IT UNIVERSITY OF COPENHAGEN

Large Scale Data Analysis Exam

BSLASDA1KU

Alexander Thoren
Student No.: 21381
alct@itu.dk

I hereby declare that I have answered these exam questions myself without any outside help.

Alexander Christian Thoren *29/05/2024*

BSc in Data Science
IT University of Copenhagen
May, 2024

Figure 1: List of influencers

1 Scalable data processing

This section involves data processing of Yelp reviews.

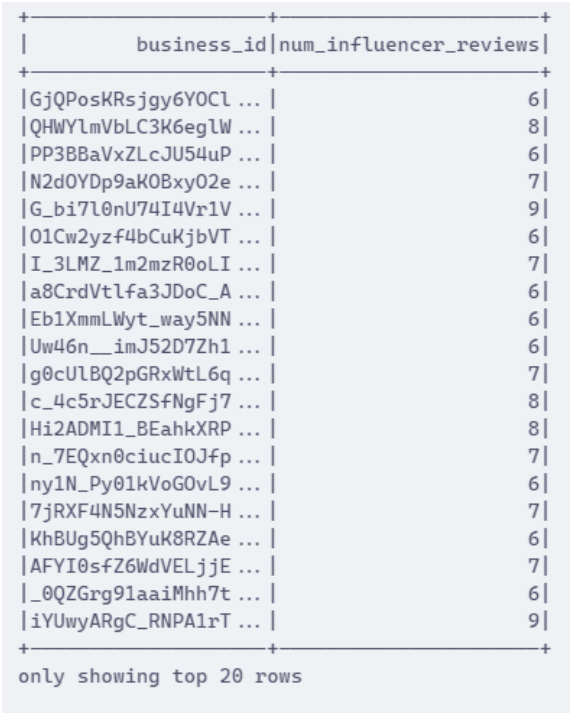
1.1 Influencers and popular businesses

I first looked to find businesses that had been reviewed by influential users, defined as users who have reviewed at least 1000 businesses. These influential users were first found by counting the number of reviews of each user, and subsequently filtering the users by those with a count higher than 1000. The counting made use of the PySpark `groupBy` and `agg` methods to first group the list of reviews by the review authors, and then aggregate the reviews of each user by, for each review, assigning it the number 1 using `F.lit(1)` and then using `F.count` to count the number of 1s.

```
userReviewCount = reviews.groupBy("user_id").agg(
    F.count(F.lit(1)).alias("num_reviews")
)
influencers = userReviewCount\
                .filter(userReviewCount.num_reviews > 1000)\
                .select("user_id")
influencers.show()
```

The step of finding the actual influencers was then trivial. I could simply make use of PySparks `filter` method to select the users with more than 1000 reviews. See Figure 1 for the output of the above command, showing a list of users determined to be influencers.

The next step was to find the most popular businesses, those that had been left a review by more than 5 influencers. This was done in a very similar way to finding the influencers themselves. First, the number of influencer reviews, that is, reviews from influencers, each business had was calculated. First the reviews were filtered by those that had been reviewed by an influencer. This was done using an inner join between the DataFrame of reviews, and that of the influencers. Since an inner join leaves out rows that do not contain a match in either DataFrame, only the reviews authored by influencers remained.



```
+----------------------+-----------------------+
|         business_id|num_influencer_reviews|
+----------------------+-----------------------+
|GjQPosKRsjgy6YOCl ... |                      6|
|QHWYlmVbLC3K6eglW ... |                      8|
|PP3BBaVxZLcJU54uP ... |                      6|
|N2dOYDp9aKOBxyO2e ... |                      7|
|G_bi7l0nU74I4Vr1V ... |                      9|
|O1Cw2yzf4bCuKjbVT ... |                      6|
|I_3LMZ_1m2mzR0oLI ... |                      7|
|a8CrdVtlfa3JDoC_A ... |                      6|
|Eb1XmmLWyt_way5NN ... |                      6|
|Uw46n__imJ52D7Zh1 ... |                      6|
|g0cUlBQ2pGRxWtL6q ... |                      7|
|c_4c5rJECZSfNgFj7 ... |                      8|
|Hi2ADMI1_BEahkXRP ... |                      8|
|n_7EQxn0ciucIOJfp ... |                      7|
|ny1N_Py01kVoGOvL9 ... |                      6|
|7jRXF4N5NzxYuNN-H ... |                      7|
|KhBUg5QhBYuK8RZAe ... |                      6|
|AFYI0sfZ6WdVELjjE ... |                      7|
|_0QZGrg91aaiMhh7t ... |                      6|
|iYUwyARgC_RNPA1rT ... |                      9|
+----------------------+-----------------------+
only showing top 20 rows
```

Figure 2: Popular businesses: businesses reviewed by more than 5 influencers.

```
influencerReviews = reviews.join(influencers, "user_id", "inner")
```

Next, the same procedure as above was used to count the numbers of reviews for each business. Since reviews were now filtered by those left by influencers, this effectively counted the number of influencer reviews left at each business, again using the `groupBy` and `agg` methods from PySpark, as well as the counting trick.

```
influencerReviewCount = influencerReviews.groupBy("business_id").agg(
    F.count(F.lit(1)).alias("num_influencer_reviews")
)
```

Finally, to find the popular businesses with more than 5 influencer reviews, another trivial filter can be applied.

```
popularBusinesses = influencerReviewCount.filter(
    influencerReviewCount.num_influencer_reviews > 5
)
popularBusinesses.show()
```

See Figure 2 for a shortlist of popular businesses, the output from the above code.

