homework8

November 8, 2019

NAME: Jacob Duvall
SECTION: C S-5970-995
CS 5970: Machine Learning Practices

1 Homework 8: Support Vector Machines

1.1 Assignment Overview

Follow the TODOs and read through and understand any provided code.
Post any questions regarding the assignment, to the Canvas discussion. For all plots, make sure all necessary axes and curves are clearly and accurately labeled. Include figure/plot titles appropriately as well.

1.1.1 Task

For this assignment you will be exploring support vector machines (SVMs) using GridsearchCV and working with highly unbalanced datasets.

1.1.2 Data set

European Cardholder Credit Card Transactions, September 2013
This dataset presents transactions that occurred over two days. There were 492 incidents of frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) accounts for 0.197% of all transactions.

Features
* V1, V2, … V28: are principal components obtained with PCA
* Time: the seconds elapsed between each transaction and the first transaction
* Amount: is the transaction Amount
* Class: the predicted variable; 1 in case of fraud and 0 otherwise.

Given the class imbalance ratio, it is recommended to use precision, recall and the Area Under the Precision-Recall Curve (AUPRC) to evaluate skill. Traditional accuracy and AUC are not meaningful for highly unbalanced classification. These scores are misleading due to the high impact of the large number of negative cases that can easily be identified.

Examining precision and recall is more informative as these disregard the number of correctly identified negative cases (i.e. TN) and focus on the number of correctly identified positive cases (TP) and mis-identified negative cases (FP). Another useful metric is the F1 score which is the harmonic mean of the precision and recall; 1 is the best F1 score.

Confusion Matrix
[TN FP]
[FN TP]

Accuracy = $\frac{TN+TP}{TN+TP+FN+FP}$
TPR = $\frac{TP}{TP+FN}$
FPR = $\frac{FP}{FP+TN}$

Recall = TPR = $\frac{TP}{TP+FN}$
Precision = $\frac{TP}{TP+FP}$
F1 Score = 2 * $\frac{precision*recall}{precision+recall}$

See the references below for more details on precision, recall, and the F1 score.

The dataset was collected and analysed during a research collaboration of Worldline and the Machine Learning Group (http://mlg.ulb.ac.be) of ULB (Université Libre de Bruxelles) on big data mining and fraud detection [1]

[1] Andrea Dal Pozzolo, Olivier Caelen, Reid A. Johnson and Gianluca Bontempi. Calibrating Probability with Undersampling for Unbalanced Classification. In Symposium on Computational Intelligence and Data Mining (CIDM), IEEE, 2015. http://mlg.ulb.ac.be/BruFence . http://mlg.ulb.ac.be/ARTML

1.1.3 Objectives

- Understanding Support Vector Machines
- GridSearch with Classification
- Creating Scoring functions
- Stratification

1.1.4 Notes

- Do not save work within the ml_practices folder

1.1.5 General References

- Guide to Jupyter
- Python Built-in Functions
- Python Data Structures
- Numpy Reference
- Numpy Cheat Sheet
- Summary of matplotlib
- DataCamp: Matplotlib

- Pandas DataFrames
- Sci-kit Learn Linear Models
- Sci-kit Learn Ensemble Models
- Sci-kit Learn Metrics
- Sci-kit Learn Model Selection
- Scoring Parameter
- Scoring
- Plot ROC
- Precision, Recall, F1 Score
- Precision-Recall Curve
- Probability Plot

[1]:
```
# THESE FIRST 3 IMPORTS ARE CUSTOM .py FILES AND CAN BE FOUND ON THE SERVER
# AND GIT
import visualize
import metrics_plots
from pipeline_components import DataSampleDropper, DataFrameSelector

import pandas as pd
import numpy as np
import scipy.stats as stats
import os, re, fnmatch
import pathlib, itertools
import time as timelib
import matplotlib.pyplot as plt

from math import floor, ceil
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import RobustScaler, StandardScaler
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.model_selection import learning_curve, StratifiedKFold
from sklearn.metrics import make_scorer, precision_recall_curve
from sklearn.metrics import confusion_matrix, precision_score
from sklearn.metrics import roc_curve, auc, f1_score, recall_score
from sklearn.svm import SVC
from sklearn.externals import joblib

HOME_DIR = pathlib.Path.home()
CW_DIR = pathlib.Path.cwd()

FIGW = 12
FIGH = 5
```

```
FONTSIZE = 8

plt.rcParams['figure.figsize'] = (FIGW, FIGH)
plt.rcParams['font.size'] = FONTSIZE

plt.rcParams['xtick.labelsize'] = FONTSIZE
plt.rcParams['ytick.labelsize'] = FONTSIZE

%matplotlib inline
```

2 LOAD DATA

[2]:
```
# 284806 rows, 'None' to read whole file
nRowsRead = None

# TODO: set appropriately
filename = 'creditcard.csv'

crime_stats_full = pd.read_csv(filename, delimiter=',', nrows=nRowsRead)
crime_stats_full.dataframeName = 'creditcard.csv'
nRows, nCols = crime_stats_full.shape
print(f'There are {nRows} rows and {nCols} columns')
```

```
There are 284806 rows and 31 columns
```

[3]:
```
""" PROVIDED
good (negative case = 0)
fraud (positive case = 1)
"""
targetnames = ['good', 'fraud']

pos_full = crime_stats_full.loc[crime_stats_full['Class'] == 1]
neg_full = crime_stats_full.loc[crime_stats_full['Class'] == 0]

pos_full.shape, neg_full.shape
```

[3]:
```
((492, 31), (284314, 31))
```

[4]:
```
""" PROVIDED
Compute the postive fraction
"""
pos_fraction = pos_full.shape[0] / nRows
neg_fraction = 1 - pos_fraction

pos_fraction, neg_fraction
```

[4]: (0.001727491696101908, 0.9982725083038981)

[5]:
```
""" PROVIDED
Select Random Subset of data
"""
np.random.seed(42)
subset_size = 20000
selected_indices = np.random.choice(range(nRows), size=subset_size,␣
 ↪replace=False)
selected_indices
```

[5]: array([43428, 49906, 29474, …, 192406, 124100, 12947])

[6]:
```
""" PROVIDED
List the features and shape of the data
"""
crime_stats = crime_stats_full.loc[selected_indices, :]
crime_stats.columns, crime_stats.shape
```

```
[6]: (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'), (20000, 31))
```

[7]:
```
""" PROVIDED
Display whether there are any NaNs
"""
crime_stats.isna().any()
```

```
[7]: Time      False
V1        False
V2        False
V3        False
V4        False
V5        False
V6        False
V7        False
V8        False
V9        False
V10       False
V11       False
V12       False
V13       False
V14       False
V15       False
V16       False
```

```
V17       False
V18       False
V19       False
V20       False
V21       False
V22       False
V23       False
V24       False
V25       False
V26       False
V27       False
V28       False
Amount    False
Class     False
dtype: bool
```

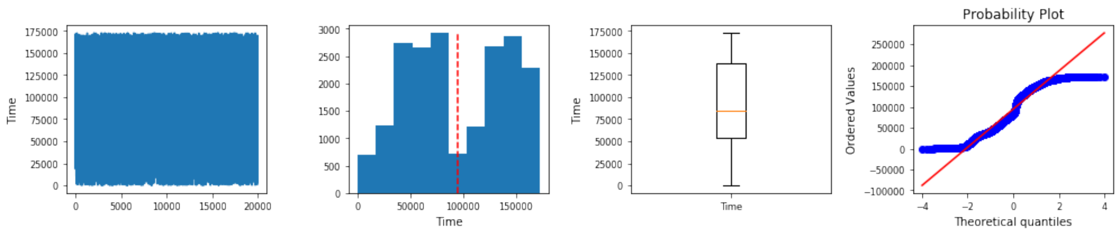
[8]:
```python
""" TODO
Display summary statistics for each feature of the dataframe
"""
crime_stats.describe()
```

[8]:
```
                Time            V1            V2            V3            V4  \
count   20000.000000  20000.000000  20000.000000  20000.000000  20000.000000
mean    94490.802400      0.002913     -0.029847     -0.001526      0.018716
std     47313.538305      2.011012      1.721684      1.545744      1.414560
min         0.000000    -40.042538    -48.060856    -30.177317     -5.266509
25%     54111.000000     -0.916870     -0.607590     -0.904430     -0.840008
50%     84335.500000      0.041402      0.053039      0.186126      0.003204
75%    139023.250000      1.329557      0.780855      1.047085      0.758450
max    172782.000000      2.451888     16.497472      9.382558     12.699542

                 V5            V6            V7            V8            V9  \
count  20000.000000  20000.000000  20000.000000  20000.000000  20000.000000
mean      -0.009522     -0.002003     -0.008675      0.004225     -0.000767
std        1.390694      1.325199      1.223386      1.172031      1.105181
min      -23.611865    -20.869626    -31.197329    -37.353443     -9.462573
25%       -0.713130     -0.761379     -0.564197     -0.206495     -0.644663
50%       -0.066121     -0.270283      0.025205      0.021737     -0.048547
75%        0.593397      0.393435      0.562905      0.325365      0.597407
max       26.647697     16.493227     21.437514     17.052566     15.594995

       …           V21           V22           V23           V24  \
count  …  20000.000000  20000.000000  20000.000000  20000.000000
mean   …     -0.002876      0.000937      0.002760      0.000499
std    …      0.714353      0.719430      0.616109      0.603601
min    …    -13.963731     -8.887017    -22.575000     -2.824849
25%    …     -0.228380     -0.543027     -0.161554     -0.352267
```

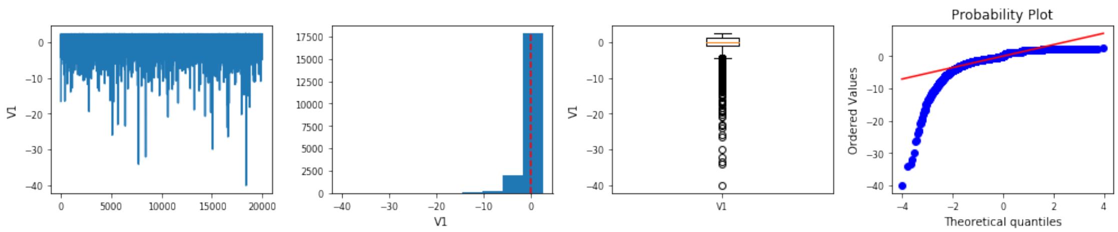
```
50%     …    -0.030045    0.007540    -0.011669    0.044262
75%     …     0.181191    0.526424     0.147149    0.441184
max     …    27.202839    4.080214    19.002942    3.546031

                V25           V26           V27           V28        Amount  \
count  20000.000000  20000.000000  20000.000000  20000.000000  20000.000000
mean       0.004572     -0.003928     -0.000498     -0.001587     89.525975
std        0.517540      0.478031      0.437142      0.349640    247.838774
min       -4.196468     -2.068561    -22.565679    -11.710896      0.000000
25%       -0.311738     -0.325381     -0.070359     -0.052049      5.760000
50%        0.027412     -0.055531      0.001234      0.010908     22.035000
75%        0.351777      0.231973      0.088768      0.078558     77.720000
max        4.513681      2.952093      9.200883     16.129609   8787.000000

             Class
count  20000.00000
mean       0.00155
std        0.03934
min        0.00000
25%        0.00000
50%        0.00000
75%        0.00000
max        1.00000

[8 rows x 31 columns]
```

3 VISUALIZE DATA

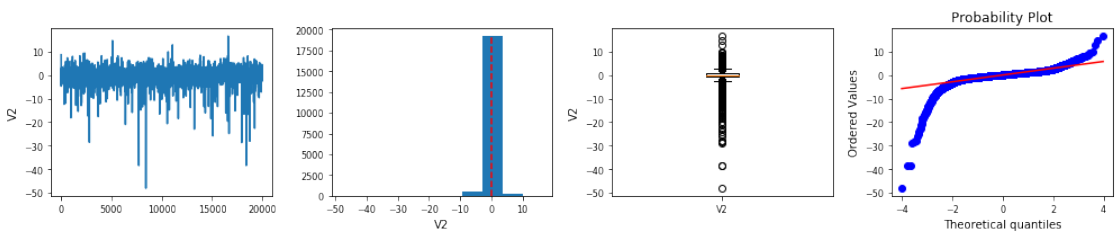
[9]:
```python
""" TODO
Display the distributions of the data
use visualize.featureplots(crime_stats_dropna.values, crime_stats.columns)
to generate trace plots, histograms, boxplots, and probability plots for
each feature.

A probability plot is utilized to evaulate the normality of a distribution.
The data are plot against a theoritical distribution, such that if the data
are normal, they'll follow the diagonal line. See the reference above for
more information.
"""
crime_stats_dropna = crime_stats.dropna()
# TODO: visualize the features
visualize.featureplots(crime_stats_dropna.values, crime_stats.columns)

# Right click to enable scrolling
```



