

UNICAL · UNIVERSITÀ DELLA CALABRIA

CORSO DI LAUREA MAGISTRALE IN INTELLIGENZA ARTIFICIALE

Progetto Big Data: Dataset Reporting Carrier On-Time Performance

Presentato da:
Giuseppe Zappia
Domenico Macrì

Matricola n°:
268784
268798

Anno Accademico
2024-2025

Contents

1	Introduzione	2
1.1	Il Dataset	2
1.2	Tecnologie utilizzate	2
1.3	Struttura del progetto	2
1.4	Metodologia di lavoro	3
2	Studio della letteratura	4
2.1	Ricerche Precedenti e Applicazioni	4
2.2	Pattern di Analisi Comuni	5
2.3	Approcci Metodologici	5
3	Sezione Home	6
3.1	Passi Iniziali	6
3.2	GUI sezione Home	8
4	Sezione Aeroporti	13
5	Sezione Paesi	27
6	Sezione Compagnie	44
7	Sezione Rotte	54
8	Sezione AI	68

1 Introduzione

1.1 Il Dataset

Il progetto assegnatoci prevede la realizzazione di un'applicazione che permetta di effettuare interrogazioni aggregate su un dataset contenente informazioni di dettaglio su un gran numero di voli effettuati sul territorio americano nell'anno 2013. Il dataset in particolare è stato estratto dal *Reporting Carrier On-Time Performance (1987-present)* e contiene 6.369.482 righe e 110 colonne che provvedono a fornire una panoramica completa su qualunque volo: dai luoghi di partenza e arrivo ai ritardi, dalle cancellazioni alle loro cause, fino ai dirottamenti e agli stati o città coinvolti. Dando una rapida occhiata al dataset si nota subito come non necessiti di particolari operazioni di pre-processing dal momento che quasi tutte le righe sono complete e i valori in un formato adatto per l'elaborazione. Questo ha ridotto il tempo necessario per operazioni preliminari di pre-processing, consentendoci di focalizzarci maggiormente sull'analisi dei dati e sulla costruzione dell'applicazione.

1.2 Tecnologie utilizzate

A livello di back-end per gestire il grande quantitativo di dati a nostra disposizione e per effettuare query in maniera efficiente sugli stessi abbiamo utilizzato Spark, implementando le operazioni in Python attraverso la libreria PySpark. Questo framework ci ha permesso di sfruttare moduli avanzati come `pyspark.sql` per la manipolazione dei dati e `pyspark.ml` per eventuali operazioni di machine learning. Per ciò che concerne il frontend invece, abbiamo cercato di mantenere continuità con l'ambiente usato per il back-end affidandoci a Streamlit, un framework open-source che consente il rapido sviluppo di applicazioni web-based in Python, particolarmente utile a Data Scientist e Machine Learning Engineers per esporre le loro analisi e creazioni. Grazie a Streamlit l'interfaccia creata risulta particolarmente intuitiva e soprattutto interattiva, permettendo all'utente di interfacciarsi con essa in maniera rapida ed agevole. La visualizzazione dei dati è inoltre semplificata dalle numerose strutture e i molteplici widgets messi a disposizione. Infine il caricamento delle varie schermate è reso efficiente dai meccanismi di caching che abbiamo sfruttato.

1.3 Struttura del progetto

Per garantire modularità e scalabilità del progetto, abbiamo separato il codice relativo a backend e frontend. In particolare Il backend gestisce il carico computazionale ed è composto da due file:

- **queries.py**: in cui sono presenti i metodi chiamati dal frontend che hanno lo scopo di lavorare sul dataset estraendone informazioni e restituendole.

- **utils.py**: contenente funzioni di supporto che vengono utilizzate dalle varie classi

Per quanto riguarda il frontend invece, la gui consta di sei schermate in cui l'utente può visualizzare i dati ed interagirci.

Ognuna di esse è rappresentata dall'omonimo file:

- Home.py
- Aeroporti.py
- Compagnie.py
- Paesi.py
- Rotte.py
- AI.py

In particolare oltre Home.py gli altri file sono contenuti in una cartella pages, come da prassi Streamlit, per garantire la corrispondenza di un endpoint specifico per ognuno di essi.

1.4 Metodologia di lavoro

Abbiamo adottato un approccio iterativo per lo sviluppo del progetto, suddividendo il lavoro in fasi:

1. Analisi del dataset e di eventuale materiale presente in letteratura che utilizzasse lo stesso.
2. Pianificazione delle funzionalità principali.
3. Implementazione del backend con focus sull'efficienza delle query.
4. Sviluppo del frontend con particolare attenzione alla user experience.
5. Test delle funzionalità, ottimizzazione delle prestazioni e correzione di eventuali bug.

Le sezioni successive di questa relazione forniranno un'analisi dettagliata di tutti gli aspetti introdotti in questa sezione iniziale.

2 Studio della letteratura

Prima di iniziare col lavoro vero e proprio abbiamo fatto un'analisi della letteratura per capire se esistessero lavori con dataset simili al nostro e quali fossero le principali queries fatte. Il dataset in questione è stato ampiamente utilizzato in vari contesti accademici e di ricerca, in particolare in studi focalizzati sulla previsione dei ritardi dei voli, l'analisi delle prestazioni e l'efficienza operativa nel settore dell'aviazione.

2.1 Ricerche Precedenti e Applicazioni

Uno dei lavori seminali in questo ambito è stato condotto da [1], che ha sviluppato un modello predittivo per i ritardi del traffico aereo utilizzando questo dataset. La loro analisi si è concentrata principalmente sui ritardi legati alle condizioni meteorologiche e ha utilizzato tecniche di clustering per identificare modelli nella propagazione dei ritardi attraverso la rete di trasporto aereo statunitense. Successivamente [2] ha eseguito un'analisi approfondita dei ritardi dei voli utilizzando sia metodi statistici tradizionali che approcci di machine learning. Le loro query si sono concentrate in particolare su:

- Correlazione tra ritardi in partenza e ritardi in arrivo
- Impatto delle diverse cause di ritardo (vettore, meteo, NAS, sicurezza e ritardo dell'aeromobile)
- Modelli stagionali nell'occorrenza dei ritardi

Uno studio completo di [3] ha utilizzato questo dataset per sviluppare un framework di analisi comparativa delle prestazioni delle compagnie aeree. Le loro principali query SQL si sono concentrate su:

```
SELECT Origin, Dest,  
       AVG(ArrDelay) as ritardo_medio,  
       COUNT(*) as numero_voli  
FROM voli  
GROUP BY Origin, Dest  
HAVING COUNT(*) > 1000  
ORDER BY ritardo_medio DESC
```

Un lavoro più recente di [5] ha implementato tecniche di deep learning utilizzando questo dataset per prevedere i ritardi dei voli. Le loro query di preprocessing hanno incluso operazioni estese di pulizia dei dati e feature engineering:

```
SELECT v.*, m.temperatura, m.visibilita  
FROM voli v
```

```
JOIN dati_meteo m
ON v.origine = m.aeroporto
AND v.DataVolo = m.data
WHERE v.Cancellato = 0
AND v.Deviato = 0
```

2.2 Pattern di Analisi Comuni

Da questi studi emergono diversi pattern analitici comuni:

1. **Classificazione dei Ritardi:** La maggior parte degli studi categorizza i ritardi in base alle loro cause (meteo, NAS, sicurezza, ritardo dell'aeromobile) per comprendere i principali fattori che influenzano le prestazioni delle compagnie aeree.
2. **Analisi di Rete:** I ricercatori analizzano frequentemente la relazione tra coppie di aeroporti e come i ritardi si propagano attraverso la rete.
3. **Pattern Temporali:** Molti studi si concentrano sull'identificazione di tendenze stagionali e pattern giornalieri nei ritardi dei voli.
4. **Performance dei Vettori:** L'analisi comparativa delle prestazioni di diversi vettori in varie condizioni è un tema comune.

2.3 Approcci Metodologici

Gli approcci metodologici in questi studi tipicamente coinvolgono:

- Analisi statistica dei pattern di ritardo e delle correlazioni
- Modelli di machine learning per la previsione dei ritardi
- Tecniche di analisi di rete per comprendere la propagazione dei ritardi
- Analisi delle serie temporali per l'identificazione di pattern temporali

Questi studi dimostrano la versatilità del dataset nella comprensione e nell'analisi di vari aspetti delle operazioni e delle prestazioni delle compagnie aeree, degli aeroporti e degli stati. Basandoci su questo abbiamo cercato di impostare il nostro lavoro cercando di portare avanti le nostre analisi.

3 Sezione Home

3.1 Passi Iniziali

Prima di capire nel dettaglio la struttura e le funzionalità offerte dalla sezione home, verranno analizzate le istruzioni relative all'inizializzazione della *SparkSession*, al caricamento del dataset ed all'analisi preliminare sul dataset.

```
spark=SparkSession.builder.master("local[*]").appName("Progetto BigData.com").config("spark.driver.memory", "8g").getOrCreate()
spark.sparkContext.setLogLevel("OFF")
```

SparkSession È stata inizializzata una *SparkSession* configurata con i seguenti parametri principali:

- **Master:** configurato come "local[*]" per sfruttare tutti i core disponibili nella macchina locale.
- **Configurazione della memoria:** è stato specificato che il driver Spark può utilizzare fino a 8GB di memoria.

Inoltre per evitare che log inutili riempissero la console è stato settato il livello di log su "OFF".

```
folder_path = r"C:\Users\giuse\Desktop\UNIVERSITA'\MAGISTRALE\1° ANNO\1° SEMESTRE\MODELLI E TECNICHE PER BIG DATA\PROGETTO\DATI"
file_list = [os.path.join(folder_path, file) for file in os.listdir(folder_path) if file.endswith('.csv')]

df = spark.read.options(delimiter=',').csv(file_list, header=True, inferSchema=True).cache()
```

Caricamento dei dati I dati utilizzati sono stati forniti in formato CSV e distribuiti su più file. Questi file sono stati caricati dinamicamente dalla directory specifica dove si trovavano utilizzando un filtro per l'estensione .csv. È stato utilizzato il metodo read di Spark per caricare i file in un unico dataframe pyspark, i parametri utilizzati permettono di:

- **delimiter=,:** per specificare come ogni colonna sia separata da una virgola dalle successive
- **header=True:** in modo che vengano riconosciuti automaticamente i nomi delle colonne inserite come header nei vari file excel
- **inferSchema=True:** attivato per dedurre automaticamente i tipi di dati evitando di dover fare cast nelle fase successive.

```
|-- Year: integer (nullable = true)
|-- Quarter: integer (nullable = true)
|-- Month: integer (nullable = true)
|-- DayOfMonth: integer (nullable = true)
|-- DayOfWeek: integer (nullable = true)
|-- FlightDate: date (nullable = true)
|-- Reporting_Airline: string (nullable = true)
|-- DOT_ID_Reporting_Airline: integer (nullable = true)
|-- IATA_CODE_Reporting_Airline: string (nullable = true)
|-- Tail_Number: string (nullable = true)
|-- Flight_Number_Reporting_Airline: integer (nullable = true)
|-- OriginAirportID: integer (nullable = true)
```

Analisi preliminare sul dataset Python inferisce questo schema che viene stampato con l'istruzione `df.printSchema()`, e mostra per ogni colonna il tipo inferito. Si noti come nell'immagine non sono riportate tutte le colonne per non essere troppo prolissi. Analizzando questo schema salta all'occhio una colonna finale `"c109"` inferita erroneamente da python a causa di una virgola finale in ogni riga del dataset. Abbiamo quindi provveduto a *droppare* questa colonna per evitare di appesantire un dataset di per sé già pesante.

```
|-- _c109: string (nullable = true)
```

```
df = spark.read.options(delimiter=',').csv(file_list, header=True, inferSchema=True).drop("_c109").cache()
```

Si noti inoltre che dati sono stati memorizzati in cache per migliorare le prestazioni nelle analisi successive. Questa operazione consente di calcolare anticipatamente i dati, mantenendoli in memoria per un accesso più rapido nelle fasi successive. Utilizzando la cache, si fa affidamento al principio della lazy evaluation il quale prevede che le trasformazioni su un DataFrame vengano effettivamente eseguite solo quando viene invocata un'azione, ottimizzando così l'uso delle risorse quando non è strettamente necessario elaborare i dati immediatamente.

Per garantire la qualità dei dati, sono state condotte alcune analisi preliminari per identificare eventuali problemi o anomalie:

- **Controllo delle Righe Duplicate:** utilizzando una query basata su Spark SQL, è stato verificato il numero di righe duplicate all'interno del dataset che risulta essere pari a 0.
- **Identificazione delle Anomalie nei Dati:** sono state definite condizioni specifiche per individuare anomalie nelle colonne principali, ad esempio distanze e tempi di volo non positivi, disallineamenti tra i ritardi indicati in colonne correlate e molto altro. Ogni condizione è stata applicata ai dati per contare il numero di anomalie rilevate, che anche in questo caso risultano pari a 0.

3.2 GUI sezione Home

Di seguito verrà illustrata la *Graphic User Interface* della sezione Home chiarendo cosa rappresenti ogni componente e concentrandoci sugli aspetti principali relativi al codice. Nella schermata Home come in tutte le altre si è prestata particolare attenzione non solo all'efficienza delle queries e della GUI, ma anche a garantire una User Experience di alto livello che dia all'utente la possibilità di ottenere le informazioni principali dal dataset in maniera visiva, visualizzandole in maniera agevole tramite mappe geografiche, grafici con progress bar e widgets moderni.

Dashboard Voli - Panoramica

Anteprima Dataset

Voli totali

6369482

Aereo con più miglia percorse

N751AT

Giorno della settimana con più voli:

Venerdì - 952033

Percentuali

In orario



In ritardo

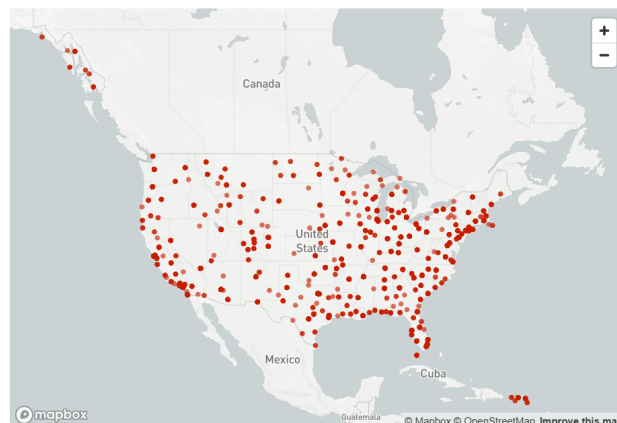
Mappa

Aeroporti di partenza

Aeroporti di destinazione

Aeroporti coinvolti

Ogni punto sulla mappa rappresenta un aeroporto da cui è partito un aereo



Stati più visitati

Stato	Voli atterrati
California	755926
Texas	745515
Florida	443169
Georgia	416251
Illinois	414827
New York	289544
Colorado	250555
North Carolina	221079
Arizona	201509
Michigan	199349

Top Rotte Aeree

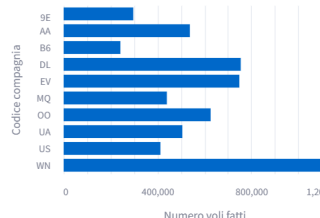
Partenza	Arrivo	NumeroVoli
SFO	LAX	1583
LAX	SFO	1579
OGG	HNL	1180

In ritardo

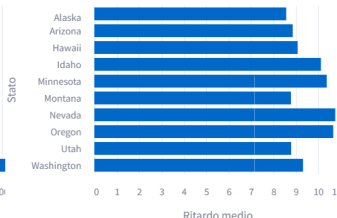


Grafici

Principali compagnie per voli fatti



Stati più puntuali

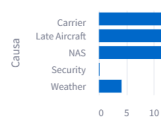


OGG	HNL	1180
HNL	OGG	1174
LAX	LAS	1156
LAS	LAX	1152
LAX	JFK	1126
JFK	LAX	1126
LGA	ATL	1026
ATL	LGA	1025

Compagnie con più miglia percorse

Compagnia	Miglia percorse
WN	782392393mi
DL	663448972mi
UA	656997085mi
AA	568996515mi
US	361588400mi
EV	357814587mi
OO	288530971mi
B6	258125810mi
MQ	218186501mi
AS	182564626mi

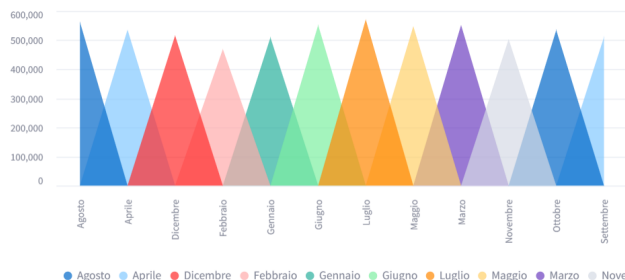
Principali cause ritardo



Ritardi medi per stagione



Numero di voli per mese



Questa prima sezione offre una visione generale sulle informazioni principali estratte dal dataset. La pagina è stata divisa in 3 colonne, ognuna contenente informazioni specifiche.

Colonna a sinistra Da informazioni riguardo il numero di voli totali contenuti nel dataset, le percentuali di voli in orario e ritardo, il TailNumber dell'aereo con più km percorsi e il giorno della settimana in cui sono stati effettuati più voli. Per modellare queste info sono state usate query basilari come:

```
1 @st.cache_data(show_spinner=False)
2 def conta_righe_totali():
3     return df.count()
```

In questa query come in molte altre utilizzate durante il progetto, è presente il decoratore `@st.cache_data` che Streamlit mette a disposizione per ottimizzare le prestazioni

garantendo il caching delle informazioni restituite dalla query, Streamlit verifica due aspetti: i valori dei parametri di input (se presenti) e il codice all'interno della funzione. Se è la prima volta che Streamlit rileva questi valori, esegue la funzione e memorizza il valore restituito nella cache. La volta successiva che la funzione viene chiamata con gli stessi parametri e lo stesso codice (ad esempio, quando un utente cambia sezione e poi ci torna), Streamlit salta l'esecuzione della funzione e restituisce direttamente il valore memorizzato nella cache. Un altro aspetto importante è legato al fatto che Streamlit lavora meglio con DataFrame di tipo pandas, perciò dal momento in cui le queries restituiscono dataframe di tipo spark (questo per garantire scalabilità ed evitare di legarci ad aspetti del frontend), c'è bisogno di effettuare una conversione tramite il metodo `spark_to_pandas` contenuto nel file `utils.py`, un esempio è:

```
1 giorno_piu_voli=spark_to_pandas(  
    giorno_della_settimana_con_piu_voli()  
2 giorno_num=giorno_piu_voli.iloc[0]["DayOfWeek"]  
3 numero_voli=giorno_piu_voli.iloc[0]["count"]
```

dove il risultato del metodo `giorno_della_settimana_con_piu_voli` viene convertito col suddetto metodo.

```
1 def spark_to_pandas(spark_df):  
2     return spark_df.toPandas()
```

Le percentuali dei voli in orario e in ritardo vengono raffigurate in una *donut* colorata, il codice di quest'ultima è stato reperito da una repository GitHub riguardante Streamlit, dal momento che le librerie di base non mettono a disposizione un widget del genere.

