```
******************
**CLIP originalus**
******************

import os
import cv2
import
timm
import torch
import itertools
import numpy as np
import pandas as pd
import torch.nn as
nn
import albumentations as A
import matplotlib.pyplot as plt
from tqdm.autonotebook import
tqdm
from transformers import DistilBertModel, DistilBertConfig, DistilBertTokenizer
import
torch
from torch import nn
import torch.nn.functional as F

class CFG:
    debug = False

image_path = "/content/drive/MyDrive/BACHELOR'S DATA/librosa-images"

captions_path = "."
    batch_size = 32
    num_workers = 4
    head_lr = 1e-3

image_encoder_lr = 1e-3
    text_encoder_lr = 1e-3
    weight_decay = 1e-3
    patience = 1

factor = 0.8
    epochs = 100
    device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")

    model_name = 'resnet50'

image_embedding = 2048
    text_encoder_model = "distilbert-base-uncased"

text_embedding = 768
    text_tokenizer = "distilbert-base-uncased"
    max_length =
200

    pretrained = True # for both image encoder and text encoder
    trainable = True # for
both image encoder and text encoder
    temperature = 1.0

    # image size
    size = 224


# for projection head; used for both image and text encoders
    num_projection_layers = 1

projection_dim = 256
    dropout = 0.1

class AvgMeter:
    def __init__(self,
name="Metric"):
        self.name = name
        self.reset()
```

```
    def reset(self):

      self.avg, self.sum, self.count = [0] * 3

    def update(self, val, count=1):

self.count += count
        self.sum += val * count
        self.avg = self.sum / self.count


  def __repr__(self):
        text = f"{self.name}: {self.avg:.4f}"
        return
text

def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return
param_group["lr"]


class CLIPDataset(torch.utils.data.Dataset):
    def
__init__(self, image_filenames, captions, tokenizer, transforms):
        """

     image_filenames and cpations must have the same length; so, if there are
        multiple
captions for each image, the image_filenames must have repetitive
        file names

"""

        self.image_filenames = image_filenames
        self.captions =
list(captions)
        self.encoded_captions = tokenizer(
            list(captions),
padding=True, truncation=True, max_length=CFG.max_length
        )
        self.transforms =
transforms

    def __getitem__(self, idx):
        item = {
            key:
torch.tensor(values[idx])
            for key, values in self.encoded_captions.items()

}

        image = cv2.imread(f"{CFG.image_path}/{self.image_filenames[idx]}")

 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image =
self.transforms(image=image)['image']
        item['image'] = torch.tensor(image).permute(2, 0,
1).float()
        item['caption'] = self.captions[idx]

        return item


    def
__len__(self):
        return len(self.captions)



def
get_transforms(mode="train"):
    if mode == "train":
        return
```

```
A.Compose(
            [
                A.Resize(CFG.size, CFG.size, always_apply=True),

          A.Normalize(max_pixel_value=255.0, always_apply=True),
            ]
        )

else:
        return A.Compose(
            [
                A.Resize(CFG.size, CFG.size,
always_apply=True),
                A.Normalize(max_pixel_value=255.0, always_apply=True),

       ]
        )

class ImageEncoder(nn.Module):
    """
    Encode
images to a fixed size vector
    """

    def __init__(
        self,
model_name=CFG.model_name, pretrained=CFG.pretrained, trainable=CFG.trainable
    ):

super().__init__()
        self.model = timm.create_model(
            model_name, pretrained,
num_classes=0, global_pool="avg"
        )
        for p in self.model.parameters():

           p.requires_grad = trainable

    def forward(self, x):
        return self.model(x)


class TextEncoder(nn.Module):
    def __init__(self, model_name=CFG.text_encoder_model,
pretrained=CFG.pretrained, trainable=CFG.trainable):
        super().__init__()
        if
pretrained:
            self.model = DistilBertModel.from_pretrained(model_name)
        else:

           self.model = DistilBertModel(config=DistilBertConfig())

        for p
in self.model.parameters():
            p.requires_grad = trainable

        # we are using the
CLS token hidden representation as the sentence's embedding
        self.target_token_idx = 0


   def forward(self, input_ids, attention_mask):
        output =
self.model(input_ids=input_ids, attention_mask=attention_mask)
        last_hidden_state =
output.last_hidden_state
        return last_hidden_state[:, self.target_token_idx, :]


class ProjectionHead(nn.Module):
    def __init__(
        self,
        embedding_dim,
```

```
 projection_dim=CFG.projection_dim,
        dropout=CFG.dropout
    ):

super().__init__()
        self.projection = nn.Linear(embedding_dim, projection_dim)

self.gelu = nn.GELU()
        self.fc = nn.Linear(projection_dim, projection_dim)

self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(projection_dim)


   def forward(self, x):
        projected = self.projection(x)
        x =
self.gelu(projected)
        x = self.fc(x)
        x = self.dropout(x)
        x = x +
projected
        x = self.layer_norm(x)