1.2 Analysing authentic language

I also analysed the reviews for their use of authentic language linked to various cuisines. This involved finding food reviews, determining for each review whether it was authentic, as well as what cuisine was reviewed. I started by determining authenticity. A review was determined authentic if it contained any of the words "authentic", "veritable" or "legitimate". This was done by reducing the DataFrame of reviews by using the PySpark `where` and `reduce` methods for filtering, as well as the `lower` and `like` functions for the filter condition.

```
authent_words = ["authentic", "veritable", "legitimate"]
authentic_reviews = reviews.where(
```

```
    reduce(
        lambda a, b: a|b,
        (lower(reviews.text).like(f'%{word}%') for word in authent_words)
    )
)
num_authentic = authentic_reviews.rdd.count()
num_total = reviews.rdd.count()
print(num_authentic)
# 129945
```

This simply checked for each review whether it contained any of the words deemed to be indicators of authenticity, and removed those that did not. The next step was to determine which cuisine the reviews referring to. This was done by looking at the review categories. Each Yelp review contains a list of categories the review belongs to. These categories refer to establishments such as "Casino" and "Hotel", but also restaurants and their cuisines, such as "Sri Lankan" and "Indian". To assign a cuisine to each review, I therefore manually investigated the review categories to determine the major unique Yelp cuisines, then compiled a list of these.

```
cuisines = [
    "Indian",          "Taiwanese",      "Chinese",         "American (Traditional)",
    "American (New)", "French",          "Mexican",         "Vietnamese",
    "Lebanese",        "Greek",          "Italian",         "Trinidadian",
    "Filipino",        "Cuban",          "Korean",          "Thai",
    "Japanese",        "Hawaiian",       "Latin American", "Peruvian",
    "Spanish",         "Ukrainian",      "Irish",           "Brazilian",
    "Senegalese",      "Argentine",      "German",          "Sri Lankan",
    "Soul Food",       "Ethiopian",      "Caribbean",       "Mediterranean",
    "African",         "Middle Eastern"
]
```

I then used PySparks `udf` or User Defined Function feature to create a function that assigns a given review a specific cuisine if it contains one of the above cuisines, or `None` if none were found. The function was then applied to the DataFrame of reviews and businesses, where its result was added as a new column:

```
def find_cuisine(categories):
    for cuisine in cuisines:
        if cuisine in categories:
            return cuisine
    return None
find_cuisine_udf = udf(find_cuisine, StringType())

reviews_business = reviews.join(business, on="business_id")
reviews_business_cuisines = reviews_business.withColumn(
    "cuisine", find_cuisine_udf(reviews_business.categories)
)
```

To ensure only reviews of food were included, I also added a filter that removed any reviews that did not include the "food" category:

```
reviews_business_cuisines_food = reviews_business_cuisines.filter(
    lower(reviews_business_cuisines.categories).contains('food')
)
```

To determine the prevalence of authentic language for each cuisine, I filtered the food reviews by those that contained authentic languages by using the aforementioned technique to only include reviews that mentioned authentic words. By then grouping by cuisine and counting the number of reviews for each cuisine, I finally arrived at a DataFrame that showed a breakdown over the number of authentic reviews for each cuisine, shown in figure 3

cuisine	num_reviews_authentic
null	9915
Mexican	9741
Chinese	4234
Italian	3406
American (Traditi ...	3041
American (New)	2137
Indian	1842
Vietnamese	1629
Greek	1113
Japanese	1059
Thai	1026
Cuban	871
Latin American	837
Soul Food	677
Korean	629
Mediterranean	627
German	617
French	583
Taiwanese	448
Caribbean	378
Middle Eastern	242
Hawaiian	198
Filipino	182
Lebanese	158
Brazilian	134
Ethiopian	121
Spanish	99
Irish	94
Peruvian	90
Ukrainian	74
Trinidadian	61
African	31
Argentine	27
Sri Lankan	2
Senegalese	1

Figure 3: Number of authentic reviews for each cuisine

```
authentic_reviews_business_cuisines_food = reviews_business_cuisines_food.where(
    reduce(
        lambda a, b: a|b,
        (lower(reviews_business_cuisines_food.text)\
            .like('%' + word + '%') for word in authent_words)
    )
)
authentic_reviews_count_by_cuisine = authentic_reviews_business_cuisines_food\
    .groupBy("cuisine").agg(
        F.count(F.lit(1)).alias("num_reviews_authentic")
    ).sort("cuisine")
authentic_reviews_count_by_cuisine\
    .sort("num_reviews_authentic", ascending=False)\
    .show(authentic_reviews_count_by_cuisine.rdd.count())
```

This by itself did not give much information, as any notable difference in the number of authentic reviews may simply be a result of that cuisine having more overall reviews. I therefore calculated

cuisine	num_reviews_all	reviews_percent_authentic
Ukrainian	418	17.703349282296653
Trinidadian	499	12.224448897795591
German	5477	11.265291217819975
Peruvian	853	10.550996483001173
Ethiopian	1463	8.270676691729323
Filipino	2241	8.121374386434628
Caribbean	4842	7.806691449814126
Mexican	129678	7.511682783509925
Taiwanese	6131	7.307127711629424
Lebanese	2176	7.2610294117647065
Indian	25854	7.12462288233929
Cuban	12346	7.054916572169123
Sri Lankan	29	6.896551724137931
Thai	15123	6.784368180916485
Vietnamese	24059	6.770854981503803
Chinese	69490	6.092963016261333
Korean	10540	5.967741935483871
Argentine	458	5.895196506550218
Spanish	1747	5.666857469948483
Greek	20027	5.557497378538972
Latin American	15149	5.525117169450128
African	575	5.391304347826087
Senegalese	19	5.263157894736842
Mediterranean	13880	4.517291066282421
Middle Eastern	7274	3.326917789386857
Italian	105445	3.230119967755702
Japanese	35350	2.995756718528996
Irish	3387	2.775317390020667
Brazilian	5002	2.6789284286285486
French	22872	2.548968170689052
Hawaiian	7836	2.5267993874425727
Soul Food	32012	2.1148319380232414
American (New)	302749	0.7058652547159529
American (Traditi ...	462232	0.6578947368421052

