

**TEMPLATE
PROJECT WORK**

| | |
|--|--|
| Corso di Studio | INFORMATICA PER LE AZIENDE DIGITALI (L-31) |
| Dimensione dell'elaborato | Minimo 6.000 – Massimo 10.000 parole (<i>pari a circa Minimo 12 – Massimo 20 pagine</i>) |
| Formato del file da caricare in piattaforma | PDF |
| Nome e Cognome | Alex Di Paolo |
| Numero di matricola | 0312300365 |
| Tema n. (Indicare il numero del tema scelto): | 5 |
| Titolo del tema (Indicare il titolo del tema scelto): | Machine Learning per Processi Aziendali |
| Traccia del PW n. (Indicare il numero della traccia scelta): | 18 |
| Titolo della traccia (Indicare il titolo della traccia scelta): | Triage automatico dei ticket con Machine Learning |
| Titolo dell'elaborato (Attribuire un titolo al proprio elaborato progettuale): | AutoTriage NLP: Sistema intelligente di classificazione e prioritarizzazione ticket per l'assistenza aziendale |

PARTE PRIMA – DESCRIZIONE DEL PROCESSO

Utilizzo delle conoscenze e abilità derivate dal percorso di studio

(Descrivere quali conoscenze e abilità apprese durante il percorso di studio sono state utilizzate per la redazione dell'elaborato, facendo eventualmente riferimento agli insegnamenti che hanno contribuito a maturarle):

L'idea alla base di questo elaborato nasce da un'esigenza molto concreta delle aziende: ottimizzare i tempi di risposta e la gestione delle richieste nel supporto clienti. Partendo da questo presupposto, ho sviluppato il progetto **AutoTriage NLP**, un sistema basato sul Machine Learning pensato per automatizzare lo smistamento dei ticket. Nel realizzarlo, ho cercato di bilanciare le sfide tecniche dell'elaborazione del linguaggio naturale con le normali necessità e i vincoli di un'impresa reale. Per farlo, ho curato l'intera architettura software, dalla preparazione dei dati fino alla creazione di un'interfaccia utente interattiva.

L'obiettivo principale del progetto è dimostrare a livello pratico come gestire, manipolare e analizzare dati testuali non strutturati utilizzando Python. Non volevo limitarmi a un semplice esercizio tecnico, ma ho cercato di far convergere in questo lavoro gran parte delle competenze multidisciplinari apprese durante il Corso di Laurea in Informatica per le Aziende Digitali.

Di seguito, riassumo come i vari insegnamenti mi sono stati utili per costruire le diverse componenti del sistema:

1. Programmazione e Algoritmica

Le basi di programmazione sono state fondamentali per la stesura del codice.

- **Programmazione 2:** È stato l'insegnamento centrale, dato che il progetto è scritto interamente in Python. Ho applicato quanto studiato sulle strutture dati e utilizzato ampiamente librerie come Pandas e NumPy per pulire il dataset scaricato da Kaggle (nel file **prepare_data.py**). Sempre grazie a questo corso, ho usato Matplotlib e Seaborn per i grafici della dashboard e implementato blocchi **try-except** per gestire in sicurezza le eccezioni nella pipeline di traduzione.
- **Programmazione 1:** Anche se incentrato sul C, mi ha dato l'impostazione logica per il problem solving. Ho usato questo approccio strutturato per la pipeline di preprocessing, organizzando il codice in moduli separati (nella cartella **src/**) per mantenerlo pulito e leggibile.
- **Algoritmi e Strutture Dati:** Mi è servito per scegliere le strutture dati più adatte, come dizionari e hash map, per gestire i vocabolari in modo efficiente. Inoltre, per evitare che la traduzione in batch di CSV molto grandi diventasse un collo di bottiglia, ho parallelizzato le operazioni con **ThreadPoolExecutor**, mettendo in pratica i concetti di complessità computazionale.

2. Ingegneria del Software e Basi di Dati

Ho cercato di scrivere il codice pensando fin da subito alla sua manutenzione e scalabilità.

- **Ingegneria del Software:** Mi ha dato il metodo. Ho diviso il progetto in layer distinti (Data, Model e Frontend) e utilizzato i concetti dei design pattern per creare la classe **UnifiedModel**, che gestisce due classificatori paralleli mantenendo l'interfaccia pulita.
- **Basi di Dati:** Anche se per comodità ho usato file CSV invece di un database SQL tradizionale, ho applicato le regole di normalizzazione per ristrutturare il dataset originale in **tickets_it_augmented.csv**. Questo mi ha garantito di lavorare su dati di training coerenti, integri e senza duplicati.

3. Matematica e Statistica

I modelli predittivi utilizzati si basano su fondamenti matematici ben precisi.

- **Calcolo delle Probabilità e Statistica:** Senza studiare le distribuzioni di probabilità, sarebbe stato difficile comprendere a fondo metriche come la TF-IDF o capire come la Regressione Logistica stima la probabilità di assegnare un ticket a una certa categoria. Inoltre, tutte le metriche usate nel report (Accuracy, Precision, Recall, F1-Score) e la lettura della matrice di confusione derivano direttamente da questo esame.
- **Matematica Discreta e Analisi Matematica:** L'algebra lineare mi ha permesso di capire cosa succede dietro le quinte quando il testo viene vettorizzato o quando l'algoritmo calcola le distanze tra le categorie. Perfino la libreria LIME, usata per rendere il modello interpretabile, si basa sullo studio dello spazio vettoriale.

4. Web e Tecnologie di Rete

Per rendere il sistema facilmente testabile, ho creato una web app.

- **Tecnologie Web:** Ho usato le conoscenze di HTML e CSS per personalizzare l'interfaccia grafica (tramite il file **style.css**). Inoltre, aver chiaro come funziona il protocollo HTTP e l'architettura Client-Server mi ha aiutato a capire come il framework Streamlit gestisce le interazioni dell'utente e come integrare le API esterne di traduzione senza interrompere il flusso dell'applicazione.

5. Area Giuridica ed Economica

Ci tengo a evidenziare che il progetto non si limita a essere un prototipo tecnico, ma mira a risolvere un problema di business reale nel totale rispetto dell'impianto normativo vigente.

- **Diritto per le Aziende Digitali:** Essendo consapevole dei vincoli del GDPR in materia di privacy, ho deciso di non fare scraping di dati reali per evitare rischi legati al trattamento di informazioni personali. Applicando il principio di "Privacy by Design", ho optato per un dataset pubblico open source già anonimizzato, creando un sistema sicuro e "compliant" dalla base.
- **Strategia, Organizzazione e Marketing:** L'idea di un triage automatico ha senso se porta un reale vantaggio in termini di efficienza. Ho inserito un calcolo del ROI nel report proprio per dimostrare come questa automazione possa ridurre i colli di bottiglia e abbattere i costi operativi, unendo l'aspetto tecnico a quello aziendale.

In sintesi, questo progetto rappresenta per me il vero punto d'incontro del mio percorso universitario: la teoria statistica e la pratica della programmazione si uniscono in uno strumento fruibile, calato in un contesto aziendale e normativo realistico.

Fasi di lavoro e relativi tempi di implementazione per la predisposizione dell'elaborato

(Descrivere le attività svolte in corrispondenza di ciascuna fase di redazione dell'elaborato. Indicare il tempo dedicato alla realizzazione di ciascuna fase, le difficoltà incontrate e come sono state superate):

Lo sviluppo pratico del progetto mi ha impegnato per circa quattro settimane, per un totale stimato di 80 ore di lavoro. Per gestire al meglio le tempistiche e non perdermi nella complessità dell'architettura, ho preferito strutturare fin dall'inizio l'attività in cinque macro-fasi.

