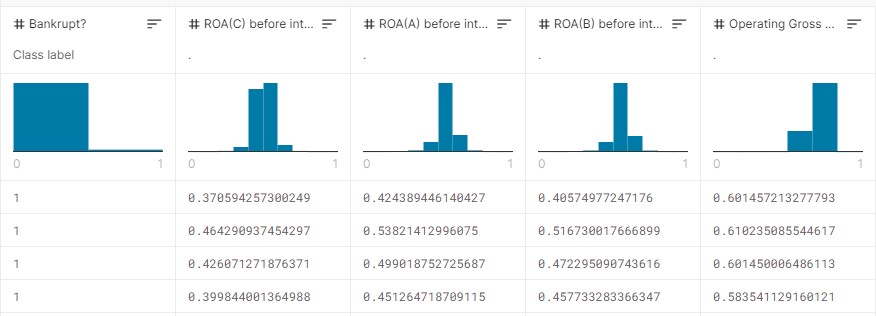
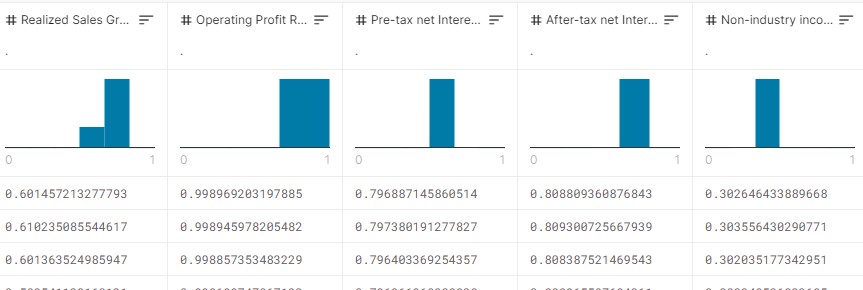
Поскольку выбранный набор данных нацелен на классификацию, мы можем считать, что в рамках данной работы мы осуществляем предсказание (predict) банкротства предприятия на основе переменных, которые характеризуют бизнес. То есть перед нами находится стандартная задача классификации.

Данное исследование может быть актуально в рамках бизнес аналитики. Инвесторам может быть полезно применение подобной системы в задачах отбора акций в бизнес портфель.

Таблица 2 – Классы датасета





# 3. Работа с данными

# 3.1 Чтение данных

Для начала установим и подключим все необходимые библиотеки для работы:

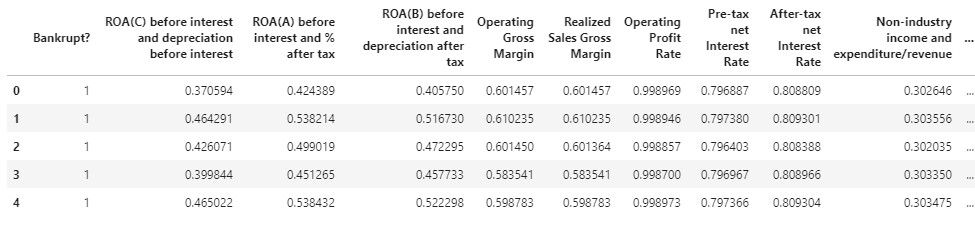
*#подключим нужные библиотеки и читаем данные*import pandas  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
from matplotlib.colors import ListedColormap  
%matplotlib inline  
import hashlib  
from sklearn import preprocessing  
from sklearn.utils import shuffle  
from sklearn.linear\_model import LinearRegression  
from pandas.plotting import scatter\_matrix  
from sklearn.metrics import mean\_squared\_error  
from sklearn.ensemble import AdaBoostClassifier  
from tensorflow.keras.models import Sequentialfrom keras.layers import Dense  
from keras.layers import LSTMfrom tensorflow import keras  
from tensorflow.keras.models import Sequential  
import tensorflow as tf

После чего загрузим датасет и взглянем что в себя включают первые записи:

data = pandas.read\_csv(**"Company Bankruptcy Prediction.csv"**)

data[:5]

Вывод представлен на рисунке 1.

Рисунок 1 – вывод первых пяти строк данных

К сожалению, мы не можем вывести в строку все 96 столбцов, но по полученным данным видно, что многие свойства не требуют нормализации.

Немного подробнее взглянуть на датасет позволит команда библиотеки Pandas describe, которая используется для просмотра основных статистических показателей count – число элементов, Mean – среднее, Std – среднеквадратическое отклонение, min – минимальное значение показателя

25%, 50% и 75% - значение показателя в указанных значениях max - максимальное значение показателя

data.describe()

Вывод представлен на рисунке 2.

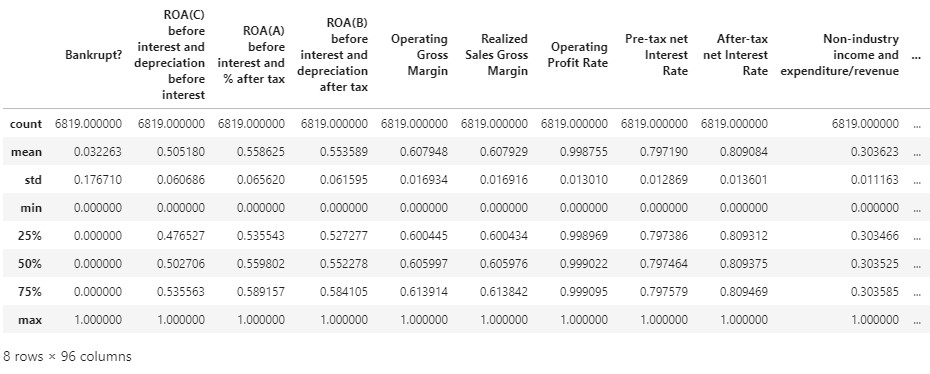


Рисунок 3 – общие статистические сведения о наборе данных

Здесь мы наблюдаем, что распределение лэйблов – банкротство компании клонится в сторону 0, т.е. не банкротства, однако если мы оставим набор данных в таком виде, то любая модель будет указывать всегда 0 и будет права в 95% случаев, поэтому в будущем нам потребуется уравнять случаи 0 и 1.