        return x


class CLIPModel(nn.Module):

def __init__(
        self,
        temperature=CFG.temperature,

image_embedding=CFG.image_embedding,
        text_embedding=CFG.text_embedding,
    ):

super().__init__()
        self.image_encoder = ImageEncoder()
        self.text_encoder =
TextEncoder()
        self.image_projection = ProjectionHead(embedding_dim=image_embedding)

    self.text_projection = ProjectionHead(embedding_dim=text_embedding)

self.temperature = temperature

    def forward(self, batch):
        # Getting Image and Text
Features
        image_features = self.image_encoder(batch["image"])

text_features = self.text_encoder(
            input_ids=batch["input_ids"],
attention_mask=batch["attention_mask"]
        )
        # Getting Image and Text
Embeddings (with same dimension)
        image_embeddings =
self.image_projection(image_features)
        text_embeddings =
self.text_projection(text_features)

        # Calculating the Loss
        logits =
(text_embeddings @ image_embeddings.T) / self.temperature
        images_similarity =
image_embeddings @ image_embeddings.T
        texts_similarity = text_embeddings @
text_embeddings.T
        targets = F.softmax(
            (images_similarity +
texts_similarity) / 2 * self.temperature, dim=-1
        )
        texts_loss =
cross_entropy(logits, targets, reduction='none')
```

```
        images_loss = cross_entropy(logits.T,
targets.T, reduction='none')
        loss =  (images_loss + texts_loss) / 2.0 # shape:
(batch_size)
        return loss.mean()


def cross_entropy(preds, targets, reduction='none'):

   log_softmax = nn.LogSoftmax(dim=-1)
    loss = (-targets * log_softmax(preds)).sum(1)
    if
reduction == "none":
        return loss
    elif reduction == "mean":

   return loss.mean()

def make_train_valid_dfs():
    dataframe =
pd.read_csv(f"/content/drive/MyDrive/BACHELOR'S DATA/captions.csv")
    max_id =
dataframe["id"].max() + 1 if not CFG.debug else 100
    image_ids = np.arange(0,
max_id)
    np.random.seed(42)
    valid_ids = np.random.choice(
        image_ids,
size=int(0.2 * len(image_ids)), replace=False
    )
    train_ids = [id_ for id_ in image_ids
if id_ not in valid_ids]
    train_dataframe =
dataframe[dataframe["id"].isin(train_ids)].reset_index(drop=True)
    valid_dataframe
= dataframe[dataframe["id"].isin(valid_ids)].reset_index(drop=True)
    return
train_dataframe, valid_dataframe


def build_loaders(dataframe, tokenizer, mode):

transforms = get_transforms(mode=mode)
    dataset = CLIPDataset(

dataframe["image"].values,
        dataframe["caption"].values,

tokenizer=tokenizer,
        transforms=transforms,
    )
    dataloader =
torch.utils.data.DataLoader(
        dataset,
        batch_size=CFG.batch_size,

num_workers=CFG.num_workers,
        shuffle=True if mode == "train" else False,

)
    return dataloader

def train_epoch(model, train_loader, optimizer, lr_scheduler, step):

  loss_meter = AvgMeter()
    tqdm_object = tqdm(train_loader, total=len(train_loader))
    for
batch in tqdm_object:
        batch = {k: v.to(CFG.device) for k, v in batch.items() if k !=
"caption"}
        loss = model(batch)
        optimizer.zero_grad()

loss.backward()
        optimizer.step()
```

```
        if step == "batch":

lr_scheduler.step()

        count = batch["image"].size(0)

loss_meter.update(loss.item(), count)


tqdm_object.set_postfix(train_loss=loss_meter.avg, lr=get_lr(optimizer))
    return
loss_meter


def valid_epoch(model, valid_loader):
    loss_meter = AvgMeter()

    tqdm_object
= tqdm(valid_loader, total=len(valid_loader))
    for batch in tqdm_object:
        batch = {k:
v.to(CFG.device) for k, v in batch.items() if k != "caption"}
        loss =
model(batch)

        count = batch["image"].size(0)

loss_meter.update(loss.item(), count)


tqdm_object.set_postfix(valid_loss=loss_meter.avg)
    return loss_meter


def main():

train_df, valid_df = make_train_valid_dfs()
    tokenizer =
DistilBertTokenizer.from_pretrained(CFG.text_tokenizer)
    train_loader =
build_loaders(train_df, tokenizer, mode="train")
    valid_loader =
build_loaders(valid_df, tokenizer, mode="valid")


    model =
CLIPModel().to(CFG.device)
    params = [
        {"params":
model.image_encoder.parameters(), "lr": CFG.image_encoder_lr},

{"params": model.text_encoder.parameters(), "lr": CFG.text_encoder_lr},

    {"params": itertools.chain(
            model.image_projection.parameters(),
model.text_projection.parameters()
        ), "lr": CFG.head_lr,
"weight_decay": CFG.weight_decay}
    ]
    optimizer = torch.optim.AdamW(params,
weight_decay=0.)
    lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(

optimizer, mode="min", patience=CFG.patience, factor=CFG.factor
    )
    step =
"epoch"

    best_loss = float('inf')
    for epoch in range(CFG.epochs):

print(f"Epoch: {epoch + 1}")
        model.train()
        train_loss =
train_epoch(model, train_loader, optimizer, lr_scheduler, step)
```

```
        model.eval()

with torch.no_grad():
            valid_loss = valid_epoch(model, valid_loader)


 if valid_loss.avg < best_loss:
            best_loss = valid_loss.avg

torch.save(model.state_dict(), "best.pt")
            print("Saved Best
Model!")