Fase 1: Scelta della traccia, ricerca e setup (circa 10 ore)

Inizialmente ho analizzato le varie tracce proposte e ho scelto di dedicarmi al progetto “AutoTriage NLP”, perché mi sembrava il caso d'uso più attuale e spendibile in un contesto aziendale reale. Ho iniziato raccogliendo il materiale didattico tramite la biblioteca online di UniPegaso (in particolare su Pearson Bookshelf) e ripassando le dispense di Programmazione 2. All'inizio temevo di allargare troppo il perimetro del progetto e di non rientrare nei tempi, ma le sessioni di didattica interattiva con i docenti mi sono state di grande aiuto per definire dei confini chiari e focalizzarmi sull'obiettivo principale: ottenere un prototipo funzionante. In parallelo, ho configurato l'ambiente di sviluppo su Antygravity per iniziare a scrivere i primi script.

Fase 2: Analisi del problema e preparazione dei dati (circa 20 ore)

Prima di toccare gli algoritmi, ho studiato le dinamiche reali di un Help Desk aziendale. Volevo evitare di usare dati generati casualmente, perché avrebbero reso il sistema poco verosimile, così ho recuperato da Kaggle il dataset “Customer Support Ticket”. Essendo in inglese, ho dovuto scrivere uno script (**prepare_data.py**) per tradurre massivamente oltre 20.000 ticket in italiano appoggiandomi alle API di **deep_translator**. Qui ho incontrato la prima vera difficoltà: c'era il rischio che la traduzione automatica stravolgesse il senso dei termini tecnici (ad esempio “billing issue”). Per evitare blocchi delle API dovuti alle troppe richieste e migliorare la qualità testuale, ho implementato un sistema di caching delle traduzioni e applicato dei filtri Regex per ripulire il testo. Infine, ho mappato i ticket sulle tre categorie di mio interesse (Amministrazione, Tecnico, Commerciale), ottenendo il dataset finale **tickets_it_augmented.csv**.

Fase 3: Sviluppo del modello di Machine Learning (circa 25 ore)

Questa è stata la fase più lunga e complessa, concentrata nel file **train_unified_model.py**. Dopo aver impostato il preprocessing del testo (rimozione di punteggiatura e stopwords), ho strutturato il modello. Invece di addestrare due classificatori separati, ho creato una classe **UnifiedModel** basata su un **MultiOutputClassifier**, in modo che il sistema potesse classificare sia la Categoria che la Priorità simultaneamente, ottimizzando i tempi di calcolo. Lavorando con dati reali, ho dovuto gestire un forte sbilanciamento delle classi, che rischiava di portare il modello in overfitting. Ho affrontato il problema dividendo i dati con un classico split 80/20 per il training e il test, monitorando costantemente metriche come Accuracy e F1-Score e analizzando le matrici di confusione per capire dove l'algoritmo facesse più fatica a distinguere le categorie.

Fase 4: Interfaccia Web e LIME (circa 15 ore)

Per non limitare il progetto a un semplice script da terminale, ho sviluppato una web app interattiva usando Streamlit (**app.py**). Il mio scopo principale in questa fase era integrare la libreria LIME per rendere il modello “interpretabile”. Spiegare le decisioni di un modello matematico a un utente non tecnico non è semplice (il classico problema della “black-box”), ma LIME mi ha permesso di mostrare visivamente all’operatore quali specifiche parole (es. “fattura” o “virus”) avessero spinto l’algoritmo a prendere una determinata decisione. Oltre a questo, per rendere il sistema ancora più utile a livello pratico, ho inserito una logica ibrida per il calcolo della priorità che unisce le previsioni statistiche a regole di sicurezza fisse basate su parole chiave.

Fase 5: Test, validazione e stesura del report (circa 10 ore)

L’ultima fase è stata dedicata al collaudo. Ho effettuato dei test funzionali sull’interfaccia inserendo di proposito ticket molto brevi o ambigui per testare la tenuta del sistema, e ho sistemato gli ultimi bug. Dopodiché, ho generato i grafici comparativi (come quelli tra dati sintetici e reali) e mi sono dedicato alla stesura di questo report. La sfida conclusiva è stata sintetizzare tutto il lavoro tecnico in una relazione scorrevole e conforme alle richieste dell’Ateneo, seguendo scrupolosamente il template fornito e curando con attenzione la bibliografia.

Risorse e strumenti impiegati

(Descrivere quali risorse - bibliografia, banche dati, ecc. - e strumenti - software, modelli teorici, ecc. - sono stati individuati ed utilizzati per la redazione dell’elaborato. Descrivere, inoltre, i motivi che hanno orientato la scelta delle risorse e degli strumenti, la modalità di individuazione e reperimento delle risorse e degli strumenti, le eventuali difficoltà affrontate nell’individuazione e nell’utilizzo di risorse e strumenti ed il modo in cui sono state superate):

Per la realizzazione di questo elaborato, ho selezionato strumenti e librerie cercando il miglior compromesso tra le richieste didattiche e l’effettiva utilità in un contesto lavorativo. Di seguito presento le risorse e le tecnologie impiegate per lo sviluppo del progetto.

1. Risorse e Dati

- **Dataset Kaggle (“Customer Support Ticket”)**: Per addestrare il modello avevo bisogno di dati verosimili. Invece di generare ticket fittizi che avrebbero introdotto distorsioni (bias), ho preferito usare un dataset pubblico contenente oltre 20.000 richieste di assistenza reali e già anonimizzate. Questa scelta mi ha permesso di lavorare su un caso di studio statisticamente valido e di evitare in partenza le problematiche legate alla privacy e alla raccolta di dati sensibili aziendali.
- **Traduzione e Data Augmentation**: Poiché il dataset originale era in inglese, ho utilizzato le API di Google Translate (tramite la libreria **deep-translator**) non solo come semplice traduttore, ma come vero e proprio strumento di **Data Augmentation**, per convertire e adattare l’intero corpus testuale in italiano rendendolo coerente per il training del modello.

- **Bibliografia e Fonti:** Per consolidare le basi teoriche e pratiche mi sono affidato a diversi testi accademici e manuali, consultati principalmente tramite la piattaforma universitaria Pearson Bookshelf, oltre alla documentazione ufficiale online delle librerie utilizzate:
 - *Gaddis, T. (2016). Introduzione a Python. Pearson.*
 - *Zinoviev, D. (2017). Data science con Python. Apogeo.*
 - *Maggi, G. (2020). Data science con Python. Edizioni LSWR.*
 - *Marin, I., et al. (2019). L'analisi dei Big Data con Python. Tecniche nuove.*

