Detecting Fraud and Non-fraud Credit Transactions

Use various predictive models to see how accurate they are in detecting whether a transaction is a normal payment or a fraud. The features are scaled and the names of the features are not shown due to privacy reasons.

The Goals:

- Determine the Classifiers we are going to use and decide which one has a higher accuracy.
- Create a Neural Network and compare the accuracy to our best classifier.
- Understand common mistaked made with imbalanced datasets.

Import Libraries

In [1]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA, TruncatedSVD
import matplotlib.patches as mpatches
import time

# Classifier Libraries
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import collections


# Other Libraries
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from imblearn.pipeline import make_pipeline as imbalanced_make_pipeline
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import NearMiss
from imblearn.metrics import classification_report_imbalanced
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score, ac
from collections import Counter
from sklearn.model_selection import KFold, StratifiedKFold
import warnings
warnings.filterwarnings("ignore")


df = pd.read_csv('../input/creditcard.csv')
df.head()
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/externals/six.py:31: DeprecationWarnin
g: The module is deprecated in version 0.21 and will be removed in version 0.23 since
we've dropped support for Python 2.7. Please rely on the official version of six (htt
ps://pypi.org/project/six/).
  "(https://pypi.org/project/six/).", DeprecationWarning)
```

Out[1]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.3637
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.2554
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.5146
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.3870
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.8177

In [2]:
```
df.describe()
```

Out[2]:

	Time	V1	V2	V3	V4	V5	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8
mean	94813.859575	3.919560e-15	5.688174e-16	-8.769071e-15	2.782312e-15	-1.552563e-15	2.
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.

In [3]:
```
# No Null Values
df.isnull().sum().max()
```

Out[3]:
```
0
```

In [4]:
```
df.columns
```

Out[4]:
```
Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10',
       'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20',
       'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount',
       'Class'],
      dtype='object')
```

The classes are heavily skewed we need to solve this issue later.

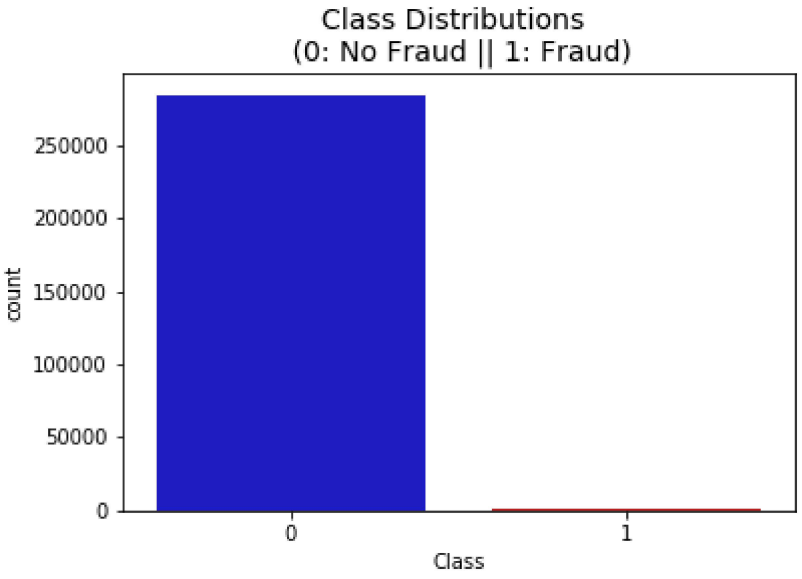
In [5]:
```
print('No Frauds', round(df['Class'].value_counts()[0]/len(df) * 100,2), '% of the dat
print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,2), '% of the datase
```

```
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
```

Note: Notice how imbalanced our original dataset is. Most of the transactions are non-fraudulent. If we use this dataframe as the base for our predictive models and analysis, we might get a lot of errors, and our algorithms will probably overfit, since they will "assume" that most transactions are not fraudulent. But we don't want our model to assume; we want our model to detect patterns that give signs of fraud.

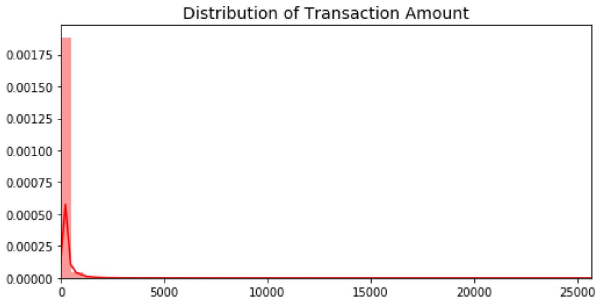
In [6]:
```
colors = ["#0101DF", "#DF0101"]

sns.countplot('Class', data=df, palette=colors)
plt.title('Class Distributions \n (0: No Fraud || 1: Fraud)', fontsize=14)
```

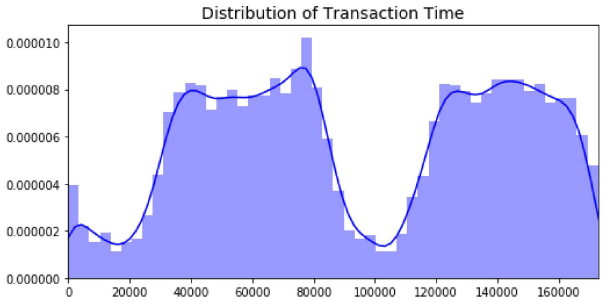
Out[6]:
```
Text(0.5, 1.0, 'Class Distributions \n (0: No Fraud || 1: Fraud)')
```



Distributions: By examining these distributions, we can gain insight into the degree of skewness in these features. Furthermore, we can explore the distributions of other features. In this notebook, we will also implement techniques to reduce the skewness in these distributions in the future.

In [7]:
```
fig, ax = plt.subplots(1, 2, figsize=(18,4))

amount_val = df['Amount'].values
time_val = df['Time'].values

sns.distplot(amount_val, ax=ax[0], color='r')
ax[0].set_title('Distribution of Transaction Amount', fontsize=14)
ax[0].set_xlim([min(amount_val), max(amount_val)])

sns.distplot(time_val, ax=ax[1], color='b')
ax[1].set_title('Distribution of Transaction Time', fontsize=14)
ax[1].set_xlim([min(time_val), max(time_val)])

plt.show()
```





Scaling and Distributing

We will begin by scaling the columns that comprise 'Time' and 'Amount.' 'Time' and 'Amount' should be scaled in the same way as the other columns. Additionally, we need to create a subsample of the dataframe to ensure an equal number of fraud and non-fraud cases, which will help our algorithms better understand the patterns that determine whether a transaction is fraudulent or not.

What is a sub-Sample?

In this scenario, our subsample will consist of a dataframe with an equal 50/50 ratio of fraud and non-fraud transactions, meaning it will contain an equal number of both types of transactions.

Why do we create a sub-Sample?

At the beginning of this notebook, we observed that the original dataframe was heavily imbalanced. Using the original dataframe can lead to the following issues:

Overfitting: Our classification models may assume that fraud cases are rare, potentially leading to overfitting. What we want is a model that can accurately detect fraud cases.

Incorrect Correlations: While we may not know the exact meaning of the "V" features, understanding how each feature influences the outcome (Fraud or No Fraud) can be valuable. An imbalanced dataframe can obscure the true correlations between the class and features.

Summary:

The columns with scaled values are Scaled Amount and Scaled Time.

In our dataset, there are 492 cases of fraud. To create our new sub-dataframe, we will randomly select 492 cases of non-fraud.

We will concatenate the 492 cases of fraud and non-fraud, thus creating a new sub-sample.

In [8]:
```
# Since most of our data has already been scaled we should scale the columns that are
from sklearn.preprocessing import StandardScaler, RobustScaler

# RobustScaler is less prone to outliers.
```

```
std_scaler = StandardScaler()
rob_scaler = RobustScaler()

df['scaled_amount'] = rob_scaler.fit_transform(df['Amount'].values.reshape(-1,1))
df['scaled_time'] = rob_scaler.fit_transform(df['Time'].values.reshape(-1,1))

df.drop(['Time','Amount'], axis=1, inplace=True)
```

In [9]:
```
scaled_amount = df['scaled_amount']
scaled_time = df['scaled_time']

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

# Amount and Time are Scaled

df.head()
```

Out[9]:

	scaled_amount	scaled_time	V1	V2	V3	V4	V5	V6	
0	1.783274	-0.994983	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.2395
1	-0.269825	-0.994983	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.0788
2	4.983721	-0.994972	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.7914
3	1.418291	-0.994972	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.2376
4	0.670579	-0.994960	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.5929

Splitting the Original DataFrame

Before proceeding with the Random UnderSampling technique, we need to separate the original dataframe. This is necessary for testing purposes. Remember that, although we split the data when implementing Random UnderSampling or OverSampling techniques, our ultimate goal is to test our models on the original testing set, not on the testing set created by either of these techniques. The primary objective is to train the model using the undersampled and oversampled dataframes to help it detect patterns, and then evaluate its performance on the original testing set.

In [10]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

print('No Frauds', round(df['Class'].value_counts()[0]/len(df) * 100,2), '% of the dat
print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,2), '% of the datase

X = df.drop('Class', axis=1)
y = df['Class']

sss = StratifiedKFold(n_splits=5, random_state=None, shuffle=False)

for train_index, test_index in sss.split(X, y):
    print("Train:", train_index, "Test:", test_index)
    original_Xtrain, original_Xtest = X.iloc[train_index], X.iloc[test_index]
```

```
    original_ytrain, original_ytest = y.iloc[train_index], y.iloc[test_index]

# We already have X_train and y_train for undersample data thats why I am using origin
# original_Xtrain, original_Xtest, original_ytrain, original_ytest = train_test_split(

# Check the Distribution of the labels


# Turn into an array
original_Xtrain = original_Xtrain.values
original_Xtest = original_Xtest.values
original_ytrain = original_ytrain.values
original_ytest = original_ytest.values

# See if both the train and test label distribution are similarly distributed
train_unique_label, train_counts_label = np.unique(original_ytrain, return_counts=True
test_unique_label, test_counts_label = np.unique(original_ytest, return_counts=True)
print('-' * 100)

print('Label Distributions: \n')
print(train_counts_label/ len(original_ytrain))
print(test_counts_label/ len(original_ytest))
```

```
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
Train: [ 30473  30496  31002 ... 284804 284805 284806] Test: [    0     1     2 ... 5
7017 57018 57019]
Train: [     0      1      2 ... 284804 284805 284806] Test: [ 30473  30496  31002
... 113964 113965 113966]
Train: [     0      1      2 ... 284804 284805 284806] Test: [ 81609  82400  83053
... 170946 170947 170948]
Train: [     0      1      2 ... 284804 284805 284806] Test: [150654 150660 150661
... 227866 227867 227868]
Train: [     0      1      2 ... 227866 227867 227868] Test: [212516 212644 213092
... 284804 284805 284806]
----------------------------------------------------------------------------------
--------------
Label Distributions:

[0.99827076 0.00172924]
[0.99827952 0.00172048]
```

Random Under-Sampling:

In this phase of the project, we will implement Random Under Sampling, which involves removing data to create a more balanced dataset, thereby preventing our models from overfitting.

Steps: First, we need to determine how imbalanced our class is. You can use value_counts() on the class column to calculate the number of instances for each label.

Once we have determined the number of fraudulent transactions (Fraud = "1"), we should adjust the number of non-fraudulent transactions to match the number of fraudulent transactions. Assuming a 50/50 ratio, this would be equivalent to having 492 cases of both fraudulent and non-fraudulent transactions.

After implementing this technique, we will have a sub-sample of our dataframe with a 50/50 ratio for our classes. The next step is to shuffle the data to see if our models can consistently maintain accuracy every time we run this script.

Note: The primary issue with Random Under-Sampling is the risk that our classification models may not perform as accurately as desired due to a significant amount of information loss (reducing 492 non-fraud transactions from the original 284,315 non-fraud transactions).