        lr_scheduler.step(valid_loss.avg)



**********
**CLIP modifikuotas**
**********

import numpy as np
import
pandas as pd
import itertools
from tqdm.autonotebook import tqdm
import matplotlib.pyplot as
plt

import torch
from torch import nn
import torch.nn.functional as F
from transformers import
AutoModel, AutoTokenizer, AutoConfig

TSCOLS = 256
TRAIN_DATA =
"train_val_data/BTC_train.csv"

class CFG:
    debug = False
    batch_size = 32

num_workers = 0
    head_lr = 1e-3
    timeseries_encoder_lr = 1e-4
    text_encoder_lr = 1e-5

   weight_decay = 1e-3
    patience = 1
    factor = 0.8
    epochs = 20
    device =
torch.device("cuda" if torch.cuda.is_available() else "cpu")


model_name = 'resnet50'
    text_encoder_model = "distilbert-base-uncased"

text_embedding = 768
    text_tokenizer = "distilbert-base-uncased"

timeseries_embedding = 256
    max_length = 200

    pretrained = True # for  text encoder

trainable = True # for text encoder
    temperature = 1.0

    # image size
    # size = 224


  # for projection head; used for both image and text encoders
```

```
    num_projection_layers = 1

 projection_dim = 256
    dropout = 0.1

class AvgMeter:
    def __init__(self,
name="Metric"):
        self.name = name
        self.reset()

    def reset(self):

      self.avg, self.sum, self.count = [0] * 3

    def update(self, val, count=1):

self.count += count
        self.sum += val * count
        self.avg = self.sum / self.count


  def __repr__(self):
        text = f"{self.name}: {self.avg:.4f}"
        return
text

def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return
param_group["lr"]


class CLIPDataset(torch.utils.data.Dataset):
    def
__init__(self, timeseries, captions, tokenizer):
        self.timeseries = list(timeseries)

    self.captions = list(captions)
        self.encoded_captions = tokenizer(

list(captions), padding=True, truncation=True, max_length=CFG.max_length
        )

self.normalizer = 'none'

    def __getitem__(self, idx):
        item = {
            key:
torch.tensor(values[idx])
            for key, values in self.encoded_captions.items()

}

        if self.normalizer =='none':
            item['timeseries'] =
torch.tensor(self.timeseries[idx]).float()
        if self.normalizer =='minmax':

item['timeseries'] = (torch.tensor(self.timeseries[idx]).float() -
torch.tensor(self.timeseries[idx]).float().min()) /
(torch.tensor(self.timeseries[idx]).float().max() -
torch.tensor(self.timeseries[idx]).float().min())
        if self.normalizer =='meansd':

     item['timeseries'] = (torch.tensor(self.timeseries[idx]).float() -
torch.tensor(self.timeseries[idx]).float().mean()) /
torch.tensor(self.timeseries[idx]).float().std()

        item['caption'] =
self.captions[idx]

        return item
```

```
    def __len__(self):
        return
len(self.captions)

class TextEncoder(nn.Module):
    def __init__(self,
model_name=CFG.text_encoder_model, pretrained=CFG.pretrained, trainable=CFG.trainable):

super().__init__()
        if pretrained:
            self.model =
AutoModel.from_pretrained(model_name)
        else:
            self.model =
AutoModel(config=AutoConfig())

        for p in self.model.parameters():

  p.requires_grad = False

        # we are using the CLS token hidden representation as the
sentence's embedding
        self.target_token_idx = 0

    def forward(self, input_ids,
attention_mask):
        output = self.model(input_ids=input_ids,
attention_mask=attention_mask)
        last_hidden_state = output.last_hidden_state

return last_hidden_state[:, self.target_token_idx, :]

class ProjectionHead(nn.Module):

def __init__(
        self,
        embedding_dim,
        projection_dim=CFG.projection_dim,

      dropout=CFG.dropout
    ):
        super().__init__()
        self.projection =
nn.Linear(embedding_dim, projection_dim)
        self.gelu = nn.GELU()
        self.fc =
nn.Linear(projection_dim, projection_dim)
        self.dropout = nn.Dropout(dropout)

self.layer_norm = nn.LayerNorm(projection_dim)

    def forward(self, x):
        projected
= self.projection(x)
        x = self.gelu(projected)
        x = self.fc(x)
        x =
self.dropout(x)
        x = x + projected
        x = self.layer_norm(x)
        return x


class CLIPModel(nn.Module):
    def __init__(
        self,

temperature=CFG.temperature,
        text_embedding=CFG.text_embedding,

timeseries_embedding=CFG.timeseries_embedding,
        use_timeseries_projection_head=True,

):
        super().__init__()
```

```
        self.text_encoder = TextEncoder()

self.text_projection = ProjectionHead(embedding_dim=text_embedding)
        self.ts_projection
= ProjectionHead(embedding_dim=timeseries_embedding)
        self.temperature = temperature

    self.use_timeseries_projection_head = use_timeseries_projection_head

    def forward(self,
batch):
        timeseries_features = batch["timeseries"]
        text_features =
self.text_encoder(
            input_ids=batch["input_ids"],
attention_mask=batch["attention_mask"]
        )

        if
self.use_timeseries_projection_head:
            timeseries_embeddings =