Figure 4: Percentage of reviews that include authentic language for each cuisine

the percentage of reviews for each cuisine that contained authentic language, by counting the number of reviews within each cuisine, and then dividing the number of authentic reviews by the total number of reviews for each cuisine.

```
reviews_count_by_cuisine = reviews_business_cuisines_food\
    .groupBy("cuisine").agg(
        F.count(F.lit(1)).alias("num_reviews_all")
    ).sort("cuisine")
reviews_counts_percentages = authentic_reviews_count_by_cuisine\
    .join(reviews_count_by_cuisine, "cuisine")\
    .withColumn(
        "reviews_percent_authentic",
        (F.col("num_reviews_authentic") / F.col("num_reviews_all"))
    )
display = reviews_counts_percentages\
    .select("cuisine", "num_reviews_all", "reviews_percent_authentic")\
    .withColumn("reviews_percent_authentic", (F.col("reviews_percent_authentic") * 100))\
    .sort("reviews_percent_authentic", ascending=False)
display.show(display.rdd.count())
```

The results of this is shown in Figure 4.

From Figure 4 we can see that a significantly higher percentage of Yelp reviews use authentic language to describe Ukranian cuisine when compared to, say, American cuisine. American cuisine may be so low on the list as a majority of Yelp users are likely American, and thus don't consider their own cuisine authentic. Based on the number of reviews however, it may be hard to conclude that the percentages displayed in Figure 4 are general tendencies. As an example, Ukranian cuisine only has 418 reviews, and so the high percentage of authentic reviews may be a result of either random noise in the sample, or due to a cultural under-representation of Ukranian food.

1.3 Authenticity, negativity and stereotypes

The hypothesis with this data was that reviews that contained both negatibe and authentic language tended to be about non-"western" restaurants (western here referring mainly to the anglicized parts of the world with), such as Asian and Mexican cuisines. To explore this, I wanted to see what percentage of reviews for each cuisine were both authentic and negative, as well as compare this percentage across wester/european vs. non-wester/european cuisines.

I did this by first defining a review as negative or positive if it included negative or positive words respectively. See the code below for a list of positive and negative words. This resultet in 2 DataFrames: one for authentic and positive reviews, as well as one for authentic and negative reviews:

```
neg_words  = [ "dirt",      "kitsch",    "cheap",      "rude",
               "simple",    "bland",     "dodgy",      "poison" ]
pos_words  = [ "clean",     "refined",   "elegant",    "stylish" ]
auth_words = [ "authentic", "veritable", "legitimate" ]

rest_rs = business[
    business.categories.contains('Restaurants')
].join(reviews, "business_id")
rest_rs = rest_rs.withColumn("isAuthentic", rest_rs.text.rlike(f'({")|(".join(auth_words)})'))
rest_rs = rest_rs.withColumn("isNeg", rest_rs.text.rlike(f'({")|(".join(neg_words)})'))
rest_rs = rest_rs.withColumn("isPos", rest_rs.text.rlike(f'({")|(".join(pos_words)})'))
rest_rs = rest_rs.withColumn("cuisine", find_cuisine_udf(rest_rs.categories))
rrs_auth = rest_rs.where(rest_rs.isAuthentic)
rrs_auth_neg = rrs_auth.where(rrs_auth.isNeg)
rrs_auth_pos = rrs_auth.where(rrs_auth.isPos)
```

Like earlier, I could then count the number of negative and positive reviews for each cuisine using the PySpark `groupBy` and `agg` methods. This gave me a DataFrame where each row is a cuisine, and column for negative and positive review counts. To this I also added the total number of reviews, as well as calculating the fraction of negative and positive reviews for each cuisine based on these figures.

```
neg_cuis = rrs_auth_neg.groupBy("cuisine").agg(
    F.count(F.lit(1)).alias("neg")
)
pos_cuis = rrs_auth_pos.groupBy("cuisine").agg(
    F.count(F.lit(1)).alias("pos")
)
cuis_revs = neg_cuis.join(pos_cuis, on="cuisine")
reviews_count_by_cuisine = rest_rs.groupBy("cuisine").agg(
    F.count(F.lit(1)).alias("tot_revs")
)
cuis_revs = cuis_revs.join(reviews_count_by_cuisine, "cuisine")

cuis_revs = cuis_revs.withColumn("neg_p", (cuis_revs.neg / cuis_revs.tot_revs) * 100)
cuis_revs = cuis_revs.withColumn("pos_p", (cuis_revs.pos / cuis_revs.tot_revs) * 100)

cuis_revs.show(cuis_revs.rdd.count())
```

See Figure 5 to see the DataFrame generated by the above code. It is important to again mention that these counts are of reviews that are both considered authentic as well as positive/negative. In actuality, there are significantly more positive/negative reviews, but most of them do not contain authentic language. The fractions in the DataFrame have also been multiplied by 100 for readability. Already we can see that, for example, the Mexican cuisine has more negative than positive reviews, having around 0.9% negative reviews and 0.5% positive.