Выведем список всех параметров

data.info()

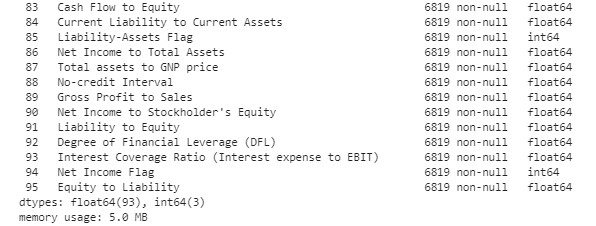


Рисунок 4 – список параметров

Исходя из полученного списка можно понять, что датасет не нормализован предварительно, поскольку присутствуют параметры типы int

Построим гистограммы по каждому параметру:

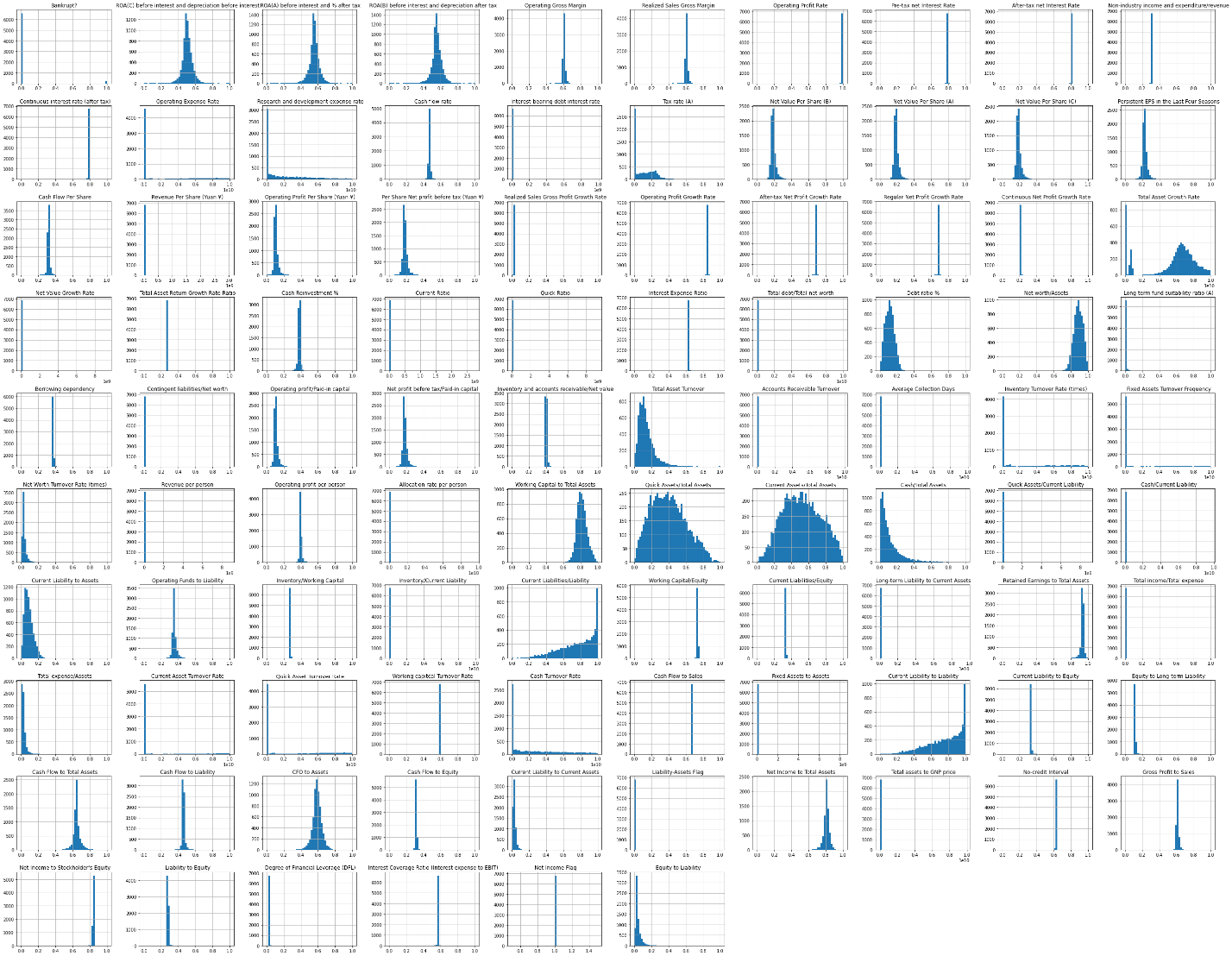


Рисунок 5 – Матрица гистограмм

Параметров очень много, но даже так видно, что не все параметры имеют хорошее распределение. Многие формируют столбцы. Такие необходимо будет отфильтровать. Пример негодного параметра:

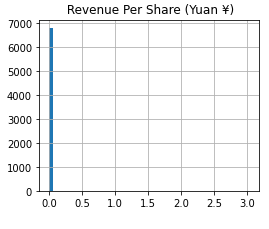


Рисунок 6 – Нерепрезентативный параметр

# 3.2 Препроцессинг данных

Предварительная обработка данных является важным шагом в процессе интеллектуального анализа данных. Любая модель не может быть обучена верно, если данные имеют коллизии, выбросы и в целом не репрезентативны.

В рамках данного пункта будут рассмотрены следующие методы:

* Удаление параметров с малым распределением;
* Удаление случаев, содержащих null - значения признаков;
* Перемешивание данных.
* Обновление индекса.
* Разбитие данных на тренировочную и тестовую выборки
* Нормализация данных

Поскольку у нас все параметры не содержат null и None значений, мы можем не заполнять пустые позиции с помощью линейной регрессии. Более того операция удаления null элементов делается на всякий случай, дабы избежать неприятных ошибок при тренировке модели.

data.dropna(inplace=True)

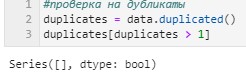


Рисунок 7 –Проверка на дубликаты.

