Data Mining – Final Report

Project Name – Credit card fraud detection

Somanadh Koneti
NJIT ID: 31608747
sk3395@njit.edu

Ying Wu College of Computing

Contents

1. Introduction

In this project, we will use various supervised learning algorithms to detect credit card fraud utilizing the transaction dataset obtained from https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud. We will choose the most promising algorithm from the preliminary results and refine it to create the best model for the given data and needs. The objective is to create a model that correctly detects whether a credit card transaction is fraudulent.

2. Dataset

The dataset contains transactions made by credit cards in September 2013 by European cardholders.

It presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) accounts for 0.172% of all transactions.

Number of attributes/columns in the data set – 31

Number of records – 284,807 records.

This dataset only has numeric input variables that have undergone PCA transformation. Due to confidentiality issues, the original features and their background information are not provided. Features V1 through V28 are the principal components obtained with PCA, the only features that have not been transformed with PCA are **Time** and **Amount**.

3. Importing Libraries

First, we import all the required libraries for the project. We then download the dataset from https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud and place it in the root of the project folder, allowing for easy access.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import train_test_split
```

4. Creating a pandas data frame

Read the dataset into a pandas data frame and display the head of the dataset.

In [2]:
```
df = pd.read_csv('./data/creditcard.csv')
df.head()
```

Out[2]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206

5 rows × 31 columns

5. Understanding the data

In this phase, we get to know our data better. As mentioned earlier, we know that except for the transaction and amount, we do not have any information on the other columns (due to confidentiality reasons).

Firstly, make sure that there are no null values in the data.

```
In [3]: df.isnull().sum().max()
Out[3]: 0
```

Then we use **describe()** to get the mean, median, quartile ranges, etc.

```
In [4]: df.describe()
```

Out[4]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15	-5.556467e-16	1.213481e-16	-2.406331e-15
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00	1.098632e+00
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01	-1.343407e+01
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01	-6.430976e-01
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02	-5.142873e-02
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01	5.971390e-01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01	1.559499e+01

8 rows × 31 columns

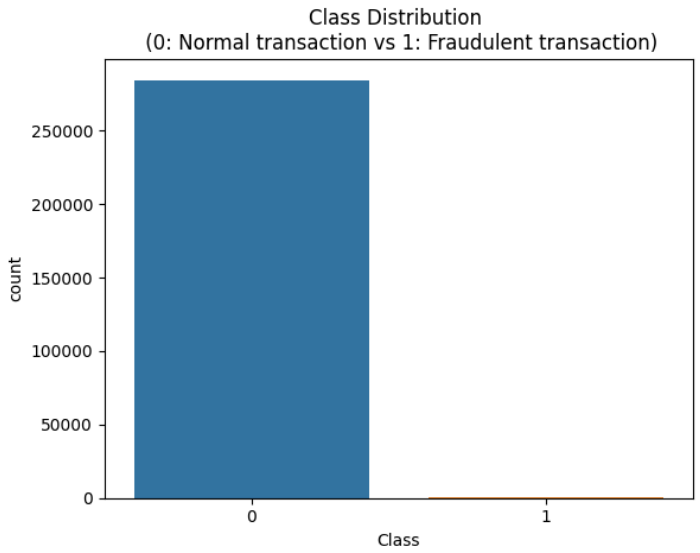
Below we use the **Class** attribute from the dataset to separate the normal transactions from the fraudulent ones.

```
In [5]: print('Normal transactions:', f"{df['Class'].value_counts()[0]/len(df) * 100:.2f}", '% of the dataset')
        print('Fraudulent transactions:', f"{df['Class'].value_counts()[1]/len(df) * 100:.2f}", '% of the dataset')

        Normal transactions: 99.83 % of the dataset
        Fraudulent transactions: 0.17 % of the dataset
```

As we can see, our original dataset is highly imbalanced. If we base our predictive models and analyses on this data frame, we may receive a lot of inaccuracies since our algorithm could believe that most transactions are not fraudulent. Below we can see the **Class** distribution.