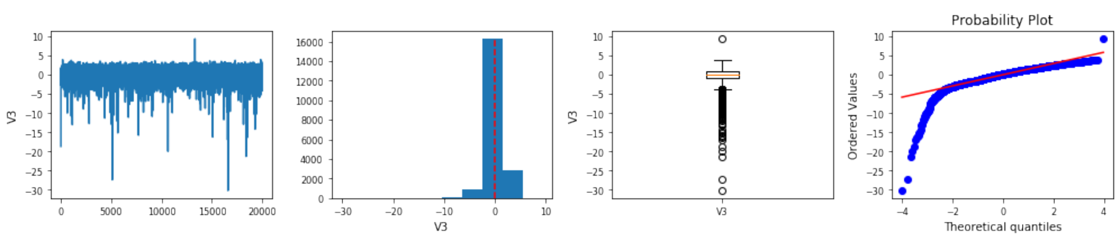
myplots Time



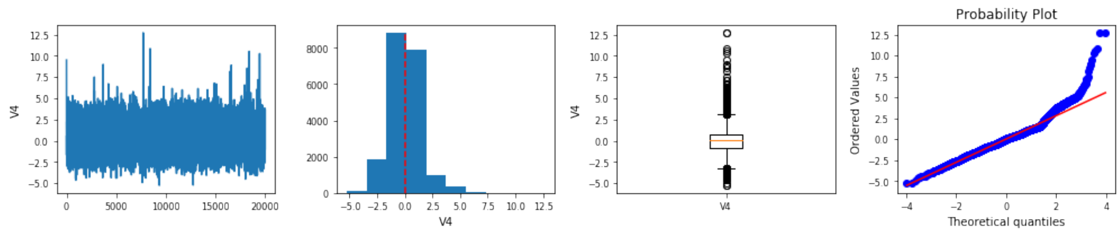
myplots V1



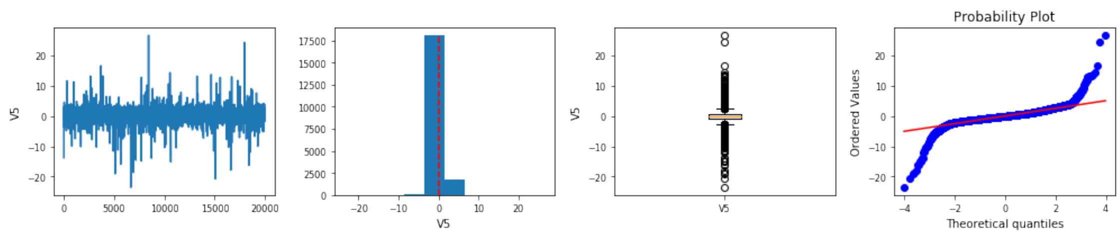
myplots V2



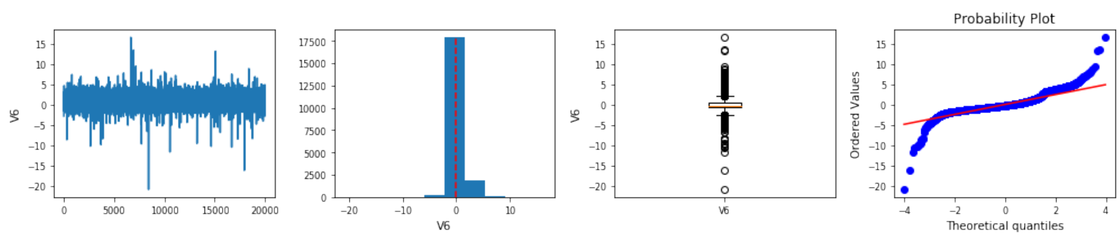
myplots V3



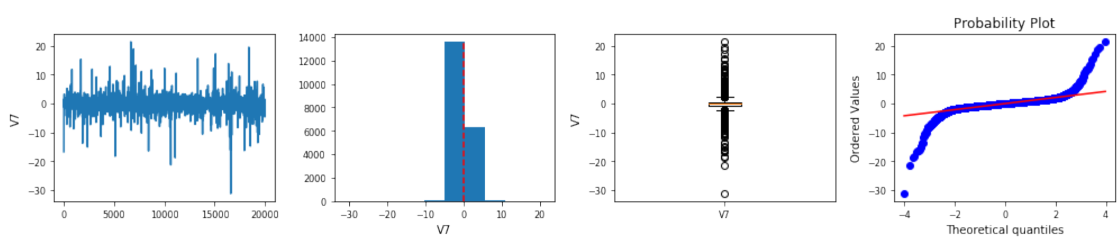
myplots V4



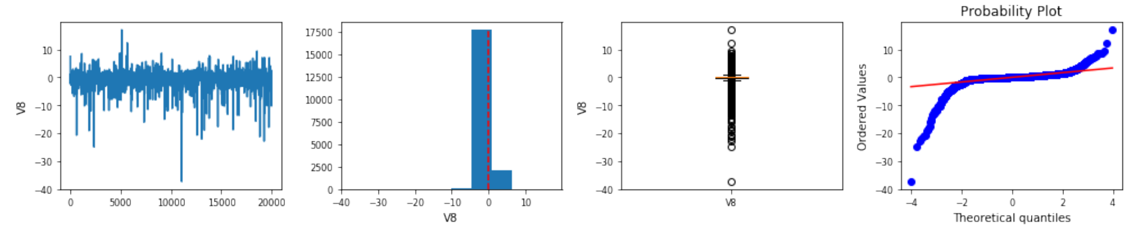
myplots V5



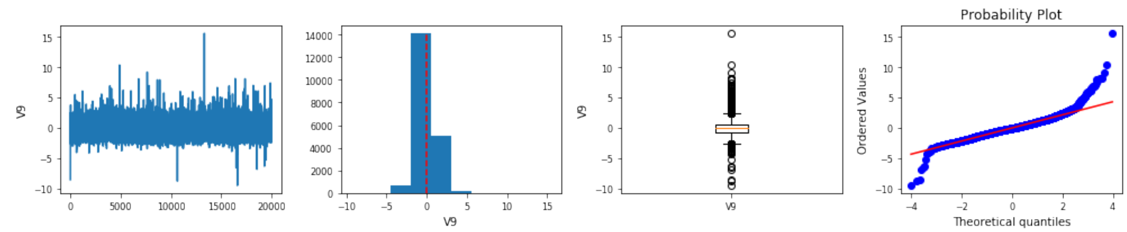
myplots V6



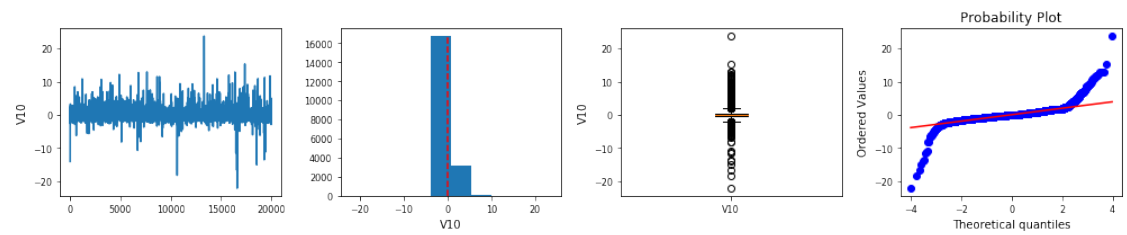
myplots V7



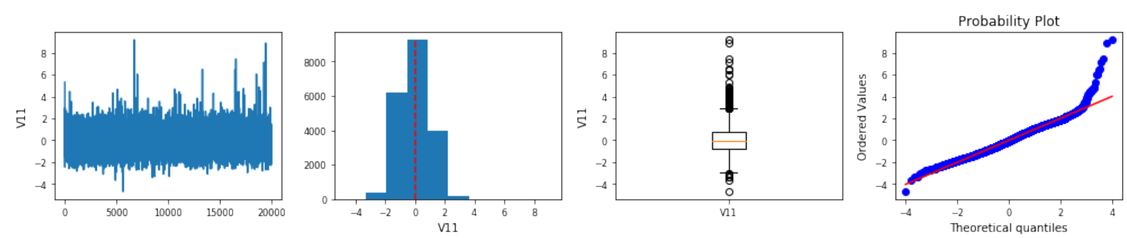
myplots V8



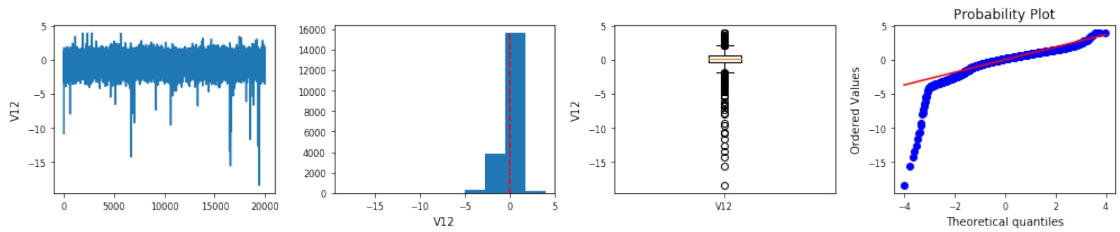
myplots V9



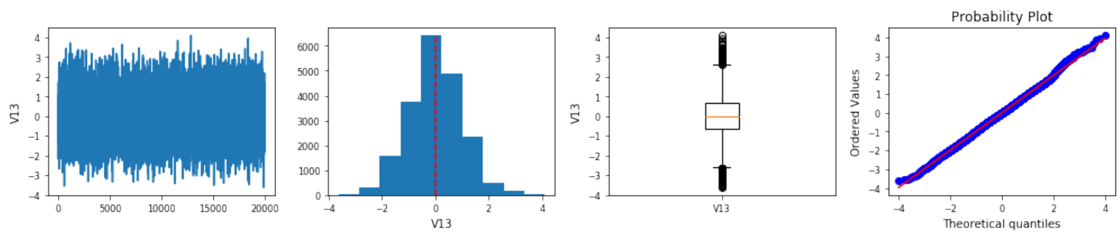
myplots V10



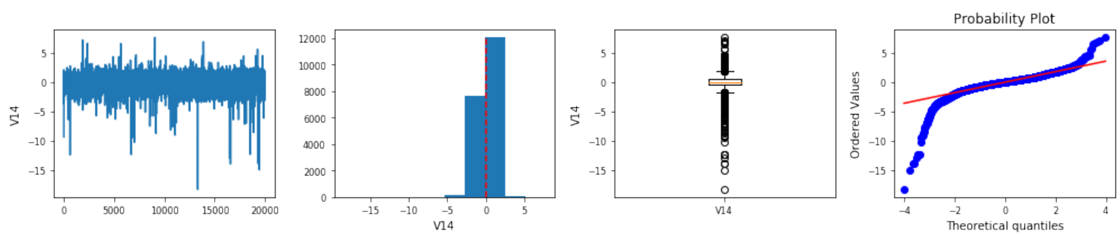
myplots V11



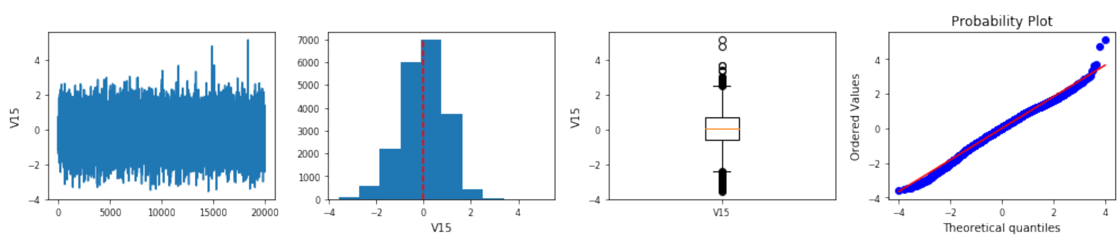
myplots V12



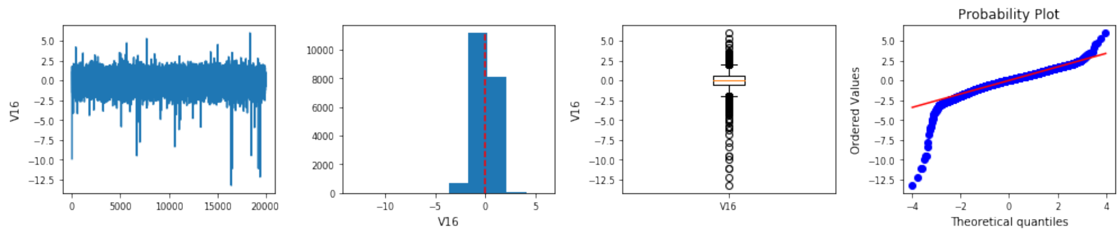
myplots V13



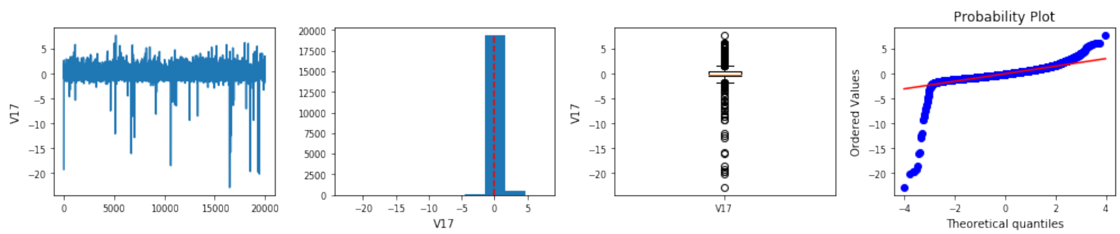
myplots V14



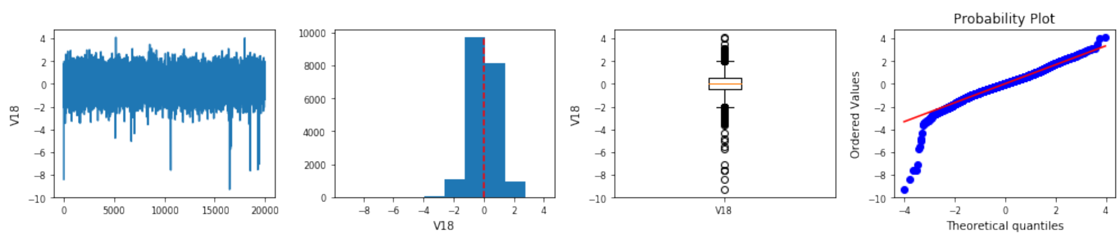
myplots V15



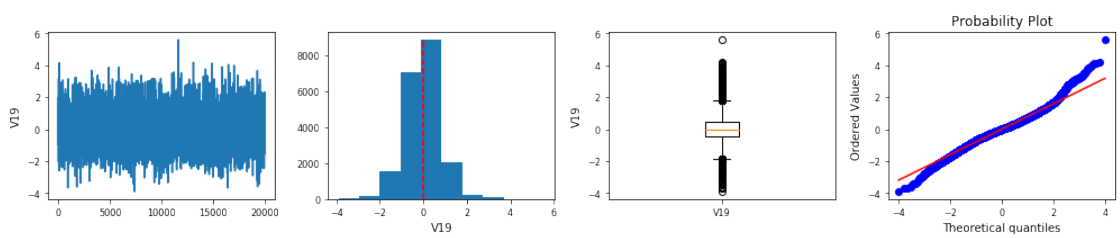

```
myplots V16
```



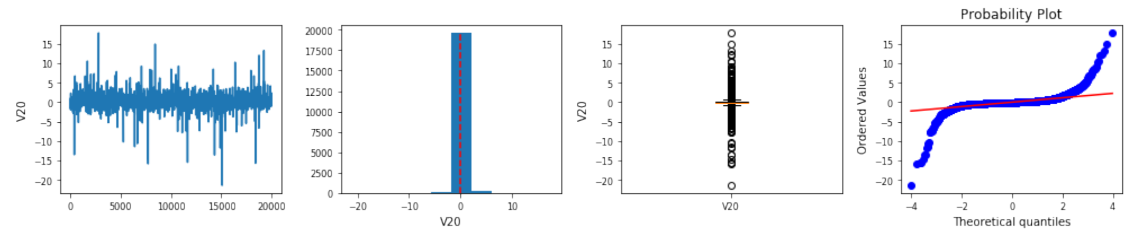

```
myplots V17
```



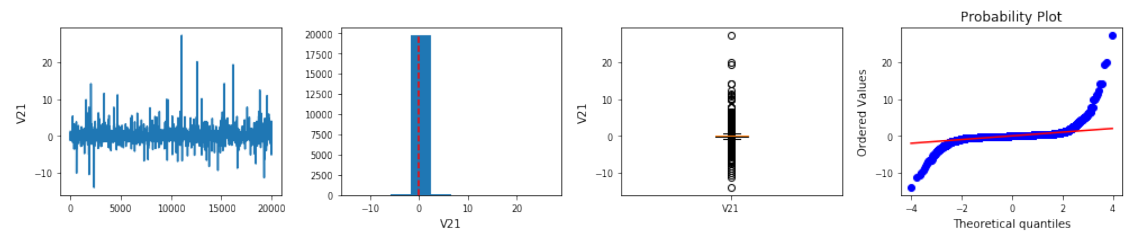

```
myplots V18
```



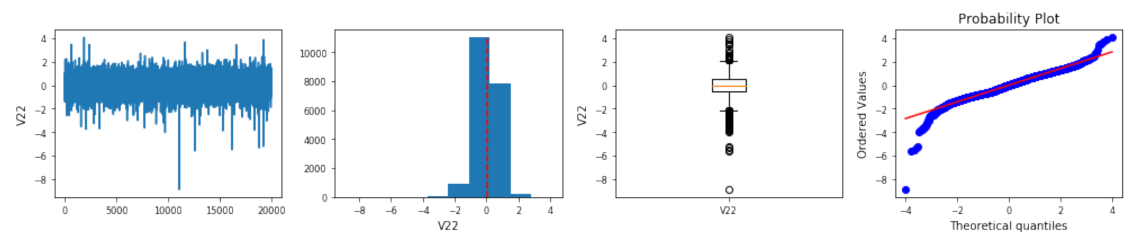

```