Теперь займемся удалением параметров, имеющих ниское распределение. Прежде всего нам следует взглянуть на примеры нескольких таких параметров:

print(data.std(axis=0)[53:59])

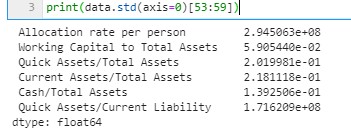


Рисунок 8 –распределение выбранных параметров.

Для удаления нерепрезентативных колонок мы воспользуемся среднеквадратическим распределением, однако прежде нам необходимо нормализовать данные, поскольку среднеквадратическое распределение возвращает число в значениях параметра, что не позволит нам проводить общее для всех сравнение.

x = data.values *#returns a numpy array*min\_max\_scaler = preprocessing.MinMaxScaler()  
x\_scaled = min\_max\_scaler.fit\_transform(x)  
dataRescaled = pd.DataFrame(x\_scaled)

Теперь мы можем провести фильтрацию параметров:

*#выберем атрибуты, дисперсии которых меньше 0.032 и удалим их.*columns = list(dataRescaled)  
  
to\_delete = abs(dataRescaled.std(axis=0))<0.032  
  
columnsToDelete = []  
for i in columns:  
 delete\_this\_col = to\_delete[columns.index(i)]  
 if delete\_this\_col:  
 columnsToDelete.append(columns.index(i))  
  
datafiltered = data.drop(data.columns[columnsToDelete], axis=1)  
print(**"удалено: "**, len(columnsToDelete))  
print(**"Осталось атрибутов: "**, len(columns)-len(columnsToDelete))



Рисунок 9 –Результат удаления.

Посмотрим на матрицу оставшихся атрибутов:

datafiltered.hist(bins=50, figsize=(50,40))  
plt.show()

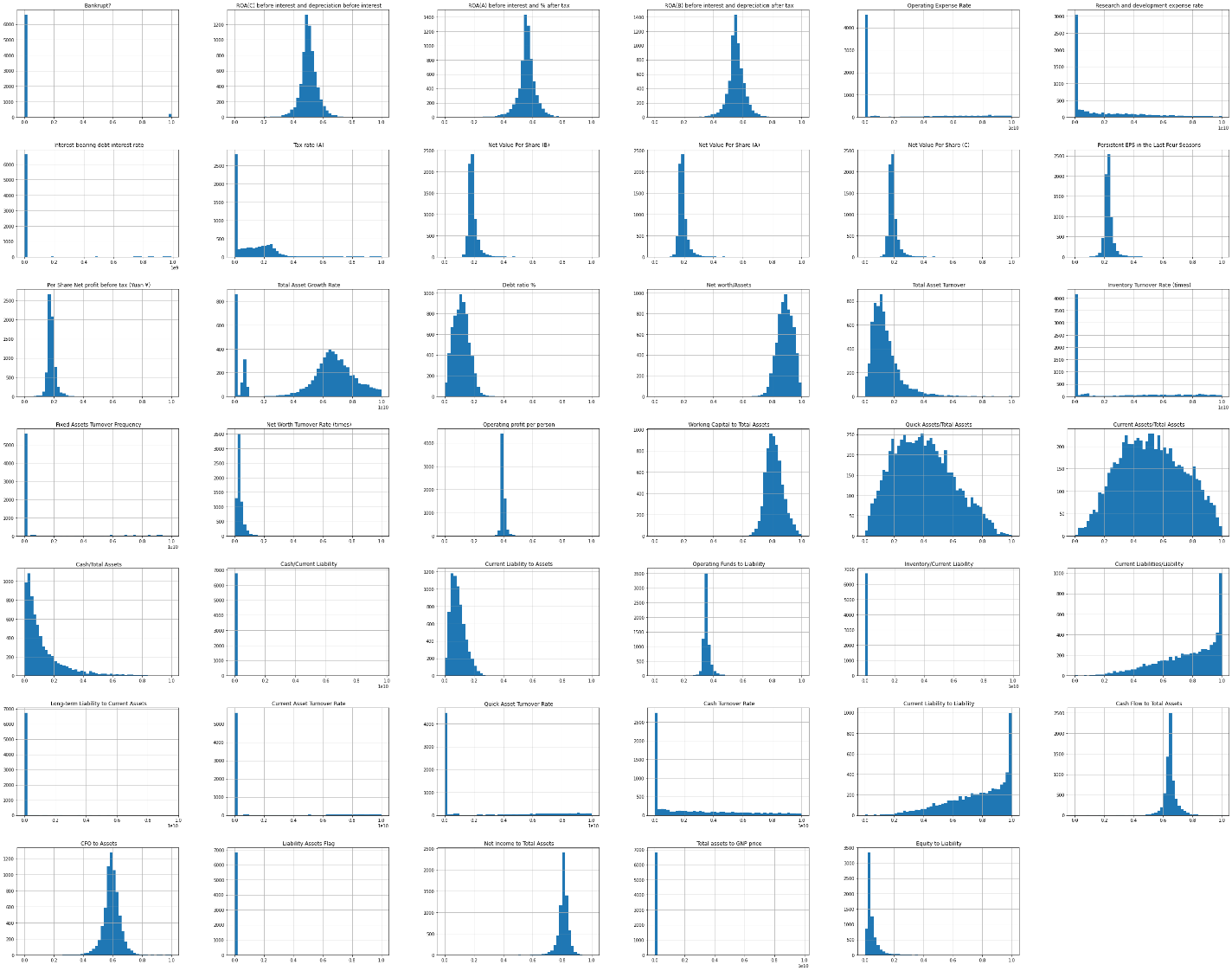


Рисунок 10 – Матрица гистограмм репрезентативных атрибутов

Такая картина нас устраивает намного больше. Конечно остались некоторые сомнительные фичи, однако увеличение значения среднеквадратического распределения удалит почти 80% изначального датасета, что тоже не хорошо.

