Final code with MLP

April 19, 2018

```
In [ ]: from sklearn.ensemble import RandomForestClassifier
        from sklearn.linear_model import LogisticRegression
        from sklearn.feature_selection import SelectKBest
        from sklearn.feature_selection import chi2
        from sklearn.metrics import roc_auc_score
        from sklearn.tree import DecisionTreeClassifier
        #deep
        from sklearn.neural_network import MLPClassifier
        from sklearn.neural_network import MLPRegressor

        import datetime
        import pandas as pd
        import numpy as np
        import time


        def log(text, t_start=None):
            if t_start is None:
                print(text)
            else:
                elapsed_time = round(time.time() - t_start, 2)
                print(text + "\t(" + str(elapsed_time) + "s)")

        t = time.time()
        customers = pd.read_csv("data/customers.csv")
        products = pd.read_csv("data/products.csv")
        x_train = pd.read_csv("data/X_train.csv")
        y_train = pd.read_csv("data/y_train.csv")
```

```
x_test = pd.read_csv("data/X_test.csv")
log("files loaded", t)

# SizeAdviceDescription
SizeAdviceDescriptionCleaner = {}
SizeAdviceDescriptionCleaner['nan'] = 0
SizeAdviceDescriptionCleaner['Ce mod\xc3\x83\xc2\xa8le chausse normalement'] = 0
SizeAdviceDescriptionCleaner['Mod\xc3\x83\xc2\xa8le confortable, convient aux pieds larges'] = -.5
SizeAdviceDescriptionCleaner['Mod\xc3\x83\xc2\xa8le \xc3\x83\xc2\xa9troit, convient aux pieds fins'] = .5
SizeAdviceDescriptionCleaner['Prenez votre pointure habituelle'] = 0
SizeAdviceDescriptionCleaner['Chaussant particuli\xc3\x83\xc2\xa8rement g\xc3\x83\xc2\xa9n\xc3\x83\xc2\xa9reux. Nous vous conseill
SizeAdviceDescriptionCleaner['Chaussant petit. Si vous \xc3\x83\xc2\xaates habituellement entre deux pointures, nous vous conseill
SizeAdviceDescriptionCleaner['Prenez une taille au-dessus de sa pointure !'] = 1
SizeAdviceDescriptionCleaner['Prenez une taille au-dessus de votre pointure habituelle'] = 1
SizeAdviceDescriptionCleaner['Prenez une taille en dessous de sa pointure !'] = -1
SizeAdviceDescriptionCleaner['Prenez une taille en dessous de votre pointure habituelle'] = -1

# BirthDate
def age(birthdate):
    if type(birthdate) == type(" "):
        return 2016 - int(birthdate[:4])
    return None

# OrderCreationDate and SeasonLabel
def order_season(orderdate):
    month = int(orderdate[5:7])
    if month >= 4 and month <= 9:
        return "Printemps/Et\xc3\x83\xc2\xa9"
    return "Automne/Hiver"


def build_df(x):
    """Builds a pandas DataFrame with clean columns from a read CSV"""

    t = time.time()
    m = None

    # join
    m = pd.merge(x, products, how='left', on='VariantId', suffixes=('_pr', ''))
    m = pd.merge(m, customers, how='left', on='CustomerId', suffixes=('_cs', ''))

    # converting UnitPMPEUR
    m.UnitPMPEUR = m["UnitPMPEUR"].map(lambda row: float(row.replace(',', '.')))

    # building news columns
    m["MatchGender"] = m["Gender"] == m["GenderLabel"]
    m["MatchSeason"] = m["SeasonLabel_pr"] == m["SeasonLabel"]
    m["OrderSeason"] = m["OrderCreationDate"].map(order_season)
    m["MatchOrderSeason"] = m["OrderSeason"] == m["SeasonLabel"]

    # cleaning
    m["SizeAdviceDescription"] = m["SizeAdviceDescription"].map(SizeAdviceDescriptionCleaner)
    m["BirthDate"] = m["BirthDate"].map(age)

    # removing useless columns
    blacklist = ['VariantId', 'CustomerId', 'OrderNumber', 'LineItem',
                 'ProductColorId', 'BrandId', 'SupplierColor', 'OrderShipDate',
                 'ProductId', 'BillingPostalCode', 'FirstOrderDate',
                 'OrderStatusLabel', 'MinSize', 'MaxSize', 'OrderSeason',
                 'OrderCreationDate', 'SubtypeLabel', 'ProductType'
                ]
    whitelist = None
    if blacklist is not None:
        m = m.drop(blacklist, axis=1)
    if whitelist is not None:
        for col in m.columns:
            if col not in whitelist:
                m = m.drop([col], axis=1)

    print "dataframe shape:", m.shape
    log("dataframe built", t)
    return m


df_test = build_df(x_test)
df_train = build_df(x_train)
```

```
def mask(m):
    columns2bin = [col for col in m.columns if m[col].dtype == 'object']
    other_cols = m.drop(columns2bin, axis=1)
    new_cols = pd.get_dummies(m.loc[:, columns2bin])
    res = pd.concat([other_cols, new_cols], axis=1)
    res = res.fillna(0)
    print "new shape:", res.shape
    return res

def compute(name, clf, x1, x2, slc=100000):
    print "\n-----", name, "-----"
    clf.fit(x1.iloc[:slc], y_train.ReturnQuantityBin[:slc])

    predict_train = clf.predict_proba(x1.iloc[:slc])
    score_train = roc_auc_score(y_train.ReturnQuantityBin[:slc], predict_train[:, 1])
    print "train score:", score_train

    predict_test = clf.predict_proba(x1.iloc[slc:2 * slc])
    score_test = roc_auc_score(y_train.ReturnQuantityBin[slc:2 * slc], predict_test[:, 1])
    print "test score:", score_test
    return score_train, score_test

def compute_all(x1, x2, slc=100000):
    """Tries different classifiers and returns the best one (best test score)"""
    t = time.time()
    best_index, best_score = None, None

    print "train shape:\t", x1.shape, "\t", y_train.shape
    print "test shape:\t", x2.shape, "\t", y_test.shape

    classifiers = [("random forest", RandomForestClassifier()),
                   ("decision tree", DecisionTreeClassifier()),
                   ("logistic regression", LogisticRegression()),
                    ("DEEP",MLPClassifier(solver='lbfgs', alpha=1e-5,
                    hidden_layer_sizes=(100, 10), random_state=1))]

    for i, (name, clf) in enumerate(classifiers):
        score_train, score_test = compute(name, clf, x1, x2, slc)
        if best_score is None or score_test > best_score:
            best_index, best_score = i, score_test

    log("\nbest classifier: " + classifiers[best_index][0], t)
    return classifiers[best_index][1]

def output(clf, x1, x2):
    t = time.time()
    y_tosubmit = clf.predict_proba(x2.loc[:, x1.columns].fillna(0))

    timestamp = '{0:%Y_%m_%d_%H_%M_%S}'.format(datetime.datetime.now())
    filename = "ypred_{0}_sgd.txt".format(timestamp)
    np.savetxt(filename, y_tosubmit[:,1], fmt='%f')

    f = open("predictions.txt", 'a')
    f.write(timestamp + '\n' + repr(clf).replace('\n          ', '') + '\n\n')
    f.close()

    print "shape:", y_tosubmit.shape
    log("generated output at " + filename, t)


t = time.time()
x1 = mask(df_train)
x2 = mask(df_test)
log("applied mask", t)



def shuffle(x, y, steps=10, slc=100000, plot=True):
    scores_train, scores_test = [], []
    best_clf, best_score = None, None

    z = x.copy(deep=True)
    z["ReturnQuantityBin"] = y.ReturnQuantityBin

    for k in range(steps):
        u = z.sample(frac=1)
        v = u.loc[:, ["ReturnQuantityBin"]]
        u = u.drop(["ReturnQuantityBin"], axis=1)
        #clf = LogisticRegression()
        clf = MLPRegressor(solver='sgd', alpha=1e-5,
                    hidden_layer_sizes=(100, 100,100,100), random_state=1)
        clf.fit(u.iloc[:slc], v.ReturnQuantityBin[:slc])

        predict_train = clf.predict_proba(u.iloc[:slc])
        score_train = roc_auc_score(v.ReturnQuantityBin[:slc], predict_train[:, 1])

        predict_test = clf.predict_proba(u.iloc[slc:2 * slc])
        score_test = roc_auc_score(v.ReturnQuantityBin[slc:2 * slc], predict_test[:, 1])

        if best_clf is None or score_test > best_score:
            best_clf, best_score = clf, score_test

        if plot:
            print "test", k, "\ttrain:", score_train, "\ttest:", score_test

        scores_train.append(score_train)
        scores_test.append(score_test)

    if plot:
        plt.figure(figsize=(16, 10))
        plt.xlabel("train score")
        plt.ylabel("test score")
        plt.plot(scores_train, scores_test, '+')
        plt.show()

    return scores_train, scores_test, best_clf, best_score


sc_train, sc_test, clf, score = shuffle(x1, y_train, slc=100000, steps=10, plot=False)



output(clf, x1, x2)
```

Code with all trials

April 19, 2018

1 AlphaShoe

SD210 Challenge

- scoreboard and submissions
- starting kit
- training data
- dictionnary

1.0.1 Importations

```
In [1]: # coding: utf-8

        from sklearn.ensemble import RandomForestClassifier
        from sklearn.decomposition import PCA
        from sklearn.linear_model import LogisticRegression
        from sklearn.model_selection import cross_val_score
        from sklearn.model_selection import cross_val_predict
        from sklearn.feature_selection import SelectKBest
        from sklearn.feature_selection import chi2
        from sklearn.metrics import roc_auc_score
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.preprocessing import LabelEncoder
        from matplotlib import pyplot as plt
        import datetime
        import pandas as pd
        import numpy as np
        import time
        %matplotlib inline

        def log(text, t_start=None):
            if t_start is None:
                print(text)
            else:
                elapsed_time = round(time.time() - t_start, 2)
                print(text + "\t(" + str(elapsed_time) + "s)")
```

1.0.2 Loading files

```
In [2]: t = time.time()
        customers = pd.read_csv("data/customers.csv")
        products = pd.read_csv("data/products.csv")
        x_train = pd.read_csv("data/X_train.csv")
        y_train = pd.read_csv("data/y_train.csv")
        x_test = pd.read_csv("data/X_test.csv")
        y_test = pd.read_csv("data/y_test.csv")
        log("files loaded", t)
```

```
files loaded        (43.61s)
```

1.0.3 Data cleaning

```
In [3]: # SizeAdviceDescription
        SizeAdviceDescriptionCleaner = {}
        SizeAdviceDescriptionCleaner['nan'] = 0
        SizeAdviceDescriptionCleaner['Ce mod\xc3\x83\xc2\xa8le chausse normalement'] = 0
        SizeAdviceDescriptionCleaner['Mod\xc3\x83\xc2\xa8le confortable, convient aux pieds larges'] = -.5
        SizeAdviceDescriptionCleaner['Mod\xc3\x83\xc2\xa8le \xc3\x83\xc2\xa9troit, convient aux pieds fins'] = .5
        SizeAdviceDescriptionCleaner['Prenez votre pointure habituelle'] = 0
        SizeAdviceDescriptionCleaner['Chaussant particuli\xc3\x83\xc2\xa8rement g\xc3\x83\xc2\xa9n\xc3\x83\xc2\xa9reux. Nous vous conseill
        SizeAdviceDescriptionCleaner['Chaussant petit. Si vous \xc3\x83\xc2\xaates habituellement entre deux pointures, nous vous conseill
        SizeAdviceDescriptionCleaner['Prenez une taille au-dessus de sa pointure !'] = 1
        SizeAdviceDescriptionCleaner['Prenez une taille au-dessus de votre pointure habituelle'] = 1
        SizeAdviceDescriptionCleaner['Prenez une taille en dessous de sa pointure !'] = -1
        SizeAdviceDescriptionCleaner['Prenez une taille en dessous de votre pointure habituelle'] = -1
```

```
# datetime format:
# YYYY-MM-DD HH:mm:SS

# BirthDate
def age(birthdate):
    if type(birthdate) == type(" "):
        return 2016 - int(birthdate[:4])
    return None

def week_of_the_year(date):
    if type(date) == type(" "):
        year = int(date[:4])
        month = int(date[5:7])
        day = int(date[8:10])
        return datetime.date(year, month, day).isocalendar()[1]
    return None

def hour(date):
    if type(date) == type(" "):
        return int(date[11:13])
    return None

# OrderCreationDate and SeasonLabel
def order_season(orderdate):
    month = int(orderdate[5:7])
    if month >= 4 and month <= 9:
        return "Printemps/Et\xc3\x83\xc2\xa9"
    return "Automne/Hiver"

def build_df(x):
    """Builds a pandas DataFrame with clean columns from a read CSV"""

    t = time.time()
    m = None

    # join
    m = pd.merge(x, products, how='left', on='VariantId', suffixes=('_pr', ''))
    m = pd.merge(m, customers, how='left', on='CustomerId', suffixes=('_cs', ''))
    # converting UnitPMPEUR
    m.UnitPMPEUR = m["UnitPMPEUR"].map(lambda row: float(row.replace(',', '.')))

    # building news columns
    m["MatchGender"] = m["Gender"] == m["GenderLabel"]
    m["MatchSeason"] = m["SeasonLabel_pr"] == m["SeasonLabel"]
    m["OrderSeason"] = m["OrderCreationDate"].map(order_season)
    m["MatchOrderSeason"] = m["OrderSeason"] == m["SeasonLabel"]
    m["OrderHour"] = m["OrderCreationDate"].map(hour)
    m["FirstOrderDate"] = m["FirstOrderDate"].map(week_of_the_year)

    # cleaning
    m["SizeAdviceDescription"] = m["SizeAdviceDescription"].map(SizeAdviceDescriptionCleaner)
    m["BirthDate"] = m["BirthDate"].map(age)

    # removing useless columns
    blacklist = ['VariantId', 'CustomerId', 'OrderNumber', 'LineItem',
                 'ProductColorId', 'OrderShipDate',
                 'ProductId', 'BillingPostalCode', 'SupplierColor',
                 'OrderStatusLabel',
                 'OrderCreationDate', 'SubtypeLabel', 'ProductType',
                ]
    whitelist = None
    if blacklist is not None:
        m = m.drop(blacklist, axis=1)
    if whitelist is not None:
        for col in m.columns:
            if col not in whitelist:
                m = m.drop([col], axis=1)

    print "dataframe shape:", m.shape
    log("dataframe built", t)
    return m
```