```
In [6]: sns.countplot(df, x='Class')
        plt.title('Class Distribution \n (0: Normal transaction vs 1: Fraudulent transaction)')
Out[6]: Text(0.5, 1.0, 'Class Distribution \n (0: Normal transaction vs 1: Fraudulent transaction)')
```



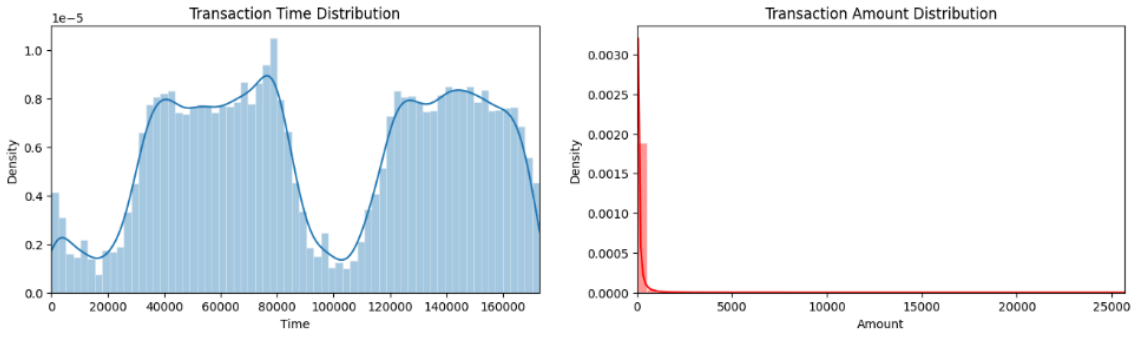
The distributions below will give us an understanding of how skewed these features are.

In [7]:
```
fig, ax = plt.subplots(1, 2, figsize=(16,4))

sns.histplot(
    df['Time'], kde=True,
    stat="density", kde_kws=dict(cut=3),
    alpha=.4, edgecolor=(1, 1, 1, .4),
    ax=ax[0],
)
ax[0].set_title('Transaction Time Distribution')
ax[0].set_xlim([min(df['Time']), max(df['Time'])])

sns.histplot(
    df['Amount'], kde=True,
    stat="density", kde_kws=dict(cut=3),
    alpha=.4, edgecolor=(1, 1, 1, .4),
    ax=ax[1],
    bins=50,
    color='r'
)
ax[1].set_title('Transaction Amount Distribution')
ax[1].set_xlim([min(df['Amount']), max(df['Amount'])])

plt.show()
```



6. Data Preprocessing

6.1. Scaling

Since all the other features are scaled, we only need to scale **Time** and **Amount**.

In [8]:
```
scaler = RobustScaler()

if 'Time' in df:
    scaled_time = scaler.fit_transform(df['Time'].values.reshape(-1,1))
    scaled_amount = scaler.fit_transform(df['Amount'].values.reshape(-1,1))

    df.drop(['Time','Amount'], axis=1, inplace=True)
    df.insert(0, 'scaled_time', scaled_time)
    df.insert(1, 'scaled_amount', scaled_amount)

df.head()
```

Out[8]:

	scaled_time	scaled_amount	V1	V2	V3	V4	V5	V6	V7	V8	...	V20	V21	V22	
0	-0.994983	1.783274	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	...	0.251412	-0.018307	0.277838	-0.1
1	-0.994983	-0.269825	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	...	-0.069083	-0.225775	-0.638672	0.1
2	-0.994972	4.983721	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	...	0.524980	0.247998	0.771679	0.9
3	-0.994972	1.418291	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	...	-0.208038	-0.108300	0.005274	-0.1
4	-0.994960	0.670579	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	...	0.408542	-0.009431	0.798278	-0.1

5 rows × 31 columns

6.2. Splitting the data frame

We will use train_test_split to split the data frame.

```
In [9]: x = df.drop('Class', axis=1)
        y = df['Class']

        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
```

Now that we have split the data, we can use the training and test sets to perform analysis on the dataset using different algorithms.

7. Random Under-Sampling

We will now implement random under-sampling to avoid overfitting. This is done by removing data to have a more balanced dataset. We create a subsample of our dataframe containing a 1:1 ratio of fraudulent transactions to non-fraudulent transactions, this will be equivalent to 492 fraudulent and non-fraudulent transactions each. The aim is to randomize the data each time this script is run to determine if our models can retain a particular level of accuracy.