Теперь можно заняться исправлением неравного распределения событий банкротства и выживания компании :

datafiltered [**'Bankrupt?'**].value\_counts()

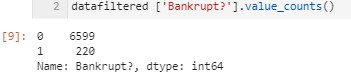


Рисунок 11 – распределение лэйблов.

datacopy = datafiltered.copy()  
datacopy = datacopy[datacopy[**'Bankrupt?'**]==1]  
need = datafiltered[datafiltered[**'Bankrupt?'**]==0]  
need2 = abs(int(len(need)/len(datacopy)))  
  
for each in range(need2):  
 datafiltered = datafiltered.append(datacopy,ignore\_index=True)  
  
  
print(**"теперь размер данных"**,len(datafiltered))

*#перемешаем данные и проверим количество лэйблов*dataproceesed = shuffle(datafiltered)  
dataproceesed [**'Bankrupt?'**].value\_counts()


Рисунок 12 – Результат исправления пропорции лэйблов.

На этом этапе мы можем проверить зависимость фич нашего датасета. Для этого мы выведем значения корреляции каждого параметра с лэйблом :

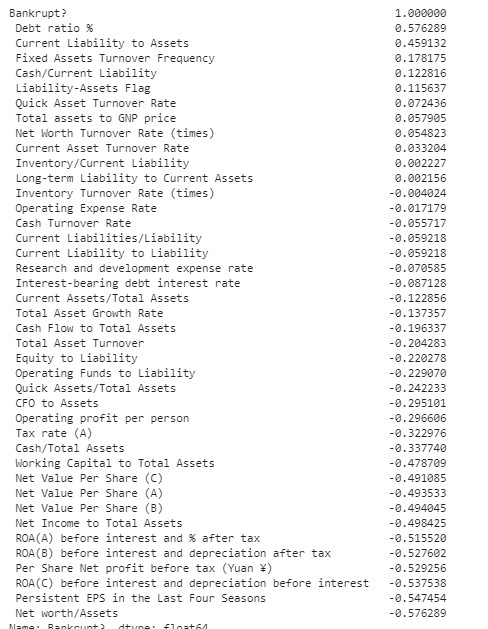


Рисунок 13 – Корреляция лэйбла с другими параметрами.

Мы видим сильную линейную корреляцию почти у половины параметров.

Давайте взглянем на данные более подробно, для этого мы воспользуемся параметром scatter\_matrix, который выводит матрицу графиков типа scatter из всех имеющихся параметров,

scatter\_matrix(dataproceesed, alpha=0.3, figsize=(200,180), diagonal=**'kde'**)  
*# plt.savefig('BIG\_PLOTS.png',dpi = 100)*

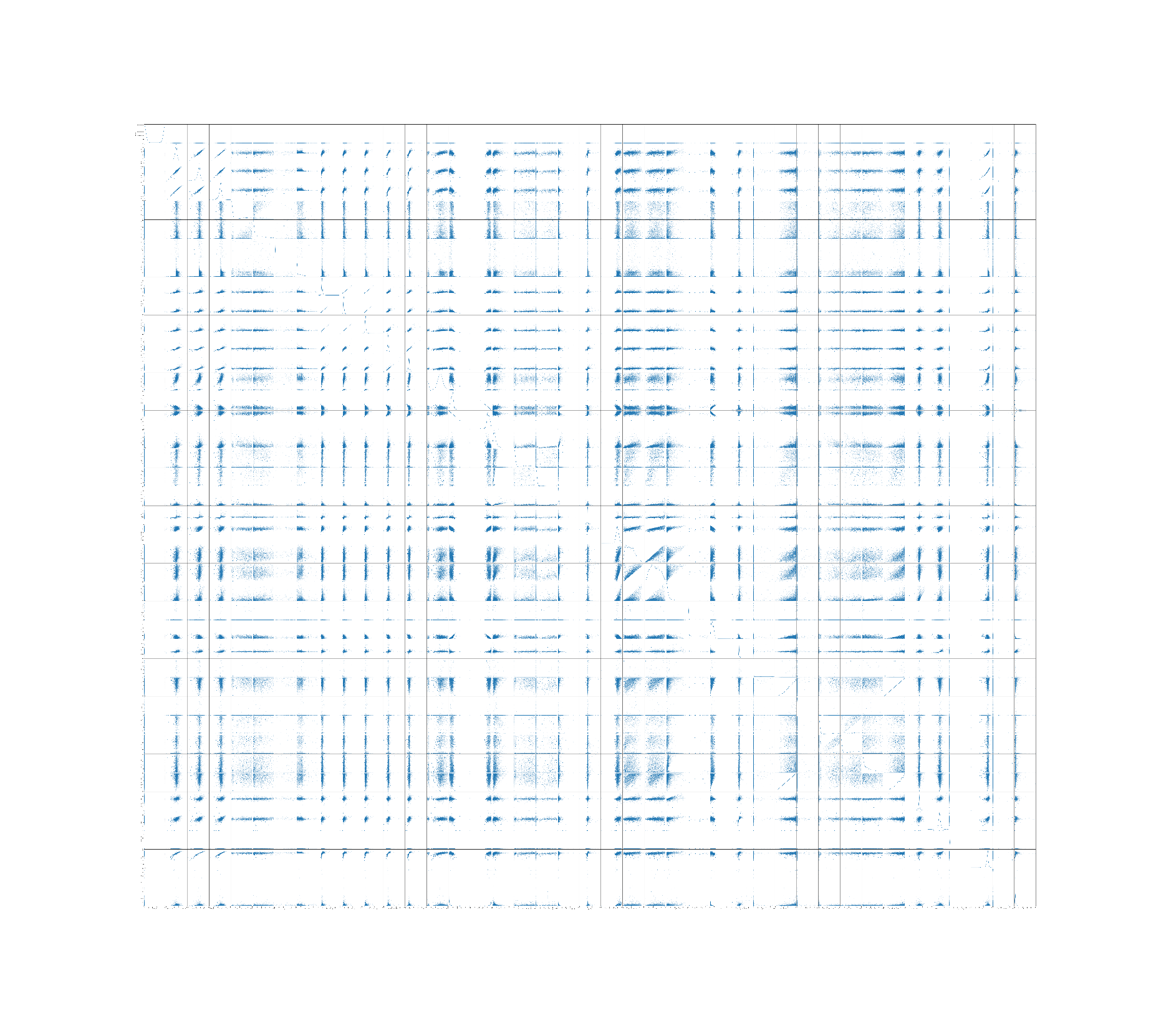


Рисунок 14 – матрица отношения всех параметров друг к другу.