cuisine	neg	pos	tot_revs	neg_p	pos_p
Mexican	3592	1839	380123	0.9449572901402967	0.48379077298663853
Ethiopian	22	27	5549	0.3964678320418093	0.4865741575058569
Thai	456	386	80192	0.568635275339186	0.48134477254588987
Indian	432	421	79229	0.5452548940413232	0.53137108886897948
Chinese	1348	909	213612	0.6310506900361403	0.42553789112971185
Soul Food	56	30	32012	0.1749343996001 4995	0.09371485692865175
Taiwanese	85	85	10697	0.794615312704 4966	0.7946153127044966
Ukrainian	7	2	483	1.44927536231884 06	0.41407867494824 02
Irish	30	19	14246	0.2105854274884 1777	0.13337077074266 46
Hawaiian	22	25	11251	0.1955381743844 9916	0.22220247089147632
Japanese	386	227	152984	0.2523139674737227	0.14838153009465 04
Latin American	153	134	29636	0.5162640032393035	0.45215278715076257
Vietnamese	516	470	68512	0.7531527323680522	0.6860112097150864
Italian	892	431	348203	0.2561724051774396	0.12377837066309 02
Caribbean	71	68	14201	0.4999647912118 8653	0.47883951834377786
Korean	240	207	39066	0.6143449546920596	0.52987252342190 14
French	102	67	45003	0.2266515565628958	0.14887896362464725
American (Traditional)	620	315	1005619	0.0616535685980475 76	0.03132399049 7395135
Middle Eastern	59	46	14479	0.4074867048829339	0.31770149872228 74
Greek	206	183	53095	0.3879838026179 4896	0.34466522271400324
Cuban	172	130	25814	0.6663051057565662	0.50360269621135 81
American (New)	435	237	636333	0.0683604339237 4747	0.03724465020673138
German	149	55	10832	1.3755539143279172	0.507754800590842
Mediterranean	160	129	41836	0.3824457405105 6507	0.30834687828664 31
Filipino	25	23	3774	0.6624271330153684	0.60943296237413 88
Spanish	38	20	12408	0.3062540296582 8497	0.16118633139909735
Senegalese	1	1	125	0.8	0.8
Trinidadian	2	7	648	0.3086419753086 4196	1.08024691358024 68
Peruvian	12	8	2132	0.5628517823639775	0.37523452157598497
Lebanese	36	40	8302	0.4336304504938569 6	0.48181161165984 1
African	8	4	1927	0.4151530877010898	0.20757654385054 49
Brazilian	25	11	10723	0.2331437097827 1007	0.10258323230439242

Figure 5: Negative, positive and total review count by cuisine.

isWestern	avg(pos_p)	avg(neg_p)
true	0.20059476483121436	0.3417258991065201
false	0.47714908536035 82	0.57658234166804 43

Figure 6: Table of average positive review percentage and negative review percentage for both western and non-western cuisines.

From this table, I calculated the average negative and positive fraction of reviews for cuisines determined to be western and non-western. A western cuisine was determined to be cuisines often associated with high-end dining experiences. See the below code for a list of western cuisines. This resulted in a final DataFrame that included figures for average fraction of positive reviews and the average fraction of negative reviews for both western and non-western cuisines.

```
western_cuisines = [
    "American (New)", "American (Traditional)", "French", "German",   "Greek",
    "Hawaiian",       "Mediterranean",          "Italian" "Japanese", "Irish"
]

cuis_revs = cuis_revs.withColumn("isWestern", cuis_revs.cuisine.isin(western_cuisines))

ctr_p = cuis_revs.groupBy("isWestern").agg(F.mean("pos_p"), F.mean("neg_p"))
ctr_p.show(ctr_p.rdd.count(), False)
```

The DataFrame from the above is displayed in Figure 6. As we can see from the table, there is a higher fraction of negative reviews for both western and non-western cuisines, though western cuisine has around 0.14% more negative than positive reviews, where non-western cuisines only have around 0.1%. The most notable difference between western and non-western cuisines is that botn the positive and the negative fraction is significantly higher for non-western cuisines. This is likely fraction of "authentic" reviews being higher for non-western cuisines, thus inflating the overall number of reviews captured.

While the results don't show anything that would confirm that negative authentic language is more prevalent for non-western cuisines, it does show that non-western cuisines tend to be described by authentic language more often than western cuisine. As mentioned earlier, this may be due to the demographic of the review website Yelp being predominantly westerners that do not

consider their own cuisine "authentic".

1.4 Machine learning on Spark

Finally, I trained a linear regression model on various features of the reviews to predict the number of stars a review would give. First, I used the existing binary features included in the data. This was whether a review was cool, funny, and useful. Then, I included the number of reviews held by the particular business being reviewed. I then engineered features based on authenticity, negative and positive language by counting the number of authentic, negative and positive words used in the review respectively. The next feature I created was based on the review date. Both year, month and day of the review were used as three separate features. Finally, I also used the American state that the business was located in by one-hot encoding it into 27 different binary variables. In total, I had the following features:

```
features = [
    "cool",  "funny", "useful",     "review_count", "year",
    "month", "day",   "auth_count", "neg_count",    "pos_count",
] + states # list of US states
```

I trained a Linear Regression model to predict the review star count. This was done using Cross-Validation, by randomly splitting the reviews into 5 splits. Then training on 4 of the splits, and evaluating on the 5th. This was repeated until all 5 splits had been evaluated once, and the evaluation metrics were averaged. Using Cross Validation ensures that the result is representative of what you can expect to see when evaluating on unseen data.

The optimization metric I chose was Sum of Squared Error (SSE). SSE is the most common metric used when training Linear Regression models. The squared error is calculated by taking the square of the difference between the prediction and the true value:

$$\text{squared error} = \left(Y_i - \hat{Y}_i\right)^2$$

The square is used both to ensure that the error is always positive, as well as to punish large errors exponentially more than small errors. The sum of all these errors is then taken to be the models current performance:

$$\text{SSE} = \sum_{i=1}^{N} \left(Y_i - \hat{Y}_i\right)^2$$

The goal of the model is to minimize this value by finding the line of best fit, which can be done analytically.

