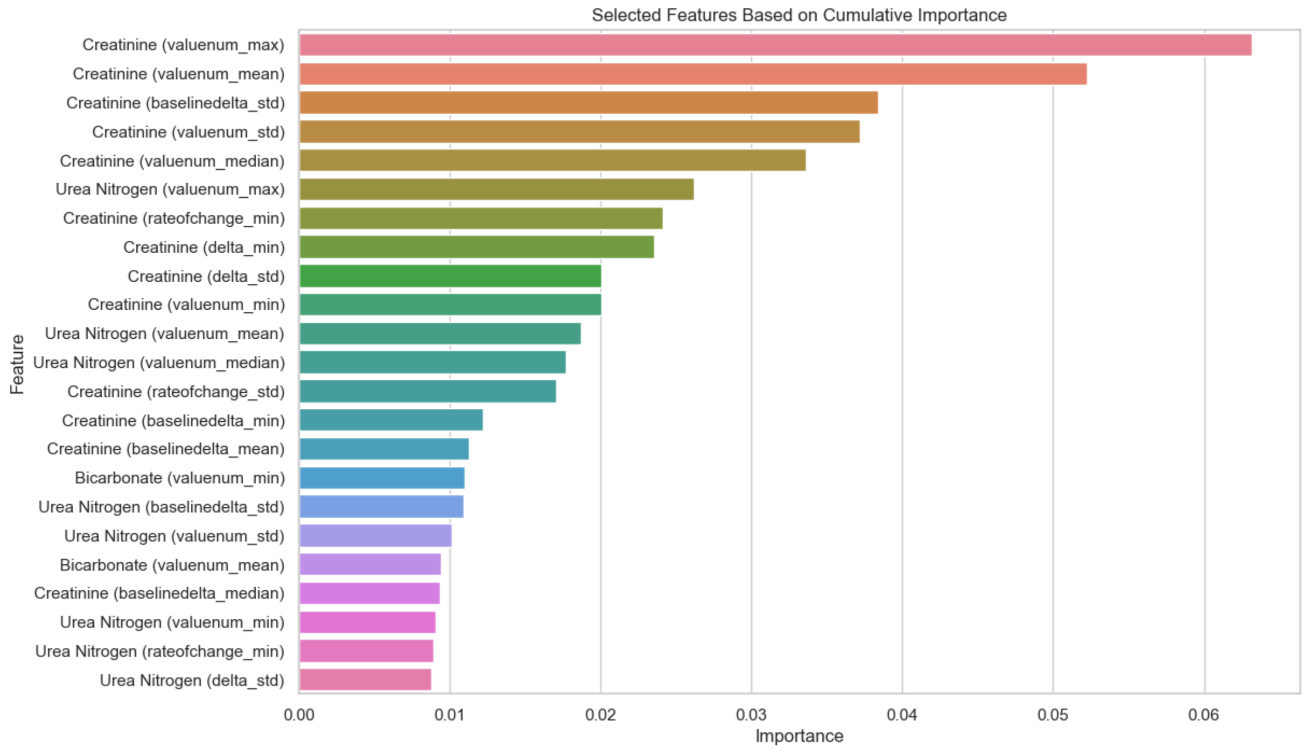


label	importance
Creatinine (valuenum_max)	0.063209
Creatinine (valuenum_mean)	0.052289
Creatinine (baselinedelta_std)	0.038444
Creatinine (valuenum_std)	0.037219
Creatinine (valuenum_median)	0.033666
Urea Nitrogen (valuenum_max)	0.026235
Creatinine (rateofchange_min)	0.024141
Creatinine (delta_min)	0.023566
Creatinine (delta_std)	0.020052
Creatinine (valuenum_min)	0.020039
Urea Nitrogen (valuenum_mean)	0.018684
Urea Nitrogen (valuenum_median)	0.017740
Creatinine (rateofchange_std)	0.017104
Creatinine (baselinedelta_min)	0.012206
Creatinine (baselinedelta_mean)	0.011325
Bicarbonate (valuenum_min)	0.010990
Urea Nitrogen (baselinedelta_std)	0.010921
Urea Nitrogen (valuenum_std)	0.010125
Bicarbonate (valuenum_mean)	0.009403
Creatinine (baselinedelta_median)	0.009381