Результат работы выдает очень большую матрицу, но не смотря на неудобство мы можем наблюдать отношение каждого параметра к каждому другому. Наиболее интересны для нас случаи, формирующие линейные зависимости. Пример:

dataproceesed.plot(kind=**"scatter"**, x=2, y=38, alpha=0.1)

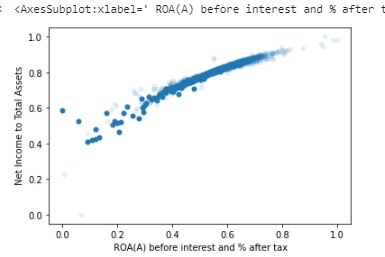


Рисунок 15 – Линейная корреляция параметров.

Нормализуем данные:

x = dataproceesed.values *#returns a numpy array*min\_max\_scaler = preprocessing.MinMaxScaler()  
x\_scaled = min\_max\_scaler.fit\_transform(x)  
dataproceesed = pd.DataFrame(x\_scaled)

Разделение на тестовую выборку и тренировочную методом хеширования, перемешивание данных, обновление индекса:

def test\_set\_check(identifier, test\_ratio, hash):  
 return hash(np.int64(identifier)).digest()[-1] < 256 \* test\_ratio  
def split\_train\_test\_by\_id(data, test\_ratio, id\_column, hash=hashlib.md5):  
 ids = data[id\_column]  
 in\_test\_set = ids.apply(lambda id\_: test\_set\_check(id\_, test\_ratio, hash))  
 return data.loc[~in\_test\_set], data.loc[in\_test\_set]  
  
  
dataproceesed = shuffle(dataproceesed)  
pd.notnull(dataproceesed)  
  
  
data\_with\_id = dataproceesed.reset\_index() *# добавляет колонку с индексом*train\_set, test\_set = split\_train\_test\_by\_id(data\_with\_id, 0.2, **"index"**)  
  
print(len(train\_set), **"train +"**, len(test\_set), **"test"**)

Разделим каждую выборку на лэйблы и фичи:

columns = list(dataproceesed)  
columns.pop(0)  
  
train\_set\_labeles = train\_set[0]  
train\_set\_features = train\_set.loc[:, columns]  
  
test\_set\_labeles = test\_set[0]  
test\_set\_features = test\_set.loc[:, columns]  
  
test\_set\_features.head(3)

# 4. Описание алгоритмов машинного обучения

В рамках данной работы были выбраны следующие алгоритмы:

* Линейная регрессия
* Классификатор AdaBoostClassifier
* Recurrent Neural Networks

Опишем каждый из них в подпунктах.

# Линейная регрессия

Линейная регрессия – одна из важнейших и широко используемых техник регрессии. Эта самый простой метод регрессии. Одним из его достоинств является лёгкость интерпретации результатов.

Регрессия полезна для прогнозирования ответа на новые условия. Можно угадать потребление электроэнергии в жилом доме из данных температуры, времени суток и количества жильцов.

Регрессия используется во многих отраслях: экономика, компьютерные и социальные науки, прочее. Её важность растёт с доступностью больших данных.

Простая или одномерная линейная регрессия – случай линейной регрессии с единственной независимой переменной x.

Несмотря на свою простоту она может показывать хороший результат. В нашем случае она интересна поскольку мы наблюдали сильную корреляцию параметра банкротства от остальных показателей

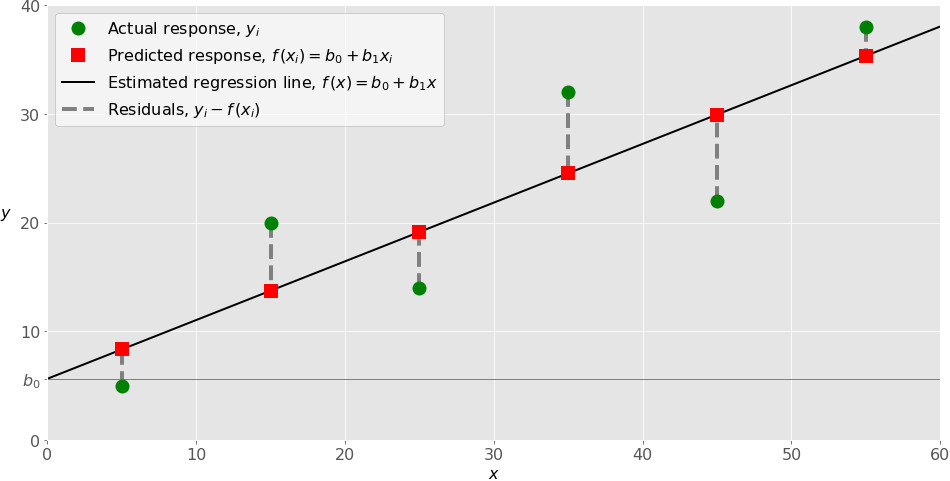


Рисунок 16 – Линейная регрессия.

# 4.2 Классификатор AdaBoostClassifier

В машинном обучении бустинг нужен для преобразования слабых классификаторов в сильные. Слабый обучающий алгоритм или классификатор – это обучающий алгоритм, который работает лучше, чем случайное угадывание, и работать хорошо он будет в случае переобучения, поскольку при большом наборе слабых классификаторов, любой слабый классификатор будет работать лучше, чем случайная выборка. В качестве слабого классификатора часто используется обычный threshold по определённому признаку. Если признак лежит выше threshold (порогового значения), чем было спрогнозировано, он относится к положительной области, в противном случае – к отрицательной.