myplots V19
```



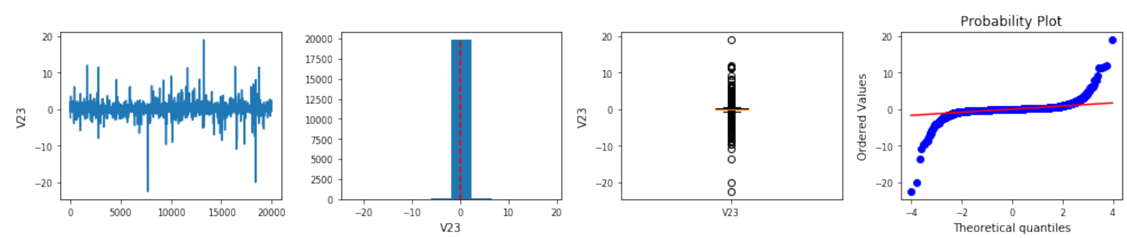
myplots V20



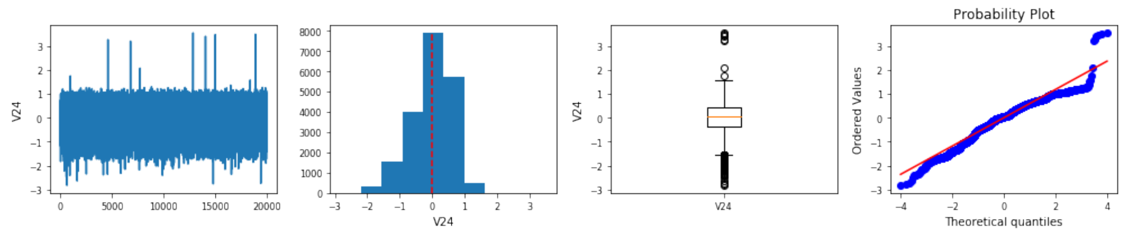
myplots V21



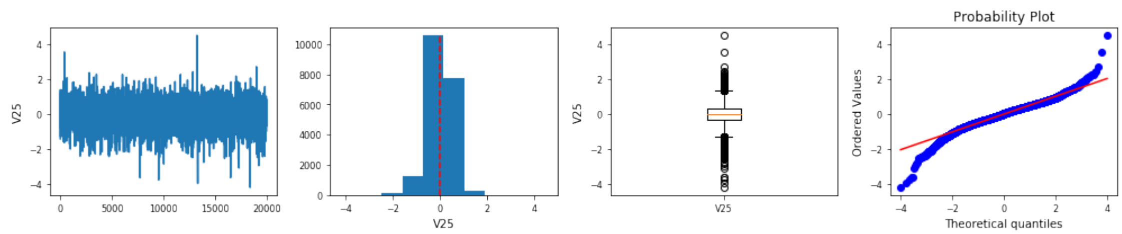
myplots V22



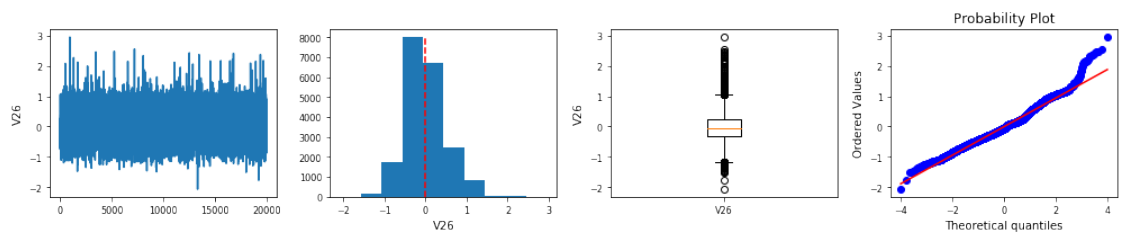
myplots V23



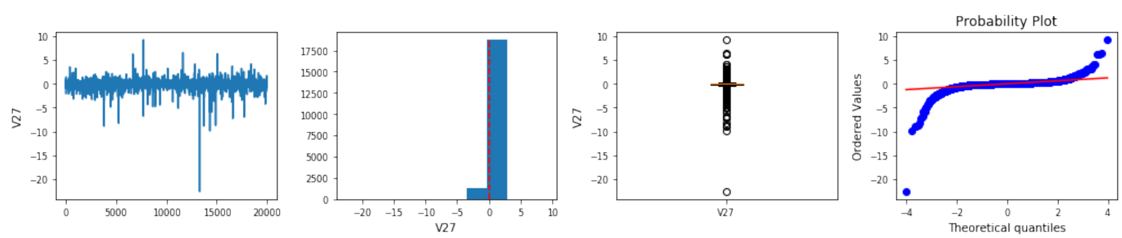
myplots V24



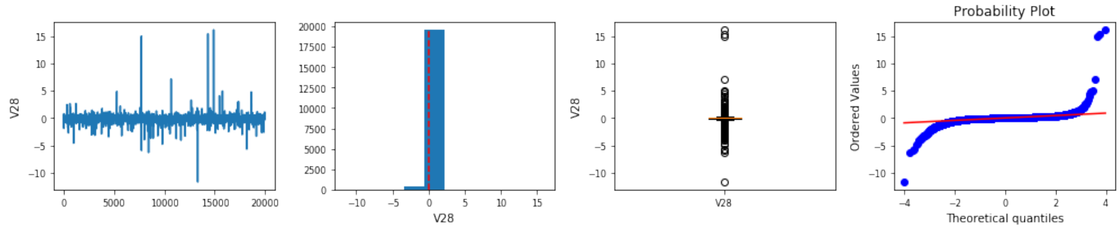
myplots V25



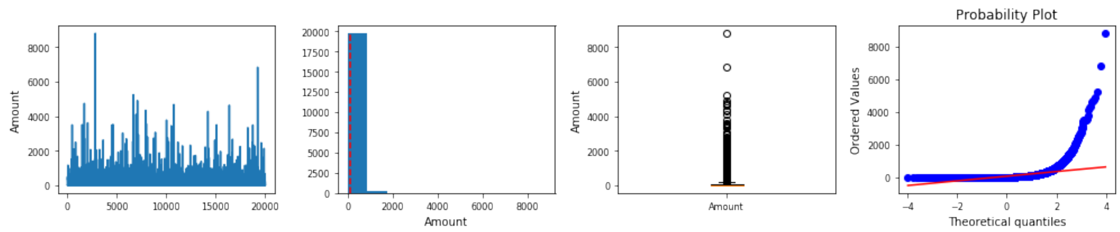
myplots V26



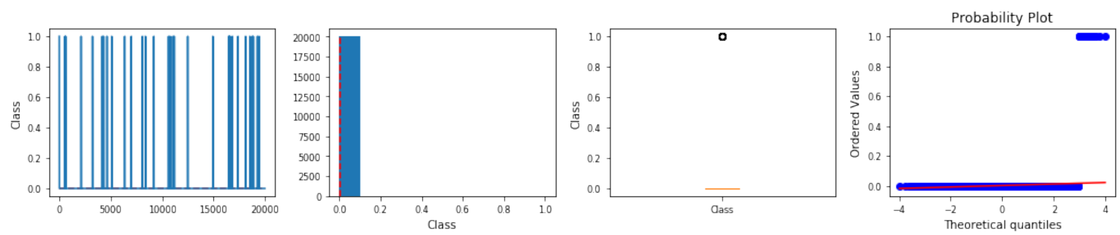
myplots V27



myplots V28

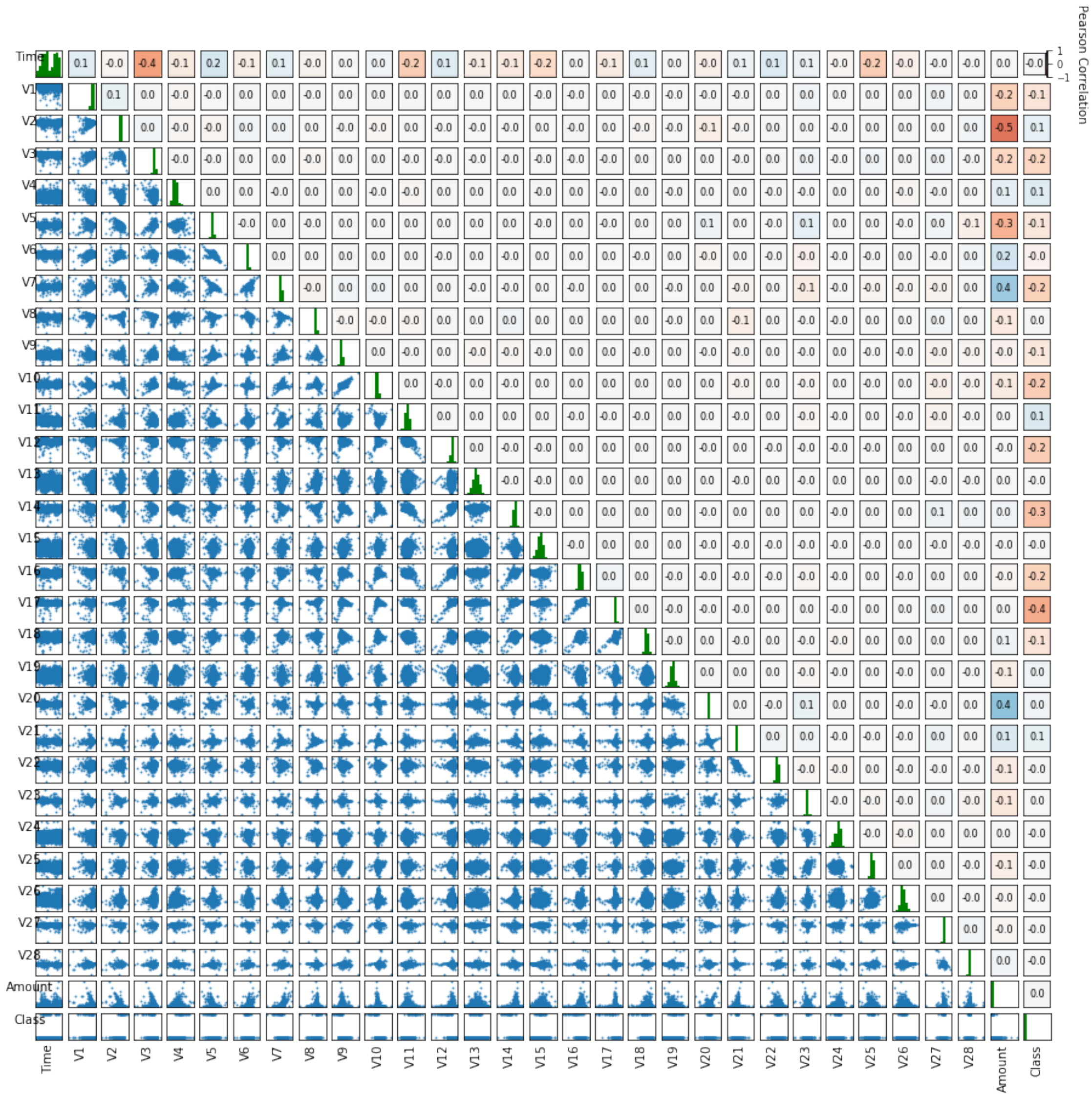


myplots Amount



myplots Class

[10]:
```
""" PROVIDED
Display the Pearson correlation between all pairs of the features
use visualize.scatter_corrplots(crime_stats_dropna.values, crime_stats.columns,
 ↪corrfmt="%.1f", FIGW=15)
"""
visualize.scatter_corrplots(crime_stats_dropna.values, crime_stats.columns,
 ↪corrfmt="%.1f", FIGW=15)
```



```
[11]: """ PROVIDED
      Separate the postive and negative examples
      """
      pos = crime_stats.loc[crime_stats['Class'] == 1]
      neg = crime_stats.loc[crime_stats['Class'] == 0]

      pos.shape, neg.shape
```

```
[11]: ((31, 31), (19969, 31))
```