Colonna Centrale Questa sezione della Home ha l'obiettivo di mostrare graficamente informazioni rilevanti sui voli. Nella parte in alto è presente una mappa geografica contenente dei pin rappresentanti gli aeroporti del dataset, ¹ è possibile zoomare la mappa e vedere nel dettaglio ogni aeroporto. Dal momento che nel dataset non sono presenti latitudine e longitudine dei singoli aeroporti ma solo codice di riferimento e città di appartenenza, si è fatto uso della libreria `geopy` di python, in particolare del modulo `Nominatim`. `Nominatim` è un servizio geocodifica fornito da `OpenStreetMap` (OSM) che consente di trasformare indirizzi o descrizioni di luoghi in coordinate geografiche e viceversa. Sfruttando quindi il codice dell'aeroporto (e successivamente il nome della città se la ricerca per codice desse esito negativo) viene inviata una richiesta http diretta al servizio che restituisce le coordinate. Tuttavia il server limita le richieste che è possibile inviare al secondo, il che si traduce con l'inserimento di `sleep` fra le varie chiamate. Perciò per evitare che questa funzionalità richiedesse troppo tempo si è pensato di non

¹In particolare è possibile scegliere se visualizzare solo quelli da cui è partito un volo, in cui è atterrato uno o entrambi

eseguirli a runtime ma preliminarmente salvare le coordinate degli aeroporti di origine e destinazione dei vari voli in un file .csv in locale da leggere tramite la libreria pandas per graficare le coordinate. Una delle tre mappe verrà quindi generata come segue:

```
1 dfTem = pd.read_csv(coordinateAeroporti, delimiter=",")
2 dfMap1 = pd.DataFrame()
3 dfMap1 = dfTem[['OriginLat', 'OriginLon']]
4 dfMap1.rename(columns={'OriginLat': 'LAT', 'OriginLon': 'LON'},
5               inplace=True)
6 st.markdown("Ogni punto sulla mappa rappresenta un aeroporto da cui e' partito un aereo")
7 st.map(data=dfMap1, zoom=2)
```

La sezione relativa ai grafici offre una panoramica generale su compagnie, ritardi e stati. Sono stati utilizzati i bar_chart offerti da Streamlit che prendono in input i dataframe pandas ottenuti a partire dalle rispettive queries. Ad esempio, per graficare le prime 10 compagnie per voli fatti la query è:

```
1 def compagnie_piu_voli_fatti():
2     compagnie=df.groupby("Reporting_Airline").count().orderBy(col
3         ("count").desc()).limit(10)
4     return compagnie
```

che effettua una groupBy sulla colonna relativa al codice della compagnia aerea, conta tutte le righe ottenute e le ordina in maniera decrescente prendendo e restituendo le prime 10 righe. Il risultato viene poi convertito in un df pandas e plottato come segue:

```
1 dati_compagnie=spark_to_pandas(compagnie_piu_voli_fatti())
2 dati_compagnie = dati_compagnie.set_index("Reporting_Airline")
3 st.subheader("Grafici", divider="red")
4 st.markdown('#### Principali compagnie per voli fatti')
5 st.bar_chart(data=dati_compagnie, x=None, y=None, color=None, x_label
6             = "Numero voli fatti", y_label="Codice compagnia", horizontal=
7             True, use_container_width=False)
```

Impostandola la colonna contenente i codici delle compagnie come indice, si facilita la visualizzazione dei dati in una rappresentazione grafica come il grafico a barre. Infatti, quando si crea il grafico, l'indice può essere automaticamente utilizzato come etichetta per le categorie (asse verticale in un grafico a barre orizzontale come in questo caso). Questo avviene in maniera totalmente analoga per gli altri grafici contenuti in questa colonna.

Colonna a destra Quest'ultima sezione della pagina mostra tre tabelle, la prima contenente i dati degli stati più visitati (cioè presenti più volte nel dataset come destinazione di un volo) è così ottenuta:

```

1 def stati_piu_visitati():
2     stati=stati_filtrati.groupBy("DestStateName").count().orderBy(
3         col("count").desc()).limit(10)
4     return stati

```

La query raggruppa le righe per stato, le conta ordinandole in senso decrescente e prendendo solo le prime 10 righe.

```

1 dataframe_query2=spark_to_pandas(stati_piu_visitati())
2 st.dataframe(dataframe_query2,column_order=("DestStateName", "
3     count"),hide_index=True,width=None,
4     column_config={
5         "DestStateName": st.column_config.TextColumn("
6             Stato"),,
7         "count": st.column_config.ProgressColumn("Voli
8             atterrati",help="Numero di voli atterrati
9             nello stato in relazione ai totali",format="%
10             f",min_value=0,max_value=conta_righe_totali())
11     })

```

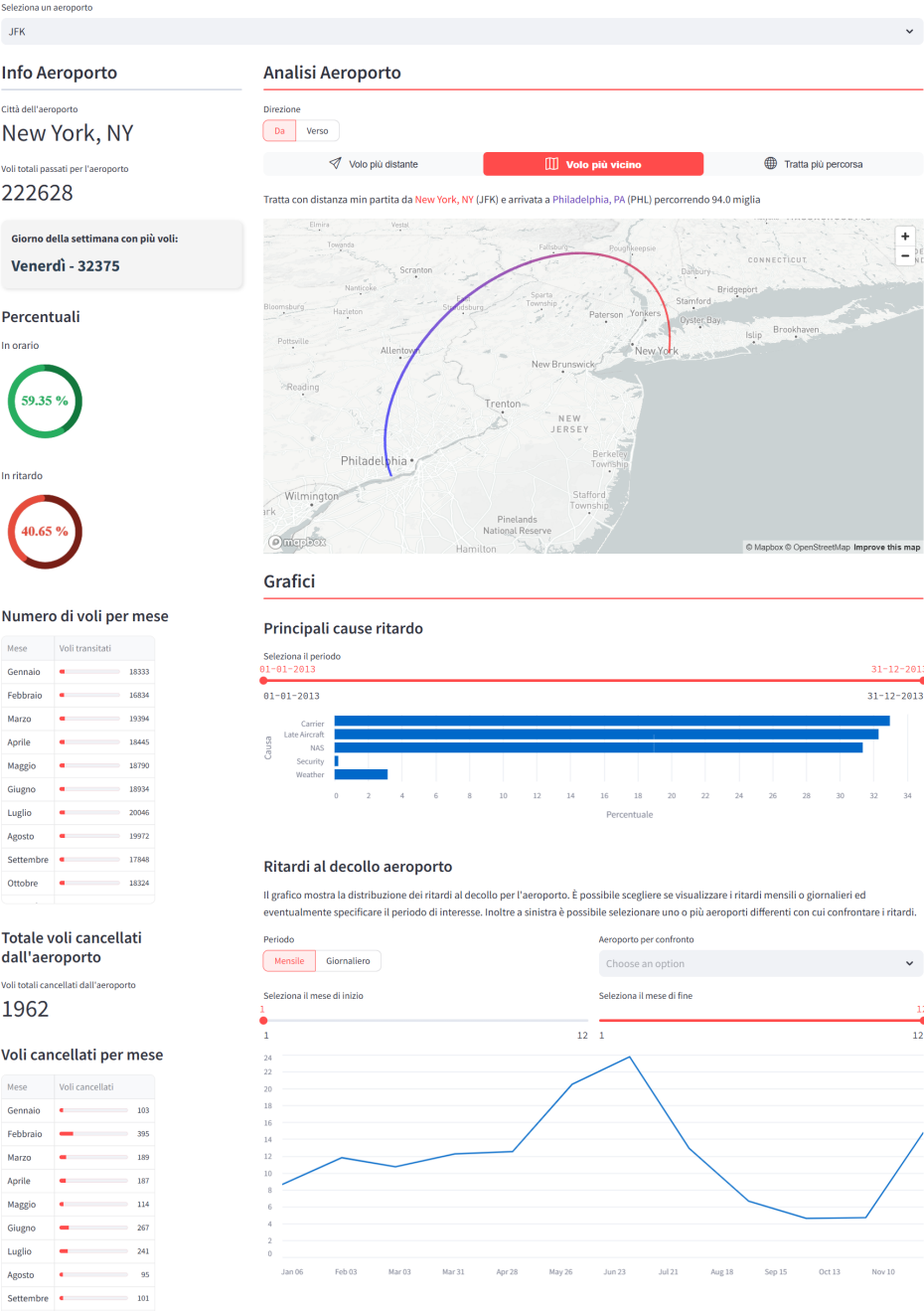
La tabella viene creata con l'istruzione `st.dataframe` specificando il dataframe da cui prendere i dati, l'ordine delle colonne nello stesso, di nascondere l'indice della tabella e la configurazione per ogni colonna. In particolare le colonne della tabella saranno due:

- Quella associata alla colonna `DestStateName` rappresentata con il nome di Stato
- Quella associata al contatore con il nome di Voli atterrati a cui è associata una progress bar che mostra la relazione tra il numero di voli atterrati in quello stato rispetto a tutti quelli effettuati


Discorso analogo per la tabella *Top Rotte Aeree e Compagnie con più km percorsi*.


4 Sezione Aeroporti


Analisi dei dati relativi ai singoli aeroporti




Operatività nei giorni più importanti dell'anno

 **Voli partiti**

 Voli atterrati

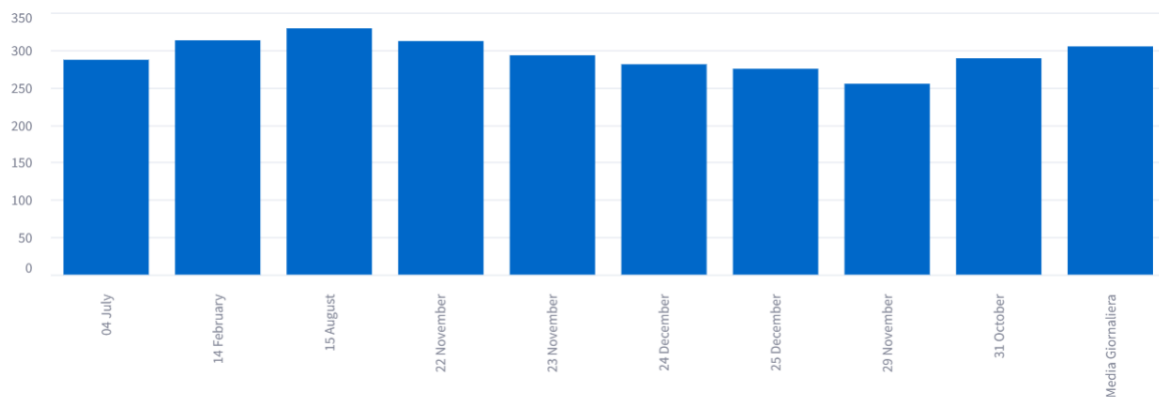
 Voli totali

 Percentuale Cancellazioni

Stato per confronto

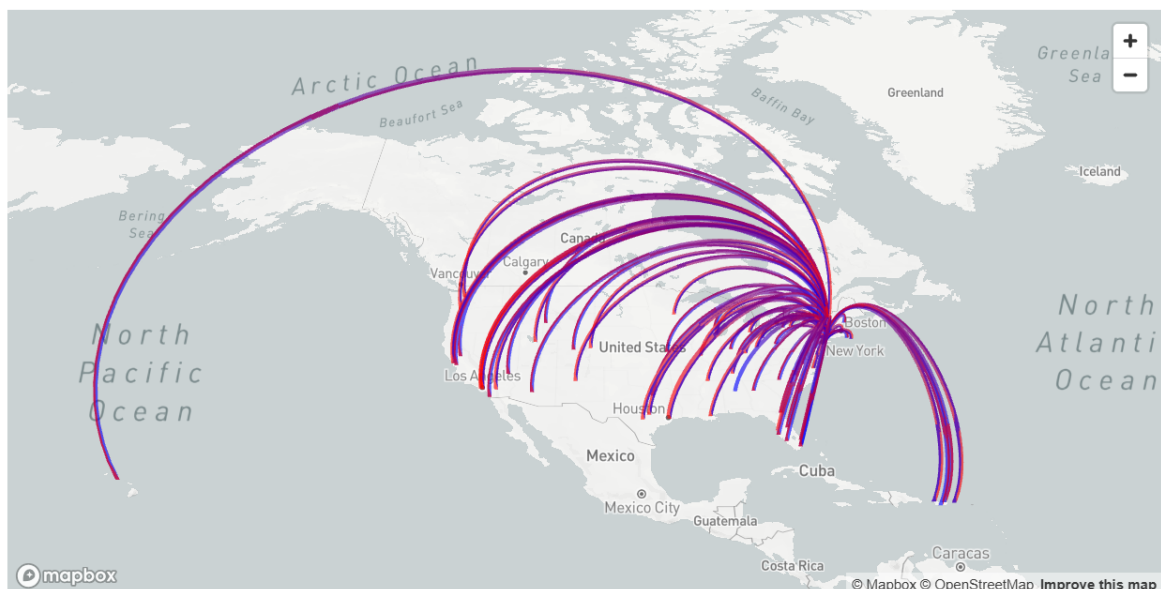
Choose an option

Il grafico mostra il numero di voli partiti dall'aeroporto nelle date principali dell'anno. L'ultima barra rappresenta la media giornaliera dei voli per ogni aeroporto, permettendo un confronto immediato tra l'operatività normale e quella nei giorni speciali.



Aeroporti collegati

Il grafico mostra le rotte aeree che collegano l'aeroporto selezionato ad altri aeroporti



Questa sezione, come intuibile dal nome, permette di ottenere qualunque informazione utile ottenibile dal dataset sui singoli aeroporti. Grazie ad un drop menù è possibile selezionare uno specifico aeroporto di cui visualizzarne le info, è stata perciò scritta una query che ottiene i codici dei vari aeroporti e si è fatto uso del caching streamlit per garantire che il risultato venisse memorizzato essendo una funzione richiamata più volte.

```
1 def codici_aeroporti():
2     aeroporti_origine = df.select("Origin").distinct()
3     aeroporti_destinazione = df.select("Dest").distinct()
4     tutti = aeroporti_origine.union(aeroporti_destinazione).
5         distinct()
6     return tutti
7
8 @st.cache_data(show_spinner=False)
9 def load_airport_codes():
10     return spark_to_pandas(codici_aeroporti())
11
12 aeroporto_selezionato = st.selectbox("Seleziona un aeroporto",
13     load_airport_codes())
```

La query seleziona i codici distinti degli aeroporti di origine e destinazione e provvede ad unirli. Anche in questo caso a livello di frontend abbiamo diviso la pagina in due colonne di dimensione differente e contenenti tipi di informazioni diverse. In quella a sinistra oltre a paese e stato dell'aeroporto è possibile visualizzare tabelle relative a voli e cancellazioni suddivise per mese². La colonna più grande contiene invece sezioni con le quali l'utente può interagire per visualizzare info specifiche e fare confronti tra i vari aeroporti. La mappa geografica permette di visualizzare, sia per i voli partiti dall'aeroporto selezionato che per quelli atterrati, le seguenti informazioni:

- Volo più distante effettuato
- Volo meno distante effettuato
- Volo effettuato più spesso

Mappa con info sulle tratte Vediamo ad esempio il caso in cui si vuole ottenere la mappa raffigurante la tratta più distante partita dall'aeroporto scelto, la query che trova il volo è la seguente:

```
1 def volo_distanza_max_da_aeroporto(aeroporto):
2     volo=df.filter((col("Origin")==aeroporto)).orderBy(col("
3         Distance").desc()).limit(1)
```

²Anche per queste tabelle si è seguita l'impostazione citata nella sezione Home


```

3         return volo.select(col("Origin"),col("OriginCityName"),col("
        Dest"),col("DestCityName"),col("Distance"))

```

Il risultato viene convertito in df pandas e si estraggono le colonne da usare per disegnare l'arco sulla mappa con le specifiche funzioni:

```

1 volo_distanza_max_da=spark_to_pandas(
    volo_distanza_max_da_aeroporto(aeroporto_selezionato))
2 citta_partenza=volo_distanza_max_da.iloc[0]["OriginCityName"]
3 partenza_code=volo_distanza_max_da.iloc[0]["Origin"]
4 citta_arrivo=volo_distanza_max_da.iloc[0]["DestCityName"]
5 dest_code=volo_distanza_max_da.iloc[0]["Dest"]
6 coordinate_df=load_coordinate_data()
7 tratte_coords = get_coordinates(volo_distanza_max_da,
    coordinate_df)
8 st.markdown(f"Tratta con distanza max partita da :red[{
    citta_partenza}] ({partenza_code}) e arrivata a :violet[{
    citta_arrivo}] ({dest_code}) percorrendo {volo_distanza_max_da
    .iloc[0]['Distance']} miglia")
9 disegna_tratta(tratte_coords,colonne_tab3[1])

```

Anche in questo caso per disegnare l'arco associato alla rotta c'è bisogno di avere a disposizione latitudine e longitudine dell'aeroporto di partenza e di arrivo del volo. Per evitare l'overhead dovuto alle sleep nel fare le richieste tramite Nominatim, si è creato un altro file .csv salvato in locale ³, dove ad ogni aeroporto sono associate le coordinate, con questo script python eseguito a parte:

```

1 # Filtro le colonne necessarie
2 aeroporti = data[["Origin", "OriginCityName"]].drop_duplicates()
3
4 # Inizializzo il geolocator
5 locator = Nominatim(user_agent="myNewGeocoder")
6
7 def get_coordinate(origin, nome_citta):
8     try:
9         trovato = locator.geocode(f"{origin} Airport {nome_citta}
            ", timeout=None)
10        if trovato is None:
11            time.sleep(1)
12            trovato = locator.geocode(nome_citta, timeout=None)
13        if trovato:
14            return trovato.latitude, trovato.longitude

```

³quello usato in precedenza non era utilizzabile perche riportava le coordinate degli aeroporti di partenza e destinazione delle varie tratte senza specificare quali fossero

```

15     except Exception as e:
16         print(f"Errore durante la ricerca per {origin}, {
            nome_citta}: {e}")
17     return None, None
18
19     # Aggiungo le colonne per latitudine e longitudine
20     latitudes = []
21     longitudes = []
22
23     for _, row in aeroporti.iterrows():
24         origin = row["Origin"]
25         nome_citta = row["OriginCityName"]
26         lat, lon = get_coordinate(origin, nome_citta)
27         latitudes.append(lat)
28         longitudes.append(lon)
29
30     aeroporti["OriginLat"] = latitudes
31     aeroporti["OriginLon"] = longitudes
32
33     # Salvo il risultato in un file CSV
34     output_path = "C:/Users/xdomy/Desktop/Universita/MAGISTRALE/1
        Anno 1 Semestre/Modelli e Tecniche per Big Data/Progetto Voli/
        ProgettoBigData/output_airports.csv"
35     aeroporti.to_csv(output_path, index=False)

```

Tornando al disegno della tratta sulla mappa, dopo aver chiamato la query ed ottenuto le principali info dalla stessa, chiamiamo la funzione `load_coordinate_data()` che viene messa in cache streamlit e legge dal file csv appena creato.

```

1 @st.cache_data(show_spinner=False)
2 def load_coordinate_data():
3     return pd.read_csv(coordinate_aeroporto)

```

La funzione `get_coordinates(volo.distanza_max.da, coordinate_df)` del file `utils.py` riceve una lista di tratte (in questo caso una sola) ottenute con le query e il file con le coordinate, legge da quest'ultimo e restituisce le coordinate degli aeroporti coinvolti nella tratta:

```

1 @st.cache_data(show_spinner=False)
2 def get_coordinates(tratte_pd, coordinate_df):
3     lista_coordinate = []
4     for _, row in tratte_pd.iterrows():
5         origin_coords = coordinate_df[coordinate_df["Origin"] ==
            row["Origin"]]

```

```

6         dest_coords = coordinate_df[coordinate_df["Origin"] ==
          row["Dest"]]
7     if not origin_coords.empty and not dest_coords.empty:
8         coords = {"OriginLatitude": origin_coords.iloc[0]["
          OriginLat"],
9                     "OriginLongitude": origin_coords.iloc
10                    [0]["OriginLon"],
11                    "DestLatitude": dest_coords.iloc[0]["
          OriginLat"],
12                    "DestLongitude": dest_coords.iloc[0][
          "OriginLon"],
13                    "outbound": 1 # Valore per la
          larghezza dell'arco
14                }
15        lista_coordinate.append(coords)
16    else:
17        if origin_coords.empty:
18            st.warning(f"Coordinate mancanti per l'origine: {
19                row['Origin']}")
20        if dest_coords.empty:
21            st.warning(f"Coordinate mancanti per la
22                destinazione: {row['Dest']}")
23    return pd.DataFrame(lista_coordinate)

```

la funzione per ogni tratta ricevuta trova nel .csv la riga corrispondente, cioè quella dove l'aeroporto nella colonna "Origin" è uguale a quello della tratta (sia per l'origine che per la destinazione), prende le coordinate e le inserisce in una lista che sarà poi restituita come dataframe pandas. Infine la funzione `disegna_tratta(tratte_coords,colonne_tab3[1])` riceve le coordinate degli archi (cioè le rotte, in questo caso una sola) da disegnare sulla mappa e la colonna della GUI su cui si trova la mappa e provvede a disegnare l'arco corrispondente alla rotta:

```

1 def disegna_tratta(coord_df,colonna_dove_disegnare):
2     colonna_dove_disegnare.pydeck_chart(
3         pdk.Deck(
4             map_style="mapbox://styles/mapbox/light-
          v10",
5             initial_view_state=pdk.ViewState(
6                 latitude=coord_df.iloc[0]["
          OriginLatitude"],
7                 longitude=coord_df.iloc[0]["
          OriginLongitude"],
8                 zoom=5,
9                 pitch=50,

```

```

10         height=250,
11     ),
12     layers=[
13         pdk.Layer(
14             "ArcLayer",
15             data=coord_df,
16             get_source_position=["
17                 OriginLongitude", "
18                 OriginLatitude"],
19             get_target_position=["
20                 DestLongitude", "DestLatitude"
21             ],
22             get_source_color=[255, 0, 0,
23                             160],
24             get_target_color=[0, 0, 255,
25                             160],
26             auto_highlight=True,
27             width_scale=0.0001,
28             get_width="outbound",
29             width_min_pixels=3,
30             width_max_pixels=30,
31
32             radius=200,
33             elevation_scale=4,
34             elevation_range=[0, 1000],
35             pickable=True,
36             extruded=True,
37         ),
38     ],
39 )

```

vengono impostati il tipo di mappa, le coordinate del punto in cui centrare la mappa e successivamente le info sull'arco da disegnare, cioè le due coordinate il colore ed il raggio tra tutte.