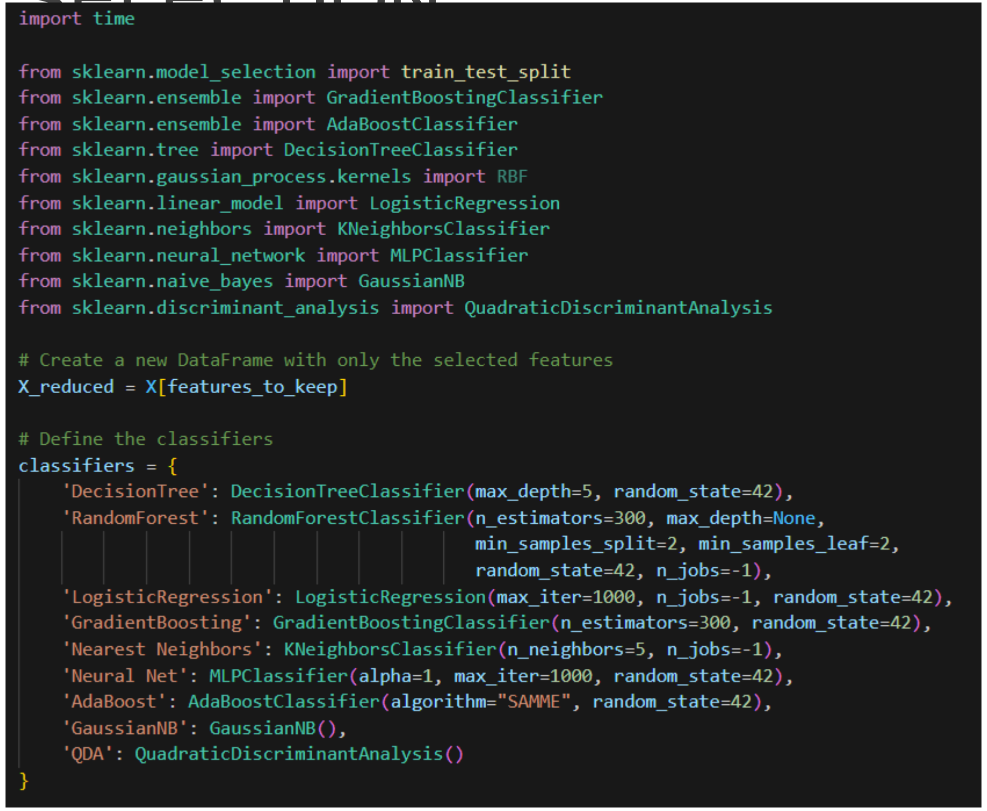
FEATURE SELECTION BASED ON CUMULATIVE IMPORTANCE



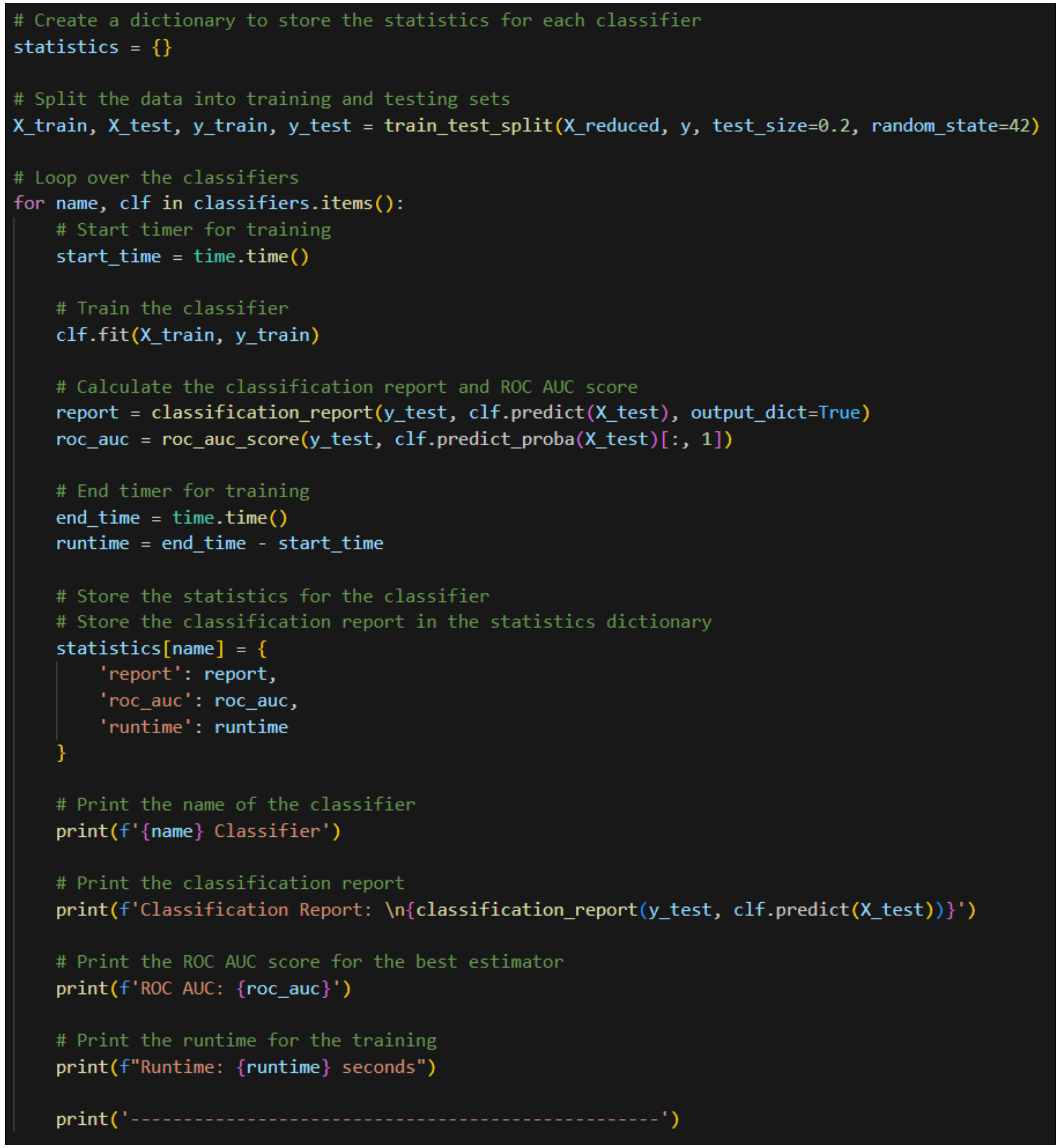
```python
# Select a threshold for which features to keep (e.g., top 50%)
threshold = 0.5

# Calculate the cumulative importance and find the number of features that exceed the threshold
cumulative_importance = np.cumsum(feature_importances_df['importance']).sort_values()
features_to_keep = feature_importances_df['feature'][cumulative_importance <= threshold]

# Plot the feature selection importance
plt.figure(figsize=(12, 8))
sns.barplot(x='importance', y='label',
            data=feature_importances_df[feature_importances_df['feature'].isin(features_to_keep)],
            hue='label')
plt.title('Selected Features Based on Cumulative Importance')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()
```