For evaluating the final results of the model, I used both Mean Squared Error (MSE) and Adjusted R^2. MSE is similar to SSE, but takes the mean instead of the sum of the squared errors. This is helpful to see if there are some very large errors in the final model, as the SSE would change significantly based on the number of predictions. Adjusted R^2 is a variant of the popular R^2 metric. R^2 is a metric that measures how well the linear regression model fits the data, or more specifically, it measures the proportion of the total variation in the dependent variable that is explained by the predictor variables. R^2 is a very popular metric to use, but it has some shortcomings, especially when using Multiple Linear Regression. R^2 fails to take into account the number of predictors in the model, and will always prioritise models with more predictors. Since I made use of quite a lot of predictors, I used Adjusted R^2 in hopes of comparing different nested models with each other, though I did not end up implementing this. Despite this, Adjusted R^2 is still a useful metric to determine how well the model fits the data.

The result of training the model using 5 folds of cross-validation and averaging the results are given in Table 1

The results aren't great, with an Adjusted R^2 of only 0.09. This indicates that most of the variance isn't explained by the predictors, and thus, the model is not very good at predicting the number of stars a review will give. Improvements could probably be made by doing regularization and removing unneeded features, doing more data preprocessing such as LDA or PCA, and even using a language model to make predictions based on the entire review text.

	MSE	Adj. R^2
5-split Cross Validation results	1.98	0.0939

Table 1: Results from training a Linear Regression model through 5-fold cross validation.

Renewable Energy Generation		
Key	Type	Description
time	float	Time of measurement
ANM	float	Not relevant
Non-ANM	float	Not relevant
Total	float	Renewable power generation in MegaWatts

Weather Forecasts		
Key	Type	Description
time	float	Target time of forecasts
Speed	float	Wind speed in M/S
Direction	string	Wind direction, e.g. "S" or "NW"
Source_time	integer	Time of forecast generation
Lead_hours	string	Forecast horizon in hours

Table 2: Schema over datasets

2 ML lifecycle

This section revolves around machine learning using MLflow.

2.1 Data Alignment

The goal of this assignment was predicting renewable energy generation based on wind forecast data. I had access to two datasets, a time series tracking energy generation and a time series tracking wind speed and direction. See Table 2 for a schema over the two datasets. This data was imported in the form of two Pandas (pandas development team, 2020, McKinney, 2010) DataFrames.

The first step involved joining the two dataset into a single DataFrame. This was quite tricky as the datasets measurements intervals. The Weather Forecast dataset was measured on intervals of 3 hours while the Power Generation dataset was measured on intervals of 1 minute. Simply performing an inner join would discard 59/60 of the data. I therefore decided to fill in the gaps in the Forecast data for each datapoint in the Power Generation dataset. This was done by finding the two closest Forecast datapoints to each Power Generation datapoint. That is, for a Power Generation datapoint from 13:41:00, the two closest Forecast datapoints would be at 12:00:00 and at 15:00:00. Data for these two Forecast datapoints were then averaged, except wind direction, where the closest Forecast was used:

```
def wind_average(closest, before, after):
    direction = closest['Direction']
    lead_hours = (int(before['Lead_hours']) + int(after['Lead_hours'])) / 2
    source_time = (int(before['Source_time']) + int(after['Source_time'])) / 2
    speed = (int(before['Speed']) + int(after['Speed'])) / 2
    return direction, lead_hours, source_time, speed
```

The calculated vales were then used to fill new `Direction`, `Lead_hours`, `Source_time` and `Speed` columns. Most adjacent Forecast datapoints are very similar as the wind changes very gradually, and thus, averaging is a good approximation for the true Forecast at each Power Generation interval.

```
power_df['Direction'] = [None] * len(power_df)
power_df['Lead_hours'] = [None] * len(power_df)
```

```
power_df['Source_time'] = [None] * len(power_df)
power_df['Speed'] = [None] * len(power_df)

for index, row in power_df.iterrows():
    time = index
    # Find the wind_df times that surround this power row
    closest_two = wind_df.iloc[
        abs((wind_df.index - time).total_seconds()).argsort()[:2]
    ]
    closest = closest_two.iloc[0]
    before, after = list(closest_two.sort_index().iterrows())
    # if time > after, then this power observation is newer than any wind observation
    # in that case, only use after, the newest one.
    if time > after[0]:
        before = after

    direction, lead_hours, source_time, speed = wind_average(
        closest, before[1], after[1]
    )
    power_df.at[index, 'Direction'] = direction
    power_df.at[index, 'Lead_hours'] = lead_hours
    power_df.at[index, 'Source_time'] = source_time
    power_df.at[index, 'Speed'] = speed
```

This method ensured that every datapoint remained in use by estimating missing data. An issue with this approach however, is that it is still not a perfect estimate. This method causes every Power Generation datapoint between two Forecast datapoints to have the exact same Forecast values. That is, a Power Generation datapoint at 13:30:00 and one at 14:54:00 would both use an average of the Forecast datapoints at 12:00:00 and 15:00:00. This causes a lot of data to be identical. A potential technique to help address this issue may be to use a weighted average instead, where the weight is determined by the closeness of each Forecast datapoint to the specific Power Generation datapoint. This would ensure that every Power Generation datapoint has different data for their Weather Forecast information.

2.2 Feature Transformation

Before training an actual machine learning model on the data, it is important to do perform feature engineering techniques such as feature transformation. A good candidate for this is the Direction property. This property encodes the direction wind from in a given Forecast is coming from. Since this property is given as a string, it cannot be used for most machine learning algorithms as they rely on doing calculations with numbers. The property therefore needs to be transformed into a number, and to do so I applied One Hot Encoding, also known as converting properties to Dummy Variables. This is a technique applied to categorical variables, that turns a variable of n possible categories into $n - 1$ different boolean features. When applied to the Direction property, this technique would create a new column for each direction except one. The column with a value of 1 for a specific datapoint then corresponds to the wind direction. This is very easy to do with Pandas DataFrames, as Pandas contains a method, `get_dummies`, that takes a DataFrame and converts specified columns into Dummy Variables:

```
>>> joined_encoded = pd.get_dummies(power_df, columns=['Direction'])
>>> direction_columns = [c for c in joined_encoded.columns if 'Direction' in c]
>>> direction_columns
['Direction_E',  'Direction_ENE', 'Direction_ESE', 'Direction_N',
 'Direction_NE', 'Direction_NNE', 'Direction_NNW', 'Direction_NW',
 'Direction_S',  'Direction_SE',  'Direction_SSE', 'Direction_SSW',
 'Direction_SW', 'Direction_W',   'Direction_WNW', 'Direction_WSW']
```

This removes the original Direction column, and leaves a new column for each original category, which in this case is 16 new columns. Each column is a boolean, either True or False, and each row can only have a single Direction column be True at a time. This allows a machine learning model to adopt each of directional column as a different feature, applying different weights to each wind direction.