In [11]:
```
# Since our classes are highly skewed we should make them equivalent in order to have

# Lets shuffle the data before creating the subsamples

df = df.sample(frac=1)

# amount of fraud classes 492 rows.
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[11]:

	scaled_amount	scaled_time	V1	V2	V3	V4	V5	V6
153591	-0.293440	0.174567	-1.280786	1.530958	1.700818	-0.444377	1.035234	1.277053
6446	-0.293440	-0.904851	0.702710	2.426433	-5.234513	4.416661	-2.170806	-2.667554
66047	-0.097813	-0.385778	0.006388	-0.112461	0.374538	-2.372199	-0.367237	-1.530939
541	-0.307413	-0.990214	-2.312227	1.951992	-1.609851	3.997906	-0.522188	-1.426545
83053	-0.219800	-0.293977	0.326007	1.286638	-2.007181	2.419675	-1.532902	-1.432803

Equally Distributing and Correlating:

Now that we have our dataframe correctly balanced, we can go further with our analysis and data preprocessing.

In [12]:
```
print('Distribution of the Classes in the subsample dataset')
print(new_df['Class'].value_counts()/len(new_df))



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

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

Correlation Matrices

Correlation matrices are essential for understanding our data. We aim to identify features that significantly influence whether a transaction is fraudulent. It's crucial to use the correct dataframe (subsample) to determine which features exhibit a strong positive or negative correlation with respect to fraud transactions.

Summary and Explanation:

Negative Correlations: V17, V14, V12, and V10 exhibit negative correlations. Lower values of these features are associated with a higher likelihood of a transaction being fraudulent.

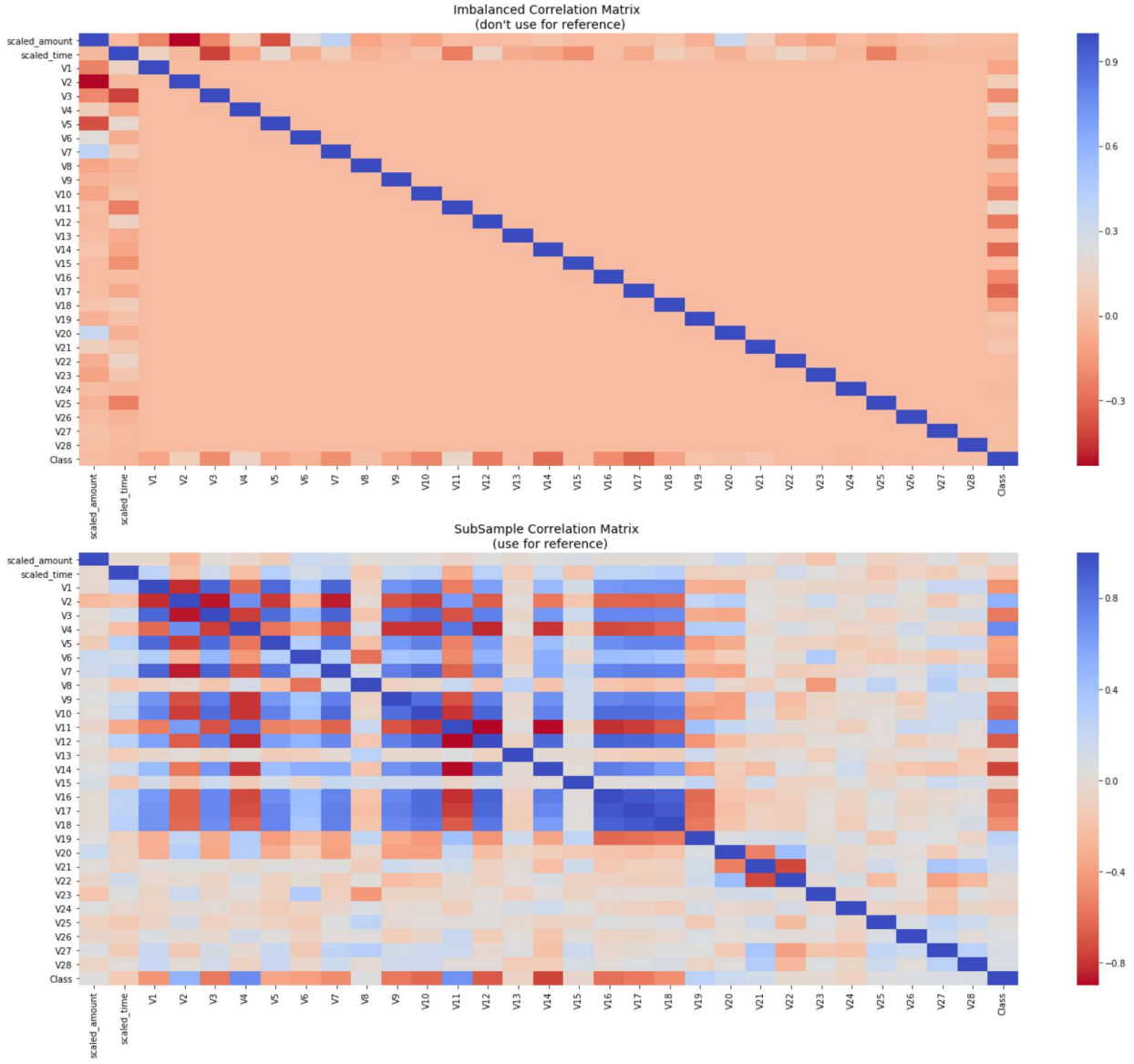
Positive Correlations: V2, V4, V11, and V19 show positive correlations. Higher values of these features are associated with a higher likelihood of a transaction being fraudulent.

Box Plots: We will use box plots to better understand the distribution of these features in fraudulent and non-fraudulent transactions.

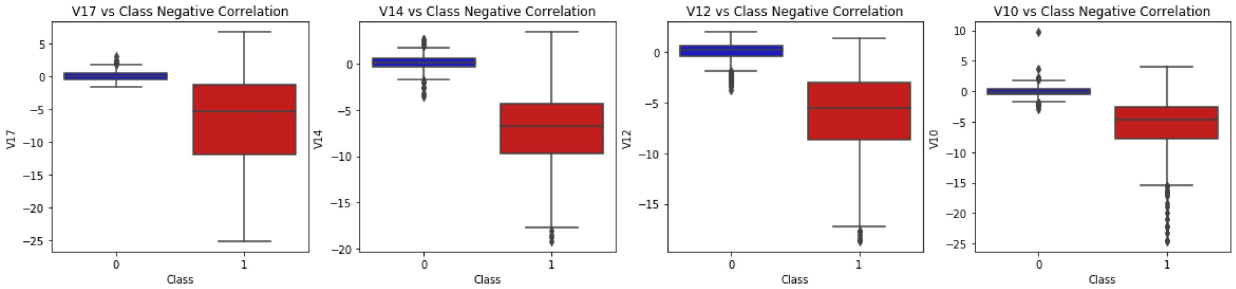
Note: It's essential to use the subsample in our correlation matrix to avoid the impact of high class imbalance between our classes. This imbalance results from the original dataframe.

In [13]:
```
# Make sure we use the subsample in our correlation

f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

# Entire DataFrame
corr = df.corr()
sns.heatmap(corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax1)
ax1.set_title("Imbalanced Correlation Matrix \n (don't use for reference)", fontsize=1


sub_sample_corr = new_df.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax2)
```

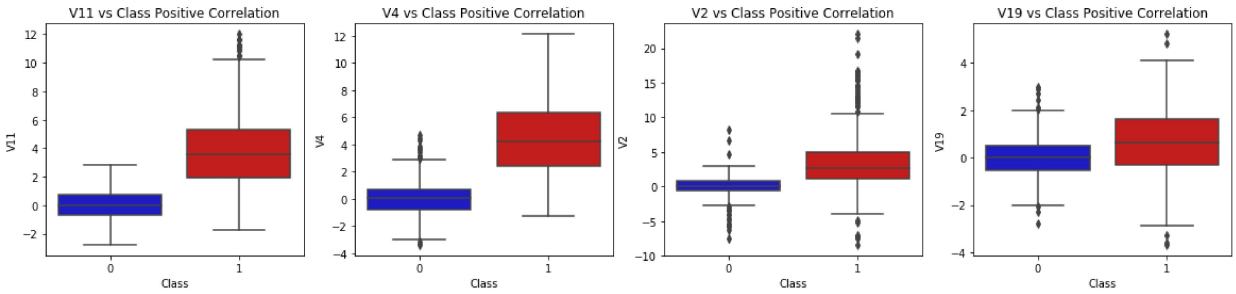
```
ax2.set_title('SubSample Correlation Matrix \n (use for reference)', fontsize=14)
plt.show()
```



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

# "Negative Correlations with our Class: A lower feature value is indicative of a high
sns.boxplot(x="Class", y="V17", data=new_df, palette=colors, ax=axes[0])
axes[0].set_title('V17 vs Class Negative Correlation')

sns.boxplot(x="Class", y="V14", data=new_df, palette=colors, ax=axes[1])
axes[1].set_title('V14 vs Class Negative Correlation')


sns.boxplot(x="Class", y="V12", data=new_df, palette=colors, ax=axes[2])
axes[2].set_title('V12 vs Class Negative Correlation')


sns.boxplot(x="Class", y="V10", data=new_df, palette=colors, ax=axes[3])
axes[3].set_title('V10 vs Class Negative Correlation')

plt.show()
```



In [15]:
```
f, axes = plt.subplots(ncols=4, figsize=(20,4))

# Positive Correlations: When a feature value is higher, the probability of the transa
sns.boxplot(x="Class", y="V11", data=new_df, palette=colors, ax=axes[0])
axes[0].set_title('V11 vs Class Positive Correlation')

sns.boxplot(x="Class", y="V4", data=new_df, palette=colors, ax=axes[1])
axes[1].set_title('V4 vs Class Positive Correlation')


sns.boxplot(x="Class", y="V2", data=new_df, palette=colors, ax=axes[2])
axes[2].set_title('V2 vs Class Positive Correlation')


sns.boxplot(x="Class", y="V19", data=new_df, palette=colors, ax=axes[3])
axes[3].set_title('V19 vs Class Positive Correlation')

plt.show()
```



Anomaly Detection:

The primary goal in this section is to remove 'extreme outliers' from features with a strong correlation to our classes, thereby improving our model's accuracy.

Interquartile Range Method:

Interquartile Range (IQR): We calculate this as the difference between the 75th percentile and the 25th percentile. Our goal is to establish a threshold beyond these percentiles, so that any instance surpassing this threshold can be removed.

Boxplots: Boxplots not only provide a visual representation of the 25th and 75th percentiles (the ends of the boxes) but also make it easy to identify extreme outliers (data points beyond the lower and upper extremes).

Outlier Removal Tradeoff:

We need to carefully choose the threshold for removing outliers. This threshold is determined by multiplying a number (e.g., 1.5) by the Interquartile Range (IQR). The higher the threshold, the fewer outliers will be detected (multiplying by a higher number, e.g., 3), while a lower threshold will identify more outliers.

The Tradeoff: A lower threshold removes more outliers, but we want to focus on 'extreme outliers' rather than just outliers. This is because an excessively low threshold may lead to information loss, resulting in lower model accuracy. Experimenting with different thresholds can help us understand their impact on the accuracy of our classification models.

Summary:

Visualize Distributions: We start by visualizing the distribution of the feature we plan to use for removing some outliers. Among features V14, V12, and V10, only V14 exhibits a Gaussian distribution.

Determining the Threshold: Once we've chosen the multiplier for the IQR (with a lower value removing more outliers), we calculate the upper and lower thresholds by subtracting q25 - threshold (the lower extreme threshold) and adding q75 + threshold (the upper extreme threshold).

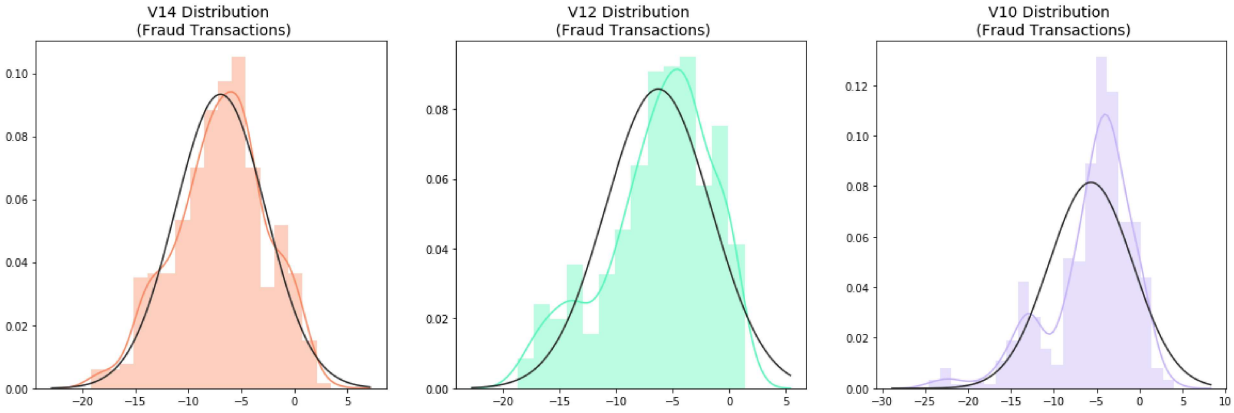
Conditional Dropping: We create a conditional rule: if the "threshold" is exceeded at both extremes, the instances are removed.

Boxplot Representation: We use boxplots to visually confirm that the number of "extreme outliers" has been significantly reduced.

Note: Implementing outlier reduction has improved our accuracy by over 3%. While outliers can affect model accuracy, we must strike a balance to avoid excessive information loss, which can lead to underfitting.

In [16]:
```
from scipy.stats import norm

f, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(20, 6))

v14_fraud_dist = new_df['V14'].loc[new_df['Class'] == 1].values
sns.distplot(v14_fraud_dist,ax=ax1, fit=norm, color='#FB8861')
ax1.set_title('V14 Distribution \n (Fraud Transactions)', fontsize=14)

v12_fraud_dist = new_df['V12'].loc[new_df['Class'] == 1].values
sns.distplot(v12_fraud_dist,ax=ax2, fit=norm, color='#56F9BB')
ax2.set_title('V12 Distribution \n (Fraud Transactions)', fontsize=14)


v10_fraud_dist = new_df['V10'].loc[new_df['Class'] == 1].values
sns.distplot(v10_fraud_dist,ax=ax3, fit=norm, color='#C5B3F9')
ax3.set_title('V10 Distribution \n (Fraud Transactions)', fontsize=14)
```

```
plt.show()
```



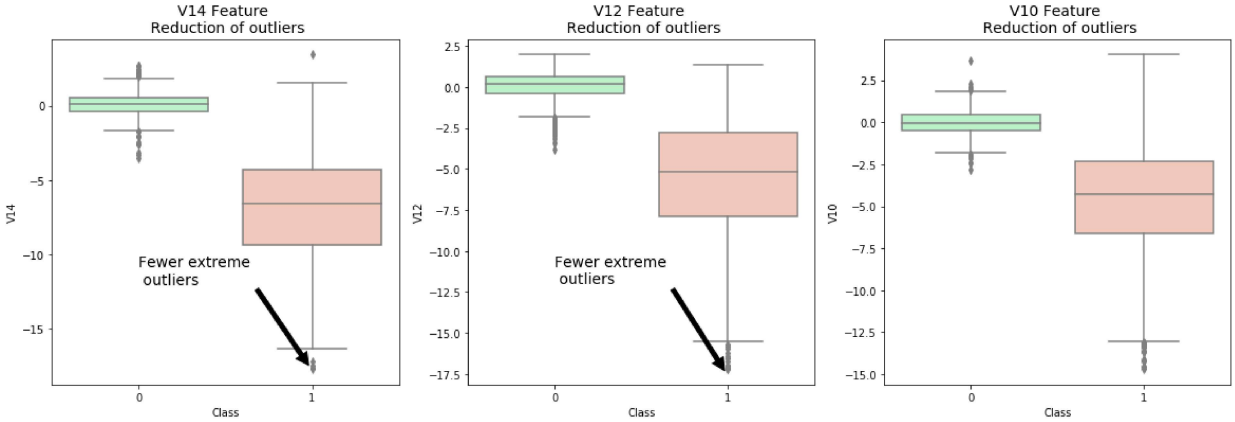
In [17]:
```
# # -----> V14 Removing Outliers (Highest Negative Correlated with Labels)
v14_fraud = new_df['V14'].loc[new_df['Class'] == 1].values
q25, q75 = np.percentile(v14_fraud, 25), np.percentile(v14_fraud, 75)
print('Quartile 25: {} | Quartile 75: {}'.format(q25, q75))
v14_iqr = q75 - q25
print('iqr: {}'.format(v14_iqr))

v14_cut_off = v14_iqr * 1.5
v14_lower, v14_upper = q25 - v14_cut_off, q75 + v14_cut_off
print('Cut Off: {}'.format(v14_cut_off))
print('V14 Lower: {}'.format(v14_lower))
print('V14 Upper: {}'.format(v14_upper))

outliers = [x for x in v14_fraud if x < v14_lower or x > v14_upper]
print('Feature V14 Outliers for Fraud Cases: {}'.format(len(outliers)))
print('V10 outliers:{}'.format(outliers))

new_df = new_df.drop(new_df[(new_df['V14'] > v14_upper) | (new_df['V14'] < v14_lower)]
print('----' * 44)

# -----> V12 removing outliers from fraud transactions
v12_fraud = new_df['V12'].loc[new_df['Class'] == 1].values
q25, q75 = np.percentile(v12_fraud, 25), np.percentile(v12_fraud, 75)
v12_iqr = q75 - q25

v12_cut_off = v12_iqr * 1.5
v12_lower, v12_upper = q25 - v12_cut_off, q75 + v12_cut_off
print('V12 Lower: {}'.format(v12_lower))
print('V12 Upper: {}'.format(v12_upper))
outliers = [x for x in v12_fraud if x < v12_lower or x > v12_upper]
print('V12 outliers: {}'.format(outliers))
print('Feature V12 Outliers for Fraud Cases: {}'.format(len(outliers)))
new_df = new_df.drop(new_df[(new_df['V12'] > v12_upper) | (new_df['V12'] < v12_lower)]
print('Number of Instances after outliers removal: {}'.format(len(new_df)))
print('----' * 44)


# Removing outliers V10 Feature
v10_fraud = new_df['V10'].loc[new_df['Class'] == 1].values
q25, q75 = np.percentile(v10_fraud, 25), np.percentile(v10_fraud, 75)
v10_iqr = q75 - q25

v10_cut_off = v10_iqr * 1.5
```

```
v10_lower, v10_upper = q25 - v10_cut_off, q75 + v10_cut_off
print('V10 Lower: {}'.format(v10_lower))
print('V10 Upper: {}'.format(v10_upper))
outliers = [x for x in v10_fraud if x < v10_lower or x > v10_upper]
print('V10 outliers: {}'.format(outliers))
print('Feature V10 Outliers for Fraud Cases: {}'.format(len(outliers)))
new_df = new_df.drop(new_df[(new_df['V10'] > v10_upper) | (new_df['V10'] < v10_lower)]
print('Number of Instances after outliers removal: {}'.format(len(new_df)))
```

```
Quartile 25: -9.692722964972385 | Quartile 75: -4.282820849486866
iqr: 5.409902115485519
Cut Off: 8.114853173228278
V14 Lower: -17.807576138200663
V14 Upper: 3.8320323237414122
Feature V14 Outliers for Fraud Cases: 4
V10 outliers:[-18.049997689859396, -18.8220867423816, -19.2143254902614, -18.49377335
51053]
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
------
V12 Lower: -17.3430371579634
V12 Upper: 5.776973384895937
V12 outliers: [-18.683714633344298, -18.4311310279993, -18.553697009645802, -18.04759
6570821604]
Feature V12 Outliers for Fraud Cases: 4
Number of Instances after outliers removal: 976
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
------
V10 Lower: -14.89885463232024
V10 Upper: 4.920334958342141
V10 outliers: [-15.2399619587112, -16.6011969664137, -17.141513641289198, -15.2318333
653018, -19.836148851696, -16.6496281595399, -15.1237521803455, -24.5882624372475, -1
5.563791338730098, -14.9246547735487, -22.1870885620007, -22.1870885620007, -15.12416
2814494698, -14.9246547735487, -22.1870885620007, -15.346098846877501, -15.5637913387
30098, -18.2711681738888, -15.2399619587112, -24.403184969972802, -16.7460441053944,
-16.2556117491401, -18.9132433348732, -22.1870885620007, -20.949191554361104, -16.303
5376590131, -23.2282548357516]
Feature V10 Outliers for Fraud Cases: 27
Number of Instances after outliers removal: 948
```

In [18]:
```
f,(ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,6))

colors = ['#B3F9C5', '#f9c5b3']
# Boxplots with outliers removed
# Feature V14
sns.boxplot(x="Class", y="V14", data=new_df,ax=ax1, palette=colors)
ax1.set_title("V14 Feature \n Reduction of outliers", fontsize=14)
ax1.annotate('Fewer extreme \n outliers', xy=(0.98, -17.5), xytext=(0, -12),
            arrowprops=dict(facecolor='black'),
            fontsize=14)

# Feature 12
sns.boxplot(x="Class", y="V12", data=new_df, ax=ax2, palette=colors)
ax2.set_title("V12 Feature \n Reduction of outliers", fontsize=14)
ax2.annotate('Fewer extreme \n outliers', xy=(0.98, -17.3), xytext=(0, -12),
            arrowprops=dict(facecolor='black'),
            fontsize=14)

# Feature V10
```

```
sns.boxplot(x="Class", y="V10", data=new_df, ax=ax3, palette=colors)
ax3.set_title("V10 Feature \n Reduction of outliers", fontsize=14)
ax3.annotate('Fewer extreme \n outliers', xy=(0.95, -16.5), xytext=(0, -12),
             arrowprops=dict(facecolor='black'),
             fontsize=14)


plt.show()
```



Dimensionality Reduction and Clustering:

Understanding t-SNE:

To grasp this algorithm, you need to be familiar with the following terms:

Euclidean Distance

Conditional Probability

Normal and T-Distribution Plots

Summary:

The t-SNE algorithm effectively clusters fraud and non-fraud cases in our dataset.

Despite the small subsample, t-SNE accurately detects clusters in various scenarios (dataset shuffling).

This suggests that predictive models will likely perform well in distinguishing fraud from non-fraud cases.

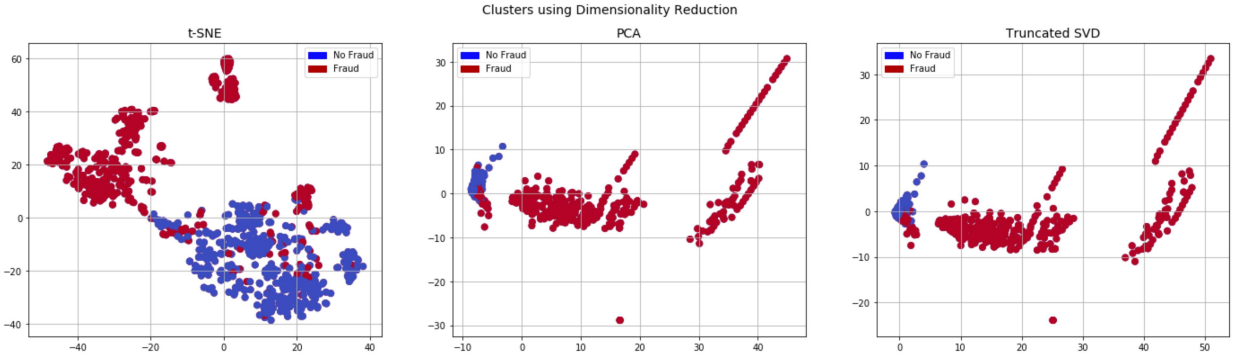
In [19]:
```
# New_df is from the random undersample data (fewer instances)
X = new_df.drop('Class', axis=1)
y = new_df['Class']


# T-SNE Implementation
t0 = time.time()
X_reduced_tsne = TSNE(n_components=2, random_state=42).fit_transform(X.values)
t1 = time.time()
```