self.ts_projection(timeseries_features)
        else:
            timeseries_embeddings =
timeseries_features

        text_embeddings =
self.text_projection(text_features)

        # Calculating the Loss
        logits =
(text_embeddings @ timeseries_embeddings.T) / self.temperature
        timeseries_similarity =
timeseries_embeddings @ timeseries_embeddings.T
        texts_similarity = text_embeddings @
text_embeddings.T
        targets = F.softmax(
            (timeseries_similarity +
texts_similarity) / 2 * self.temperature, dim=-1
        )
        texts_loss =
cross_entropy(logits, targets, reduction='none')
        images_loss = cross_entropy(logits.T,
targets.T, reduction='none')
        loss =  (images_loss + texts_loss) / 2.0 # shape:
(batch_size)
        return loss.mean()


def cross_entropy(preds, targets, reduction='none'):

   log_softmax = nn.LogSoftmax(dim=-1)
    loss = (-targets * log_softmax(preds)).sum(1)
    if
reduction == "none":
        return loss
    elif reduction == "mean":

   return loss.mean()

def make_train_valid_dfs(TRAIN_DATA_PATH):
    dataframe =
pd.read_csv(TRAIN_DATA_PATH, lineterminator='\n')
    max_id = dataframe.shape[0]

timeseries_ids = np.arange(0, max_id)
    np.random.seed(42)
    valid_ids = np.random.choice(

       timeseries_ids, size=int(0.2 * len(timeseries_ids)), replace=False
    )
    train_ids =
[id_ for id_ in timeseries_ids if id_ not in valid_ids]
    train_dataframe =
dataframe[dataframe.index.isin(train_ids)].reset_index(drop=True)
```

```
    valid_dataframe =
dataframe[dataframe.index.isin(valid_ids)].reset_index(drop=True)
    return train_dataframe,
valid_dataframe


def build_loaders(dataframe, tokenizer, mode):
    dataset = CLIPDataset(

    dataframe.iloc[:, -TSCOLS:].values,
        dataframe.iloc[:, 0].values,

tokenizer=tokenizer,
    )
    dataloader = torch.utils.data.DataLoader(
        dataset,

  batch_size=CFG.batch_size,
        num_workers=CFG.num_workers,
        shuffle=True if mode
== "train" else False,
    )
    return dataloader


def train_epoch(model,
train_loader, optimizer, lr_scheduler, step):
    loss_meter = AvgMeter()
    tqdm_object =
tqdm(train_loader, total=len(train_loader))
    for batch in tqdm_object:
        batch = {k:
v.to(CFG.device) for k, v in batch.items() if k != "caption"}
        loss =
model(batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if step == "batch":
            lr_scheduler.step()

        count =
batch["timeseries"].size(0)
        loss_meter.update(loss.item(), count)


tqdm_object.set_postfix(train_loss=loss_meter.avg, lr=get_lr(optimizer))
    return
loss_meter


def valid_epoch(model, valid_loader):
    loss_meter = AvgMeter()

    tqdm_object
= tqdm(valid_loader, total=len(valid_loader))
    for batch in tqdm_object:
        batch = {k:
v.to(CFG.device) for k, v in batch.items() if k != "caption"}
        loss =
model(batch)

        count = batch["timeseries"].size(0)

loss_meter.update(loss.item(), count)


tqdm_object.set_postfix(valid_loss=loss_meter.avg)
    return loss_meter

def
get_ts_embeddings(valid_df, model_path):
    tokenizer =
AutoTokenizer.from_pretrained(CFG.text_tokenizer)
```

```
    valid_loader = build_loaders(valid_df,
tokenizer, mode="valid")

    model = CLIPModel()

model.load_state_dict(torch.load(model_path) if CFG.device == "cuda" else
torch.load(model_path, map_location=torch.device('cpu')))
    model.eval()


valid_timeseries_embeddings = []
    with torch.no_grad():
        for batch in
tqdm(valid_loader):

valid_timeseries_embeddings.append(batch["timeseries"])
    return model,
torch.cat(valid_timeseries_embeddings)

def find_matches(model, timeseries_embeddings, query,
n=9):
    tokenizer = AutoTokenizer.from_pretrained(CFG.text_tokenizer)
    encoded_query =
tokenizer([query])
    batch = {
        key: torch.tensor(values)
        for key, values in
encoded_query.items()
    }
    with torch.no_grad():
        text_features =
model.text_encoder(
            input_ids=batch["input_ids"],
attention_mask=batch["attention_mask"]
        )
        text_embeddings =
model.text_projection(text_features)

    timeseries_embeddings_n =
F.normalize(timeseries_embeddings, p=2, dim=-1)
    text_embeddings_n =
F.normalize(text_embeddings, p=2, dim=-1)
    dot_similarity = text_embeddings_n @
timeseries_embeddings_n.T

    values, indices = torch.topk(dot_similarity.squeeze(0), n *
5)
    matches =  indices[::5]

    return matches

def main(TRAIN_DATA_PATH, WEIGHTS_PATH):

 train_df, valid_df = make_train_valid_dfs(TRAIN_DATA_PATH)
    tokenizer =
AutoTokenizer.from_pretrained(CFG.text_tokenizer)
    train_loader = build_loaders(train_df,
tokenizer, mode="train")
    valid_loader = build_loaders(valid_df, tokenizer,
mode="valid")


    model = CLIPModel().to(CFG.device)
    params = [

{"params": {}, "lr": CFG.timeseries_encoder_lr},

{"params": model.text_encoder.parameters(), "lr": CFG.text_encoder_lr},

    {"params": itertools.chain({}, model.text_projection.parameters()
        ),
"lr": CFG.head_lr, "weight_decay": CFG.weight_decay}
    ]
    optimizer =
torch.optim.AdamW(params, weight_decay=0.)