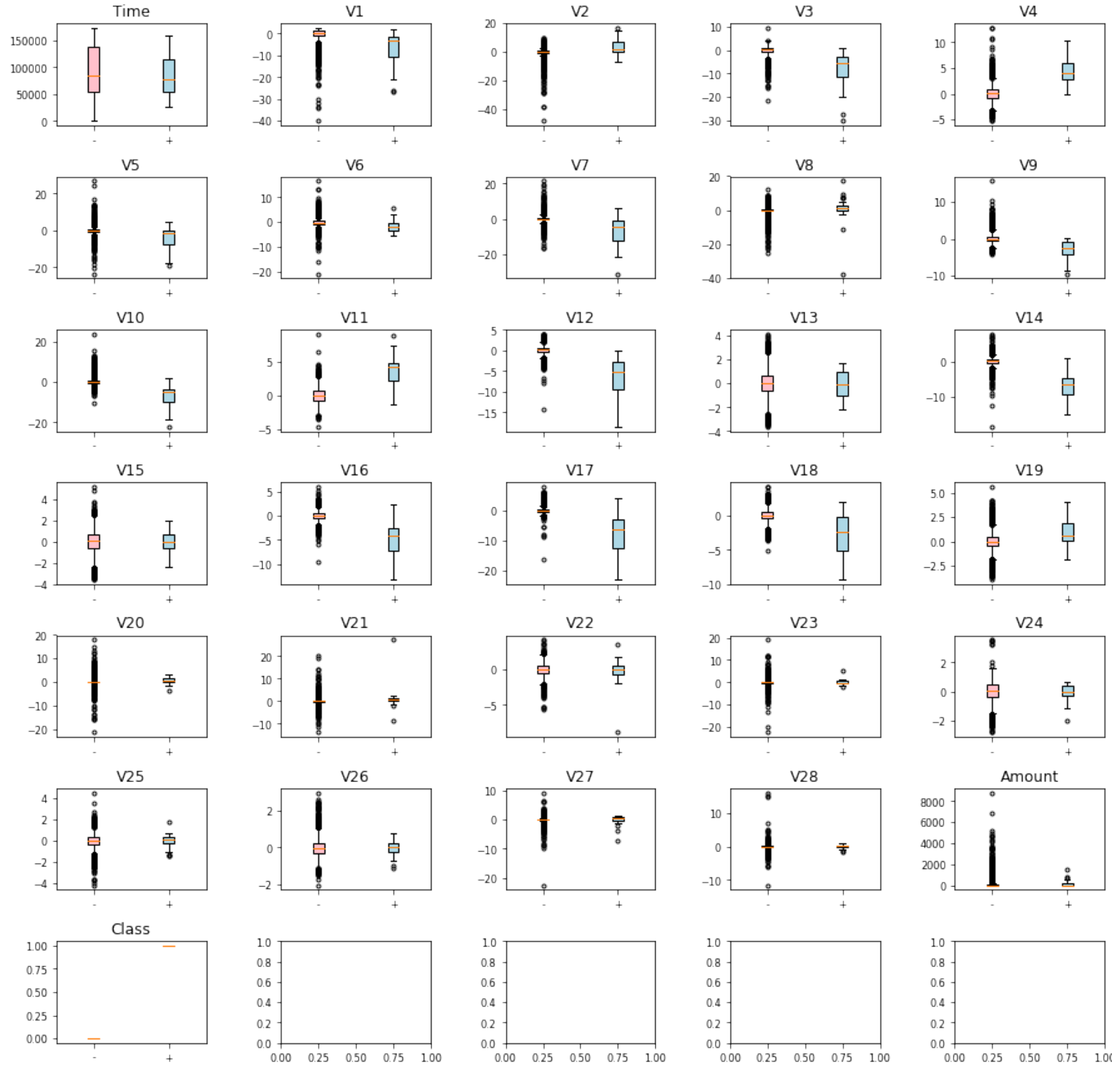
```
[12]: """ PROVIDED
      Compute the postive fraction
      """
```

```
pos_fraction = pos.shape[0] / nRows
neg_fraction = 1 - pos_fraction

pos_fraction, neg_fraction
```

[12]: (0.00010884602150235599, 0.9998911539784976)

[13]:
```
""" PROVIDED
Compare the features for the positive and negative examples
"""
features_displayed = pos.columns
ndisplayed = len(features_displayed)
ncols = 5
nrows = ceil(ndisplayed / ncols)

fig, axs = plt.subplots(nrows, ncols, figsize=(15, 15))
fig.subplots_adjust(wspace=.5, hspace=.5)
axs = axs.ravel()
for ax, feat_name in zip(axs, features_displayed):
    boxplot = ax.boxplot([neg[feat_name], pos[feat_name]], patch_artist=True,␣
 ↪sym='.')
    boxplot['boxes'][0].set_facecolor('pink')
    boxplot['boxes'][1].set_facecolor('lightblue')
    ax.set_xticklabels(['-', '+'])
    ax.set(title=feat_name)
""""""
```

[13]: ''



4 PRE-PROCESS DATA

4.1 Data Clean Up and Feature Selection

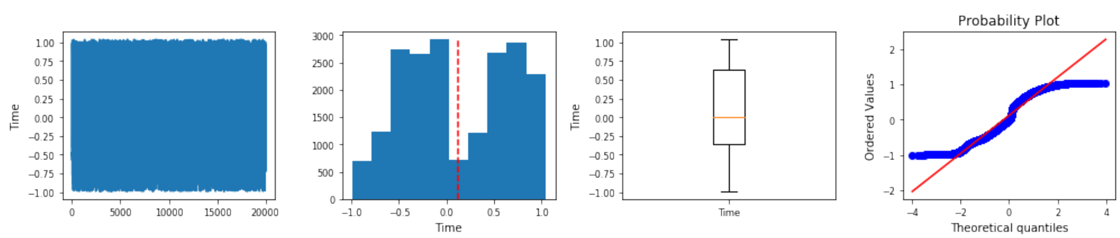
[14]:
```
""" PROVIDED
Construct Pipeline to pre-process data
"""
feature_names = crime_stats.columns.drop(['Class'])
pipe_X = Pipeline([
    ("NaNrowDropper", DataSampleDropper()),
    ("selectAttribs", DataFrameSelector(feature_names)),
    ("scaler", RobustScaler())
```

```
])

pipe_y = Pipeline([
    ("NaNrowDropper", DataSampleDropper()),
    ("selectAttribs", DataFrameSelector(['Class']))
])
```

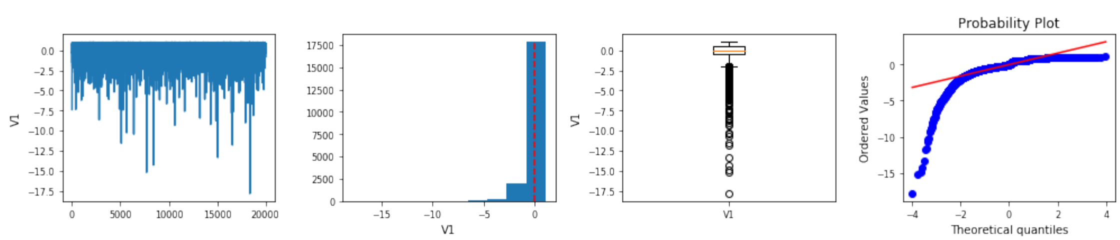
[15]:
```
""" TODO
Pre-process the data using the pipeliine
"""

X = pipe_X.fit_transform(crime_stats)
y = pipe_y.fit_transform(crime_stats)
np.any(np.isnan(X))
```

[15]: False

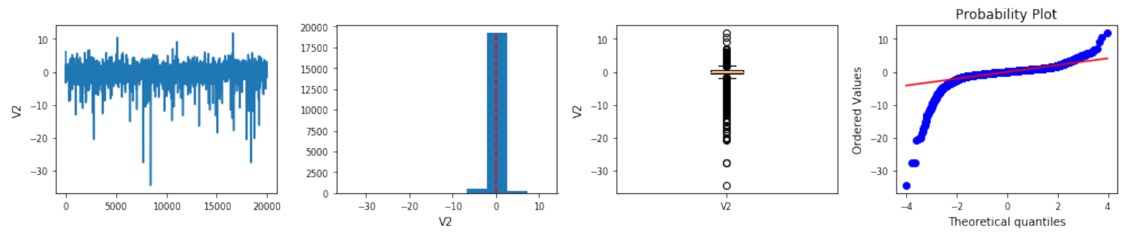
[16]:
```
""" TODO
Re-visualize the pre-processed data
use visualize.featureplots()
"""
visualize.featureplots(X, crime_stats.columns[:-1])
visualize.featureplots(y.values, y.columns)
```