1.0.4 Statistics

```
In [21]: def returns_frequency(x, y, col, step):
             """Returns the returns frequencies for each value of a column"""
```

```
    counter = 0

    # counting occurences of each column value
    occurrences = {}
    for i, o in x.loc[::step].iterrows():
        counter += 1
        if str(o[col]) not in occurrences.keys():
            occurrences[str(o[col])] = [0., 0.]
        if y.loc[i, ["ReturnQuantityBin"]][0] == 0.0:
            occurrences[str(o[col])][0] += 1.
        else:
            occurrences[str(o[col])][1] += 1.

    # computing the returns frequency, stored in `recap`
    recap, values = [], []
    for val, (zeros, ones) in occurrences.items():
        values.append(((ones / (zeros + ones))))
        recap.append((val, values[-1]))
    recap.sort(key=lambda row: row[0])

    # computing variance and relative variance
    var = np.var(values)
    rel = var / len(values)

    return recap, var, rel, counter


def column_stats(x, y, col, step, verbose):
    """Computes the statistics for one column"""

    print "\n----- " + col + " -----"
    recap, var, rel, counter = returns_frequency(x, y, col, step)

    if verbose:
        for (val, freq) in recap:
            print val, "\t", freq, "returns"

    print "variance:", round(var, 5), "\tvalues count:", len(recap)
    return recap, var, rel, counter
```

```
def compute_statistics(x, y, blacklist=[], whitelist=[], step=100, verbose=False):
    ignored = []
    labels_g, labels_n = [], []
    scattering_g, scattering_n = [], []      # variances
    r_scattering_g, r_scattering_n = [], []  # variance divided by the number of differents values

    counter_cols = 0
    counter_rows = 0

    for col in x.columns:
        if col not in blacklist and x[col].dtype in ["object", "bool"]:
            counter_cols += 1
            labels_g.append(col)
            recap, var, rel, counter_rows = column_stats(x, y, col, step, verbose)
            scattering_g.append(var)
            r_scattering_g.append(rel)
        elif col not in blacklist and x[col].dtype in ["float64", "int64"]:
            counter_cols += 1
            labels_n.append(col)
            recap, var, rel, counter_rows = column_stats(x, y, col, step, verbose)
            scattering_n.append(var)
            r_scattering_n.append(rel)
            if col in whitelist:
                plot_dots([float(v[0]) for v in recap], [v[1] for v in recap], col)
        else:
            ignored.append(col)

    print "\n\nanalyzed", counter_cols, "columns and", counter_rows, "rows"
    print "\nignored columns:", ignored
    plot_barchart(labels_g, scattering_g, r_scattering_g, "general columns")
    plot_barchart(labels_g, scattering_g, r_scattering_g, "numerical columns")


def plot_barchart(labels, values_l, values_r, title):
    fig, ax = plt.subplots(figsize=(16, 10))
    ind = np.arange(len(values_l))
    width = .35
    ax.bar(ind - width/2, values_l, width)
    ax.bar(ind + width/2, values_r, width)
    ax.set_xticks(ind)
    ax.set_xticklabels(labels)
    plt.xticks(rotation=70)
    plt.title(title)
    plt.show()


def plot_dots(xs, ys, xlabel):
    plt.figure(figsize=(8, 5))
    plt.xlabel(xlabel)
    plt.ylabel("returns frequency")
    plt.plot(xs, ys, 'o')
    plt.show()
```

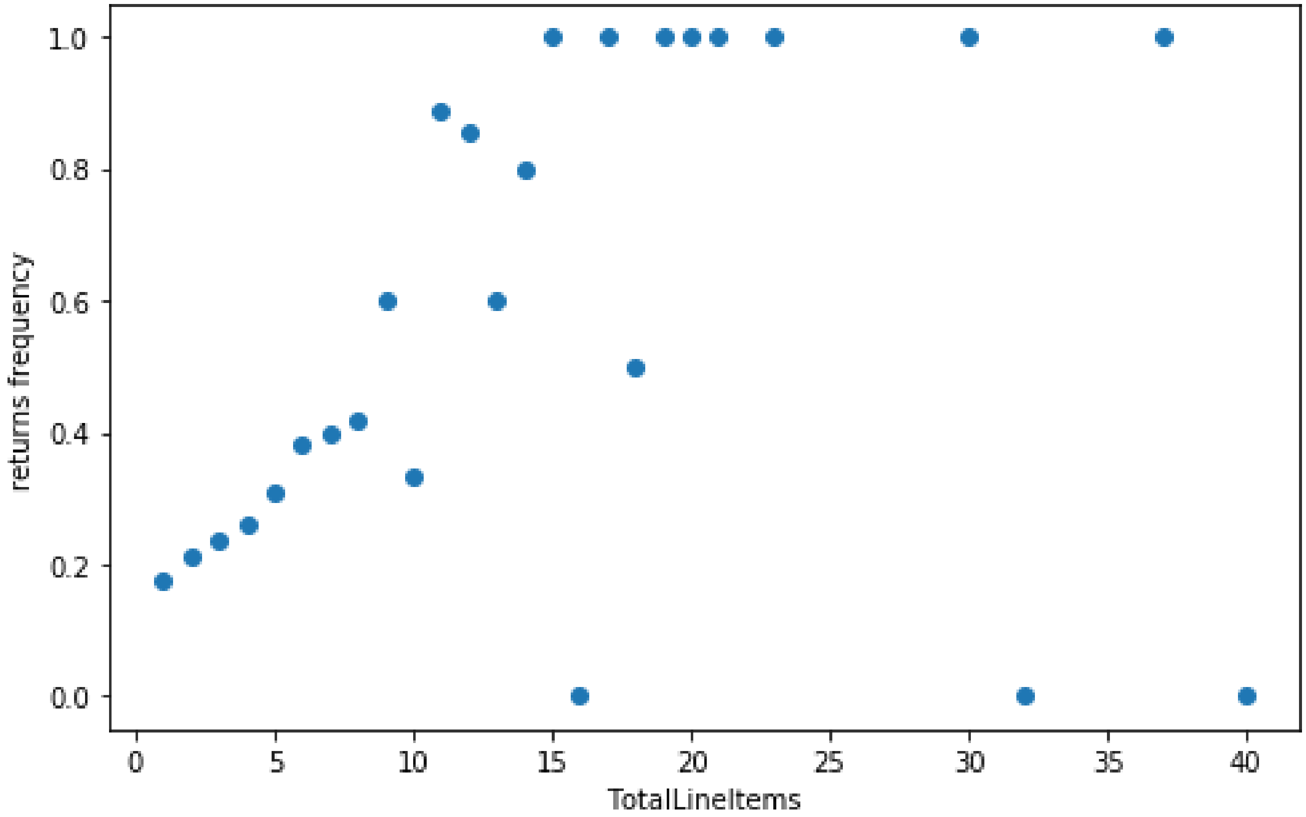
```
In [22]: df_stats = build_df(x_train)
```

```
dataframe shape: (1067290, 40)
dataframe built        (76.23s)
```

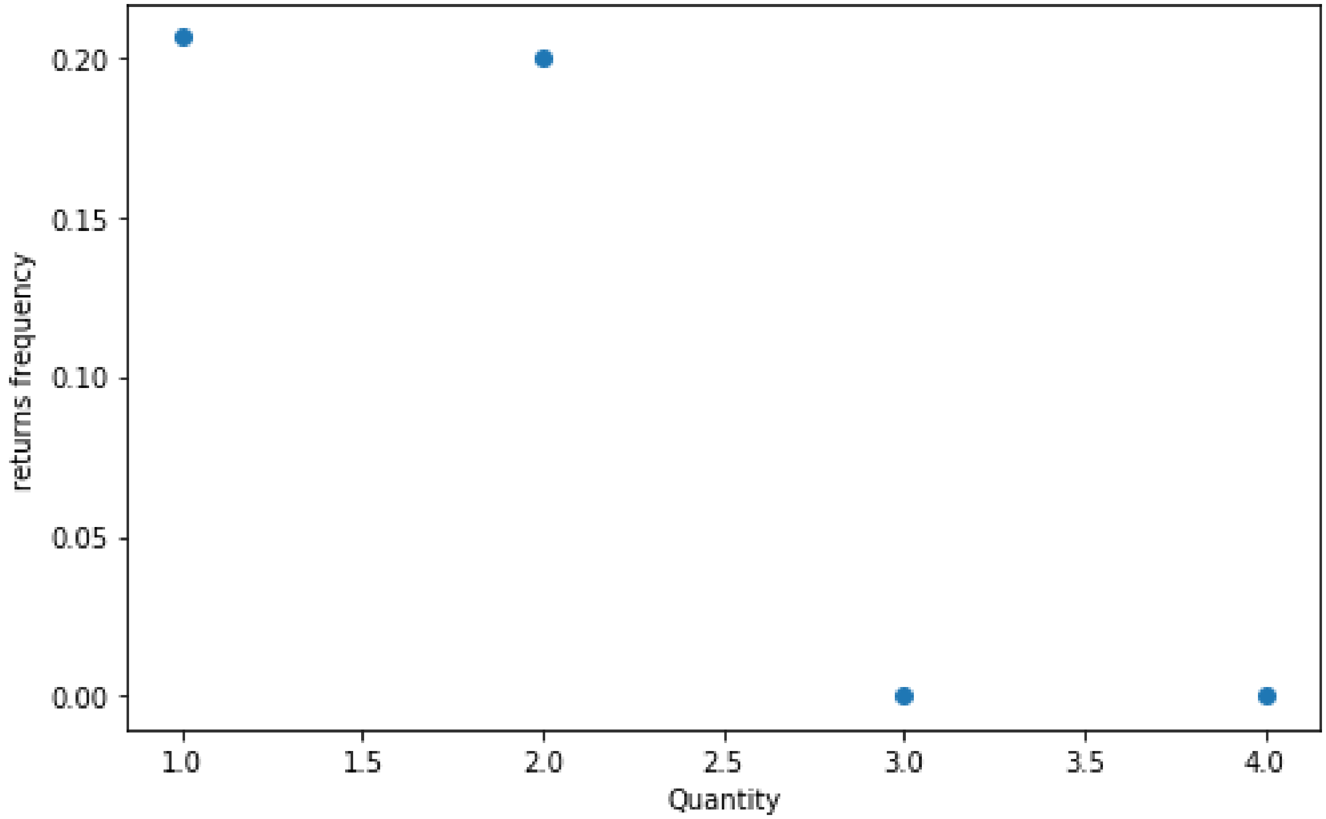
```
In [6]: compute_statistics(df_stats, y_train, blacklist=[], whitelist=df_stats.columns, step=100)
```

```

----- OrderTypelabel -----
variance: 0.00017         values count: 2

----- SeasonLabel_pr -----
variance: 7e-05         values count: 2

----- PayementModeLabel -----
variance: 0.00472         values count: 10

----- CustomerTypeLabel -----
variance: 0.00022         values count: 2

----- IsoCode -----
variance: 0.04536         values count: 20
```

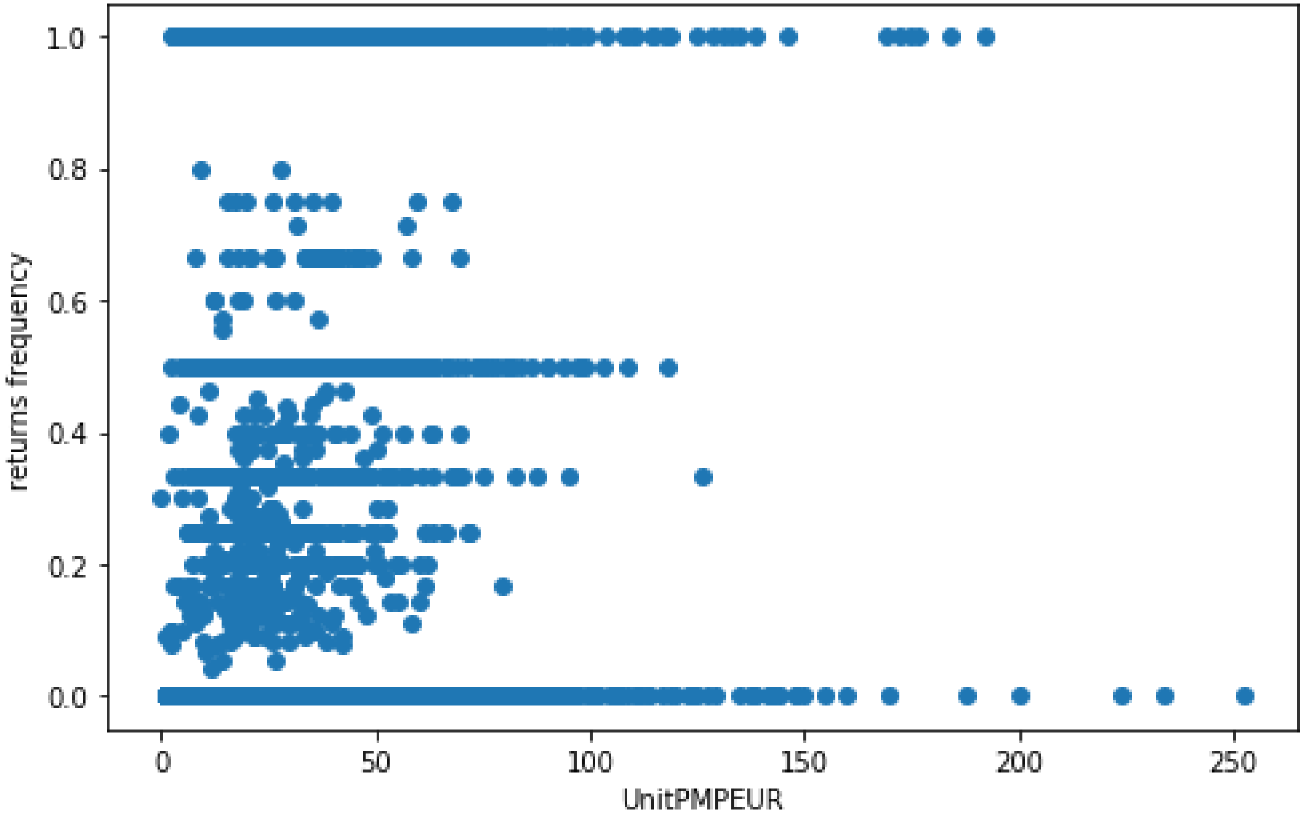
```
----- DeviceTypeLabel -----
variance: 0.00124         values count: 4

----- PricingTypeLabel -----
variance: 0.00072         values count: 5

----- TotalLineItems -----
variance: 0.13073         values count: 26
```