```

```
    lr_scheduler =
torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode="min",
patience=CFG.patience, factor=CFG.factor
    )
    step = "epoch"

    best_loss =
float('inf')
    for epoch in range(CFG.epochs):
        print(f"Epoch: {epoch +
1}")
        model.train()
        train_loss = train_epoch(model, train_loader,
optimizer, lr_scheduler, step)
        model.eval()
        with torch.no_grad():

valid_loss = valid_epoch(model, valid_loader)

        if valid_loss.avg <
best_loss:
            best_loss = valid_loss.avg
            torch.save(model.state_dict(),
WEIGHTS_PATH)
            print("Saved Best Model!")


lr_scheduler.step(valid_loss.avg)


**********************
**Duomenu
griauzimas**
**********************

import twint
import pandas as pd
import datetime
import
sys

def count_lines(filename):
    with open(filename) as fp:
        count = 0
        for _
in fp:
            count += 1
    return count

TICKER = sys.argv[1]

c =
twint.Config()
c.Search = '$' + TICKER
c.Hide_output = True
c.Store_csv = True
c.Custom_csv =
["language", "created_at", "tweet",
"username"]

start_date = datetime.date(int(sys.argv[2]), int(sys.argv[3]),
int(sys.argv[4]))
end_date   = datetime.date(2023, 2, 19)

date_generated = [ str(start_date +
datetime.timedelta(n)) for n in range(int ((end_date - start_date).days))]

for i in
range(len(date_generated) - 1):
    c.Since = date_generated[i]
    c.Until =
date_generated[i+1]
    filename = "scraped_data/" + TICKER + date_generated[i] +
".csv"
```

```
    c.Output =  filename
    twint.run.Search(c)
    tlist =
c.search_tweet_list
    print(len(tlist))
    print(date_generated[i], count_lines(filename))


************************
**Parametru vertinimas**
************************

from
CLIP_timeseries import main
import sys

main(sys.argv[1],
sys.argv[2])

************************
**HPC paleidimo
kodas**
************************

#!/bin/sh
#SBATCH -p gpu
#SBATCH -n1
#SBATCH --gres
gpu
#SBATCH -t 300
. gpu_env/bin/activate
python3 train.py $1
$2


***********************************
**Kriptovaliutu duomenu
paruosimas**
***********************************

#!/usr/bin/env python
# coding:
utf-8

import pandas as pd

IN_DATA_FOLDER = "raw_data/prices/"
OUT_DATA_FOLDER =
"clean_data/prices/"

def read_from_cryptodatadownload(filename):
    # This
particular site returns data with 1 row of their link to website
    # Also the time series is
in reverse so need to do adjustments
    cols = ["Date", "Open",
"High", "Low", "Close", "Volume USDT"]
    df =
pd.read_csv(filename, skiprows = 1, parse_dates = ['Date'])[::-1].reset_index(drop = True)

df = df[cols]
    df = df.rename({'Volume USDT' : 'Volume'}, axis = 1)
    return df

# ##
Day

read_from_cryptodatadownload(IN_DATA_FOLDER +
"Binance_BTCUSDT_d.csv").to_csv(OUT_DATA_FOLDER + "BTCUSDT_day.csv", index
= False)
read_from_cryptodatadownload(IN_DATA_FOLDER +
"Binance_ETHUSDT_d.csv").to_csv(OUT_DATA_FOLDER + "ETHUSDT_day.csv", index
= False)
read_from_cryptodatadownload(IN_DATA_FOLDER +
"Binance_XRPUSDT_d.csv").to_csv(OUT_DATA_FOLDER + "XRPUSDT_day.csv", index
```

```
= False)

# ## Hour

read_from_cryptodatadownload(IN_DATA_FOLDER +
"Binance_BTCUSDT_1h.csv").to_csv(OUT_DATA_FOLDER + "BTCUSDT_hour.csv",
index = False)
read_from_cryptodatadownload(IN_DATA_FOLDER +
"Binance_ETHUSDT_1h.csv").to_csv(OUT_DATA_FOLDER + "ETHUSDT_hour.csv",
index = False)
read_from_cryptodatadownload(IN_DATA_FOLDER +
"Binance_XRPUSDT_1h.csv").to_csv(OUT_DATA_FOLDER + "XRPUSDT_hour.csv",
index = False)

# ## Minute

def read_minute_from_cryptodatadownload(ticker, year):
    # This
particular site returns data with 1 row of their link to website
    # Also the time series is