2.3 Data Drift

Data drift is an important problem one needs to keep in mind when deploying machine learning models to a production environment. Data drift refers to when the distribution of the dataset changes, or 'drifts', in some way over time, whether it be the features, the labels or the response variable. This can be quite a common problem as trends tend to come and go, and data available at the point of a models training may be different from the data encountered after model deployment.

An example of this in practice could be changes in equipment. Wind turbines, transmission lines and transformers can all loose performance if not taken care of, or their performance can increase as they are upgraded or more turbines are added. Since the model is trained on recent data covering the entire electricity generation of the Orkney power grid, any of the changes mentioned would negatively impact the predictive power of the machine learning model. For example, installing additional wind turbines would cause power production to increase, with no change in the actual weather forecasts. This is known as Label Shift, where the response variable changes without the predictor variables being affected.

To mitigate data drift, the power generation prediction model has been trained in an easily reusable pipeline format on recent data automatically gathered. This makes it exceedingly simple to retrain the model as the circumstances regarding the data collection change, as the only step required to update the model to one trained on the newest data is simply to execute the training script.

2.4 Hyperparameter Optimisation

To predict the power generation from the weather forecast, I decided to go with a more complex machine learning model, that is, a feed-forward neural network. I went with a neural network due to the impressive results neural networks have shown, but also due to the fact that they have a lot of tunable hyperparameters, which allowed me to explore how MLflow keeps track of model parameters during trials.

A neural network is at its core, a collection of many different linear models separated by non-linear functions called activation functions. The combination of a linear model and an activation function is called a neuron. Each input feature is passed as a parameter to k different linear models, along with all other input features. This is known as a hidden layer. The output from each linear model is then passed to an activation function that makes some non-linear operation on its input value. This is usually taken to be ReLU which can be expressed in Python as `lambda x: max(0, x)`, effectively clamping the lowest value to 0. This output is passed along, either to another hidden layer of the same or a different size, or to an output layer. The output layer contains a linear and an activation function for each response variable.

In my case, I decided to make the size and number of hidden layers each a hyperparameter, with each hidden layer being the same size. The output layer had a single neuron, predicting the megawatts of power generation based on the input features. Additionally, I included a dropout layer after every hidden layer. A dropout layer is a training method that artificially sets the output for some neurons to 0 in the layer preceding it, forcing the next layer to optimize the information gain from each input. The fraction of neurons disabled by the dropout layers was also included as a hyperparameter. The final two hyperparameters were related to the actual model training. These were the learning rate and the number of epochs. Each epoch is a single pass of infering on a set of data, evaluating the result and optimizing the weights for the model. The learning rate is how much the weights are optimized for each epoch.

This left me with 6 hyperparameters that I used Optuna (Akiba et al., 2019) to optimize for each trial. I used the Mean Squared Error for the evaluation metric as it is a good indicator of

MSE	Dropout	Epochs	Hidden layers	Hidden neurons	Learning rate
26.576	0.40589	89	3	93	0.0055492
26.751	0.42559	77	3	94	0.0049576
27.043	0.40451	78	3	94	0.0064252

Table 3: Top 3 models from optimizing using Optuna and tracking using MLflow

error in regression tasks, as it weighs larger errors more. After 2500 trials, the best MSE was 26.6. See Table 3 for an overview over the hyperparameters for the top 3 models.

2.5 Reproducibility

Reproducibility has become an increasing problem in the academic world, to the point that it is considered a crisis today (Baker, 2016). It's therefore very important that any new research and new machine learning models are reproducible. This means that someone reading the about a machine learning model should be able to effectively locally train a model that is a reproduction of the original model.

When training machine learning models using Python, there are several different ways to ensure reproducibility at various levels. The most important element to reproducibility in this scenario is access to the code used to train the model. This can either be through an artifact, a feature provided by libraries like MLflow, or through a public version control system like Git, which allows users to view source code at every iteration. When dynamically determining hyperparameters of models like done in this project, it is also important to make the final chosen hyperparameters public. Usually papers and reports shorten long floating point numbers, so figures from such sources are usually not reliable. Training data should also be provided to users seeking to replicate an experiment. This may not always be possible if training data is private or sensitive in nature, but in scenarios where this is not the case, such data should be provided. Finally, dependencies and their versions should be made clear. This is for Python libraries used, which can be specified in a `requirements.txt` file, or using a library like MLflow, but also extends to larger dependencies like the operating system and system architecture. These can be encapsulated in larger virtualization systems like Containerization using Docker or Podman, or full Virtual Machines using tools like Proxmox.

This project aids reproducibility in a few ways, mainly relying on MLflow. The code for the project is available through GitHub, a provider of servers for Git projects, where it can be copied and reused. In this project, the data is dynamically fetched from a data source every time the script is ran. This means it is important for the training data to be available for each version of the model. MLflow is used to save both the model hyperparameters as well as the model training data as an artifact. This ensures that, as the data changes in the future, old versions of the training data will always be available for users to verify and compare. Finally, I have included both a `requirements.txt` file specifying all used Python dependencies and their versions, as well as `docker-compose.yaml` file for automated model training using a Docker container for optimal reproducibility.