Grafico cause ritardi Il grafico sotto la mappa mostra quelle che sono le principali cause di ritardo per l'aeroporto selezionato. In particolare viene data all'utente la possibilità di selezionare un periodo specifico in cui fare indagine grazie ad uno slider:

```

1 selezione_data_cause_ritardi=st.slider("Seleziona il periodo",
    value=[datetime(2013,1,1),datetime(2013,12,31)],format="DD-MM-
    YYYY",min_value=datetime(2013,1,1),max_value=datetime
    (2013,12,31))

```

```

2 percentuali_ritardo_aeroporto=spark_to_pandas(
    percentuali_cause_ritardo(data_inizio=
        selezione_data_cause_ritardi[0].date(),data_fine=
        selezione_data_cause_ritardi[1].date()))
3 st.bar_chart(data=percentuali_ritardo_aeroporto,x=None,y=None,
    color=None,x_label="Percentuale",y_label="Causa",horizontal=
    True,use_container_width=True)

```

Lo slider setta date minime e massime selezionabili, inizialmente impostiamo il periodo selezionato a tutto l'anno. La query che restituisce le percentuali per i vari ritardi riceve le date selezionate tramite lo slider e opera in questo modo:

```

1 def percentuali_cause_ritardo(filtro_compagnia=None,
    causa_specifica=None, data_inizio=None, data_fine=None, stato=
    None, aeroporto=None):
2     df_filtrato = df
3     if aeroporto:
4         df_filtrato = df_filtrato.filter((col("Origin")==
            aeroporto))
5     if stato:
6         df_filtrato=df_filtrato.filter((col("OriginStateName")==
            stato) | (col("DestStateName")==stato))
7     if filtro_compagnia:
8         df_filtrato = df_filtrato.filter(col("Reporting_Airline")
            == filtro_compagnia)
9     if data_inizio and data_fine:
10        df_filtrato = df_filtrato.filter((col("FlightDate") >=
            data_inizio) & (col("FlightDate") <= data_fine))
11
12    df_filtrato = df_filtrato.fillna(0, subset=["CarrierDelay", "
        WeatherDelay", "NASDelay", "SecurityDelay", "
        LateAircraftDelay"])
13
14    ritardi_cause = df_filtrato.select(
15        ["CarrierDelay", "WeatherDelay", "NASDelay", "
            SecurityDelay", "LateAircraftDelay"]
16    ).agg(
17        sum("CarrierDelay").alias("CarrierDelay"),
18        sum("WeatherDelay").alias("WeatherDelay"),
19        sum("NASDelay").alias("NASDelay"),
20        sum("SecurityDelay").alias("SecurityDelay"),
21        sum("LateAircraftDelay").alias("LateAircraftDelay")
22    )
23

```

```

24     ritardi_somma = ritardi_cause.select(
25         (col("CarrierDelay") +
26          col("WeatherDelay") +
27          col("NASDelay") +
28          col("SecurityDelay") +
29          col("LateAircraftDelay")).alias("TotaleRitardo")
30     ).collect()[0]["TotaleRitardo"]
31
32     if causa_specifica:
33         ritardi_percentuali = ritardi_cause.select(
34             pyspark_round((col(causa_specifica) / ritardi_somma *
35                             100),2).alias(f"{causa_specifica}_Percent")
36         )
37     else:
38         ritardi_percentuali = ritardi_cause.select(
39             pyspark_round((col("CarrierDelay") / ritardi_somma *
40                             100),2).alias("Carrier"),
41             pyspark_round((col("WeatherDelay") / ritardi_somma *
42                             100),2).alias("Weather"),
43             pyspark_round((col("NASDelay") / ritardi_somma * 100),
44                             2).alias("NAS"),
45             pyspark_round((col("SecurityDelay") / ritardi_somma *
46                             100),2).alias("Security"),
47             pyspark_round((col("LateAircraftDelay") /
48                             ritardi_somma * 100),2).alias("Late Aircraft")
49         )
50     return ritardi_percentuali

```

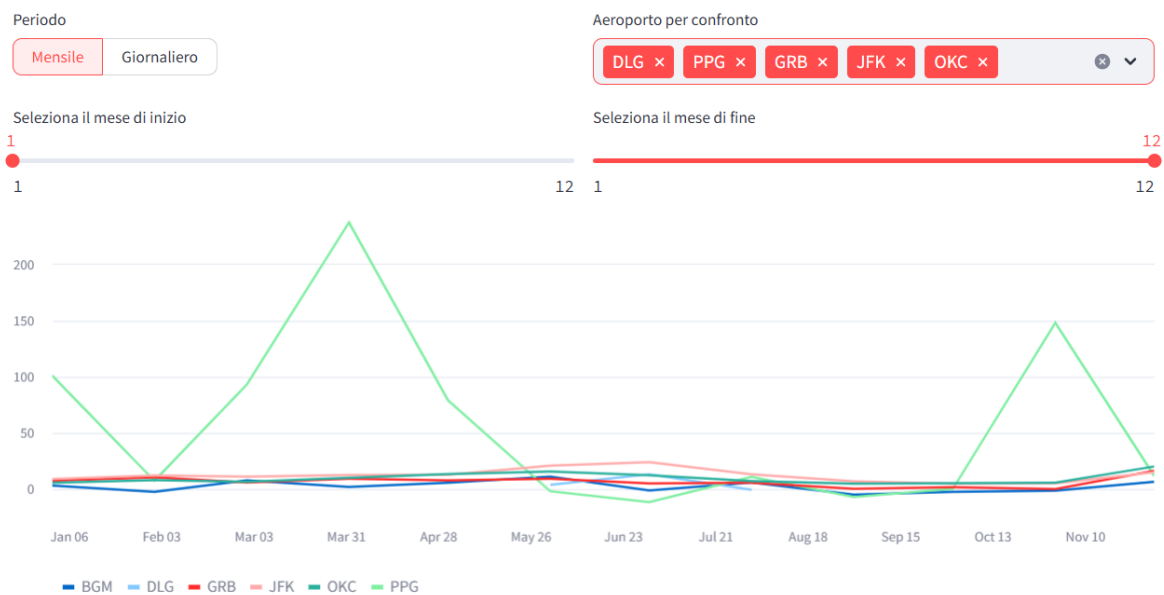
Per questa query come per molte altre d'ora in poi, si fa uso dei parametri opzionali di python, permettendo di eseguire la query anche passando solo alcuni dei parametri opzionali richiesti (in questo caso le date). Dopo aver filtrato il dataset in base ai parametri ricevuti, le colonne relative alle cause dei ritardi vengono riempite con 0 qualora dovessero avere valori mancanti. La funzione somma poi i ritardi per ciascuna causa usando agg che permette di effettuare operazioni di aggregazione sui dati, viene anche calcolata la somma totale di tutti i ritardi da usare per calcolare la percentuale. Infine se non è specificata una causa specifica calcola le percentuali per tutte le cause, altrimenti solo per quella di interesse.

Ritardi al decollo aeroporto In questa sezione vengono plottati i ritardi medi al decollo per ogni aeroporto, la sezione permette di vedere sia la media mensile che quelli giornalieri, in particolare sono messe a disposizione dell'utente due funzionalità specifiche

che consistono nella selezione di un periodo specifico di indagine ⁴ e nella scelta di uno o più aeroporti differenti con cui effettuare un confronto plottando i vari andamenti come segue:

Ritardi al decollo aeroporto

Il grafico mostra la distribuzione dei ritardi al decollo per l'aeroporto. È possibile scegliere se visualizzare i ritardi mensili o giornalieri ed eventualmente specificare il periodo di interesse. Inoltre a sinistra è possibile selezionare uno o più aeroporti differenti con cui confrontare i ritardi.



```

1 dati_aeroporti = {}
2 dati_principale = spark_to_pandas(calcolo_ritardo_per_mesi(
    aeroporto_selezionato, mese_inizio, mese_fine))
3 dati_principale['Data'] = pd.to_datetime(dati_principale['Month',
    ], format='%m').map(lambda d: d.replace(year=2013))
4 dati_principale = dati_principale[['Data', 'ritardo_medio']].
    rename(columns={"ritardo_medio": aeroporto_selezionato})
5 dati_aeroporti[aeroporto_selezionato] = dati_principale
6
7 for aeroporto in confronto:
8     if aeroporto != aeroporto_selezionato:
9         dati_confronto = spark_to_pandas(calcolo_ritardo_per_mesi(
            aeroporto, mese_inizio, mese_fine))

```

⁴il cui funzionamento è analogo a livello di logica a quanto visto sopra per le percentuali delle cause dei ritardi

```

10     dati_confronto['Data'] = pd.to_datetime(dati_confronto['
        Month'], format='%m').map(lambda d: d.replace(year
            =2013))
11     dati_confronto = dati_confronto[['Data', 'ritardo_medio'
        ]].rename(columns={"ritardo_medio": aeroporto})
12     dati_aeroporti[aeroporto] = dati_confronto
13 chart_data = pd.DataFrame()
14 for aeroporto, dati in dati_aeroporti.items():
15     if chart_data.empty:
16         chart_data = dati.set_index("Data")
17     else:
18         chart_data = chart_data.join(dati.set_index("Data"), how=
            "outer")
19 st.line_chart(chart_data)

```

Dalla query, che verrà mostrata in seguito, vengono estratti i ritardi, in particolare creiamo la colonna Data convertendo la colonna Month in oggetti datetime di Pandas, dove il mese è specificato dal valore nella colonna e l'anno è fissato al 2013, il risultato è una colonna Data che rappresenta la data con il mese corretto, ma l'anno fisso a 2013. Vengono poi mantenute solo due colonne rinominando quella 'ritardio medio' con il nome dell'aeroporto selezionato. Nel dizionario che conterrà i dati di tutti gli aeroporti con da plottare (può essere anche solo quello dell'aeroporto corrente se non si vogliono fare confronti) viene inserita una nuova coppia chiave-valore che è proprio quella relativa all'aeroporto scelto. Se nella variabile confronto (cioè il widget multiselect di streamlit) sono contenuti altri aeroporti si procede con la stessa logica inserendo anche questi nel dizionario. Infine viene creato un dataframe, si scorre il dizionario, se chart_data è vuoto (nel primo ciclo), il codice imposta chart_data uguale ai dati dell'aeroporto corrente, ma impostando la colonna Data come indice del DataFrame. Per i cicli successivi viene fatta un'operazione di join tra chart_data e il DataFrame dell'aeroporto corrente, ancora una volta usando la colonna Data come indice. L'operazione di join è di tipo "outer", il che significa che il risultato includerà tutte le date da entrambi i DataFrame, anche se non c'è corrispondenza per alcuni aeroporti in alcune date. Le celle senza dati verranno riempite con valori NaN. Infine line_chart di streamlit si occuperà di plottare il tutto inserendo in automatico la legenda con i vari colori per gli aeroporti. Nel backend c'è bisogno di avere a disposizione una query per ottenere i ritardi (mensili o giornalieri) di un aeroporto in date specifiche. La query è la seguente ⁵:

```

1 def calcolo_ritardo_per_mesi(aeroporto, mese_inizio, mese_fine):
2     ritardo_per_mesi = (df.filter((col("Origin") == aeroporto) &
        (col("Month") >= mese_inizio) & (col("Month") <= mese_fine
            )).groupBy("Month").agg(pyspark_round(avg("DepDelay"), 2)).

```

⁵si vedrà il caso dei ritardi mensili ma il discorso è equivalente per quelli giornalieri

3

```
alias("ritardo_medio"), count("*").alias("voli_totali")).  
orderBy("Month"))  
return ritardo_per_mesi
```

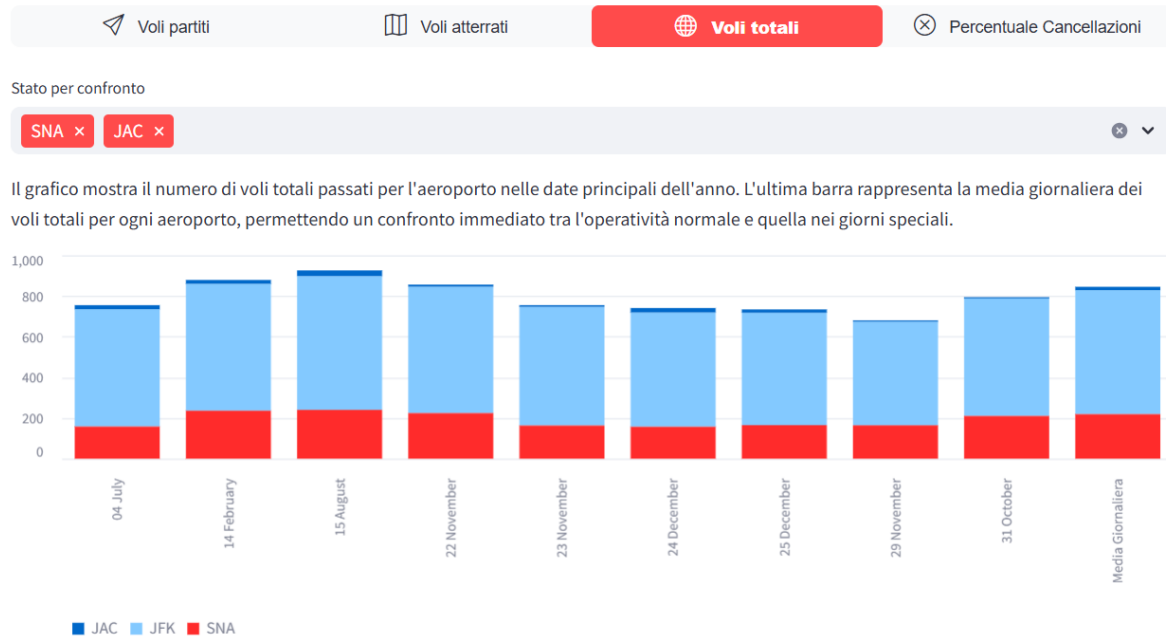
Si filtra il df per l'aeroporto di interesse e nel periodo specifico scelto, si raggruppa per mesi e si aggrega calcolando il ritardo medio (approssimato a due cifre decimali) ed il numero totale di voli. Si ordina per mesi e si restituisce.

Operatività nei giorni più importanti dell'anno Questa sezione risulta essere molto d'aiuto per comprendere a pieno le tendenze di voli partiti, atterrati e cancellati in alcuni dei giorni più importanti per il popolo americano anche rispetto la media giornaliera. Sono stati presi come riferimento i seguenti giorni:

- Indipendenza
- Natale
- Vigilia di Natale
- Ferragosto
- Halloween
- Giorno del Ringraziamento
- Giorno prima del giorno del ringraziamento
- Giorno di San Valentino
- Black Friday
- Media giornaliera

Anche in questo caso è possibile effettuare un confronto con altri aeroporti ottenendo un bar chart del genere:

Operatività nei giorni più importanti dell'anno



Vediamo il codice per graficare i voli partiti dall'aeroporto (o dagli aeroporti in quei giorni):

```

1 dati_aeroporti = {}
2 voli_date_importanti = {}
3 totale_voli_anno = totale_voli_da_aeroporto(aeroporto_selezionato
4 )
5 media_giornaliera = totale_voli_anno / 365
6 voli_date_importanti["Media Giornaliera"] = media_giornaliera
7
8 for data in date_importanti:
9     num_voli = totale_voli_da_aeroporto(aeroporto_selezionato,
10     data)
11     data_dt = pd.to_datetime(data)
12     data_formattata = data_dt.strftime('%d %B')
13     voli_date_importanti[data_formattata] = num_voli
14
15 df_principale = pd.DataFrame(list(voli_date_importanti.items()),
16     columns=['Data', aeroporto_selezionato])
17 df_principale.set_index('Data', inplace=True)
18 dati_aeroporti[aeroporto_selezionato] = df_principale
  
```

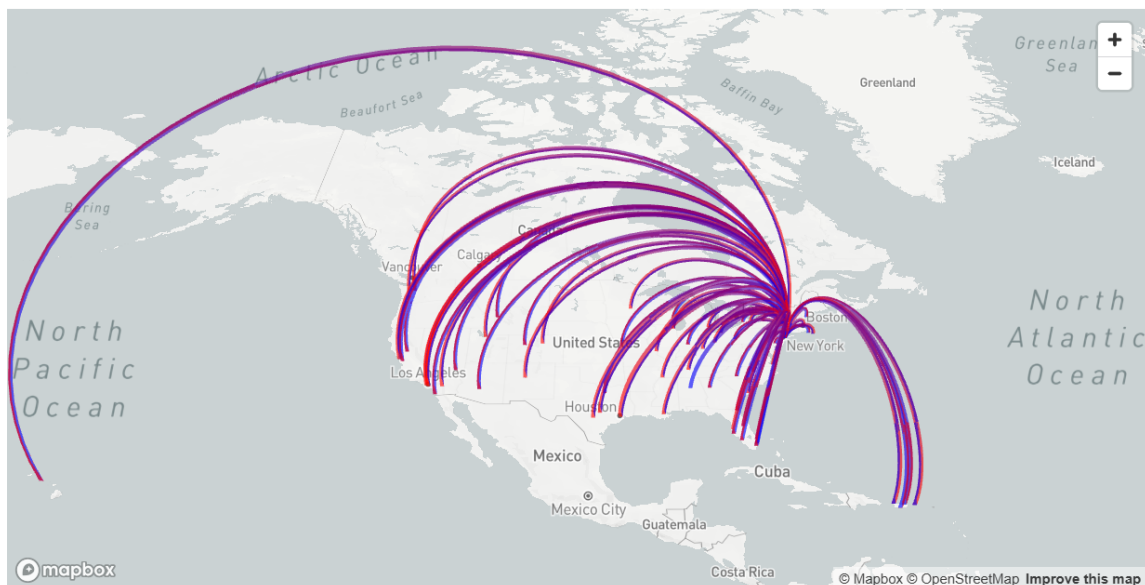
Come avveniva nel caso dei ritardi si crea un dizionario che conterrà i dati per tutti gli

aeroporti nelle specifiche date, ed un altro che contiene i dati nelle specifiche date per il singolo aeroporto. Si calcola la media giornaliera e si scorrono le date per riempire il dizionario dei dati per quell'aeroporto in quei giorni grazie alle specifiche queries. Una volta calcolato per ogni data si inserisce il tutto nel dizionario dati_aeroporti e se presenti altri con cui fare il confronto si prosegue con la stessa logica come fatto sopra nel caso dei ritardi medi. Alla fine viene creato uno streamlit.bar_chart che in automatico provvederà a garantire la legenda e il plot dei vari dati.