2. Strumenti Tecnologici e Librerie

- **Python 3.10 e Pandas:** Ho scelto Python come linguaggio base per la sua comodità nella gestione delle stringhe e la rapidità di sviluppo. Per la manipolazione dei dati mi sono appoggiato quasi esclusivamente a Pandas: è stato fondamentale per importare il grosso file CSV da Kaggle, strutturarlo in Dataframe, pulire il testo e gestire l'I/O tra la fase di traduzione e quella di addestramento.
- **Scikit-Learn:** Piuttosto che scomodare framework molto pesanti (come TensorFlow o PyTorch), ho optato per Scikit-Learn. Seguendo il principio della soluzione più semplice ed efficace, questa libreria si è rivelata perfetta e performante per un dataset testuale di medie dimensioni. L'ho utilizzata per gestire l'intera pipeline: dalla tokenizzazione alla vettorizzazione TF-IDF, fino alla classificazione di Categoria e Priorità.
- **Explainability con LIME:** Per evitare l'effetto "black-box" tipico dei modelli di Machine Learning, ho integrato la libreria LIME. Questo strumento mi ha permesso di rendere trasparente l'algoritmo, mostrando all'utente quali parole esatte (es. "bloccato", "urgente") hanno pesato maggiormente sulla classificazione finale di un ticket.
- **Streamlit:** Per rendere il progetto fruibile ho creato un'interfaccia web con Streamlit. È stata una scelta strategica: mi ha permesso di sviluppare una dashboard interattiva direttamente in Python, senza dover disperdere tempo e codice scrivendo file HTML, CSS o JavaScript separati, concentrandomi così solo sulla logica del modello.
- **Ambiente di Sviluppo e Versionamento:** Ho scritto il codice sulla piattaforma cloud Antygravity, sfruttando le sue funzionalità di autocompletamento e debugging per snellire il lavoro. Per il versionamento ho utilizzato Git: mi ha garantito un flusso di lavoro sicuro, permettendomi di testare nuove funzionalità senza "rompere" la versione stabile. Inoltre, l'uso di Git ha soddisfatto la richiesta di rendere open source il progetto. Tutto il codice sorgente e la documentazione sono consultabili sul mio repository GitHub al seguente indirizzo: <https://github.com/dipaoloalex-dev/AutoTriageNLP>.

Durante lo sviluppo pratico ho dovuto affrontare un paio di criticità tecniche rilevanti. La prima ha riguardato la fase di traduzione massiva del dataset: interrogando continuamente le API, incappavo spesso in blocchi dovuti al rate-limiting. Per risolvere, ho dovuto ingegnerizzare uno script di batch processing, inserendo dei ritardi programmati (**sleep**) e una gestione rigorosa delle eccezioni (**try-except**) per far sì che il processo non si interrompesse a metà.

La seconda criticità è emersa in fase di visualizzazione sulla web app. L'integrazione tra Streamlit e i grafici generati con Matplotlib creava dei conflitti di rendering quando i dati si aggiornavano in tempo reale. Studiando la documentazione di Streamlit, ho risolto il problema implementando un sistema di caching dei dati tramite il decoratore `@st.cache_resource`, che ha stabilizzato la visualizzazione e migliorato nettamente le prestazioni dell'interfaccia.

PARTE SECONDA – PREDISPOSIZIONE DELL'ELABORATO

Obiettivi del progetto

(Descrivere gli obiettivi raggiunti dall'elaborato, indicando in che modo esso risponde a quanto richiesto dalla traccia):

Quando ho iniziato a lavorare ad AutoTriage NLP, lo scopo principale era ovviamente rispettare le richieste della traccia: creare un sistema per automatizzare la classificazione dei ticket di supporto. Tuttavia, ho cercato di realizzare un progetto che fosse il più possibile vicino a un caso d'uso reale, prestando attenzione non solo alla precisione dell'algoritmo, ma anche all'efficienza e all'interpretabilità del modello.

Nello specifico, il lavoro si è concentrato sui seguenti aspetti tecnici:

1. Gestione dei dati e Privacy

Per l'addestramento del modello, ho preferito evitare la generazione di ticket casuali o sintetici, che avrebbero reso il sistema poco verosimile. Ho quindi recuperato da Kaggle un dataset reale e già anonimizzato, garantendo l'assenza di dati sensibili e rispettando di base le normative sulla privacy. Poiché il dataset era in lingua inglese, ho sviluppato uno script per tradurlo massivamente in italiano. Questo mi ha permesso di addestrare il modello su testi autentici e complessi, molto più vicini al reale linguaggio utilizzato dagli utenti rispetto a frasi costruite a tavolino.

2. Architettura del modello unificata

Per quanto riguarda l'architettura di Machine Learning, anziché addestrare due classificatori separati per stimare la Categoria e la Priorità, ho implementato un singolo modello basato su una Regressione Logistica Multi-Output. Questa scelta mi ha permesso di ottenere entrambe le predizioni simultaneamente con un unico passaggio, alleggerendo il carico computazionale. Le performance sono state poi valutate in modo rigoroso utilizzando metriche standard come Accuracy e F1-Macro, calcolate a valle di una fase di preprocessing del testo che ha incluso tokenizzazione, rimozione delle stop-words e analisi degli n-grammi.

3. Interpretabilità dei risultati (LIME)

Un aspetto a cui ho dedicato particolare attenzione è stata la trasparenza delle decisioni prese dall'algoritmo. Non volevo che il sistema operasse come una “scatola nera”. Per questo motivo ho integrato la libreria LIME (Local Interpretable Model-agnostic Explanations), che analizza la singola predizione e mostra a schermo quali parole esatte (ad esempio “server”, “blocco” o “scadenza”) hanno spinto il modello a classificare un ticket in un certo modo. È una funzionalità essenziale per permettere all'operatore di verificare e fidarsi del suggerimento fornito dalla macchina.

4. Interfaccia utente e usabilità

Infine, ho curato l'aspetto applicativo del progetto. Utilizzando il framework Streamlit, ho costruito una web app che astrae la complessità del codice Python sottostante, rendendo il sistema accessibile anche a personale non tecnico. L'interfaccia permette sia di testare l'algoritmo in tempo reale (inserendo il testo di un singolo ticket), sia di caricare un file CSV per elaborare uno storico di richieste in modalità batch, simulando così uno strumento capace di ridurre concretamente i tempi di smistamento manuale in un Help Desk.

Contestualizzazione

(Descrivere il contesto teorico e quello applicativo dell'elaborato realizzato):

1. Contesto Applicativo e Ottimizzazione dei Processi

Il progetto si inserisce nell'ambito della Digital Transformation, con un focus specifico sull'ottimizzazione dei processi aziendali legati al supporto clienti. Analizzando il flusso di lavoro di un tipico servizio di Help Desk, emergono spesso criticità sistemiche nella fase iniziale di triage (smistamento): si creano colli di bottiglia, si verificano errori di assegnazione manuale e i sistemi tradizionali faticano a rilevare tempestivamente le urgenze. In uno scenario convenzionale, l'errata assegnazione di un singolo ticket comporta un re-instradamento manuale tra i reparti (Amministrazione, Tecnico, Commerciale), raddoppiando i tempi di latenza e aumentando i costi operativi.

Per ovviare a queste inefficienze, il prototipo sviluppato agisce come un middleware intelligente, interponendosi tra i canali di contatto (form web, email) e il sistema di ticketing interno. Il modello analizza la richiesta in entrata e la arricchisce con metadati semantici, suggerendo automaticamente il reparto di destinazione e il livello di priorità prima dell'intervento umano. Questo approccio mira a ottimizzare il ciclo di vita del ticket, riducendo il Mean Time To Repair (MTTR) e migliorando la qualità percepita dall'utente.

2. Contesto Applicativo e Ottimizzazione dei Processi

Il problema affrontato rientra nel dominio della Text Classification, una delle principali aree del Natural Language Processing (NLP). Storicamente, lo smistamento dei ticket è stato gestito da operatori umani o tramite sistemi rule-based fondati su costrutti logici rigidi (IF-THEN). Tuttavia, come evidenziato in letteratura (Gaddis, 2016), questi approcci risultano fragili: faticano a gestire l'ambiguità del linguaggio naturale, non riconoscono i sinonimi e sono vulnerabili ai refusi di battitura.