```
In [10]: # Get a random sample of items from the dataset
         df = df.sample(frac=1)

         # 492 is used since it is the amound of fraud classes in the dataset
         fraud_df = df.loc[df['Class'] == 1]
         non_fraud_df = df.loc[df['Class'] == 0][:492]

         normal_distributed_df = pd.concat([fraud_df, non_fraud_df])

         # Shuffle dataframe rows
         new_df = normal_distributed_df.sample(frac=1, random_state=42)

         new_df.head()
```

Out[10]:

	scaled_time	scaled_amount	V1	V2	V3	V4	V5	V6	V7	V8	...	V20	V21	V22
175362	0.441828	-0.070146	-2.365240	-1.190453	-0.901472	0.098546	1.896948	-1.676791	0.468094	-0.059877	...	-1.016841	-0.694021	-0.554366
116139	-0.123744	0.767694	-1.548788	1.808698	-0.953509	2.213085	-2.015728	-0.913457	-2.356013	1.197169	...	0.390786	0.855138	0.774745
79352	-0.313808	-0.056033	1.173256	0.158617	0.711974	1.326027	-0.534195	-0.541976	-0.024042	-0.090440	...	-0.120791	-0.261346	-0.589933
182992	0.480739	-0.262419	1.889618	1.073099	-1.678018	4.173268	1.015516	-0.009389	-0.079706	0.064071	...	-0.153570	0.203728	0.733796
249828	0.821285	-0.213233	0.667714	3.041502	-5.845112	5.967587	0.213863	-1.462923	-2.688761	0.677764	...	0.558425	0.329760	-0.941383

5 rows × 31 columns

Below, we can see the class distribution of the subsampled dataset.

```
In [11]: print('Class Distribution in the subsample dataset')
         print(new_df['Class'].value_counts()/len(new_df))

         sns.countplot(new_df, x='Class')
         plt.title('Equally Distributed Classes', fontsize=14)
         plt.show()
```

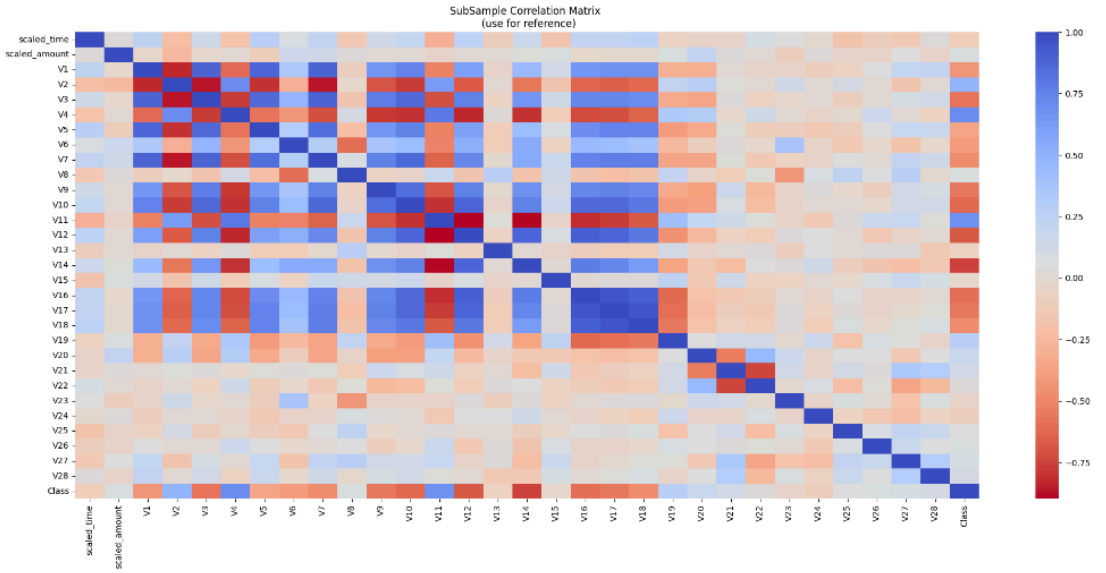
```
Class Distribution in the subsample dataset
0    0.5
1    0.5
Name: Class, dtype: float64
```

7.1. Correlation Matrices

The key to comprehending our data is correlation matrices. If there are elements that significantly affect whether a particular transaction is fraudulent, we want to know about them. To examine which characteristics have a strong positive or negative association with regard to fraudulent transactions, we must use the subsample dataframe.