AdaBoost означает «Adaptive Boosting» или адаптивный бустинг. Он превращает слабые обучающие алгоритмы в сильные для решения проблем классификации.

 Алгоритм усиливает классификаторы, объединяя их в «комитет». AdaBoost является адаптивным в том смысле, что каждый следующий комитет классификаторов строится по объектам, неверно классифицированным предыдущими комитетами. AdaBoost чувствителен к шуму в данных и [выбросам](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D0%B1%D1%80%D0%BE%D1%81_(%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B0)). Однако он менее подвержен [переобучению](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BE%D0%B1%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) по сравнению с другими алгоритмами машинного обучения.

AdaBoost вызывает слабые классификаторы в цикле t = 1,…,T. После каждого вызова обновляется распределение весов Dt , которые отвечают важности каждого из объектов обучающего множества для классификации. На каждой итерации веса каждого неверно классифицированного объекта возрастают, таким образом новый комитет классификаторов «фокусирует своё внимание» на этих объектах.

# 4.3 Recurrent Neural Networks

Рекуррентные нейронные сети (РНС, [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Recurrent neural network; RNN) — вид [нейронных сетей](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BA%D1%83%D1%81%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BD%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C), где связи между элементами образуют направленную последовательность. Благодаря этому появляется возможность обрабатывать серии событий во времени или последовательные пространственные цепочки. В отличие от многослойных [перцептронов](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D1%86%D0%B5%D0%BF%D1%82%D1%80%D0%BE%D0%BD), рекуррентные сети могут использовать свою внутреннюю память для обработки последовательностей произвольной длины.

Идея RNN заключается в последовательном использовании информации. В традиционных нейронных сетях подразумевается, что все входы и выходы независимы. Но для многих задач это не подходит. Если вы хотите предсказать следующее слово в предложении, лучше учитывать предшествующие ему слова. RNN называются рекуррентными, потому что они выполняют одну и ту же задачу для каждого элемента последовательности, причем выход зависит от предыдущих вычислений. Еще одна интерпретация RNN: это сети, у которых есть «память», которая учитывает предшествующую информацию. Теоретически RNN могут использовать информацию в произвольно длинных последовательностях, но на практике они ограничены лишь несколькими шагами

Рекуррентные нейронные сети продемонстрировали большой успех во многих задачах NLP. На этом этапе нужно упомянуть, что наиболее часто используемым типом RNN являются [LSTM](https://neurohive.io/ru/osnovy-data-science/lstm-nejronnaja-set/), которые намного лучше захватывают (хранят) долгосрочные зависимости, чем RNN. Но не волнуйтесь, [LSTM](https://neurohive.io/ru/osnovy-data-science/lstm-nejronnaja-set/) — это, по сути, то же самое, что и RNN, которые мы разберем в этом уроке, у них просто есть другой способ вычисления скрытого состояния. Более подробно мы рассмотрим LSTM в другом посте. Вот некоторые примеры приложений RNN в NLP

# 5. Анализ данных c помощью выбранных алгоритмов

Как уже было сказано для запуска кода понадобятся пакеты numpy, scipy, scikit-learn, matplotlib, pandas и tensorflow, а также keras

# 5.1 Анализ данных с помощью линейной регрессии

Воспользуемся готовой моделью из пакета scikit-learn для решения нашей задачи с помощью линейной регрессии:

lin\_reg = LinearRegression()  
lin\_reg.fit(train\_set\_features, train\_set\_labeles)

Модель построена и натренирована, теперь необходимо провести анализ её эффективности:

*# проверка на тренировочной выборке*predictions = lin\_reg.predict(train\_set\_features)  
lin\_mse = mean\_squared\_error(train\_set\_labeles, predictions)  
lin\_rmse = np.sqrt(lin\_mse)  
  
  
  
*# проверка на тестовой выборке*predictions2 = lin\_reg.predict(test\_set\_features)  
lin\_mse2 = mean\_squared\_error(test\_set\_labeles, predictions2)  
lin\_rmse2 = np.sqrt(lin\_mse2)  
  
print(**"ошибка на тренировочном датасете: "**,lin\_rmse\*100,**"%**\n**"**,**"ошибка на тестовом датасете: "**,lin\_rmse2\*100,**"%"**)



Рисунок 16 – Результат работы линейной регресии.

Как мы видим, линейная регрессия хотя и справляется с заданной задачей, но делает это с большой ошибкой. Причина очень проста – линейная регрессия не дает ответ в формате 1\0. Вместо этого она дает результат в вероятности выпадания одного из вариантов. Различия между ошибками тестовой и тренировочной выборок минимальны, что свидетельствует о адекватности и не перетренированности модели.

# 5.2 Анализ данных Классификатором AdaBoost

Следующим рассмотрим Классификатор AdaBoost:

ada = AdaBoostClassifier(base\_estimator=None,learning\_rate=1.4, algorithm=**"SAMME.R"**, n\_estimators=450)  
ada.fit(train\_set\_features, train\_set\_labeles)

Как и в предыдущем случае, мы воспользовались готовым инструментом из библиотеки scikit-learn. При подборе параметров получилось установить шаг обучения 1.4, количество оценщиков оказалось оптимальным на уровне 450. Проведем оценку модели:

*# проверка на тренировочной выборке*predictionsada = ada.predict(train\_set\_features)  
lin\_mseada = mean\_squared\_error(train\_set\_labeles, predictionsada)  
lin\_rmseada = np.sqrt(lin\_mseada)  
  
  
*# проверка на тестовой выборке*predictions2ada = ada.predict(test\_set\_features)  
lin\_mse2ada = mean\_squared\_error(test\_set\_labeles, predictions2ada)  
lin\_rmse2ada = np.sqrt(lin\_mse2ada)  
  
print(**"ошибка на тренировочном датасете: "**,lin\_rmseada\*100,**"%**\n**"**,**"ошибка на тестовом датасете: "**,lin\_rmse2ada\*100,**"%"**)



Рисунок 17 – Результат работы классификатора AdaBoost.

