Pontifícia Universidade Católica do Rio de Janeiro

Classificação supervisionada de mensagens de texto de autenticação em dois-fatores

Antonio Carlos De Oliveira E Resende

Monografia Final de Pós-Graduação

Departamento de Engenharia Elétrica
Curso de Pós-Graduação em Ciência de Dados:
BI MASTER - Business Intelligence Master - Sistemas Inteligentes de Apoio à Decisão em Negócios

Rio de Janeiro, dezembro de 2019

Antonio Carlos De Oliveira E Resende

Classificação supervisionada de mensagens de texto de autenticação em dois-fatores

Monografia de final de curso, apresentada ao **Departamento de Engenharia Elétrica** da PUC-Rio como requisito parcial para a obtenção do título de Especialização em Business Intelligence.

Orientadora: Prof. Manoela Kohler

Rio de Janeiro
Dezembro de 2019

Resumo

Resende, Antonio; Kohler, Manoela. **Classificação supervisionada de mensagens de texto de autenticação em dois-fatores.** Rio de Janeiro, 2019. 38p. Monografia de final de curso – BI MASTER - Business Intelligence Master - Sistemas Inteligentes de Apoio à Decisão em Negócios – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

A segurança em validação de autenticação em dois fatores com a utilização de mensagens de texto passa pelo rápido envio e priorização destas mensagens frente à outros tipos das mesmas. Porém para que esta priorização seja rápida e eficiente é necessária uma automatização. O objetivo desta monografia é analisar e classificar mensagens de texto de acordo com um padrão estabelecido de mensagens de autenticação em dois fatores. Busca-se uma resolução do problema que seja robusta à um volume grande de dados para que o desempenho não seja afetado pelo mesmo.

Palavras-chave

Mensagens de Texto; Classificação Supervisionada; Autenticação em dois fatores.

Abstract

Resende, Antonio; Kohler, Manoela. **Supervised learning of two-factor authentication text messages.** Rio de Janeiro, 2019. 38p. Final Monography – BI MASTER - Business Intelligence Master – Intelligent Systems of Business Decision Support – Electric Engineering Department, Pontifical Catholic University of Rio de Janeiro.

The security involved with two-factor authentication with the use of text messages needs a fast sending and a good prioritizing that places them before other types of messages. In spite of this prioritizing to be fast and efficient needs to be automated. The objective of this work is to analyze and classify text messages according with messages with two-factor authentication pattern. It is desirable to find a solution that can support a big volume of data where the performance is not affected by it.

Keywords

Text Messages; Supervised Learning; Two-Factor Authentication.

Dedicatória

À minha vó, que sempre esteve comigo desde cedo, aos meu pais que sempre me estimularam a buscar mais e a minha irmã que me ajudou a crescer como pessoa.

Sumário

Figuras: gráficos e tabelas

1
Introdução

1.1
Contexto

Em uma realidade cada vez mais digital a importância que contas e senhas têm é algo crescente. Uma fraude online hoje muitas vezes é muito mais impactante e atinge proporções maiores que fraudes clássicas, praticadas há anos pela humanidade. “O roubo de contas de serviço pode prejudicar tanto suas finanças pessoais como sua imagem com o roubo de determinada conta de algum serviço. *Data Breach Investigations Report 2017* [1] da empresa Verizon indica que 81% das brechas analisadas para roubar contas se deve a senhas fracas.

Dado este fato, vários serviços enfatizam o uso de senhas cada vez mais fortes. Mas somente senhas fortes não cobrem totalmente já que temos vazamento de dados de diferentes fontes acontecendo em todo momento, então o simples fato de reutilizar uma senha pode resultar em uma intrusão não desejada por um *Hacker*.

Então como as empresas tentam resolver este problema? O uso de autenticação em dois fatores, ou como também é conhecido, *Two-Factor Authentication* [2]. Esse tipo de autenticação visa colocar mais uma camada de segurança entre o usuário e sua conta, verificando de alguma forma que a requisição de acesso veio realmente do dono da conta. Essa verificação pode ser feita basicamente de duas formas, chamadas de OTP (*One-Time Password*) [3] e TOTP (*Time-based One-Time Password*) [4]. A primeira se baseia em um código gerado pelo servidor e enviado para um e-mail ou número de telefone associado a conta. A segunda forma se baseia no uso de um código gerado por um dispositivo previamente autenticado, que não necessita de conectividade para gerar o código, e um novo código é gerado a cada determinado número de segundos, tornando difícil qualquer pessoa não autorizada se apossar do mesmo.

Neste contexto, dado o envio de mensagens de SMS e E-mail de autorização em dois fatores com códigos, é necessário que essas mensagens possuam uma entrega rápida dada uma requisição do usuário, e em meio a diferente tipos e volumes de mensagens seria muito difícil selecioná-las manualmente para que haja esta priorização no envio.

1.2
Objetivos Gerais

Tem-se como objetivo solucionar o problema de classificar as mensagens a serem enviadas para que elas tenham prioridade na fila de envio aos usuários. Para isso serão utilizadas técnicas de classificação binária supervisionada. Foram utilizadas técnicas de processamento de texto para preparar o mesmo e traçar características que evidenciam o caráter da mensagem pertencente ao grupo desejado.

Dado o grande volume a se trabalhar será utilizado o Spark por meio da interface de acesso à api pySpark.

1.3
Objetivos Específicos

O trabalho foi desenvolvido utilizando-se em quase sua totalidade de pacotes fornecidos dentro da plataforma pySpark para que o processamento em concorrência dele traga eficiência e velocidade a todo o processo, dado que é uma operação que deve ser realizada com certa rapidez devido sua natureza e contexto em que se encontra.

1.4
Motivação

A motivação deste trabalho é solucionar um problema de classificação de forma eficiente com o uso de processamento em paralelo e técnicas de *machine learning* a fim do projeto ser eficiente tanto em termos de classificação quanto de execução.

2
Fundamentação Teórica

O trabalho foi realizado com dados de textos curtos em formato .csv e explorados por meio de código python com Jupyter Notebooks e auxílio de ferramentas Apache Spark por meio de sua implementação python (PySpark), além de usos pontuais de bibliotecas numpy e pandas.