TRAIN MULTIPLE CLASSIFICATION MODELS BASED ON FEATURE SELECTION

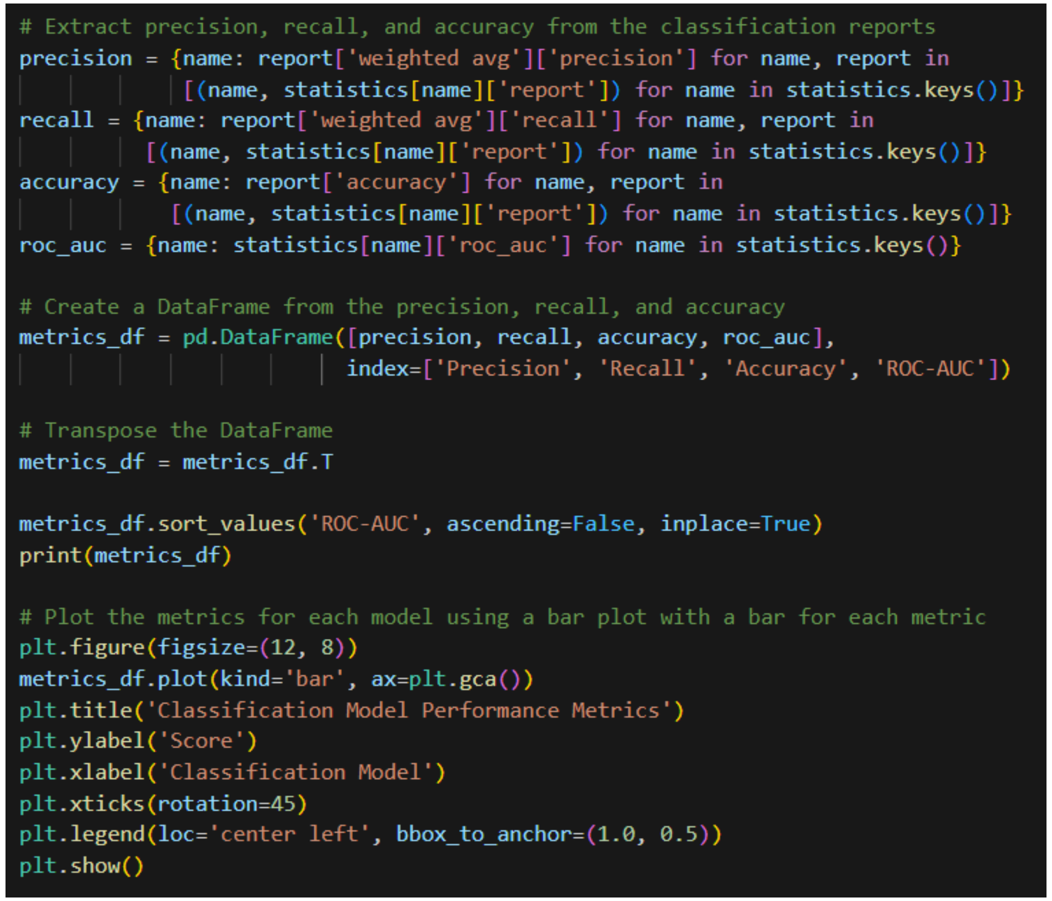


```python
import time

from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

# Create a new DataFrame with only the selected features
X_reduced = X[features_to_keep]

# Define the classifiers
classifiers = {
    'DecisionTree': DecisionTreeClassifier(max_depth=5, random_state=42),
    'RandomForest': RandomForestClassifier(n_estimators=300, max_depth=None,
                                           min_samples_split=2, min_samples_leaf=2,
                                           random_state=42, n_jobs=-1),
    'LogisticRegression': LogisticRegression(max_iter=1000, n_jobs=-1, random_state=42),
    'GradientBoosting': GradientBoostingClassifier(n_estimators=300, random_state=42),
    'Nearest Neighbors': KNeighborsClassifier(n_neighbors=5, n_jobs=-1),
    'Neural Net': MLPClassifier(alpha=1, max_iter=1000, random_state=42),
    'AdaBoost': AdaBoostClassifier(algorithm="SAMME", random_state=42),
    'GaussianNB': GaussianNB(),
    'QDA': QuadraticDiscriminantAnalysis()
}
```