```
print("T-SNE took {:.2} s".format(t1 - t0))

# PCA Implementation
t0 = time.time()
X_reduced_pca = PCA(n_components=2, random_state=42).fit_transform(X.values)
t1 = time.time()
print("PCA took {:.2} s".format(t1 - t0))

# TruncatedSVD
t0 = time.time()
X_reduced_svd = TruncatedSVD(n_components=2, algorithm='randomized', random_state=42).
t1 = time.time()
print("Truncated SVD took {:.2} s".format(t1 - t0))
```

```
T-SNE took 3.9 s
PCA took 0.019 s
Truncated SVD took 0.0026 s
```

In [20]:

```
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24,6))
# labels = ['No Fraud', 'Fraud']
f.suptitle('Clusters using Dimensionality Reduction', fontsize=14)


blue_patch = mpatches.Patch(color='#0A0AFF', label='No Fraud')
red_patch = mpatches.Patch(color='#AF0000', label='Fraud')


# t-SNE scatter plot
ax1.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], c=(y == 0), cmap='coolwarm', lab
ax1.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], c=(y == 1), cmap='coolwarm', lab
ax1.set_title('t-SNE', fontsize=14)

ax1.grid(True)

ax1.legend(handles=[blue_patch, red_patch])


# PCA scatter plot
ax2.scatter(X_reduced_pca[:,0], X_reduced_pca[:,1], c=(y == 0), cmap='coolwarm', label
ax2.scatter(X_reduced_pca[:,0], X_reduced_pca[:,1], c=(y == 1), cmap='coolwarm', label
ax2.set_title('PCA', fontsize=14)

ax2.grid(True)

ax2.legend(handles=[blue_patch, red_patch])

# TruncatedSVD scatter plot
ax3.scatter(X_reduced_svd[:,0], X_reduced_svd[:,1], c=(y == 0), cmap='coolwarm', label
ax3.scatter(X_reduced_svd[:,0], X_reduced_svd[:,1], c=(y == 1), cmap='coolwarm', label
ax3.set_title('Truncated SVD', fontsize=14)

ax3.grid(True)

ax3.legend(handles=[blue_patch, red_patch])

plt.show()
```



Classifiers (UnderSampling):

Classifiers:

In this section, we will train four types of classifiers to determine which one is most effective in detecting fraud transactions. Before that, we will split our data into training and testing sets and separate the features from the labels.

Summary:

Logistic Regression is generally more accurate than the other three classifiers, which we will further analyze. We use GridSearchCV to find the best parameters for the classifiers. Logistic Regression achieves the best Receiving Operating Characteristic score (ROC), indicating its accuracy in distinguishing fraud and non-fraud transactions. Learning Curves:

A wider gap between the training and cross-validation scores suggests overfitting (high variance). If the score is low in both training and cross-validation sets, it indicates underfitting (high bias). The Logistic Regression Classifier shows the best score in both training and cross-validation sets.

In [21]:
```
# Undersampling before cross validating (prone to overfit)
X = new_df.drop('Class', axis=1)
y = new_df['Class']
```

In [22]:
```
# Our data is already scaled we should split our training and test sets
from sklearn.model_selection import train_test_split

# 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=
```

In [23]:
```
# 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
```

In [24]:
```
# Let's implement simple classifiers

classifiers = {
    "LogisiticRegression": LogisticRegression(),
    "KNearest": KNeighborsClassifier(),
```

```
    "Support Vector Classifier": SVC(),
    "DecisionTreeClassifier": DecisionTreeClassifier()
}
```

In [25]:
```
# our scores are getting even high scores even when applying cross validation.
from sklearn.model_selection import cross_val_score


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__, "Has a training score of", r
```

```
Classifiers:  LogisticRegression Has a training score of 94.0 % accuracy score
Classifiers:  KNeighborsClassifier Has a training score of 93.0 % accuracy score
Classifiers:  SVC Has a training score of 92.0 % accuracy score
Classifiers:  DecisionTreeClassifier Has a training score of 90.0 % accuracy score
```

In [26]:
```
# Use GridSearchCV to find the best parameters.
from sklearn.model_selection import GridSearchCV


# Logistic Regression
log_reg_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}



grid_log_reg = GridSearchCV(LogisticRegression(), log_reg_params)
grid_log_reg.fit(X_train, y_train)
# We automatically get the logistic regression with the best parameters.
log_reg = grid_log_reg.best_estimator_

knears_params = {"n_neighbors": list(range(2,5,1)), 'algorithm': ['auto', 'ball_tree',

grid_knears = GridSearchCV(KNeighborsClassifier(), knears_params)
grid_knears.fit(X_train, y_train)
# KNears best estimator
knears_neighbors = grid_knears.best_estimator_

# Support Vector Classifier
svc_params = {'C': [0.5, 0.7, 0.9, 1], 'kernel': ['rbf', 'poly', 'sigmoid', 'linear']}
grid_svc = GridSearchCV(SVC(), svc_params)
grid_svc.fit(X_train, y_train)

# SVC best estimator
svc = grid_svc.best_estimator_

# DecisionTree Classifier
tree_params = {"criterion": ["gini", "entropy"], "max_depth": list(range(2,4,1)),
              "min_samples_leaf": list(range(5,7,1))}
grid_tree = GridSearchCV(DecisionTreeClassifier(), tree_params)
grid_tree.fit(X_train, y_train)

# tree best estimator
tree_clf = grid_tree.best_estimator_
```

In [27]:
```
# 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


knears_score = cross_val_score(knears_neighbors, X_train, y_train, cv=5)
print('Knears Neighbors Cross Validation Score', round(knears_score.mean() * 100, 2).a

svc_score = cross_val_score(svc, X_train, y_train, cv=5)
print('Support Vector Classifier Cross Validation Score', round(svc_score.mean() * 100

tree_score = cross_val_score(tree_clf, X_train, y_train, cv=5)
print('DecisionTree Classifier Cross Validation Score', round(tree_score.mean() * 100,
```

```
Logistic Regression Cross Validation Score:  93.4%
Knears Neighbors Cross Validation Score 92.6%
Support Vector Classifier Cross Validation Score 93.27%
DecisionTree Classifier Cross Validation Score 91.29%
```

In [28]:

```
# We will undersample during cross validating
undersample_X = df.drop('Class', axis=1)
undersample_y = df['Class']

for train_index, test_index in sss.split(undersample_X, undersample_y):
    print("Train:", train_index, "Test:", test_index)
    undersample_Xtrain, undersample_Xtest = undersample_X.iloc[train_index], undersamp
    undersample_ytrain, undersample_ytest = undersample_y.iloc[train_index], undersamp

undersample_Xtrain = undersample_Xtrain.values
undersample_Xtest = undersample_Xtest.values
undersample_ytrain = undersample_ytrain.values
undersample_ytest = undersample_ytest.values

undersample_accuracy = []
undersample_precision = []
undersample_recall = []
undersample_f1 = []
undersample_auc = []

# Implementing NearMiss Technique
# Distribution of NearMiss (Just to see how it distributes the labels we won't use the
X_nearmiss, y_nearmiss = NearMiss().fit_sample(undersample_X.values, undersample_y.val
print('NearMiss Label Distribution: {}'.format(Counter(y_nearmiss)))
# Cross Validating the right way

for train, test in sss.split(undersample_Xtrain, undersample_ytrain):
    undersample_pipeline = imbalanced_make_pipeline(NearMiss(sampling_strategy='majori
    undersample_model = undersample_pipeline.fit(undersample_Xtrain[train], undersampl
    undersample_prediction = undersample_model.predict(undersample_Xtrain[test])

    undersample_accuracy.append(undersample_pipeline.score(original_Xtrain[test], orig
    undersample_precision.append(precision_score(original_ytrain[test], undersample_pr
    undersample_recall.append(recall_score(original_ytrain[test], undersample_predicti
    undersample_f1.append(f1_score(original_ytrain[test], undersample_prediction))
    undersample_auc.append(roc_auc_score(original_ytrain[test], undersample_predictior
```

```
Train: [ 56957  56958  56959 ... 284804 284805 284806] Test: [    0     1     2 ... 5
8507 58694 59233]
Train: [     0      1      2 ... 284804 284805 284806] Test: [ 56957  56958  56959
... 113921 114217 114467]
Train: [     0      1      2 ... 284804 284805 284806] Test: [113922 113923 113924
... 171307 171698 171735]
Train: [     0      1      2 ... 284804 284805 284806] Test: [170882 170883 170884
... 228538 229221 229591]
Train: [     0      1      2 ... 228538 229221 229591] Test: [227842 227843 227844
... 284804 284805 284806]
NearMiss Label Distribution: Counter({0: 492, 1: 492})
```

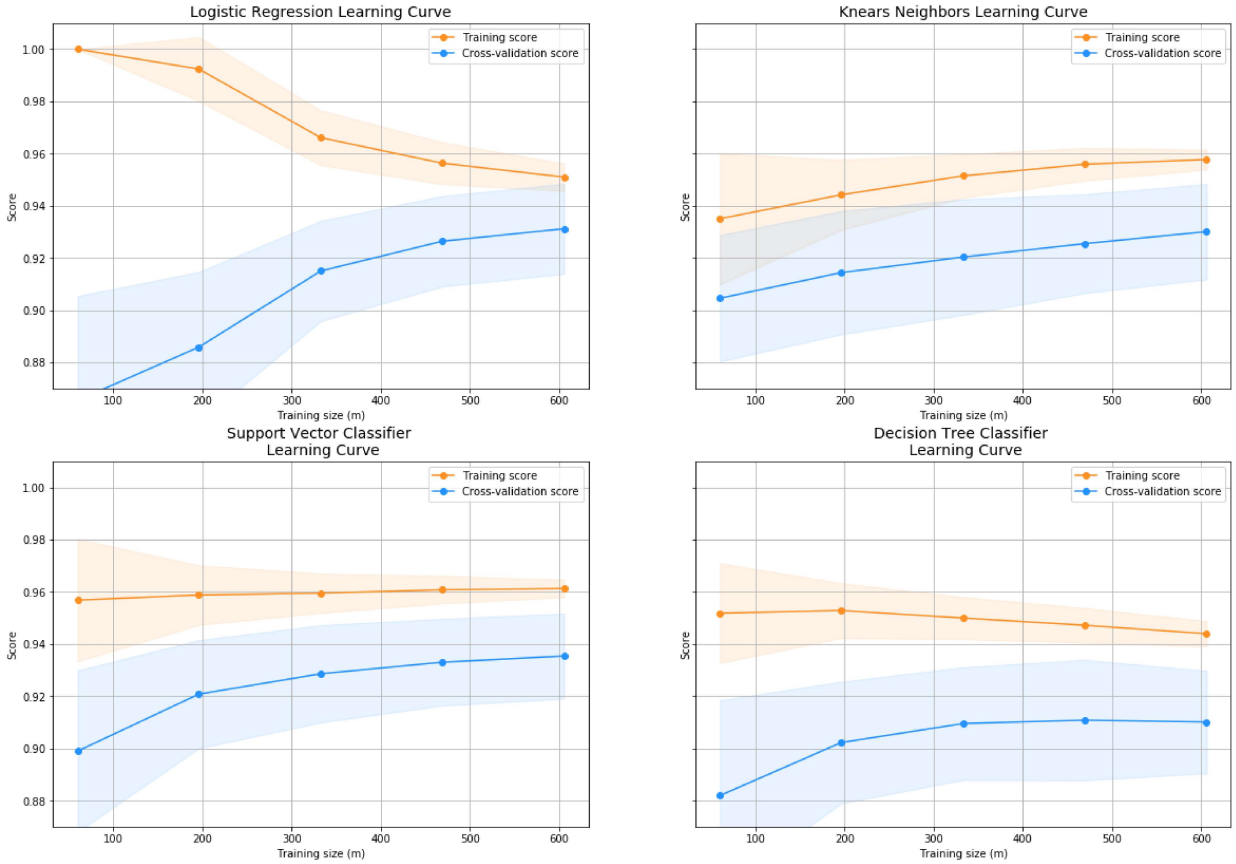
In [29]:

```
# Let's Plot LogisticRegression Learning Curve
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import learning_curve

def plot_learning_curve(estimator1, estimator2, estimator3, estimator4, X, y, ylim=Nor
                        n_jobs=1, train_sizes=np.linspace(.1, 1.0, 5)):
    f, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2,2, figsize=(20,14), sharey=True)
    if ylim is not None:
        plt.ylim(*ylim)
    # First Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator1, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax1.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
    ax1.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="#2492ff")
    ax1.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax1.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax1.set_title("Logistic Regression Learning Curve", fontsize=14)
    ax1.set_xlabel('Training size (m)')
    ax1.set_ylabel('Score')
    ax1.grid(True)
    ax1.legend(loc="best")

    # Second Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator2, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax2.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
    ax2.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="#2492ff")
    ax2.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax2.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax2.set_title("Knears Neighbors Learning Curve", fontsize=14)
```

```
    ax2.set_xlabel('Training size (m)')
    ax2.set_ylabel('Score')
    ax2.grid(True)
    ax2.legend(loc="best")

    # Third Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator3, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax3.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
    ax3.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="#2492ff")
    ax3.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax3.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax3.set_title("Support Vector Classifier \n Learning Curve", fontsize=14)
    ax3.set_xlabel('Training size (m)')
    ax3.set_ylabel('Score')
    ax3.grid(True)
    ax3.legend(loc="best")

    # Fourth Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator4, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax4.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
    ax4.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="#2492ff")
    ax4.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax4.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax4.set_title("Decision Tree Classifier \n Learning Curve", fontsize=14)
    ax4.set_xlabel('Training size (m)')
    ax4.set_ylabel('Score')
    ax4.grid(True)
    ax4.legend(loc="best")
    return plt
```

In [30]:
```
cv = ShuffleSplit(n_splits=100, test_size=0.2, random_state=42)
plot_learning_curve(log_reg, knears_neighbors, svc, tree_clf, X_train, y_train, (0.87,
```

Out[30]:
```
<module 'matplotlib.pyplot' from '/opt/conda/lib/python3.6/site-packages/matplotlib/p
yplot.py'>
```



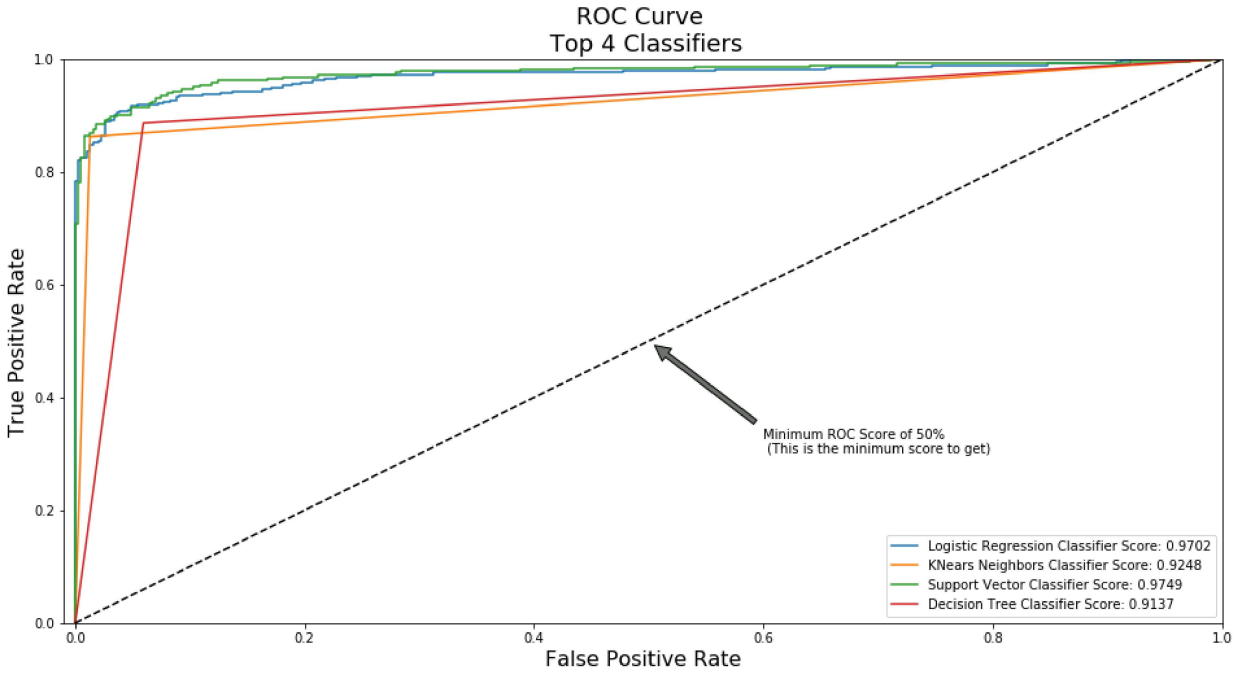
In [31]:
```
from sklearn.metrics import roc_curve
from sklearn.model_selection import cross_val_predict
# Create a DataFrame with all the scores and the classifiers names.

log_reg_pred = cross_val_predict(log_reg, X_train, y_train, cv=5,
                                 method="decision_function")

knears_pred = cross_val_predict(knears_neighbors, X_train, y_train, cv=5)

svc_pred = cross_val_predict(svc, X_train, y_train, cv=5,
                             method="decision_function")

tree_pred = cross_val_predict(tree_clf, X_train, y_train, cv=5)
```

In [32]:
```
from sklearn.metrics import roc_auc_score

print('Logistic Regression: ', roc_auc_score(y_train, log_reg_pred))
print('KNears Neighbors: ', roc_auc_score(y_train, knears_pred))
print('Support Vector Classifier: ', roc_auc_score(y_train, svc_pred))
print('Decision Tree Classifier: ', roc_auc_score(y_train, tree_pred))
```

```
Logistic Regression:  0.9701971764279793
KNears Neighbors:  0.9248068980407725
Support Vector Classifier:  0.9749263461417915
Decision Tree Classifier:  0.9136804641411925
```

In [33]:
```
log_fpr, log_tpr, log_thresold = roc_curve(y_train, log_reg_pred)
knear_fpr, knear_tpr, knear_threshold = roc_curve(y_train, knears_pred)
svc_fpr, svc_tpr, svc_threshold = roc_curve(y_train, svc_pred)
tree_fpr, tree_tpr, tree_threshold = roc_curve(y_train, tree_pred)
```

```
def graph_roc_curve_multiple(log_fpr, log_tpr, knear_fpr, knear_tpr, svc_fpr, svc_tpr,
    plt.figure(figsize=(16,8))
    plt.title('ROC Curve \n Top 4 Classifiers', fontsize=18)
    plt.plot(log_fpr, log_tpr, label='Logistic Regression Classifier Score: {:.4f}'.fo
    plt.plot(knear_fpr, knear_tpr, label='KNears Neighbors Classifier Score: {:.4f}'.f
    plt.plot(svc_fpr, svc_tpr, label='Support Vector Classifier Score: {:.4f}'.format(
    plt.plot(tree_fpr, tree_tpr, label='Decision Tree Classifier Score: {:.4f}'.format
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.01, 1, 0, 1])
    plt.xlabel('False Positive Rate', fontsize=16)
    plt.ylabel('True Positive Rate', fontsize=16)
    plt.annotate('Minimum ROC Score of 50% \n (This is the minimum score to get)', xy=
                arrowprops=dict(facecolor='#6E726D', shrink=0.05),
                )
    plt.legend()

graph_roc_curve_multiple(log_fpr, log_tpr, knear_fpr, knear_tpr, svc_fpr, svc_tpr, tre
plt.show()
```



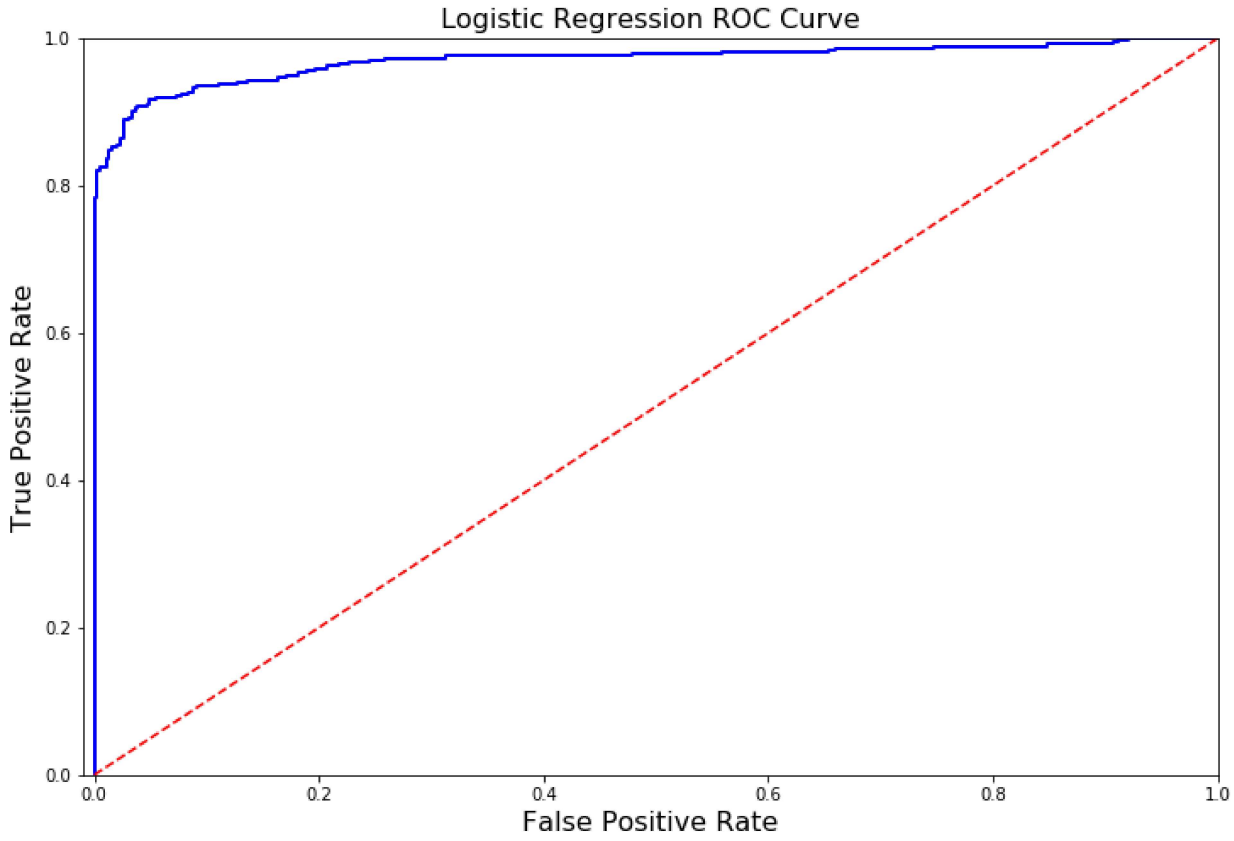
LogisticRegression:

Terms:

True Positives: Transactions correctly classified as fraud. False Positives: Transactions incorrectly classified as fraud. True Negatives: Non-fraudulent transactions correctly classified. False Negatives: Non-fraudulent transactions incorrectly classified. Precision: True Positives / (True Positives + False Positives) Recall: True Positives / (True Positives + False Negatives) Precision measures how precise or certain our model is in detecting fraud, while recall indicates the number of fraud cases our model can detect. Precision/Recall Tradeoff: A more precise model will detect fewer cases. For instance, if our model has a precision of 95%, it will confidently identify some cases as fraud. However, if we lower the precision threshold, the model can detect more cases. Summary:

Precision begins to decrease between 0.90 and 0.92. Nevertheless, our precision score remains high, and our recall score is still decent.

In [34]:
```
def logistic_roc_curve(log_fpr, log_tpr):
    plt.figure(figsize=(12,8))
    plt.title('Logistic Regression ROC Curve', fontsize=16)
    plt.plot(log_fpr, log_tpr, 'b-', linewidth=2)
    plt.plot([0, 1], [0, 1], 'r--')
    plt.xlabel('False Positive Rate', fontsize=16)
    plt.ylabel('True Positive Rate', fontsize=16)
    plt.axis([-0.01,1,0,1])


logistic_roc_curve(log_fpr, log_tpr)
plt.show()
```



In [35]:
```
from sklearn.metrics import precision_recall_curve

precision, recall, threshold = precision_recall_curve(y_train, log_reg_pred)
```

In [36]:
```
from sklearn.metrics import recall_score, precision_score, f1_score, accuracy_score
y_pred = log_reg.predict(X_train)