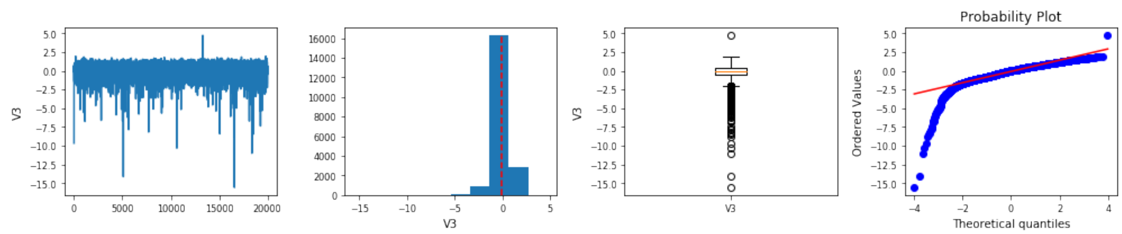
myplots Time



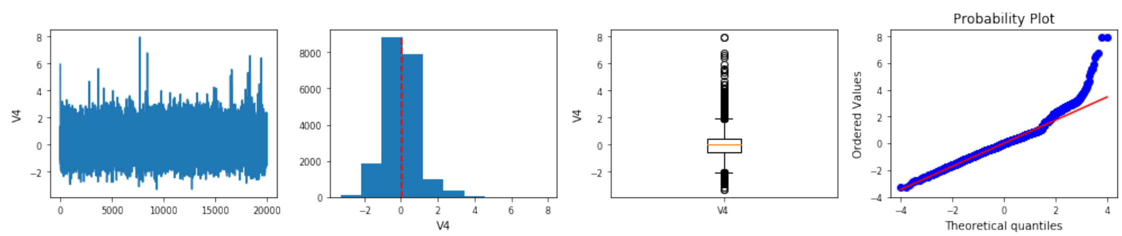
myplots V1



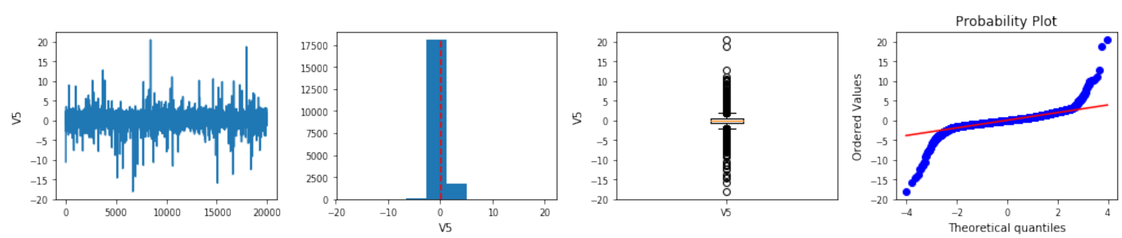
myplots V2



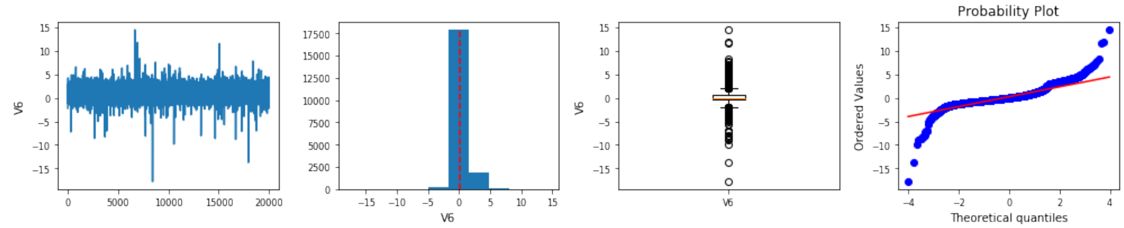
myplots V3



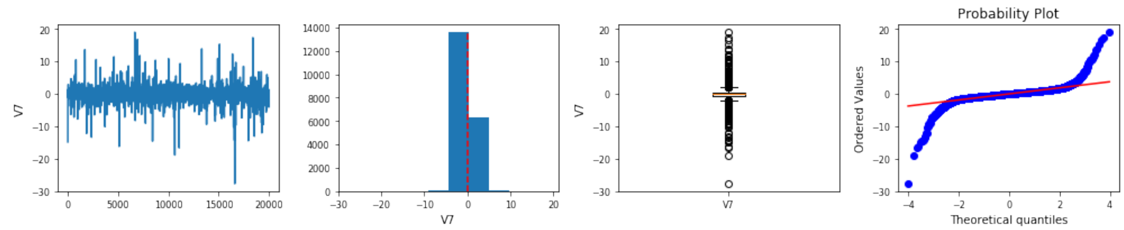
myplots V4



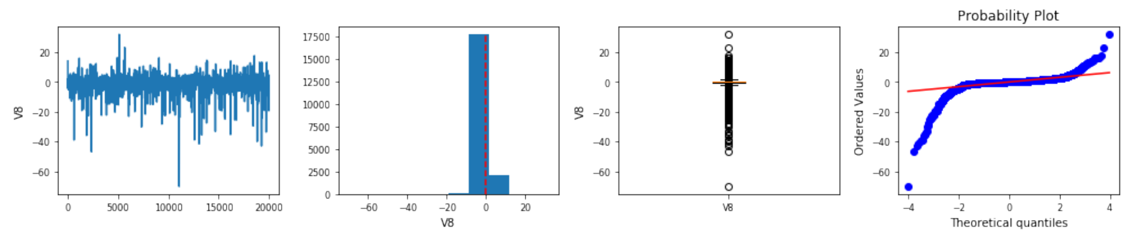
myplots V5



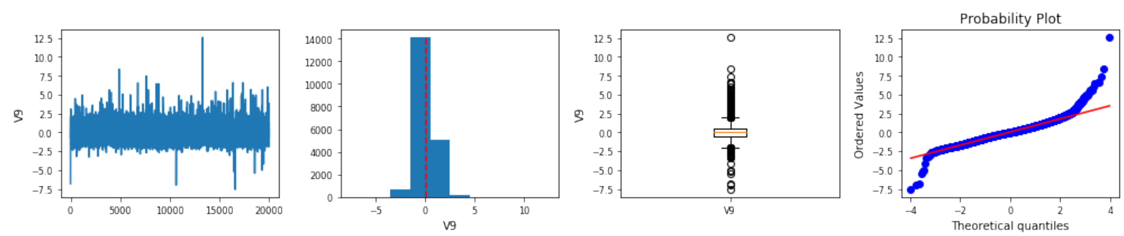

```
myplots V6
```



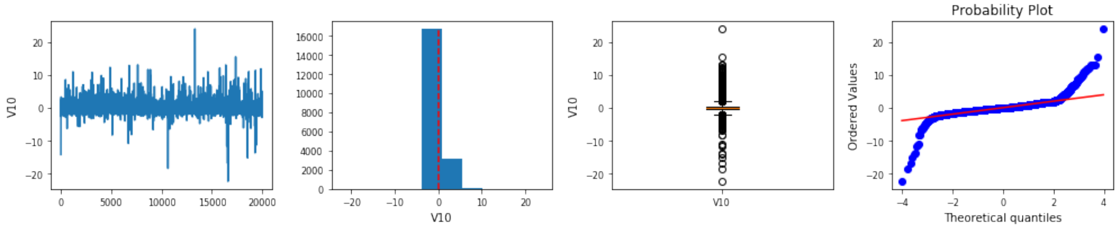

```
myplots V7
```



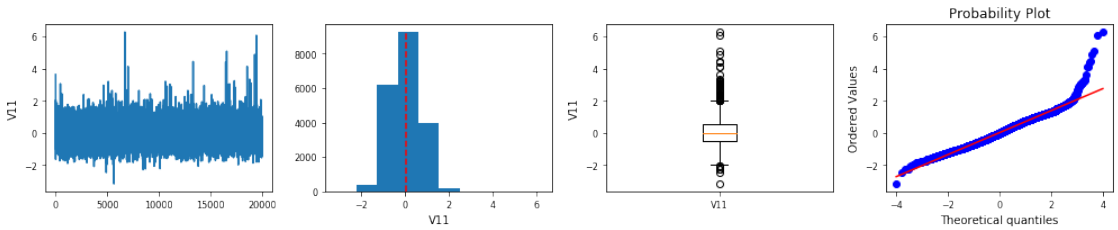

```
myplots V8
```



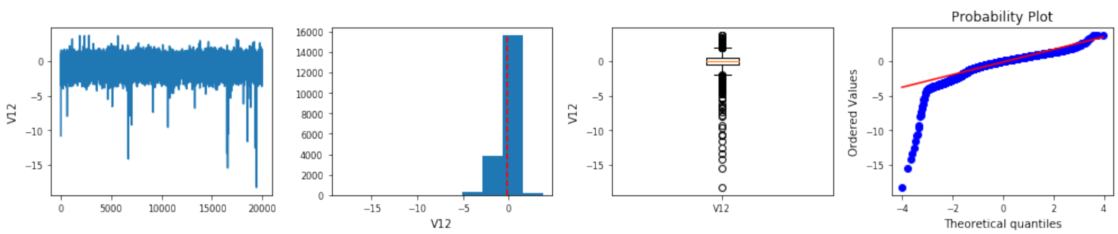

```
myplots V9
```



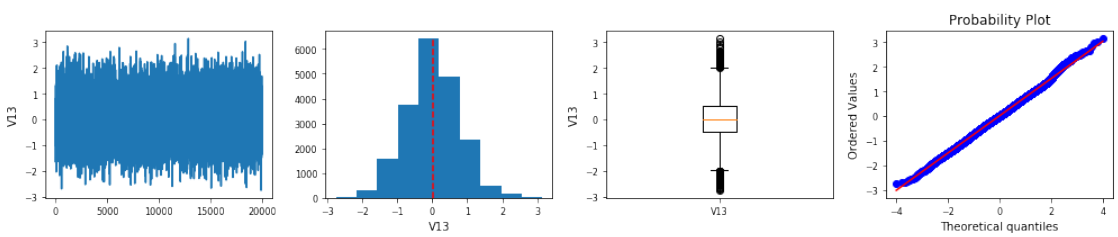
myplots V10



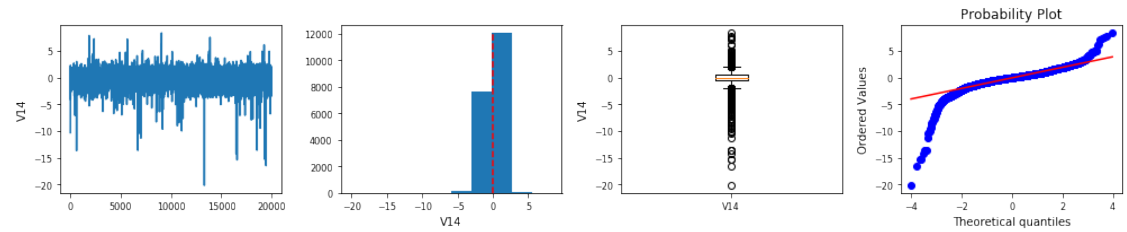
myplots V11



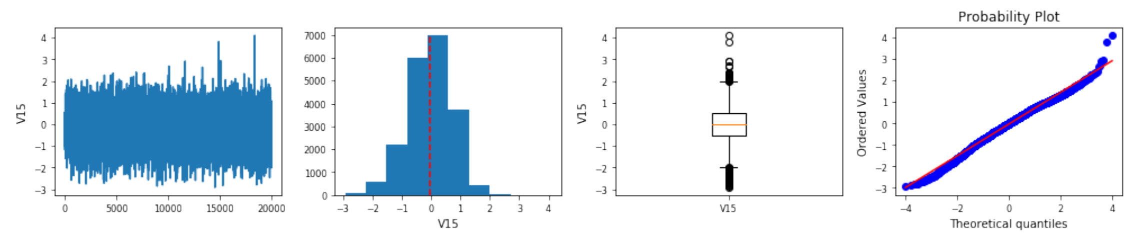
myplots V12



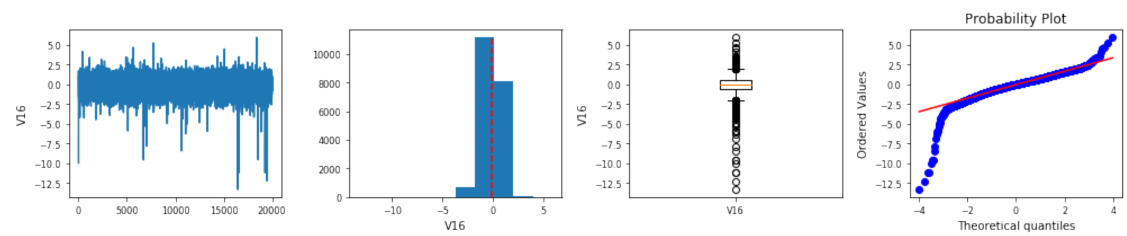
myplots V13



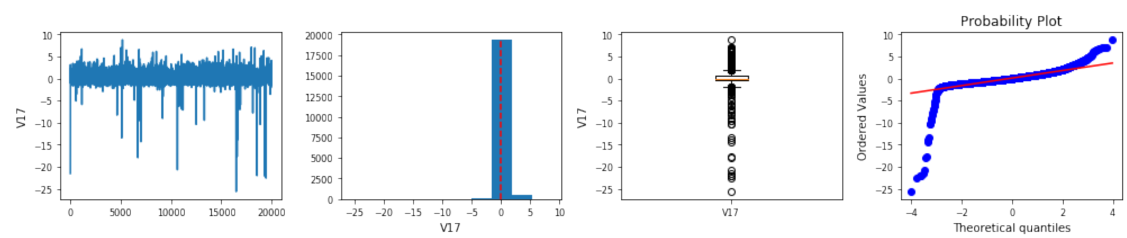
myplots V14



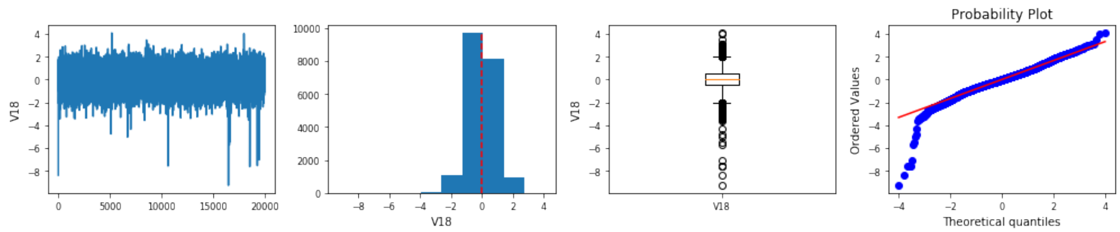
myplots V15



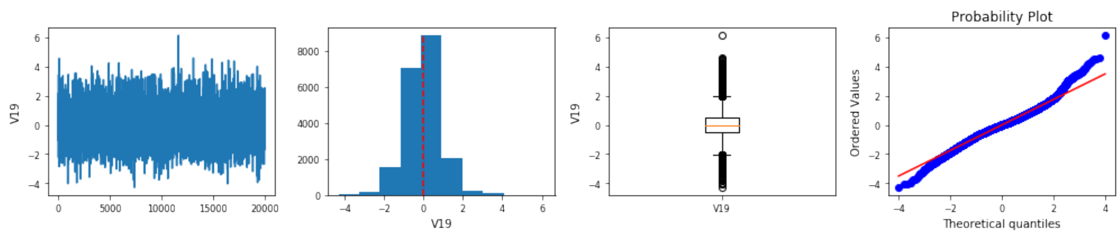
myplots V16



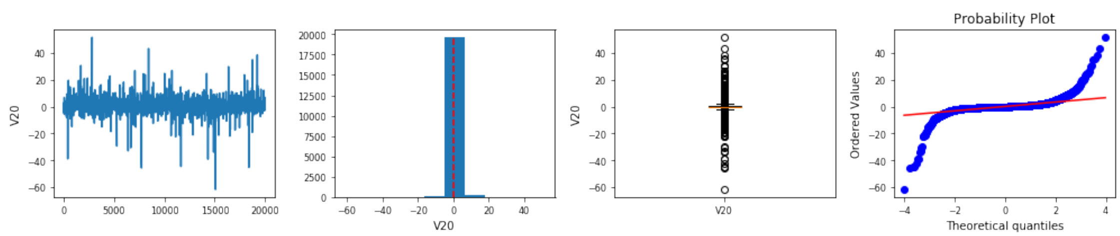
myplots V17



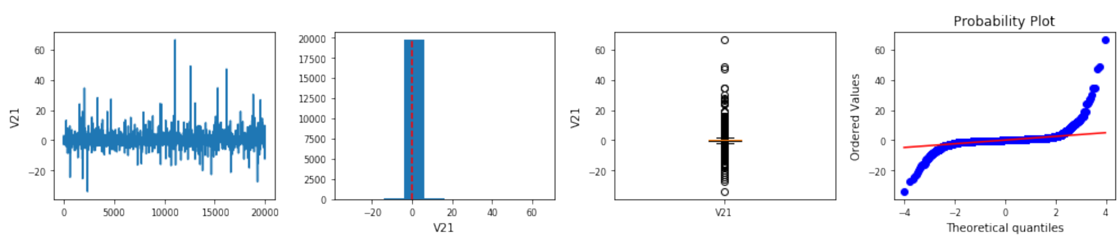
myplots V18



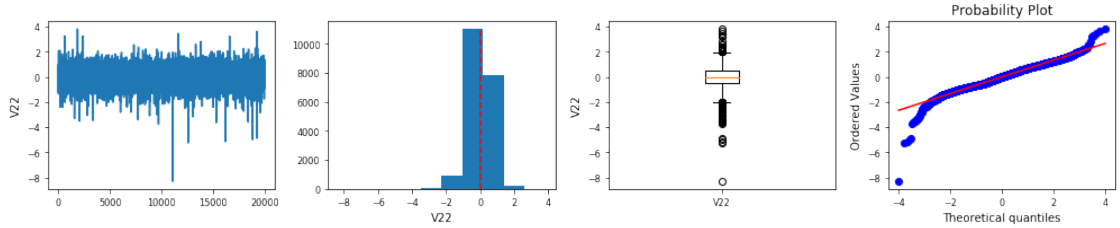
myplots V19



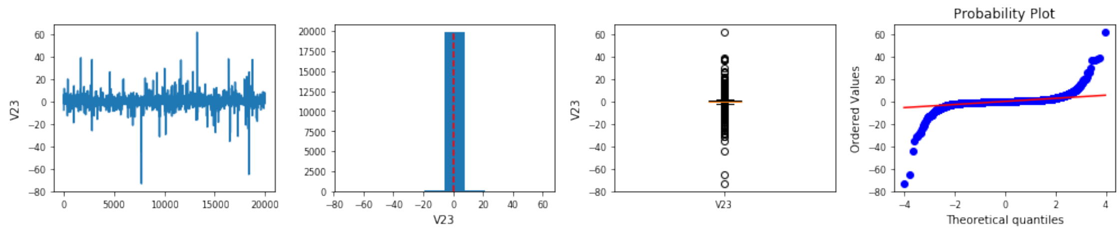
myplots V20



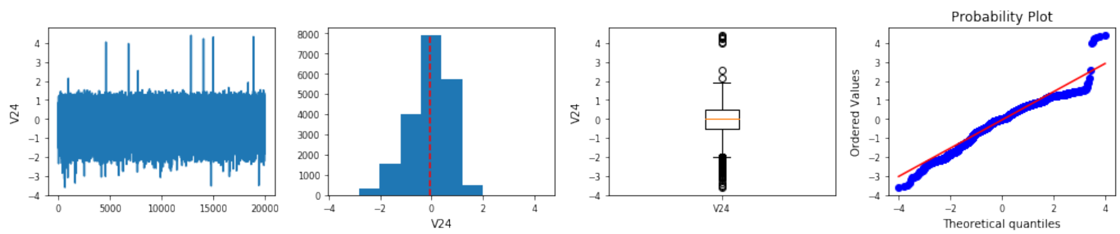
myplots V21



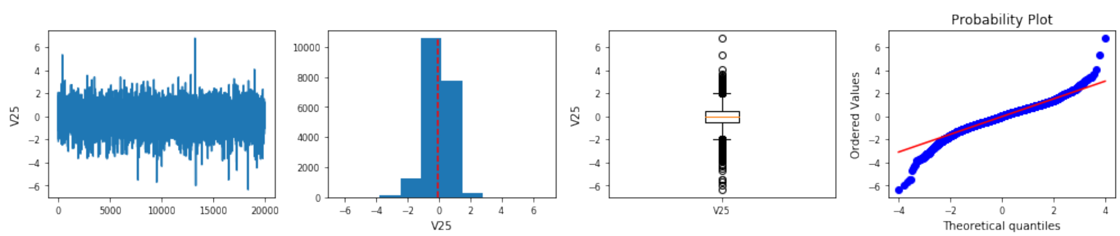
myplots V22



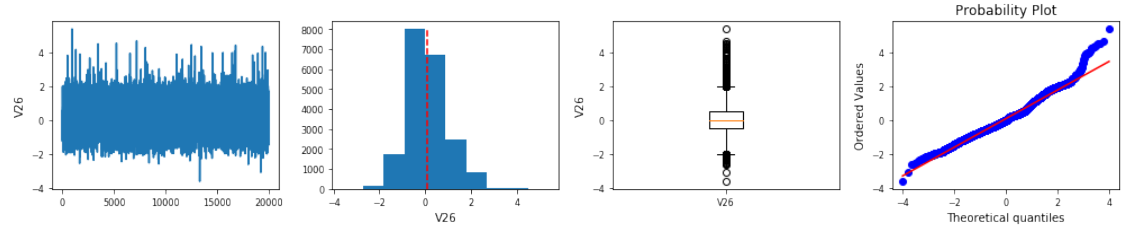
myplots V23



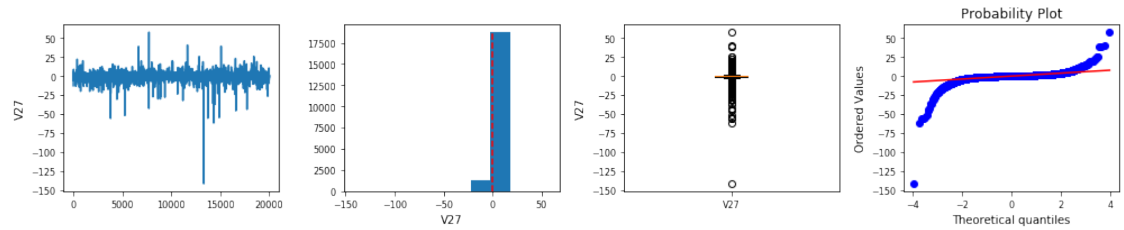
myplots V24



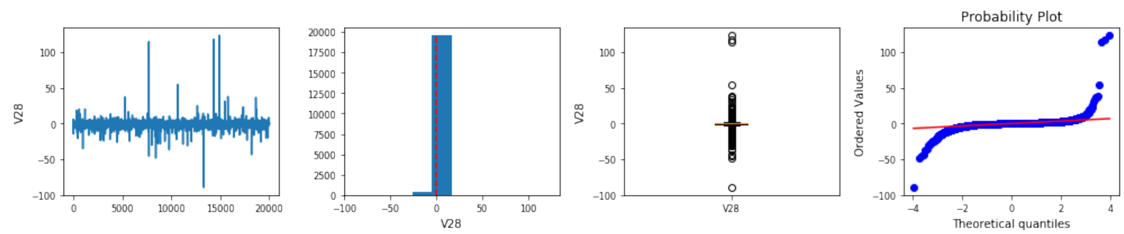
myplots V25



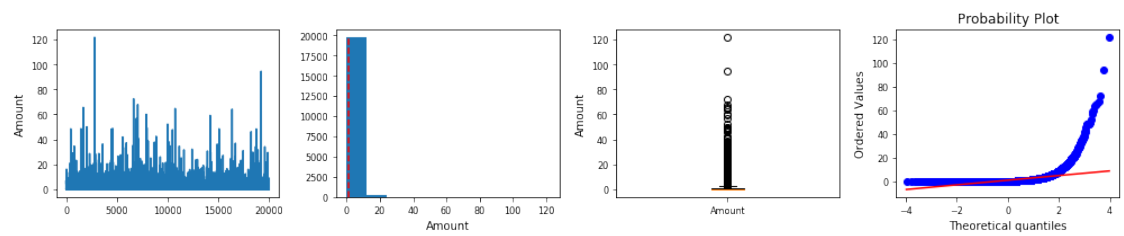
myplots V26



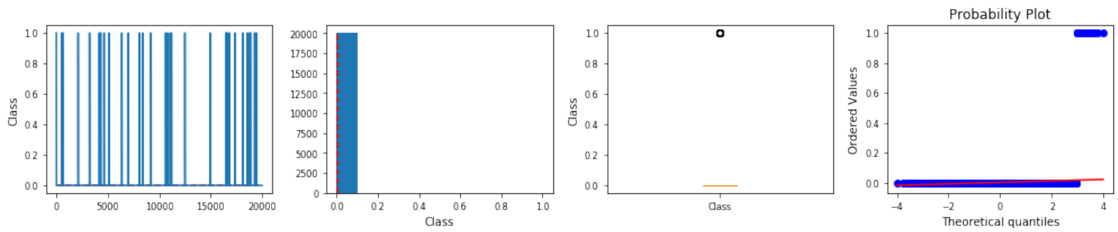
myplots V27



myplots V28



myplots Amount



myplots Class

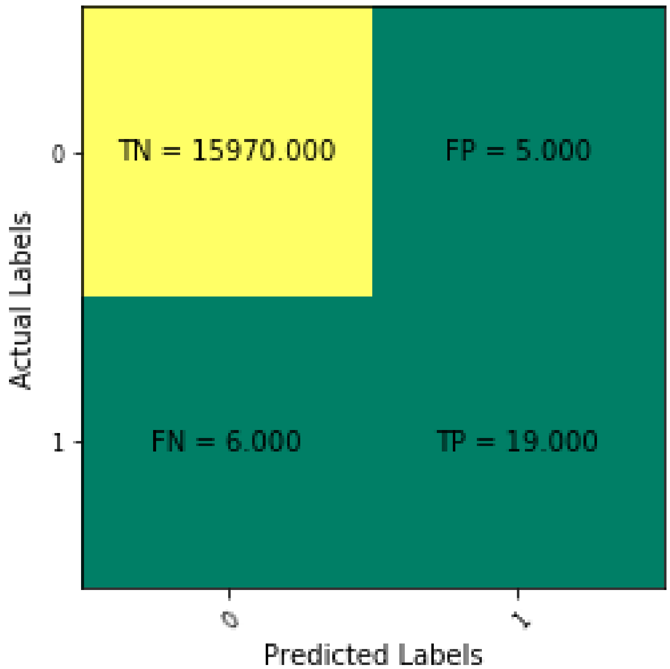
5 SVMs: EXPLORATION

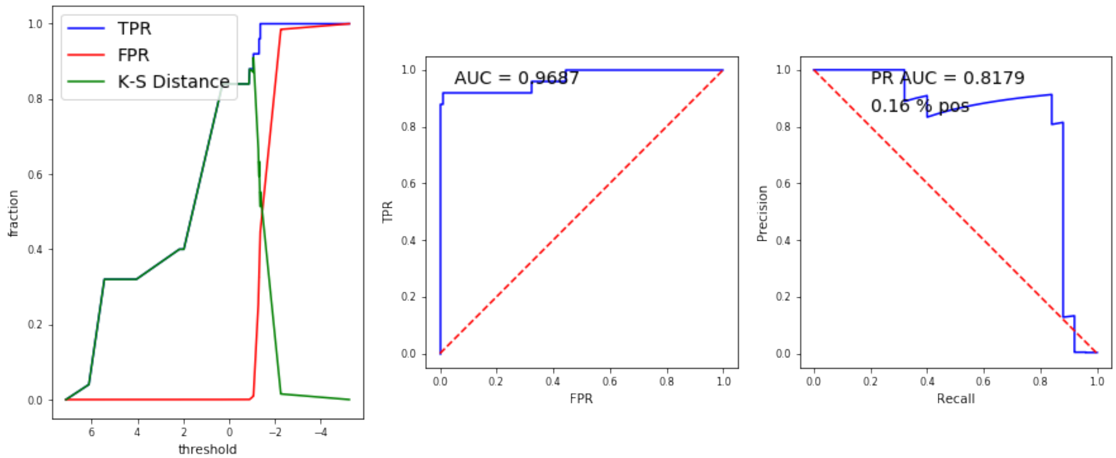
[17]:
```
""" TODO
Hold out a subset of the data, before training and cross validation
using train_test_split, with stratify NOT equal to None, and a test_size
fraction of .2.

For this exploratory section, the held out set of data is a validation set.
For the GridSearch section, the held out set of data is a test set.
"""
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪stratify= y)
```

[18]:
```
""" TODO
Create and train SVC models.
Explore various configurations of the hyper-parameters.
Train the models on the training set and evaluate them for the training and
validation sets.

Play around with C, gamma, and class_weight. Feel free to play with other hyper-