2.1
Descrição do Problema

Os envios de mensagens de verificações em dois fatores vêm aumentando [5] com a crescente preocupação com privacidade e segurança da informação atualmente. Com este aumento a necessidade da classificar de forma rápida e eficiente as mensagens que devem ser enviadas aos usuários para que as mensagens de maior prioridade, sendo as de autenticação de segurança uma das maiores, cheguem ao usuário sem maiores problemas.

A questão gera um problema de classificação de texto, porém com dificuldades inerentes à característica que textos curtos provocam sobre a análise e classificação, já que traçar perfis e características que diferenciem o bastante a classe alvo se torna uma tarefa mais complicada.

Existe ainda o problema do grande volume de dados. Então há a necessidade de buscar uma solução otimizada que suporte processar de forma paralela uma quantidade grande de dados em pouco tempo.

2.2
Regressão Logística

A modelagem logística é usada em estatística para modelar a probabilidade de um evento binário acontecer ou não.

A regressão logística [6] é um modelo estatístico que, em geral, usa uma função logística para modelar uma variável binária dependente. É uma forma de regressão binária, em que os parâmetros de um modelo logístico são estimados para que se possa prever corretamente a variável binária, que é representada por 0 e 1 matematicamente falando.

2.3
Random Forests

Também conhecidas como *random decision forests* (árvores de decisão randômicas), o *random forest* é um método de aprendizado *ensemble* usado para regressão, classificação e outras tarefas. Um método de aprendizado *ensemble* consiste em um conjunto de classificadores treinados individualmente, estes que servem de base para que seja construído um classificador final “forte” que combine os individuais, mais “fracos” e mais especializados.

O *random forest* opera por meio da construção de várias árvores de decisão em seu treinamento e como saída a classe que é a moda das classes, no caso de uma classificação ou a previsão média, no caso de uma regressão, das árvores individuais. Este processo acaba por corrigir as árvores de decisão que tendem a sofrer com o *overfit*, se especializar demais, com os dados usados de input para seu treinamento.

A árvore de decisão consiste em um modelo preditivo que traça conclusões sobre a variável alvo da previsão construindo uma árvore de folhas, que representam classes possíveis para o alvo, e galhos, que representam conjuntos de variáveis que caracterizam melhor tal classe.

2.4
Apache Spark

Apache Spark [7] é um sistema de processamento distribuído muito utilizado para tarefas que envolvem uma quantidade grande de dados. Seu código é aberto e ele utiliza armazenamento em cache na memória e otimiza a execução para que se obtenha alta performance em suas diferentes funções, dentre elas *machine learning*.

Hadoop MapReduce é um modelo de programação para processar muitos dados com algoritmos paralelizados e distribuídos. Ele gerencia a alocação de trabalho para o desenvolvedor e todo sistema de tolerância a falhas. Um problema do *MapReduce* é o processo sequencial de várias etapas que é necessário para se executar alguma tarefa. Para cada etapa existe um fluxo do *MapReduce* de ler os dados do cluster, realizar as operações desejadas, e escrever os dados novamente no HDFS (*Hadoop File System*). Por causa deste processo de leitura e escrita os trabalhos do MapReduce tem sua eficiência reduzida devido a latência em disco de operações de *input* e *output*.

O Spark foi criado para solucionar algumas limitações do *MapReduce*, realizando o processamento em memória, reduzindo o número de etapas em determinada tarefa, e reutilizando dados sempre que possível no maior número de operações paralelas possíveis. Reduzindo assim a tarefa a uma só leitura e escrita, com as operações realizadas entre estas duas etapas. A reutilização de dados se dá por meio da criação de *DataFrames*, uma abstração de RDD (*Resilient Distributed Dataset*), uma coleção de objetos em cache na memória que são reutilizados através das diversas operações do Spark.

Nativamente o Apache possui suporte para diversas linguagens como Scala, Java, Python e R, e a *API* torna mais simples o processo de programação distribuída para os desenvolvedores.

2.5 Expressões Regulares (Regex)

Expressões regulares ou *regex*, a abreviação em inglês é uma sintaxe independentemente de contexto que oferece uma forma simplificada de identificar cadeias de caracteres desejados [8]. Essas expressões possuem uma linguagem que pode ser interpretada para capturar caracteres específicos ou padrões. Baseada na expressão definida o interpretador irá procurar todas as *strings* ou o conjunto delas que se encaixa na sintaxe definida.

2.6 Tokenization

Tokenization é como é chamado o processo de demarcação e classificação de seções de *string* de caracteres. Uma frase contendo por exemplo um número x de palavras é transformada em um conjunto de x tokens após o processo de tokenização. Este resultado então é repassado a outro processo que utilizará essa separação para facilitar determinada análise ou processamento.

2.7 Stop words

Stopwords são palavras definidas como palavras mais comuns em uma determinada linguagem, e que em geral não carregam significado por si só. Para processamento e análise de textos em geral são palavras que geralmente são removidas para não poluírem a análise como um todo.

As *stopwords* podem variar de acordo com o contexto da análise ou do processamento que se deseja fazer.

2.8 TF-IDF

O TF-IDF, é uma abreviação para *term frequency-inverse document frequency,* que em português seria frequência de termo, frequência inversa de documento. O peso atribuído pelo TF-IDF é em geral um peso que é utilizado em processos de recuperação de informação ou mineração de texto. Esse peso é uma medida estatística atribui o quão importante é uma palavra frente ao texto completo ou a uma coleção de textos, no caso do trabalho desenvolvido, o quanto dada palavra é relevante frente ao conjunto completo de mensagens usadas na avaliação. O peso é proporcional ao número de vezes que dada palavra aparece no conjunto de textos e sofre um offset de acordo com a frequência da palavra no conjunto.

Como o nome indica, o peso é composto de duas componentes, o primeiro é a frequência normalizada de dado termo (TF), ou seja, o número de vezes que dada palavra aparece no texto dividida pelo número total de palavras no texto. O segundo é o IDF, que consiste no logaritmo do número de documentos que compõe o corpus, dividido pelo número de documentos onde o termo específico aparece.

3
Estudo de Caso

Uma empresa ao enviar milhões de SMSs diariamente precisa que eles sejam classificados de acordo com áreas de interesse como marketing ou acompanhamentos de pedidos. Com o aumento do volume de tráfego de dados, a tarefa que anteriormente era realizada de forma errática por funcionários humanos por meio de ações repetitivas, passou a se tornar cada vez mais cara e propensa a erros dado que as mensagens precisavam ser liberadas rapidamente para os clientes.