Aeroporti collegati Può infine essere d'aiuto avere una vista chiara su mappa di tutti gli aeroporti raggiungibili da quello dato in base ai dati del dataset, perciò è stata creata una mappa con sopra tutte le rotte partite dall'aeroporto:

Aeroporti collegati

Il grafico mostra le rotte aeree che collegano l'aeroporto selezionato ad altri aeroporti

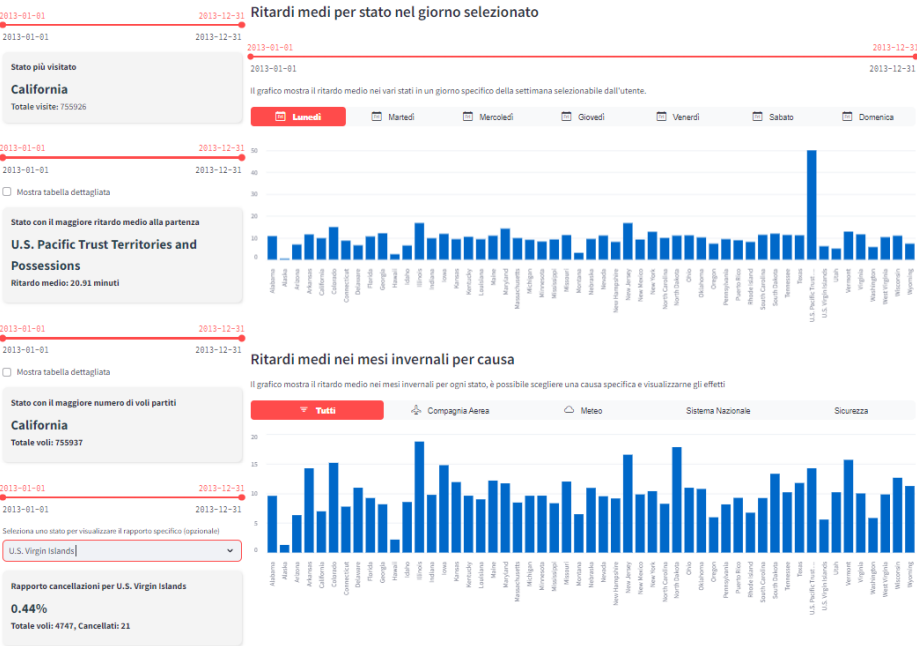


Le funzioni utilizzate sono le stesse citate all'inizio di questa sezione relativa agli aeroporti.

```
1 aeroporti_collegati= spark_to_pandas(
    tratte_distinte_per_aeroporto(aeroporto_selezionato))
2 coordinate_df=load_coordinate_data()
3 tratte_coords = get_coordinates(aeroporti_collegati,
    coordinate_df)
4 disegna_tratta(tratte_coords, colonne_tab3[1])
```

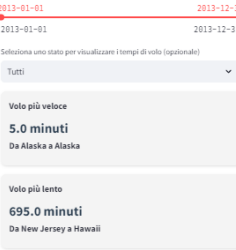
5 Sezione Paesi

Analisi dei dati relativi ai paesi



Dettaglio voli per lo stato selezionato

OriginStateName	TotaliVoli	VoliCancellati
U.S. Virgin Islands	4,747	21



Grafici

- Minor Ritardo Medio
- Stati più Visitati
- Maggiore Traffico Aereo
- Incremento Ritardi invernali rispetto estate in minuti

OriginStateName	RitardoMedio	DestStateName	TotaliVoli	OriginStateName	ChilometriTotali	OriginStateName	IncrementoRitardi
Alaska	1.6185	California	755,926	California	679,998,114	Delaware	19.6861
Montana	2.1788	Texas	745,545	Texas	517,269,280	North Dakota	14.4531
Hawaii	2.5447	Florida	443,369	Florida	368,826,092	Utah	6.938
Delaware	4.1065	Georgia	435,251	Illinois	256,813,914	Wyoming	6.644
U.S. Virgin Islands	4.2728	Illinois	434,827	New York	266,962,830	South Dakota	6.2182
Idaho	4.5965	New York	289,544	Georgia	252,505,306	Montana	4.4288
Utah	4.8744	Colorado	259,555	Colorado	205,735,923	Minnesota	4.1757
Wyoming	5.1198	North Carolina	221,079	Arizona	171,246,372	U.S. Pacific Trust Terr	4.1605
Washington	5.8777	Arizona	261,569	Virginia	144,038,897	Iowa	4.0887
Oregon	6.3176	Michigan	199,349	Washington	143,403,873	Wisconsin	3.3928

Analisi delle percentuali di cause di cancellazioni

Il grafico mostra la percentuale delle cause di cancellazione, è possibile scegliere un periodo specifico e visualizzarne le percentuali

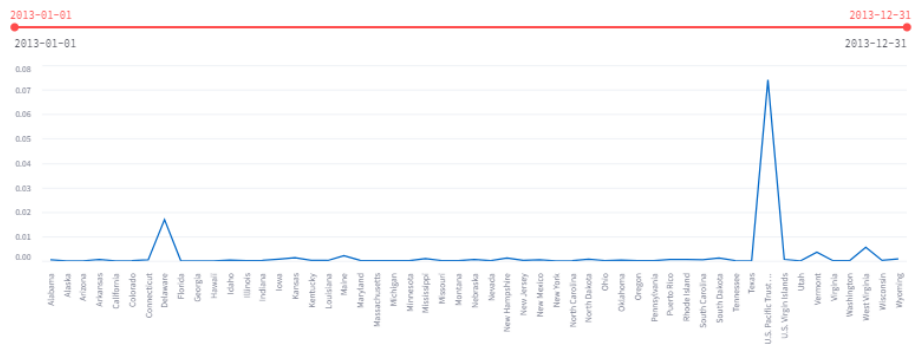


Percentuali delle cause di cancellazioni



Analisi dell'efficienza degli stati (ritardi minimi rispetto al volume di voli)

In questa sezione è graficata l'efficienza dei singoli stati, intesa come ritardo medio su numero di voli effettuati



Seleziona uno stato

Utah



Voli partiti



Voli atterrati



Voli totali

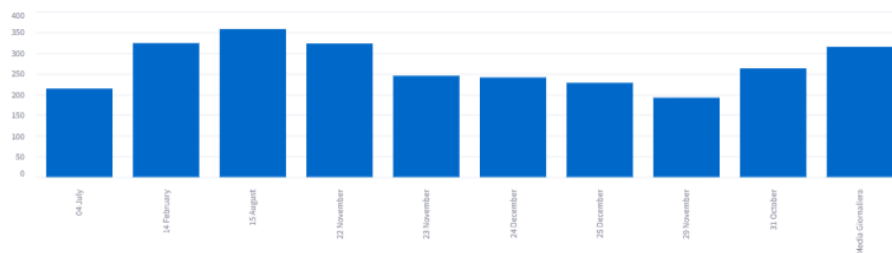


Percentuale Cancellazioni

Aeroporto per confronto

Choose an option

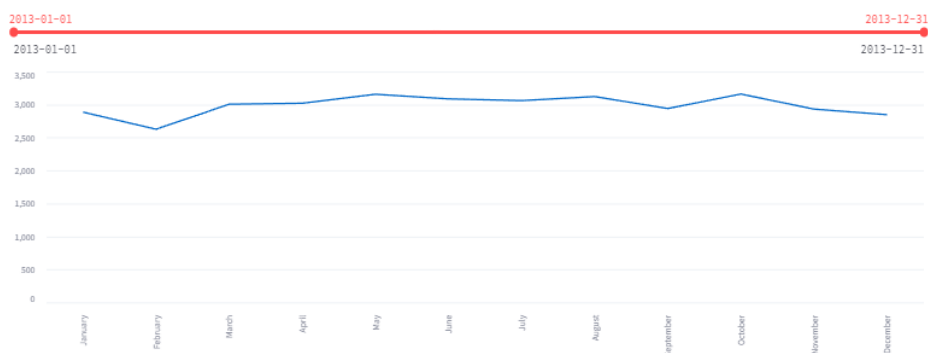
Il grafico mostra il numero di voli partiti dallo stato nelle date principali dell'anno. L'ultima barra rappresenta la media giornaliera dei voli per ogni stato, permettendo un confronto immediato tra l'operatività normale e quella nei giorni speciali.



Stagionalità dei voli per stato specifico

Seleziona uno stato

Alabama



La sezione "Paesi" è dedicata all'analisi e alla visualizzazione dei dati relativi agli stati coinvolti nei voli registrati nel dataset. Attraverso questa sezione, è possibile ottenere informazioni sugli stati di origine e destinazione, come il numero di voli, i ritardi medi, le cancellazioni e il traffico aereo. L'interfaccia interattiva consente agli utenti di filtrare i dati per periodo, selezionare stati specifici e visualizzare informazioni tramite grafici, tabelle e mappe.

La sezione è divisa in due colonne; nella prima, sulla sinistra è possibile osservare informazioni specifiche mentre nella colonna di destra si osservano informazioni più generali che implicano quindi l'uso di grafici.

Per ogni "widget" inserito nella pagina è possibile selezionare tramite uno slider il periodo o una data specifica per visualizzare le informazioni relative a quella finestra temporale.

Colonna di Sinistra Iniziando dalla colonna di sinistra, la prima statistica che si incontra è lo **stato più visitato**. Oltre a indicare il nome, indica il numero di visite totali di quello stato.

```
1 def stato_piu_visitato_date(data_inizio=None, data_fine=None):
2     query = df.filter(col("DestStateName").isNotNull())
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     stati_visitati = query.groupBy("DestStateName").agg(count("*")
7                                                         ).alias("TotaleVisite")
8     stato_piu_visitato = stati_visitati.orderBy(col("TotaleVisite")
9                                                  ).desc().limit(1)
10    return {"stato_piu_visitato": stato_piu_visitato}
```

I dati vengono ricavati da questa query che identifica lo stato di destinazione più visitato in un intervallo temporale specifico. Filtra i dati per rimuovere valori nulli, applica un filtro opzionale sulle date, raggruppa per stato di destinazione e calcola il totale dei voli ricevuti. Ordina i risultati in ordine decrescente e restituisce lo stato con il maggior numero di visite.

La seconda statistica riguarda lo **stato con il maggiore ritardo medio alla partenza**. Viene identificato il nome dello stato con il ritardo medio più alto, seguito dal valore del ritardo medio in minuti.

```
1 def ritardo_medio_per_stato(data_inizio=None, data_fine=None):
2     query = df.filter(col("OriginStateName").isNotNull() & col("
3         DepDelay").isNotNull())
```

```

3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     ritardi = query.groupBy("OriginStateName").agg(avg("DepDelay"
7               ).alias("RitardoMedio"))
8     ritardi_ordinati = ritardi.orderBy(col("RitardoMedio").desc()
9               )
10    return ritardi_ordinati

```

Questa query calcola il ritardo medio alla partenza per ogni stato di origine. I dati vengono filtrati per rimuovere valori nulli nei campi relativi allo stato di origine e al ritardo di partenza. Se vengono forniti intervalli di date, i dati vengono filtrati in base a queste date. I risultati vengono raggruppati per stato di origine e viene calcolato il ritardo medio di partenza. Infine, i risultati vengono ordinati in ordine decrescente in base al ritardo medio, restituendo lo stato con il maggiore ritardo medio.

Siccome il widget visualizza solo lo stato con il ritardo medio maggiore, spuntando la checkbox è possibile vedere in maniera tabellare anche per gli altri stati, in ordine decrescente, il loro valore di ritardo medio alla partenza.

☒ Mostra tabella dettagliata

Stato con il maggiore ritardo medio alla partenza

U.S. Pacific Trust Territories and Possessions

Ritardo medio: 20.91 minuti

Ritardo medio alla partenza per ogni stato di origine

OriginStateName	RitardoMedio
U.S. Pacific Trust Territories and Possessions	20.9085
Illinois	15.7794
New Jersey	15.0938
Colorado	13.4142
Maryland	12.8704
Vermont	11.747
Texas	11.1799
West Virginia	11.104
Nevada	10.7837
New York	10.6895

La terza statistica riguarda lo **stato con il maggiore numero di voli partiti**. Viene identificato il nome dello stato con il maggior numero di voli, seguito dal totale dei voli partiti da quello stato.

```
1 def voli_per_stato_origine(data_inizio=None, data_fine=None):
2     query = df.filter(col("OriginStateName").isNotNull())
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     voli = query.groupBy("OriginStateName").agg(count("*").alias(
7         "TotaleVoli"))
8     voli_ordinati = voli.orderBy(col("TotaleVoli").desc())
9     return voli_ordinati
```

Questa query calcola il numero totale di voli partiti da ciascuno stato di origine. Viene eseguito un filtro per rimuovere valori nulli nel campo relativo allo stato di origine. Se vengono forniti intervalli di date, i dati vengono filtrati di conseguenza. Successivamente, i dati vengono raggruppati per stato di origine e viene calcolato il totale dei voli partiti da ciascuno stato. I risultati vengono ordinati in ordine decrescente in base al numero di voli, restituendo lo stato con il maggior numero di voli.

Inoltre, anche in questa sezione è possibile spuntare la checkbox per visualizzare in ordine decrescente il numero totale di voli per ogni stato di origine.

Successivamente, si può vedere il **rapporto tra voli cancellati e voli totali per stato**. Viene calcolato il rapporto di cancellazione per ciascuno stato e, se selezionato un stato specifico, viene visualizzato il rapporto per quello stato.

```
1 def rapporto_voli_cancellati(data_inizio=None, data_fine=None,
2                               stato=None):
3     query = df.filter(col("OriginStateName").isNotNull())
4     if data_inizio and data_fine:
5         query = query.filter((col("FlightDate") >= data_inizio) &
6                               (col("FlightDate") <= data_fine))
7     if stato:
8         query = query.filter(col("OriginStateName") == stato)
9     rapporto = query.groupBy("OriginStateName").agg(
10         count("*").alias("TotaleVoli"),
11         sum(col("Cancelled").cast("int")).alias("VoliCancellati")
12     ).withColumn(
13         "RapportoCancellati", col("VoliCancellati") / col("
14         TotaleVoli")
15 )
```



```

13     rapporto_ordinato = rapporto.orderBy(col("RapportoCancellati"
14         ).desc())
    return rapporto_ordinato

```

Questa query calcola il rapporto tra il numero di voli cancellati e il totale dei voli per ogni stato di origine. I dati vengono filtrati per rimuovere valori nulli nel campo relativo allo stato di origine. Se vengono forniti intervalli di date, i dati vengono filtrati in base a queste date. Se uno stato specifico viene selezionato, i dati vengono ulteriormente filtrati per quello stato. Il rapporto viene calcolato come la somma dei voli cancellati divisa per il totale dei voli. Infine, i risultati vengono ordinati in ordine decrescente in base al rapporto di cancellazioni.

Se non viene selezionato nessuno Stato, anche qui verrà mostrata una tabella che presenterà i dati per ogni stato e conterrà i parametri raffiguranti il totale dei voli per quel paese e i voli cancellati relativi ad esso.

Tramite i due seguenti widget, è possibile visualizzare i **tempi di volo più veloci e più lenti** per uno stato selezionato, con la possibilità di filtrare i dati per un intervallo temporale specifico.

In entrambi i widget è consentito selezionare un intervallo di date tramite uno **slider** e scegliere uno stato di origine. Vengono quindi calcolati il volo più veloce e il volo più lento, con la visualizzazione dei dettagli per ciascuno.

```

1 def tempi_volo_per_stato(data_inizio=None, data_fine=None, stato=
  None):
2     query = df.filter(col("OriginStateName").isNotNull() & col("
      AirTime").isNotNull())
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
          (col("FlightDate") <= data_fine))
5     if stato:
6         query = query.filter(col("OriginStateName") == stato)
7     volo_piu_veloce = query.orderBy(col("AirTime").asc()).limit
      (1)
8     volo_piu_lento = query.orderBy(col("AirTime").desc()).limit
      (1)
9     return {
10         "volo_piu_veloce": volo_piu_veloce,
11         "volo_piu_lento": volo_piu_lento
12     }

```

Questa query calcola il volo con il tempo di volo più breve (più veloce) e quello con il tempo di volo più lungo (più lento). I dati vengono filtrati per rimuovere valori nulli nei

campi relativi allo stato di origine e al tempo di volo. Se vengono forniti intervalli di date, i dati vengono filtrati di conseguenza. Se uno stato specifico viene selezionato, i dati vengono ulteriormente filtrati per quello stato. I voli vengono quindi ordinati in ordine crescente per il volo più veloce e in ordine decrescente per il volo più lento, restituendo i voli con i tempi di volo estremi.

Colonna di Destra Passando ora alla colonna di destra, sarà possibile trovare grafici che rappresentano informazioni di stampo più generale come ritardi per causa, per giorno della settimana o informazioni relative alla stagionalità dei voli.

Questo primo grafico visualizza il **ritardo medio per stato in un giorno specifico della settimana**. L'utente può selezionare un intervallo di date tramite uno **slider** e un giorno specifico della settimana tramite un menu orizzontale. I risultati vengono poi mostrati tramite un grafico a barre che rappresenta il ritardo medio per ogni stato di origine.

Il processo avviene come segue:

```
1 def ritardi_medi_per_giorno(data_inizio=None, data_fine=None,
2   giorno=None):
3     query = df.filter(col("OriginStateName").isNotNull() & col("
4       DepDelay").isNotNull())
5
6     if data_inizio and data_fine:
7       query = query.filter((col("FlightDate") >= data_inizio) &
8         (col("FlightDate") <= data_fine))
9
10    if giorno is not None:
11      query = query.filter(col("DayOfWeek") == giorno)
12
13    ritardi = query.groupBy("OriginStateName").agg(avg("DepDelay"
14      ).alias("RitardoMedio"))
15
16    return ritardi.orderBy(col("RitardoMedio").desc())
```

La query filtra i voli per stato di origine e ritardo alla partenza, escludendo valori nulli. Se vengono selezionati un intervallo di date, i dati vengono filtrati per quelle date. Inoltre, se un giorno specifico della settimana viene selezionato, viene applicato un ulteriore filtro per il giorno della settimana. Infine, la query raggruppa i dati per stato di origine e calcola il ritardo medio alla partenza, ordinando i risultati in ordine decrescente per ritardo.

La prossima sezione visualizza il **ritardo medio nei mesi invernali per causa specifica**. Gli utenti possono selezionare una causa (ad esempio, *Compagnia Aerea*,

Meteo, Sistema Nazionale, Sicurezza) tramite un menu orizzontale. I risultati vengono mostrati tramite un grafico a barre che rappresenta il ritardo medio per ogni stato di origine nei mesi di dicembre, gennaio e febbraio.

Il processo avviene come segue:

```
1 def incremento_ritardi_invernali(causa=None):
2     query = df.filter(
3         col("OriginStateName").isNotNull() &
4         col("DepDelay").isNotNull() &
5         col("Month").isin([12, 1, 2]) # Selezione dei mesi
6         )
7     if causa:
8         query = query.filter(col("CancellationCode") == causa)
9     ritardi = query.groupBy("OriginStateName").agg(
10         avg("DepDelay").alias("RitardoMedio")
11     )
12
13     return ritardi.orderBy(col("RitardoMedio").desc())
```

La query filtra i dati per stato di origine, ritardo alla partenza, e mesi invernali (dicembre, gennaio, febbraio). Se viene selezionata una causa specifica, viene applicato un ulteriore filtro per quella causa. Successivamente, la query calcola il ritardo medio per stato di origine e ordina i risultati in ordine decrescente di ritardo.

Successivamente ci sarà una sezione formata da colonne con lo scopo di fornire un'analisi approfondita delle performance aeree in base a diversi parametri, con tabelle che visualizzano i dati per vari stati di origine. Gli utenti possono selezionare un periodo di tempo specifico tramite uno *slider*, e vengono visualizzate le seguenti informazioni:

- **Minor Ritardo Medio:** Stati con il minor ritardo medio alla partenza.
- **Stati Più Visitati:** Stati con il maggior numero di voli in destinazione.
- **Maggiore Traffico Aereo:** Stati con il maggiore traffico aereo, misurato in chilometri percorsi.
- **Incremento Ritardi Invernali Rispetto Estate:** Stati con il maggiore incremento nei ritardi durante i mesi invernali (dicembre, gennaio, febbraio) rispetto ai mesi estivi (giugno, luglio, agosto).