parameters as well. See the API for more details.
C is a regularization parameter, gamma is the inverse of the radius of influence
of the support vectors (i.e. lower gamma means a higher radius of influence of␣
 ↪the
support vectors), and class weight determines whether to adjust the weights␣
 ↪inversely
to the class fractions.
"""
#{'C': 10, 'class_weight': None, 'gamma': 0.0016, 'tol': 0.0001}
#classifier = SVC(C = 10, gamma = 0.0016, tol=0.0001, class_weight=None)
classifier = SVC(kernel = 'linear', C=2) #76, 78
#classifier = SVC(kernel = 'poly', C= 1.0, degree = 2, gamma = 'auto',␣
 ↪probability = True) #64, 73
```

[19]:
```python
""" TODO
Evaluate training set performance.
Display the confusion matrix, KS plot with
the cumulative distributions of the TPR and FPR, the ROC curve and the
precision-recall curve (PRC). use metrics_plots.ks_roc_prc_plot(ytrue, scores)

The PRC, unlike the AUC, does not consider the true negative (i.e. TN) counts,
making the PRC more robust to unbalanced datasets.
"""
# TODO: Confusion matrix
# First, compute the predictions for the training set
# Second, use confusion_matrix
# Third, use metrics_plots.confusion_mtx_colormap() to display the matrix
preds = cross_val_predict(classifier, X_train, y_train.values.ravel(), cv= 4)
confusion = confusion_matrix(y_train.values.ravel(), preds)
metrics_plots.confusion_mtx_colormap(confusion, [0,1],[0,1])

# TODO: Curves
# First, use the model's decision function to compute the scores
# Second, use metrics_plots.ks_roc_prc_plot() to display the KS plot, ROC, and␣
↪PRC

classifier.fit(X_train, y_train.values.ravel())
scores = classifier.decision_function(X_train)
metrics_plots.ks_roc_prc_plot(y_train, scores)



pss_train = metrics_plots.skillScore(y_train.values, preds)
f1_train = f1_score(y_train.values.ravel(), preds)
print("PSS: %.4f" % pss_train[0])
print("F1 Score %.4f" % f1_train)
```

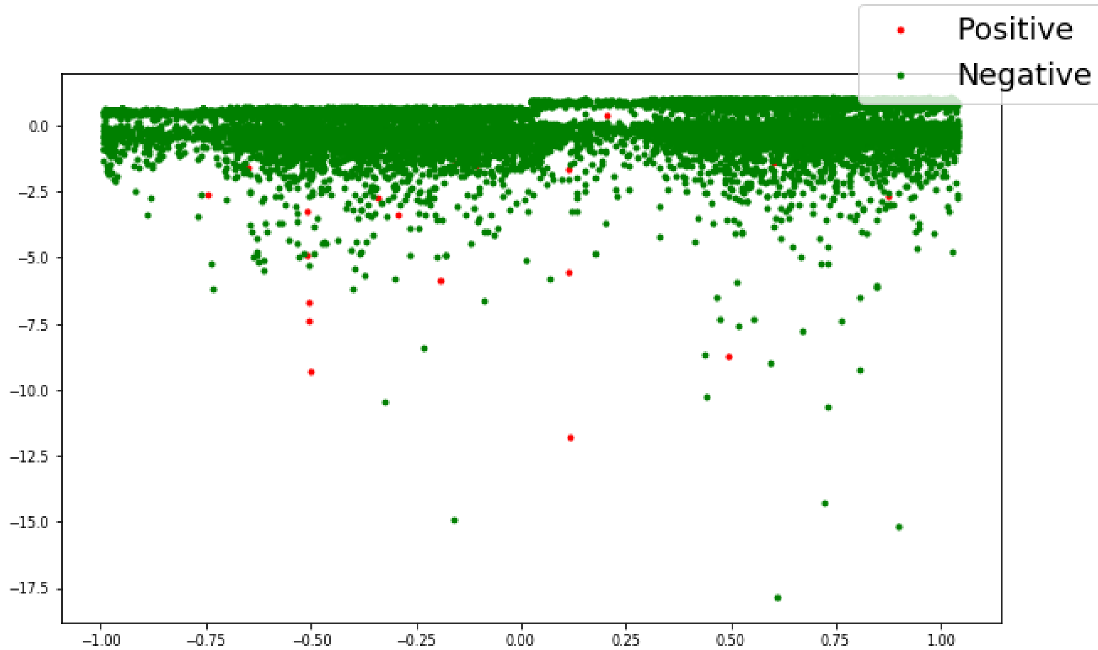
```
ROC AUC: 0.9686585289514867
PRC AUC: 0.8178628858521355
PSS: 0.7597
F1 Score 0.7755
```





[20]:
```
def scatter_plot(ins, pred):
    elems_true = np.where(pred == 1)[0]
    elems_false = np.where(pred == 0)[0]

    fig, ax = plt.subplots(figsize = (10,6))
    ax.plot(ins[elems_true, 0], ins[elems_true,1], 'r.')
    ax.plot(ins[elems_false, 0], ins[elems_false,1], 'g.')
    fig.legend(['Positive', 'Negative'], fontsize = 18)
```