Neste cenário, surgem dois problemas centrais: um de desempenho computacional e um de classificação textual. Pode-se então usar a solução apresentada pelo Spark para lidar com o grande volume de dados e realizar rapidamente as transformações e preparações do dado cru para que a classificação seja entregue da forma mais rápida possível.

3.1
Formato dos dados e exploração inicial

Os dados foram fornecidos inicialmente em formato .csv obtidos através de um banco de dados. As verificações iniciais deixaram claro a variação dos dados que ocorriam sazonalmente, dado que há variações substanciais no número total de mensagens percebidas nos dias de semana (segunda-feira, terça-feira, quarta-feira, quinta-feira e sexta-feira) frente aos dias de final de semana (sábado e domingo), que possuem um número absoluto de mensagens menor.

Os números de mensagens totais variam desde 10 milhões por dia até cerca de 50 milhões de mensagens por dia. Destas mensagens, observou-se que cerca de 5% a 10% são da classe de interesse. Portanto estamos tratando de um problema desbalanceado, em que a representatividade em uma classe é muito maior que a outra, podendo causar problemas para o classificador ser efetivo.

Os dados apresentam somente três colunas: Um Id da conta cuja mensagem foi enviada, a data e hora deste envio e o conteúdo da mensagem. As duas primeiras colunas não apresentam características definidoras de nenhuma das classes, então foi escolhido trabalhar somente com o conteúdo delas.

3.2
Análise inicial textual

Após uma visualização inicial e exploração do conteúdo das mensagens presentes é possível notar alguns tipos de mensagens como monitoramento, marketing e confirmações diversas de aplicativos. Com alguns grupos definidos e com o uso de *regex* com palavras-chave visualizou-se as diferentes estruturas e possíveis características que estavam presentes em cada grupos.

Uma característica muito marcante de várias mensagens foi a presença de links, que poderia ser transformada em *feature* para descrever o comportamento de determinadas mensagens. Como contraexemplo temos mensagens com teor de monitoramento, em que a presença de tal característica não foi percebida.

A presença de datas em determinadas mensagens como avisos e alertas foi algo que chamou atenção, algo que é passível de ser capturado em massa para transformar em uma coluna característica.

Valores monetários também apareceram em diversas mensagens. Esses valores possuem marcadores bem definidos como “$” que tornam a capturas de tais blocos de texto facilitadas.

Timestamps aparecem em diversas mensagens, ou seja, é um padrão que pode ser facilmente detectado e utilizado para identificar os textos.

Foram feitos processos de tokenização de palavras para a visualização e contagem (Figura 1) do número das mesmas e perceber que possuíam maior impacto no grupo de mensagens classificados como *Two-Factor.*

words	count
codigo	3304422
whatsapp	2250655
verificar	1141464
link	1134992
neste	1102091
numero	1094128
toque	1063543
compartilhe	1062904
nnao	1041830
nou	1024768
facebook	913151
cartao	910431
compra	721504
2019	660906
fatura	572954
valor	565175
final	545557
use	503904
aprovada	500396
n4sglq1p5sv6	471169
voce	461637
https	442026
senha	422296
nao	371793
pagseguro	363107
account	329173
kit	323562
ate	305304
http	290317
login	271856
pagamento	256947
ola	255187

Figura 1 - contagens de palavras nos arrays produzidos

Com os *arrays* de palavras produzidos a contagem mostrou muitas palavras conhecidas como palavras vazias ou *stopwords* que são palavras que não apresentam grande significado e geralmente servem funções puramente gramaticais em uma frase. Então foi necessário remover as mesmas por meio da função “StopWordsRemover” [10] que já possui uma lista de palavras deste tipo para o Português Brasileiro.

Com a remoção foram obtidas palavras que apareciam mais vezes no grupo desejado e foram selecionadas as que mais se adequavam ao grupo desejado de forma semântica.

Figura 2 - Porcentagem de presença de palavras nos grupos estudados para palavras relevantes

A imagem acima (Figura 2) representa algumas palavras-chave identificadas como mais relevantes e a porcentagem de mensagens do grupo de *Two-Factor* que as possuíam. Assim conseguiu-se verificar uma relevância e certa correlação delas com a classe que se deseja identificar.

3.3 Criação de *features*

O primeiro passo foi utilizar a visualização das palavras-chave para transformar tais palavras em características relevantes para representar cada mensagem. Por meio de uso das funções do próprio pySpark foram criadas colunas binárias que representam a existência ou não da palavra em questão.

Na tentativa de criar *features* semânticas foram testadas duas bibliotecas: Polyglot e SpaCy. O Spacy apresentou quebra em sentenças e algumas classificações semânticas de pessoa, localização/endereço e outros, porém nas cerca de 20 frases testadas não apresentou bons resultados para o português contando com muitos erros, o que poderia agir contra o objetivo de caracterizar bem uma mensagem e confundir mais o treinamento do modelo.

Já o Polyglot apresentou divisões em 3 grupos, localização, organização e outros. Apesar dos testes se mostrarem melhores quando aumentada a carga de dados para processamento o desempenho foi fora do aceitável com os recursos computacionais

disponíveis e o volume de dados, então essa opção também acabou por ser descartada ao longo do desenvolvimento.

Foi necessário o uso de alguma técnica mais genérica para representar as mensagens como um todo e não ficar somente nas palavras mais relevantes. Assim seria possível diferenciar cada entrada por suas características gerais e não só pela presença de uma ou outra palavra. . A técnica escolhida foi o TF-IDF, porém, com esta técnica corremos o risco de termos palavras sem importância, as chamadas *StopWords*, com alto peso atrapalhando as palavras com real significância. Para contornar esse problema foi realizado o pré-processamento de retirada de *StopWords* para que essa situação não ocorresse.

Com a finalidade de melhorar a semelhança entre certas mensagens foram criadas algumas tokenizações para tornar o processo de TF-IDF mais eficiente e trazendo mais sentido a marcas textuais que estão presentes em diversas mensagens, porém nem sempre no mesmo formato.