3 Lecture material

This section regards material taught during class.

3.1 Apache Wayang

Apache Wayang (Beedkar et al., 2023) is a framework that provides systematic solution to the problem of having to select and learn a specialized platform for data analytics. Wayang is a unified framework that can execute a data pipeline over any set of platforms seamlessly and efficiently.

There are four main issues with the current data analytics workflow that Apache Wayang aims to address. First is **Platform Independence**. There are currently many different data analytics platforms, all with different APIs, written in different languages, and with different strengths

and weaknesses. A typical data analysis pipeline involves using several different platforms for different parts of the pipeline to ensure efficiency. This is a large burden for developers as it may require re-implementing pipelines in, and learning, new APIs and languages, as well as possibly requiring custom-made scripts for interfacing between these various platforms. Wayang solves these problems by being platform-agnostic, as well as capable of automatically determining what platform individual parts of the pipeline should run on, ensuring that the developer only needs to use a single framework.

This also extends into the second issue and solution, **Opportunistic Cross-Platform**. As mention above, it can be valuable to make use of several different platforms in the same pipeline to make use of the strengths and weaknesses of each platform. Since Wayang can automatically determine the best platform for a given task, it can always choose the best platform for each task in a pipeline, being cross-platform and efficient. Adjacent to this is also the problem of being **Mandatory Cross-Platform**. Even if efficiency is irrelevant for the data analysis pipeline, it may still be necessary to use multiple platforms, as different data analytics platforms have different sets of features. If a pipeline requires two functions, each of which are exclusive to two different platforms, then the pipeline will be forces to make use of both. Wayang's dynamic platform selection trivialises this.

Finally is the issue of **Polystore**, that is, having data in various different data stores. Normally, this would have the developer write different preprocessing scripts for importing the various data formats, but Wayang can dynamically import data from various data lakes, without a developer having to specify how.

A fifth boon not mentioned in the original paper is **Migration**. As an organization expands their architecture, Apache Wayang expand their platform support, or as platforms get deprecated or obsolete, code written for Apache Wayang can be easily migrated to these new platforms by simply changing which plugins are being used. This allows pipeline migrations that would normally require significant investment to be done very easily, thus future-proofing Apache Wayang pipelines for as long as Wayang is actively maintained.

3.2 Request log analysis

This section involves extracting the total number of distinct IP addresses that have connected to each domain in a log file from a web server. The log is formatted as follows:

```
66.24.69.97 -- [17/May/2024:10:25:44 +0000] "GET http://www.google.com/bot.html"
66.24.69.97 -- [17/May/2024:10:26:44 +0000] "GET https://itu.dk/research.html"
66.24.69.97 -- [17/May/2024:10:28:44 +0000] "GET https://itu.dk/programmes/bds.html"
71.19.157.179 -- [17/May/2024:10:30:12 +0000] "GET http://www.google.com/faq.html"
66.24.69.97 -- [17/May/2024:31:10:44 +0000] "GET https://itu.dk/contact.html"
```

This can be achieved in 3 simple steps. The first step is extracting the IP addresses and the domains. This can be done using a simple regular expression. I have created a function that takes a line from the log as an input, and produces a key-value pair of the matched domain to the matched IP address:

```
>>> def findMatches(s: str):
...     x_IP = "(?<IP>(?:\d{1,3}\.){3}\d{1,3})"
...     x_DOMAIN = "(?<DOMAIN>https?:\/\/(?:www)?[^\/]*\.[a-zA-Z]{2,3})"
...     match = re.search(f"{x_IP}.*{x_DOMAIN}", s)
...     return (match.group("DOMAIN"), match.group("IP"))
```

This function can then be used in the PySpark `map` method after reading the file. This produces an RDD of the above mentioned key-value pairs. Since one IP address can visit the same domain multiple times, the distinct domain-IP pairs need to be extracted. This can be done using the `distinct` method which also works for tuples. Finally, the number of occurrences of each key can now be counted using the `countByKey` method, finally giving us the desired output.

```
>>> sc.textFile("hdfs://log.txt")\
...     .map(findMatches)\
```

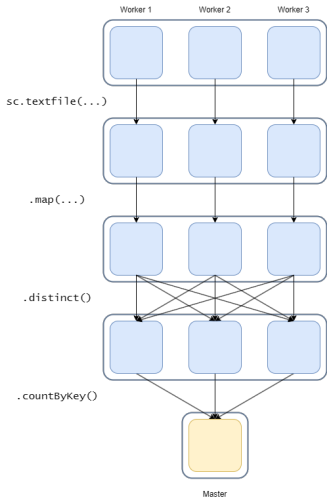


Figure 7: Execution of simple spark query

```
...     .distinct()\
...     .countByKey().items()
[("http://www.google.com", 2), ("https://itu.dk", 1)]
```

Assuming that the underlying Spark platform has 3 available workers, a few steps will be made under the hood to answer this query. First, in the example above, the data is loaded from HDFS, or the Hadoop Distributed File System. This means that the 3 workers can load their part of the data individually, without having to wait for the driver to send their respective data. The subsequent `map` can be done each worker independently of the others. The `distinct` method however, relies on a call to `reduceByKey` under the hood. `reduceByKey` is an action that required aggregating the data from each worker to calculate the result. `countByKey` is similar. Since the results of both `distinct` and `countByKey` rely on information from all workers to ensure that no duplicates are present and all keys are counted respectively, they cannot be entirely run in an isolated manner, and network communication is necessary. See Figure 7 for a visualization of the query.