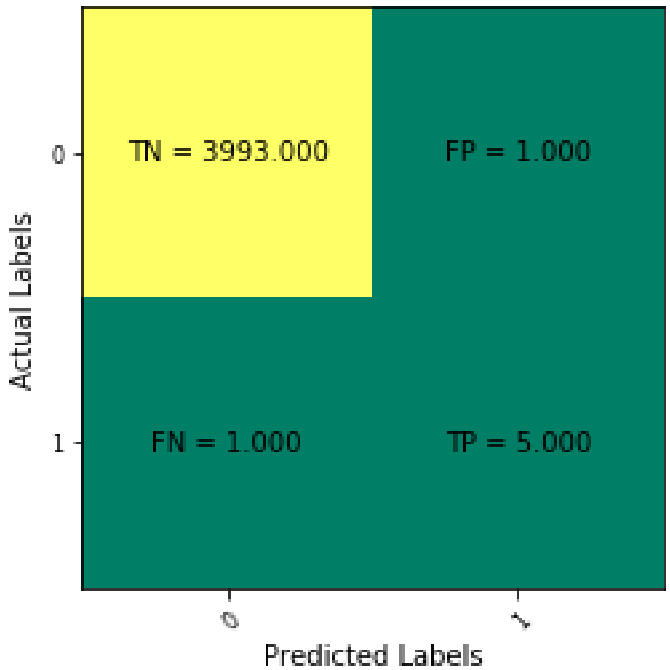
[21]:
```python
scatter_plot(X_train, preds)
```

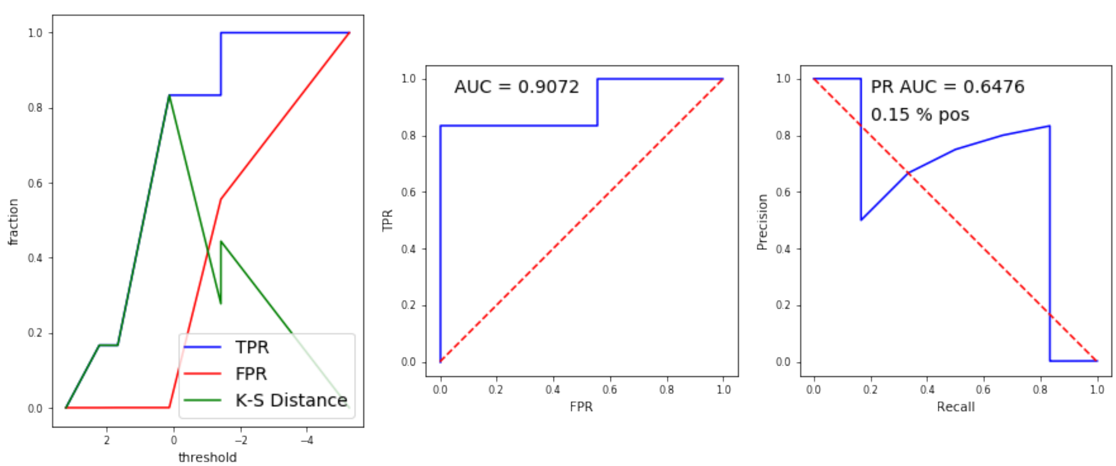


[22]:
```python
""" TODO
Evaluate validation performance.
Display the confusion matrix, KS plot with the cumulative distributions of the␣
↪TPR
and FPR, the ROC curve and the precision-recall curve (PRC).
"""
# TODO: Confusion matrix
pred_val = classifier.predict(X_val)
#pred_val = cross_val_predict(classifier, X_val, y_val.values.ravel(), cv= 4)
confusion = confusion_matrix(y_val.values.ravel(), pred_val)
metrics_plots.confusion_mtx_colormap(confusion, [0,1],[0,1])


# TODO: Curves
scores = classifier.decision_function(X_val)
metrics_plots.ks_roc_prc_plot(y_val, scores)





pss_test = metrics_plots.skillScore(y_val.values, pred_val)
f1_test = f1_score(y_val.values.ravel(), pred_val)
print("PSS: %.4f" % pss_test[0])
```

```
print("F1 Score %.4f" % f1_test)
```

```
ROC AUC: 0.9071523952595559
PRC AUC: 0.6476339213010088
PSS: 0.8331
F1 Score 0.8333
```





6 SVMs: STRATIFIED GRID SEARCH

6.1 Scorers

[23]:
```python
""" PROVIDED
List of available scoring functions from the sklearn module
"""
import sklearn
sorted(sklearn.metrics.SCORERS.keys())
```

[23]:
```
['accuracy',
 'adjusted_mutual_info_score',
 'adjusted_rand_score',
 'average_precision',
 'balanced_accuracy',
 'brier_score_loss',
 'completeness_score',
 'explained_variance',
 'f1',
 'f1_macro',
 'f1_micro',
 'f1_samples',
 'f1_weighted',
 'fowlkes_mallows_score',
 'homogeneity_score',
 'mutual_info_score',
 'neg_log_loss',
 'neg_mean_absolute_error',
 'neg_mean_squared_error',
 'neg_mean_squared_log_error',
 'neg_median_absolute_error',
 'normalized_mutual_info_score',
 'precision',
 'precision_macro',
 'precision_micro',
 'precision_samples',
 'precision_weighted',
 'r2',
 'recall',
 'recall_macro',
 'recall_micro',
 'recall_samples',
 'recall_weighted',
 'roc_auc',
 'v_measure_score']
```

6.2 Execute Grid Search

[24]:
```
""" TODO
Estimated time: ~30 min on mlserver
Set up and run the grid search using GridSearchCV and the following
settings:
* SVC for the model,
* The above scoring dictionary for scoring,
* refit set to 'f1' as the optimized metric
* Three for the number of cv folds,
* n_jobs=3,
* verbose=2,
* return_train_score=True
"""
# Optimized metric
opt_metric = 'f1'
scoring = {'f1':'f1'}

# Flag to re-load previous run
force = False
# File previous run is saved to
srchfname = "hw8_search_" + opt_metric + ".pkl"

# SETUP EXPERIMENT HYPERPARAMETERS
Cs = [.5, 1, 10, 100, 200]
gammas = np.logspace(-4, 0, num=5, endpoint=True, base=5)

nCs = len(Cs)
ngammas = len(gammas)

hyperparams = {'C':Cs, 'gamma':gammas, 'tol':[1e-4],
               'class_weight':[None, 'balanced']}

# RUN EXPERIMENT
time0 = timelib.time()
search = None
if force or (not os.path.exists(srchfname)):
    # TODO: Create the GridSearchCV object
    search = GridSearchCV(SVC(), param_grid=hyperparams, scoring=scoring,
                          refit=opt_metric, cv=3, n_jobs=3,
                          verbose=2, return_train_score=True)

    # TODO: Execute the grid search by calling fit using the training data
    search.fit(X_train, y_train)

    # TODO: Save the grid search object
    joblib.dump(search, srchfname)
```

```
    print("Saved %s" % srchfname)
else:
    search = joblib.load(srchfname)
    print("Loaded %s" % srchfname)

time1 = timelib.time()
duration = time1 - time0
print("Elapsed Time: %.2f min" % (duration / 60))

search
```

```
Loaded hw8_search_f1.pkl
Elapsed Time: 0.00 min
```

[24]:
```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False),
       fit_params=None, iid='warn', n_jobs=3,
       param_grid={'C': [0.5, 1, 10, 100, 200], 'gamma': array([0.0016, 0.008 ,
0.04  , 0.2   , 1.    ]), 'tol': [0.0001], 'class_weight': [None, 'balanced']},
       pre_dispatch='2*n_jobs', refit='f1', return_train_score=True,
       scoring={'f1': 'f1'}, verbose=2)
```

[27]:
```
search.best_params_
```

[27]:
```
{'C': 10, 'class_weight': None, 'gamma': 0.0016, 'tol': 0.0001}
```

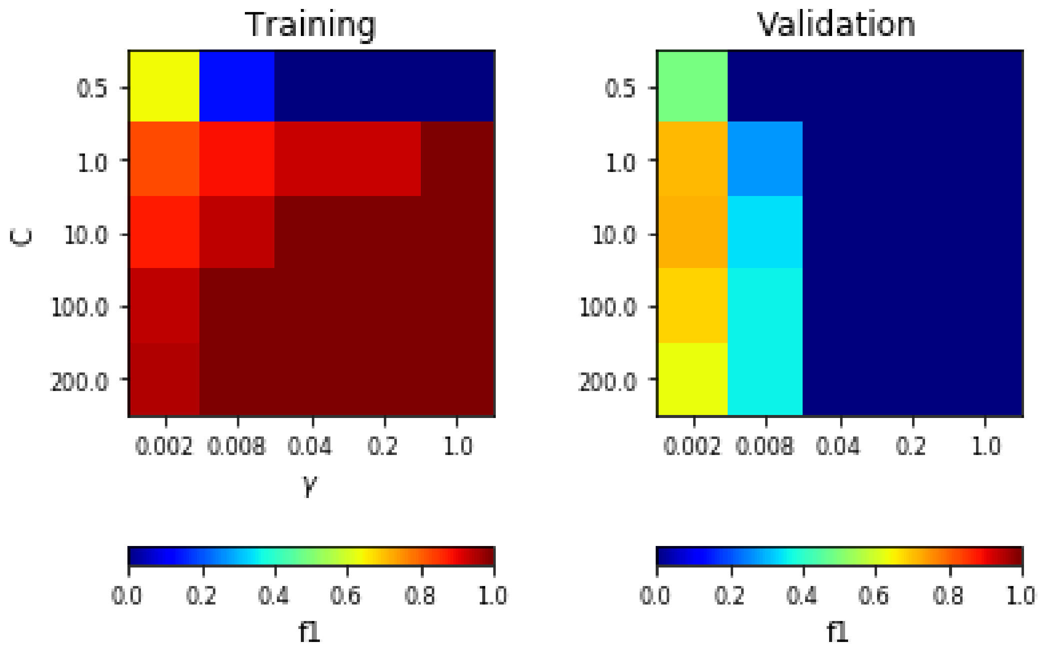
[96]:
```
# RESULTS

""" PROVIDED
Display the head of the results for the grid search
See the cv_results_ attribute
"""
all_results = search.cv_results_
df_res = pd.DataFrame(all_results)
df_res.head()

""" PROVIDED
Plot the mean training and validation results from the grid search as a
colormap, for C (y-axis) vs the gamma (x-axis), for class_weight=None
"""
results_grid_train = df_res['mean_train_'+opt_metric].values.reshape(nCs, 2,␣
↪ngammas)
```

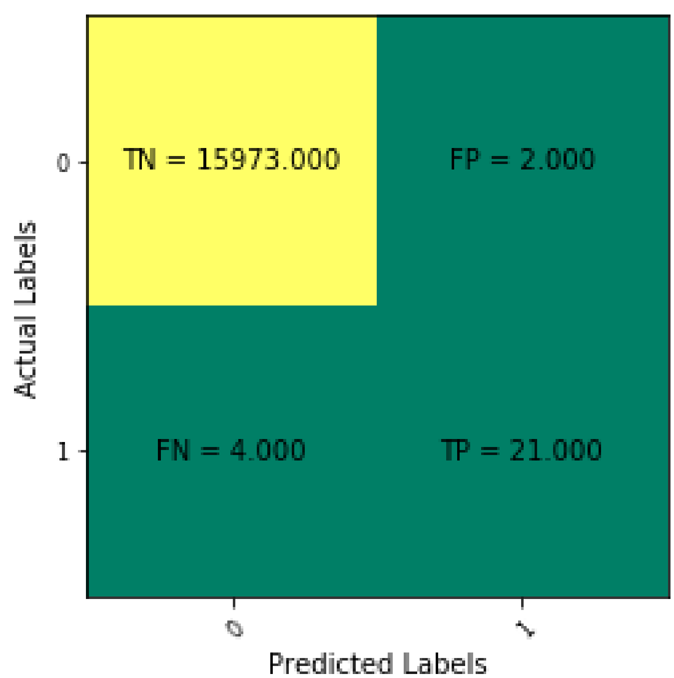
```python
results_grid_val = df_res['mean_test_'+opt_metric].values.reshape(nCs, 2,␣
↪ngammas)

fig, axs = plt.subplots(1, 2, figsize=(6,6))
fig.subplots_adjust(wspace=.45)
axs = axs.ravel()
means = [("Training", results_grid_train),
         ("Validation", results_grid_val)]
for i, (name, result) in enumerate(means):
    img = axs[i].imshow(result[:,0,:], cmap="jet", vmin=0, vmax=1)
    axs[i].set_title(name)
    axs[i].set_xticks(range(ngammas))
    axs[i].set_yticks(range(nCs))
    axs[i].set_xticklabels(np.around(gammas, 3))
    axs[i].set_yticklabels(np.around(Cs, 3))
    axs[i].figure.colorbar(img, ax=axs[i], label=opt_metric,
                           orientation='horizontal')
    if i == 0:
        axs[i].set_xlabel(r"$\gamma$")
        axs[i].set_ylabel("C")
#fig.suptitle('class_weight=None')

""" TODO
Obtain the best model from the grid search and
fit it to the full training data
"""
classifier_best = SVC(C = search.best_params_['C'],
                 gamma = search.best_params_['gamma'],
                 tol=search.best_params_['tol'],
                 class_weight=search.best_params_['class_weight'])

#preds_best = cross_val_predict(classifier_best, X_train, y_train.values.
↪ravel(), cv= 4)
classifier_best.fit(X_train, y_train.values.ravel())

preds_best = classifier_best.predict(X_train)

""" TODO
For the best model, display the confusion matrix, KS plot, ROC curve,
and PR curve for the training set
"""
# TODO: Confusion Matrix
confusion = confusion_matrix(y_train.values.ravel(), preds_best)
metrics_plots.confusion_mtx_colormap(confusion, [0,1],[0,1])


# TODO: Curves
```