Foram desenvolvidos cerca de 4 tokenizações que envolviam marcas textuais que denotavam certas características nas mensagens dentre elas:

- Tokenização de dígito: Para deixar as mensagens que seguem um padrão e tem dígitos diferentes, como por exemplo mensagens de autenticação em dois fatores que dada empresa ou serviço utiliza a mesma mensagem somente mudando o código de autorização. Os dígitos foram todos modificados para um mesmo dígito.
- Tokenização de datas: As datas e suas diversas formas com formatos xx/xx/xx e formatos com nomes de meses como por exemplo “janeiro” ou “Jan” foram alteradas para uma dada palavra (DDATE).
- Tokenização de tempo: Os horários também foram substituídos por um uma palavra, TTIME.
- Tokenização de dinheiro: As marcações de dinheiro foram todas alteradas para a palavra MMONEY.

Para cada uma dessas tokenizações foram criadas colunas binárias que descrevem se há ou não a presença deste token no texto analisado.

Foram criadas ainda duas *features* para capturar a presença ou não de dois formatos muito presentes nas mensagens que se deseja classificar, são elas: XXX-, XXXX-, -XXX, -XXXX.

3.4
Implementação de tratamento de dados

Para se trabalhar com o spark são necessárias as implementações de *transformers* e *pipelines* para melhor trabalhar com as funções e deixar as mesmas mais testáveis para garantir que estão com o funcionamento correto.

Os *transformers* são responsáveis por modificar e transformar *dataframes*. Eles implementam um método chamado *transform (),* que converte um *DataFrame* em outro por meio da adição de uma ou mais colunas.

Já as *pipelines* são classes que agrupam uma sequência de *transformers* e *estimators* (funções que recebem *DataFrames* e criam modelos).

Os scripts que implementam todas as transformações necessárias foram organizados em três arquivos distintos:

- transformers.py – Aqui se concentram as operações de preprocessamento em sua maioria. *Transformers* que removem *Stopwords*,
- tokenizers.py – Aqui se encontram as funções que trocam marcadores de texto que foram citados anteriormente como valores monetários, datas etc.
- features.py – Aqui se encontram as funções que criam as diversas colunas que compõem as *features* que descrevem cada uma das mensagens como as colunas de presenças de palavras específicas, o TF-IDF e as colunas de presença de *tokens*, por exemplo.

Os códigos comentados nas seções abaixo se encontram no apêndice ao final do trabalho.

3.4.1
Implementação de Transformers

O arquivo transformers.py possui as seguintes classes:

LowerCaseTransformer

A classe acima possui opções de inicialização de dois parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação

O método *transform* aplica na coluna de input a função *lower* da biblioteca *pyspark.sql. functions* que substitui todas as letras para suas versões minúsculas.

RemoveAccentTransformer

A classe acima possui opções de inicialização de dois parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação

O método *transform* possui uma lista com caracteres com acentos e suas substituições além de aplicar uma função de *replace* em cada linha da coluna para limpar estes caracteres indesejados.

RemoveSpecialCharactersTransformer

A classe acima possui opções de inicialização de dois parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação

O método *transform* aplica um *replace* ao encontrar os caracteres presentes na *regex* definida com caracteres especiais por espaço.

RemoveStopWordsTransformer

A classe acima possui opções de inicialização de dois parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação

O método *transform* aplica a função *RegexTokenizer* que divide os textos em *arrays* de palavras para posteriormente com a lista de *StopWords* em português do pySpark realizar a retirada do *array* e manter somente as palavras desejadas.

RemoveDuplicatesTransformer

A classe acima possui opções de inicialização de dois parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação

A função *transform* apaga as linhas que possuem textos iguais. O *transformer* é relevante em detectar mensagens iguais antes do pré-processamento para não gerar um viés de alguma forma no momento de treinamento dos algoritmos.

SMSTokenizer

A classe genérica *SMSTokenization* possui opções de inicialização dos seguintes parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação

- *regex*: *string* da *regex* que será usada para capturar a marca textual para ser substituída por um token
- *token*: o *token* que será usado para substituir as palavras capturadas pela *regex* definida

O método *transform* utiliza a *regex* definida na chamada da função para capturar a estrutura textual e substituí-la pelo token determinado.

Para cada estrutura textual desejada a classe abstrata *SMSTokenization* é reimplementada com sua respectiva *regex* e *token*, como o exemplo acima implementa para os dígitos numéricos e os troca para o numeral 9.

RegexMaskFeatureTransformer

A classe abstrata *SMSTokenization* possui opções de inicialização dos seguintes parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação
- *regex*: string da regex que será usada para capturar a marca textual e criar a coluna binária de acordo com a existência ou não da estrutura definida

O método *transform* irá procurar no texto presente na *iputColumn* e criar uma coluna binária baseada na existência ou não da estrutura definida pela *regex*.

A classe abstrata implementa duas *features* distintas *DigitsFirstMaskFeatureTransformer* e *DigitsSecondMaskFeatureTransformer* que buscam por estruturas de dígitos de formato XXXX- e -XXXX respectivamente.

HashingFeaturesTransformer

A classe HashingFeaturesTransformer possui opções de inicialização dos seguintes parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação
- *numFeatures*: número de features que serão criadas no vetor de TF
- *booleanHashing*: variável boleana para indicar se o vetor criado deverá conter valores diversos para caracterizar a mensagem ou apenas valores binários nas respectivas posições.

O método *transform* aplica o *hashingTF* com as colunas de input e devolve um *dataframe* com a coluna que representa o vetor criado para cada texto.

ContainsWordFeatureTransformer

A classe abstrata *ContainsWordFeatureTransformer* possui opções de inicialização dos seguintes parâmetros:

- *inputCol*: nome da coluna que será aplicada a operação
- *outputCol*: nome da coluna que será criada como resultado da operação
- *word*: palavra que será procurada para criar a coluna binária que representa a existência ou não da palavra

O método *transform* converte o texto da coluna de input em um array de palavras e busca nele a palavra definida na variável word. Após a operação uma coluna binária é criada de acordo com o resultado positivo ou negativo dessa busca, que é então escrito na linha respectiva.

Para cada palavra a classe abstrata é reimplementada com as variáveis de acordo, como mostra o exemplo acima com a palavra código.

3.4.2 Implementação de Pipelines

SMSPreprocess

A classe *SMSPreprocess* implementa uma *pipeline* de pré-processamento com o com os *transformers* de aplicação de letras minúsculas, remoção de acentos, caracteres, e *stop words*. Todas essas operações são padrões para os fluxos do projeto então são implementadas em uma classe separada para melhor legibilidade e organização do código.