Il codice che calcola i risultati per ciascuna sezione è il seguente:

```

1 def stati_con_minor_ritardo_date(data_inizio=None, data_fine=None
2 ):
3     query = df.filter(col("OriginStateName").isNotNull() & col("
4         DepDelay").isNotNull())
5
6     if data_inizio and data_fine:
7         query = query.filter((col("FlightDate") >= data_inizio) &
8             (col("FlightDate") <= data_fine))
9
10    ritardi = query.groupBy("OriginStateName").agg(avg("DepDelay"
11        ).alias("RitardoMedio"))
12
13    return ritardi.orderBy(col("RitardoMedio").asc())

```

Questa query calcola il ritardo medio alla partenza (DepDelay) per ogni stato di origine (OriginStateName) nel periodo selezionato. I dati vengono aggregati per stato e ordinati in ordine crescente in base al ritardo medio.

```

1 def stati_piu_visitati_date(data_inizio=None, data_fine=None):
2     query = df.filter(col("DestStateName").isNotNull())
3
4     if data_inizio and data_fine:
5         query = query.filter((col("FlightDate") >= data_inizio) &
6             (col("FlightDate") <= data_fine))
7
8     voli = query.groupBy("DestStateName").agg(count("*").alias("
9         TotaleVoli"))
10
11    return voli.orderBy(col("TotaleVoli").desc())

```

Questa query calcola il numero totale di voli verso ogni stato di destinazione (DestStateName) nel periodo selezionato. I voli vengono raggruppati per stato di destinazione e ordinati in ordine decrescente in base al numero totale di voli.

```

1 def stati_maggiore_traffico_date(data_inizio=None, data_fine=None
2 ):
3     query = df.filter(
4         (col("OriginStateName").isNotNull()) &
5         (col("DestStateName").isNotNull()) &
6         (col("Distance").isNotNull())
7     )

```

```

8     if data_inizio and data_fine:
9         query = query.filter((col("FlightDate") >= data_inizio) &
10                               (col("FlightDate") <= data_fine))
11
12     traffico = query.groupBy("OriginStateName").agg(
13         sum("Distance").alias("ChilometriTotali")
14     )
15
16     return traffico.orderBy(col("ChilometriTotali").desc())

```

Questa query calcola il traffico aereo totale per ogni stato di origine (OriginStateName) in termini di distanza percorsa (Distance). La distanza totale viene aggregata per stato di origine e ordinata in ordine decrescente.

```

1 def stati_con_maggiore_increm_ritardo_inverno_rispetto_estate(
2     data_inizio=None, data_fine=None):
3     mesi_invernali = [12, 1, 2]
4     mesi_estivi = [6, 7, 8]
5     if (data_inizio and data_fine):
6         ritardi_invernali = df.filter((col("Month").isin(
7             mesi_invernali)) & (col("FlightDate") >= data_inizio)
8             & (col("FlightDate") <= data_fine)) \
9             .groupBy("OriginStateName") \
10            .agg(avg("ArrDelayMinutes").alias("
11                WinterAvgDelay"))
12         ritardi_estivi = df.filter((col("Month").isin(mesi_estivi)
13             )) & (col("FlightDate") >= data_inizio) & (col("
14                FlightDate") <= data_fine)) \
15            .groupBy("OriginStateName") \
16            .agg(avg("ArrDelayMinutes").alias("
17                SummerAvgDelay"))
18         confronto_ritardi = ritardi_invernali.join(ritardi_estivi
19             , on="OriginStateName", how="inner")
20         confronto_ritardi = confronto_ritardi.withColumn("
21             IncrementoRitardi", col("WinterAvgDelay") - col("
22                 SummerAvgDelay"))
23         return confronto_ritardi.orderBy(col("IncrementoRitardi")
24             .desc()).limit(10)
25     else:
26         ritardi_invernali = df.filter(col("Month").isin(
27             mesi_invernali)) \
28             .groupBy("OriginStateName") \
29             .agg(avg("ArrDelayMinutes").alias("

```

```

18         WinterAvgDelay"))
19     ritardi_estivi = df.filter(col("Month").isin(mesi_estivi)
20         ) \
21         .groupBy("OriginStateName") \
22         .agg(avg("ArrDelayMinutes").alias("
23             SummerAvgDelay"))
24
25     confronto_ritardi = ritardi_invernali.join(ritardi_estivi
26         , on="OriginStateName", how="inner")
27     confronto_ritardi = confronto_ritardi.withColumn("
28         IncrementoRitardi", col("WinterAvgDelay") - col("
29             SummerAvgDelay"))
30     return confronto_ritardi.orderBy(col("IncrementoRitardi")
31         .desc()).limit(10)

```

Questa query calcola l'incremento dei ritardi medi tra i mesi invernali (dicembre, gennaio, febbraio) e i mesi estivi (giugno, luglio, agosto) per ogni stato di origine (`OriginStateName`). I ritardi invernali ed estivi vengono calcolati separatamente e confrontati, restituendo la differenza (incremento) in ritardo per ogni stato.

Nella prossima sezione si calcolano le percentuali delle cause di cancellazione dei voli. La funzione `percentuali_cause_cancellazioni` filtra i voli cancellati (`Cancelled == 1`) e calcola la percentuale di ogni causa di cancellazione in base al numero di casi totali. La mappatura delle cause è effettuata tramite un dizionario `cause_map` che associa i codici delle cause ai nomi descrittivi. La percentuale di ogni causa è calcolata come il rapporto tra il numero di casi per quella causa e il totale dei voli cancellati, moltiplicato per 100. Il risultato è poi ordinato in ordine decrescente in base alla percentuale.

```

1 def percentuali_cause_cancellazioni(data_inizio=None, data_fine=
2     None):
3     query = df.filter(col("Cancelled") == 1)
4
5     if data_inizio and data_fine:
6         query = query.filter((col("FlightDate") >= data_inizio) &
7             (col("FlightDate") <= data_fine))
8
9     cause = query.groupBy("CancellationCode").agg(
10         count("*").alias("TotaleCasi")
11     ).withColumn(
12         "Percentuale", (col("TotaleCasi") / query.count()) * 100

```

```
13     return cause.orderBy(col("Percentuale").desc())
```

Successivamente, il risultato delle percentuali viene mappato per visualizzare i nomi delle cause (ad esempio, "Compagnia Aerea", "Meteo", "Sistema Nazionale", "Sicurezza") utilizzando la mappatura definita in `cause_map`. Il grafico a torta, creato con la libreria Plotly, visualizza le percentuali delle diverse cause di cancellazione. Le etichette e i valori sono associati alla colonna `Causa` e `Percentuale` nel dataframe `cause_df`. Il grafico viene mostrato utilizzando `st.plotly_chart`.

```
1 fig = px.pie(  
2     cause_df,  
3     names="Causa",  
4     values="Percentuale",  
5     title="Percentuali delle cause di cancellazioni",  
6     color_discrete_sequence=px.colors.qualitative.Set3  
7 )  
8 st.plotly_chart(fig, use_container_width=True)
```

Viene poi presentata una sezione che analizza l'efficienza degli stati, intesa come il rapporto tra il ritardo medio e il numero di voli effettuati. Questo concetto viene descritto tramite un'analisi basata sulle percentuali di cancellazioni e sui ritardi medi, con l'obiettivo di visualizzare l'efficienza operativa di ciascuno stato in relazione al volume di voli e ai ritardi.

Il codice inizia con un'interfaccia utente per la selezione del periodo tramite uno slider, che permette all'utente di scegliere un intervallo di date per il quale calcolare l'efficienza. Successivamente, viene eseguita una query utilizzando la funzione `stati_efficienza` che calcola l'efficienza per ogni stato, eseguendo un raggruppamento per stato (`OriginStateName`) e calcolando il ritardo medio (la somma dei ritardi di arrivo e partenza) e il numero totale di voli effettuati. La colonna `Efficienza` è ottenuta come il rapporto tra il ritardo medio e il numero di voli.

```
1 def stati_efficienza(data_inizio=None, data_fine=None):  
2     query = df.filter(  
3         (col("OriginStateName").isNotNull()) & (col("ArrDelay").  
4             isNotNull()) & (col("DepDelay").isNotNull())  
5     )  
6     if data_inizio and data_fine:  
7         query = query.filter((col("FlightDate") >= data_inizio) &  
8             (col("FlightDate") <= data_fine))  
9     efficienza = query.groupBy("OriginStateName").agg(  
10         avg(col("ArrDelay") + col("DepDelay")).alias("RitardoMedio"),
```

```

11         count("*").alias("NumeroVoli")
12     ).withColumn(
13         "Efficienza", col("RitardoMedio") / col("NumeroVoli")
14     ).select("OriginStateName", "Efficienza").orderBy("Efficienza")
15
16     return efficienza

```

Se non ci sono dati disponibili per il periodo selezionato, viene mostrato un messaggio di avviso tramite `st.warning`. In caso contrario, i dati vengono elaborati per la visualizzazione: vengono rinominate alcune colonne, ordinati in base all'indice di efficienza e infine presentati tramite un grafico a linee con `st.line_chart`.

```

1 st.line_chart(
2     data=stati_efficienza_df.set_index("Stato")["IndiceEfficienza"],
3     use_container_width=True,
4     height=400
5 )

```

Il grafico mostra l'indice di efficienza per ciascun stato, con un valore più basso che indica una maggiore efficienza.

La penultima sezione permette all'utente di selezionare uno stato tramite una `selectbox` e visualizzare vari grafici a barre in base alla selezione. Le opzioni del grafico includono il numero di voli partiti, atterrati, totali e la percentuale di cancellazioni. Un menu orizzontale consente di scegliere l'opzione desiderata per il grafico. Sono anche presenti delle date importanti (come festività e giornate speciali) per confrontare i dati di voli negli aeroporti durante quei periodi.

Inizia con la selezione dello stato attraverso la funzione `selectbox`:

```

1 stato_selezionato_giorni = st.selectbox(
2     "Seleziona uno stato",
3     options=stati_disponibili["OriginStateName"].dropna().tolist()
4 )

```

Viene poi creato un menu a opzioni orizzontale tramite la funzione `option_menu`, che permette di scegliere una delle quattro opzioni: "Voli partiti", "Voli atterrati", "Voli totali" e "Percentuale Cancellazioni":

```

1 grafico_a_barre_opzione = option_menu(
2     menu_title=None,
3     options=["Voli partiti", "Voli atterrati", "Voli totali", "Percentuale Cancellazioni"],

```



```

4     icons=["send", "map", "globe", "x-circle"],
5     default_index=0,
6     orientation="horizontal",
7     styles={
8         "container": {"padding": "0px", "background-color": "#
          f8f9fa"},
9         "icon": {"color": "light-grey", "font-size": "18px"},
10        "nav-link": {"font-size": "14px", "text-align": "center",
          "margin": "0px"},
11        "nav-link-selected": {"background-color": "light-grey"},
12    },
13 )

```

Le date speciali da confrontare sono definite come una lista di date significative. Queste date includono eventi come Natale, Ferragosto, e il Giorno del Ringraziamento:

```

1 date_importanti = [
2     "2013-07-04", "2013-12-25", "2013-12-24", "2013-08-15",
3     "2013-10-31", "2013-11-23", "2013-11-22", "2013-02-14", "
      2013-11-29"
4 ]
5 confronto_date_importanti = st.multiselect("Aeroporto per
      confronto", load_airport_codes(), key='date_importanti')

```

A seconda dell'opzione selezionata nel menu a barre, il codice esegue il calcolo dei voli partiti, atterrati, totali, o la percentuale di voli cancellati, utilizzando le funzioni `totale_voli_da_stato`, `totale_voli_verso_stato`, `numero_voli_per_stato`, e `percentuale_voli_cancellati`.

Il risultato finale è un grafico a barre che mostra il numero di voli per ogni giorno speciale, confrontando la media giornaliera e le operazioni durante i giorni speciali. Un comportamento simile si applica agli altri tipi di grafico, come "Voli atterrati", "Voli totali" e "Percentuale Cancellazioni".

```

1 @st.cache_data(show_spinner=False)
2 def totale_voli_da_stato(stato, data=None):
3     if(data is None):
4         return df.filter((col("OriginStateName")==stato)).count()
5     if(data):
6         return df.filter((col("OriginStateName")==stato)&(col("
          FlightDate")==data)).count()

```

La funzione `totale_voli_da_stato` restituisce il numero totale di voli partiti dallo stato specificato, basandosi sulla colonna `OriginStateName`. Se la data non è specificata, la funzione conta tutti i voli partiti dallo stato. Se la data è fornita, viene filtrato il DataFrame per includere solo i voli partiti dallo stato nella data specifica.

```

1 @st.cache_data(show_spinner=False)
2 def totale_voli_verso_stato(stato, data=None):
3     if(data is None):
4         return df.filter((col("DestStateName")==stato)).count()
5     if(data):
6         return df.filter((col("DestStateName")==stato)&(col("
            FlightDate")==data)).count()

```

La funzione `totale_voli_verso_stato` restituisce il numero totale di voli arrivati nello stato specificato, basandosi sulla colonna `DestStateName`. Se la data non è fornita, la funzione conta tutti i voli che sono arrivati nello stato. Se la data è specificata, il DataFrame viene filtrato per includere solo i voli verso lo stato nella data fornita.

```

1 @st.cache_data(show_spinner=False)
2 def numero_voli_per_stato(stato, data=None):
3     if data is None:
4         voli_filtrati = df.filter((col("OriginStateName") ==
            stato) | (col("DestStateName") == stato))
5         numero_di_voli = voli_filtrati.count()
6         return numero_di_voli
7     if data:
8         voli_filtrati = df.filter(((col("OriginStateName") ==
            stato) | (col("DestStateName") == stato)) & (col("
            FlightDate") == data))
9         numero_di_voli = voli_filtrati.count()
10        return numero_di_voli

```

La funzione `numero_voli_per_stato` calcola il numero di voli totali (sia in partenza che in arrivo) per uno stato. Se la data non è specificata, la funzione restituisce il numero totale di voli che hanno origine o destinazione nello stato. Se la data è fornita, la funzione filtra i voli in base alla data e calcola i voli per quella giornata.

```

1 @st.cache_data(show_spinner=False)
2 def percentuale_voli_cancellati_stato(stato, data=None):
3     if(data is None):
4         cancellati = df.filter((col("Cancelled")==1.00)&(col("
            OriginStateName")==stato)).count()
5         voli_totali_da_stato = totale_voli_da_stato(stato)
6         if(voli_totali_da_stato==0):
7             return 0
8         perc = cancellati*100/voli_totali_da_stato
9         return round(perc,2)
10    if(data):
11        cancellati = df.filter((col("Cancelled")==1.00)&(col("

```

```

12         OriginStateName")==stato)&(col("FlightDate")==data)).
13         count()
14     voli_totali_da_stato = totale_voli_da_stato(stato, data)
15     if(voli_totali_da_stato==0):
16         return 0
17     perc = cancellati*100/voli_totali_da_stato
18     return round(perc,2)

```

La funzione `percentuale_voli_cancellati_stato` calcola la percentuale di voli cancellati per uno stato. Se la data non è specificata, la funzione restituisce la percentuale di voli cancellati rispetto al totale dei voli partiti dallo stato. Se la data è fornita, la funzione calcola la percentuale di cancellazioni per quella data specifica. La percentuale viene calcolata come il numero di voli cancellati diviso il numero totale di voli e moltiplicato per 100.

L'ultima sezione nella pagina permette di esplorare la stagionalità dei voli per uno stato specifico, utilizzando un'interfaccia interattiva con una selezione di stato e un intervallo di date.

Viene chiamata la funzione `stagionalita_voli_per_stato`, passando le date selezionate e lo stato scelto come parametri. La funzione esegue una query per calcolare il numero di voli per ogni mese, considerando i filtri applicati.

```

1     if stagionalita_df.empty:
2         st.warning(f"Nessun dato trovato per lo stato selezionato
3             : {stato_selezionato}.")

```

Se i dati risultano vuoti, viene visualizzato un messaggio di avviso.

```

1     else:
2         stagionalita_df["Mese"] = stagionalita_df["Month"].apply(
3             lambda x: calendar.month_name[x])
4         stagionalita_df = stagionalita_df.rename(columns={"
5             NumeroVoli": "Voli"})
6         mesi_ordinati = list(calendar.month_name)[1:]
7         stagionalita_df["Mese"] = pd.Categorical(stagionalita_df[
8             "Mese"], categories=mesi_ordinati, ordered=True)
9         stagionalita_df = stagionalita_df.sort_values("Mese")

```

Se i dati sono presenti, vengono trasformati per preparare il grafico. Il numero del mese viene sostituito dal nome del mese utilizzando la libreria `calendar`, e la colonna `NumeroVoli` viene rinominata in `Voli`. I mesi vengono ordinati nell'ordine corretto, e il DataFrame viene ordinato per mese.

```

1     st.line_chart(
2         data=stagionalita_df.set_index("Mese")["Voli"],

```

```

3         use_container_width=True,
4         height=400
5     )

```

Infine, viene visualizzato un grafico a linee che mostra la stagionalità dei voli per lo stato selezionato, con i mesi sull'asse x e il numero di voli sull'asse y.

La funzione `stagionalita_voli_per_stato` esegue la seguente query:

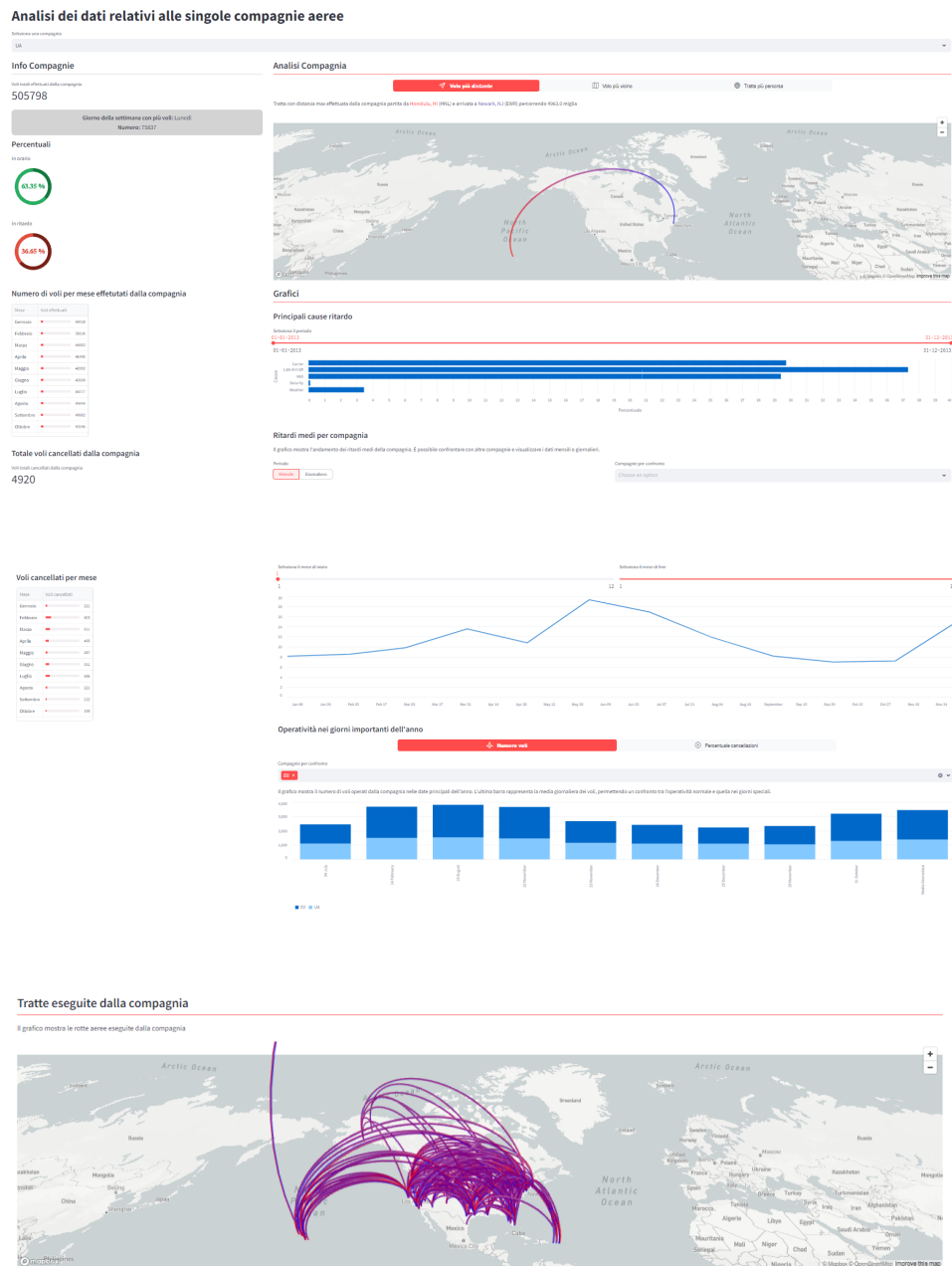
```

1 def stagionalita_voli_per_stato(data_inizio=None, data_fine=None,
2   stato_selezionato=None):
3     query = df.filter(
4         (col("OriginStateName").isNotNull()) & (col("Month").
5             isNotNull())
6     )
7     if stato_selezionato:
8         query = query.filter(col("OriginStateName") ==
9             stato_selezionato)
10    if data_inizio and data_fine:
11        query = query.filter((col("FlightDate") >= data_inizio) &
12            (col("FlightDate") <= data_fine))
13    stagionalita = query.groupBy("Month").agg(
14        count("*").alias("NumeroVoli")
15    ).orderBy("Month")
16    return stagionalita

```

Questa funzione filtra i voli in base allo stato, alla data di inizio e di fine (se specificati) e raggruppa i dati per mese, calcolando il numero di voli per ogni mese. Il risultato è ordinato per mese.

6 Sezione Compagnie



La sezione dedicata alle Compagnie Aeree ha lo scopo di analizzare e visualizzare le performance delle compagnie operanti in un determinato periodo di tempo. Questa sezione inizia con una scelta della compagnia, che permetterà ai vari grafici e

tabelle di mostrare i dati relativi ad essa.

Anche questa pagina è divisa in due colonne, la colonna di sinistra che contiene widget per informazioni più specifiche e immediate e la colonna principale di destra che contiene grafici e mappe.