```
In [12]: f, (ax) = plt.subplots(1, 1, figsize=(24,10))

sub_sample_corr = new_df.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax)
ax.set_title('SubSample Correlation Matrix \n (use for reference)')
plt.show()
```



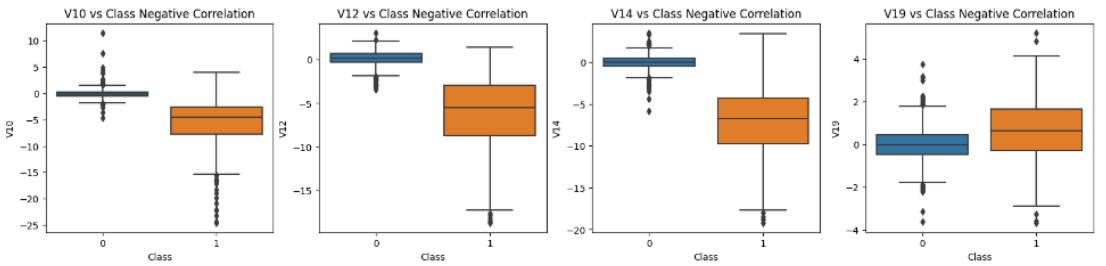
We use box plots to gain a better understanding of the distribution. Below, we plot the boxplots of the features with negative correlations against Class. The lower the feature value, the more likely that the transaction is fraudulent.

```
In [13]: f, ax = plt.subplots(ncols=4, figsize=(20,4))

def boxplot_negative_corr(column, ax):
    sns.boxplot(x="Class", y=column, data=new_df, ax=ax)
    ax.set_title(f'{column} vs Class Negative Correlation')

# Negative Correlations with Class (The lower the feature value the more likely it will be a fraud transaction)
boxplot_negative_corr('V10', ax[0])
boxplot_negative_corr('V12', ax[1])
boxplot_negative_corr('V14', ax[2])
boxplot_negative_corr('V19', ax[3])

plt.show()
```



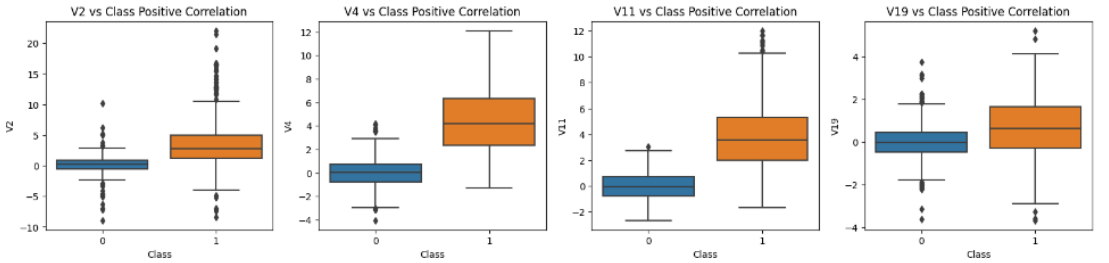
Now we plot the boxplots of the features with positive correlations against Class. The higher the feature value, the more likely that the transaction is fraudulent.

In [14]:
```
f, ax = plt.subplots(ncols=4, figsize=(20,4))

def boxplot_positive_corr(column, ax):
    sns.boxplot(x="Class", y=column, data=new_df, ax=ax)
    ax.set_title(f'{column} vs Class Positive Correlation')

# Positive correlations with Class (The higher the feature value the more likely it will be a fraud transaction)
boxplot_positive_corr('V2', ax[0])
boxplot_positive_corr('V4', ax[1])
boxplot_positive_corr('V11', ax[2])
boxplot_positive_corr('V19', ax[3])

plt.show()
```



7.2. Anomaly Detection

In this phase, we remove the outliers from features in order to have a high correlation with our classes. This will improve the accuracy of our models.

We start off by visualizing the distribution of the feature we will use to eliminate outliers. Since V14 is the only feature that has a Gaussian distribution similar to V12 and V10, we will use it.

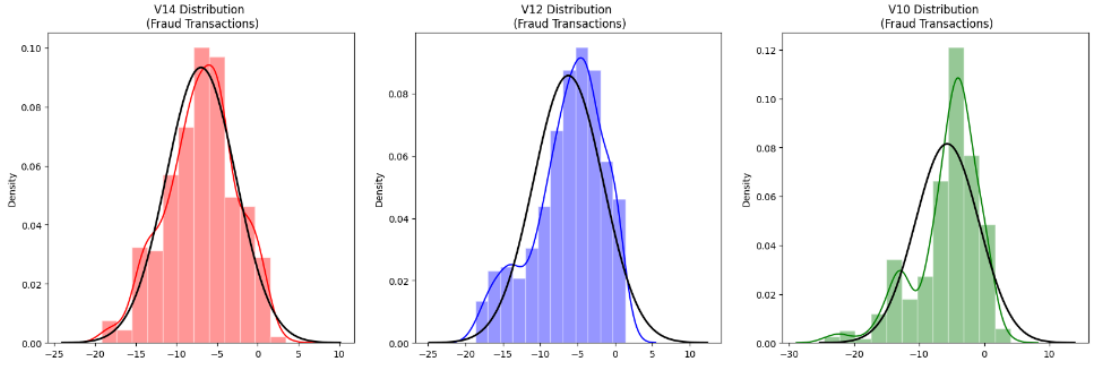
In [15]:
```
f, ax = plt.subplots(1,3, figsize=(20, 6))