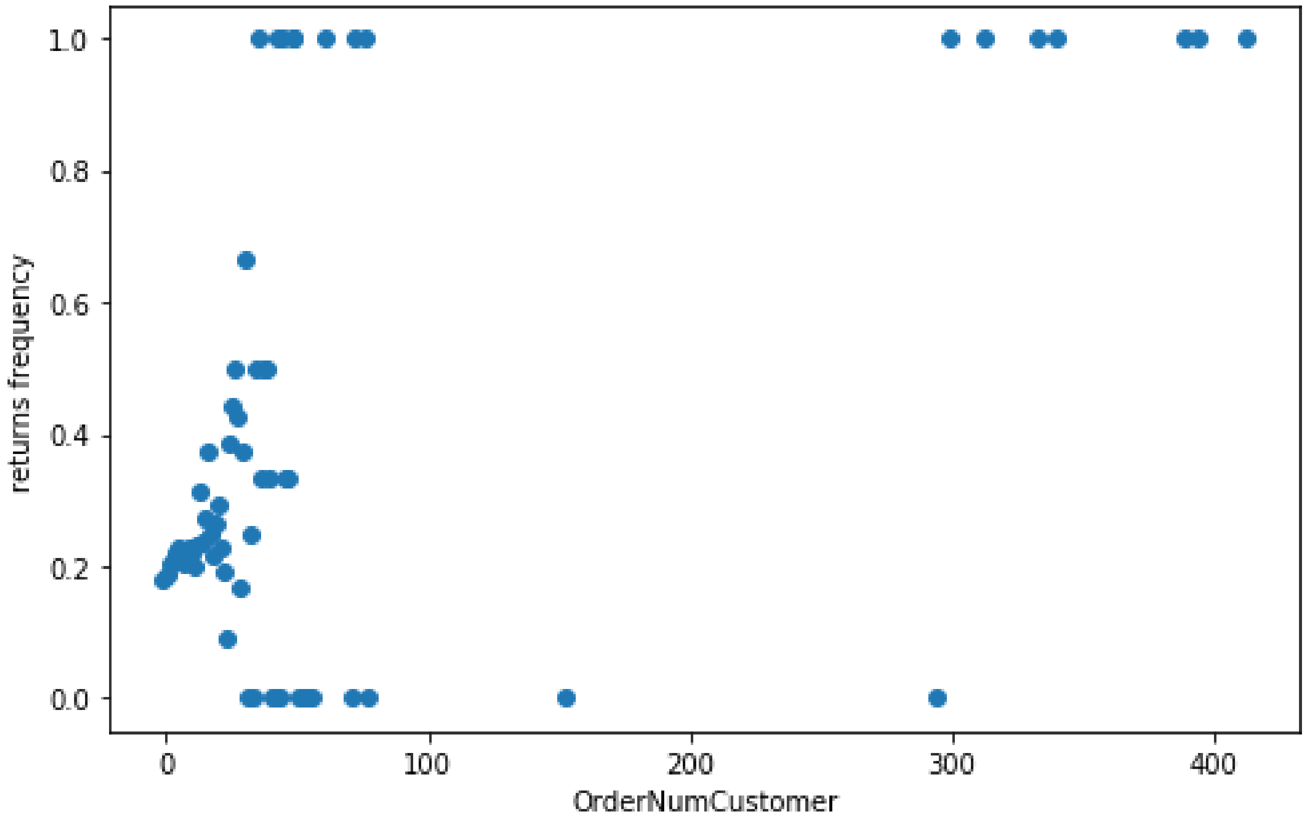
```
----- Quantity -----
variance: 0.01033        values count: 4
```



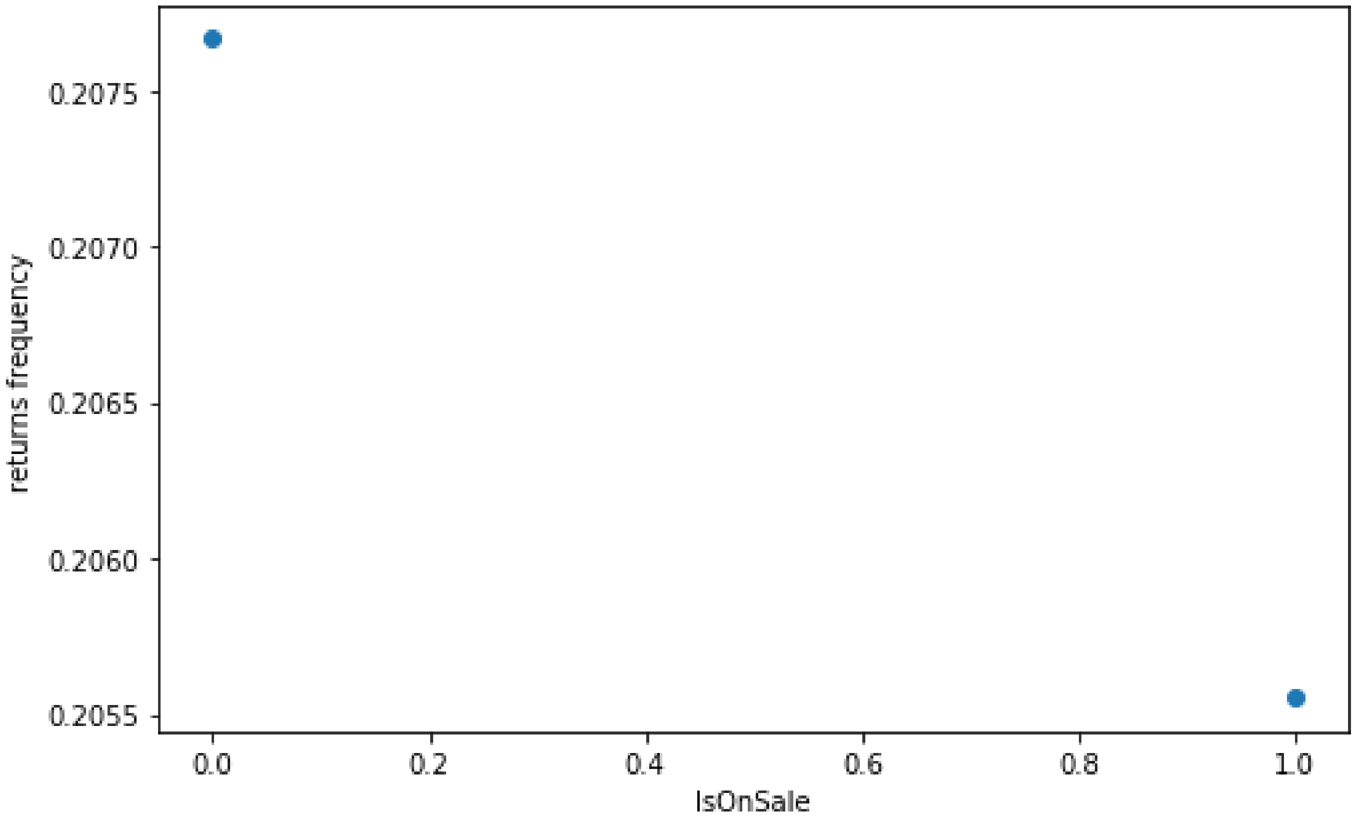
```
----- UnitPMPEUR -----
variance: 0.13264        values count: 5066
```



```
----- OrderNumCustomer -----
variance: 0.12625        values count: 67
```

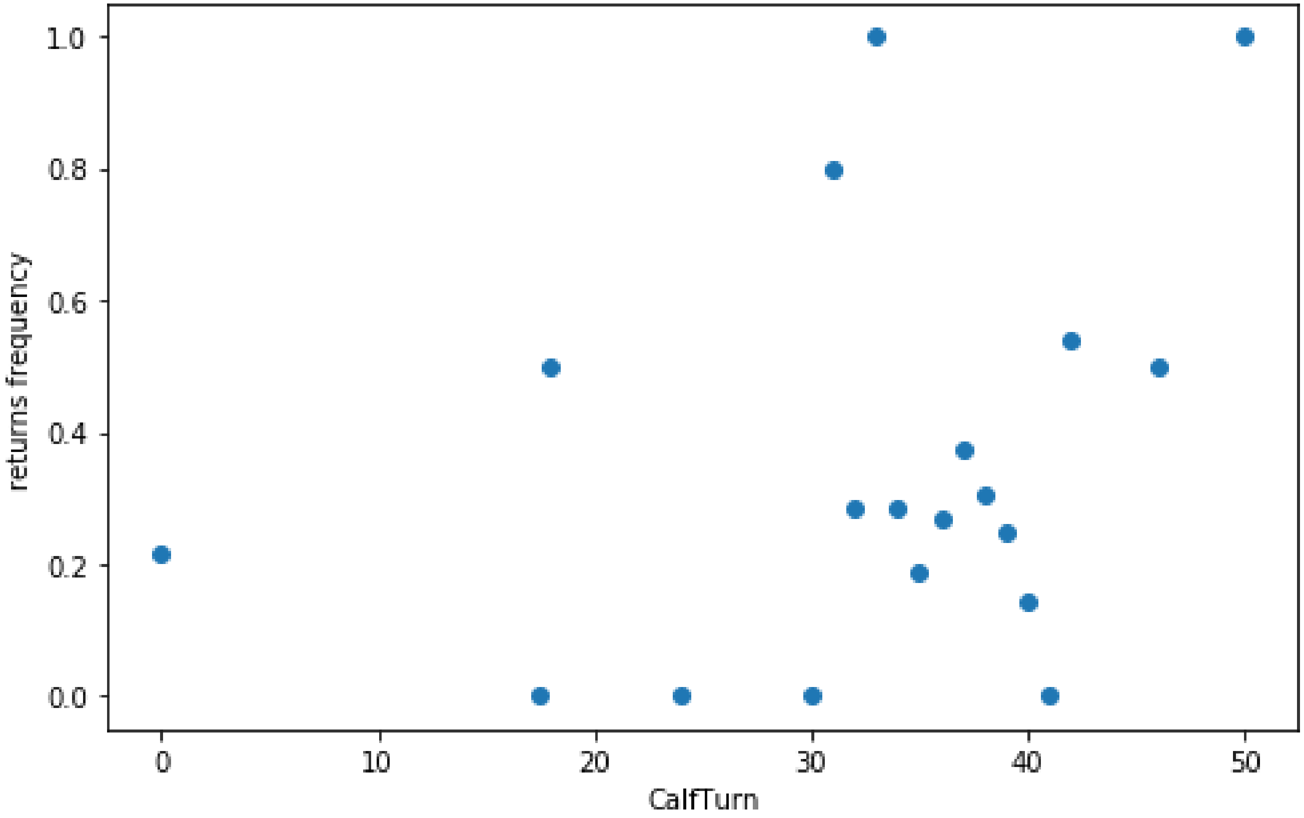


```
----- IsOnSale -----
variance: 0.00176        values count: 3
```

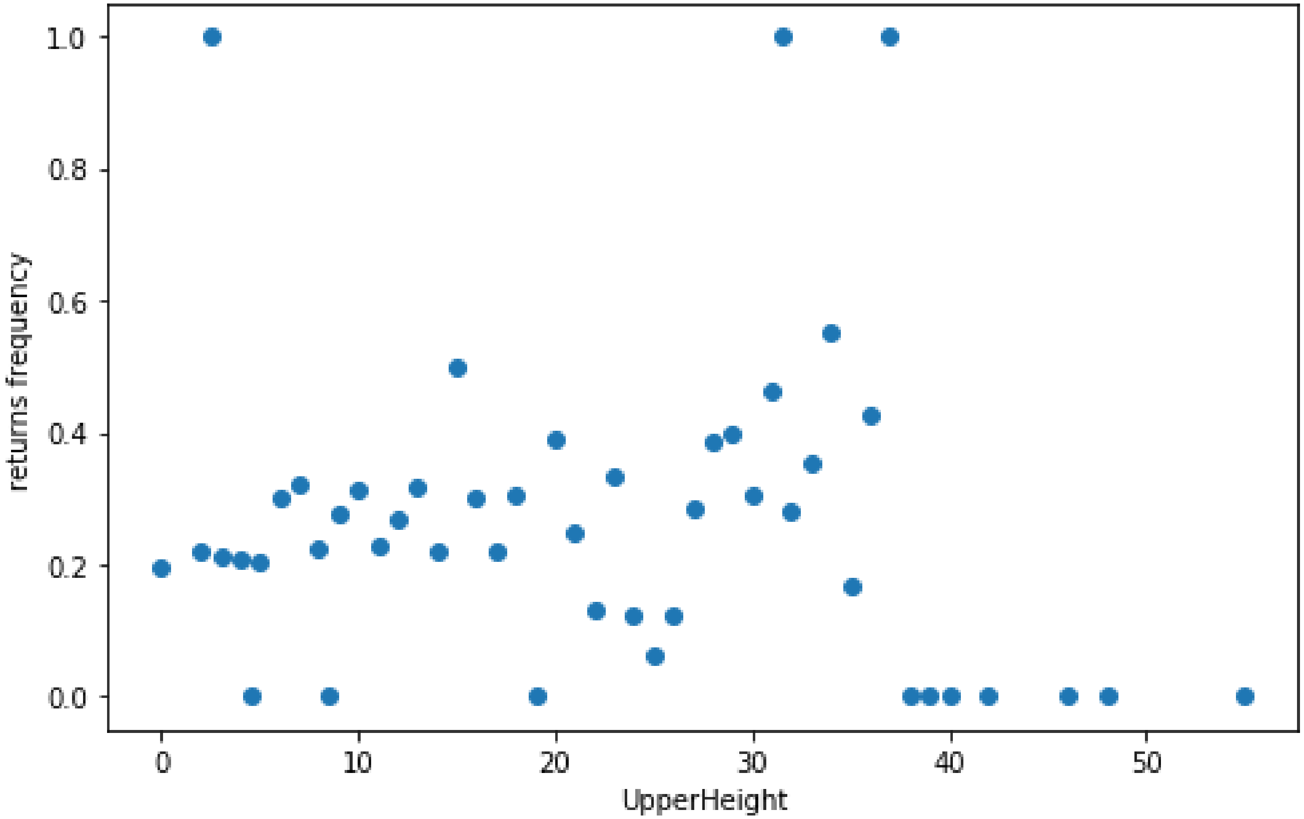


```
----- GenderLabel -----
variance: 0.00338         values count: 6

----- MarketTargetLabel -----
variance: 0.00735         values count: 14

----- SeasonLabel -----
variance: 0.00025         values count: 3
```