in reverse so need to do adjustments

    filename = "Binance_" + ticker +
"USDT_" + year + "_minute.csv"
    cols = ['date', 'open', 'high', 'low',
'close', "Volume USDT"]
    df = pd.read_csv(IN_DATA_FOLDER + filename, skiprows = 1,
parse_dates = ['date'])[::-1].reset_index(drop = True)
    df = df[cols]
    df = df.rename({

      "Volume USDT" : 'Volume',
        'date' : 'Date',
        'open' : 'Open',

    'high' : 'High',
        'low' : 'Low',
        'close' : 'Close',
    }, axis = 1)

return df

for ticker in ["BTC", "ETH", "XRP"]:
    df =
read_minute_from_cryptodatadownload(ticker, "2021")
    df.to_csv(OUT_DATA_FOLDER +
ticker + "USDT_minute.csv", index = False)


*****************************
**Twitter duomenu
paruosimas**
*****************************

#!/usr/bin/env python
# coding: utf-8

import
pandas as pd
from tqdm import tqdm
import os

IN_DATA_FOLDER =
'raw_data/tweets/'
OUT_DATA_FOLDER = 'clean_data/tweets/'
TICKERS = ["BTC",
"ETH", "XRP"]

def clean_df(data):
    # My collected data about tweets
contains a lot of unusable information
    # So removing that plus some sanity check filtering

   df = data.copy()
    df = df[df.language == 'en']
```

```
    df["timestamp"] =
pd.to_datetime(df.date + " " + df.time)
    df = df[['tweet', 'username',
'timestamp']]
    df = df.dropna()
    df = df.reset_index(drop=True)
    return df


for
ticker in TICKERS:
    files = [f for f in os.listdir(IN_DATA_FOLDER) if f.startswith(ticker)]

   dfs = []
    for f in tqdm(files):
        temp = pd.read_csv(IN_DATA_FOLDER + f,
engine='python')[['language', 'date', 'time','tweet', 'username']]
        dfs.append(temp)

df = pd.concat(dfs)
    df = clean_df(df)
    df.to_csv(OUT_DATA_FOLDER + ticker +
"_tweets.csv", index=False)

**********
**Paveiksleliu
kurimas**
**********

#!/usr/bin/env python
# coding: utf-8

import pandas as
pd
import librosa
import librosa.display
import numpy as np
import matplotlib.pyplot as
plt
from tqdm import tqdm

BTC_OUT_DATA_FOLDER =
"librosa-images/"
ETH_OUT_DATA_FOLDER =
"eth-librosa-images/"
XRP_OUT_DATA_FOLDER = "xrp-librosa-images/"

def
visualize_series(timeseries, image_name, out_folder):
    chroma =
librosa.feature.chroma_stft(S=np.abs(librosa.stft(timeseries, n_fft=256)), sr=4000)
    fig =
plt.figure(frameon=False)
    fig.set_size_inches(1, 1)
    img =
librosa.display.specshow(chroma)
    fig.savefig(out_folder + image_name + '.png',
bbox_inches='tight', pad_inches=0)
    plt.close()

btc =
pd.read_csv("train_val_data/BTC_train.csv", lineterminator='\n')

for i in
tqdm(range(min(10000, btc.shape[0]))):
    timeseries = np.array(btc.iloc[i, 1:],
dtype='float32')
    visualize_series(timeseries, str(i), BTC_OUT_DATA_FOLDER)

eth =
pd.read_csv("train_val_data/ETH_train.csv", lineterminator='\n')

for i in
tqdm(range(min(10000, eth.shape[0]))):
    timeseries = np.array(eth.iloc[i, 1:],
```

```
dtype='float32')
    visualize_series(timeseries, str(i), ETH_OUT_DATA_FOLDER)

xrp =
pd.read_csv("train_val_data/XRP_train.csv", lineterminator='\n')

for i in
tqdm(range(min(10000, xrp.shape[0]))):
    timeseries = np.array(xrp.iloc[i, 1:],
dtype='float32')
    visualize_series(timeseries, str(i),
XRP_OUT_DATA_FOLDER)

**************************
**Duomenu aibiu
paruosimas**
**************************

#!/usr/bin/env python
# coding: utf-8

import pandas
as pd
from tqdm import tqdm
from sklearn.model_selection import
train_test_split

TIME_SERIES_SIZE = 256
IN_PRICES_DATA_FOLDER =
"clean_data/prices/"
IN_TWEETS_DATA_FOLDER =
"clean_data/tweets/"
OUT_DATA_FOLDER = "train_val_data/"
USEROWS =
100000

def prepare_dataset(ticker):
    tweets = pd.read_csv(IN_TWEETS_DATA_FOLDER + ticker +