# Overfitting Case
print('---' * 45)
print('Overfitting: \n')
print('Recall Score: {:.2f}'.format(recall_score(y_train, y_pred)))
print('Precision Score: {:.2f}'.format(precision_score(y_train, y_pred)))
print('F1 Score: {:.2f}'.format(f1_score(y_train, y_pred)))
print('Accuracy Score: {:.2f}'.format(accuracy_score(y_train, y_pred)))
print('---' * 45)
```

```
# How it should look like
print('---' * 45)
print('How it should be:\n')
print("Accuracy Score: {:.2f}".format(np.mean(undersample_accuracy)))
print("Precision Score: {:.2f}".format(np.mean(undersample_precision)))
print("Recall Score: {:.2f}".format(np.mean(undersample_recall)))
print("F1 Score: {:.2f}".format(np.mean(undersample_f1)))
print('---' * 45)
```

```
---------------------------------------------------------------------------------------------
-------------------------------------------------
Overfitting:

Recall Score: 0.92
Precision Score: 0.71
F1 Score: 0.81
Accuracy Score: 0.78
---------------------------------------------------------------------------------------------
-------------------------------------------------
---------------------------------------------------------------------------------------------
-------------------------------------------------
How it should be:

Accuracy Score: 0.60
Precision Score: 0.00
Recall Score: 0.43
F1 Score: 0.00
---------------------------------------------------------------------------------------------
-------------------------------------------------
```

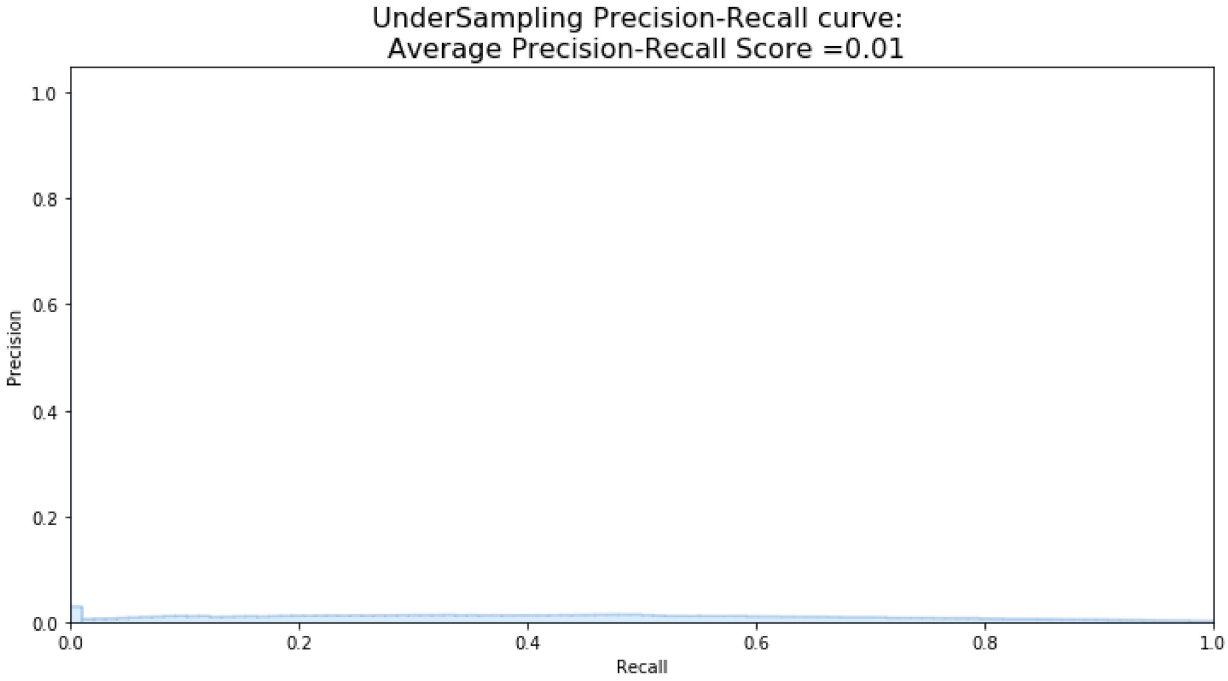
In [37]:
```
undersample_y_score = log_reg.decision_function(original_Xtest)
```

In [38]:
```
from sklearn.metrics import average_precision_score

undersample_average_precision = average_precision_score(original_ytest, undersample_y_

print('Average precision-recall score: {0:0.2f}'.format(
      undersample_average_precision))
```

```
Average precision-recall score: 0.01
```

In [39]:
```
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12,6))

precision, recall, _ = precision_recall_curve(original_ytest, undersample_y_score)

plt.step(recall, precision, color='#004a93', alpha=0.2,
         where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,
                 color='#48a6ff')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('UnderSampling Precision-Recall curve: \n Average Precision-Recall Score ={0
          undersample_average_precision), fontsize=16)
```

Out[39]: Text(0.5, 1.0, 'UnderSampling Precision-Recall curve: \n Average Precision-Recall Score =0.01')



SMOTE Technique (Over-Sampling):

SMOTE stands for Synthetic Minority Over-sampling Technique. Unlike Random UnderSampling, SMOTE creates new synthetic points to balance the classes and is an alternative for addressing class imbalance problems.

Understanding SMOTE:

Solving the Class Imbalance: SMOTE generates synthetic points from the minority class to achieve balance with the majority class. Location of the Synthetic Points: SMOTE selects distances between the closest neighbors of the minority class to create synthetic points within those distances. Final Effect: Unlike random undersampling, SMOTE retains more information since no rows are deleted. Accuracy vs. Time Tradeoff: SMOTE is likely to be more accurate than random undersampling but requires more time to train due to the absence of row deletion.

Overfitting during Cross Validation:

In our undersample analysis, I want to highlight a common mistake – performing undersampling or oversampling before cross-validation. This approach can lead to 'data leakage' issues, where you influence the validation set before applying cross-validation, resulting in overfitting.

As previously mentioned, it's crucial to avoid this by creating synthetic data points 'during' cross-validation rather than before. When using cross-validation, a portion of the dataset serves as the validation set, and the test set should remain untouched. This ensures a more accurate evaluation of your model's performance.

In [40]:
```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, RandomizedSearchCV


print('Length of X (train): {} | Length of y (train): {}'.format(len(original_Xtrain),
print('Length of X (test): {} | Length of y (test): {}'.format(len(original_Xtest), le

# List to append the score and then find the average
accuracy_lst = []
precision_lst = []
recall_lst = []
f1_lst = []
auc_lst = []

# Classifier with optimal parameters
# log_reg_sm = grid_log_reg.best_estimator_
log_reg_sm = LogisticRegression()




rand_log_reg = RandomizedSearchCV(LogisticRegression(), log_reg_params, n_iter=4)


# Implementing SMOTE Technique
# Cross Validating the right way
# Parameters
log_reg_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}
for train, test in sss.split(original_Xtrain, original_ytrain):
    pipeline = imbalanced_make_pipeline(SMOTE(sampling_strategy='minority'), rand_log_
    model = pipeline.fit(original_Xtrain[train], original_ytrain[train])
    best_est = rand_log_reg.best_estimator_
    prediction = best_est.predict(original_Xtrain[test])

    accuracy_lst.append(pipeline.score(original_Xtrain[test], original_ytrain[test]))
    precision_lst.append(precision_score(original_ytrain[test], prediction))
    recall_lst.append(recall_score(original_ytrain[test], prediction))
    f1_lst.append(f1_score(original_ytrain[test], prediction))
    auc_lst.append(roc_auc_score(original_ytrain[test], prediction))

print('---' * 45)
print('')
print("accuracy: {}".format(np.mean(accuracy_lst)))
print("precision: {}".format(np.mean(precision_lst)))
print("recall: {}".format(np.mean(recall_lst)))
print("f1: {}".format(np.mean(f1_lst)))
print('---' * 45)
```

```
Length of X (train): 227846 | Length of y (train): 227846
Length of X (test): 56961 | Length of y (test): 56961
---------------------------------------------------------------------------------
-------------------------------------------------

accuracy: 0.9604033631543759
precision: 0.058698042961212046
recall: 0.9136319376825707
f1: 0.10895012225437542
---------------------------------------------------------------------------------
-------------------------------------------------
```

In [41]:
```
labels = ['No Fraud', 'Fraud']
smote_prediction = best_est.predict(original_Xtest)
print(classification_report(original_ytest, smote_prediction, target_names=labels))
```

```
              precision    recall  f1-score   support

    No Fraud       1.00      0.98      0.99     56863
       Fraud       0.06      0.87      0.11        98

    accuracy                           0.98     56961
   macro avg       0.53      0.92      0.55     56961
weighted avg       1.00      0.98      0.99     56961
```

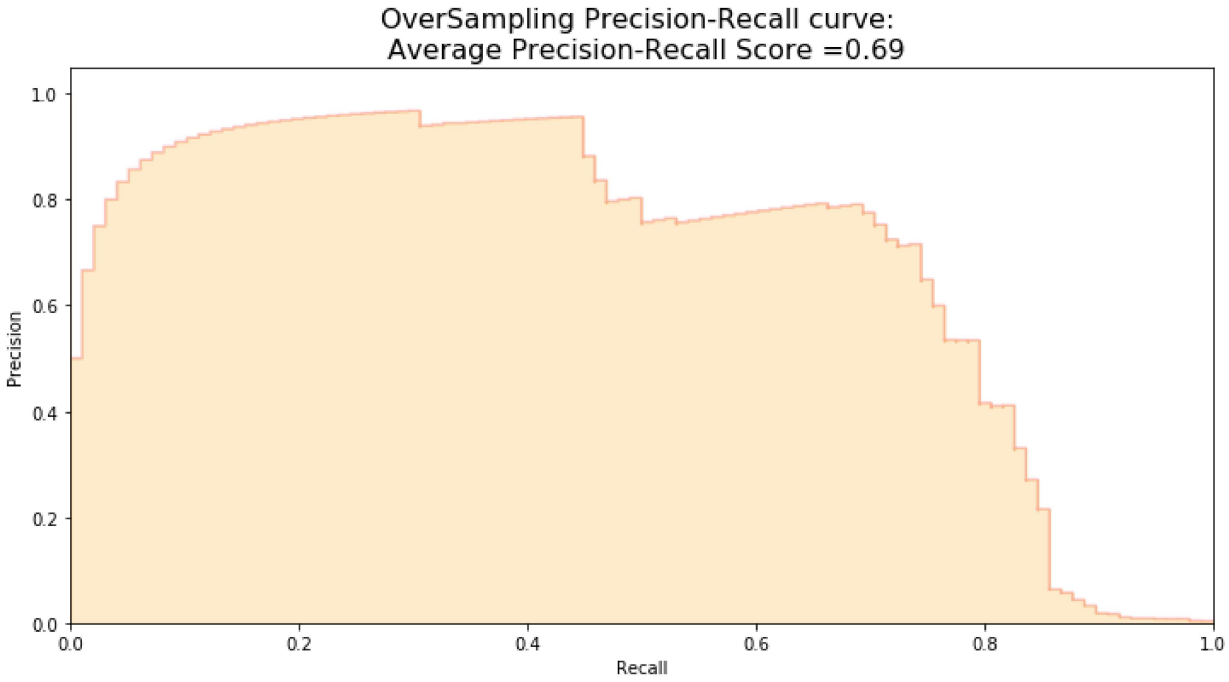
In [42]:
```
y_score = best_est.decision_function(original_Xtest)
```

In [43]:
```
average_precision = average_precision_score(original_ytest, y_score)

print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))
```

```
Average precision-recall score: 0.69
```

In [44]:
```
fig = plt.figure(figsize=(12,6))

precision, recall, _ = precision_recall_curve(original_ytest, y_score)

plt.step(recall, precision, color='r', alpha=0.2,
         where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,
                 color='#F59B00')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('OverSampling Precision-Recall curve: \n Average Precision-Recall Score ={0:
          average_precision), fontsize=16)
```

Out[44]:
```
Text(0.5, 1.0, 'OverSampling Precision-Recall curve: \n Average Precision-Recall Scor
e =0.69')
```



In [45]:
```
# SMOTE Technique (OverSampling) After splitting and Cross Validating
sm = SMOTE(ratio='minority', random_state=42)
# Xsm_train, ysm_train = sm.fit_sample(X_train, y_train)


# This will be the data were we are going to
Xsm_train, ysm_train = sm.fit_sample(original_Xtrain, original_ytrain)
```

In [46]:
```
# We Improve the score by 2% points approximately
# Implement GridSearchCV and the other models.