SMSTokenization

A classe *SMSTokenization* implementa além das partes de pré-processamento os *transformers* de tokenização para diferentes estruturas definidas anteriormente.

FeaturePipeline

A classe *FeaturePipeline* implementa todos os *transformers* que criam *features* que serão usadas para criação dos modelos.

3.4.3
Modelos

Foram testados 2 algoritmos: *Regressão Logística*, *Random Forest*.

Nos três casos foi aplicada uma validação cruzada estratificada para evitar possíveis problemas de *overfitting* já que as classes são desbalanceadas nos *datasets* com a presença muito maior de textos que não pertencem a classe de autenticação em dois fatores.

4
Resultados

O objetivo deste trabalho foi detectar mensagens que podem ser categorizadas como autorizações de dois fatores por meio de um algoritmo de classificação supervisionada.

O problema foi apresentado com características que envolviam um grande volume de dados e a necessidade de criação completa de *features* para que os textos pudessem ser mais bem caracterizados.

4.1
Distribuição dos dados

É Two-Factor?	Contagem (Milhões)
VERDADEIRO	4,9
FALSO	20,2

Foram usados cerca de 100 milhões de mensagens para treinar e cerca de 25 milhões (tabela acima) para validar o modelo gerado.
Todo o processamento exigiu uma capacidade computacional grande e algumas horas de processamento.
Foram gerados e testados três modelos: *Logistic Regression e Random Forest.*
Como é possível perceber, temos um certo desbalanceamento de classes no conjunto de teste, que se repete no conjunto de treino portanto foi preciso utilizar um treinamento estratificado com validação cruzada para garantir que os modelos não acabassem realizando um *overfit* em determinada classe. A estratificação é usada para rearranjar cada conjunto de treino com um número de amostras de cada classe representativo do conjunto total. Foram usados nos treinamentos 5 conjuntos na estratificação.

4.2
Métricas de avaliação

Foi utilizada como principal métrica de avaliação dos modelos a chamada AUC-ROC. AUC (área embaixo da curva) representa o grau de separação e a ROC é uma curva de probabilidade. As duas mostram o quanto o modelo foi capaz de separar e distinguir as duas classes possíveis. Quanto maior a AUC melhor o modelo acerta, ou seja, possui um menor número de falsos-positivos e falsos-negativos.

O gráfico da curva (Figura 3) é feito com a taxa de falsos positivos no eixo x e a taxa de verdadeiros positivos no eixo y.

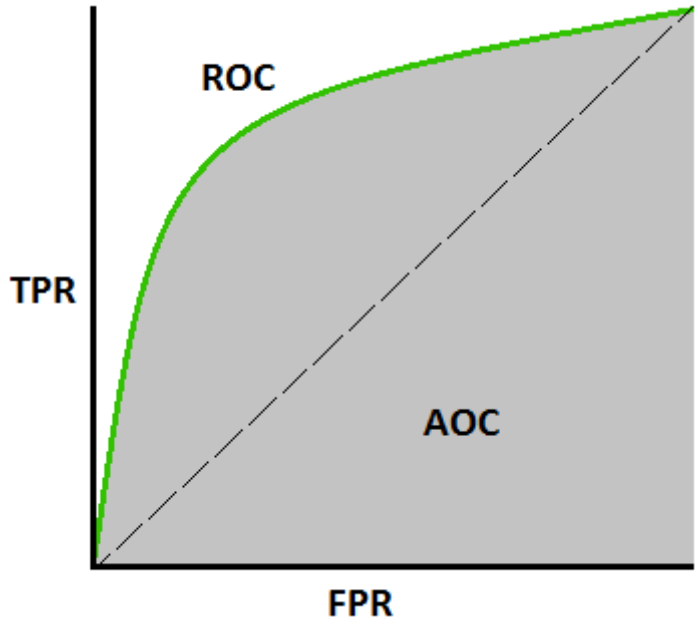

Figura 3 - Representação gráfica da curva ROC e da AOC

Para melhor visualização também foram geradas matrizes de confusão para as amostras classificadas.

4.3
Regressão Logística

AUC-ROC: 0.95 (95%)

	Predição Outros	**Predição Two-factor**
Classe Two-factor	4%	90%
Classe Outros	96%	10%

O modelo utilizando regressão logística obtivemos uma área embaixo da curva ROC de 0.95 indicando um bom desempenho e uma boa distinção das duas classes. Olhando para as taxas de acerto específicas para cada classe temos um acerto menor para a classe *two-factor* do que para a classe outros.

4.4

Random Forest

AUC-ROC: 0.96 (96%)

	Predição Outros	**Predição Two-factor**
Classe Two-factor	6%	99%
Classe Outros	94%	1%

O modelo utilizando *random forest* obtivemos uma área embaixo da curva ROC de 0.96 indicando um bom desempenho e uma boa distinção das duas classes. Em comparação com a regressão logística o modelo de *random forest* teve uma melhora pequena que virtualmente torna os dois modelos semelhantes quanto a resultados gerais.

Olhando para as taxas de acerto específicas para cada classe temos uma melhora importante quanto a classe de *two-factor* [11] e a classificação da classe outros se manteve com desempenho semelhante.

Para o problema proposto é importante que haja uma taxa de acerto maior para predição de *two-factor* pois é melhor que erre menos quanto a classe de interesse.

5
Conclusão e próximos passos

A análise dos textos, em conjunto com a criação de *features* que culminaram em um modelo mostraram que foi possível separar bem as classes de textos que eram de autenticação em dois fatores das que não pertenciam a esse grupo. Na análise das métricas utilizadas para avaliação dos modelos foi possível notar que o modelo *random forest* obteve um desempenho ligeiramente melhor do que o modelo que utilizou regressão logística. Portanto seria o modelo mais indicado para o problema proposto já que possui uma confiabilidade maior quanto a classificação da classe de interesse.

A decisão de utilizar pySpark para a solução também se confirmou uma boa resposta ao problema de processamento de uma grande quantidade de dados sem que o desempenho fosse imensamente afetado.

Para o futuro seria possível explorar outras técnicas como *Bag Of Words[12], word2vec [13], doc2vec [14], glove [15],* novas *features* que possam representar melhor outros subgrupos não contidos na classe de autenticação em dois fatores para que os textos possam ficar mais evidenciados à que classe pertencem.