"_tweets.csv", lineterminator='\n', parse_dates=['timestamp']).head(USEROWS)

prices = pd.read_csv(IN_PRICES_DATA_FOLDER + ticker + "USDT_minute.csv",
parse_dates=['Date'])
    prices["price"] = (prices.Open + prices.Close) / 2

timeseries = pd.DataFrame([], columns=[str(i) for i in range(TIME_SERIES_SIZE)])
    indexes =
[]
    for i in tqdm(range(tweets.timestamp.shape[0])):
        tempdata =
prices[prices.Date>=tweets.timestamp[i]].price.head(TIME_SERIES_SIZE)
        tempdata =
(100 * (tempdata / tempdata.iat[0] - 1))
        if tempdata.shape[0] != 0:

indexes.append(i)
            tempdf = pd.DataFrame(tempdata.array.reshape(1,
TIME_SERIES_SIZE), columns=[str(i) for i in range(TIME_SERIES_SIZE)])
            timeseries =
pd.concat([timeseries, tempdf])
    timeseries.reset_index(drop=True, inplace=True)
    df =
pd.concat([tweets.iloc[indexes], timeseries], axis=1)
    df = df.drop(['username',
'timestamp'], axis=1)
    train, val = train_test_split(df, test_size=0.25, random_state=42)

 train.to_csv("train_val_data/" + ticker + "_train.csv", index=False)

val.to_csv("train_val_data/" + ticker + "_val.csv", index=False)

for
ticker in ["BTC", "ETH", "XRP"]:

prepare_dataset(ticker)
```

```
**********
**Parametru vertinimo
paleidimas**
**********

#!/usr/bin/env python
# coding: utf-8

from
CLIP import main

main()

**********
**Darbo
grafikai**
**********

#!/usr/bin/env python
# coding: utf-8

import pandas as pd
from
tqdm import tqdm
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 15})

ls
clean_data/prices

ls clean_data/tweets

# # Tweet count x price plots

# ## BTC

p =
pd.read_csv("clean_data/prices/BTCUSDT_day.csv", parse_dates=['Date'])[['Date',
'Close', 'Volume']]
p.head()

t = pd.read_csv("clean_data/tweets/BTC_tweets.csv",
lineterminator='\n', parse_dates = ['timestamp'])
t['Date'] = t.timestamp.dt.date
t['counter']
= 1
t.head()

tt = t.groupby(['Date']).counter.value_counts()
tt = tt.reset_index(name =
'total')[['Date', 'total']]
tt.Date = pd.to_datetime(tt.Date)
tt.head()

j = pd.merge(p,tt,
on='Date')
j.head()

j.corr()

fig, ax1 = plt.subplots(figsize = (15, 7))

ax2 =
ax1.twinx()
ax2.plot(j.Date, j.total, 'b-', alpha=0.8)
ax1.plot(j.Date, j.Close, 'g-',
alpha=0.8)

ax1.set_xlabel('Data', fontsize=20)
ax1.set_ylabel('Kaina', color='g',
fontsize=20)
ax2.set_ylabel('?raš? kiekis', color='b',
fontsize=20)
```

```
plt.savefig('btc_close_count.png', dpi=100)
plt.show()

fig, ax1 =
plt.subplots(figsize = (15, 7))

ax2 = ax1.twinx()
ax2.plot(j.Date, j.total, 'b-',
alpha=0.8)
ax1.plot(j.Date, j.Volume, 'g-', alpha=0.8)

ax1.set_xlabel('Data',
fontsize=20)
ax1.set_ylabel('Suprekiautas kiekis', color='g',
fontsize=20)
ax2.set_ylabel('?raš? kiekis', color='b',
fontsize=20)

plt.savefig('btc_volume_count.png', dpi=100)
plt.show()


# ## XRP

p =
pd.read_csv("clean_data/prices/XRPUSDT_day.csv", parse_dates=['Date'])[['Date',
'Close', 'Volume']]
p.head()

t = pd.read_csv("clean_data/tweets/XRP_tweets.csv",
lineterminator='\n', parse_dates = ['timestamp'])
t['Date'] = t.timestamp.dt.date
t['counter']
= 1
t.head()

tt = t.groupby(['Date']).counter.value_counts()
tt = tt.reset_index(name =
'total')[['Date', 'total']]
tt.Date = pd.to_datetime(tt.Date)
tt.head()

j = pd.merge(p,tt,
on='Date')
j.head()

j.corr()

fig, ax1 = plt.subplots(figsize = (15, 7))

ax2 =
ax1.twinx()
ax2.plot(j.Date, j.total, 'b-', alpha=0.8)
ax1.plot(j.Date, j.Close, 'y-',
alpha=0.8)

ax1.set_xlabel('Data', fontsize=20)
ax1.set_ylabel('Kaina', color='y',
fontsize=20)
ax2.set_ylabel('?raš? kiekis', color='b',
fontsize=20)

plt.savefig('xrp_close_count.png', dpi=100)
plt.show()

fig, ax1 =
plt.subplots(figsize = (15, 7))

ax2 = ax1.twinx()
ax2.plot(j.Date, j.total, 'b-',
alpha=0.8)
ax1.plot(j.Date, j.Volume, 'y-', alpha=0.8)