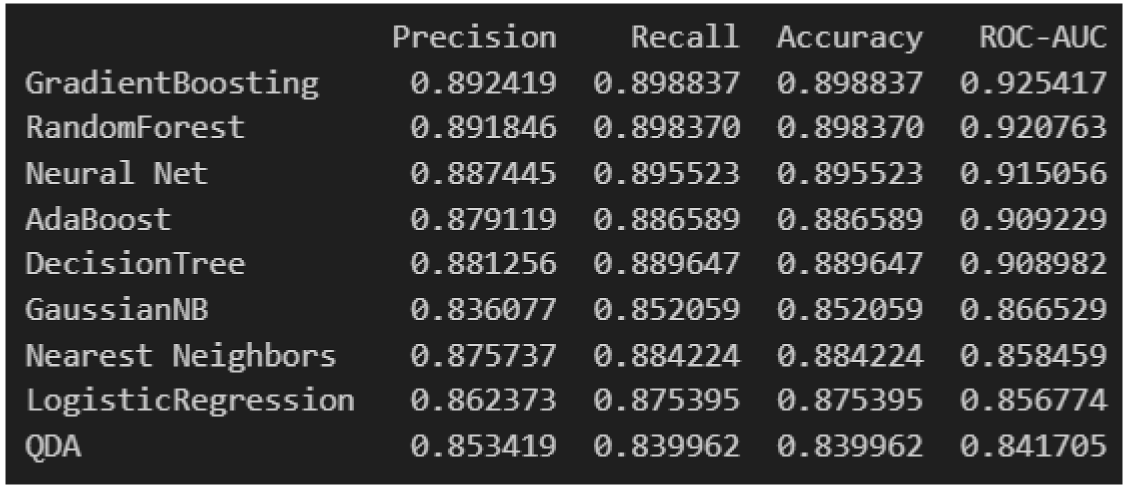


```python
# Create a dictionary to store the statistics for each classifier
statistics = {}

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_reduced, y, test_size=0.2, random_state=42)

# Loop over the classifiers
for name, clf in classifiers.items():
    # Start timer for training
    start_time = time.time()

    # Train the classifier
    clf.fit(X_train, y_train)

    # Calculate the classification report and ROC AUC score
    report = classification_report(y_test, clf.predict(X_test), output_dict=True)
    roc_auc = roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])

    # End timer for training
    end_time = time.time()
    runtime = end_time - start_time

    # Store the statistics for the classifier
    # Store the classification report in the statistics dictionary
    statistics[name] = {
        'report': report,
        'roc_auc': roc_auc,
        'runtime': runtime
    }

    # Print the name of the classifier
    print(f'{name} Classifier')

    # Print the classification report
    print(f'Classification Report: \n{classification_report(y_test, clf.predict(X_test))}')

    # Print the ROC AUC score for the best estimator
    print(f'ROC AUC: {roc_auc}')

    # Print the runtime for the training
    print(f"Runtime: {runtime} seconds")

    print('--------------------------------------------------')
```