Per superare questi limiti strutturali, si è optato per un approccio basato sul Machine Learning supervisionato. Invece di programmare istruzioni esplicite su cosa cercare nel testo, si è permesso all'algoritmo di apprendere autonomamente le correlazioni statistiche tra i termini utilizzati e le categorie di destinazione, addestrandolo su un dataset reale preventivamente etichettato.

3. Metodologie e Scelte Implementative

Per trasformare i dati testuali in un formato elaborabile dalla macchina e procedere alla classificazione, sono state adottate le seguenti tecniche:

- **Rappresentazione Vettoriale (TF-IDF e N-grammi):** I modelli di Machine Learning richiedono input numerici. Per la vettorizzazione del testo è stata scelta la tecnica TF-IDF (Term Frequency - Inverse Document Frequency). Rispetto a un più semplice approccio Bag of Words (basato sul mero conteggio delle parole), il TF-IDF fornisce una rappresentazione pesata, valutando la rilevanza di un termine nel singolo documento rispetto all'intero corpus (Zinoviev, 2017). La formula utilizzata per calcolare il peso statistico è la seguente:

$$w_{i,j} = tf_{i,j} \times \log(N / df_i)$$

Il termine logaritmico $\log(N / df_i)$ svolge un ruolo fondamentale: penalizza matematicamente le parole molto frequenti e non informative (come gli articoli o termini generici come “problema”), assegnando invece un peso maggiore a parole specifiche e discriminanti (come “fattura”, “server” o “python”). Inoltre, per preservare il contesto locale della frase, è stata introdotta l'analisi dei bigrammi (coppie di parole consecutive), essenziale per far cogliere all'algoritmo la differenza semantica tra espressioni come “non funziona” e “funziona”.

- **Algoritmo di Classificazione (Logistic Regression):** Una volta vettorizzati i testi in uno spazio ad alta dimensionalità, per il task di classificazione è stata scelta la Regressione Logistica. Rispetto ad algoritmi basati puramente su margini geometrici come le Support Vector Machines (SVM), la Regressione Logistica modella esplicitamente le probabilità di appartenenza a una classe. In un contesto produttivo, questa caratteristica è preziosa poiché permette di fornire all'operatore un grado di confidenza sulla previsione (es. “Priorità Alta al 92%”). L'algoritmo garantisce inoltre bassa complessità temporale ed elevata velocità di inferenza. Per ottimizzare le risorse, il modello è stato incapsulato in un **MultiOutputClassifier** (Maggi, 2020), permettendo di calcolare simultaneamente Categoria e Priorità. Infine, l'integrazione di tecniche di Explainable AI (LIME) ha reso il modello ispezionabile, permettendo di evidenziare visivamente quali termini hanno influenzato la classificazione, garantendo la trasparenza necessaria per l'adozione aziendale.

Descrizione dei principali aspetti progettuali (Sviluppare l'elaborato richiesto dalla traccia prescelta):

Lo sviluppo di AutoTriage NLP si basa su un'architettura modulare scritta in Python, pensata per gestire l'intero ciclo di vita del dato, dall'acquisizione grezza fino all'inferenza. Per mantenere il codice ordinato e manutenibile, ho strutturato il progetto in tre macro-aree logiche: Data Engineering, Modello di Machine Learning e Frontend.

Di seguito descrivo l'implementazione tecnica dei vari moduli:

1. Data Engineering e Preparazione dei Dati

La prima fase ha riguardato la costruzione di un dataset di addestramento in italiano che fosse qualitativamente valido.

- **Traduzione (translate_data.py):** Dovendo tradurre oltre 20.000 ticket, un approccio sequenziale sarebbe stato troppo lento. Ho quindi scritto uno script che parallelizza le chiamate alle API di Google Translate tramite **ThreadPoolExecutor**. Per gestire eventuali interruzioni o blocchi temporanei delle API (rate limiting), ho inserito dei ritardi programmati tra le richieste e implementato un sistema di salvataggio intermedio, in modo da poter riprendere la traduzione dal punto di crash senza perdere i dati già elaborati.
- **Normalizzazione (prepare_data.py):** Questo script si occupa della pulizia dei dati. Verifica la presenza del dataset tradotto (se manca, avvia la procedura per generarlo) e si occupa di mappare le etichette originali di Kaggle sulle tre macro-categorie definite per il progetto (Amministrazione, Tecnico, Commerciale) utilizzando dei semplici dizionari di conversione. Per far capire meglio come ho riorganizzato i dati, ecco come ho strutturato nel codice la mappatura delle categorie di Kaggle verso le tre macro-classi aziendali del progetto:

```
# Mappatura per ridurre le categorie originali di Kaggle alle mie 3 macro-classi
CATEGORY_MAP: Dict[str, str] = {
    "Technical Support": "Tecnico",
    "IT Support": "Tecnico",
    "Service Outages and Maintenance": "Tecnico",
    "Product Support": "Tecnico",
    "Billing and Payments": "Amministrativo",
    "Returns and Exchanges": "Amministrativo",
    "Human Resources": "Amministrativo",
    "General Inquiry": "Commerciale",
    "Customer Service": "Commerciale",
    "Sales and Pre-Sales": "Commerciale"
}
```

Oltre alla semplice riassegnazione delle etichette, una parte consistente del lavoro di Data Engineering, che spesso viene sottovalutata a livello teorico ma che si rivela cruciale all'atto pratico, è stata la vera e propria “pulizia” (cleansing) delle stringhe di testo. Lavorando con ticket reali estratti da Kaggle, mi sono reso conto subito che il testo grezzo era pieno di “rumore” che avrebbe confuso l'algoritmo TF-IDF. Ad esempio, moltissimi ticket contenevano caratteri di formattazione nascosti, come i ritorni a capo (`\n` o `\r`), doppi spazi, o firme automatiche generate dai client di posta elettronica.

Per risolvere questo problema alla radice, prima ancora di passare i testi al traduttore, ho dovuto implementare delle espressioni regolari (Regex) tramite la libreria **re** di Python per normalizzare le stringhe. Ho forzato tutto il testo in minuscolo (lowercasing) per evitare che il modello considerasse “Server” e “server” come due concetti distinti, raddoppiando inutilmente la dimensione del vocabolario. Inoltre, analizzando i primi risultati di classificazione durante i test esplorativi, ho notato che parole come “salve”, “grazie”, “cordiali saluti” o “buongiorno” ottenevano un peso statistico immotivato. Essendo formule di cortesia onnipresenti in quasi tutti i ticket, a prescindere dal reparto di competenza, non portavano alcuna informazione utile per la classificazione, ma anzi, “diluivano” l’importanza dei termini veramente tecnici. Per ovviare a questa anomalia, invece di usare solo le liste di stop-words standard fornite dalle librerie NLP, ho dovuto costruire un dizionario di stop-words custom in italiano, integrandovi manualmente tutti questi convenevoli aziendali per costringere l’algoritmo a concentrarsi solo sul nocciolo del problema.

2. Architettura del Modello ML (**unified_model.py**)

Per la componente predittiva, ho creato una classe **UnifiedModel** che racchiude in un’unica pipeline sia la fase di vettorizzazione che quella di classificazione.

- **Vettorizzazione TF-IDF**: La trasformazione del testo in vettori numerici tiene conto non solo delle singole parole (unigrammi) ma anche dei bigrammi (coppie di parole). Questo è utile per non perdere il contesto locale della frase, permettendo all'algoritmo di distinguere, ad esempio, tra “funziona” e “non funziona”.
- **Classificazione Multi-Output**: Piuttosto che gestire due modelli separati, ho usato la classe **MultiOutputClassifier** di Scikit-Learn. Questo mi ha permesso di addestrare due regressori logistici in parallelo (uno per la Categoria, uno per la Priorità) all’interno della stessa esecuzione, ottimizzando i tempi di inferenza.