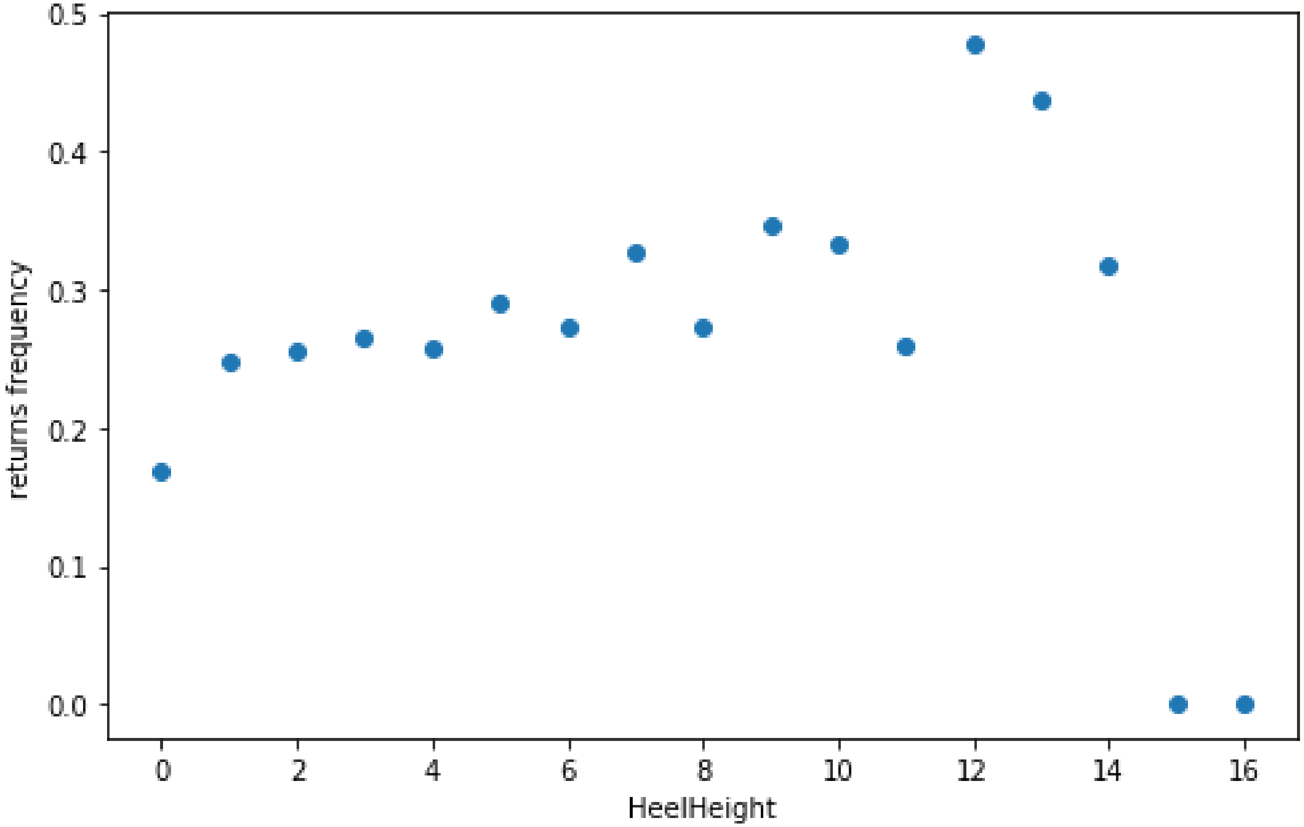
```
----- SeasonalityLabel -----
variance: 0.00057         values count: 4

----- UniverseLabel -----
variance: 0.00268         values count: 9

----- TypeBrand -----
variance: 0.00581         values count: 4

----- CalfTurn -----
variance: 0.08768         values count: 20
```



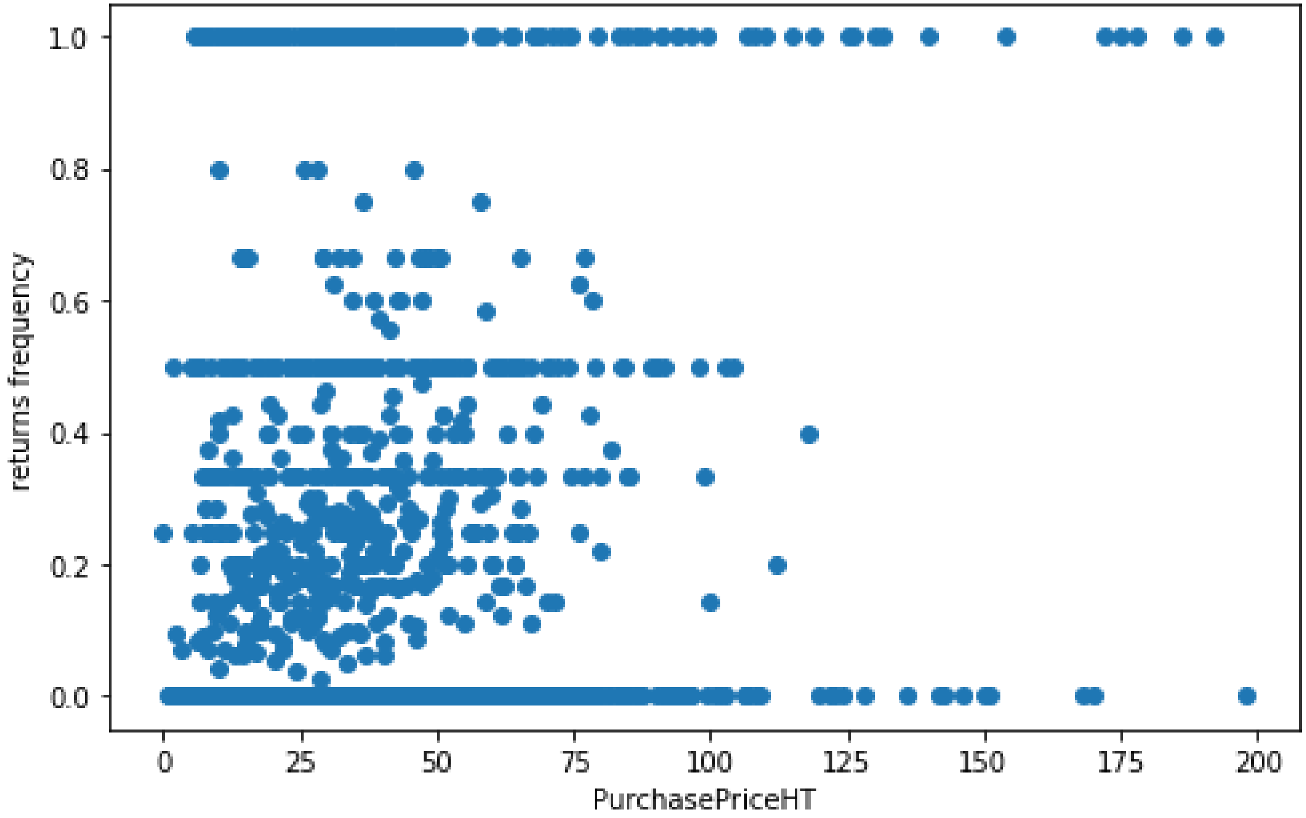
```
----- UpperHeight -----
variance: 0.05583        values count: 49
```



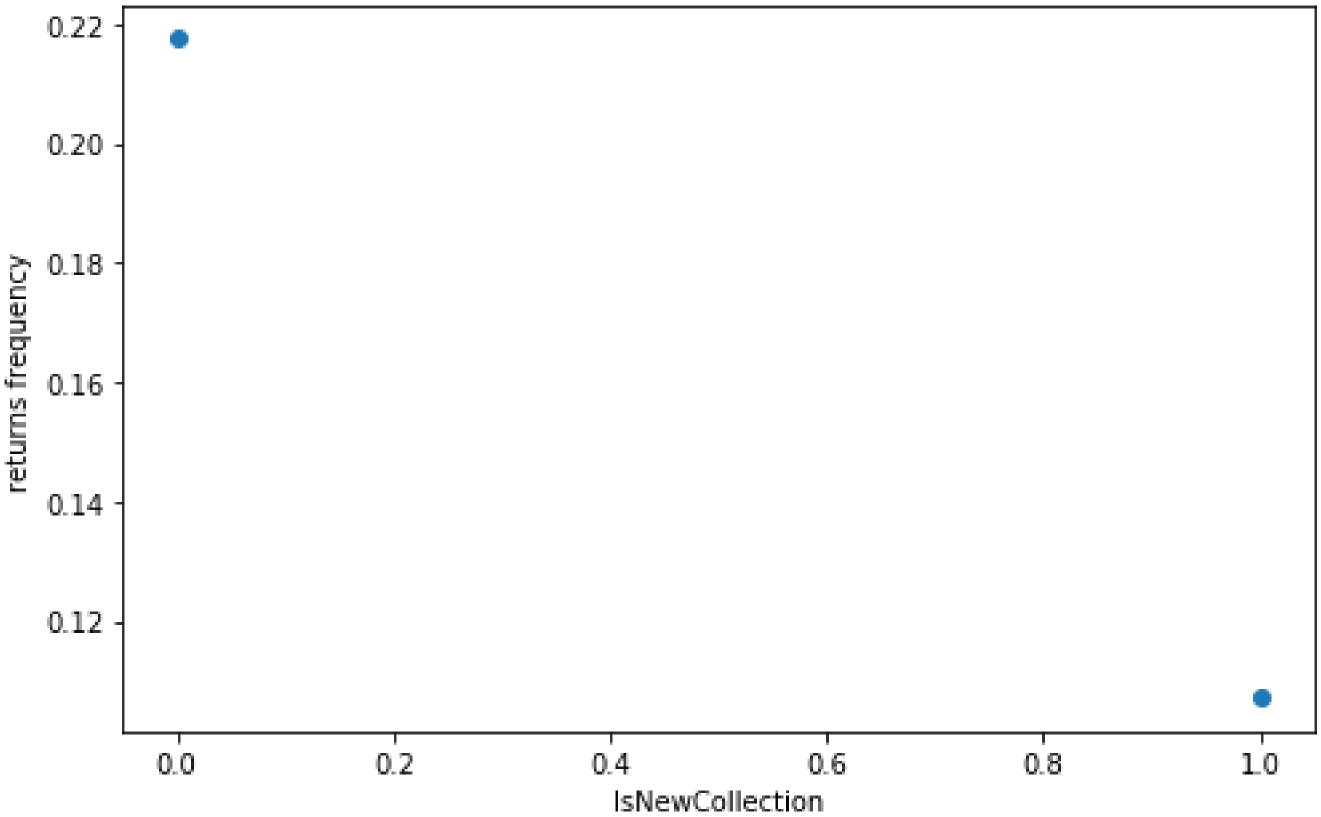
```
----- HeelHeight -----
variance: 0.01392        values count: 18
```



```
----- PurchasePriceHT -----
variance: 0.11134         values count: 1446
```

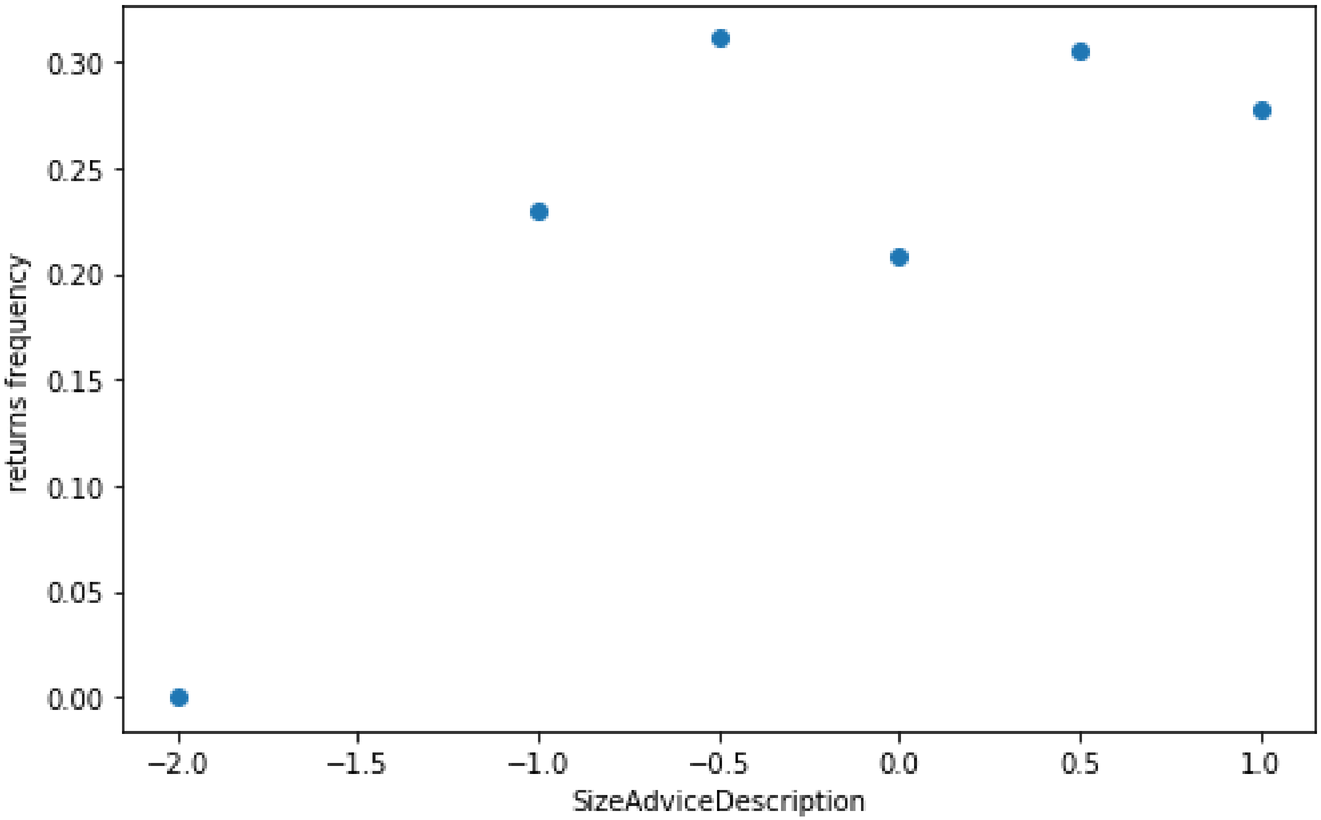


```
----- IsNewCollection -----
variance: 0.00217         values count: 3
```