def normal(mean, std, ax, color="black"):
    x = np.linspace(mean-4*std, mean+4*std, 200)
    p = norm.pdf(x, mean, std)
    z = ax.plot(x, p, color, linewidth=2)

def plot(column, ax, color):
    fraud_dist = new_df[column].loc[new_df['Class'] == 1].values
    sns.histplot(
        fraud_dist, kde=True,
        stat="density", kde_kws=dict(cut=3),
        alpha=.4, edgecolor=(1, 1, 1, .4),
        ax=ax,
        bins=12,
        color=color
    )
    ax.set_title(f'{column} Distribution \n (Fraud Transactions)')
    normal(fraud_dist.mean(), fraud_dist.std(), ax)

plot('V14', ax[0], 'r')
plot('V12', ax[1], 'b')
plot('V10', ax[2], 'g')

plt.show()
```



Next, we will determine the threshold. We do this by multiplying the IQR by 1.5. This will become the cut-off. We will then subtract and add this value to q25 and q75 to obtain the lower extreme threshold and upper extreme threshold respectively. Finally, we will remove the instances which exceed the threshold in both extremes.

In [16]:
```
def remove_outliers(column):
    global new_df
    fraud = new_df[column].loc[new_df['Class'] == 1].values
    q25, q75 = np.percentile(fraud, 25), np.percentile(fraud, 75)
    print('Quartile 25: {} | Quartile 75: {}'.format(q25, q75))
    iqr = q75 - q25
    print('IQR: {}'.format(iqr))

    cut_off = iqr * 1.5
    lower, upper = q25 - cut_off, q75 + cut_off
    print('Cut Off: {}'.format(cut_off))
    print('{} Lower: {}'.format(column, lower))
    print('{} Upper: {}'.format(column, upper))

    outliers = [x for x in fraud if x < lower or x > upper]
    print('Feature {} Outliers for Fraud Cases: {}'.format(column, len(outliers)))
    print('{} outliers:{}'.format(column, outliers))

    new_df = new_df.drop(new_df[(new_df[column] > upper) | (new_df[column] < lower)].index)
    print('Number of Instances after outliers removal: {}'.format(len(new_df)))
    print('----' * 28)