A livello di codice, ho tradotto questa architettura creando una singola **Pipeline** all’interno della classe **UnifiedModel**. Come si vede dallo snippet, questo blocco gestisce in un solo colpo sia la vettorizzazione che la classificazione:

```
# Costruisco la pipeline
self.pipeline: Pipeline = Pipeline([

    # 1. Vettorizzazione
    # Uso max_features=5000 per non appesantire il modello e n_gram=(1,2)
    # per catturare concetti come "non funziona"
    ('tfidf', TfidfVectorizer(
        stop_words=self.stopwords,
        max_features=5000,
        ngram_range=(1, 2)
    )),

    # 2. Classificazione Multipla
    # Il class_weight='balanced' mi aiuta a gestire le classi minoritarie
    ('clf', MultiOutputClassifier(
        LogisticRegression(
            solver='lbfgs',
            max_iter=1000,
            random_state=42,
            class_weight='balanced'
        )
    ))

])
```

3. Training e Serializzazione (train_unified_model.py)

Questo script gestisce la fase di addestramento vero e proprio. Suddivide il dataset pre-processato con un classico split 80/20 (Training/Test Set) per evitare problemi di overfitting. Una volta addestrato il modello e calcolate le metriche di validazione, lo script utilizza la libreria **joblib** per serializzare e salvare il modello in un file binario **.pkl**. In questo modo, il modello diventa persistente e può essere caricato direttamente dalla web app senza doverlo riaddestrare a ogni esecuzione.

4. Interfaccia Utente e Logica Ibrida (app.py)

Il frontend, che ho sviluppato interamente con Streamlit, non l'ho concepito come una semplice vetrina estetica, ma vi ho integrato alcune logiche di business avanzate.

- **Priorità Ibrida:** Per evitare che il modello statistico sottostimi ticket critici, ho affiancato all'output del Machine Learning un controllo deterministico basato su keyword. Se nel testo del ticket vengono rilevate parole sensibili (es. "hacker", "fermo", "scadenza"), l'applicativo forza automaticamente la priorità su "Alta", garantendo un livello di sicurezza aggiuntivo. Per dare un'idea di come funziona questo blocco di sicurezza, ecco un estratto della funzione **calculate_priority** (dal file **app.py**). Qui ho definito esplicitamente la lista delle parole chiave che scavalcano il modello e forzano l'urgenza:

```
# Dizionari per il controllo regole
critical_keywords = [
    "virus", "hacker", "attacco", "violazione", "perso dati", "cancellato",
    "fermo", "blocco", "bloccato", "scadenza", "entro domani", "urgent",
    "subito", "velocemente", "critico", "panico", "terribile"
]

dampening_keywords = [
    "con calma", "non è urgente", "non urgente", "nessuna fretta",
    "quando potete", "appena possibile", "senza urgenza",
    "normale", "nessun problema", "funziona", "tutto ok", "informazione"
]

text_lower = text.lower()
is_critical = any(kw in text_lower for kw in critical_keywords)
is_dampener = any(kw in text_lower for kw in dampening_keywords)

# Controllo 1: Presenza di termini molto critici senza formule di cortesia/calma
if is_critical and not is_dampener:
    trigger_word = next((kw for kw in critical_keywords if kw in text_lower), "keyword")
    debug_probs['Alta'] = 0.99
    debug_probs['Media'] = 0.01
    debug_probs['Bassa'] = 0.00
    return 'Alta', 0.99, debug_probs, [trigger_word]
```

- **Interpretabilità (LIME):** Per rendere le decisioni del modello verificabili, la dashboard integra un modulo basato sulla libreria LIME. Quando viene analizzato un ticket, l'interfaccia genera un grafico che mostra all'operatore quali parole specifiche hanno influenzato maggiormente l'assegnazione della categoria, rendendo il comportamento dell'algoritmo trasparente.

5. Modulo di Sperimentazione (Script Ausiliari)

Per convalidare la scelta di utilizzare un dataset reale (più costoso in termini di tempo di preparazione) rispetto a uno generato artificialmente, ho implementato due script di test:

- **Generazione Dati (generate_synthetic_data.py):** Uno script procedurale che genera una baseline di 500 ticket fittizi basandosi su template predefiniti.

- **Benchmark (compare_models.py):** Questo script addestra due versioni del modello: una sui dati sintetici e una sui dati reali. Successivamente, esegue una valutazione incrociata (testa il modello sintetico sui dati reali e viceversa). I risultati, salvati come matrici di confusione nella cartella **assets/img/png**, dimostrano quantitativamente che il modello addestrato sui dati reali ha una capacità di generalizzazione nettamente superiore. Questo test giustifica a livello empirico il tempo dedicato alla fase iniziale di Data Engineering.

Campi di applicazione

(Descrivere gli ambiti di applicazione dell'elaborato progettuale e i vantaggi derivanti della sua applicazione):

Sebbene il progetto AutoTriage NLP sia stato sviluppato pensando alle esigenze di un Help Desk IT, l'architettura software è stata concepita per essere indipendente dal dominio applicativo. Sfruttando il binomio TF-IDF e Regressione Logistica, il sistema può essere adattato a contesti operativi diversi semplicemente fornendo un nuovo dataset di addestramento, senza la necessità di modificare la logica del codice sottostante.

L'implementazione di un sistema di triage basato sul Machine Learning permette di passare da una gestione reattiva a una proattiva, portando a vantaggi organizzativi misurabili:

- **Riduzione dei tempi di risoluzione (TTR):** L'assegnazione istantanea del ticket al reparto di competenza elimina i colli di bottiglia legati allo smistamento manuale.
- **Continuità del servizio (H24):** Il classificatore automatico non è soggetto a limiti di orario, garantendo la gestione immediata dei picchi di richieste.
- **Rispetto degli SLA (Service Level Agreements):** L'identificazione immediata delle criticità (supportata dalle logiche ibride basate su keyword) previene le violazioni delle tempistiche contrattuali.
- **Ottimizzazione delle risorse umane:** Demandando alla macchina le attività di classificazione ripetitive, il personale può essere ricollocato su compiti a maggior valore aggiunto che richiedono empatia o capacità decisionali complesse.

Per dimostrare la versatilità dell'approccio supervisionato, ho analizzato cinque possibili scenari in cui la stessa architettura potrebbe essere impiegata con successo:

1. **E-commerce (Automazione del supporto):** Addestrando il classificatore su uno storico di chat ed email, il sistema può distinguere richieste informative di base (es. "tracking", "spedizione") da criticità operative (es. "reso", "rimborso", "merce danneggiata"). Questo permette di attivare risposte automatiche per le spedizioni e di instradare subito i clienti insoddisfatti verso gli operatori.
2. **Settore Bancario e Assicurativo (Gestione Documentale):** Analizzando i testi delle PEC e delle comunicazioni formali, l'algoritmo può separare le pratiche amministrative standard (come l'aggiornamento anagrafico) da eventi che richiedono tempistiche legali rigorose (es. "sinistro", "frode", "blocco carta"), riducendo il rischio di sanzioni per l'istituto.