Colonna di Sinistra Il primo widget mostra il numero totale di voli tramite `st.metric`, calcolato con `numero_voli_per_compagnia`. Determina poi il giorno della settimana con più voli usando la funzione `giorno_della_settimana_con_piu_voli`, definita così:

```
1 def giorno_della_settimana_con_piu_voli(aeroporto=None, compagnia
  =None):
2     if aeroporto:
3         return df.filter((col("Dest") == aeroporto) |
4                           (col("Origin") == aeroporto)) \
5                           .groupBy("DayOfWeek") \
6                           .count() \
7                           .orderBy(col("count").desc()) \
8                           .limit(1)
9     if compagnia:
10        return df.filter((col("Reporting_Airline") == compagnia))
11        \
12        .groupBy("DayOfWeek") \
13        .count() \
14        .orderBy(col("count").desc()) \
15        .limit(1)
16    return df.groupBy("DayOfWeek") \
17        .count() \
18        .orderBy(col("count").desc()) \
19        .limit(1)
```

La funzione filtra il `DataFrame` per aeroporto o compagnia, raggruppa per `DayOfWeek`, conta i voli, ordina in ordine decrescente e restituisce il giorno con più voli. Il risultato è convertito in pandas per ottenere il giorno e il numero di voli, e il giorno è convertito in formato testuale con `converti_giorno`.

Andando avanti i due widget incontrati sono le percentuali di voli in orario e in ritardo per una compagnia selezionata. Le percentuali sono calcolate rispettivamente con le funzioni `percentuale_voli_orario` e `percentuale_voli_ritardo`, e rappresentate graficamente tramite diagrammi a ciambella creati con `make_donut`. I grafici sono visualizzati utilizzando `st.altair_chart`. Le funzioni utilizzate sono definite così:

```
1 @st.cache_data(show_spinner=False)
2 def percentuale_voli_orario(codice_aeroporto=None, compagnia=None
  ):
```

```

3     voli_filtrati = df.filter(col("ArrDelayMinutes").isNotNull())
4     if compagnia is None and codice_aeroporto is None:
5         return round((voli_filtrati.filter(col("ArrDelayMinutes")
6             == 0).count() * 100) / voli_filtrati.count(), 2)
7     if codice_aeroporto:
8         voli_filtrati = voli_filtrati.filter((col("Dest") ==
9             codice_aeroporto))
10    if compagnia:
11        voli_filtrati = voli_filtrati.filter((col("
        Reporting_Airline") == compagnia))
voli = (voli_filtrati.filter(col("ArrDelayMinutes") == 0).
    count() * 100) / voli_filtrati.count()
return round(voli, 2)

```

La successiva funzione `percentuale_voli_orario` filtra il `DataFrame` per includere voli con un valore definito di ritardo. Se non sono specificati parametri, calcola la percentuale di voli senza ritardo (`ArrDelayMinutes == 0`) rispetto al totale. Se vengono specificati un aeroporto o una compagnia, applica i relativi filtri prima del calcolo.

```

1 @st.cache_data(show_spinner=False)
2 def percentuale_voli_ritardo(aeroporto=None, compagnia=None):
3     voli_filtrati = df.filter(col("ArrDelayMinutes").isNotNull())
4     if compagnia is None and aeroporto is None:
5         return round((voli_filtrati.filter(col("ArrDelayMinutes")
6             > 0).count() * 100) / voli_filtrati.count(), 2)
7     if aeroporto:
8         voli_filtrati = voli_filtrati.filter((col("Dest") ==
9             aeroporto))
10    if compagnia:
11        voli_filtrati = voli_filtrati.filter((col("
        Reporting_Airline") == compagnia))
voli = (voli_filtrati.filter(col("ArrDelayMinutes") > 0).
    count() * 100) / voli_filtrati.count()
return round(voli, 2)

```

La funzione `percentuale_voli_ritardo` segue un approccio analogo, ma calcola la percentuale di voli con ritardi (`ArrDelayMinutes > 0`). Entrambe le funzioni utilizzano `@st.cache_data` per migliorare le prestazioni, memorizzando in cache i risultati.

La tabella successiva visualizza i dati relativi al numero di voli effettuati da una compagnia nei singoli mesi. I dati sono ottenuti chiamando la funzione `totaleVoliPerMese` e convertiti in un `DataFrame` pandas tramite `spark.to_pandas`. Successivamente, i numeri dei mesi sono mappati ai relativi nomi tramite un dizionario `mesi`. I dati risultanti vengono mostrati in una tabella con due colonne: la colonna dei mesi è etichettata come

Mese, mentre la colonna del numero di voli è rappresentata da una progress bar il cui valore massimo corrisponde al totale dei voli effettuati dalla compagnia. Il totale è calcolato con la funzione `numero_voli_per_compagnia`, definita così:

```

1 @st.cache_data(show_spinner=False)
2 def numero_voli_per_compagnia(compagnia, data=None):
3     if data is None:
4         voli_filtrati = df.filter((col("Reporting_Airline") ==
5                                     compagnia))
6         numero_di_voli = voli_filtrati.count()
7         return numero_di_voli
8     if data:
9         voli_filtrati = df.filter((col("Reporting_Airline") ==
10                                     compagnia) & (col("FlightDate") == data))
11         numero_di_voli = voli_filtrati.count()
12         return numero_di_voli

```

La funzione `numero_voli_per_compagnia` filtra il `DataFrame` in base alla compagnia selezionata. Se è specificata una data, include solo i voli effettuati in quella data; altrimenti considera tutti i voli della compagnia. Conta quindi i voli filtrati e restituisce il totale. La funzione utilizza `@st.cache_data` per ottimizzare le prestazioni memorizzando i risultati.

La colonna di sinistra termina con due widget. Il primo utilizza la funzione `totale_voli_cancellati` per presentare il numero totale di voli cancellati dalla compagnia. Il secondo widget mostra una tabella interattiva con il numero di voli cancellati per mese, generata utilizzando `totale_voli_cancellati_per_mese` e configurata con una progress bar per indicare il numero di voli rispetto al totale della compagnia. Le funzioni utilizzate sono definite come segue:

```

1 @st.cache_data(show_spinner=False)
2 def totale_voli_cancellati(aeroporto=None, compagnia=None):
3     cancellati = df.filter(col("Cancelled") == "1.00").count()
4     if aeroporto:
5         cancellati = df.filter((col("Cancelled") == 1.00) & (col(
6                                     "Origin") == aeroporto)).count()
7     if compagnia:
8         cancellati = df.filter((col("Cancelled") == 1.00) & (col(
9                                     "Reporting_Airline") == compagnia)).count()
10    return cancellati

```

La funzione `totale_voli_cancellati` calcola il numero totale di voli cancellati filtrando il `DataFrame` per la colonna `Cancelled`. Se viene specificato un aeroporto o una compagnia, applica filtri aggiuntivi in base ai parametri forniti.


```

1 def totale_voli_cancellati_per_mese(aeroporto=None, compagnia=
  None):
2     voli_cancellati_per_mese = df.filter(col("Cancelled") ==
      1.00) \
3         .groupBy("Month") \
4         .agg(count("*").alias("NumeroVoliCancellati")) \
5         .orderBy("Month")
6     if aeroporto:
7         voli_cancellati_per_mese = df.filter((col("Origin") ==
      aeroporto) & (col("Cancelled") == 1.00)) \
8         .groupBy("Month") \
9         .agg(count("*").alias("NumeroVoliCancellati")) \
10        .orderBy("Month")
11    if compagnia:
12        voli_cancellati_per_mese = df.filter((col("
      Reporting_Airline") == compagnia) & (col("Cancelled")
      == 1.00)) \
13        .groupBy("Month") \
14        .agg(count("*").alias("NumeroVoliCancellati")) \
15        .orderBy("Month")
16    return voli_cancellati_per_mese

```

La funzione `totale_voli_cancellati_per_mese` calcola il numero di voli cancellati raggruppati per mese. Filtra i voli cancellati (`Cancelled == 1.00`) e raggruppa i dati per `Month`, conteggiando i voli. Anche qui se sono forniti parametri, applica filtri aggiuntivi in base all'aeroporto o alla compagnia. Il risultato è un `DataFrame Spark` ordinato per mese.

Colonna di Destra La colonna di destra inizia con una mappa che mostra informazioni su tratte aeree relative a una compagnia selezionata: *Volo più distante*, *Volo più vicino* e *Tratta più percorsa*. Il menu è creato con `option_menu`. In base all'opzione scelta, vengono recuperati i dati specifici utilizzando le funzioni `volo_distanza_max_compagnia`, `volo_distanza_min_compagnia` o `tratta piu percorsa compagnia`, i quali calcolano rispettivamente il volo più distante, il più vicino e la tratta più percorsa. I dati sono convertiti in un `DataFrame pandas` e vengono visualizzati con un messaggio testuale e una mappa che rappresenta graficamente le coordinate della tratta. Le funzioni utilizzate sono definite come segue:

```

1 def volo_distanza_max_compagnia(compagnia):
2     volo = df.filter((col("Reporting_Airline") == compagnia)) \
3         .orderBy(col("Distance").desc()) \

```

```

4         .limit(1)
5     return volo.select("Origin", "Dest", "OriginCityName", "
        DestCityName", "Distance")
6
7 def volo_distanza_min_compagnia(compagnia):
8     volo = df.filter((col("Reporting_Airline") == compagnia)) \
9         .orderBy(col("Distance").asc()) \
10        .limit(1)
11     return volo.select("Origin", "Dest", "OriginCityName", "
        DestCityName", "Distance")
12
13 def tratta_piu_percorsa_compagnia(compagnia):
14     tratta = df.filter((col("Reporting_Airline") == compagnia)) \
15        .groupBy("Origin", "Dest", "OriginCityName", "
        DestCityName") \
16        .count() \
17        .orderBy(desc("count")) \
18        .limit(1)
19     return tratta.select("Origin", "Dest", "OriginCityName", "
        DestCityName", "count")

```

`volo_distanza_max_compagnia` e `volo_distanza_min_compagnia` filtrano i voli effettuati dalla compagnia specificata e restituiscono rispettivamente il volo con la distanza massima e minima. La funzione `tratta_piu_percorsa_compagnia` raggruppa i dati per città di origine e destinazione, conteggia le occorrenze di ogni tratta e restituisce quella più percorsa, ordinata per frequenza.

Successivamente è presente un grafico che consente di analizzare le cause di ritardo dei voli per una compagnia selezionata in un periodo specifico. L'intervallo temporale è selezionabile tramite uno `slider` che definisce le date di inizio e fine. I dati relativi alle percentuali delle cause di ritardo vengono calcolati con la funzione `percentuali_cause_ritardo`, che elabora i ritardi in base a vari criteri di filtro, inclusa la compagnia e il periodo selezionato. Le percentuali delle cause di ritardo sono visualizzate con un `bar_chart` orizzontale. La funzione utilizzata è definita come segue:

```

1 def percentuali_cause_ritardo(filtro_compagnia=None,
    causa_specifica=None, data_inizio=None, data_fine=None, stato=
    None, aeroporto=None):
2     df_filtrato = df
3     if aeroporto:
4         df_filtrato = df_filtrato.filter((col("Origin") ==
            aeroporto))
5     if stato:

```

```

6         df_filtrato = df_filtrato.filter((col("OriginStateName")
7             == stato) | (col("DestStateName") == stato))
8     if filtro_compagnia:
9         df_filtrato = df_filtrato.filter(col("Reporting_Airline")
10            == filtro_compagnia)
11     if data_inizio and data_fine:
12         df_filtrato = df_filtrato.filter((col("FlightDate") >=
13            data_inizio) & (col("FlightDate") <= data_fine))
14
15     df_filtrato = df_filtrato.fillna(0, subset=["CarrierDelay", "
16        WeatherDelay", "NASDelay", "SecurityDelay", "
17        LateAircraftDelay"])
18
19     ritardi_cause = df_filtrato.select(
20         ["CarrierDelay", "WeatherDelay", "NASDelay", "
21            SecurityDelay", "LateAircraftDelay"]
22     ).agg(
23         sum("CarrierDelay").alias("CarrierDelay"),
24         sum("WeatherDelay").alias("WeatherDelay"),
25         sum("NASDelay").alias("NASDelay"),
26         sum("SecurityDelay").alias("SecurityDelay"),
27         sum("LateAircraftDelay").alias("LateAircraftDelay")
28     )
29
30     ritardi_somma = ritardi_cause.select(
31         (col("CarrierDelay") +
32            col("WeatherDelay") +
33            col("NASDelay") +
34            col("SecurityDelay") +
35            col("LateAircraftDelay")).alias("TotaleRitardo")
36     ).collect()[0]["TotaleRitardo"]
37
38     if causa_specifica:
39         ritardi_percentuali = ritardi_cause.select(
40             pyspark_round((col(causa_specifica) / ritardi_somma *
41                100), 2).alias(f"{causa_specifica}_Percent")
42         )
43     else:
44         ritardi_percentuali = ritardi_cause.select(
45             pyspark_round((col("CarrierDelay") / ritardi_somma *
46                100), 2).alias("Carrier"),
47             pyspark_round((col("WeatherDelay") / ritardi_somma *
48                100), 2).alias("Weather"),

```

```

40         pyspark_round((col("NASDelay") / ritardi_somma * 100)
41             , 2).alias("NAS"),
42         pyspark_round((col("SecurityDelay") / ritardi_somma *
43             100), 2).alias("Security"),
44         pyspark_round((col("LateAircraftDelay") /
45             ritardi_somma * 100), 2).alias("Late Aircraft")
46     )
47     return ritardi_percentuali

```

La funzione `percentuali_cause_ritardo` applica filtri per selezionare solo i voli pertinenti. Le colonne relative ai tipi di ritardo (`CarrierDelay`, `WeatherDelay`, ecc.) sono sommate per ottenere il totale dei ritardi. Le percentuali sono calcolate come rapporto tra il ritardo di una specifica causa e il totale, arrotondate a due cifre decimali. Quando non è specificata una causa particolare, vengono restituite le percentuali di tutte le categorie principali di ritardo.

Successivamente è stato inserito un grafico che consente di confrontare i ritardi medi di voli di diverse compagnie aeree in due modalità: "Mensile" o "Giornaliero". L'utente può selezionare il periodo di tempo (mese o giorno) e le compagnie per il confronto tramite controlli di selezione (`segmented_control` e `multiselect`). A seconda della modalità selezionata, vengono mostrati i ritardi medi mensili o giornalieri per la compagnia principale e le compagnie selezionate. I dati vengono raccolti e visualizzati in un grafico a linee, dove ogni compagnia è rappresentata da una linea separata. La funzione utilizzata per il calcolo dei ritardi mensili e giornalieri è definita come segue:

```

1 def calcolo_ritardo_per_mesi_compagnia(compagnia, mese_inizio,
2     mese_fine):
3     ritardo_per_mesi = (df.filter((col("Reporting_Airline") ==
4         compagnia) & (col("Month") >= mese_inizio) & (col("Month")
5         <= mese_fine)))
6         .groupBy("Month").agg(pyspark_round(avg("
7             DepDelay"), 2).alias("ritardo_medio"),
8             count("*").alias("voli_totali")).
9             orderBy("Month"))
10    return ritardo_per_mesi
11
12 def calcola_ritardo_giornaliero_compagnia(compagnia, data_inizio,
13     data_fine):
14     ritardo_giornaliero = df.filter((col("Reporting_Airline") ==
15         compagnia) & (col("FlightDate") >= data_inizio) & (col("
16             FlightDate") <= data_fine))).groupBy("FlightDate").agg(avg("
17             DepDelay").alias("ritardo_medio_giornaliero"), count("*")
18             .alias("voli_totali")).orderBy("FlightDate")

```

```
8     return ritardo_giornaliero
```

La funzione `calcolo_ritardo_per_mesi_compagnia` calcola il ritardo medio mensile per una compagnia aerea in un intervallo di mesi specificato. I dati vengono aggregati per mese e la media dei ritardi di partenza (`DepDelay`) viene calcolata. La funzione `calcola_ritardo_giornaliero_compagnia` invece calcola il ritardo medio giornaliero per la compagnia aerea in un intervallo di date.

Una volta ottenuti i dati, vengono uniti in un `DataFrame` unico e visualizzati tramite un `line_chart`, in cui ogni compagnia aerea è rappresentata da una linea nel grafico.

Come penultimo grafico è presente un barchart che consente una visualizzazione del numero di voli o della percentuale di cancellazioni durante i giorni speciali dell'anno, selezionabili tramite un'opzione di menu orizzontale.

Quando viene scelto "Numero voli", il programma calcola il numero di voli operati dalla compagnia nelle date importanti dell'anno. La funzione `numero_voli_per_compagnia` viene utilizzata per calcolare il numero di voli per ogni data importante, e il risultato viene mostrato in un grafico a barre. L'ultima barra rappresenta la media giornaliera dei voli, utile per un confronto tra i giorni speciali e l'operatività normale della compagnia. Il codice per questo caso è:

```
1 if grafico_giorni_importanti == "Numero voli":
2     dati_compagnie = {}
3     voli_date_importanti = {}
4     totale_voli_anno = numero_voli_per_compagnia(
5         compagnia_selezionata)
6     media_giornaliera = totale_voli_anno / 365
7     voli_date_importanti["Media Giornaliera"] = media_giornaliera
8
9     for data in date_importanti:
10         num_voli = numero_voli_per_compagnia(
11             compagnia_selezionata, data)
12         data_dt = pd.to_datetime(data)
13         data_formattata = data_dt.strftime('%d %B')
14         voli_date_importanti[data_formattata] = num_voli
15
16     df_principale = pd.DataFrame(
17         list(voli_date_importanti.items()),
18         columns=['Data', compagnia_selezionata]
19     )
20     df_principale.set_index('Data', inplace=True)
21     dati_compagnie[compagnia_selezionata] = df_principale
```

Nel caso della "Percentuale cancellazioni", il codice calcola la percentuale di voli can-

cellati dalle compagnie aeree nelle stesse date, utilizzando la funzione `percentuale_voli_cancellati_compagnia` e visualizza questi dati in un altro grafico a barre. La funzione per il calcolo delle cancellazioni è la seguente:

```

1 @st.cache_data(show_spinner=False)
2 def percentuale_voli_cancellati_compagnia(compagnia, data=None):
3     if data is None:
4         voli_filtrati = df.filter((col("Cancelled") == 1.00) & (
5             col("Reporting_Airline") == compagnia))
6         cancellati = voli_filtrati.count()
7         voli_totali_compagnia = totale_voli_compagnia(compagnia)
8         if(voli_totali_compagnia == 0):
9             return 0
10        perc = cancellati * 100 / voli_totali_compagnia
11        return round(perc, 2)
12    if data:
13        voli_filtrati = df.filter((col("Cancelled") == 1.00) & (
14            col("Reporting_Airline") == compagnia) & (col("
15                FlightDate") == data))
16        cancellati = voli_filtrati.count()
17        voli_totali_compagnia = totale_voli_compagnia(compagnia,
18            data)
19        if(voli_totali_compagnia == 0):
20            return 0
21        perc = cancellati * 100 / voli_totali_compagnia
22        return round(perc, 2)

```

Per entrambi i casi, i dati vengono uniti in un unico DataFrame per ogni compagnia selezionata, e il grafico finale mostra il confronto tra le compagnie sulle date principali dell'anno, visualizzando il numero di voli o la percentuale di cancellazioni.

Come ultima funzione della pagina è presente una mappa che visualizza le tratte percorse da una compagnia aerea specificata, utilizzando le coordinate geografiche per tracciare le rotte su una mappa.