# # -----> V14 Removing Outliers (Highest Negative Correlated with Labels)
remove_outliers('V14')

# -----> V12 removing outliers from fraud transactions
remove_outliers('V12')

# Removing outliers V10 Feature
remove_outliers('V10')
```

The output below indicates the instances that were removed

```
Quartile 25: -9.692722964972386 | Quartile 75: -4.282820849486865
IQR: 5.409902115485521
Cut Off: 8.114853173228282
V14 Lower: -17.807576138200666
V14 Upper: 3.8320323237414167
Feature V14 Outliers for Fraud Cases: 4
V14 outliers:[-18.4937733551053, -18.8220867423816, -18.0499976898594, -19.2143254902614]
Number of Instances after outliers removal: 980
----------------------------------------------------------------------------------------------------
Quartile 25: -8.67303320439115 | Quartile 75: -2.893030568676315
IQR: 5.780002635714835
Cut Off: 8.670003953572252
V12 Lower: -17.3430371579634
V12 Upper: 5.776973384895937
Feature V12 Outliers for Fraud Cases: 4
V12 outliers:[-18.5536970096458, -18.0475965708216, -18.4311310279993, -18.6837146333443]
Number of Instances after outliers removal: 976
----------------------------------------------------------------------------------------------------
Quartile 25: -7.466658535821847 | Quartile 75: -2.5118611381562523
IQR: 4.954797397665595
Cut Off: 7.432196096498393
V10 Lower: -14.89885463232024
V10 Upper: 4.92033495834214
Feature V10 Outliers for Fraud Cases: 27
V10 outliers:[-23.2282548357516, -15.1237521803455, -17.1415136412892, -14.9246547735487, -19.836148851696, -15.23
99619587112, -20.9491915543611, -18.9132433348732, -16.6496281595399, -16.6011969664137, -16.2556117491401, -15.12
41628144947, -24.4031849699728, -15.2399619587112, -15.2318333653018, -18.2711681738888, -16.7460441053944, -15.56
37913387301, -22.1870885620007, -16.3035376590131, -24.5882624372475, -22.1870885620007, -15.5637913387301, -14.92
46547735487, -22.1870885620007, -15.3460988468775, -22.1870885620007]
Number of Instances after outliers removal: 947
----------------------------------------------------------------------------------------------------
```

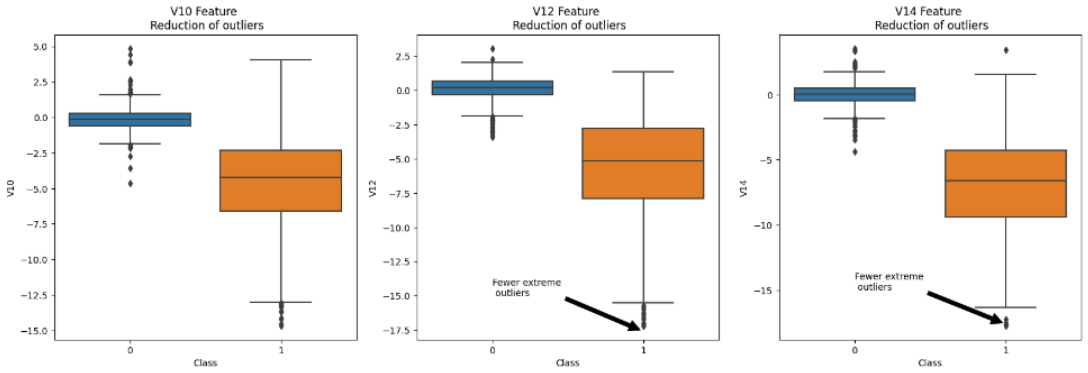
Using boxplot representation, we can visualize that the outliers have been reduced.

In [17]:
```
f,ax = plt.subplots(1, 3, figsize=(20,6))

def boxplot_without_outliers(column, ax):
    sns.boxplot(x="Class", y=column, data=new_df,ax=ax)
    ax.set_title(f"{column} Feature \n Reduction of outliers")
    ax.annotate('Fewer extreme \n outliers', xy=(0.98, -17.5), xytext=(0, -15),
                arrowprops=dict(facecolor='black'))

# Boxplots with outliers removed
# Feature V10
boxplot_without_outliers('V10', ax[0])
# Feature 12
boxplot_without_outliers('V12', ax[1])
# Feature V14
boxplot_without_outliers('V14', ax[2])

plt.show()
```



7.3. Classifiers

Here, we will train 4 different types of classifiers and determine which classifier will be more effective in detecting fraud transactions. The classifiers we will train are Logistic regression, K-neighbors, Random forest, and Decision tree.

First, we will prepare our data for classification. We will use undersampling to create a training set and a testing set.

In [18]:
```
# Undersampling before cross validating (prone to overfit)
x = new_df.drop('Class', axis=1)
y = new_df['Class']

# This is explicitly used for undersampling.
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# Turn the values into an array for feeding the classification algorithms.
x_train = x_train.values
x_test = x_test.values
y_train = y_train.values
y_test = y_test.values

# Let's implement these classifiers
classifiers = {
    "LogisiticRegression": LogisticRegression(),
    "KNearest": KNeighborsClassifier(),
    "Random Forest Classifier": RandomForestClassifier(),
    "DecisionTreeClassifier": DecisionTreeClassifier()
}
```

Then we will evaluate the cross-validation score for the 4 classifiers.

In [19]:
```
for key, classifier in classifiers.items():
    classifier.fit(x_train, y_train)
    training_score = cross_val_score(classifier, x_train, y_train, cv=5)
    print("Classifiers: ", classifier.__class__.__name__, "Training score -",
          f"{training_score.mean() * 100:.2f}", "% accuracy")
```