- 3. Sanità e Telemedicina (Triage Digitale preventivo):** Senza ovviamente sostituire il giudizio clinico, il modello può agire da primo filtro semantico sui portali di prenotazione. Riconoscendo termini burocratici (“ricetta”, “certificato”) li separa dalle descrizioni sintomatologiche (“dolore”, “febbre”), assegnando un codice di priorità per accelerare la presa in carico da parte del personale medico.
- 4. Marketing B2B (Qualificazione dei Lead):** Analizzando il campo testuale dei form di contatto, l’algoritmo può imparare a distinguere messaggi con una reale intenzione d’acquisto (contenenti termini come “preventivo”, “budget”, “incontro”) da richieste generiche o spam, permettendo alla forza vendita di concentrarsi solo sui contatti più promettenti.
- 5. Analisi della Reputazione (Sentiment Analysis):** Cambiando il focus della classificazione dall'argomento alla “polarità” emotiva del testo, il modello può analizzare le recensioni online. Assegnando pesi a termini critici (“truffa”, “pessimo”, “vergogna”), il sistema può generare alert in tempo reale, permettendo all’azienda di intervenire tempestivamente per gestire eventuali crisi d’immagine.

In sintesi, il progetto dimostra come l’elaborazione del linguaggio naturale possa tradursi in uno strumento applicativo concreto. L’obiettivo raggiunto non si limita alla stesura di un algoritmo funzionante, ma consiste nella realizzazione di un’architettura software scalabile e interpretabile, capace di ottimizzare la gestione delle informazioni non strutturate e di supportare attivamente i processi decisionali all'interno di un’organizzazione aziendale.

Valutazione dei risultati
(Descrivere le potenzialità e i limiti ai quali i risultati dell’elaborato sono potenzialmente esposti):

L’ultima fase del progetto è stata dedicata alla validazione dell’ipotesi iniziale, ovvero verificare se un algoritmo tradizionale e leggero come la Regressione Logistica potesse gestire efficacemente la classificazione dei ticket, a patto di operare su dati pre-processati correttamente.

I test sono stati condotti sul Test Set, corrispondente al 20% del corpus totale (circa 4.000 ticket inediti per il modello). Valutare le performance su un dataset reale e tradotto in automatico fornisce una metrica di base (baseline) molto più vicina a un reale scenario produttivo rispetto ai risultati, spesso irrealisticamente alti, che si ottengono in ambito didattico su dati fittizi.

Per dimostrare l’importanza di utilizzare dati reali, è stato condotto un test comparativo (A/B Test) tra due versioni dell’algoritmo: il Modello A (addestrato su un campione di dati sintetici generati tramite template) e il Modello B (addestrato sul dataset Kaggle).

Tabella Riassuntiva delle Performance

| Task di Classificazione | Accuracy (Globale) | Precision | Recall | F1-Score |
|-------------------------|--------------------|-----------|--------|----------|
| Categoria del Ticket | 59% | 58% | 59% | 57% |
| Priorità del Ticket | 46% | 45% | 46% | 44% |

Dettaglio per Classe (F1-Score)

Per avere un quadro più preciso, la tabella seguente mostra la reale capacità del modello di distinguere tra i vari reparti aziendali:

| Categoria | F1-Score | Note |
|----------------|----------|--|
| Tecnico | 0.69 | La classe più performante grazie a una terminologia specifica e discriminante. |
| Amministrativo | 0.51 | Risente di sovrapposizioni semantiche con l'ambito commerciale. |
| Commerciale | 0.41 | Classe con maggiore variabilità lessicale nel dataset di training. |

I dati che presento di seguito derivano dall'ultima validazione effettuata sul Test Set, che copre il 20% dei ticket a disposizione. Per garantire che l'esperimento fosse riproducibile e che i risultati non fossero casuali, ho fissato il seed (impostando **random_state=42**), in modo che lo split dei dati e il training rimanessero costanti.

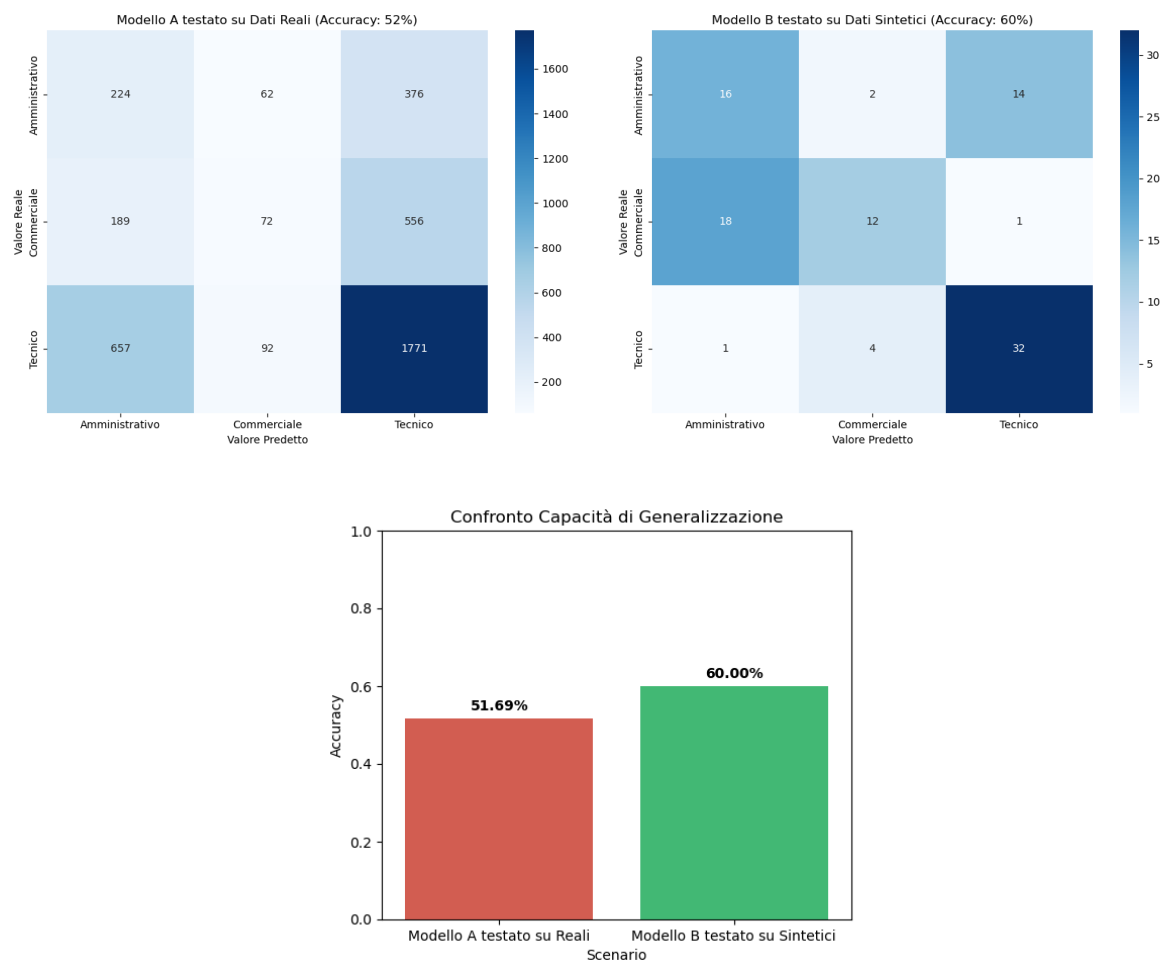
Guardando questi numeri a mente fredda, e analizzando nel dettaglio i falsi positivi e i falsi negativi emersi dalle matrici di confusione generate dallo script **compare_models.py**, mi sono posto alcune domande sul perché l'algoritmo facesse più fatica su specifiche categorie. Mentre la classe "Tecnico" viene riconosciuta con grande facilità (probabilmente perché termini come "router", "database" o "schermata blu" non lasciano spazio a interpretazioni), ho notato una forte "zona grigia" tra i ticket classificati come "Amministrativo" e "Commerciale".