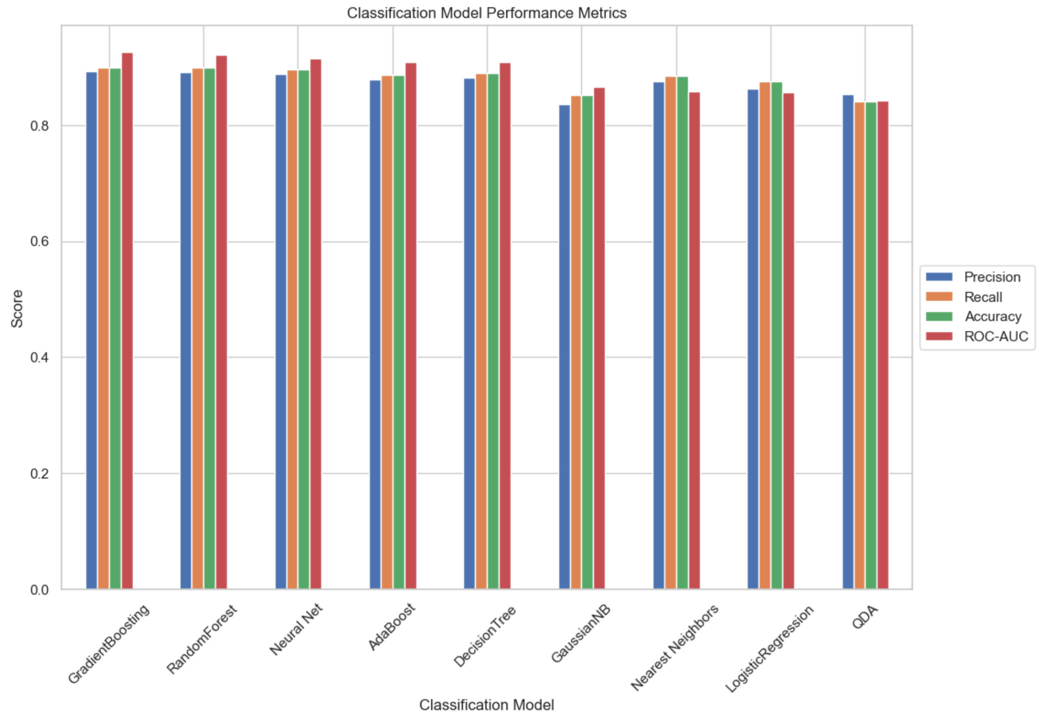
EVALUATE THE PERFORMANCE OF EACH MODEL



```python
# Extract precision, recall, and accuracy from the classification reports
precision = {name: report['weighted avg']['precision'] for name, report in
                [(name, statistics[name]['report']) for name in statistics.keys()]}
recall = {name: report['weighted avg']['recall'] for name, report in
            [(name, statistics[name]['report']) for name in statistics.keys()]}
accuracy = {name: report['accuracy'] for name, report in
              [(name, statistics[name]['report']) for name in statistics.keys()]}
roc_auc = {name: statistics[name]['roc_auc'] for name in statistics.keys()}

# Create a DataFrame from the precision, recall, and accuracy
metrics_df = pd.DataFrame([precision, recall, accuracy, roc_auc],
                          index=['Precision', 'Recall', 'Accuracy', 'ROC-AUC'])

# Transpose the DataFrame
metrics_df = metrics_df.T

metrics_df.sort_values('ROC-AUC', ascending=False, inplace=True)
print(metrics_df)

# Plot the metrics for each model using a bar plot with a bar for each metric
plt.figure(figsize=(12, 8))
metrics_df.plot(kind='bar', ax=plt.gca())
plt.title('Classification Model Performance Metrics')
plt.ylabel('Score')
plt.xlabel('Classification Model')
plt.xticks(rotation=45)
plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
plt.show()
```



```
                    Precision    Recall  Accuracy   ROC-AUC
GradientBoosting     0.892419  0.898837  0.898837  0.925417
RandomForest         0.891846  0.898370  0.898370  0.920763
Neural Net           0.887445  0.895523  0.895523  0.915056
AdaBoost             0.879119  0.886589  0.886589  0.909229
DecisionTree         0.881256  0.889647  0.889647  0.908982
GaussianNB           0.836077  0.852059  0.852059  0.866529
Nearest Neighbors    0.875737  0.884224  0.884224  0.858459
LogisticRegression   0.862373  0.875395  0.875395  0.856774
QDA                  0.853419  0.839962  0.839962  0.841705
```



MODEL PERFORMANCE ANALYSIS

GradientBoostingClassifier

- **ROC AUC**: The GradientBoostingClassifier achieved the highest ROC AUC score of approximately 0.925. This score indicates a strong ability to differentiate between patients who will experience acute kidney failure and those who will not, with a higher score reflecting a model's better performance at all classification thresholds.
- **Precision and Recall**: The model also shows a good balance between precision (0.892) and recall (0.899) for the weighted average, suggesting it is effective at identifying positive cases while maintaining a low false-positive rate. This balance is crucial in medical applications where both missing true cases (low recall) and raising false alarms (low precision) have significant implications.

RandomForestClassifier

- The RandomForestClassifier, with a ROC AUC score of approximately 0.921, also demonstrates strong performance, though slightly below the GradientBoostingClassifier. Its precision and recall metrics indicate a similar effectiveness in identifying acute kidney failure cases.

Neural Net

- The Neural Net (MLPClassifier) shows competitive performance with a ROC AUC score of about 0.915. This model type, given its complexity, can capture complex nonlinear relationships in the data but might be more challenging to interpret than tree-based models.