```
Classifiers:  LogisticRegression Training score - 93.92 % accuracy
Classifiers:  KNeighborsClassifier Training score - 92.47 % accuracy
Classifiers:  RandomForestClassifier Training score - 93.53 % accuracy
Classifiers:  DecisionTreeClassifier Training score - 90.75 % accuracy
```

Next, we will use **GridSearchCV** to determine the parameters that will give the best predictive scores for the classifiers. This will give us the best estimators for each classifier. We will in turn use these estimators to evaluate the cross-validation score for the 4 classifiers.

In [20]:
```
# Use GridSearchCV to find the determine parameters which give best predictive scores.
log_reg_params = {"penalty": ['l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}
knears_params = {"n_neighbors": list(range(2,5,1)), 'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']}
rf_params = { 'n_estimators': [200, 500], 'max_depth' : [4,5,6,7,8], 'criterion' :['gini', 'entropy']}
tree_params = {"criterion": ["gini", "entropy"], "max_depth": list(range(2,4,1)),
               "min_samples_leaf": list(range(5,7,1))}

def get_pred(classifier, params, decision = False):
    grid = GridSearchCV(classifier, params)
    grid.fit(x_train, y_train)
    # We automatically get the best parameters using the estimator
    estimator = grid.best_estimator_
    pred = cross_val_predict(estimator, x_train, y_train, cv=5, method="decision_function") if decision else
    cross_val_predict(estimator, x_train, y_train, cv=5)
    return estimator, pred

log_reg, log_reg_pred = get_pred(LogisticRegression(solver='lbfgs', max_iter=1000), log_reg_params, True)
knears, knears_pred = get_pred(KNeighborsClassifier(), knears_params)
rf, rf_pred = get_pred(RandomForestClassifier(), rf_params)
tree, tree_pred = get_pred(DecisionTreeClassifier(), tree_params, False)

# Overfitting Case

log_reg_score = cross_val_score(log_reg, x_train, y_train, cv=5)
print('Logistic Regression Cross Validation Score: ', round(log_reg_score.mean() * 100, 2).astype(str) + '%')

knears_score = cross_val_score(knears, x_train, y_train, cv=5)
print('Knears Neighbors Cross Validation Score', round(knears_score.mean() * 100, 2).astype(str) + '%')

rf_score = cross_val_score(rf, x_train, y_train, cv=5)
print('Random Forest Classifier Cross Validation Score', round(rf_score.mean() * 100, 2).astype(str) + '%')

tree_score = cross_val_score(tree, x_train, y_train, cv=5)
print('DecisionTree Classifier Cross Validation Score', round(tree_score.mean() * 100, 2).astype(str) + '%')
```

```
Logistic Regression Cross Validation Score:  93.92%
Knears Neighbors Cross Validation Score 92.47%
Random Forest Classifier Cross Validation Score 93.53%
DecisionTree Classifier Cross Validation Score 92.07%
```

Finally, we will use the predictions obtained above to calculate the Receiver Operating Characteristic Area Under Curve (ROC AUC) score.

In [21]:
```
# Comparing roc_auc_score of the classifiers
classifier_pred_map = {
  "LogisticRegression": log_reg_pred,
  "KNeighborsClassifier": knears_pred,
  "RandomForestClassifier": rf_pred,
  "DecisionTreeClassifier": tree_pred
}

for key, classifier in classifiers.items():
    pred = classifier_pred_map[classifier.__class__.__name__]
    print(classifier.__class__.__name__, 'ROC AUC score:', f"{roc_auc_score(y_train, pred) * 100:.2f}%")
```

```
LogisticRegression ROC AUC score: 97.47%
KNeighborsClassifier ROC AUC score: 92.31%
RandomForestClassifier ROC AUC score: 93.66%
DecisionTreeClassifier ROC AUC score: 91.57%
```

As we can see, Logistic Regression has the best ROC AUC score, which means Logistic Regression will most accurately separate **fraud** and **non-fraud** transactions.

7.4. Comparing Classifiers

In this section, we will compare the 4 classification models to further determine which model fits best for our use case. First, we compare the accuracy score, precision score, recall score, f1 score, and ROC AUC score of the 4 classifiers.

In [22]:
```
# Comparing accuracy_score, precision_score, recall_score, f1_score and roc_auc_score of the classifiers
classifier_pred_map = {
  "LogisticRegression": log_reg_pred,
  "KNeighborsClassifier": knears_pred,
  "RandomForestClassifier": rf_pred,
  "DecisionTreeClassifier": tree_pred
}

def get_pred(c):
    c.fit(x_train, y_train)
    return c.predict(x_test)

for key, classifier in classifiers.items():
    pred = get_pred(classifier)
    for score in [accuracy_score, precision_score, recall_score, f1_score, roc_auc_score]:
        print(classifier.__class__.__name__, f'{score.__name__}:', f"{score(y_test, pred) * 100:.2f}%")
    print('----' * 28)
```