Andando a leggere manualmente alcuni dei ticket che il modello ha predetto in modo errato, ho capito che l'errore non era dovuto a un bug del codice, ma alla natura intrinsecamente ambigua dei processi aziendali. Molto spesso, infatti, un agente di vendita (Commerciale) apre un ticket per chiedere spiegazioni su una fattura errata inviata a un suo cliente (Amministrazione). In questi casi, il testo del ticket contiene un mix perfetto di vocaboli appartenenti a entrambe le sfere semantiche (es. "preventivo", "cliente Rossi", "storno", "bonifico").

In queste situazioni limite, l'algoritmo TF-IDF va inevitabilmente in difficoltà perché il peso matematico dei termini si bilancia. Questo fenomeno, noto nel Machine Learning come class overlapping (sovrapposizione delle classi), dimostra ancora una volta l'importanza di avere un operatore umano a supervisione del sistema. Il modello non deve essere visto come un decisore assoluto e infallibile, ma piuttosto come un "copilota" che smaltisce in autonomia l'80% del lavoro banale e ripetitivo, lasciando le richieste ambigue (quella minoranza di ticket in cui le probabilità calcolate dal modello per due categorie sono molto vicine, es. 48% contro 52%) all'analisi logica e all'intuito dell'operatore umano.

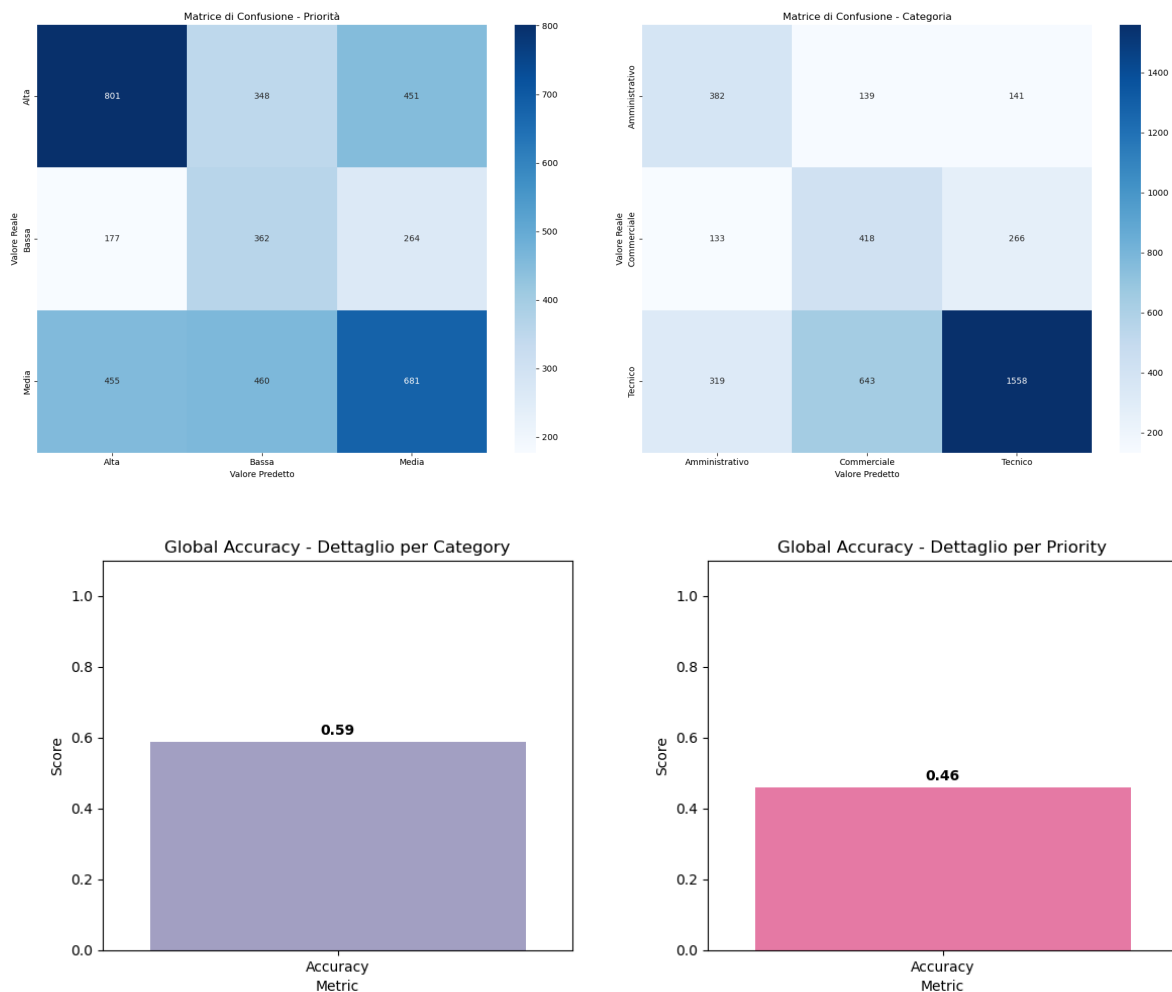
Vorrei però fare una precisazione sul dataset: avendo tradotto i testi originali tramite le API di Google Translate, far ripartire l'intera pipeline da zero oggi potrebbe produrre metriche leggermente diverse. Questo perché il motore di Google si aggiorna di continuo e potrebbe restituire sinonimi o formulazioni differenti rispetto a quando ho scaricato i dati, alterando di fatto la distribuzione del vocabolario. Per questo motivo, i grafici e i numeri che documento qui fanno riferimento allo snapshot esatto del modello addestrato sul file **tickets_it_augmented.csv**.

Guardando ai risultati sul Test Set, l'accuratezza globale si attesta al 59% per la scelta della Categoria e al 46% per la Priorità. Di seguito ho inserito le matrici di confusione per dare un'idea più chiara di come si distribuiscono le predizioni e dove il modello tende a confondersi.