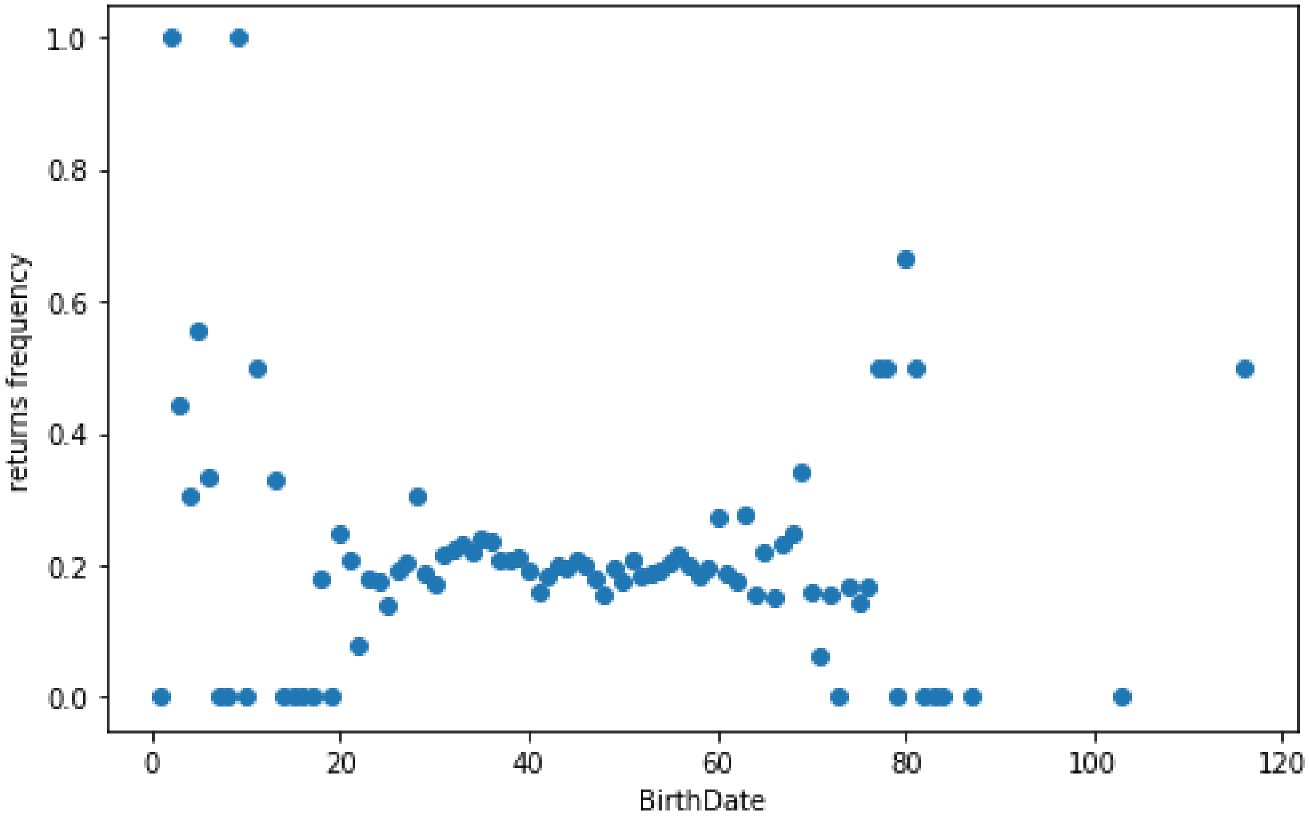
```
----- UpperMaterialLabel -----
variance: 0.00883        values count: 9

----- LiningMaterialLabel -----
variance: 0.07434        values count: 11

----- OutSoleMaterialLabel -----
variance: 0.01128        values count: 8
```

```
----- RemovableSole -----
variance: 0.00039         values count: 3

----- SizeAdviceDescription -----
variance: 0.0099         values count: 7
```



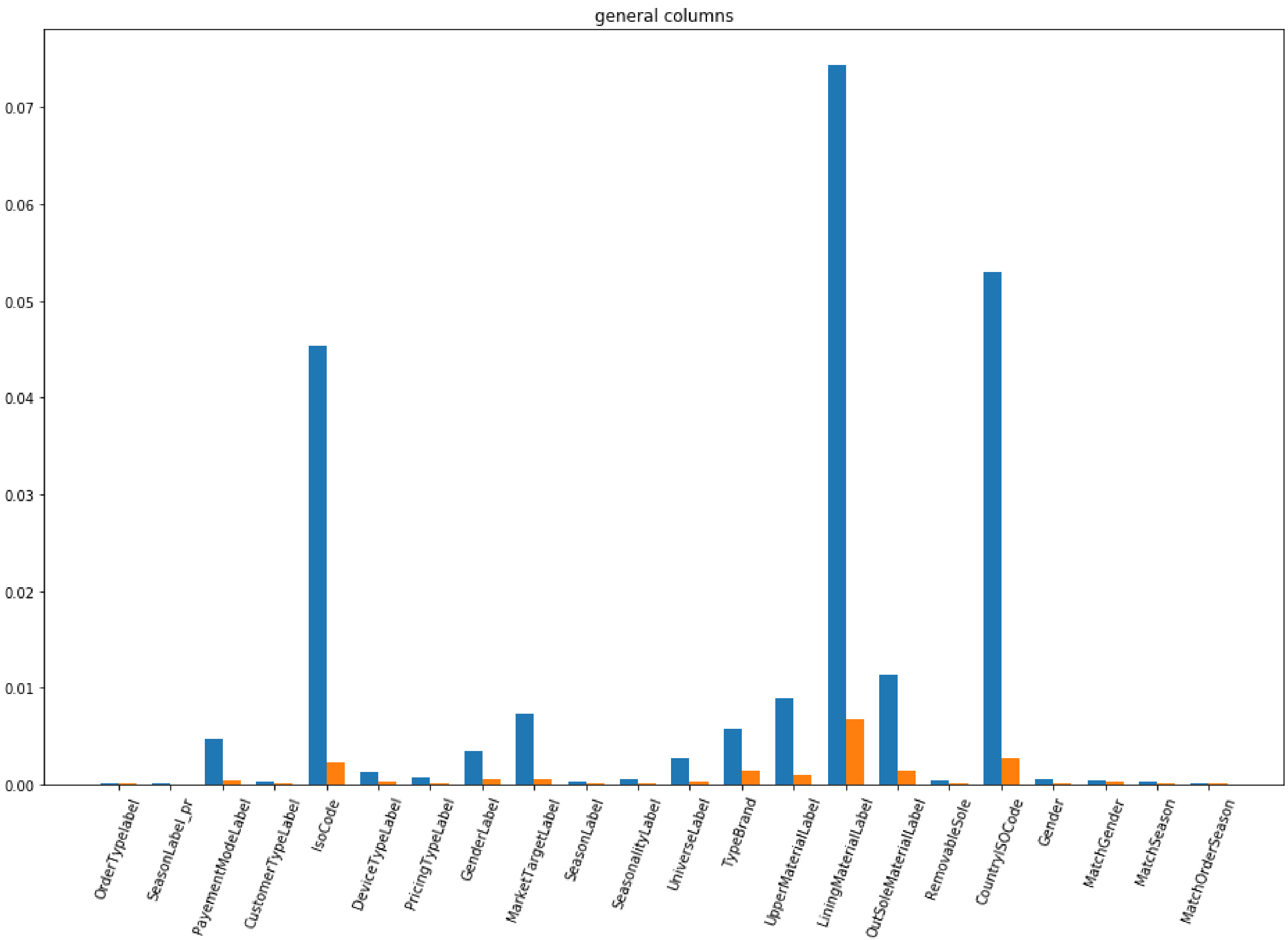
```
----- CountryISOCode -----
```

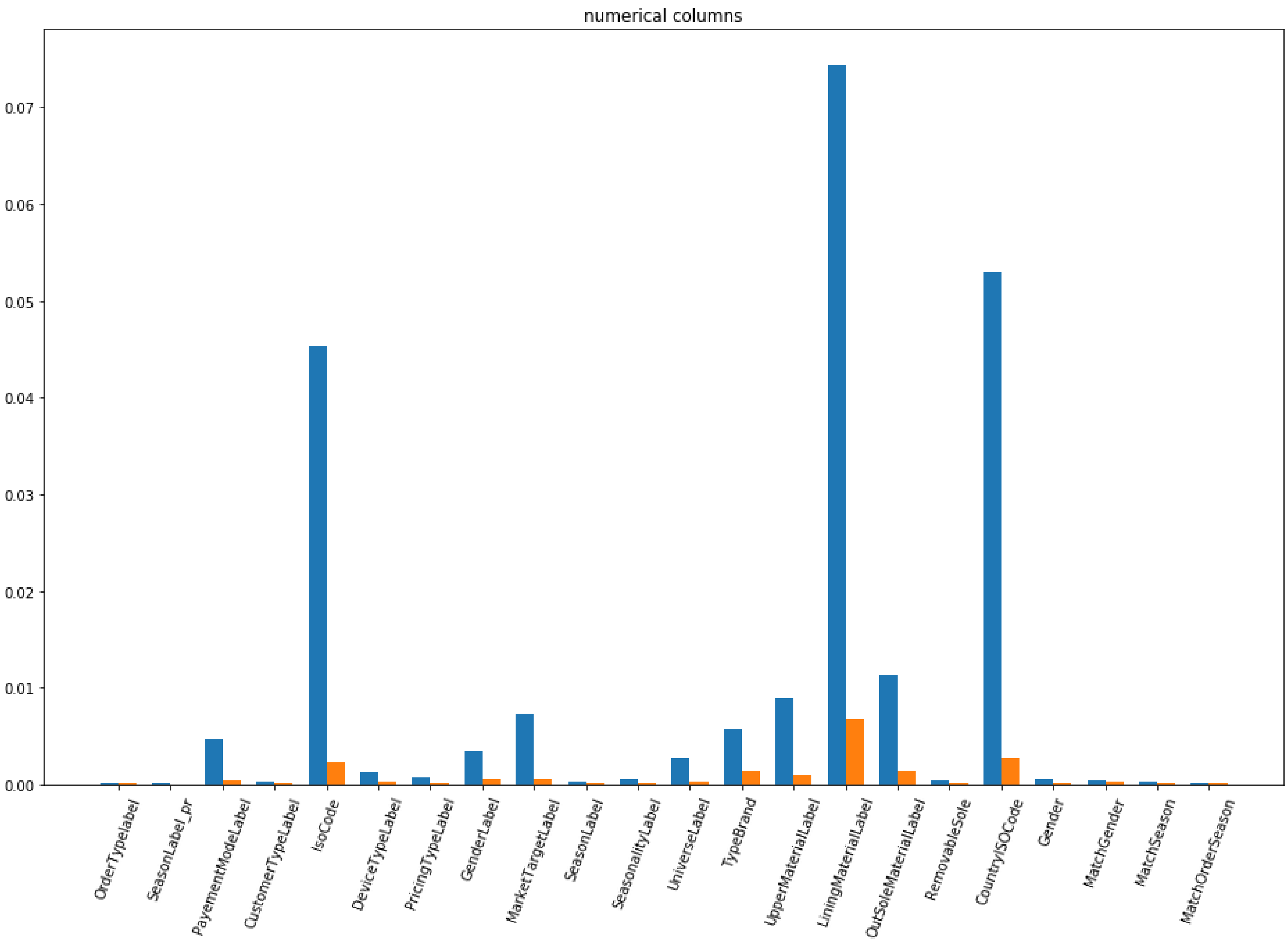
```
variance: 0.05302         values count: 19

----- BirthDate -----
variance: 0.03355         values count: 87
```



```
----- Gender -----
variance: 0.00061         values count: 3
```

```
----- MatchGender -----
variance: 0.00048         values count: 2

----- MatchSeason -----
variance: 0.00032         values count: 2

----- MatchOrderSeason -----
variance: 0.00017         values count: 2


analyzed 34 columns and 10673 rows

ignored columns: []
```


general columns
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
OrderTypelabel
SeasonLabel_pr
PayementModeLabel
CustomerTypeLabel
IsoCode
DeviceTypeLabel
PricingTypeLabel
GenderLabel
MarketTargetLabel
SeasonLabel
SeasonalityLabel
UniverseLabel
TypeBrand
UpperMaterialLabel
LiningMaterialLabel
OutSoleMaterialLabel
RemovableSole
CountryISOCode
Gender
MatchGender
MatchSeason
MatchOrderSeason


numerical columns
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
OrderTypelabel
SeasonLabel_pr
PayementModeLabel
CustomerTypeLabel
IsoCode
DeviceTypeLabel
PricingTypeLabel
GenderLabel
MarketTargetLabel
SeasonLabel
SeasonalityLabel
UniverseLabel
TypeBrand
UpperMaterialLabel
LiningMaterialLabel
OutSoleMaterialLabel
RemovableSole
CountryISOCode
Gender
MatchGender
MatchSeason
MatchOrderSeason

```
In [26]: column_stats(df_train, y_train, "Gender", 100, True)
```

```
----- Gender -----
Femme         0.212231857599 returns
Homme         0.160953800298 returns
nan         0.214285714286 returns
variance: 0.00061         values count: 3
```

```
Out[26]: ([('Femme', 0.21223185759926974),
           ('Homme', 0.16095380029806258),
           ('nan', 0.21428571428571427)],
          0.00060866117082895 73,
          0.00020288705694298 58,
          10673)
```