На тренировочном датасете ошибка составила всего 8%, что пожалуй неплохо для такой простой в построении модели, однако она оказалась перетренирована. Ошибка на тестовом наборе данных составляет 14.2%, что почти в двое больше тренировочной выборки. Тем не менее оба этих значения наименьшие из всего, что удалось достичь настройкой параметров.

# 5.3 Анализ данных с помощью Recurrent Neural Networks

Создание нейронной сети процесс более сложный, чем настройка параметров готовой модели. Рекурентная нейронная сеть была выбрана по той причине, что рекуррентные слои способны запоминать комбинации парметров и в целом система, имеющая 2 или 3 рекуррентных слоя в решающем ядре дает намного более качественные результаты. Для проектирования нейросети мы будем использовать библиотеку keras, входящую в пакет фрэймворка tensorflow

Прежде чем начать проектировать модель, нам необходимо подготовить данные . В первую очередь мы преобразуем массивы pandas, которые использовали до этого в матрицы numpy:

test\_set\_features=np.array(test\_set\_features,dtype=float)  
train\_set\_features=np.array(train\_set\_features,dtype=float)  
test\_set\_labeles=np.array(test\_set\_labeles,dtype=float)  
train\_set\_labeles=np.array(train\_set\_labeles,dtype=float)

Следующим шагом для нас станет преобразование свойств в вектора, поскольку рекурентная нейросеть работает с векторами. Пример преобразования : [[1,2,3],[4,5,6]] => [[[1],[2],[3]],[[4],[5],[6]]]

def make\_1d(matr):  
 list1 = []  
 list2 = []  
 list3 = []  
 for instnce in matr:  
 list2 = []   
 for element in instnce:  
 list3 = []  
 list3.append(float(element))  
 list2.append(list3)  
 list2.append(list3)  
 list1.append(list2)  
 return np.array(list1,dtype=float)

test\_set\_features = make\_1d(test\_set\_features)  
train\_set\_features = make\_1d(train\_set\_features)  
test\_set\_labeles = np.array(test\_set\_labeles,dtype=float)   
train\_set\_labeles = np.array(train\_set\_labeles,dtype=float)



Рисунок 18 – Размерность набора данных.

Проектируем и тестируем простейшую нейросеть с одним рекуррентным нейронным слоем:

model= tf.keras.Sequential()  
model.add(keras.layers.LSTM(1,batch\_input\_shape=(None,41,1)))  
model.compile(loss=**'mean\_absolute\_error'**, optimizer = **"adam"**,metrics=[**'accuracy'**])  
history = model.fit(train\_set\_features,train\_set\_labeles, epochs = 12)

Как можно наблюдать, у нас есть всего один слой типа LSTM, он выдает всего один параметр на выходе, на входе получает неопределенное количество входных данных. Я не указывал точное число, поскольку его не удобно искать, и оно не улучшает модель – с указанием None Фреймворк сам определяет его. 41 – количество фич в данных + количество лэйблов. 1 – размерность векторов в входных данных.

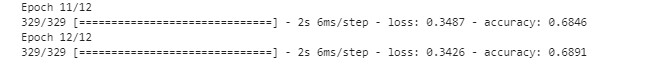


Рисунок 19 – Точность работы нейросети.

Точность нейросети составляет 69%, что очень хорошо для однослойной нейросети. Попробуем добавить простых слоев:

model= tf.keras.Sequential()  
  
  
model.add(keras.layers.Dense(41,batch\_input\_shape=(None,41,1), activation=tf.keras.activations.sigmoid))  
model.add(keras.layers.Dense(82,batch\_input\_shape=(None,41,1), activation=tf.keras.activations.sigmoid))  
model.add(keras.layers.Dense(164,batch\_input\_shape=(None,82,1), activation=tf.keras.activations.sigmoid))  
model.add(keras.layers.LSTM(20,batch\_input\_shape=(None,164,1), activation=tf.keras.activations.sigmoid))  
model.add(keras.layers.Dense(10,batch\_input\_shape=(None,20,1), activation=**'linear'**))  
model.add(keras.layers.Dense(1,batch\_input\_shape=(None,10,1), activation=**'linear'**))  
  
  
model.compile(loss=**'mean\_absolute\_error'**, optimizer = **"adam"**,metrics=[**'accuracy'**])  
model.summary()

history = model.fit(train\_set\_features,train\_set\_labeles, epochs = 18, validation\_data=(test\_set\_features,test\_set\_labeles))

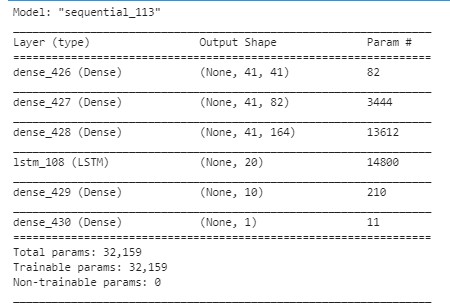


Рисунок 20 – описание нейросети.

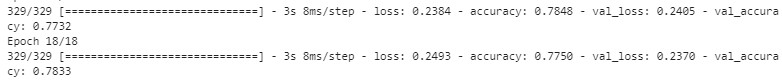


Рисунок 21 – точность нейросети.