Il processo inizia con l'ottenere le tratte percorse dalla compagnia tramite la funzione `tratte_distinte_per_compagnia`, che filtra il DataFrame per la compagnia selezionata e seleziona le colonne relative agli aeroporti di origine e destinazione (codici e nomi delle città). La funzione utilizzata è la seguente:

```

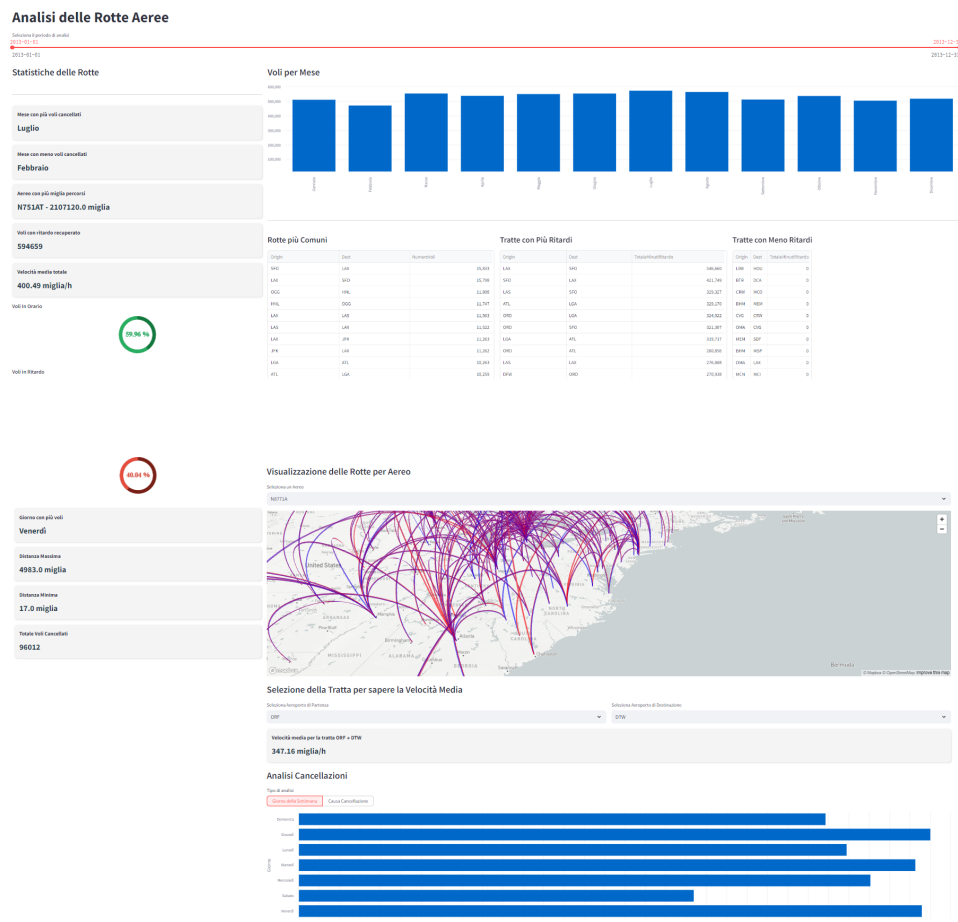
1 def tratte_distinte_per_compagnia(compagnia):
2     tratte_filtrate = df.filter(col("Reporting_Airline") ==
3         compagnia)
4     tratte_distinte = tratte_filtrate.select(col("Origin"), col("
5         OriginCityName"), col("Dest"), col("DestCityName")).

```

```
distinct()
return tratte_distinte
```

Successivamente, i dati ottenuti vengono trasformati in un DataFrame Pandas con la funzione `spark_to_pandas`, e le coordinate geografiche delle tratte vengono caricate tramite la funzione `load_coordinate_data`. Questa funzione restituisce un DataFrame con le coordinate (latitudine e longitudine) di ciascun aeroporto. La funzione `get_coordinates` viene utilizzata per associare le coordinate alle tratte percorse dalla compagnia aerea. Infine, le tratte vengono visualizzate su una mappa, utilizzando la funzione `disegna_tratta`, che disegna le rotte sulle coordinate geografiche permettendo di tracciare sulla mappa tutte le tratte percorse dalla compagnia selezionata.

7 Sezione Rotte



La sezione dedicata alle rotte aeree si concentra sull'analisi dei collegamenti tra aeroporti, con l'obiettivo di offrire una visione dei flussi di traffico aereo. Attraverso questa sezione è possibile ottenere informazioni utili per comprendere i pattern di viaggio e identificare eventuali trend o anomalie.

Per quanto riguarda la gestione della pagina, anche questa è stata divisa in due sezioni principali, precedute da uno slider che consente di selezionare il periodo o la data di cui si vogliono ottenere informazioni.

Come descritto in precedenza, la pagina è divisa in due colonne, la colonna di sinistra contiene dei widget che descrivono informazioni specifiche in maniera immediata, nella colonna di destra invece sono presenti informazioni più generali come grafici, mappe e tabelle.

Colonna di Sinistra Partendo dalla colonna di sinistra, i primi due widget che si incontrano, si occupano di analizzare i dati relativi ai mesi con il maggior e il minor numero di voli cancellati.

```
1 mese_piu_cancellati = spark_to_pandas(mese_con_piu_cancellati(  
    data_inizio, data_fine))  
2 mese_piu = mese_piu_cancellati.iloc[0]["Month"]  
3 mese_piu_nome = mesi[mese_piu]
```

In questo blocco, viene calcolato il mese con il numero maggiore di voli cancellati tramite la funzione `mese_con_piu_cancellati`. Il risultato della query viene trasformato in un DataFrame pandas, dal quale viene estratto il numero del mese (`Month`). Successivamente, questo numero è mappato al corrispondente nome del mese utilizzando un dizionario.

```
1 mese_meno_cancellati = spark_to_pandas(mese_con_meno_cancellati(  
    data_inizio, data_fine))  
2 mese_meno = mese_meno_cancellati.iloc[0]["Month"]  
3 mese_meno_nome = mesi[mese_meno]
```

Analogamente, viene calcolato il mese con il minor numero di voli cancellati utilizzando la funzione `mese_con_meno_cancellati`. I risultati vengono elaborati nello stesso modo, mappando il numero del mese al suo nome.

```
1 def mese_con_piu_cancellati(data_inizio=None, data_fine=None):  
2     query = df.filter(col("Month").isNotNull())  
3     if data_inizio and data_fine:  
4         query = query.filter((col("FlightDate") >= data_inizio) &  
                               (col("FlightDate") <= data_fine))
```



```

5     mesi = query.groupBy("Month").count().orderBy(col("count").
        desc()).limit(1)
6     return mesi

```

La funzione `mese_con_piu_cancellati` filtra i dati per escludere valori nulli nella colonna `Month` e, se specificati, applica un filtro temporale basato sui parametri `data_inizio` e `data_fine`. Successivamente, i dati vengono raggruppati per mese e il numero di voli viene calcolato. I risultati vengono ordinati in ordine decrescente, restituendo il mese con il conteggio massimo.

```

1 def mese_con_meno_cancellati(data_inizio=None, data_fine=None):
2     query = df.filter(col("Month").isNotNull())
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     mesi = query.groupBy("Month").count().orderBy(col("count").
7               asc()).limit(1)
8     return mesi

```

La funzione `mese_con_meno_cancellati` segue lo stesso approccio, ma ordina i risultati in ordine crescente, individuando il mese con il numero minimo di cancellazioni.

Seguono 3 sezioni: l'aereo che ha percorso la maggiore distanza, il numero di voli che hanno recuperato ritardi, e la velocità media complessiva degli aeromobili.

```

1 aereo_piu_km = spark_to_pandas(aereo_piu_km_percorsi(data_inizio,
2     data_fine))
3 aereo = aereo_piu_km.iloc[0]["Tail_Number"]
4 km_percorsi = aereo_piu_km.iloc[0]["TotalDistance"]

```

Viene calcolato l'aereo che ha percorso la maggiore distanza totale nel periodo specificato. I risultati della query vengono convertiti in un DataFrame pandas, dal quale si estraggono il numero di coda (`Tail_Number`) e la distanza totale percorsa (`TotalDistance`).

```

1 numero_voli_recupero = voliRitardoDecolloArrivoAnticipo(
2     data_inizio, data_fine)

```

Viene calcolato il numero di voli che, nonostante un ritardo al decollo (`DepDelay > 0`), sono atterrati in anticipo (`ArrDelay < 0`).

```

1 velocita_media = spark_to_pandas(velocita_media_totale(
2     data_inizio, data_fine))
3 velocita_media_valore = velocita_media.iloc[0]["
4     AverageAircraftSpeed"]

```

La velocità media totale degli aeromobili viene calcolata come il rapporto tra la distanza percorsa (Distance) e il tempo di volo (AirTime). Questo valore viene aggregato su tutti i voli disponibili nel periodo specificato e visualizzato con una precisione di due decimali.

```

1 def aereo_piu_km_percorsi(data_inizio=None, data_fine=None):
2     query = df.filter(col("Tail_Number").isNotNull())
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     aereo = query.groupBy("Tail_Number").agg(
7         sum("Distance").alias("TotalDistance")
8     ).orderBy(col("TotalDistance").desc()).limit(1)
9     return aereo

```

La funzione `aereo_piu_km_percorsi` filtra i voli con un numero di coda valido e, se specificati, applica filtri temporali. I voli vengono raggruppati per numero di coda, calcolando la somma delle distanze percorse. I risultati sono ordinati in ordine decrescente e viene restituito il primo record.

```

1 @st.cache_data(show_spinner=False)
2 def voliRitardoDecolloArrivoAnticipo(data_inizio=None, data_fine=
3     None):
4     query = df.filter((col("DepDelay") > 0) & (col("ArrDelay") <
5         0))
6     if data_inizio and data_fine:
7         query = query.filter((col("FlightDate") >= data_inizio) &
8                               (col("FlightDate") <= data_fine))
9     numero_voli = query.count()
10    return numero_voli

```

La funzione `voliRitardoDecolloArrivoAnticipo` calcola il numero di voli che hanno recuperato ritardi, applicando i filtri sui ritardi al decollo e sugli anticipi all'arrivo.

```

1 def velocita_media_totale(data_inizio=None, data_fine=None):
2     query = df.filter((col("Distance").isNotNull()) & (col("
3         AirTime").isNotNull()))
4     if data_inizio and data_fine:
5         query = query.filter((col("FlightDate") >= data_inizio) &
6                               (col("FlightDate") <= data_fine))
7     velocita = query.withColumn(
8         "AverageSpeed", col("Distance") / (col("AirTime") / 60)
9     ).select(avg("AverageSpeed").alias("AverageAircraftSpeed"))
10    return velocita

```

Infine, la funzione `velocita_media_totale` calcola la velocità media degli aeromobili, considerando i voli con distanza e tempo di volo validi. La velocità media per ciascun volo è calcolata come distanza divisa per il tempo di volo (in ore), e viene restituita come media complessiva.

I due widget successivi si occupano del calcolo delle percentuali di voli in orario e in ritardo, basandosi su due funzioni distinte che filtrano i dati relativi ai ritardi di arrivo dei voli e restituiscono le percentuali desiderate. Queste informazioni vengono quindi visualizzate tramite grafici a ciambella (donut chart) con il supporto di Streamlit e la libreria Altair.

```
1 percentuale_in_orario = percentualeVoliInOrario(data_inizio,
2           data_fine)
3
4 st.markdown("**Voli In Orario**")
5 st.altair_chart(make_donut(percentuale_in_orario, "In Orario", "
6           green"), use_container_width=True)
7 st.markdown("**Voli in Ritardo**")
8 st.altair_chart(make_donut(percentuale_in_ritardo, "In Ritardo",
9           "red"), use_container_width=True)
```

Il codice esegue prima il calcolo delle percentuali di voli in orario e in ritardo chiamando le rispettive funzioni `percentualeVoliInOrario` e `percentuale_voli_ritardo_date`, passando i parametri di data di inizio e fine. Successivamente, i risultati vengono visualizzati tramite due grafici a ciambella (uno per i voli in orario e uno per i voli in ritardo), utilizzando la libreria Altair per generare i grafici. I grafici vengono mostrati con il colore verde per i voli in orario e rosso per i voli in ritardo.

```
1 def percentualeVoliInOrario(data_inizio=None, data_fine=None):
2     query = df.filter(col("ArrDelayMinutes").isNotNull())
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     voli_totali = query.count()
7     in_orario = query.filter(col("ArrDelayMinutes") <= 0).count()
8     percentuale = (in_orario / voli_totali) * 100 if voli_totali
9     > 0 else 0
10    return round(percentuale, 2)
```

La funzione `percentualeVoliInOrario` calcola la percentuale di voli arrivati in orario. Viene eseguito un filtro sui voli che contengono il campo `ArrDelayMinutes`,

che rappresenta il ritardo in minuti all'arrivo. Se il ritardo è inferiore o uguale a zero, il volo viene considerato in orario. La percentuale viene calcolata come il rapporto tra i voli in orario e i voli totali, moltiplicato per 100. Se non ci sono voli disponibili nel periodo specificato, la percentuale viene impostata a zero.

```

1 @st.cache_data(show_spinner=False)
2 def percentuale_voli_ritardo_date(data_inizio=None, data_fine=
  None):
3     query = df.filter(col("ArrDelayMinutes").isNotNull())
4     if data_inizio and data_fine:
5         query = query.filter((col("FlightDate") >= data_inizio) &
6                               (col("FlightDate") <= data_fine))
7     voli_totali = query.count()
8     in_ritardo = query.filter(col("ArrDelayMinutes") > 0).count()
9     percentuale = (in_ritardo / voli_totali) * 100 if voli_totali
    > 0 else 0
    return round(percentuale, 2)

```

La funzione `percentuale_voli_ritardo_date` calcola la percentuale di voli arrivati in ritardo. Viene utilizzato un filtro simile alla funzione precedente, ma questa volta i voli con `ArrDelayMinutes` maggiore di zero vengono considerati in ritardo. La percentuale di voli in ritardo viene calcolata e restituita nel formato richiesto.

Il codice presenta anche l'uso della decorazione `@st.cache_data`, che serve a memorizzare in cache i risultati delle funzioni per evitare calcoli ripetuti e migliorare le prestazioni quando l'utente interagisce con l'applicazione.

Le ultime 4 sezioni della colonna di sinistra iniziano con la chiamata alla funzione `giorno_della_settimana_con_piu_voli_date`, che esegue una query per determinare il giorno della settimana con il maggior numero di voli. La query filtra i voli per il periodo di interesse, e raggruppa i risultati per giorno della settimana.

```

1 giorno_piu_voli = spark_to_pandas(
    giorno_della_settimana_con_piu_voli_date(data_inizio,
2     data_fine))
3 giorno_num = giorno_piu_voli.iloc[0]["DayOfWeek"]
4 max_dist = spark_to_pandas(volo_distanza_max(data_inizio,
    data_fine))
5 min_dist = spark_to_pandas(volo_distanza_min(data_inizio,
    data_fine))
6 totale_cancellazioni = totale_voli_cancellati_date(data_inizio,
    data_fine)

```

```

1 def giorno_della_settimana_con_piu_voli_date(data_inizio=None,
    data_fine=None):
2     query = df.filter(col("DayOfWeek").isNotNull())

```

```

3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     giorno = query.groupBy("DayOfWeek").count().orderBy(col("count").desc()).limit(1)
7     return giorno

```

La funzione `giorno_della_settimana_con_piu_voli_date` filtra i voli per il campo `DayOfWeek` e calcola quale giorno della settimana ha il maggior numero di voli, restituendo il risultato ordinato per il conteggio in ordine decrescente.

Successivamente, vengono calcolate la distanza massima e minima percorsa dai voli chiamando le funzioni `volo_distanza_max` e `volo_distanza_min`. Queste funzioni ordinano i voli in base alla distanza percorsa, rispettivamente in ordine decrescente per la distanza massima e crescente per la distanza minima.

```

1 def volo_distanza_max(data_inizio=None, data_fine=None):
2     query = df
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     volo = query.orderBy(col("Distance").desc()).limit(1)
7     return volo
8
9 def volo_distanza_min(data_inizio=None, data_fine=None):
10    query = df
11    if data_inizio and data_fine:
12        query = query.filter((col("FlightDate") >= data_inizio) &
13                              (col("FlightDate") <= data_fine))
14    volo = query.orderBy(col("Distance").asc()).limit(1)
15    return volo

```

Infine, la funzione `totale_voli_cancellati_date` restituisce il numero totale di voli cancellati, basandosi sul filtro della colonna `Cancelled` che indica se un volo è stato cancellato. Viene applicato un filtro temporale e viene restituito il totale dei voli cancellati.

```

1 @st.cache_data(show_spinner=False)
2 def totale_voli_cancellati_date(data_inizio=None, data_fine=None):
3     :
4     query = df.filter(col("Cancelled") == "1.00")
5     if data_inizio and data_fine:
6         query = query.filter((col("FlightDate") >= data_inizio) &
7                               (col("FlightDate") <= data_fine))
8     cancellati = query.count()
9     return cancellati

```

Colonna di Destra Continuando ora con la colonna di destra, nel primo grafico vengono calcolati e visualizzati i voli per mese per un periodo specificato. Inizialmente viene creato un dizionario che associa ogni numero di mese al nome corrispondente, e successivamente vengono organizzati i voli in base al mese e visualizzati tramite un grafico a barre.

```
1 st.subheader("Voli per Mese")
2
3 voli_per_mese = spark_to_pandas(totaleVoliPerMeseDate(data_inizio
4     , data_fine))
5 mesi = {
6     1: "Gennaio", 2: "Febbraio", 3: "Marzo", 4: "Aprile",
7     5: "Maggio", 6: "Giugno", 7: "Luglio", 8: "Agosto",
8     9: "Settembre", 10: "Ottobre", 11: "Novembre", 12: "Dicembre"
9 }
10 voli_per_mese["Month_Name"] = voli_per_mese["Month"].map(mesi)
11 voli_per_mese = voli_per_mese.sort_values("Month")
12 voli_per_mese["Month_Name"] = pd.Categorical(
13     voli_per_mese["Month_Name"], categories=mesi_ordinati,
14     ordered=True
15 )
16 voli_per_mese.set_index("Month_Name", inplace=True)
17
18 st.bar_chart(voli_per_mese["NumeroVoli"])
```

Il codice inizia con l'invocazione della funzione `totaleVoliPerMeseDate`, la quale esegue una query sui dati per calcolare il numero di voli per mese, in un intervallo di tempo definito dalle variabili `data_inizio` e `data_fine`. La funzione `totaleVoliPerMeseDate` filtra i voli in base alla colonna `Month` e poi raggruppa i risultati per mese, calcolando il numero di voli per ciascun mese. I risultati vengono ordinati per mese.

```
1 def totaleVoliPerMeseDate(data_inizio=None, data_fine=None):
2     query = df.filter(col("Month").isNotNull())
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     voli_per_mese = query.groupBy("Month").agg(count("*").alias("NumeroVoli")).orderBy("Month")
7     return voli_per_mese
```

Successivamente, viene creato un dizionario `mesi` che associa i numeri dei mesi ai loro nomi, in italiano. Il dataframe risultante dalla query viene modificato, aggiungendo una nuova colonna `Month_Name`, in cui il mese numerico viene mappato nel nome corrispondente. Successivamente, i dati vengono ordinati in base al mese e la colonna `Month_Name`

viene trasformata in una variabile categorica, con l'ordine dei mesi definito dalla lista `mesi_ordinati`.

```
1 mesi = {  
2     1: "Gennaio", 2: "Febbraio", 3: "Marzo", 4: "Aprile",  
3     5: "Maggio", 6: "Giugno", 7: "Luglio", 8: "Agosto",  
4     9: "Settembre", 10: "Ottobre", 11: "Novembre", 12: "Dicembre"  
5 }
```

Infine, il dataframe viene impostato con l'indice basato sui nomi dei mesi e viene utilizzato un grafico a barre tramite Streamlit per visualizzare il numero di voli per ogni mese.

```
1 st.bar_chart(voli_per_mese["NumeroVoli"])
```

Successivamente sono presenti tre tabelle che mostrano informazioni sulle rotte aeree più comuni, le tratte con più ritardi e le tratte con meno ritardi. Queste tabelle sono organizzate in tre colonne, ognuna delle quali contiene una tabella specifica.

```
1 tabella1, tabella2, tabella3 = st.columns(3)  
2  
3 with tabella1:  
4     st.markdown("#### Rotte piu' Comuni")  
5     rotte_comuni = spark_to_pandas(rottePiuComuni(data_inizio,  
6     st.dataframe(rotte_comuni, use_container_width=True,  
7     hide_index=True,)  
8  
9 with tabella2:  
10     st.markdown("#### Tratte con Piu' Ritardi")  
11     tratte_piu_ritardi = spark_to_pandas(  
12     tratte_con_piu_ritardi_totali(data_inizio, data_fine))  
13     st.dataframe(tratte_piu_ritardi, use_container_width=True,  
14     hide_index=True,)  
15  
16 with tabella3:  
17     st.markdown("#### Tratte con Meno Ritardi")  
18     tratte_meno_ritardi = spark_to_pandas(  
19     tratte_con_meno_ritardi_totali(data_inizio, data_fine))  
20     st.dataframe(tratte_meno_ritardi, use_container_width=False,  
21     hide_index=True,)
```

Il codice inizia creando tre colonne tramite la funzione `st.columns(3)`, in cui ogni colonna ospiterà una tabella specifica.