ax1.set_xlabel('Data',
fontsize=20)
```

```
ax1.set_ylabel('Suprekiautas kiekis', color='b',
fontsize=20)
ax2.set_ylabel('?raš? kiekis', color='y',
fontsize=20)

plt.savefig('xrp_volume_count.png', dpi=100)
plt.show()


# # Price change
plots

plt.rcParams.update({'font.size': 22})


# ## BTC

btc =
pd.read_csv("train_val_data/BTC_train.csv",
lineterminator='\n')
plt.figure(figsize=(15,
7))
plt.xlabel("Indeksas")
plt.ylabel("Kainos pokytis %")
plt.plot([i for i
in range(256)], btc.iloc[0, 1:])
plt.savefig('btc_pct_change.png')


# ## ETH

eth =
pd.read_csv("train_val_data/ETH_train.csv",
lineterminator='\n')
plt.figure(figsize=(15,
7))
plt.xlabel("Indeksas")
plt.ylabel("Kainos pokytis %")
plt.plot([i for i
in range(256)], eth.iloc[0, 1:])
plt.savefig('eth_pct_change.png')


# ## XRP

xrp =
pd.read_csv("train_val_data/XRP_train.csv",
lineterminator='\n')
plt.figure(figsize=(15,
7))
plt.xlabel("Indeksas")
plt.ylabel("Kainos pokytis %")
plt.plot([i for i
in range(256)], xrp.iloc[0, 1:])
plt.savefig('xrp_pct_change.png')


# # Training and
validation loss plots

# ## Original CLIP

def get_losses(data):
    tl = []
    vl = []


for i in df:
        if 'train_loss' in i:

tl.append(float(i.split("train_loss=")[1].split("]")[0]))

        if
'valid_loss' in i:
```

```
vl.append(float(i.split("valid_loss=")[1].split("]")[0]))

print("Lengths", len(tl), len(vl))
    return tl, vl


# ### BTC

df =
pd.read_fwf('training_logs/BTC_images_100_epoch_10000.txt').INTRO.values
btc_tl, btc_vl =
get_losses(df)
n = 50
ids = [i for i in range(n)]

plt.figure(figsize=(8, 6),
dpi=100)
plt.xlabel("Epocha")
plt.plot(ids, btc_tl[:n], label = 'Mokymosi
paklaida')
plt.plot(ids, btc_vl[:n], label = 'Validavimo paklaida')
plt.legend(loc="upper
right")
plt.savefig('image_loss.png', dpi=100)


# ### ETH

df =
pd.read_fwf('training_logs/ETH_images_50_epoch_10000.txt').INTRO.values
eth_tl, eth_vl =
get_losses(df)
n = len(eth_tl)
ids = [i for i in range(n)]
plt.figure(figsize=(8, 6),
dpi=100)
plt.xlabel("Epocha")
plt.plot(ids, eth_tl[:n], label = 'Mokymosi
paklaida')
plt.plot(ids, eth_vl[:n], label = 'Validavimo paklaida')
plt.legend(loc="upper
right")
plt.savefig('eth_mage_loss.png', dpi=100)


# ### XRP

df =
pd.read_fwf('training_logs/XRP_images_50_epoch_10000.txt').INTRO.values
eth_tl, eth_vl =
get_losses(df)
n = len(eth_tl)
ids = [i for i in range(n)]
plt.figure(figsize=(8, 6),
dpi=100)
plt.xlabel("Epocha")
plt.plot(ids, eth_tl[:n], label = 'Mokymosi
paklaida')
plt.plot(ids, eth_vl[:n], label = 'Validavimo paklaida')
plt.legend(loc="upper
right")
plt.savefig('xrp_mage_loss.png', dpi=100)


# ## Modified CLIP

def
parse_output(filename, imagepath):
    file = open(filename,mode='r')
    df = file.read()

file.close()

    trains = []
    valids = []
```

```
    byepoch = df.split('Epoch')
    for
epoch in tqdm(byepoch[1:]):
        a = []
        b = []
        byline = epoch.split('\n')

     for i in byline:
            if 'train_loss' in i:

a.append(float(i.split("train_loss=")[1].split("]")[0]))
            if
'valid_loss' in i:

b.append(float(i.split("valid_loss=")[1].split("]")[0]))

trains.append(a[-1])
        valids.append(b[-1])

    n = len(trains)
    ids = [i for
i in range(n)]
    plt.figure(figsize=(8, 6), dpi=100)
    plt.xlabel("Epocha")

plt.plot(ids, trains[:n], label = 'Mokymosi paklaida')
    plt.plot(ids, valids[:n], label =
'Validavimo paklaida')
    plt.legend(loc="upper right")
    plt.savefig(imagepath,
dpi=100)


parse_output('training_logs/BTC_100000_training.txt',
'btc_ts_loss.png')

parse_output('training_logs/ETH_100000_training.txt',
'eth_ts_loss.png')

parse_output('training_logs/XRP_100000_training.txt', 'xrp_ts_loss.png')
```