3.3 Principal Component Analysis

Principal Component Analysis, or PCA, is a dimensionality reduction method. It works by calculating the vector along which the dataset has the largest variance. This vector is known as a Principal Component. It then repeats this process for as many principal components as there are dimension in the dataset, ensuring that each vector is orthogonal to all the previous ones. PCA is a very popular way to do dimensionality reduction because of it's ease of use and the fact that it preserves most of the variance in a dataset.

PCA has however come under criticism for several shortcomings, chief among them being lack of efficiency on large dataset and difficulty of interpretability. To help address these issues, different variants of the original PCA have been made, primarily Incremental PCA to address performance issues, and Sparse PCA to address interpretability issues.

The efficiency issues of PCA stem from the need to perform computations on the entire dataset at once, requiring that the dataset is loaded into memory. This is impossible for larger datasets, as it is not uncommon to find datasets of several gigabytes or larger, especially in areas such as gene research. Incremental PCA addresses this by performing the computations in batches. This allows the algorithm to only load a small fraction of the total dataset into memory at any given time, updating its PCA estimation as it iterates through the dataset.

Interpretability is a problem for PCA, as every datapoint that results from a transformation by PCA becomes a linear combination of the original input variables. This makes it very hard to reason about individual datapoints after a PCA transformation, as every datapoint is dependent on every other datapoint to some extent. Sparse PCA attempts to address this by using a sparse

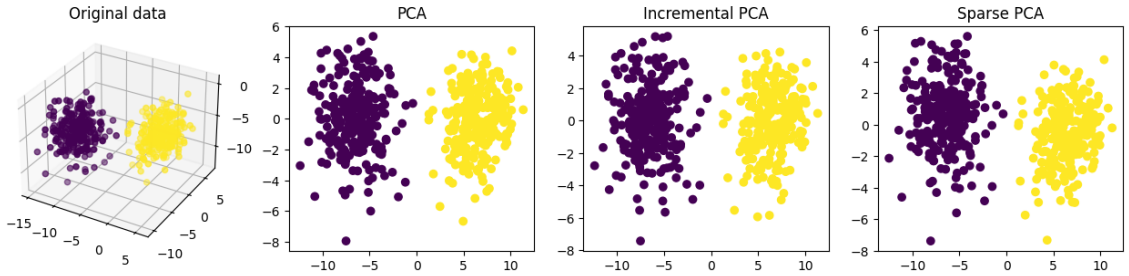


Figure 8: Original, randomly generated 3-dimensional dataset (left), as well as the same dataset reduced to 2 dimensions with PCA, Incremental PCA and Sparse PCA.

set of input variables. That is, instead of using the entire input dataset for the linear combinations, it uses just a few input variables. This means that every transformed datapoint will only be a linear combination of a few input variables, which makes it much easier to reason about.

Comparisons between the three variations of PCA can be seen in Figure 8. The plots are very similar, which shows that despite their added limitations, the PCA variations still match the performance of the original PCA very closely. It is important to note, however, that the dataset provided is already very linearly separable, which likely results in all the PCA variations very easily finding the optimal separability.

3.4 Distributed and Federated Learning

Traditionally, machine learning training tasks have been performed in a manner known as Distributed Learning. Here, a central server is responsible for training a model either by itself, or by using a distributed network of machines. This requires training data to be stored in a location accessible to each compute node involved in training the model. Data used for these tasks are usually publically accessible datasets, such as webscraping.

This can be an issue as a large amount of the data generated in the world is kept locally on individual devices such as smartphones and laptops. These data are also usually very sensitive, such as photos or text input, and are therefore difficult to collect in a secure and private manner. To address these problems and make use of the vast amount of sensitive on-device data, Federated Learning was created as an alternative to Distributed Learning. Where Distributed Learning has one or a particular set of machines serve as compute nodes for training a single central model, Federated Learning has each device contribute its own training data to optimize a local version of the global model, before then communicating weight updates to a centralized control server (McMahan et al., 2016). This allows Federated Learning to be used in cases where privacy is a big concern, such as the photos and text input from personal computing devices mentioned above, or applications such as medical patient data.

4 Bonus Question

A topic that I have become very interested, especially since the lecture about is, is Docker and containerization. Docker is a system for defining all dependencies in a single file, from the operating systems, to installed programs to libraries and files. This allows for extremely simple and reliable deployments, making heavy use of the "Write once, run anywhere" paradigm. Once a configuration for a docker container has been written, it can be clones and deployed on virtually any machine across the globe with no issues. If used properly, this makes it easy to make use of distributed or edge computing.

Normally, Docker containers are very light-weight if deployed on the right operating system, but this also means that they are usually missing a lot of the programs required by most applications. Since Docker images can be built from other Docker images, this allows for easily expandable applications, which means a developer does not necessarily need to build or install all of their own dependencies - another user may have already done so, and released it as a standalone Docker

image. Since Docker images are also version controlled with Tags, this means reproducibility is not lost when creating containers that rely on preexisting Docker images.

All of this makes Docker containers great solutions for reproducibility, ease of deployment and ease of distribution. When variables can be isolated due to containerization, it makes debugging and bug fixing significantly easier.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature*, *533*(7604), 452–454. https://doi.org/10.1038/533452a

Beedkar, K., Contreras-Rojas, B., Gavriilidis, H., Kaoudi, Z., Markl, V., Pardo-Meza, R., & Quiane Ruiz, J. (2023). Apache wayang: A unified data analytics framework. *SIGMOD Record*, *52*(3), 30–35. https://doi.org/10.1145/3631504.3631510

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61). https://doi.org/10.25080/Majora-92bf1922-00a

McMahan, H. B., Moore, E., Ramage, D., & y Arcas, B. A. (2016). Federated learning of deep networks using model averaging. *CoRR*, *abs/1602.05629*. http://arxiv.org/abs/1602.05629

pandas development team, T. (2020, February). *Pandas-dev/pandas: Pandas* (Version latest). Zenodo. https://doi.org/10.5281/zenodo.3509134