Как мы можем наблюдать, несмотря на увеличение размеров и сложности нашей нейросети, мы ухудшили показатели точности. Возможно это по той причине, что наиболее важный слой типа LSTM содержит всего 20 нейронов, очевидно рекурентный слой нужен на месте предыдущего, но мы пойдем ещё дальше. Мы заменим LSTM слой на два слоя тип SimpleRNN, которые будут располагаться сразу после входа данных, а размерность их будет равна десятикратному количеству свойств. Так-же мы увеличим длину схождения нейросети, что позволит более плавно подводить решение двух основных слоев к финальному решению:

model= tf.keras.Sequential()  
  
model.add(keras.layers.Dense(410,batch\_input\_shape=(None,41,1), activation=tf.keras.activations.linear))  
model.add(keras.layers.SimpleRNN(410,batch\_input\_shape=(None,410,1), activation=tf.keras.activations.sigmoid, return\_sequences=True))  
model.add(keras.layers.SimpleRNN(410,batch\_input\_shape=(None,410,1), activation=tf.keras.activations.sigmoid, return\_sequences=False))  
model.add(keras.layers.Dense(205,batch\_input\_shape=(None,410,1), activation=tf.keras.activations.sigmoid))  
model.add(keras.layers.Dense(100,batch\_input\_shape=(None,205,1), activation=tf.keras.activations.sigmoid))  
  
model.add(keras.layers.Dense(50,batch\_input\_shape=(None,100,1), activation=tf.keras.activations.linear))  
model.add(keras.layers.Dense(20,batch\_input\_shape=(None,50,1), activation=tf.keras.activations.linear))  
model.add(keras.layers.Dense(20,batch\_input\_shape=(None,20,1), activation=tf.keras.activations.linear))  
model.add(keras.layers.Dense(10,batch\_input\_shape=(None,20,1), activation=tf.keras.activations.linear))  
model.add(keras.layers.Dense(1,batch\_input\_shape=(None,10,1), activation=tf.keras.activations.linear))  
model.add(keras.layers.Dense(1,batch\_input\_shape=(None,1,1), activation=tf.keras.activations.linear))  
  
model.compile(loss=**'mean\_squared\_error'**, optimizer = **"adam"**,metrics=[**'accuracy'**])  
model.summary()

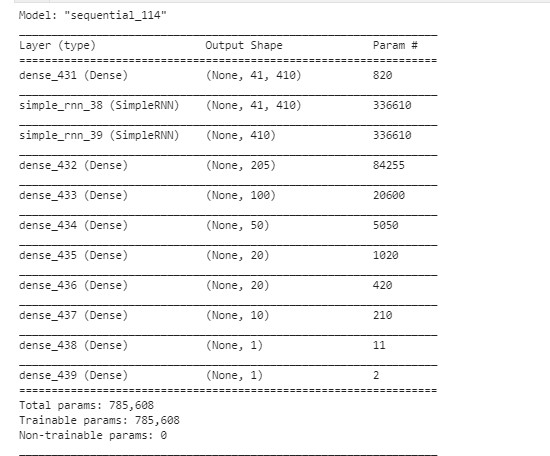


Рисунок 22 – описание нейросети.

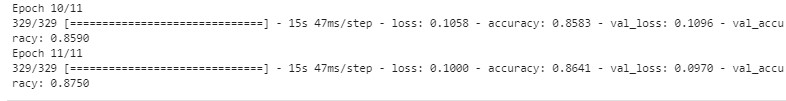


Рисунок 23 – точность нейросети.

Можно наблюдать выросшее значение точности до 86% на тренировочной выборке и как не странно 87,5% на тестовой! Наша модель не перетренирована и показывает эффективный результат, тем не менее он ниже, чем у AdaBoost. Причина проста - нейросеть, как и линейная регрессия делает вероятностный прогноз. Взглянем на показания нейросети:

results = model.predict(test\_set\_features)  
plt.scatter(range(len(results)),results,c=**"r"**)  
plt.scatter(range(len(test\_set\_labeles)),test\_set\_labeles,c=**"g"**)  
plt.show()

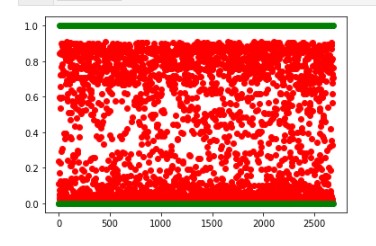


Рисунок 24 – распределение предположений нейросети в тестовой выборке.

Как мы видим, нейросеть не указывает точный вариант, однако в таком контексте 87,5% выглядят действительно хорошим результатом. Хочу заметить белую полосу в промежутке 0.9-1. Обычно такая полоса появляется при использовании слоев типа RELU, в нашей нейронной сети таковых нет, однако сдвиг наблюдается, возможно он связан с переходом от слоев с активационной функцией типа sigmoid к linear.

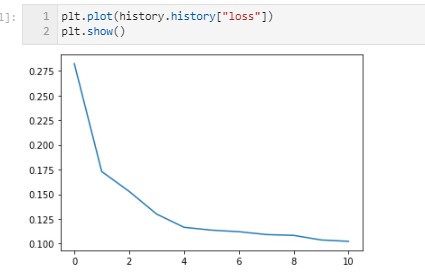


Рисунок 25 – изменение ошибки в процессе обучения.

# 6. Построение таблицы сравнения эффективности алгоритмов

Теперь произведем оценку полученных результатов в виде общей сводки.

|  |  |  |  |
| --- | --- | --- | --- |
| Категория | linear regression | AdaBoostClassifier | RNN |
| ошибка на тренировочном датасете | 34,15 | 8,16 | 13,59 |
| ошибка на тестовом датасете | 34,57 | 14,19 | 12,5 |

Исходя из результатов мы можем наблюдать , что наименьшая ошибка на тестовой выборке у рекурентной нейронной сети, она составила 12,5%. Чуть большая ошибка у классификатора AdaBoost, который при этом оказался перетренирован. Линейная регрессия закономерно имеет самый высокий показатель ошибки, стоит заметить, что из всех моделей она наименее затратна в тренировке.

# Заключение

В ходе работы были получены навыки обработки, подготовки и анализа данных. Мной было проведено исследование с помощью алгоритмов машинного обучения, построена нейронная сеть.

Чтобы решить бизнес-задачу классификации банкротства бизнеса, было использовано несколько моделей машинного обучения, а затем собрана общая сводка, чтобы увидеть, какая модель имеет лучший показатель точности. В нашем решении использовались алгоритмы линейной регрессии, классификатор AdaBoost и рекуррентной нейросети. Модель рекурентной нейросети показала наилучшие результаты точности - 87,5%, однако поскольку она дает вероятностное предположение, получить большее значение едва ли возможно. Для дальнейшего улучшения показаний следует результаты предположений нейросети передавать классификатору AdaBoost.