1.0.5 Classification

```
In [28]: def mask_bin(m):
             columns2bin = [col for col in m.columns if m[col].dtype == 'object']
             other_cols = m.drop(columns2bin, axis=1)
             new_cols = pd.get_dummies(m.loc[:, columns2bin])
             res = pd.concat([other_cols, new_cols], axis=1)
             res = res.fillna(0)
             print "new shape:", res.shape
             return res

         def mask_int(m):
             columns2int = [col for col in m.columns if m[col].dtype == 'object']
             res = m.apply(LabelEncoder().fit_transform)
             res = res.fillna(0)
             print "new shape:", res.shape
             return res

         def compute(name, clf, x1, x2, slc=100000):
             print "\n-----", name, "-----"
             clf.fit(x1.iloc[:slc], y_train.ReturnQuantityBin[:slc])

             predict_train = clf.predict_proba(x1.iloc[:slc])
```

```
    score_train = roc_auc_score(y_train.ReturnQuantityBin[:slc], predict_train[:, 1])
    print "train score:", score_train

    predict_test = clf.predict_proba(x1.iloc[slc:2 * slc])
    score_test = roc_auc_score(y_train.ReturnQuantityBin[slc:2 * slc], predict_test[:, 1])
    print "test score:", score_test
    return score_train, score_test

def compute_all(x1, x2, slc=100000):
    """Tries different classifiers and returns the best one (best test score)"""
    t = time.time()
    best_index, best_score = None, None

    print "train shape:\t", x1.shape, "\t", y_train.shape
    print "test shape:\t", x2.shape, "\t", y_test.shape

    classifiers = [("random forest", RandomForestClassifier()),
                   ("decision tree", DecisionTreeClassifier()),
                   ("logistic regression", LogisticRegression())]

    for i, (name, clf) in enumerate(classifiers):
        score_train, score_test = compute(name, clf, x1, x2, slc)
        if best_score is None or score_test > best_score:
            best_index, best_score = i, score_test

    log("\nbest classifier: " + classifiers[best_index][0], t)
    return classifiers[best_index][1]

def output(clf, x1, x2):
    t = time.time()
    y_tosubmit = clf.predict_proba(x2.loc[:, x1.columns].fillna(0))

    timestamp = '{0:%Y_%m_%d_%H_%M_%S}'.format(datetime.datetime.now())
    filename = "ypred_{0}.txt".format(timestamp)
    np.savetxt(filename, y_tosubmit[:,1], fmt='%f')

    f = open("predictions.txt", 'a')
    f.write(timestamp + '\n' + repr(clf).replace('\n          ', '') + '\n\n')
    f.close()
```

```
             print "shape:", y_tosubmit.shape
             log("generated output at " + filename, t)
```

1.0.6 Computation test loop

```
In [6]: df_train = build_df(x_train)
```

```
dataframe shape: (1067290, 40)
dataframe built        (6.42s)
```

```
In [27]: df_test = build_df(x_test)
```

```
dataframe shape: (800468, 40)
dataframe built        (11.46s)
```

```
In [14]: t = time.time()
         x1 = mask_bin(df_train)
         x2 = mask_bin(df_test)
         log("applied mask", t)
```

```
new shape: (1067290, 40)
new shape: (800468, 40)
applied mask        (24.57s)
```

```
In [9]: clf = compute_all(x1, x2)
```

```
train shape:        (1067290, 158)         (1067290, 4)
test shape:        (800468, 169)         (800468, 4)

----- random forest -----
train score: 0.9949450061129895
test score: 0.5720997532776027

----- decision tree -----
train score: 0.9993978968884667
test score: 0.5333491459838888
```

```
----- logistic regression -----
train score: 0.63903864972542
test score: 0.6329470043824317

best classifier: logistic regression        (14.0s)
```

```
In [13]: output(clf, x1, x2)
```

```
shape: (800468, 2)
generated output at ypred_2018_04_05_15_54_25.txt        (5.94s)
```

1.0.7 Dataset permutation

```
In [ ]: def shuffle(x, y, clf_fun, steps=10, slc=50000, plot=True):
            scores_train, scores_test = [], []
            best_clf, best_score = None, None

            z = x.copy(deep=True)
            z["ReturnQuantityBin"] = y.ReturnQuantityBin

            for k in range(steps):
                u = z.sample(frac=1)
                v = u.loc[:, ["ReturnQuantityBin"]]
                u = u.drop(["ReturnQuantityBin"], axis=1)

                clf = clf_fun()
                clf.fit(u.iloc[:slc], v.ReturnQuantityBin[:slc])

                predict_train = clf.predict_proba(u.iloc[:slc])
                score_train = roc_auc_score(v.ReturnQuantityBin[:slc], predict_train[:, 1])

                predict_test = clf.predict_proba(u.iloc[slc:2 * slc])
                score_test = roc_auc_score(v.ReturnQuantityBin[slc:2 * slc], predict_test[:, 1])

                if best_clf is None or score_test > best_score:
                    best_clf, best_score = clf, score_test

                if plot:
```

```
            print "test", k, "\ttrain:", score_train, "\ttest:", score_test

        scores_train.append(score_train)
        scores_test.append(score_test)

    if plot:
        plt.figure(figsize=(16, 10))
        plt.xlabel("train score")
        plt.ylabel("test score")
        plt.plot(scores_train, scores_test, '+')
        plt.show()

    return scores_train, scores_test, best_clf, best_score

def try_slice(x, y, clf_fun, plot=True, steps=10):

    slices, scores, classifiers = [], [], {}
    best_slice, best_score = None, None

    for slc in range(10000, 200001, 10000):
        sc_train, sc_test, clf, score = shuffle(x, y, clf_fun, slc=slc, steps=steps, plot=False)
        slices.append(slc)
        scores.append(score)
        if best_slice is None or best_scores < score:
            best_slice, best_score = slc, score
        classifiers[slc] = clf
        print "slice", slc, "\t", score

    if plot:
        plt.figure(figsize=(16, 10))
        plt.xlabel("slice")
        plt.ylabel("score")
        plt.plot(slices, scores, '-o')
        plt.show()

    return classifiers[best_slice]
```

```
In [ ]: sc_train, sc_test, best_clf, best_score = shuffle(x1, y_train, slc=130000, steps=10, plot=False)
        print best_score
```

```
In [ ]: output(best_clf, x1, x2)
```

1.0.8 Principal component analysis (PCA)

Best scores reached with n_components at 96: - train score: 0.6388630967451936 - test score: 0.6331956289779485

Above, scores are deacreasing.

Note: the result from .transform() is a Numpy array. Therefore the slicing is different.

```
In [29]: def try_pca(data, n_components):
             print "\n----- PCA", n_components, "-----"

             pca = PCA(n_components=n_components)
             pca.fit(data)
             x = pca.transform(data)
             clf = LogisticRegression()
             slc = 100000
             clf.fit(x[:slc, :], y_train.ReturnQuantityBin[:slc])

             predict_train = clf.predict_proba(x[:slc, :])
             score_train = roc_auc_score(y_train.ReturnQuantityBin[:slc], predict_train[:, 1])
             print "train score:", score_train

             predict_test = clf.predict_proba(x[slc:2 * slc, :])
             score_test = roc_auc_score(y_train.ReturnQuantityBin[slc:2 * slc], predict_test[:, 1])
             print "test score:", score_test

             return score_train, score_test, pca

         for n in range(1, 100, 5):
             try_pca(x1, n)
```

```
----- PCA 1 -----
train score: 0.5702366075653182
test score: 0.5647601895068738

----- PCA 6 -----
train score: 0.5999356153282647
test score: 0.5983282609508912
```

```
----- PCA 11 -----
train score: 0.6218118708758879
test score: 0.6138734988058648

----- PCA 16 -----
train score: 0.6277049263505473
test score: 0.6225947759354865

----- PCA 21 -----
train score: 0.6284750476572429
test score: 0.6244283257910828

----- PCA 26 -----
train score: 0.6293423130492306
test score: 0.6249607267334103

----- PCA 31 -----
train score: 0.6300006665601142
test score: 0.6252857562462173

----- PCA 36 -----
train score: 0.6341085795216018
test score: 0.6283580092583988

----- PCA 41 -----
train score: 0.6370415100553493
test score: 0.6315438780860905

----- PCA 46 -----
train score: 0.6369463535588813
test score: 0.6318376492846729

----- PCA 51 -----
train score: 0.6373923821818226
test score: 0.632345993128418

----- PCA 56 -----
train score: 0.6376517894809209
test score: 0.6320991622328517
```

```
----- PCA 61 -----
train score: 0.6377169758518486
test score: 0.6325997920070896

----- PCA 66 -----
train score: 0.638039089427804
test score: 0.63284863566079

----- PCA 71 -----
train score: 0.6384357857447727
test score: 0.6332645079580541

----- PCA 76 -----
train score: 0.6387240315583405
test score: 0.6331850942121828

----- PCA 81 -----
train score: 0.6387008959275011
test score: 0.6330236438208361

----- PCA 86 -----
train score: 0.6388413465503924
test score: 0.6331544222007438

----- PCA 91 -----
train score: 0.6388561386698401
test score: 0.6331928959049634

----- PCA 96 -----
train score: 0.6388630967451936
test score: 0.6331956289779485
```

1.0.9 Cross validation and random forests

This is an *attempt*, scores did not met expectation.

```
In [61]: def cross_val(name, clf, slc=100000):
             print "\n-----", name, "-----"
```

```
                predict_train = cross_val_predict(clf, x1.iloc[:slc], y_train.ReturnQuantityBin[:slc], cv=10, method='predict_proba')
                score_train = roc_auc_score(y_train.ReturnQuantityBin[:slc], predict_train[:, 1])
                print "train score:", score_train
                return score_train

            cross_val("LogisticRegression", LogisticRegression())
```

```
----- LogisticRegression -----
train score: 0.6332445684201241
```

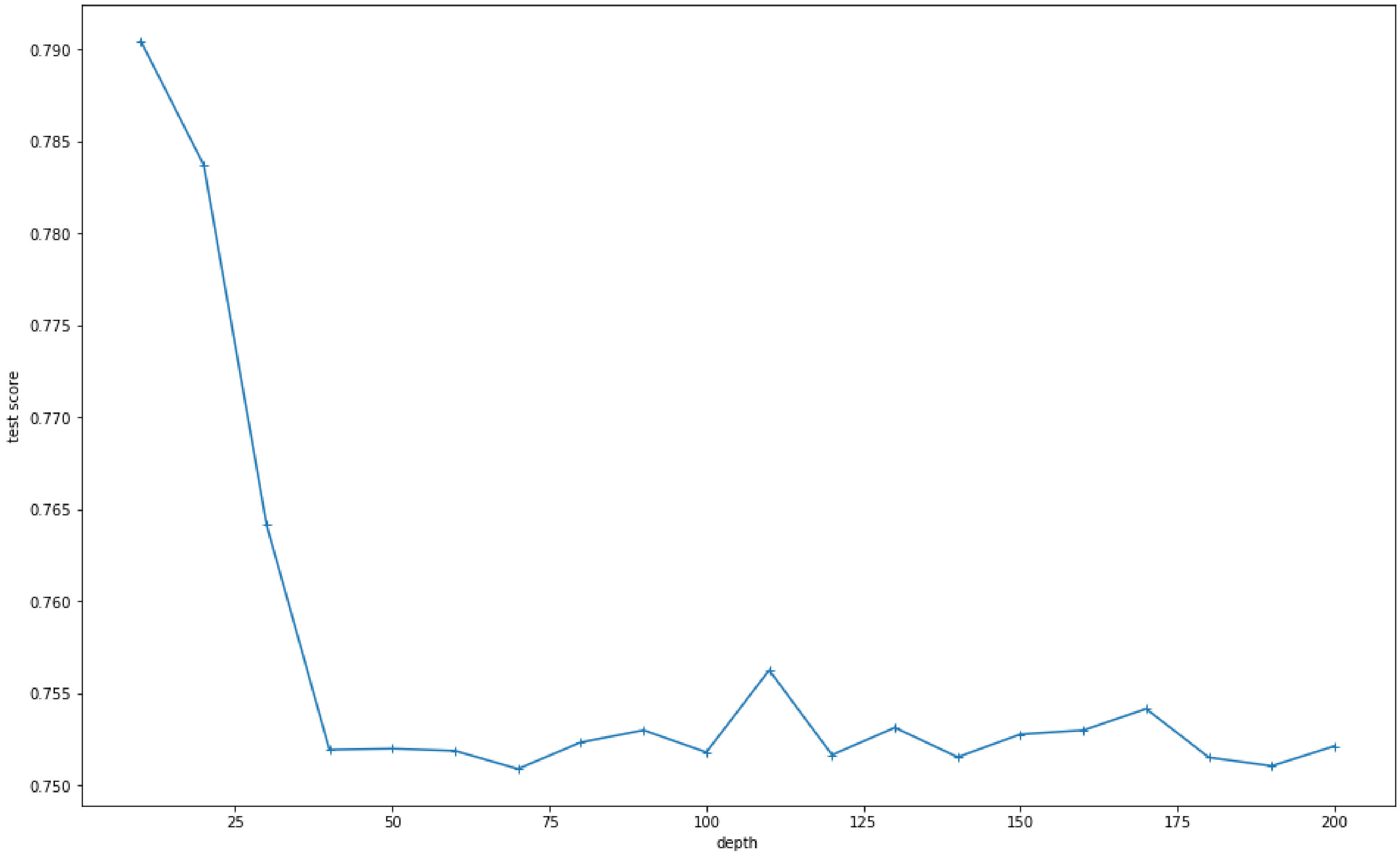
```
Out[61]: 0.6332445684201241
```

```
In [20]: depths, scores = [], []

         for max_depth in range(10, 201, 10):
             clf = RandomForestClassifier(max_depth=max_depth)
             score = cross_val_score(clf, x1.loc[:50000], y_train.loc[:50000, "ReturnQuantityBin"], cv=5).mean()
             depths.append(max_depth)
             scores.append(score)
             print 'max_depth:', max_depth, '\tscore:', score
```

```
max_depth: 10         score: 0.7904241907464419
max_depth: 20         score: 0.7837043827060438
max_depth: 30         score: 0.7641646465896466
max_depth: 40         score: 0.7519450905184509
max_depth: 50         score: 0.7520049165158491
max_depth: 60         score: 0.7518849505168496
max_depth: 70         score: 0.7509051305164514
max_depth: 80         score: 0.7523450245226503
max_depth: 90         score: 0.7530050525222506
max_depth: 100         score: 0.7518051165306512
max_depth: 110         score: 0.7562650105484501
max_depth: 120         score: 0.7516650545148506
max_depth: 130         score: 0.7531449625248496
max_depth: 140         score: 0.7515450645214508
max_depth: 150         score: 0.7527851165222512
max_depth: 160         score: 0.7530050285102503
max_depth: 170         score: 0.7541649545348494
```

```
max_depth: 180         score: 0.7515251585076517
max_depth: 190         score: 0.7510651385024513
max_depth: 200         score: 0.7521451345096513
```

```
In [21]: plt.figure(figsize=(16, 10))
         plt.xlabel("depth")
         plt.ylabel("test score")
         plt.plot(depths, scores, '-+')
         plt.show()
```