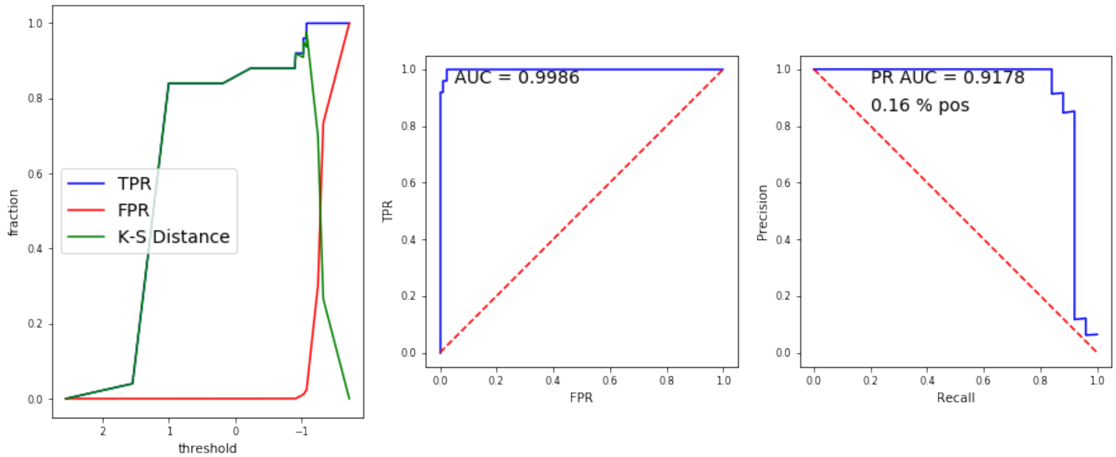
```python
scores_best = classifier_best.decision_function(X_train)
metrics_plots.ks_roc_prc_plot(y_train, scores_best)



pss_res = metrics_plots.skillScore(y_train.values, preds_best)
f1_res = f1_score(y_train.values.ravel(), preds_best)
print("PSS: %.4f" % pss_res[0])
print("F1 Score %.4f" % f1_res)

""" TODO
For the best model, display the confusion matrix, KS plot, ROC curve,
and PR curve for the test set
"""
# TODO: Confustion Matrix
preds_test = classifier_best.predict(X_val)
#preds_test = cross_val_predict(classifier_best, X_val, y_val.values.ravel(),␣
↪cv= 4)
confusion = confusion_matrix(y_val.values.ravel(), preds_test)
metrics_plots.confusion_mtx_colormap(confusion, [0,1],[0,1])


# TODO: Curves
scores = classifier_best.decision_function(X_val)
metrics_plots.ks_roc_prc_plot(y_val, scores)


pss_res_test = metrics_plots.skillScore(y_val.values, preds_test)
f1_res_test = f1_score(y_val.values.ravel(), preds_test)
print("PSS: %.4f" % pss_res_test[0])
print("F1 Score %.4f" % f1_res_test)

""" TODO
Plot a histogram of the test scores from the best model.
Compare the distribution of scores for positive and negative examples
using boxplots.

Create one subplot of the distribution of all the scores, with a histogram.
Create a second subplot comparing the distribution of the scores of the
positive examples with the distribution of the negative examples, with boxplots.
"""
# TODO: Obtain the pos and neg indices
pos = [i for i, _ in enumerate(preds_test) if _ == True]
neg = [i for i, _ in enumerate(preds_test) if _ == False]


# TODO: Separate the scores for the pos and neg examples
```

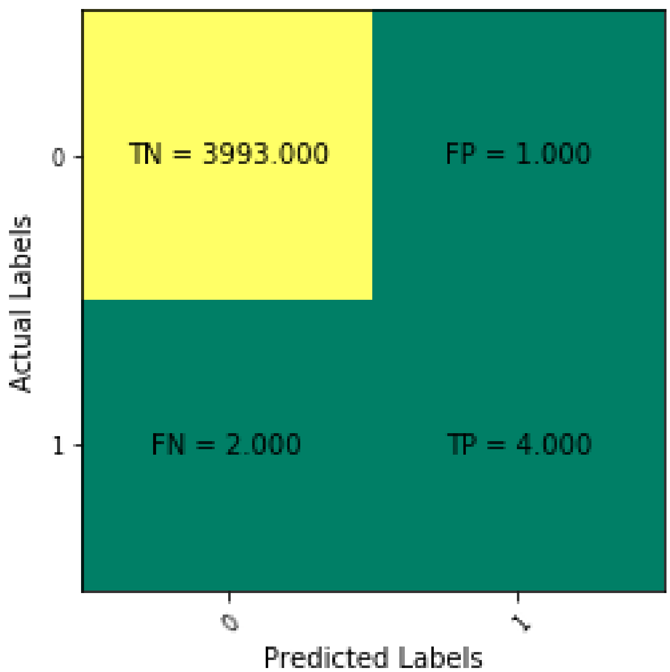
```python
scores_pos = scores[pos]
scores_neg = scores[neg]


# TODO: Plot the distribution of all scores
nbins = 100
FIGWIDTH = 10
FIGHEIGHT = 2
fig, axs = plt.subplots(1, figsize=(FIGWIDTH*2, FIGHEIGHT*5))

_ = plt.hist(scores, align = 'mid', bins = nbins)
plt.title('Test scores from the best model')
plt.xlabel('score')
plt.ylabel('score count')
plt.show()

# TODO: Plot the boxplots of the pos and neg examples

FIGWIDTH = 10
FIGHEIGHT = 2
fig, axs = plt.subplots(1, 2, figsize=(FIGWIDTH*2, FIGHEIGHT*5))

axs[0].boxplot(scores_pos)
axs[0].title.set_text('positive score distribution')
fig.show()
axs[1].boxplot(scores_neg)
axs[1].title.set_text('negative scores distribution')
fig.show()
```

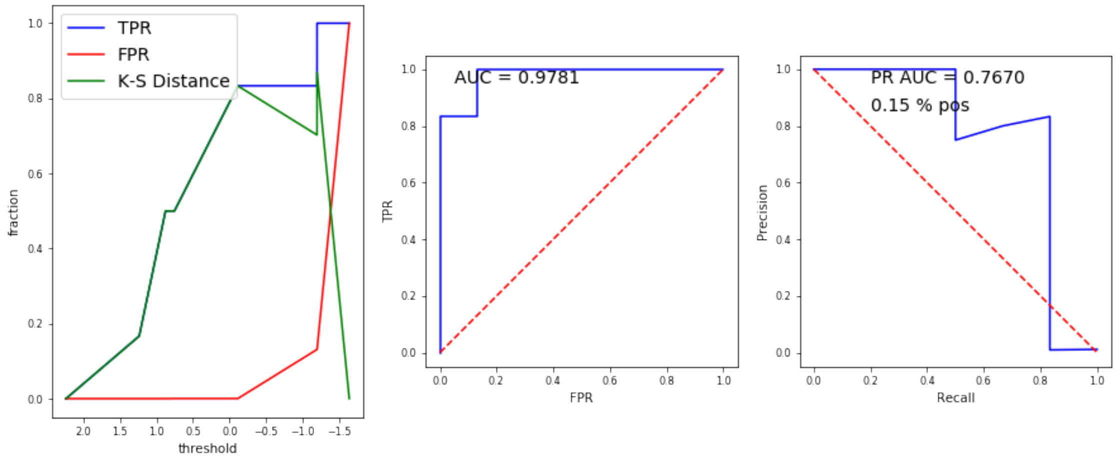
```
ROC AUC: 0.9986353677621284
PRC AUC: 0.917829569107623
PSS: 0.8399
F1 Score 0.8750
ROC AUC: 0.9780921382073109
PRC AUC: 0.7670120987760403
PSS: 0.6664
F1 Score 0.7273
```

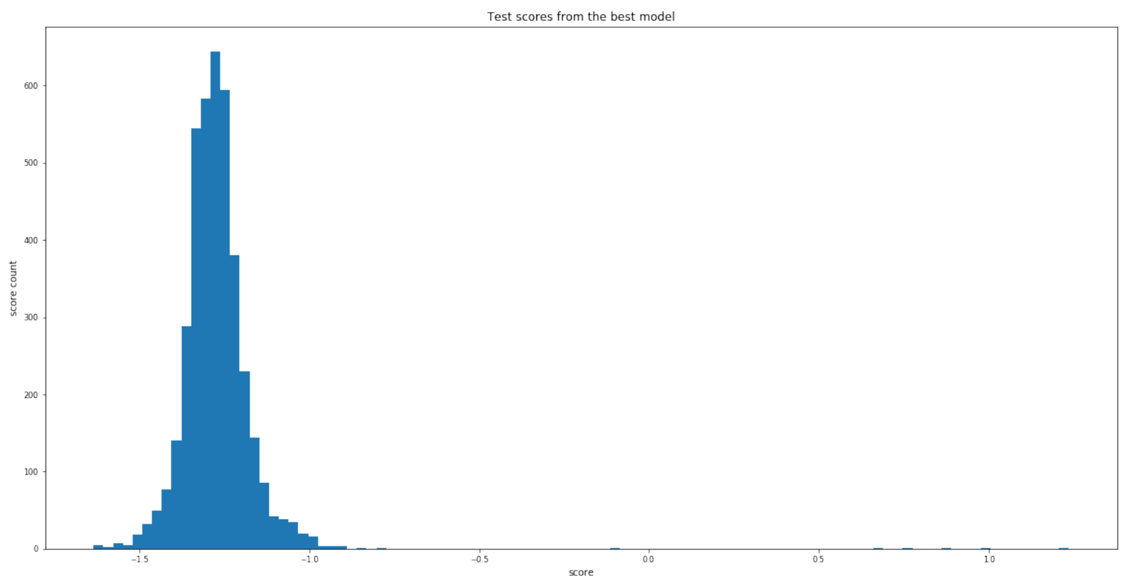

Training
Validation
C
0.5
1.0
10.0
100.0
200.0
0.002
0.008
0.04
0.2
1.0
γ
0.0
0.2
0.4
0.6
0.8
1.0
f1

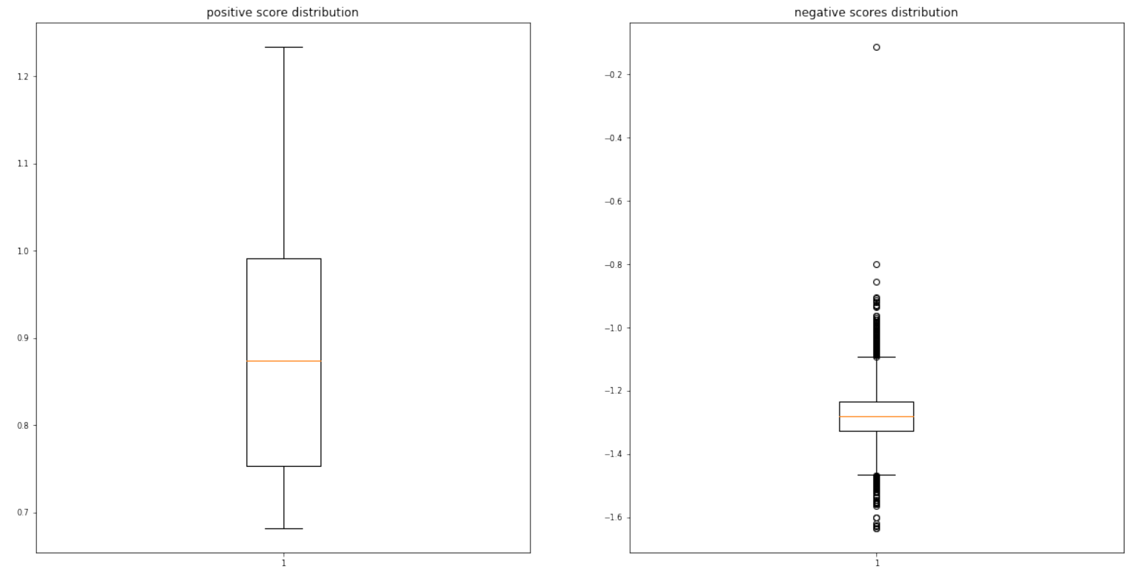

TN = 15973.000
FP = 2.000
FN = 4.000
TP = 21.000
Actual Labels
Predicted Labels
0
1


TPR
FPR
K-S Distance
fraction
threshold
AUC = 0.9986
TPR
FPR
PR AUC = 0.9178
0.16 % pos
Precision
Recall


TN = 3993.000
FP = 1.000
FN = 2.000
TP = 4.000
Actual Labels
Predicted Labels


TPR
FPR
K-S Distance
fraction
threshold
AUC = 0.9781
TPR
FPR
PR AUC = 0.7670
0.15 % pos
Precision
Recall


Test scores from the best model
score count
score



7 Discussion

#In 3 to 4 paragraphs, discuss and interpret the test results for the best model. Include a brief discussion of the difference in the meaning of the AUC for the ROC vs the AUC for the PRC. Also, discuss the histogram and boxplots of the scores.

My test results for the best model have a PSS of PSS: 0.6664and an F1 score of F1 Score 0.7273. My ROC AUC is 0.9781 and my PRC AUC is 0.7670. The PSS is a precision measurement score that is calculated by subtracting the false positive ratio from the true positive ratio. A high score here is best. My F1 score is calculated by taking the precision recall and multiplying it by the recall and then dividing that product with the sum of precision and recall, then doubling. This F1 Score is often known as the harmonic mean. The F1 score is like the bias variance trade off in machine learning in that it makes sure we are considering both the importance of recall and precision when acknowledging our classification results. My PSS and F1 scores leave room for improvement. Improvement might be seen in a larger testing size or with a different kernel. I hope I can further explore this data with other models to see improvement in my scoring.

The difference between my ROC AUC and PRC AUC scores comes from the difference between the metrics of confusion being measured. This difference goes back to the importance in understanding the tradeoff that I mentioned above. The FPR AUC is the false positive rate area under the curve. A perfect score of 1 here wasn't achieved, but the rate between false positive and true positive is good here so the score is quite high. My recall rate goes up down and all around before reaching its score of 0.7273. No recall would have made this score much higher, but unfortunately my test results were not perfect, and I didn't achieve this metric. The two scores differ in their measurements and they give us differing information, both of which provide their own respective value and insight. Despite the imperfection, the recall rate and the false positive rate seem quite acceptable in the testing sense.

Finally, my scoring plots give some interesting insight into what makes a positive test and what makes a negative test. When we look at the histogram for all the scoring data, we see that a vast majority of the scores are less than -1. But when we look at the boxplots we can see that we do in fact have some positive scores and they are concentrated between 0.8 and 1.0. The score values represent the distance from the hyperplane created by the SVM. The SVM creates a barrier that helps to identity which points belong to which classifier. Looking at these metrics, we can see into the graphical representation of the SVM and understand more deeply what it is doing.

```
[ ]:
```