Referências Bibliográficas

[1] 2017 Data Breach Investigations Report. Verizon. Acesso em 11/08/2019. https://www.ictsecuritymagazine.com/wp-content/uploads/2017-Data-Breach-Investigations-Report.pdf

[2] TWO FACTOR AUTHENTICATIONS FOR SECURED LOGIN IN SUPPORT OF EFFECTIVE INFORMATION PRESERVATION AND NETWORK SECURITY - S. Vaithyasubramanian, A. Christy and D. Saravanan – Acesso em 01/09/2019. http://www.arpnjournals.com/jeas/research_papers/rp_2015/jeas_0315_1713.pdf

[3] One-Time Passwords via SMS – G. Mohsen, G. Satar – Acesso em 01/09/2019 https://pdfs.semanticscholar.org/ccb7/a346d07ff83934d7150e102b6c1457957303.pdf

[4] Two-factor authentication with TOTP – Moretto, Nicola – Acesso em: 01/09/2019 - https://medium.com/@n.moretto/two-factor-authentication-with-totp-ccc5f828b6df

[5] Data Privacy Concerns on Rise: Report – Acesso em 01/09/2019. https://www.chiefmarketer.com/data-privacy-concerns-on-rise-report/

[6] An Introduction to Logistic Regression: From Basic Concepts to Interpretation with Particular Attention to Nursing Domain – Acesso em 03/09/2019. https://pdfs.semanticscholar.org/3305/2b1d2363aee3ad290612109dcea0aed2a89e.pdf

[7] Apache Spark Official Site – Acesso em 03/09/2019. https://spark.apache.org/

[8] "Regular Expressions". The Single UNIX ® Specification, Version 2. - https://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html - Acesso em 12/11/2019

[9] “Data Mining” - Rajaraman, A.; Ullman, J. D. http://i.stanford.edu/~ullman/mmds/ch1.pdf - Acesso em 12/11/2019

[10] Spark Apache Documents – StopWordsRemover - https://spark.apache.org/docs/2.2.0/ml-features.html#stopwordsremover

[11] Two factor authentication using mobile phones - Fadi Aloul; Syed Zahidi; Wassim El-Hajj – 2009

[12] Understanding bag-of-words model: a statistical framework - Yin Zhang; Rong Jin; Zhi-Hua Zhou - 2010

[13] word2vec Parameter Learning Explained - Xin Rong – 2014

[14] Sentiment analysis on Twitter data with semi-supervised Doc2Vec - Metin Bilgin; İzzet Fatih Şentürk - 2017

[15] GloVe: Global Vectors for Word Representation - Jeffrey Pennington; Richard Socher; Christopher D. Manning - https://www.aclweb.org/anthology/D14-1162.pdf - Acesso em 12/11/2019

Apêndice

LowerCaseTransformer

```
class LowerCaseTransformer(Transformer, HasInputCol, HasOutputCol,
                           DefaultParamsReadable, DefaultParamsWritable):
    """ This class will prepare the SMS text to the algorithms, performing
        lowercase transformation
    """

    @keyword_only
    def __init__(self, inputCol='text_message',
outputCol='text_message_clean'):
        super().__init__()
        self._setDefault(inputCol='text_message')
        self._setDefault(outputCol='text_message_clean')

    def setParams(self, inputCol='text_message',
outputCol='text_message_clean'):
        kwargs = self._input_kwargs
        self._set(**kwargs)

    def _transform(self, dataframe):
        # Aplying lowercase
        return dataframe.withColumn(self.getOutputCol(),
F.lower(F.col(self.getInputCol())))
```

RemoveAccentTransformer

```
class RemoveAccentTransformer(Transformer, HasInputCol, HasOutputCol,
                              DefaultParamsReadable,
DefaultParamsWritable):
    """This class will prepare the SMS text to the algorithms, performing
        removal of carcters defined as accent ones
    """

    @keyword_only
    def __init__(self, inputCol='text_message',
outputCol='text_message_clean'):
        super().__init__()
        self._setDefault(inputCol='text_message')
        self._setDefault(outputCol='text_message_clean')

    def setParams(self, inputCol='text_message',
outputCol='text_message_clean'):
        kwargs = self._input_kwargs
        self._set(**kwargs)

    def _transform(self, dataset):
        # defining function to remove accents
        def remove_accents(column):
            accent_replacements = [
                (u'â', 'a'),
                (u'á', 'a'),
```

```
                    (u'à', 'a'),
                    (u'é', 'e'),
                    (u'ê', 'e'),
                    (u'í', 'i'),
                    (u'ó', 'o'),
                    (u'ô', 'o'),
                    (u'ò', 'o'),
                    (u'ç', 'c'),
                    (u'ú', 'u'),
                    (u'ñ', 'n')
                ]
                r = col(column)
                for a, b in accent_replacements:
                    r = regexp_replace (r, a, b)
                return r.alias('remove_accents(' + column + ')')

            # Aplying lowercase
            return dataset.withColumn(self.getOutputCol(),
remove_accents(self.getInputCol()))
```

RemoveSpecialCharactersTransformer

```
class RemoveSpecialCharactersTransformer(Transformer, HasInputCol,
HasOutputCol,
                                         DefaultParamsReadable,
DefaultParamsWritable):
    """ This class will prepare the SMS text to the algorithms, performing
                removal of special characters
    """

    @keyword_only
    def __init__(self, inputCol='text_message',
outputCol='text_message_clean'):
        super().__init__()
        self._setDefault(inputCol='text_message')
        self._setDefault(outputCol='text_message_clean')

    def setParams(self, inputCol='text_message',
outputCol='text_message_clean'):
        kwargs = self._input_kwargs
        self._set(**kwargs)

    def _transform(self, dataframe):
        # Removing special characters and a couple of escape sequences (\n
and \r)
        special_char_regex = r"\\r|\\n|[-
'`~!@Â¤#$%^&*()_|+=?;:'\",.<>\{\}\[\]\\\/]+?"
        return dataframe.withColumn(self.getOutputCol(),
                                    F.regexp_replace(F.col(self.getInputCol(
)),
                                                     special_char_regex,
                                                     ' '))
```