INTERPRETABILITY OF MODEL RESULTS

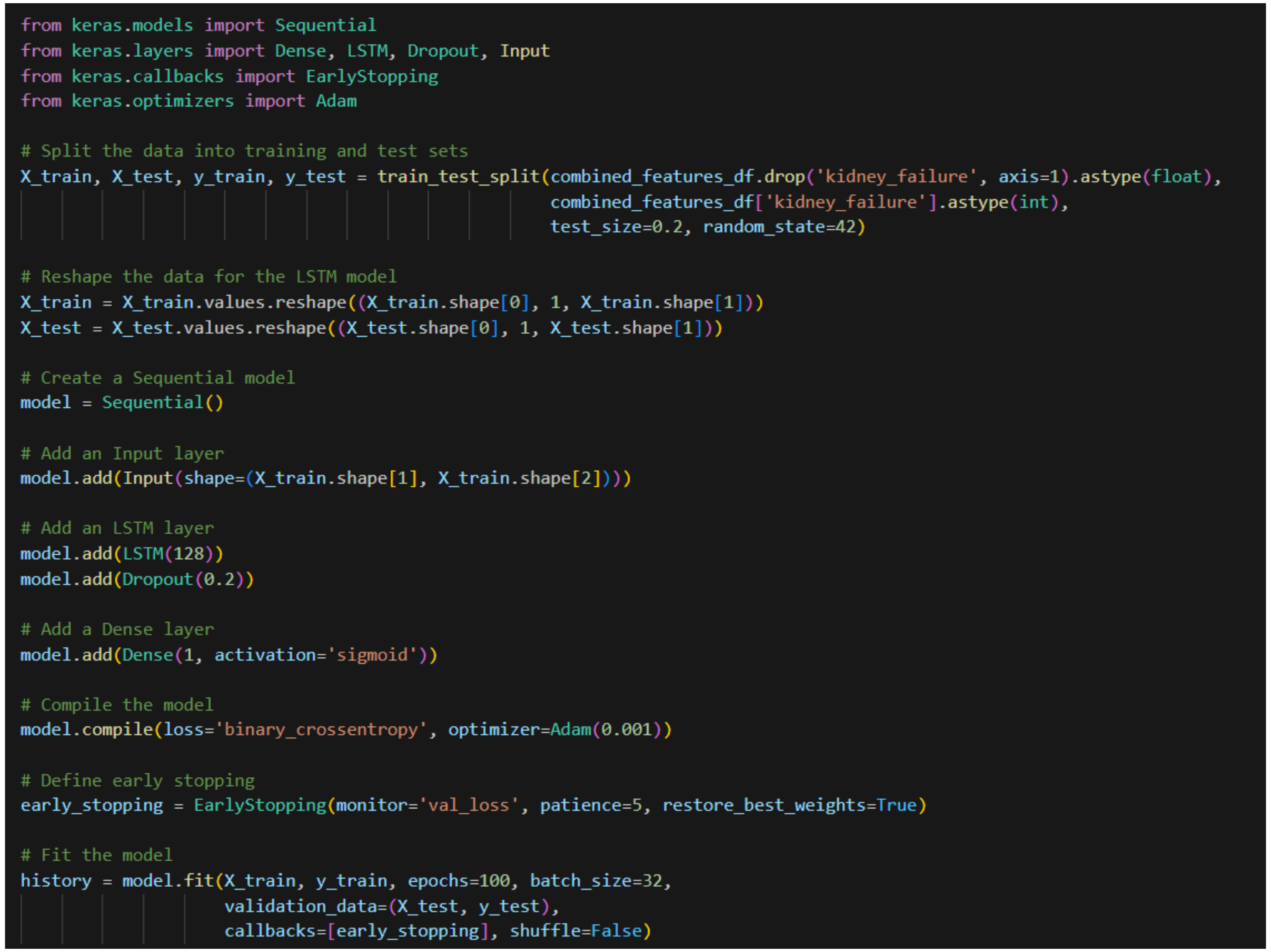
Feature Importances

- The analysis of feature importances reveals that creatinine-related features (e.g., maximum, mean, standard deviation of baselinedelta) are among the most predictive of acute kidney failure. This aligns well with medical knowledge, as changes in serum creatinine levels are a key indicator of kidney function.
- The prominence of urea nitrogen and bicarbonate in the feature importances further validates the models, as these are also clinically relevant markers of kidney function and acid-base balance.

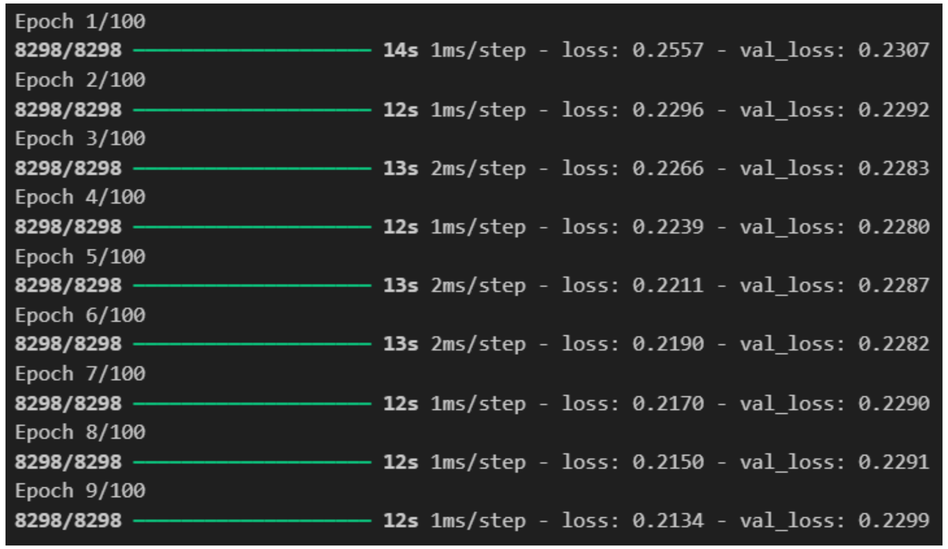
Clinical Implications

- The models identify features that are clinically meaningful, enhancing their utility and trustworthiness in a clinical setting. Clinicians can use these insights to focus on key lab markers when assessing a patient's risk of acute kidney failure.
- For instance, a significant change in the 'Creatinine (valuenum_max)' feature would likely alert a clinician to a potential decline in kidney function, prompting further investigation or intervention.