# Logistic Regression
t0 = time.time()
log_reg_sm = grid_log_reg.best_estimator_
log_reg_sm.fit(Xsm_train, ysm_train)
t1 = time.time()
print("Fitting oversample data took :{} sec".format(t1 - t0))
```

```
Fitting oversample data took :4.9667932987213135 sec
```

Test Data with Logistic Regression:

Confusion Matrix:

Positive/Negative: Refers to the class labels ["No", "Yes"]. True/False: Indicates whether the model's classification is correct or incorrect. True Negatives (Top-Left Square): Correctly classified "No" (No Fraud Detected).

False Negatives (Top-Right Square): Incorrectly classified "No" (No Fraud Detected).

False Positives (Bottom-Left Square): Incorrectly classified "Yes" (Fraud Detected).

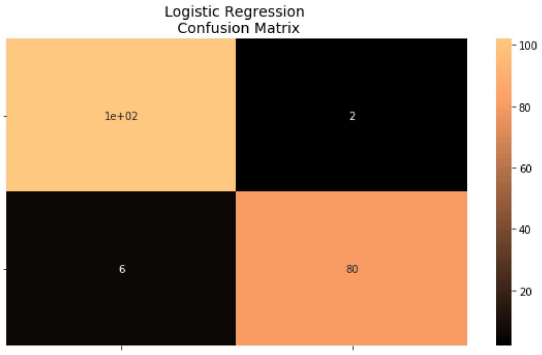
True Positives (Bottom-Right Square): Correctly classified "Yes" (Fraud Detected).

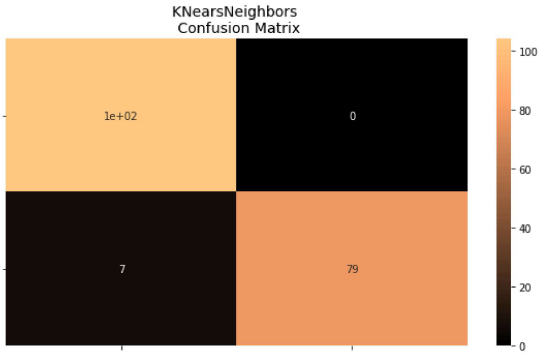
Summary:

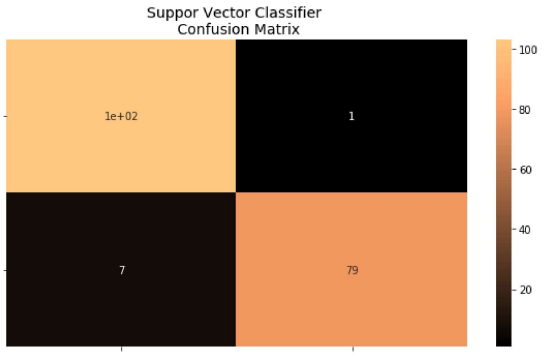
Random UnderSampling: We will assess the final performance of the classification models using the random undersampling subset, keeping in mind that this data is not from the original dataframe. Classification Models: The models that performed the best were Logistic Regression and the Support Vector Classifier (SVM).

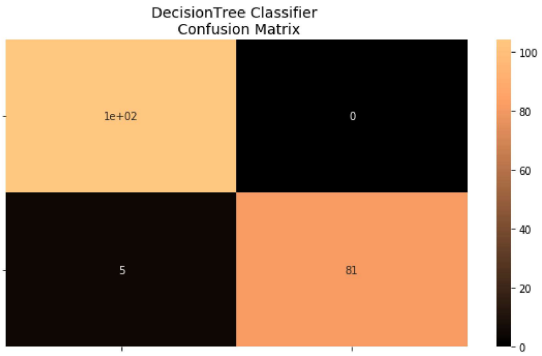
In [47]:

```
from sklearn.metrics import confusion_matrix

# Logistic Regression fitted using SMOTE technique
y_pred_log_reg = log_reg_sm.predict(X_test)

# Other models fitted with UnderSampling
y_pred_knear = knears_neighbors.predict(X_test)
y_pred_svc = svc.predict(X_test)
y_pred_tree = tree_clf.predict(X_test)


log_reg_cf = confusion_matrix(y_test, y_pred_log_reg)
kneighbors_cf = confusion_matrix(y_test, y_pred_knear)
svc_cf = confusion_matrix(y_test, y_pred_svc)
tree_cf = confusion_matrix(y_test, y_pred_tree)

fig, ax = plt.subplots(2, 2,figsize=(22,12))


sns.heatmap(log_reg_cf, ax=ax[0][0], annot=True, cmap=plt.cm.copper)
ax[0, 0].set_title("Logistic Regression \n Confusion Matrix", fontsize=14)
ax[0, 0].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[0, 0].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(kneighbors_cf, ax=ax[0][1], annot=True, cmap=plt.cm.copper)
ax[0][1].set_title("KNearsNeighbors \n Confusion Matrix", fontsize=14)
ax[0][1].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[0][1].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(svc_cf, ax=ax[1][0], annot=True, cmap=plt.cm.copper)
ax[1][0].set_title("Suppor Vector Classifier \n Confusion Matrix", fontsize=14)
ax[1][0].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[1][0].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(tree_cf, ax=ax[1][1], annot=True, cmap=plt.cm.copper)
ax[1][1].set_title("DecisionTree Classifier \n Confusion Matrix", fontsize=14)
ax[1][1].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[1][1].set_yticklabels(['', ''], fontsize=14, rotation=360)


plt.show()
```









In [48]:

```
from sklearn.metrics import classification_report


print('Logistic Regression:')
print(classification_report(y_test, y_pred_log_reg))

print('KNears Neighbors:')
print(classification_report(y_test, y_pred_knear))

print('Support Vector Classifier:')
print(classification_report(y_test, y_pred_svc))

print('Support Vector Classifier:')
print(classification_report(y_test, y_pred_tree))
```

```
Logistic Regression:
              precision    recall  f1-score   support

           0       0.94      0.98      0.96       104
           1       0.98      0.93      0.95        86

    accuracy                           0.96       190
   macro avg       0.96      0.96      0.96       190
weighted avg       0.96      0.96      0.96       190

KNears Neighbors:
              precision    recall  f1-score   support

           0       0.94      1.00      0.97       104
           1       1.00      0.92      0.96        86

    accuracy                           0.96       190
   macro avg       0.97      0.96      0.96       190
weighted avg       0.97      0.96      0.96       190

Support Vector Classifier:
              precision    recall  f1-score   support

           0       0.94      0.99      0.96       104
           1       0.99      0.92      0.95        86

    accuracy                           0.96       190
   macro avg       0.96      0.95      0.96       190
weighted avg       0.96      0.96      0.96       190

Support Vector Classifier:
              precision    recall  f1-score   support

           0       0.95      1.00      0.98       104
           1       1.00      0.94      0.97        86

    accuracy                           0.97       190
   macro avg       0.98      0.97      0.97       190
weighted avg       0.97      0.97      0.97       190
```

In [49]:

```python
# Final Score in the test set of logistic regression
from sklearn.metrics import accuracy_score

# Logistic Regression with Under-Sampling
y_pred = log_reg.predict(X_test)
undersample_score = accuracy_score(y_test, y_pred)

# Logistic Regression with SMOTE Technique (Better accuracy with SMOTE t)
y_pred_sm = best_est.predict(original_Xtest)
oversample_score = accuracy_score(original_ytest, y_pred_sm)

d = {'Technique': ['Random UnderSampling', 'Oversampling (SMOTE)'], 'Score': [undersam
final_df = pd.DataFrame(data=d)

# Move column
score = final_df['Score']
final_df.drop('Score', axis=1, inplace=True)
final_df.insert(1, 'Score', score)
```

```
# Note how high is accuracy score it can be misleading!
final_df
```

Out[49]:

	Technique	Score
0	Random UnderSampling	0.957895
1	Oversampling (SMOTE)	0.975843

Neural Networks Testing Random UnderSampling Data vs OverSampling (SMOTE):

In this section, we will implement a simple neural network with one hidden layer to determine which of the two logistic regression models we implemented (undersample or oversample using SMOTE) has better accuracy in detecting fraud and non-fraud transactions.

Our Main Goal:

Our primary objective is to assess how our simple neural network performs in both the random undersample and oversample dataframes to accurately predict both non-fraud and fraud cases. The emphasis isn't solely on detecting fraud. Consider the scenario where you are a cardholder, and after a legitimate purchase, your card gets blocked because the bank's algorithm mistakenly identifies it as fraud. This highlights the importance of correctly categorizing non-fraud transactions as well.

The Confusion Matrix:

Upper Left Square: Correctly classified non-fraud transactions by our model.

Upper Right Square: Transactions incorrectly classified as fraud by our model, but they are actually non-fraud.

Lower Left Square: Transactions incorrectly classified as non-fraud by our model, but they are actually fraud.

Lower Right Square: Correctly classified fraud transactions by our model.

Summary (Keras || Random UnderSampling):

Dataset: In this final testing phase, we will fit the model in both the random undersampled subset and the oversampled dataset (SMOTE) to predict the final results using the original dataframe's testing data.

Neural Network Structure: The model consists of one input layer with nodes equal to the number of features, plus a bias node, one hidden layer with 32 nodes, and one output node with two possible results: 0 (No fraud) or 1 (Fraud).

Other Characteristics: The model has a learning rate of 0.001, uses the AdamOptimizer as the optimizer, employs the "Relu" activation function, and utilizes sparse categorical cross-entropy for final outputs, providing probabilities for whether an instance is non-fraud or fraud, with the prediction selecting the highest probability between the two classes.

In [50]:
```
import keras
from keras import backend as K
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy

n_inputs = X_train.shape[1]

undersample_model = Sequential([
    Dense(n_inputs, input_shape=(n_inputs, ), activation='relu'),
    Dense(32, activation='relu'),
    Dense(2, activation='softmax')
])
```

```
Using TensorFlow backend.
```

In [51]:
```
undersample_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 30)                930
_________________________________________________________________
dense_2 (Dense)              (None, 32)                992
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 66
=================================================================
Total params: 1,988
Trainable params: 1,988
Non-trainable params: 0
_________________________________________________________________
```

In [52]:
```
undersample_model.compile(Adam(lr=0.001), loss='sparse_categorical_crossentropy', metr
```

In [53]:
```
undersample_model.fit(X_train, y_train, validation_split=0.2, batch_size=25, epochs=20
```