RemoveStopWordsTransformer

```
class RemoveStopWordsTransformer(Transformer, HasInputCol, HasOutputCol,
                                 DefaultParamsReadable,
DefaultParamsWritable):
    """ This class will prepare the SMS text to the algorithms, performing
                removal of stopwords
    """

    @keyword_only
    def __init__(self, inputCol='text_message',
outputCol='text_message_clean'):
        super().__init__()
        self._setDefault(inputCol='text_message')
        self._setDefault(outputCol='text_message_clean')

    def setParams(self, inputCol='text_message',
outputCol='text_message_clean'):
        kwargs = self._input_kwargs
        self._set(**kwargs)

    def _transform(self, dataframe):
        # Tokenizing
        regexTokenizer = RegexTokenizer(inputCol=self.getInputCol(),
                                        outputCol="words_remover",
pattern="\\W")
        dataframe = regexTokenizer.transform(dataframe)

        # Removing stopwords
        stopwords_pt = StopWordsRemover.loadDefaultStopWords('portuguese')
        stopwordsRemover = StopWordsRemover(inputCol="words_remover",
                                            outputCol=self.getOutputCol())\

            .setStopWords(stopwords_pt)
        return stopwordsRemover.transform(dataframe)
```

RemoveDuplicatesTransformer

```
class RemoveDuplicatesTransformer(Transformer):
    """Removes duplicated entries out of a DataFrame based on a reference
column"""

    referenceColumn = Param (Params._dummy(), "referenceColumn",
                             "Column that will be used to remove duplicated
entries",
                             typeConverter=TypeConverters.toString)

    @keyword_only
    def __init__(self, referenceColumn='text_message_clean'):
        super().__init__()
        self._setDefault(referenceColumn='text_message_clean')

        kwargs = self._input_kwargs
        self.setParams(**kwargs)

    def setParams(self, **kwargs):
        self._set(**kwargs)
```

```
•
•       def setReferenceColumn(self, value):
•           return self._set(referenceColumn=value)
•
•       def getReferenceColumn(self):
•           return self.getOrDefault(self.referenceColumn)
•
•       def _transform(self, dataset):
•           dataset =
    dataset.drop_duplicates(subset=[self.getReferenceColumn()])
•           return dataset
```

SMSTokenizer

```
1.  class SMSTokenizer(Transformer, HasInputCol, HasOutputCol, ABC,
2.                     DefaultParamsWritable, DefaultParamsReadable):
3.      """Generic class to replace regex matches with tokens on strings"""
4.
5.      regex = Param(Params._dummy(), "regex",
6.                    ("Regex to capture relevant tokens"),
7.                    typeConverter=TypeConverters.toString)
8.
9.      token = Param(Params._dummy(), "token",
10.                   ("Character to which all regex matches will be converted to."),
11.                   typeConverter=TypeConverters.toString)
12.
13.     @keyword_only
14.     def __init__(self, regex=None, token=None, inputCol=None, outputCol=None):
15.         super().__init__()
16.         self._setDefault(regex=None)
17.         self._setDefault(token=None)
18.         self._setDefault(inputCol=None)
19.         self._setDefault(outputCol=None)
20.         kwargs = self._input_kwargs
21.         self.setParams(**kwargs)
22.
23.     def setParams(self, regex=None, token=None, inputCol=None, outputCol=None):
24.         kwargs = self._input_kwargs
25.         return self._set(**kwargs)
26.
27.     def setToken(self, value):
28.         return self._set(token=value)
29.
30.     def getToken(self):
31.         return self.getOrDefault(self.token)
32.
33.     def setRegex(self, value):
34.         return self._set(regex=value)
35.
36.     def getRegex(self):
37.         return self.getOrDefault(self.regex)
38.
39.     def _transform(self, dataframe):
40.         replacer = F.regexp_replace(self.getInputCol(), self.getRegex(), self.getTo
    ken())
41.         return dataframe.withColumn(self.getOutputCol(), replacer)
```

```
1.  class DigitTokenizer(SMSTokenizer):
2.      """Replaces all digits with a selected character (default: '9')"""
3.
```

```
    @keyword_only
    def __init__(self, regex=DIGIT_REGEX, token=DIGIT_TOKEN,
                 inputCol='text_message_clean', outputCol='text_message_clean'):
        super().__init__(regex=regex, token=token, inputCol=inputCol, outputCol=out
putCol)
```

RegexMaskFeatureTransformer

```
class RegexMaskFeatureTransformer(Transformer, HasInputCol, HasOutputCol, ABC,
                                  DefaultParamsWritable, DefaultParamsReadable):
    """ This class will create columns based on regex of first half of Digits
    """

    regex = Param(Params._dummy(), "regex",
                  ("Regex which the binary column representing the match of it will
 be created"),
                  typeConverter=TypeConverters.toString)

    @keyword_only
    def __init__(self, regex=None, inputCol=None, outputCol=None):
        super().__init__()
        self._setDefault(inputCol=None)
        self._setDefault(outputCol=None)
        self._setDefault(regex=None)
        kwargs = self._input_kwargs
        self.setParams(**kwargs)

    def setParams(self, regex=None, inputCol=None, outputCol=None):
        kwargs = self._input_kwargs
        self._set(**kwargs)

    def setRegex(self, value):
        return self._set(regex=value)

    def getRegex(self):
        return self.getOrDefault(self.regex)

    def _transform(self, dataframe):
        return dataframe.withColumn(self.getOutputCol(),
                                    dataframe[self.getInputCol()]

                                   .rlike(self.getRegex()))
```

HashingFeaturesTransformer

```
class HashingFeaturesTransformer(Transformer, HasInputCol,
                                 DefaultParamsWritable, DefaultParamsReadable):
    """
    """
    numFeatures = Param(Params._dummy(), "numFeatures",
                        ("Number of column features to be created with TF-
IDF Hashing"),
                        typeConverter=TypeConverters.toInt)

    booleanHashing = Param(Params._dummy(), "booleanHashing",
                           ("Boolean to decide weather to convert values of hashing
 to binary or not"),
                           typeConverter=TypeConverters.toBoolean)

    @keyword_only
    def __init__(self, numFeatures=100, booleanHashing=False, inputCol='text_messag
e_clean'):
        super().__init__()
```

```
        self._setDefault(inputCol='text_message_clean')
        self._setDefault(numFeatures=100)
        self._setDefault(booleanHashing=False)
        kwargs = self._input_kwargs
        self.setParams(**kwargs)

    def setParams(self, numFeatures=100, booleanHashing=False, inputCol=None):
        kwargs = self._input_kwargs
        self._set(**kwargs)

    def setNumFeatures(self, value):
        return self._set(numFeatures=value)

    def getNumFeatures(self):
        return self.getOrDefault(self.numFeatures)

    def setBooleanHashing(self, value):
        return self._set(booleanHashing=value)

    def getBooleanHashing(self):
        return self.getOrDefault(self.booleanHashing)

    def _transform(self, dataframe):
        # create columns for TF-IDF
        hashingTF = HashingTF(inputCol=self.getInputCol(),
                              outputCol="hashing_features",
                              numFeatures=self.getNumFeatures(),
                              binary=self.getBooleanHashing())

        featurized_dataset = hashingTF.transform(dataframe)

        return featurized_dataset
```