BONUS: TRAINING A LONG SHORT-TERM MEMORY MODEL BASED ON THE ENGINEERED AKI FEATURES AND LABEL

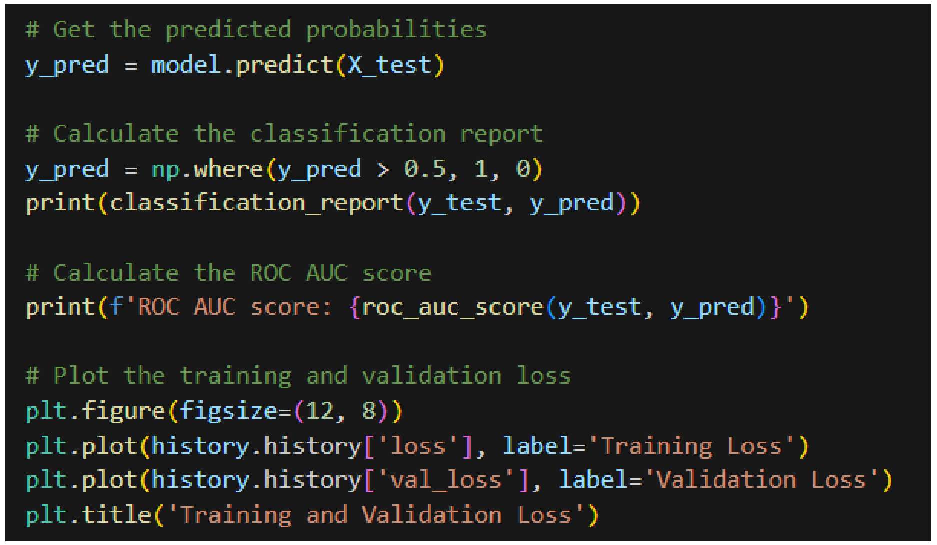


```python
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout, Input
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(combined_features_df.drop('kidney_failure', axis=1).astype(float),
                                                    combined_features_df['kidney_failure'].astype(int),
                                                    test_size=0.2, random_state=42)

# Reshape the data for the LSTM model
X_train = X_train.values.reshape((X_train.shape[0], 1, X_train.shape[1]))
X_test = X_test.values.reshape((X_test.shape[0], 1, X_test.shape[1]))

# Create a Sequential model
model = Sequential()

# Add an Input layer
model.add(Input(shape=(X_train.shape[1], X_train.shape[2])))

# Add an LSTM layer
model.add(LSTM(128))
model.add(Dropout(0.2))

# Add a Dense layer
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer=Adam(0.001))

# Define early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Fit the model
history = model.fit(X_train, y_train, epochs=100, batch_size=32,
                    validation_data=(X_test, y_test),
                    callbacks=[early_stopping], shuffle=False)
```

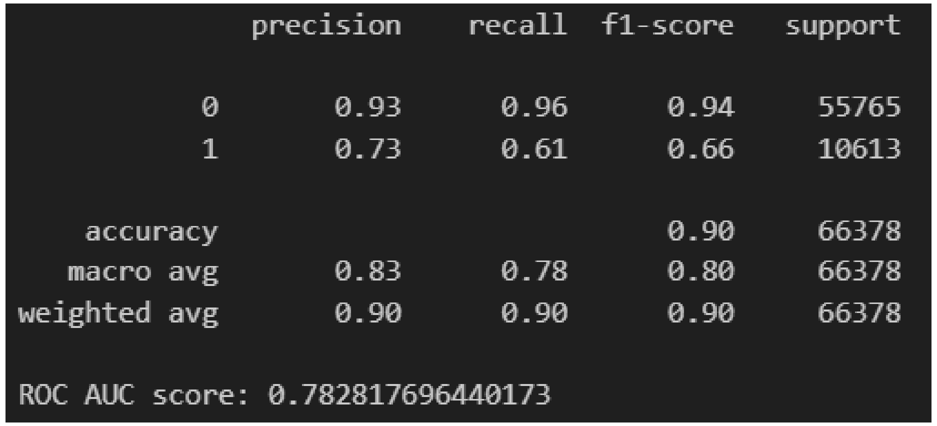


```
Epoch 1/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step - loss: 0.2557 - val_loss: 0.2307
Epoch 2/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 12s 1ms/step - loss: 0.2296 - val_loss: 0.2292
Epoch 3/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 13s 2ms/step - loss: 0.2266 - val_loss: 0.2283
Epoch 4/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 12s 1ms/step - loss: 0.2239 - val_loss: 0.2280
Epoch 5/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 13s 2ms/step - loss: 0.2211 - val_loss: 0.2287
Epoch 6/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 13s 2ms/step - loss: 0.2190 - val_loss: 0.2282
Epoch 7/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 12s 1ms/step - loss: 0.2170 - val_loss: 0.2290
Epoch 8/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 12s 1ms/step - loss: 0.2150 - val_loss: 0.2291
Epoch 9/100
8298/8298 ━━━━━━━━━━━━━━━━━━━━ 12s 1ms/step - loss: 0.2134 - val_loss: 0.2299
```