```
Train on 606 samples, validate on 152 samples
Epoch 1/20
 - 0s - loss: 0.7050 - acc: 0.6056 - val_loss: 0.4747 - val_acc: 0.7105
Epoch 2/20
 - 0s - loss: 0.3931 - acc: 0.8020 - val_loss: 0.3826 - val_acc: 0.8355
Epoch 3/20
 - 0s - loss: 0.3174 - acc: 0.8812 - val_loss: 0.3299 - val_acc: 0.8882
Epoch 4/20
 - 0s - loss: 0.2680 - acc: 0.9092 - val_loss: 0.2915 - val_acc: 0.9211
Epoch 5/20
 - 0s - loss: 0.2327 - acc: 0.9208 - val_loss: 0.2600 - val_acc: 0.9211
Epoch 6/20
 - 0s - loss: 0.2048 - acc: 0.9257 - val_loss: 0.2346 - val_acc: 0.9211
Epoch 7/20
 - 0s - loss: 0.1853 - acc: 0.9323 - val_loss: 0.2171 - val_acc: 0.9276
Epoch 8/20
 - 0s - loss: 0.1705 - acc: 0.9422 - val_loss: 0.2025 - val_acc: 0.9276
Epoch 9/20
 - 0s - loss: 0.1594 - acc: 0.9439 - val_loss: 0.1928 - val_acc: 0.9276
Epoch 10/20
 - 0s - loss: 0.1492 - acc: 0.9439 - val_loss: 0.1837 - val_acc: 0.9276
Epoch 11/20
 - 0s - loss: 0.1406 - acc: 0.9472 - val_loss: 0.1726 - val_acc: 0.9342
Epoch 12/20
 - 0s - loss: 0.1330 - acc: 0.9488 - val_loss: 0.1655 - val_acc: 0.9342
Epoch 13/20
 - 0s - loss: 0.1268 - acc: 0.9488 - val_loss: 0.1594 - val_acc: 0.9408
Epoch 14/20
 - 0s - loss: 0.1213 - acc: 0.9505 - val_loss: 0.1549 - val_acc: 0.9408
Epoch 15/20
 - 0s - loss: 0.1155 - acc: 0.9538 - val_loss: 0.1525 - val_acc: 0.9408
Epoch 16/20
 - 0s - loss: 0.1112 - acc: 0.9538 - val_loss: 0.1510 - val_acc: 0.9408
Epoch 17/20
 - 0s - loss: 0.1062 - acc: 0.9587 - val_loss: 0.1455 - val_acc: 0.9408
Epoch 18/20
 - 0s - loss: 0.1016 - acc: 0.9587 - val_loss: 0.1500 - val_acc: 0.9408
Epoch 19/20
 - 0s - loss: 0.0984 - acc: 0.9620 - val_loss: 0.1475 - val_acc: 0.9408
Epoch 20/20
 - 0s - loss: 0.0947 - acc: 0.9604 - val_loss: 0.1472 - val_acc: 0.9408
```

Out[53]:
```
<keras.callbacks.History at 0x7af4e0185be0>
```

In [54]:
```
undersample_predictions = undersample_model.predict(original_Xtest, batch_size=200, ve
```

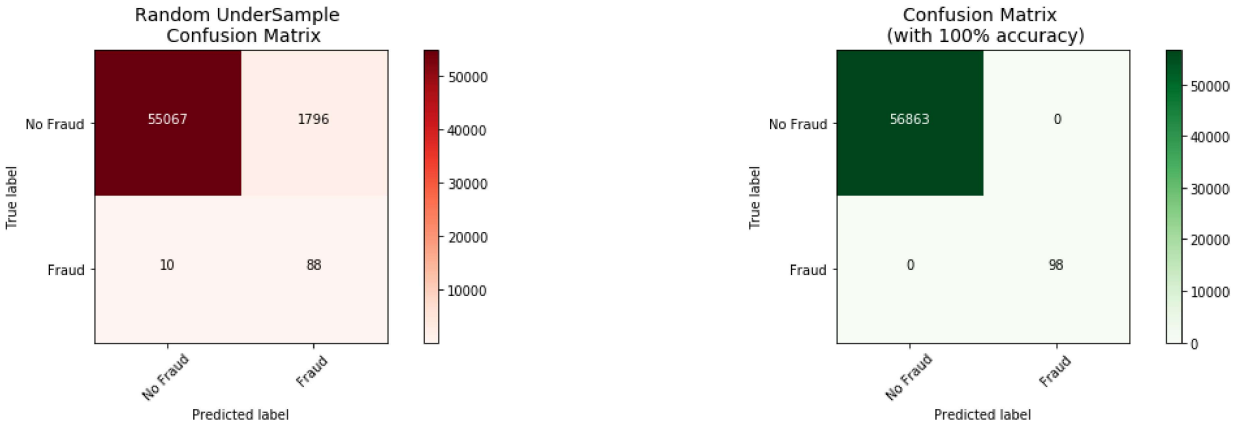
In [55]:
```
undersample_fraud_predictions = undersample_model.predict_classes(original_Xtest, batc
```

In [56]:
```
import itertools

# Create a confusion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
```

```
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title, fontsize=14)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [57]:
```
undersample_cm = confusion_matrix(original_ytest, undersample_fraud_predictions)
actual_cm = confusion_matrix(original_ytest, original_ytest)
labels = ['No Fraud', 'Fraud']

fig = plt.figure(figsize=(16,8))

fig.add_subplot(221)
plot_confusion_matrix(undersample_cm, labels, title="Random UnderSample \n Confusion M

fig.add_subplot(222)
plot_confusion_matrix(actual_cm, labels, title="Confusion Matrix \n (with 100% accurac
```

```
Confusion matrix, without normalization
[[55067  1796]
 [   10    88]]
Confusion matrix, without normalization
[[56863     0]
 [    0    98]]
```



Keras || OverSampling (SMOTE):

In [58]:
```
n_inputs = Xsm_train.shape[1]

oversample_model = Sequential([
    Dense(n_inputs, input_shape=(n_inputs, ), activation='relu'),
    Dense(32, activation='relu'),
    Dense(2, activation='softmax')
])
```

In [59]:
```
oversample_model.compile(Adam(lr=0.001), loss='sparse_categorical_crossentropy', metri
```

In [60]:
```
oversample_model.fit(Xsm_train, ysm_train, validation_split=0.2, batch_size=300, epoch
```

```
Train on 363923 samples, validate on 90981 samples
Epoch 1/20
 - 2s - loss: 0.0703 - acc: 0.9728 - val_loss: 0.0263 - val_acc: 0.9905
Epoch 2/20
 - 2s - loss: 0.0138 - acc: 0.9970 - val_loss: 0.0111 - val_acc: 0.9996
Epoch 3/20
 - 2s - loss: 0.0077 - acc: 0.9986 - val_loss: 0.0029 - val_acc: 1.0000
Epoch 4/20
 - 2s - loss: 0.0056 - acc: 0.9990 - val_loss: 0.0027 - val_acc: 1.0000
Epoch 5/20
 - 2s - loss: 0.0044 - acc: 0.9993 - val_loss: 0.0021 - val_acc: 1.0000
Epoch 6/20
 - 2s - loss: 0.0037 - acc: 0.9994 - val_loss: 0.0033 - val_acc: 1.0000
Epoch 7/20
 - 2s - loss: 0.0032 - acc: 0.9995 - val_loss: 8.7368e-04 - val_acc: 1.0000
Epoch 8/20
 - 2s - loss: 0.0030 - acc: 0.9995 - val_loss: 7.5384e-04 - val_acc: 1.0000
Epoch 9/20
 - 2s - loss: 0.0024 - acc: 0.9996 - val_loss: 0.0021 - val_acc: 1.0000
Epoch 10/20
 - 2s - loss: 0.0022 - acc: 0.9996 - val_loss: 0.0010 - val_acc: 1.0000
Epoch 11/20
 - 2s - loss: 0.0019 - acc: 0.9996 - val_loss: 3.5627e-04 - val_acc: 1.0000
Epoch 12/20
 - 2s - loss: 0.0017 - acc: 0.9997 - val_loss: 0.0014 - val_acc: 0.9998
Epoch 13/20
 - 2s - loss: 0.0015 - acc: 0.9997 - val_loss: 0.0011 - val_acc: 0.9998
Epoch 14/20
 - 2s - loss: 0.0013 - acc: 0.9997 - val_loss: 0.0011 - val_acc: 0.9998
Epoch 15/20
 - 1s - loss: 0.0013 - acc: 0.9998 - val_loss: 7.8853e-04 - val_acc: 0.9999
Epoch 16/20
 - 2s - loss: 0.0012 - acc: 0.9997 - val_loss: 6.9713e-04 - val_acc: 1.0000
Epoch 17/20
 - 2s - loss: 0.0013 - acc: 0.9997 - val_loss: 2.4048e-04 - val_acc: 1.0000
Epoch 18/20
 - 2s - loss: 8.8003e-04 - acc: 0.9998 - val_loss: 6.2304e-04 - val_acc: 1.0000
Epoch 19/20
 - 2s - loss: 0.0011 - acc: 0.9998 - val_loss: 7.8451e-04 - val_acc: 1.0000
Epoch 20/20
 - 2s - loss: 0.0010 - acc: 0.9998 - val_loss: 2.2109e-04 - val_acc: 1.0000
```

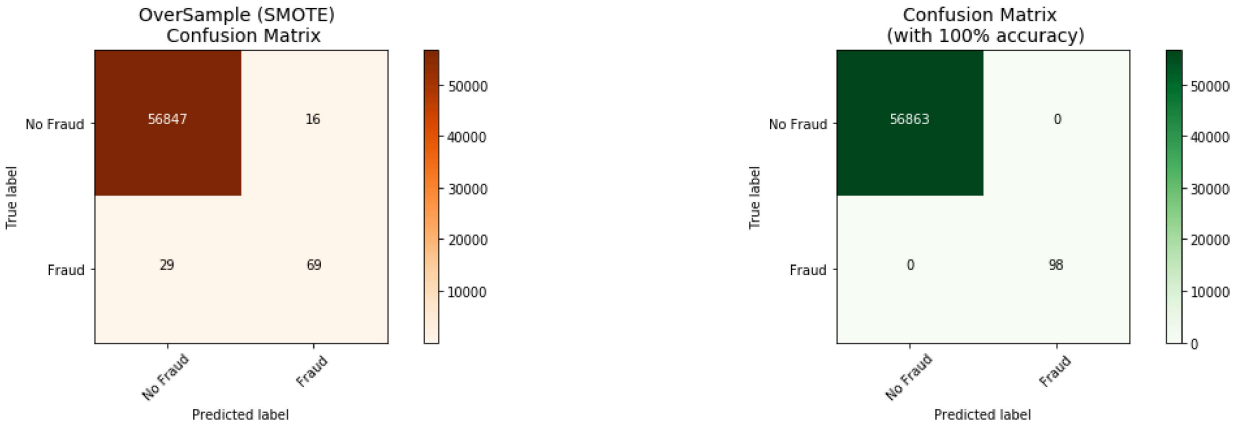
Out[60]:
```
<keras.callbacks.History at 0x7af4d46725f8>
```

In [61]:
```
oversample_predictions = oversample_model.predict(original_Xtest, batch_size=200, verb
```

In [62]:
```
oversample_fraud_predictions = oversample_model.predict_classes(original_Xtest, batch_
```

In [63]:
```
oversample_smote = confusion_matrix(original_ytest, oversample_fraud_predictions)
actual_cm = confusion_matrix(original_ytest, original_ytest)
labels = ['No Fraud', 'Fraud']

fig = plt.figure(figsize=(16,8))

fig.add_subplot(221)
plot_confusion_matrix(oversample_smote, labels, title="OverSample (SMOTE) \n Confusior

fig.add_subplot(222)
plot_confusion_matrix(actual_cm, labels, title="Confusion Matrix \n (with 100% accurac
```

```
Confusion matrix, without normalization
[[56847    16]
 [   29    69]]
Confusion matrix, without normalization
[[56863     0]
 [    0    98]]
```



Conclusion:

Implementing SMOTE on our imbalanced dataset helped address the label imbalance, as there were more non-fraud transactions than fraud transactions. However, it's worth noting that in some cases, the neural network on the oversampled dataset predicts fewer correct fraud transactions compared to our model using the undersampled dataset. It's important to remember that outlier removal was only applied to the random undersample dataset, not the oversampled one.

In the undersample data, our model struggles to correctly classify a significant number of non-fraud transactions and often misclassifies them as fraud cases. This could lead to customer dissatisfaction and an increase in complaints if legitimate transactions are blocked due to this misclassification. The next step in our analysis will involve removing outliers from the oversampled dataset to see if our test set accuracy improves.

Note: Predictions and accuracies may change due to data shuffling on both types of dataframes. The primary goal is to ensure our models can accurately classify both non-fraud and fraud transactions.