```
LogisticRegression accuracy_score: 93.16%
LogisticRegression precision_score: 96.59%
LogisticRegression recall_score: 89.47%
LogisticRegression f1_score: 92.90%
LogisticRegression roc_auc_score: 93.16%
--------------------------------------------------------------------------------------------------------------------------------
KNeighborsClassifier accuracy_score: 91.05%
KNeighborsClassifier precision_score: 98.75%
KNeighborsClassifier recall_score: 83.16%
KNeighborsClassifier f1_score: 90.29%
KNeighborsClassifier roc_auc_score: 91.05%
--------------------------------------------------------------------------------------------------------------------------------
RandomForestClassifier accuracy_score: 91.05%
RandomForestClassifier precision_score: 98.75%
RandomForestClassifier recall_score: 83.16%
RandomForestClassifier f1_score: 90.29%
RandomForestClassifier roc_auc_score: 91.05%
--------------------------------------------------------------------------------------------------------------------------------
DecisionTreeClassifier accuracy_score: 85.79%
DecisionTreeClassifier precision_score: 88.64%
DecisionTreeClassifier recall_score: 82.11%
DecisionTreeClassifier f1_score: 85.25%
DecisionTreeClassifier roc_auc_score: 85.79%
--------------------------------------------------------------------------------------------------------------------------------
```

Taking the accuracy score of the classifiers into consideration, we get the following scores

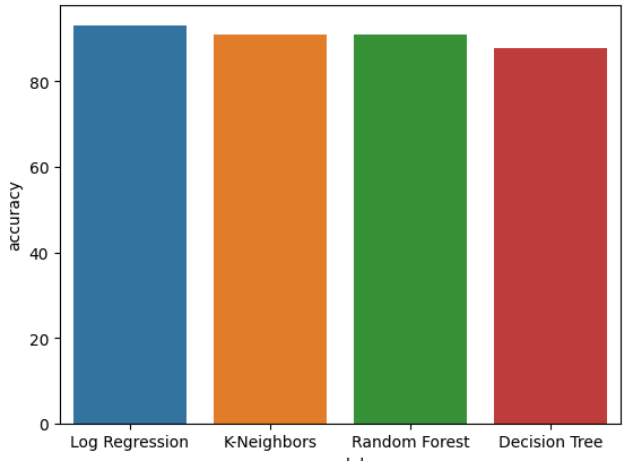
In [23]:
```
final_df = pd.DataFrame({
    "model": ["Log Regression", "K-Neighbors", "Random Forest", "Decision Tree"]
    ,"accuracy": [
        accuracy_score(y_test, get_pred(LogisticRegression()))*100,
        accuracy_score(y_test, get_pred(KNeighborsClassifier()))*100,
        accuracy_score(y_test, get_pred(RandomForestClassifier()))*100,
        accuracy_score(y_test, get_pred(DecisionTreeClassifier()))*100
    ]
})
final_df
```

Out[23]:

	model	accuracy
0	Log Regression	93.157895
1	K-Neighbors	91.052632
2	Random Forest	91.052632
3	Decision Tree	87.894737

This can be visualized better using a bar plot as seen below.

```
In [24]: sns.barplot(x="model", y="accuracy", data=final_df)
Out[24]: <Axes: xlabel='model', ylabel='accuracy'>
```



We can confirm from the above image that Logistic Regression is the best algorithm to detect fraudulent transactions with an accuracy score of 93.16%, followed by K-neighbors and Random Forest with 91.05% accuracy. Decision tree classifier has the least accuracy with 87.89%.

8. Conclusion

We got to know our data better by analyzing the various distribution patterns. Our data then underwent preprocessing and we split the data using train_test_split. Then using random under-sampling, we created a subsample of our dataset. We used a correlation matrix to identify the positive and negative correlated features in our dataset and visualized them using a boxplot. Then, we removed the outliers from our dataset to achieve a higher correlation and improve the accuracy of our model. Next, we tested our data against 4 different classification algorithms, namely Logistic Regression, K-Neighbors Classifier, Random Forest Classifier, and Decision Tree Classifier. Upon analysis, we conclude that Logistic Regression is the most appropriate algorithm to detect fraudulent transactions achieving an accuracy score of 93.16% and a ROC AUC score of 97.47%.