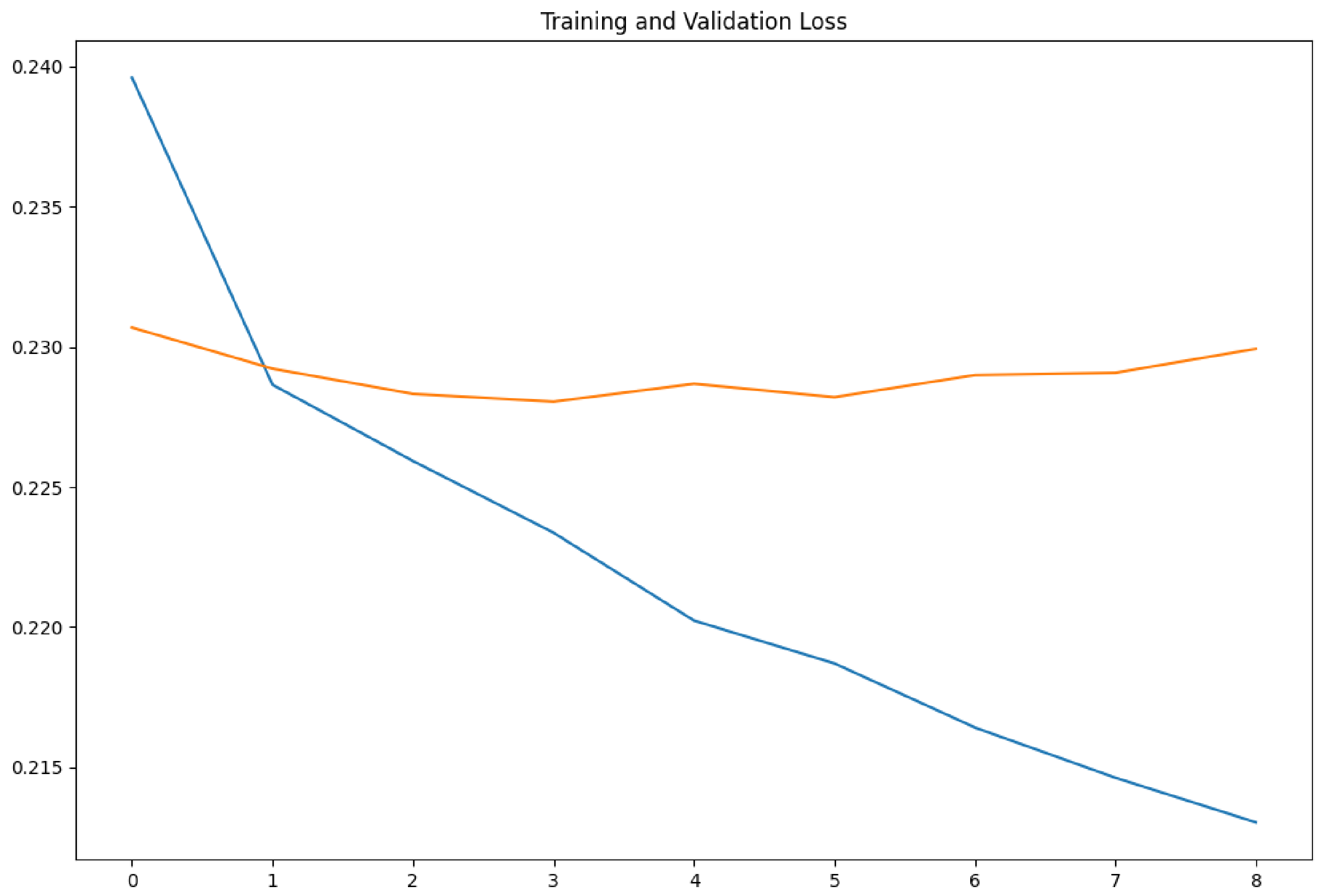
LTSM MODEL RESULTS



```
# Get the predicted probabilities
y_pred = model.predict(X_test)

# Calculate the classification report
y_pred = np.where(y_pred > 0.5, 1, 0)
print(classification_report(y_test, y_pred))

# Calculate the ROC AUC score
print(f'ROC AUC score: {roc_auc_score(y_test, y_pred)}')

# Plot the training and validation loss
plt.figure(figsize=(12, 8))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
```



```
              precision    recall  f1-score   support

           0       0.93      0.96      0.94     55765
           1       0.73      0.61      0.66     10613

    accuracy                           0.90     66378
   macro avg       0.83      0.78      0.80     66378
weighted avg       0.90      0.90      0.90     66378

ROC AUC score: 0.782817696440173
```


Training and Validation Loss