1.0.10 Feature importance

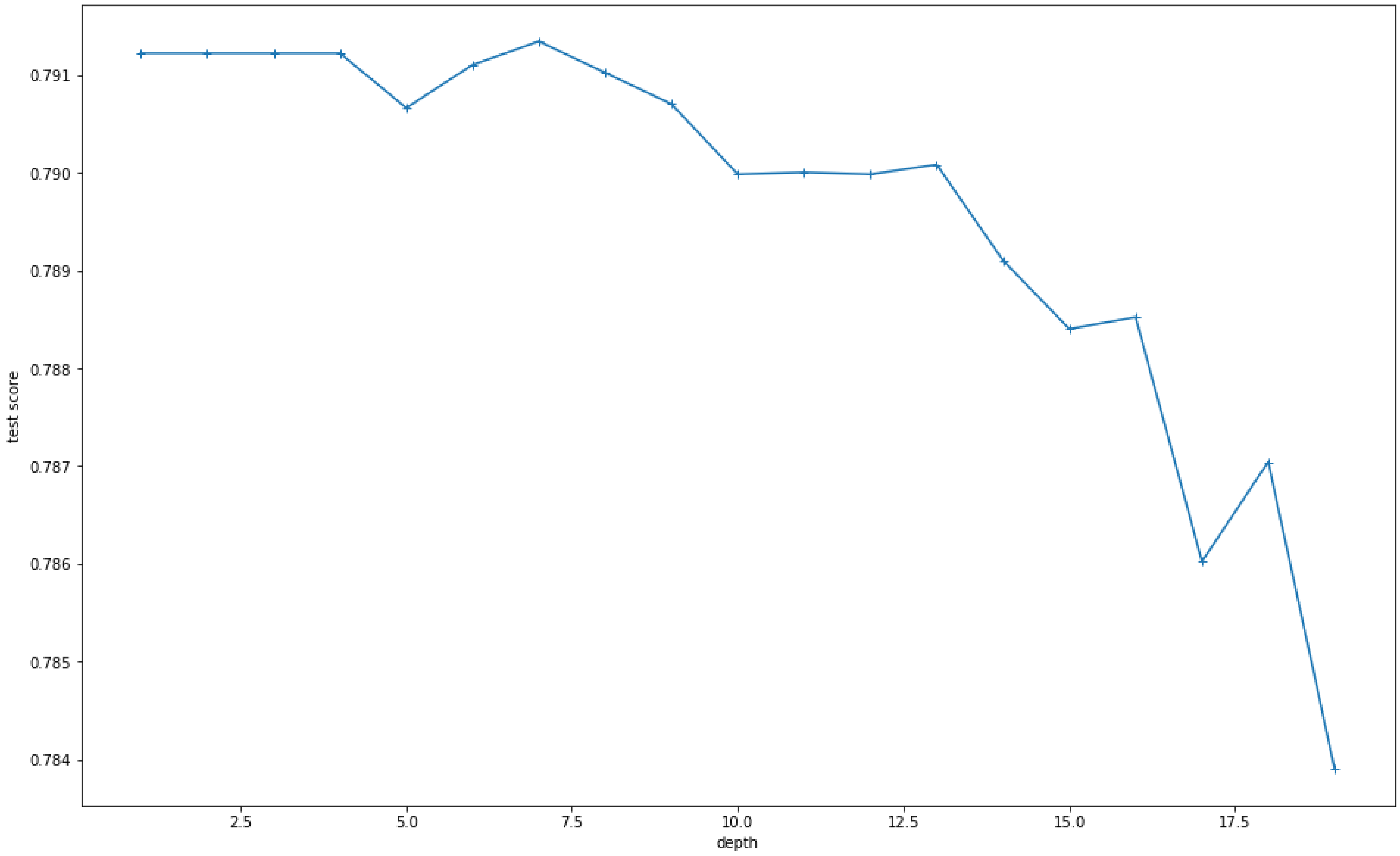
First, we try to find an *optimal maximum* depth for a random forest classifier using cross validation.

```
In [22]: depths, scores = [], []
```

```
         for max_depth in range(1, 20):
             clf = RandomForestClassifier(max_depth=max_depth)
             score = cross_val_score(clf, x1.loc[:50000], y_train.loc[:50000, "ReturnQuantityBin"], cv=5).mean()
             depths.append(max_depth)
             scores.append(score)
             print 'max_depth:', max_depth, '\tscore:', score
```

```
max_depth: 1         score: 0.7912241767474417
max_depth: 2         score: 0.7912241767474417
max_depth: 3         score: 0.7912241767474417
max_depth: 4         score: 0.7912241767474417
max_depth: 5         score: 0.7906641747476417
max_depth: 6         score: 0.7911041787476417
max_depth: 7         score: 0.7913441707488417
max_depth: 8         score: 0.7910241787496418
max_depth: 9         score: 0.790704192744642
max_depth: 10         score: 0.7899842167458422
max_depth: 11         score: 0.7900041887442419
max_depth: 12         score: 0.7899841767450418
max_depth: 13         score: 0.7900842087410421
max_depth: 14         score: 0.7891041727390418
max_depth: 15         score: 0.7884041587376416
max_depth: 16         score: 0.7885242147336422
max_depth: 17         score: 0.7860241727158417
max_depth: 18         score: 0.7870443087298431
max_depth: 19         score: 0.7839042567114426
```

```
In [23]: plt.figure(figsize=(16, 10))
         plt.xlabel("depth")
         plt.ylabel("test score")
         plt.plot(depths, scores, '-+')
         plt.show()
```



Another attempt, without cross validation, just we dataset shuffling.

```
In [19]: for depth in range(1, 20):
             scores_train, scores_test, best_clf, best_score = shuffle(x1, y_train, lambda : RandomForestClassifier(max_depth=depth), step
             print "depth:", depth, "\tscore:", best_score
```

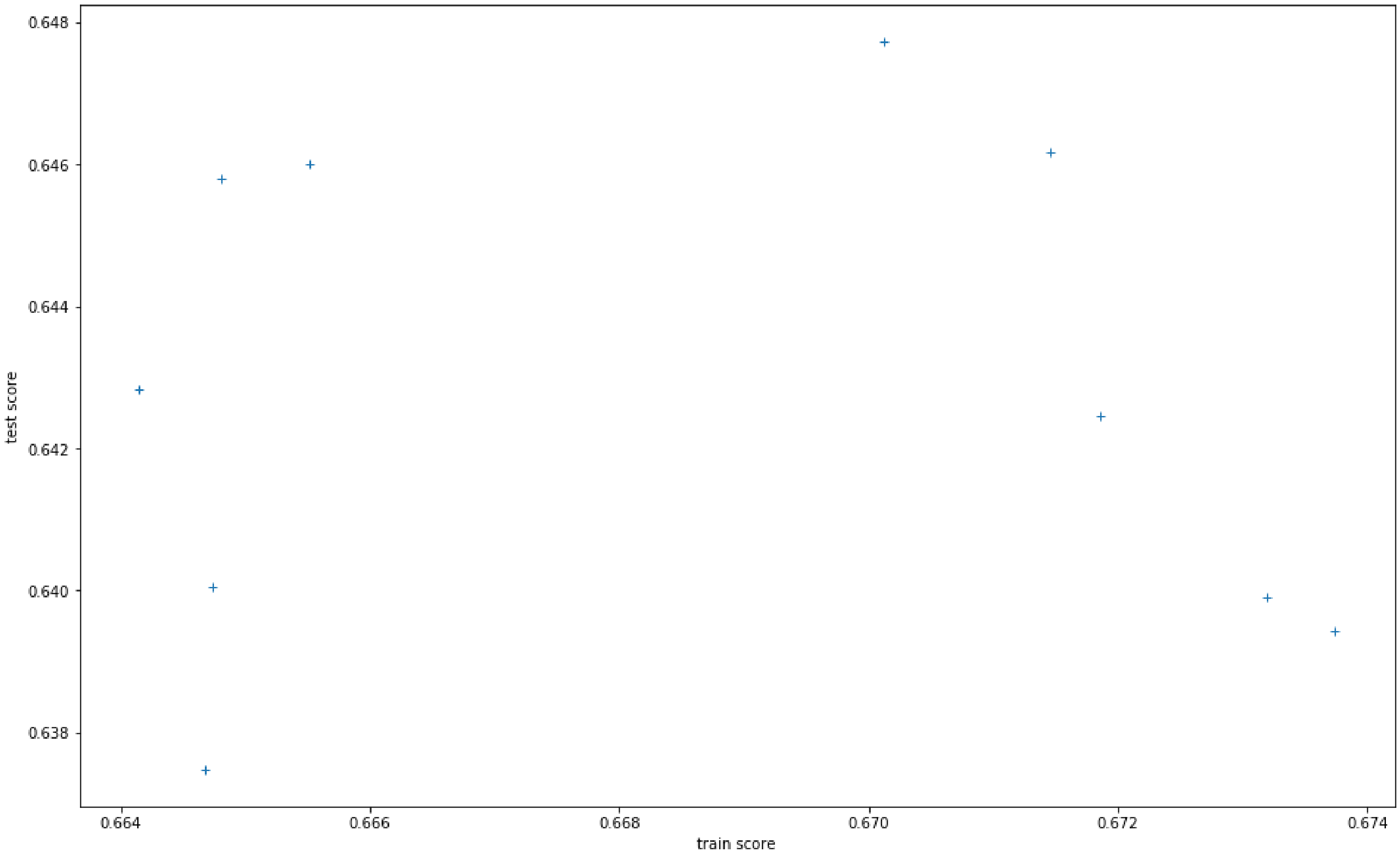
```
depth: 1         score: 0.6127363618489338
```

```
depth: 2         score: 0.6189378866971272
depth: 3         score: 0.6321598287938053
depth: 4         score: 0.6403760642500874
depth: 5         score: 0.6412143144576907
depth: 6         score: 0.6514337313548562
depth: 7         score: 0.6432620424000073
depth: 8         score: 0.6457797828784382
depth: 9         score: 0.6400541246853311
depth: 10         score: 0.6460751818421664
depth: 11         score: 0.6393927612274345
depth: 12         score: 0.6396849296339291
depth: 13         score: 0.6330789994311349
depth: 14         score: 0.6318776179566811
depth: 15         score: 0.6240696201724464
depth: 16         score: 0.6212071705903822
depth: 17         score: 0.6120505879455943
depth: 18         score: 0.6141664183891156
depth: 19         score: 0.6093554898816067
```