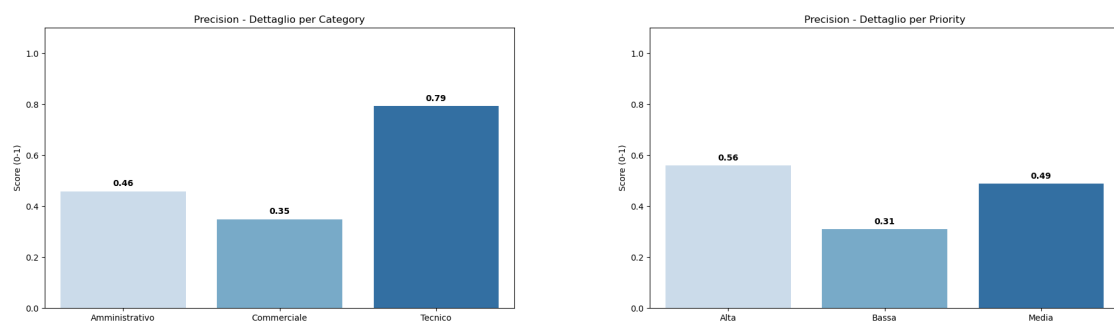


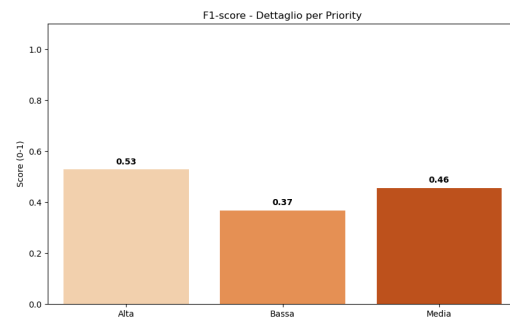
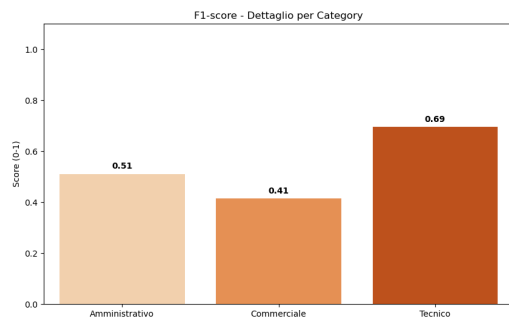
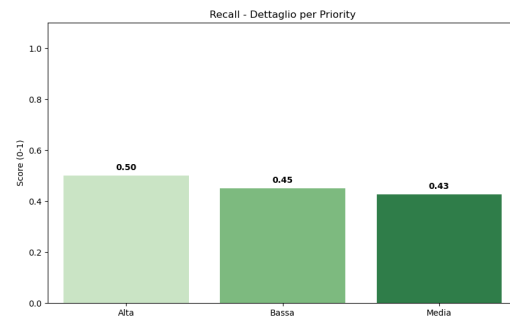
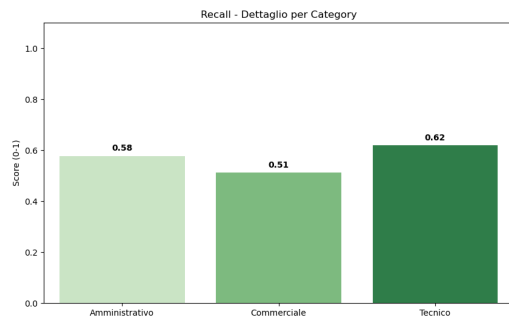
I risultati evidenziano un netto divario. Il Modello A ha registrato un crollo delle performance (Accuratezza ~52%) quando testato sui ticket reali. Questo calo è dovuto all'overfitting: il modello sintetico ha imparato a riconoscere pattern fissi e keyword esatte, ma non è in grado di generalizzare. Al contrario, il Modello B si è dimostrato più robusto, raggiungendo un'accuratezza del 60% anche quando testato sui dati sintetici mai visti in fase di training. Addestrare l'algoritmo sul "rumore" tipico del linguaggio umano (errori, sinonimi, ambiguità) lo ha costretto a generalizzare in modo più efficace.

L'analisi delle metriche diagnostiche restituisce un quadro incoraggiante. Ho inserito questi primi grafici per avere un colpo d'occhio immediato sulle performance generali, prima di analizzare le metriche classe per classe.



- **Metriche per classe:** Osservando i valori di Precision e F1-Score, il modello ottiene i risultati migliori sulla categoria “Tecnico” (F1-Score: 0.69). La matrice di confusione mostra che la sovrapposizione tra classi semanticamente vicine (come “Amministrativo” e “Commerciale”) viene mitigata in modo soddisfacente dalla vettorizzazione TF-IDF, che penalizza i termini troppo comuni.





- **Logica ibrida per le priorità:** Un punto di forza pratico dell'applicativo è l'utilizzo di una logica mista. Alla classificazione probabilistica del modello si affianca una regola deterministica basata su parole chiave: se nel testo compaiono termini critici (es. “virus”, “hacker”, “fermo”), il sistema forza l'assegnazione di una Priorità Alta. Questo strato di controllo serve a evitare che eventuali falsi negativi del modello statistico facciano ignorare ticket urgenti.
- **Interpretabilità:** L'uso della libreria LIME permette all'operatore di visualizzare quali termini abbiano influenzato la predizione. Fornire una spiegazione visiva del comportamento dell'algoritmo è fondamentale per favorirne l'adozione e aumentare la fiducia degli utenti finali verso il sistema.

A livello puramente implementativo, unire il backend di Machine Learning con il frontend di Streamlit non è stato banale e ha richiesto un po' di ingegneria del software. Streamlit, per come è disegnato, ricarica l'intero script Python ogni volta che l'utente interagisce con un widget (ad esempio, quando preme il bottone “Analizza”). Inizialmente, questo comportamento faceva sì che il modello **.pkl**, che pesa svariati megabyte, venisse ricaricato in memoria da zero a ogni singolo click, rendendo l'interfaccia estremamente lenta e scattosa.

Per risolvere questo collo di bottiglia prestazionale, mi sono dovuto documentare sui meccanismi di caching avanzati. Ho implementato il decoratore **@st.cache_resource** sopra la funzione di caricamento del modello: in questo modo, l'oggetto **UnifiedModel** viene istanziato nella memoria RAM del server una sola volta all'avvio dell'applicazione. Tutte le successive inferenze pescano direttamente il modello già in memoria, abbattendo i tempi di risposta dell'interfaccia da diversi secondi a pochi millisecondi. È stato un passaggio tecnico fondamentale per trasformare un semplice script accademico in un prototipo reattivo e realmente utilizzabile da un ipotetico operatore di Help Desk senza frustrazioni.

Nonostante i risultati raggiunti, l'architettura implementata presenta alcune limitazioni intrinseche dovute alle scelte progettuali:

- **Limiti dell'approccio Bag-of-Words / TF-IDF:** Questa tecnica tratta le parole in modo indipendente, ignorando la sintassi generale e la sequenzialità. Di conseguenza, il modello fatica a interpretare frasi complesse, doppie negazioni o sarcasmo. L'adozione di architetture a reti neurali basate su Transformer (come BERT) risolverebbe il problema, ma richiederebbe risorse computazionali nettamente superiori.
- **Qualità della traduzione automatica:** Poiché il Training Set è stato tradotto interamente tramite API, è probabile che artefatti linguistici o traduzioni letterali (es. il termine inglese term tradotto come "termine" anziché "condizione") abbiano introdotto un certo grado di "rumore" statistico nel modello.
- **Degrado delle performance nel tempo:** Il modello attuale è statico. Se il vocabolario aziendale dovesse cambiare (ad esempio con il lancio di nuovi prodotti non presenti nel vocabolario di addestramento), le performance tenderebbero a calare progressivamente, rendendo necessario pianificare cicli di ri-addestramento periodici.

Per evolvere il prototipo verso una soluzione applicabile in azienda, i prossimi step implementativi potrebbero includere l'introduzione della lemmatizzazione (tramite librerie come spaCy) per normalizzare i termini alla loro radice linguistica, e lo sviluppo di un meccanismo di feedback loop, in cui l'operatore può correggere le predizioni errate per permettere al sistema di apprendere continuamente.

Infine, per contestualizzare il progetto dal punto di vista aziendale, è stata effettuata una stima del Ritorno sull'Investimento (ROI) operativo. Considerando un tempo medio di 2 minuti spesi da un operatore per leggere e smistare un ticket manualmente, un volume ipotetico di 10.000 ticket mensili e un costo aziendale di 25 €/h, l'automazione del triage genererebbe il seguente risparmio:

$$\text{Risparmio} = (2 / 60 \text{ ore}) \times 10.000 \text{ ticket} \times 25\text{€/h} \approx 8.333\text{€} / \text{mese}$$

Questo dato, unito all'efficacia dimostrata dai test di generalizzazione, conferma che l'implementazione di tecniche di Natural Language Processing per il triage non è solo un esercizio tecnico, ma rappresenta un'opportunità concreta per ottimizzare i costi operativi dei servizi di Help Desk.