Per la prima tabella, vengono visualizzate le rotte più comuni. I dati vengono ottenuti

tramite la funzione `rottePiuComuni`, che esegue una query sui dati, raggruppando i voli per `Origin` (origine) e `Dest` (destinazione), e ordinando i risultati per il numero di voli. La tabella mostra le prime dieci rotte più comuni.

```
1 def rottePiuComuni(data_inizio=None, data_fine=None):
2     query = df.filter(col("Origin").isNotNull() & col("Dest").
3         isNotNull())
4     if data_inizio and data_fine:
5         query = query.filter((col("FlightDate") >= data_inizio) &
6             (col("FlightDate") <= data_fine))
7     rotte = query.groupBy("Origin", "Dest").agg(count("*").alias(
8         "NumeroVoli"))
9     rotte_piu_comuni = rotte.orderBy(col("NumeroVoli").desc()).
10        limit(10)
11    return rotte_piu_comuni
```

Per la seconda tabella, vengono visualizzate le tratte con più ritardi. La funzione `tratte_con_piu_ritardi_totali` calcola i ritardi totali in partenza e arrivo per ogni rotta, sommandoli e ordinando le tratte per il totale dei minuti di ritardo. Vengono visualizzate le prime dieci tratte con il maggiore ritardo totale.

```
1 def tratte_con_piu_ritardi_totali(data_inizio=None, data_fine=
2     None):
3     query = df.filter((col("ArrDelayMinutes").isNotNull()) & (col(
4         "DepDelayMinutes").isNotNull()))
5     if data_inizio and data_fine:
6         query = query.filter((col("FlightDate") >= data_inizio) &
7             (col("FlightDate") <= data_fine))
8     tratte = query.groupBy("Origin", "Dest").agg(
9         sum("DepDelayMinutes").alias("MinutiRitardoPartenza"),
10        sum("ArrDelayMinutes").alias("MinutiRitardoArrivo")
11    ).withColumn(
12        "TotaleMinutiRitardo", col("MinutiRitardoPartenza") + col(
13            "MinutiRitardoArrivo")
14    ).select("Origin", "Dest", "TotaleMinutiRitardo").orderBy(
15        desc("TotaleMinutiRitardo")).limit(10)
16    return tratte
```

Infine, la terza tabella visualizza le tratte con meno ritardi. La funzione `tratte_con_meno_ritardi_t` è simile alla precedente, ma questa volta le tratte vengono ordinate in ordine crescente di ritardo totale, visualizzando quelle con il minore ritardo.

```
1 def tratte_con_meno_ritardi_totali(data_inizio=None, data_fine=
2     None):
3     query = df.filter(col("ArrDelayMinutes").isNotNull())
```



```

3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
5                               (col("FlightDate") <= data_fine))
6     tratte = query.groupBy("Origin", "Dest").agg(
7         sum("DepDelayMinutes").alias("MinutiRitardoPartenza"),
8         sum("ArrDelayMinutes").alias("MinutiRitardoArrivo")
9     ).withColumn(
10        "TotaleMinutiRitardo", col("MinutiRitardoPartenza") + col(
11            "MinutiRitardoArrivo")
12    ).select("Origin", "Dest", "TotaleMinutiRitardo").orderBy(asc
13        ("TotaleMinutiRitardo")).limit(10)
14    return tratte

```

Ogni funzione esegue una query sui dati, estraendo e aggregando le informazioni necessarie per ciascun caso. I risultati vengono poi convertiti in un dataframe pandas tramite `spark_to_pandas` e visualizzati utilizzando `st.dataframe`.

Scendendo con le funzioni si trova una mappa in cui è possibile selezionare un aereo, visualizzare le tratte distinte per tale aereo, e disegnare le tratte su una mappa se le coordinate sono disponibili. L'utente seleziona un aereo tramite un `selectbox`, e le tratte corrispondenti vengono mostrate su una mappa interattiva. Se le coordinate non sono trovate, viene visualizzato un avviso.

```

1 aerei_disponibili = spark_to_pandas(get_aerei_disponibili(
2     data_inizio.strftime("%Y-%m-%d"), data_fine.strftime("%Y-%m-%d")
3 ))
4 selected_tail_number = st.selectbox("Seleziona un Aereo",
5     aerei_disponibili)
6 tratte_distinte = tratte_distinte_per_aereo(selected_tail_number,
7     data_inizio.strftime("%Y-%m-%d"), data_fine.strftime("%Y-%m-%d"))
8 tratte_pd = spark_to_pandas(tratte_distinte)
9 coordinate_df = load_coordinate_data()
10 tratte_coords = get_coordinates(tratte_pd, coordinate_df)
11
12 if not tratte_coords.empty:
13     disegna_tratta(tratte_coords, col_graph)
14 else:
15     st.warning("Nessuna tratta trovata o coordinate mancanti.")

```

Il codice inizia con la selezione degli aerei disponibili per il periodo specificato, utilizzando la funzione `get_aerei_disponibili`. Gli aerei vengono ottenuti dalla tabella `df` filtrata per il periodo di interesse e l'output è una lista di numeri di coda (Tail Number) distinti.

```

1 def get_aerei_disponibili(data_inizio=None, data_fine=None):
2     if data_inizio and data_fine:
3         query = df.filter((col("FlightDate") >= data_inizio) & (
4             col("FlightDate") <= data_fine)).select("Tail_Number")
5             .distinct()
6         query = df.select("Tail_Number").distinct().filter(col("
7             Tail_Number").isNotNull())
8     return query

```

Una volta selezionato un aereo tramite il `selectbox`, viene chiamata la funzione `tratte_distinte_per_aereo`, che filtra il dataframe `df` per il numero di coda selezionato, estraendo le tratte distintive per quell'aereo, ovvero le rotte uniche da origine a destinazione.

```

1 def tratte_distinte_per_aereo(tail_number, data_inizio=None,
2     data_fine=None):
3     query = df.filter((col("Tail_Number") == tail_number) & col("
4         Origin").isNotNull() & col("Dest").isNotNull())
5     if data_inizio and data_fine:
6         query = query.filter((col("FlightDate") >= data_inizio) &
7             (col("FlightDate") <= data_fine))
8     tratte_distinte = query.select("Origin", "OriginCityName", "
9         Dest", "DestCityName").distinct()
10    return tratte_distinte

```

Le coordinate delle tratte vengono quindi caricate tramite la funzione `load_coordinate_data`, che legge un file CSV contenente i dati delle coordinate aeroportuali. Successivamente, viene utilizzata la funzione `get_coordinates` per ottenere le coordinate delle tratte selezionate.

```

1 @st.cache_data(show_spinner=False)
2 def load_coordinate_data():
3     return pd.read_csv(coordinate_aeroporto)

```

Se vengono trovate le coordinate per le tratte, la funzione `disegna_tratta` viene utilizzata per visualizzare la mappa, mostrando le tratte selezionate. Se non vengono trovate tratte o coordinate, viene visualizzato un avviso tramite `st.warning`.

Successivamente si incontrerà un doppio menù che consente di selezionare due aeroporti (partenza e destinazione) e calcolare la velocità media per la tratta tra di essi. Se la tratta non è stata percorsa da alcun aereo, viene visualizzato un messaggio indicante tale situazione.

```

1 def velocita_media_per_tratta_specifica(origin, dest):
2     velocita_filtrate = df.filter((col("Origin") == origin) & (
3         col("Dest") == dest) &

```

```

3         col("Distance").isNotNull() &
           col("AirTime").isNotNull())
4     velocita = velocita_filtrate.withColumn("AverageSpeed", col("
      Distance") / (col("AirTime") / 60)) \
5         .agg(avg("AverageSpeed").alias("
      AverageSpeedForRoute"))
6     return velocita

```

La funzione `velocita_media_per_tratta_specifica` calcola la velocità media per una data tratta (da un aeroporto di partenza a uno di destinazione). Filtra il dataframe per ottenere i voli che corrispondono a quella specifica tratta e calcola la velocità media, utilizzando la distanza percorsa e il tempo di volo.

Se la query per la velocità restituisce un risultato non vuoto, viene estratto il valore della velocità media e visualizzato nella UI di Streamlit. Se la velocità non è disponibile (ad esempio, se nessun aereo ha percorso quella tratta), viene mostrato un messaggio che indica che la tratta non è stata percorsa.

Come ultima sezione della pagina è possibile selezionare il tipo di analisi da visualizzare tramite un controllo a segmenti (`segmented_control`), offrendo due opzioni: "Giorno della Settimana" e "Causa Cancellazione". A seconda della selezione, viene generato un grafico a barre che rappresenta rispettivamente le cancellazioni per giorno della settimana o le cancellazioni per causa.

```

1 def cancellazioniPerGiorno(data_inizio=None, data_fine=None,
   giorno=None):
2     query = df
3     if data_inizio and data_fine:
4         query = query.filter((col("FlightDate") >= data_inizio) &
           (col("FlightDate") <= data_fine))
5
6     if giorno:
7         cancellazioni = query.filter(col("DayOfWeek") == giorno)
           \
8             .select(sum("Cancelled").alias("TotaleCancellazioni")
           )
9         risultato = cancellazioni.collect()
10        if risultato:
11            totale = risultato[0]["TotaleCancellazioni"]
12            return totale
13        return None
14    else:
15        return query.groupBy("DayOfWeek") \

```

```

16         .agg(sum("Cancelled").alias("Cancellazioni")) \
17         .orderBy("DayOfWeek")

```

La funzione `cancellazioniPerGiorno` restituisce il numero di cancellazioni per ciascun giorno della settimana. Se è specificato un giorno, vengono filtrati i dati per quel giorno e viene calcolato il totale delle cancellazioni. Se non viene specificato un giorno, la funzione restituisce le cancellazioni aggregate per giorno della settimana.

```

1 def cancellazioniPerCausa(data_inizio=None, data_fine=None, causa
  =None):
2     query = df.filter(col("Cancelled") == 1)
3
4     if data_inizio and data_fine:
5         query = query.filter((col("FlightDate") >= data_inizio) &
6                               (col("FlightDate") <= data_fine))
7
8     if causa:
9         cancellazioni = query.filter(col("CancellationCode") ==
10                                     causa) \
11                                .select(count("*").alias("TotaleCancellazioni"))
12         risultato = cancellazioni.collect()
13         if risultato:
14             totale = risultato[0]["TotaleCancellazioni"]
15             return totale
16         return None
17     else:
18         return query.groupBy("CancellationCode") \
19                        .count() \
20                        .orderBy("count", ascending=False)

```

La funzione `cancellazioniPerCausa` restituisce il numero di cancellazioni per ciascuna causa di cancellazione. Le cause possibili includono "Compagnia Aerea", "Meteo", "Sistema Nazionale" e "Sicurezza". Se viene specificata una causa, viene restituito il numero di cancellazioni per quella causa. Altrimenti, vengono restituite le cancellazioni aggregate per causa.

8 Sezione AI

Algoritmi di Machine Learning applicati ai dati

Scegliere il tipo di analisi

☒ Clustering degli aeroporti ☐ Clustering delle rotte

Analisi Cluster degli Aeroporti

Questa analisi raggruppa gli aeroporti in cluster basati su:

- Volume totale di voli
- Ritardo medio
- Distanza media dei voli
- Tasso di cancellazione

Scegliere il numero di cluster

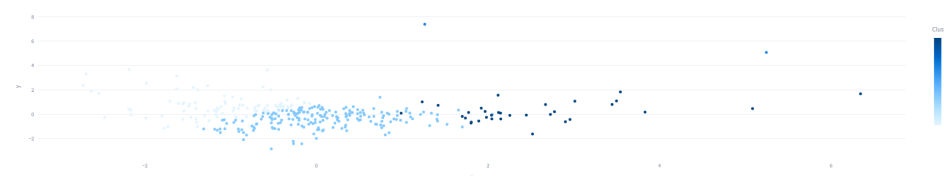


Statistiche dei Cluster

cluster	Media Voli Totali	Media Ritardo (min)	Distanza Media (miglia)	Tasso Cancellazione	Numero Aeroporti
0	5,335.34	6.29	324.79	0.04	33
1	20,133.69	5.7	487.27	0.02	232
2	106.5	49.86	1,344.5	0	2
3	253,844.18	7.55	1,022.69	0.01	33

[Panoramica Generale](#) [Analisi Dettagliata](#)

Panoramica Generale dei Cluster



Questo grafico mostra una visione d'insieme dei cluster, dove:

- Ogni punto rappresenta un aeroporto
- La posizione è determinata da tutte le caratteristiche considerate
- Aeroporti vicini hanno caratteristiche simili
- I colori indicano i diversi cluster

Aeroporti per Cluster

Scegliere un cluster da visualizzare:

0

aeroporto	voli_totale	ritardo_medio	distanza_medio	tasso_cancellazione	cluster
VNA	25,471	11.5235	539.6839	0.0389	0
TYS	24,414	12.0895	476.6332	0.0347	0
GSO	20,858	8.7252	423.9492	0.0274	0
CDI	16,714	9.0562	478.8489	0.0283	0
SHV	15,080	8.3848	334.7678	0.0268	0
AMA	14,898	10.5671	374.4325	0.0261	0
PSD	14,758	10.9459	428.0647	0.0256	0
CAC	14,035	12.7286	457.6911	0.0243	0
CRP	13,968	9.0999	255.8633	0.0243	0
FAH	13,532	10.5471	562.0309	0.0226	0

La pagina IA si concentra su un'analisi di clustering, applicata sia agli aeroporti che alle rotte aeree. Viene utilizzato il Principal Component Analysis (PCA) per ridurre la dimensionalità dei dati e visualizzare i cluster in uno spazio bidimensionale. Inoltre, vengono eseguiti calcoli statistici per ottenere una panoramica generale e dettagliata dei cluster, sia per gli aeroporti che per le rotte.

L'analisi sui cluster degli aeroporti si concentra su quattro caratteristiche principali:

- Volume totale di voli
- Ritardo medio

- Distanza media dei voli
- Tasso di cancellazione

Inizialmente, viene selezionato il numero di cluster tramite uno slider. Successivamente, il codice esegue il clustering sugli aeroporti e calcola le statistiche medie per ogni cluster, come il numero medio di voli, il ritardo medio e la distanza media. Queste informazioni vengono poi visualizzate in una tabella.

Dopo aver raggruppato gli aeroporti, il codice applica la tecnica di PCA per ridurre le dimensioni dei dati a due componenti principali, consentendo di visualizzare i cluster in un grafico a dispersione.

Ogni punto rappresenta un aeroporto, e la posizione è determinata dalle caratteristiche considerate, mentre i colori rappresentano i diversi cluster. Il codice per la creazione del grafico generale sui cluster degli aeroporti è questo:

```

1 def crea_overview_cluster(clusters_df):
2     features = ["voli_totali", "ritardo_medio", "distanza_media",
3                 "rate_cancellazione"]
4     X = clusters_df[features].values
5
6     scaler = SklearnScaler()
7     X_scaled = scaler.fit_transform(X)
8
9     pca = PCA(n_components=2)
10    X_pca = pca.fit_transform(X_scaled)
11
12    # Crea DataFrame per la visualizzazione
13    vis_df = pd.DataFrame(X_pca, columns=['x', 'y'])
14    vis_df['Cluster'] = clusters_df['cluster']
15    vis_df['Aeroporto'] = clusters_df['airport_code']
16    vis_df['Voli Totali'] = clusters_df['voli_totali']
17    vis_df['Ritardo Medio'] = clusters_df['ritardo_medio']
18    vis_df['Distanza Media'] = clusters_df['distanza_media']
19    vis_df['Tasso Cancellazioni'] = clusters_df['
20        rate_cancellazione']
21
22    fig = px.scatter(
23        vis_df,
24        x='x',
25        y='y',
26        color='Cluster',
27        hover_data=['Aeroporto', 'Voli Totali', 'Ritardo Medio',
28                    'Distanza Media', 'Tasso Cancellazioni'],
29    )

```

27

28

```
return fig
```

Per un'analisi dettagliata dei cluster, viene inclusa la possibilità di selezionare variabili specifiche da visualizzare, come "Voli vs Ritardi", "Voli vs Distanza" e "Ritardi vs Cancellazioni".

Similmente all'analisi sugli aeroporti, l'analisi delle rotte aeree si concentra su un insieme di caratteristiche, tra cui:

- Volume totale di voli
- Distanza della rotta
- Ritardo medio
- Varianza del ritardo
- Tasso di cancellazione
- Tasso di dirottamento
- Tempo medio di volo

Anche in questo caso, viene eseguito un processo di standardizzazione e di riduzione delle dimensioni tramite PCA, per visualizzare i cluster in uno spazio bidimensionale. I dati vengono poi visualizzati in un grafico a dispersione, in cui ogni punto rappresenta una rotta, e la posizione del punto è determinata dalle caratteristiche delle rotte. Per la creazione della panoramica generale sui cluster delle rotte viene usata questa funzione:

```
1 def crea_overview_rotte(clusters_df):
2     features = ["voli_totali", "distanza", "ritardo_medio", "
3                 varianza_ritardo", "rate_cancellazione", "rate_dirottamento"
4                 , "media_tempo_volo"]
5     X = clusters_df[features].values
6
7     scaler = SklearnScaler()
8     X_scaled = scaler.fit_transform(X)
9
10    pca = PCA(n_components=2)
11    X_pca = pca.fit_transform(X_scaled)
12
13    vis_df = pd.DataFrame(X_pca, columns=['x', 'y'])
14    vis_df['Cluster'] = clusters_df['cluster']
15    vis_df['Origine'] = clusters_df['Origin'] + " (" +
16                        clusters_df['OriginCityName'] + ")"
```

```

14 vis_df['Destinazione'] = clusters_df['Dest'] + " (" +
    clusters_df['DestCityName'] + ")"
15 vis_df['Voli Totali'] = clusters_df['voli_totali']
16 vis_df['Distanza'] = clusters_df['distanza']
17 vis_df['Ritardo Medio'] = clusters_df['ritardo_medio']
18
19 fig = px.scatter(
20     vis_df,
21     x='x',
22     y='y',
23     color='Cluster',
24     hover_data=['Origine', 'Destinazione', 'Voli Totali', '
        Distanza', 'Ritardo Medio'],
25 )
26
27 return fig

```

Infine, la funzione `cluster_routes` esegue il clustering delle rotte utilizzando un insieme di caratteristiche multiple, inclusi variabili come "ritardo medio" e "tasso di dirottamento".

Viene utilizzato il K-means per eseguire il clustering, e i dati vengono poi trasformati in un formato Pandas per facilitarne l'analisi successiva.

```

1 def cluster_routes(numero_cluster):
2     features_df = prepare_route_features()
3     assembler = VectorAssembler(
4         inputCols=["voli_totali", "distanza", "ritardo_medio", "
            varianza_ritardo", "rate_cancellazione", "
            rate_dirottamento", "media_tempo_volo"],
5         outputCol="features"
6     )
7     scaler = StandardScaler(inputCol="features", outputCol="
        scaled_features", withStd=True, withMean=True)
8
9     kmeans = KMeans(k=numero_cluster, featuresCol="
        scaled_features", predictionCol="cluster")
10
11     vector_df = assembler.transform(features_df)
12     scaled_df = scaler.fit(vector_df).transform(vector_df)
13     clusters = kmeans.fit(scaled_df).transform(scaled_df)
14
15     return clusters.select(
16         "Origin", "Dest", "OriginCityName", "DestCityName", "
            voli_totali", "distanza", "ritardo_medio",

```



```

17         "varianza_ritardo", "rate_cancellazione", "
18         rate_dirottamento", "media_tempo_volo", "cluster"
        ).toPandas()

```

L'operazione di clustering fornisce una comprensione approfondita delle caratteristiche delle rotte aeree e degli aeroporti.

Infine la tabella alla fine della pagina si occupa della visualizzazione e analisi di cluster sia per gli aeroporti che per le rotte aeree. L'interfaccia permette di selezionare i cluster e visualizzare statistiche per ciascun cluster.

Il codice offre quindi un'analisi interattiva dei cluster, sia per gli aeroporti che per le rotte, permettendo di esplorare le caratteristiche statistiche di ciascun cluster e di visualizzare i dati tramite grafici a dispersione, con la possibilità di eseguire selezioni multiple e di visualizzare dati specifici tramite il filtro dei cluster.

References

- [1] Rebollo, J., & Balakrishnan, H. (2014). Characterization and prediction of air traffic delays. *Transportation Research Part C: Emerging Technologies*, 44, 231-241.
- [2] Kim, Y. J., & Choi, S. (2016). Analyzing airline flight delays using machine learning algorithms. *Journal of Air Transport Management*, 52, 42-52.
- [3] Gopalakrishnan, K., & Balakrishnan, H. (2017). A comparative analysis of models for predicting delays in air traffic networks. *Transportation Research Part C: Emerging Technologies*, 84, 214-224.
- [4] Williams, J. K. (2017). *Airline Operations and Delay Management: Insights from Airline Economics, Networks and Strategic Schedule Planning*. Routledge.
- [5] Chen, J., & Li, M. (2021). Deep learning approaches for flight delay prediction using the BTS dataset. *Transportation Research Part E: Logistics and Transportation Review*, 149, 102289.