Now we try

```
In [20]: shuffle(x1, y_train, lambda : RandomForestClassifier(max_depth=6), steps=10)
```

```
test 0         train: 0.6701269236262677         test: 0.6477319675303895
test 1         train: 0.6714602896444263         test: 0.6461781632003937
test 2         train: 0.6648065317801488         test: 0.6458100691117841
test 3         train: 0.6647409730713175         test: 0.6400528431178711
test 4         train: 0.6731866436449108         test: 0.63990671741453
test 5         train: 0.6718527999500885         test: 0.6424648218756919
test 6         train: 0.6646784939167376         test: 0.6374748539568927
test 7         train: 0.673734724850982         test: 0.6394368550570327
test 8         train: 0.6641498890165085         test: 0.6428375878644402
test 9         train: 0.6655171647298975         test: 0.6460130882283992
```



```
Out[20]: ([0.6701269236262677,
           0.6714602896444263,
           0.6648065317801488,
           0.6647409730713175,
           0.6731866436449108,
 0.6718527999500885,
 0.6646784939167376,
 0.673734724850982,
 0.6641498890165085,
 0.6655171647298975],
[0.6477319675303895,
 0.6461781632003937,
 0.6458100691117841,
 0.6400528431178711,
 0.63990671741453,
 0.6424648218756919,
 0.6374748539568927,
 0.6394368550570327,
 0.6428375878644402,
 0.6460130882283992],
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=6, max_features='auto', max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False),
0.6477319675303895)
```

```
In [169]: def prune(x1, x2, y, threshold=1, slc=100000, max_depth=9, do_score=True, do_print=True, do_plot=True):
              # building the random forest classifier on a suffled training set
              slc = 100000
              z = x1.copy(deep=True)
              z["ReturnQuantityBin"] = y_train.ReturnQuantityBin
              u = z.sample(frac=1)
              v = u.loc[:, ["ReturnQuantityBin"]]
              u = u.drop(["ReturnQuantityBin"], axis=1)
              forest = RandomForestClassifier(max_depth=9)
              forest.fit(u.iloc[:slc], v.ReturnQuantityBin[:slc])

              # computing test scores
              if do_score:
                  predict_test = forest.predict_proba(u.iloc[slc:2 * slc])
                  score_test = roc_auc_score(v.ReturnQuantityBin[slc:2 * slc], predict_test[:, 1])
                  print "test score:", score_test
```

```
    importances = forest.feature_importances_
    std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)
    indices = np.argsort(importances)[::-1]

    # print the feature ranking
    if do_print:
        print "Feature ranking:"
        for f in range(u.shape[1]):
            print("%d. %s (%f)" % (f + 1, u.columns[indices[f]], importances[indices[f]]))

    # plot the feature ranking
    if do_plot:
        plt.figure(figsize=(16, 10))
        plt.title("Feature importances")
        plt.bar(range(u.shape[1]), importances[indices], color="r", yerr=std[indices], align="center")
        plt.xticks(range(u.shape[1]), indices)
        plt.xlim([-1, u.shape[1]])
        plt.show()

    # pruning x1 and x2
    whitelist = [u.columns[indices[f]] for f in range(min(threshold, len(u.columns)))]
    x3 = x1.drop([col for col in x1.columns if col not in whitelist], axis=1)
    x4 = x2.drop([col for col in x2.columns if col not in whitelist], axis=1)
    return x3, x4

def find_threshold(x1, x2, y, slc=100000, max_depth=9):
    # building the random forest classifier on a suffled training set
    slc = 100000
    z = x1.copy(deep=True)
    z["ReturnQuantityBin"] = y_train.ReturnQuantityBin
    u = z.sample(frac=1)
    v = u.loc[:, ["ReturnQuantityBin"]]
    u = u.drop(["ReturnQuantityBin"], axis=1)
    forest = RandomForestClassifier(max_depth=9)
    forest.fit(u.iloc[:slc], v.ReturnQuantityBin[:slc])

    importances = forest.feature_importances_
    std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)
    indices = np.argsort(importances)[::-1]

    best_t, best_score = None, None
    for threshold in range(1, 101):
        whitelist = [u.columns[indices[f]] for f in range(min(threshold, len(u.columns)))]
        x3 = x1.drop([col for col in x1.columns if col not in whitelist], axis=1)
        x4 = x2.drop([col for col in x2.columns if col not in whitelist], axis=1)
        sc_train, sc_test, clf, score = shuffle(x3, y_train, slc=100000, steps=2, plot=False)
        if best_t is None or best_score < score:
            best_t, best_score = threshold, score
        print "threshold:", threshold, "\tshape:", x3.shape, "\tscore:", score

    return best_t, best_score
```

```
In [170]: find_threshold(x1, x2, y_train)
```

```
threshold: 1         shape: (1067290, 1)         score: 0.574356605386431
threshold: 2         shape: (1067290, 2)         score: 0.5856617291218396
threshold: 3         shape: (1067290, 3)         score: 0.6154359104170999
threshold: 4         shape: (1067290, 4)         score: 0.6337566424354024
threshold: 5         shape: (1067290, 5)         score: 0.6351526668057986
threshold: 6         shape: (1067290, 6)         score: 0.6335189564429284
threshold: 7         shape: (1067290, 7)         score: 0.6355780083385949
threshold: 8         shape: (1067290, 8)         score: 0.636939926419611
threshold: 9         shape: (1067290, 9)         score: 0.6358006403113113
threshold: 10         shape: (1067290, 10)         score: 0.6405117479036697
threshold: 11         shape: (1067290, 11)         score: 0.6415424440098114
threshold: 12         shape: (1067290, 12)         score: 0.6412234090026194
threshold: 13         shape: (1067290, 13)         score: 0.6443138341425757
threshold: 14         shape: (1067290, 14)         score: 0.6420528173621186
threshold: 15         shape: (1067290, 15)         score: 0.6456081277227377
threshold: 16         shape: (1067290, 16)         score: 0.6447569832212464
threshold: 17         shape: (1067290, 17)         score: 0.6457928198211941
threshold: 18         shape: (1067290, 18)         score: 0.6463309977585872
threshold: 19         shape: (1067290, 19)         score: 0.6469198407627899
threshold: 20         shape: (1067290, 20)         score: 0.6459300320269035
threshold: 21         shape: (1067290, 21)         score: 0.6451033523300391
threshold: 22         shape: (1067290, 22)         score: 0.646443826512408
threshold: 23         shape: (1067290, 23)         score: 0.6476287452149596
threshold: 24         shape: (1067290, 24)         score: 0.6489343347297963
```

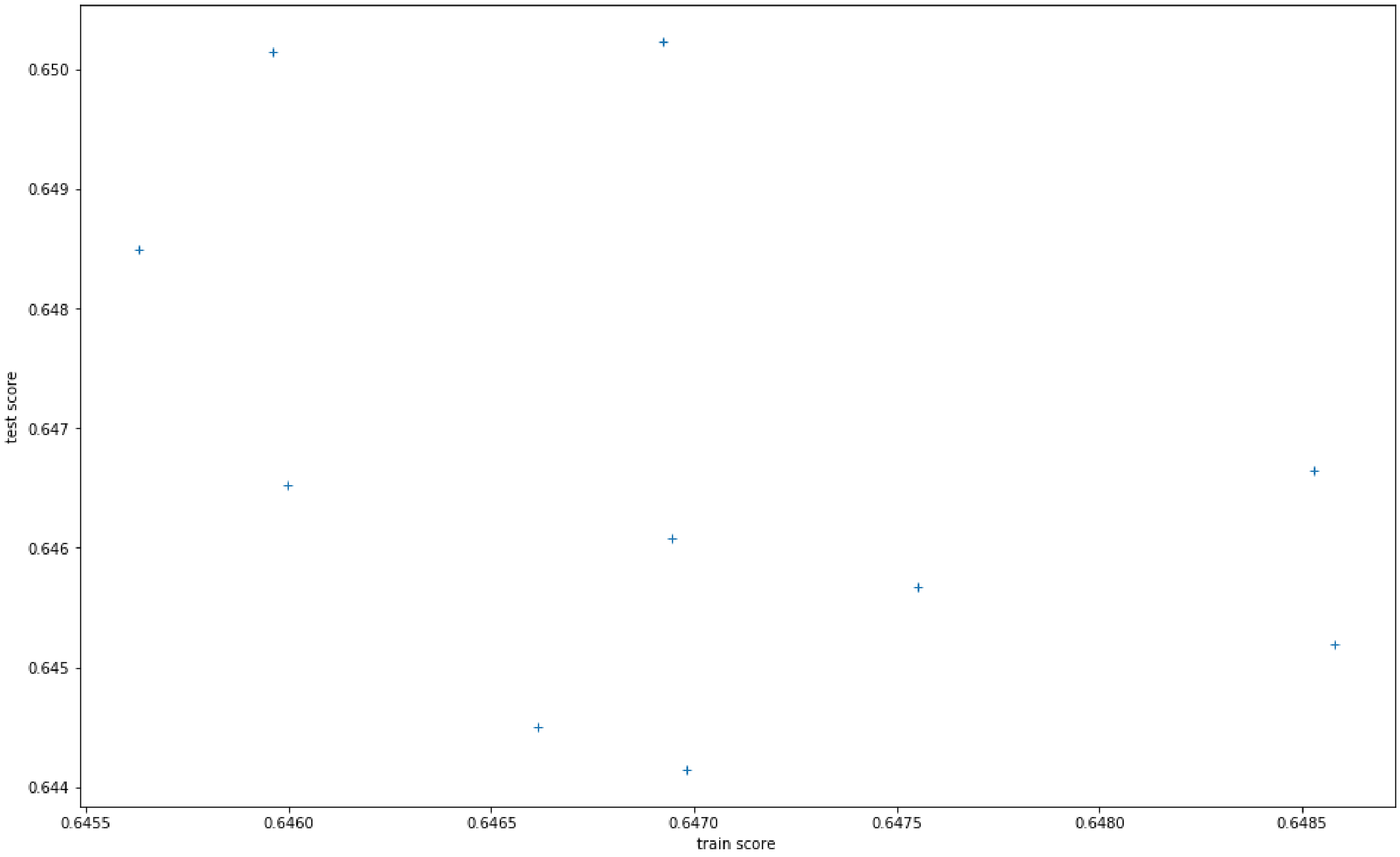
```
threshold: 25         shape: (1067290, 25)         score: 0.6487964287409811
threshold: 26         shape: (1067290, 26)         score: 0.6448783857278018
threshold: 27         shape: (1067290, 27)         score: 0.6485368465116832
threshold: 28         shape: (1067290, 28)         score: 0.6486301583341081
threshold: 29         shape: (1067290, 29)         score: 0.6485144251326611
threshold: 30         shape: (1067290, 30)         score: 0.64890651334502
threshold: 31         shape: (1067290, 31)         score: 0.6477219554626193
threshold: 32         shape: (1067290, 32)         score: 0.648766637340102
threshold: 33         shape: (1067290, 33)         score: 0.6532069089350414
threshold: 34         shape: (1067290, 34)         score: 0.6522325246425472
threshold: 35         shape: (1067290, 35)         score: 0.6504259565839989
threshold: 36         shape: (1067290, 36)         score: 0.6488052752460491
threshold: 37         shape: (1067290, 37)         score: 0.6539173510060831
threshold: 38         shape: (1067290, 38)         score: 0.6491065313888993
threshold: 39         shape: (1067290, 39)         score: 0.6502308793567172
threshold: 40         shape: (1067290, 40)         score: 0.6506077866574398
threshold: 41         shape: (1067290, 41)         score: 0.6537339130898341
threshold: 42         shape: (1067290, 42)         score: 0.6510050152717761
threshold: 43         shape: (1067290, 43)         score: 0.6530424101535781
threshold: 44         shape: (1067290, 44)         score: 0.6474548756821336
threshold: 45         shape: (1067290, 45)         score: 0.6507491900527157
threshold: 46         shape: (1067290, 46)         score: 0.653299574890476
threshold: 47         shape: (1067290, 47)         score: 0.6529056725471153
threshold: 48         shape: (1067290, 48)         score: 0.6523421940365526
threshold: 49         shape: (1067290, 49)         score: 0.6524679795342836
threshold: 50         shape: (1067290, 50)         score: 0.6482041290684625
threshold: 51         shape: (1067290, 51)         score: 0.6539802570402597
threshold: 52         shape: (1067290, 52)         score: 0.6517596281633151
threshold: 53         shape: (1067290, 53)         score: 0.6525032299485756
threshold: 54         shape: (1067290, 54)         score: 0.6518686850616213
threshold: 55         shape: (1067290, 55)         score: 0.6533365441127534
threshold: 56         shape: (1067290, 56)         score: 0.6504372939996386
threshold: 57         shape: (1067290, 57)         score: 0.653757515831481
threshold: 58         shape: (1067290, 58)         score: 0.6521883021129744
threshold: 59         shape: (1067290, 59)         score: 0.6531565501152934
threshold: 60         shape: (1067290, 60)         score: 0.6512906521925255
threshold: 61         shape: (1067290, 61)         score: 0.6534267007440926
threshold: 62         shape: (1067290, 62)         score: 0.6526592756398673
threshold: 63         shape: (1067290, 63)         score: 0.6559217414276409
```

```
threshold: 64         shape: (1067290, 64)         score: 0.6560805199436732
threshold: 65         shape: (1067290, 65)         score: 0.651927870894848
threshold: 66         shape: (1067290, 66)         score: 0.6541558660710156
threshold: 67         shape: (1067290, 67)         score: 0.6546857848590313
threshold: 68         shape: (1067290, 68)         score: 0.657293670948485
threshold: 69         shape: (1067290, 69)         score: 0.6550863599736362
threshold: 70         shape: (1067290, 70)         score: 0.6517567119092338
threshold: 71         shape: (1067290, 71)         score: 0.6535258569226026
threshold: 72         shape: (1067290, 72)         score: 0.6537079053854153
threshold: 73         shape: (1067290, 73)         score: 0.6519155054612111
threshold: 74         shape: (1067290, 74)         score: 0.6536214599344954
threshold: 75         shape: (1067290, 75)         score: 0.6534983552654517
threshold: 76         shape: (1067290, 76)         score: 0.6548824344874385
threshold: 77         shape: (1067290, 77)         score: 0.6536674290767304
threshold: 78         shape: (1067290, 78)         score: 0.6522533658478034
threshold: 79         shape: (1067290, 79)         score: 0.6533295865622228
threshold: 80         shape: (1067290, 80)         score: 0.6531193399911166
threshold: 81         shape: (1067290, 81)         score: 0.6487721276743434
threshold: 82         shape: (1067290, 82)         score: 0.6526855961912508
threshold: 83         shape: (1067290, 83)         score: 0.6546857501197395
threshold: 84         shape: (1067290, 84)         score: 0.6539547212741288
threshold: 85         shape: (1067290, 85)         score: 0.6545817634214093
threshold: 86         shape: (1067290, 86)         score: 0.6543700013215863
threshold: 87         shape: (1067290, 87)         score: 0.6520411843948619
threshold: 88         shape: (1067290, 88)         score: 0.6560924354716453
threshold: 89         shape: (1067290, 89)         score: 0.6529521079284721
threshold: 90         shape: (1067290, 90)         score: 0.6529490199540005
threshold: 91         shape: (1067290, 91)         score: 0.6554780721719459
threshold: 92         shape: (1067290, 92)         score: 0.655081067871826
threshold: 93         shape: (1067290, 93)         score: 0.6519487659659278
threshold: 94         shape: (1067290, 94)         score: 0.6541754728708041
threshold: 95         shape: (1067290, 95)         score: 0.6514389515947504
threshold: 96         shape: (1067290, 96)         score: 0.6534321968221015
threshold: 97         shape: (1067290, 97)         score: 0.6517552319654643
threshold: 98         shape: (1067290, 98)         score: 0.655481856314928
threshold: 99         shape: (1067290, 99)         score: 0.6546152950705799
threshold: 100         shape: (1067290, 100)         score: 0.6571203334358342
```

```
Out[170]: (68, 0.657293670948485)
```

```
In [158]: x3, x4 = prune(x1, x2, y_train, threshold=68)
```

```
test 0         train: 0.6459979624375067         test: 0.6465171859759469
test 1         train: 0.6469237937231148         test: 0.6502324199041669
test 2         train: 0.6456313523173876         test: 0.6484931902863063
test 3         train: 0.6485272949490786         test: 0.646645495718916
test 4         train: 0.6466164986683063         test: 0.6444993330427363
test 5         train: 0.6475535061472868         test: 0.6456752369675604
test 6         train: 0.6459622359306412         test: 0.6501444747268446
test 7         train: 0.6469440086651036         test: 0.6460775489289443
test 8         train: 0.648579527639573         test: 0.6451953263351881
test 9         train: 0.6469833710062032         test: 0.6441433060618417
```


0.650
0.649
0.648
0.647
0.646
0.645
0.644
test score
0.6455
0.6460
0.6465
0.6470
0.6475
0.6480
0.6485
train score