ContainsWordFeatureTransformer

```
class ContainsWordFeatureTransformer(Transformer, HasInputCol, HasOutputCol, ABC,
                                     DefaultParamsWritable, DefaultParamsReadable):

    """Starting from a column containing an raay of word tokens it creates a new
    binary column telling if defined word is contained at least one time in the inp
ut
    """
    word = Param(Params._dummy(), "word",
                 ("Word which the binary column representing the existance of word
will be created"),
                 typeConverter=TypeConverters.toString)

    @keyword_only
    def __init__(self, word=None, inputCol=None, outputCol=None):
        super().__init__()
        self._setDefault(inputCol=None)
        self._setDefault(outputCol=None)
        self._setDefault(word=None)
        kwargs = self._input_kwargs
        self.setParams(**kwargs)

    def setParams(self, word=None, inputCol=None, outputCol=None):
        kwargs = self._input_kwargs
        self._set(**kwargs)

    def setWord(self, value):
        return self._set(word=value)

```

```
    def getWord(self):
        return self.getOrDefault(self.word)

    def _transform(self, dataframe):
        # create column of keyword
        return dataframe.withColumn(self.getOutputCol(),
                                    F.array_contains(dataframe[self.getInputCol()],
                                                     self.getWord()))
```

```
class HasCodigoFeatureTransformer(ContainsWordFeatureTransformer):
    """Creates column has_codigo if input array contains word 'codigo' """

    @keyword_only
    def __init__(self, word='codigo', inputCol='tokens', outputCol='has_codigo'):
        super().__init__(word=word, inputCol=inputCol, outputCol=outputCol)
```

SMSPreprocess

```
class SMSPreprocess(Pipeline):
    def __init__(self):
        super().__init__()

        self.lowercase_tr = transformers.LowerCaseTransformer(inputCol="text_message",
                                                              outputCol="text_message_clean")

        self.removeAccent_tr = transformers.RemoveAccentTransformer(inputCol="text_message_clean",
                                                                    outputCol="text_message_clean")

        self.removeSpecial_tr = transformers.RemoveSpecialCharactersTransformer(inputCol="text_message_clean",
                                                                                out
putCol="text_message_clean")

        self.removeStop_tr = transformers.RemoveStopWordsTransformer(inputCol="text_message_clean",

                                                                     outputCol="tokens")
```

SMSTokenization

```
class SMSTokenization(SMSPreprocess):
    def __init__(self):
        super().__init__()

        self.DigitTokenizer_tr = tokenizers.DigitTokenizer()
        self.DateTokenizer_tr = tokenizers.DateTokenizer()
        self.TimeTokenizer_tr = tokenizers.TimeTokenizer()
        self.MoneyTokenizer_tr = tokenizers.MoneyTokenizer()

        self.removeSpecial_tr.setParams(inputCol="text_message_clean",
                                        outputCol="text_message_clean")
        self.DigitTokenizer_tr.setParams(
            inputCol="text_message_clean", outputCol="text_message_clean")

```

```
        self.removeStop_tr.setParams(inputCol="text_message_clean", outputCol="tokens")

        self.setStages(value=[self.lowercase_tr,
                              self.removeAccent_tr,
                              self.DateTokenizer_tr,
                              self.TimeTokenizer_tr,
                              self.MoneyTokenizer_tr,
                              self.removeSpecial_tr,
                              self.DigitTokenizer_tr,
                              self.removeStop_tr
                              ])
```

FeaturePipeline

```
class FeaturePipeline(Pipeline):

    def __init__(self, booleanHashing):
        super().__init__()

        self.DigitFirstFeature = features.DigitsFirstMaskFeatureTransformer(
            inputCol='text_message_clean'
        )

        self.DigitSecondFeature = features.DigitsSecondMaskFeatureTransformer(
            inputCol='text_message_clean'
        )

        self.HashingFeature = features.HashingFeaturesTransformer(
            inputCol='tokens',
            numFeatures=100,
            booleanHashing=booleanHashing
        )

        self.HasCodigoFeature = features.HasCodigoFeatureTransformer(
            inputCol='tokens'
        )

        self.HasCodeFeature = features.HasCodeFeatureTransformer(
            inputCol='tokens'
        )

        self.HasVerificacaoFeature = features.HasVerificacaoFeatureTransformer(
            inputCol='tokens'
        )

        self.HasCompartilheFeature = features.HasCompartilheFeatureTransformer(
            inputCol='tokens'
        )

        self.HasDateFeature = features.HasDateFeatureTransformer(
            inputCol='tokens'
        )

        self.HasTimeFeature = features.HasTimeFeatureTransformer(
            inputCol='tokens'
        )

        self.HasMoneyFeature = features.HasMoneyFeatureTransformer(
            inputCol='tokens'
        )
        self.HasUrlFeature = features.HasUrlFeatureTransformer(
            inputCol='tokens'
        )

```

```
        feature_cols = ['has_3_digit_1st', 'has_3_digit_2nd', 'has_codigo',
                        'has_code', 'has_verificacao', 'has_compartilhe',
                        'has_date', 'has_time', 'has_money', 'has_url'
                        ]
        numFeatures = 100
        for i in range(numFeatures):
            feature_cols.append('hashing_posicao_' + str(i))

        self.VectorAssembler = VectorAssembler(inputCols=feature_cols,
                                               outputCol="features")

        self.setStages(value=[self.DigitFirstFeature,
                              self.DigitSecondFeature,
                              self.HasCodigoFeature,
                              self.HasCodeFeature,
                              self.HasVerificacaoFeature,
                              self.HasCompartilheFeature,
                              self.HasDateFeature,
                              self.HasTimeFeature,
                              self.HasMoneyFeature,
                              self.HasUrlFeature,
                              self.HashingFeature,
                              self.VectorAssembler
                              ])
```